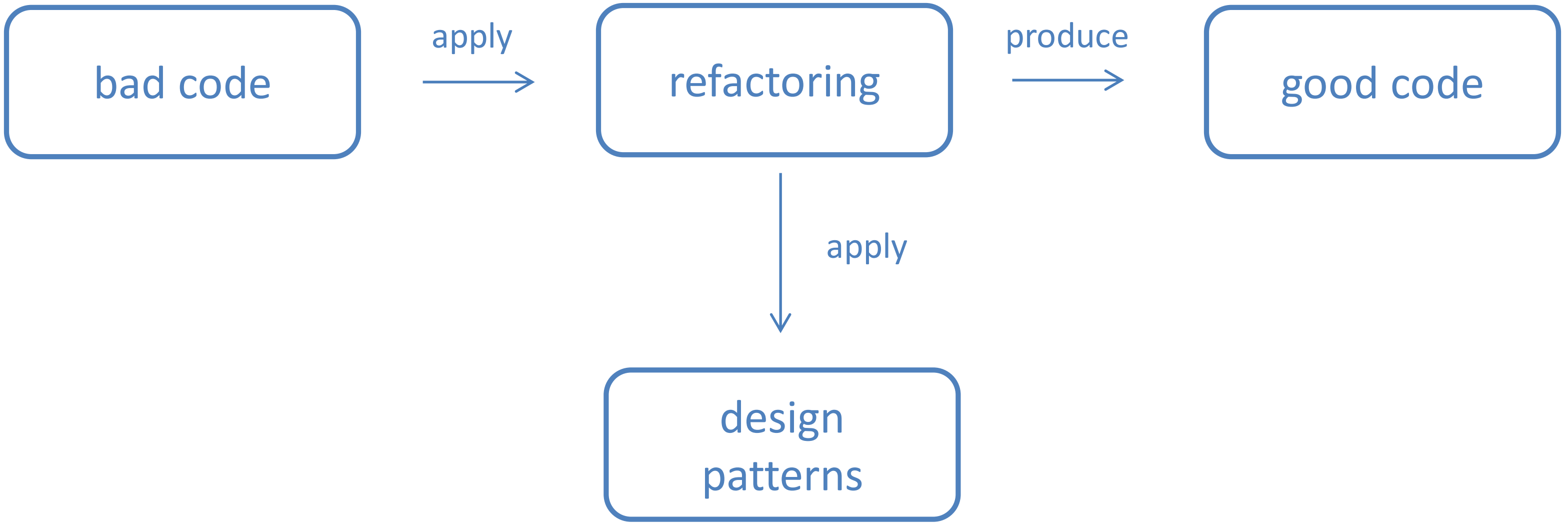
Refactoring, good and bad coding practices

Agenda

- Bad coding practices
- Catalogue of code smells
- Code smell discovery and refactoring

Bad coding practices

The big picture



bad code

apply

refactoring

produce

good code

apply

design patterns

Bad coding

- Bad code can be described as any code that causes different kinds of problems in the system such as:
  - lack or readability/maintainability/extensibility
  - general bugs
  - security and performance issues

Bad coding

- Recognizing certain anti-patterns and understanding good practices allows developers to avoid bad coding

- However in large (especially legacy) projects in might be quite a challenge to discover existing code smells and apply refactoring techniques

- For that reason automated discovery is essential not only in finding but also in preventing code smells

Reasons for code smells

- Factors leading to code smells:
  - haste
  - apathy
  - narrow-mindness
  - sloth
  - avarice (excessive details)
  - ignorance (intellectual sloth)
  - pride

Discovery of code smells

- Automated discovery can be achieved via static analysis tools

- These tools can:

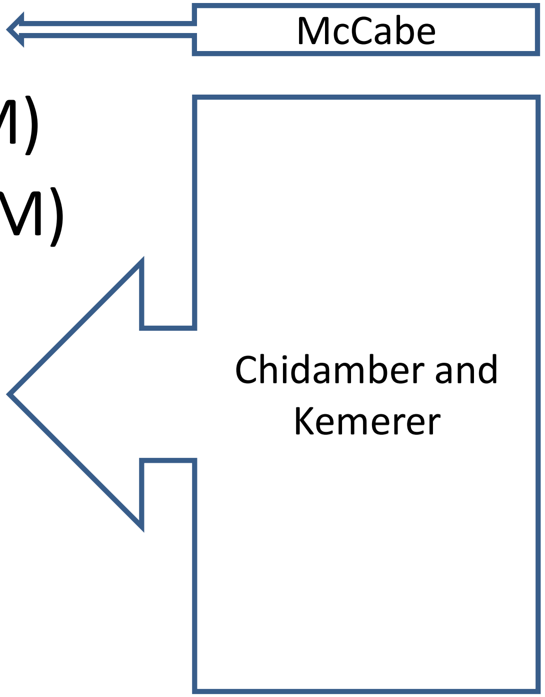
  - calculate source code metrics based on which to determine ‘smelly’ pieces of code

  - apply code patterns to determine code smells

  - enforce naming, formatting and structural rules

Source code metrics

- Lines of code (LOC)
- Cyclomatic complexity (CC)
- Lack of cohesion of methods (LCOM)
- Weighted methods per class (WCOM)
- Coupling between objects (CBO)
- Response for a class (RFC)
- Number of children (NOC)
- Depth of inheritance tree (DIP)
- Number of parameters (NP)



McCabe

Chidamber and Kemerer

Source code metrics

- Examples indicating the need for refactoring (might be different per project):
  - LOC > 80
  - CC > 10
  - NP > 4
  - DIT > 7

Source code metrics

- Nice list of tools that can be used to derive metrics from Java source code: https://www.monperrus.net/martin/java-metrics
- Many of the static analysis tools provide calculation of source code metrics in addition (such as SonarQube)

Static analysis tools

- A number of static analysis tools facilitate discovery of code smells:
  - SonarQube
  - Checkstyle
  - PMD
  - FindBugs (and its de-facto successor SpotBugs)
  - Facebook Infer
  - DesigniteJava
  - Google Error Prone
  - Qulice (combines Checkstyle, PMD, FindBugs and a few Maven plug-ins)

Static analysis tools

- Some tools provide specifically vulnerability scanning capabilities:
  - Veracode
  - OWASP DependencyCheck
  - Snyk
  - Eclipse CogniCrypt

Enforcement of conventions

- In order to facilitate development a project should define a set of conventions related to:
  - naming things (packages, classes, methods, fields, variables etc.)
  - formatting of source code
  - structural conventions
  - general code conventions

Enforcement of conventions

- Naming is a subject to some general rules such as:
  - short names must be avoided

    ~~Public class A~~

  - long names must be avoided

    ~~int paymentAccountForEndUsersWithDetailsAndSum~~

  - class names must start with a capital letter

    ```
    public class EntityManager
    ```

  - package names of companies must follow reverse domain name notation

    ```
    package com.company.model
    ```

Enforcement of conventions

- However certain project might require more specific naming conventions

- For example:

  - all classes related to persistence must end with **Entity**

    ```
    public class UserEntity
    ```

  - all interfaces must start with **I**

    ```
    public interface IListener
    ```

  - all local variables names must have at least two characters

    ```
    double orderPrice
    ```

  - all unit test classes must end with **Test**

    ```
    public class OrderProcessingServiceTest
    ```

Enforcement of conventions

- General or more specific naming requirements may be verified during code review

- There are tools that can automate the validation of some the naming rules such as:
  - Checkstyle
  - ArchUnit (in the form of a unit test)

```
classes().that().implement(IListener.class)
        .should().haveSimpleNameEndingWith(“Listener")
```

Enforcement of conventions

- Source code formatting is typically facilitated by the use of formatters in the IDE (either built-in, provided or custom-made)

- Many large organizations devise their own source code formatting rules

- There are publicly available ones from large vendors such as Oracle and Google

- However if strict formatting is to be enforced the Maven Checkstyle plug-in can be used in combination with the CI/CD system in place

Enforcement of conventions

- However if strict formatting is to be enforced the Maven Checkstyle plug-in can be used in combination with the CI/CD system in place:

```
<plugin>
          <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-checkstyle-plugin</artifactId>
           <version>2.9.1</version>
           <executions>
                    <execution>
                              <id>checkstyle</id>
                              <phase>validate</phase> <goals>
                              <goal>check</goal> </goals>
                              <configuration>
                                         <failOnViolation>true</failOnViolation>
                              </configuration>
                    </execution>
          </executions>
</plugin>
```

Enforcement of conventions

- Structural and code conventions can be enforced:
  - at the compiler level
  - by some of the static analysis tools we already listed
  - via unit tests (using ArchUnit)

Refactoring

- Benefits of refactoring:
  - improves the design of the system
  - makes software easier to understand (hence reduces maintenance costs)
  - makes software easier to adapt to changes and implement new features (hence improves extensibility)
  - helps you find and track down bugs more easily

Refactoring

- Good times to refactor:
  - when adding a new method
  - when doing a code review
  - when fixing a bug

Refactoring

- Unfortunately in many projects the push for features is greater than the need to produce quality software ...

- The need of refactoring is hence not understood properly by management and often disapproved

- With some good KPIs and justifications it is possible to show in a clear manner what benefits would a refactoring bring in the longer run ...

Refactoring

- In practice there are many limitations that prevent proper refactoring:
  - the need to keep backward compatibility (example: interfaces in the JDK and default interfaces)
  - integration with external systems (and the format of data that needs to be preserved)
  - control flow complexity
  - negative effects on performance and security

Refactoring

- If possible cover existing functionality with unit tests to avoid regressions as much as possible during refactoring

- Two approaches towards full refactoring of a project:

  – top down: project -> package -> class -> method

  – bottom up: method -> class -> package -> project

- Refactoring techniques will be demonstrated during the workshop using Eclipse and IntelliJ IDEA …

Catalogue of code smells

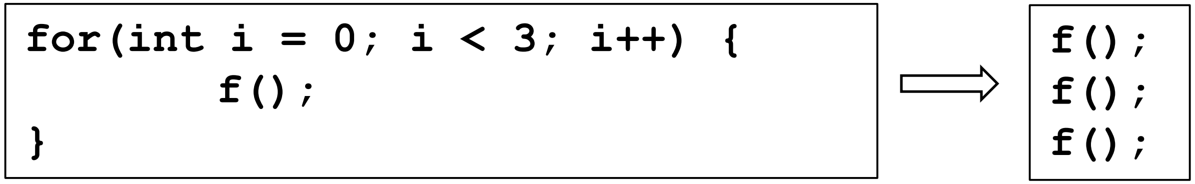
Categories of code smells

- We will organize the catalogue of code smells according to the level at which they apply:
  - application
  - class
  - method

Code smells and compilers

- Some of the code smells are handled at runtime via compiler optimizations

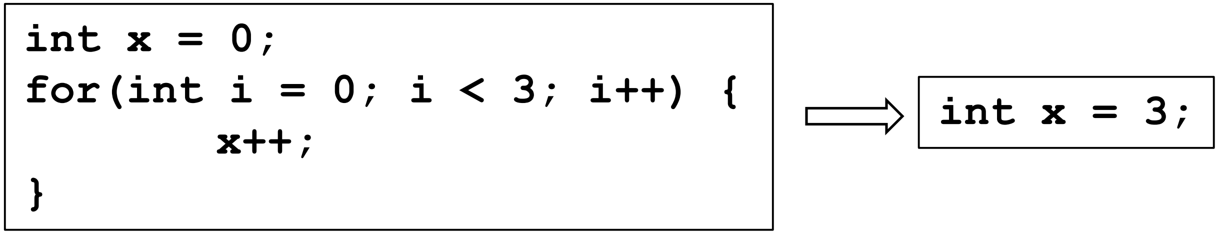
- Examples:
  - loop unrolling



```
for(int i = 0; i < 3; i++) {
        f();
}
```

```
f();
f();
f();
```

  - loop optimization



```
int x = 0;
for(int i = 0; i < 3; i++) {
        x++;
}
```

```
int x = 3;
```

Code smells and compilers

- More examples:
  - method inlining

```
public void printDetails() {
        dumpDetails();
}
public void dumpDetails() {
        Log.log(“details”);
}
```

⟹

```
public void printDetails() {
        Log.log(“details”);
}
```

  - boolean inversion

```
if(!(sum > 10)) {
        isLargeSum = true;
} else {
        isLargeSum = false;
}
```

⟹

```
if(sum > 10) {
        isLargeSum = false;
} else {
        isLargeSum = true;
}
```

Code smells and frameworks

- There are code smells specific for the particular framework in use such as:
  - JavaEE (now JakartaEE)
  - Spring Framework
  - OSGi

- An entirely separate course can be dedicated on the above …

Application level code smells at a glance

- unused/dead code
- duplicate code
- large and unstructured project
- spaghetti/lasagna/raviolli code
- shotgun surgery
- input kludge
- hardcoding
- softcoding
- excessive calls to third-party systems
- usage of vendor code
- vendor lock-in
- defensive programming
- lack of proper comments
- meaningless/misleading comments
- boat anchor
- golden hammer
- dead end

Unused/dead code

- Certain types of unused (dead) code such as unused local variables or private fields/methods can be detected by the IDE and static analysis tools

- Other forms of dead code such as unused classes or public methods can be identified on the basis of manual code review

Unused/dead code: resolution

- IDEs such as IntelliJ IDEA provide a ‘code cleanup’ feature that allows for automatic removal of identified dead code

- Otherwise manual removal of entire files or blocks of code is to be done

- In any case removal of dead code improves maintainability of the system

Duplicate code

- Duplicate code can be either intentional or unintentional

- Developers tend to intentionally copy-paste existing code and modify slightly instead of introducing a common abstraction

- As the system evolves some leftover code remains unintentionally undeleted

Duplicate code

- Static analysis tools and IDEs assist in discovery of duplicate code

- If we refer to project dependencies we can also use:
  - **duplicate-finder-maven-plugin** to find duplicate classes on the classpath
  - **maven-enforcer-plugin** to check for duplicate libraries on the classpath
  - use the **dependency:tree** goal of the **maven-dependency-plugin** to manually inspect the dependency tree of a project and identify duplicate libraries

Duplicate code

```
<plugin>
  <groupId>com.ning.maven.plugins</groupId>
  <artifactId>duplicate-finder-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>verify</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Duplicate code

```
<plugin>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.4.1</version>
  <executions>
    <execution>
      <id>enforce-no-duplicate-dependencies</id>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <banDuplicatePomDependencyVersions/>
        </rules>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Duplicate code: resolution

- Create a common abstraction or utility that eliminates duplicate code

- In case of duplicate libraries try to find a single version to use in all cases

Large and unstructured project

- A project contains a large number of classes many of which are big and complex

- There are many big and complex methods in the project

- Project provides logic for different business domains

Large and unstructured project: resolution

- The project may need to be split either into smaller subprojects

- Or the classes and the methods in the project needs to be split into smaller classes and methods

Lasagna/spaghetti/ravioli code

Lasagna/spaghetti/ravioli code: resolution

- In any of the cases typically general refactoring need to be applied in the project:
  - merge some of the excessive layers in the system in case of lasagna code
  - refactor or eliminate unstructured logic in the project in case of spaghetti code
  - merge some of loosely coupled components to larger ones based on logical grouping in case of ravioli code

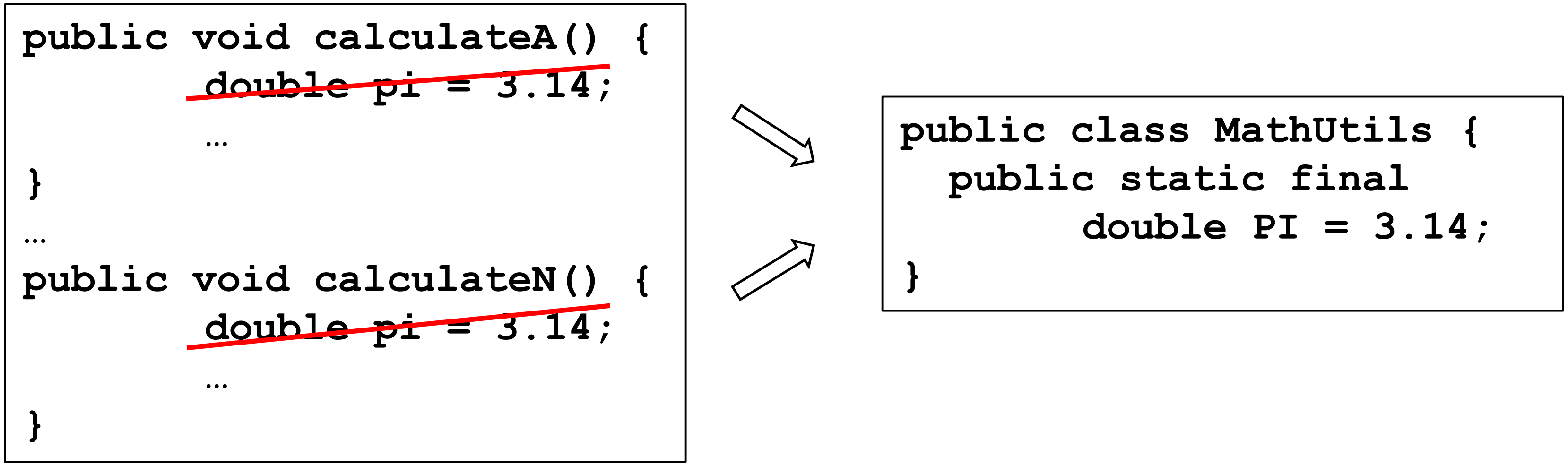
Shotgun surgery

- Shotgun surgery occurs when a single change needs to be applied to multiple classes at the same time

```
public void calculateA() {
        double pi = 3.14;
        …
}
…
public void calculateN() {
        double pi = 3.14;
        …
}
```

Shotgun surgery: resolution

- Shortgun surgery can be eliminated by extracting a common class, method or field to use based on the particular case



```
public void calculateA() {
        ~~double pi = 3.14;~~
        …
}
…
public void calculateN() {
        ~~double pi = 3.14;~~
        …
}
```

```
public class MathUtils {
  public static final
        double PI = 3.14;
}
```

Input kludge

- Input kludge is the lack of validation of user input
- It may cause unexpected exceptions, crash the system or open security vulnerabilities

```
public static void main (String[] args) {

   String query = args[1];
   …
   // is it a valid SQL query ?
   statement.executeQuery(query);
}
```

Input kludge: resolution

- Provide proper input validation along with a proper mechanism to convey validation errors to the user

```
public static void main (String[] args) {

   String query = args[1];
   validateQuery(query);
   …
   statement.executeQuery(query);
}
```

Hardcoding

- Hardcoding happens when we embed data such as integer constants or text directly inside the source code

- Every time a hardcoded value needs to be changed requires recompilation

Hardcoding: resolution

- Store hardcoded values externally and reference them from the application

- These external sources might be a properties file, RDBMS and so on

- Requires additional logic for reading of values in the application but improves maintanability

Softcoding

- Softcoding is the opposite of hardcoding
- It is typically the act of storing externally more values than needed even ones that won’t/cannot be changed or do not target the audience of users
- Makes the system difficult to configure properly

```
system.startupfunction=“() -> {…}”
```

Softcoding: resolution

- Eliminate some of the complex configuration and values that are not going to change

Excessive calls to third party systems

- These could be executing queries against the RDBMS, web service calls, Elasticsearch calls etc.
- Having too many calls to external systems makes the system difficult to maintain and also increases latency

Excessive calls to third party systems: resolution

- Remove some of the calls to third party systems if possible with application logic

- Batch calls to third party systems if that is provided as a capability (for example: ElasticSearch provides batching of queries)

- Merge multiple calls if possible (for example if you execute multiple SELECT queries on related data against the RDBMS you can use JOIN instead and merge them into a single query)

Usage of vendor code

- Directly copying-pasting vendor code (whether proprietary or open source) and modifying it may incur license violations

- In addition application developers become responsible for maintaining the vendor code being copied

Usage of vendor code: resolution

- Remove vendor code and introduce a proper library if possible

- If no proper library available or difficult to extend the logic provided use an alternative library or write your own logic

Defensive programming

- Defensive programming refers to a practice where developers tend to be “overly” protective

- Examples include excessive input validation, unnecessary checks for null etc.

```
// product ID can never be null
if(product.getId() != null) {
        validateProduct(product);
}
```

Defensive programming: resolution

- Remove unnecessary checks and unneeded blocks of code related to excessive validation …
- In many cases (such as input validation) defensive programming is a good practice so extra caution needs to be put in cleaning up extra checks

Lack of proper comments

- Many practitioners promote the fact that source code needs to be “self-documented”

- While this is true in many cases there are certain situations where comments are needed

- These include for example methods that implement certain algorithms or classes that provide complex business logic

Meaningless/misleading comments

- These are fairly common in practice …

```
// assigning product count
int productCount = products.size();
```

```
// retrieve records from MySQL
Records records = mongoDbUtil.getRecords()
```

Boat anchor

- Boat anchor refers to a piece of code that serves no particular purpose in the current project

- It is typically source code that has been intentionally added for (eventual) future use

- Similar to dead/unused code antipattern where code has been in many cases either used in previous version or unintentionally added

Boat anchor

- Example:

```
public Configuration initConfiguration() {
        Configuration config = readConfiguration();
        writeConfigurationToDB(config);
        return config;
}
```

writeConfigurationToDB() writes configuration to the RDBMS but it is not used by the application (or other applications)

Golden hammer

- Golden hammer is a software technology or concept applied obsessively in the project and based on previous usage

- Example: *The system uses the NoSQL database **NoOneIsReallyUsingThatMuch** and its API because it is so cool and I used in two of my previous projects*

Dead end

- A dead end refers to a library of component that is modified by developers but is no longer maintained and supported by the supplier

- In that manner the support burden transfers to the application developers

- Example: *We use a patched version of the* ***ESAPI*** *library from 2011 for input validation that is not longer supported but there are several critical security issues uncovered*

Class level code smells at a glance

- large (god) class
- lack of cohesion
- feature envy
- inappropriate intimacy
- excessive coupling
- refused bequest
- lazy class / freeloader
- indecent exposure
- downcasting
- constant class
- data clump
- poltergeist class
- sequential coupling
- large and complex hierarchies

Large (god) class

- Large (aka god) class may refer to classes that are too big in terms of lines of code or methods provided

- In a slightly different manner a god class refer may refer to a class that is highly complex (also referred to as “brain class”)

Large (god) class: resolution

- In most cases the standard practice is to extract extra classes from the god class

- In certain situations it is also sufficient to extract methods from the god class to existing classes in the system

Lack of cohesion

- Cohesion refers to the degree at which the components of the class relate to each other

- If a class provides multiple distinct roles it has low cohesion …

```
public class UsersAndRolesManager {
    …
}
```

- Some cases (such as the above) might be more obvious based on the naming being used

Lack of cohersion: resolution

- Distinct roles provided by the class need to be extracted to multiple other classes …

- Similar to god class in certain situations moving methods to existing classes also alleviates the lack of cohesion

Feature envy

- Feature envy refers to a situation where the class uses more methods and fields from other classes than its own

- A basic example is delegating extensively to setters and getters from other classes:

```
public class ValueHolder {
   private ValueDTO valueDTO;

   public String getName() {
      return valueDTO.getName();
   }
   ….
}
```

Feature envy: resolution

- In cases of simple delegation remove methods and use the referenced class

- In more complex scenarios moving methods and fields to the referenced class is a possible solution

- Extracting methods to the referenced class is also a possibility

Inappropriate intimacy

- Similar to feature envy but typically both classes are referring to each other and are typically used together

- From a slightly different aspect one class can refer extensively to internal members of another class

```
public class User {
   String name;

   public UserDetails createDetails
      (String name, String email) {
      this.name = name;
      return new UserDetails(mail);
   }
   ….
}
```

```
public class UserDetails {

   private User user;
   public String getName() }
       return user.name;
   }
   ….
}
```

Inappropriate intimacy: resolution

- Move the logic from one of the classes to the other by extracting methods and fields

- In the case of bidirectional communication if possible remove the relation from one of the classes to the other

Excessive coupling

- Excessive coupling refers to the dependency of a class to many other classes:

```
public class UserManager {
   private EntityManager entityManager;

   private AuthManager authManager;

   private UserValidator validator;


   ….
}
```

- Coupling introduces difficulty in extending and testing the target class

Excessive coupling: resolution

- Coupling can be reduced by:
  - introducing interfaces rather than concrete dependencies
  - extracting a class
- Dependency injection frameworks help in reducing coupling

Refused bequest

- Refers to the scenario where a child class is not using ("refusing to use") logic from the parent class

- In certain scenarios the parent and child classes are not relating logically to each other

```
public class UserUtils {
    public boolean hasRole
        (String role) {
        …
    }
    public String[]
        getBlockedUsers(){
        …
    }
}
```

```
Public class User
     extends UserUtils {

    public void export () {
         if(hasRole(
              Roles.EXPORT)) {
              …
          }
    }
}
```

Refused bequest: resolution

- Introduce delegation rather than sub-classing in related classes

- Extract methods from the parent class

- Extract interfaces from parent class and make interested children inherit from them

Lazy class/freeloader

- A class that does too little and is used rarely
- Lazy classes might be:
  - classes introduced with the intention to be used in future
  - classes that have reduced use over time

Lazy class/freeloader: resolution

- Lazy classes may be removed from the system …

Indecent exposure

- Indecent exposure occurs when a class exposes more of its internal structure to clients than needed

- These are typically fields and methods that need to be private or at least having package/protected access but are marked as public

Data clump

- A data clump refers to a set of classes typically used together

```
public class AuthManager {

   private User user;

   private UserDetails userDetails;

   …
}
```

Data clump: resolution

- Extract methods from to one of the classes in the data clump
- Introduce a wrapper object that encapsulates the data clump

Poltergeist class

- A short lived, typically stateless class used to provide initialization or supports the operations of other classes

Poltergeist class: resolution

- Inline the logic of the poltergeist …

Sequential coupling

- Sequential coupling refers to a situation where the methods of a particular class need to be used in a particular order by calling classes

```
public class Server {

   public void initLogging() {
       …
   }

   public void initDB() {
       …
   }
}
```

```
public class Client {

   private Server server;0

   public void createServer() {
       server.initLogging();
        server.initDB();
   }
}
```

Sequential coupling: resolution

- Introduce a method that implement the coupling sequence and hide details from callers

```
public class Server {

   public void initialize() {
       initLogging();
        initDB();
   }
   …
}
```

```
public class Client {

   private Server server;0

   public void createServer() {
       server.initialize();
   }
}
```

Method level code smells at a glance

- too many parameters
- large cyclomatic complexity
- deep nesting
- lack of cohesion
- long method
- excessively long/short identifiers
- use of string concatenation
- use polymorphism instead of switch
- primitive obsession
- excessive return of data
- excessively long line of code
- busy waiting
- error hiding
- magic numbers/strings
- comparing objects with ==
- not checking for null
- long message chains
- resource leaks

Concurrency-related code smells at a glance

- excessive number of threads
- excessive/unneeded locks
- non-atomic operations assumed to be atomic
- two-state access bug anti-pattern
- double-checked locking
- using sleep() for thread synchonization
- notify instead of notify all
- deadlock/livelock/race condition

Java code smells

- Good list with additional Java-specific code smells: https://www.odi.ch/prog/design/newbies.php#0
- SonarQube built-in rules for Java code smells: https://rules.sonarsource.com/java/type/Code%20Smell/RSPEC-1068
- Oracle Java secure coding guidelines: https://www.oracle.com/technetwork/java/seccodeguide-139067.html

Code smell discovery and refactoring

Questions ?