# Refactoring

# What is refactoring

- Series of equivalent transformations on the source code
  - So that is still passes the same tests
  - Adds no new functionality
  - The code becomes more readable
  - The code becomes less redundant

# Goals of refactoring

- Make code easier to read
- Remove duplications for better maintenance
  - DRY
  - Less code, hence less errors
- Eliminate technical debt

# Technical debt

- What is it?
  - Less-than-perfect design and implementation decisions
  - Like hot-fixes to make the code work right now
  - Like incomplete code with TODO comments
- Risk
  - Higher maintenance cost
- Tests show these as bugs
  - The lack of certain test cases is a technical debt

# When to refactor?

- Most responsible moment
  - When the same code piece starts appearing in multiple places in the code
  - YAGNI
- Code is unreadable for other team members
  - Pair programming
- Before release to make the code readable for everyone
- When a bug is found

# Refactoring techniques

- Extract vs. inline method

```
width = (value − 0.5)*16;
```

```
width = emToPixels(value);
…
int emToPixels(float ems) {
  return (ems − 0.5) * 16;
}
```

- When (not) to use:
  - DRY, better maintenance vs. unneeded methods
  - Better readability vs. better performance

# Refactoring techniques

- Extract vs. inline variable

```
(x-x0)*(x-x0)+(y-y0)*(y-y0)
```

```
horizontalError = (x-x0)*(x-x0)
verticalError = (y-y0)*(y-y0)
horizontalError + verticalError
```

- When (not) to use:
  - Too long line, conditional expression in **if**
  - More readable code vs. performance
  - Inline only when **return**ing a value

# Refactoring techniques

- Temporal variable vs. function call

```
width = (value − 0.5)*16;
```

```
width = emToPixels(value);
…
int emToPixels(float ems) {
  return (ems − 0.5) * 16;
}
```

- When (not) to use:
  - Same code appears in multiple places (DRY) vs. appears only once
  - Better readability vs. better performance

# Refactoring techniques

- Introducing constant

```
class A {
  width = (value − 0.5)*16;
}
```

```
class A {
  public static final int SIXTEEN = 16;
  width = (value − 0.5)*SIXTEEN;
}
```

- When (not) to use:
  - Same value appears in multiple places with the same meaning (DRY) vs. appears only once
  - Easier maintenance vs. better performance

# Refactoring techniques

- Introducing private method

```
class A {
  width = (value − 0.5)*16;
}
```

```
class A {
  width = emToPixels(value);
  …
  private int emToPixels(float ems) {
    return (ems − 0.5) * 16;
  }
}
```

- When (not) to use:
  - Same code appears in multiple places (DRY) vs. appears only once
  - Better readability of a long method vs. better performance

# Refactoring techniques

- Moving field, method

```
class A {
  B b;
  int myVariable;
  void myFunction() {…}
}
```

```
class B {
  A a;
  int myVariable;
  void myFunction() {…}
}
```

- When (not) to use:
  - The method operates more on the fields of the other class
  - The variable is more frequently used in the methods of the other class

# Refactoring techniques

- Moving class

```
package com.mycompany;
class Person {
}
```

```
package com.mycompany.common;
class Person {
}
```

- When (not) to use:
  - A class may be relevant for other projects vs. used only in this project

# Refactoring techniques

- Constants vs. enum

```
class A {
  public static final int
    MODE_SIMPLE = 0;
 public static final int
    MODE_EXTENDED = 1;
}
```

```
class A {
  enum Mode {
    SIMPLE, EXTENDED
  }
}
```

- When (not) to use:
  - Constants can be grouped by meaning vs. independent constants

# Refactoring techniques

- Type field vs. association

```
class A {
  public static final int
    MODE_SIMPLE = 0;
 public static final int
    MODE_EXTENDED = 1;
  int mode;
}
```

```
class A {
  Mode mode;
}
enum Mode {
   SIMPLE, EXTENDED
}
```

- When (not) to use:
  - Used in database design when creating a 3NF schema

# Refactoring techniques

- Type field vs. subclasses

```
class A {
  public static final int
    MODE_SIMPLE = 0;
 public static final int
    MODE_EXTENDED = 1;
  int mode;
}
```

```
class A {
}
class SimpleA extends A {
}
class ExtendedA extends A {
}
```

- When (not) to use:
  - The type field is metadata, not a state variable of a class vs. simpler architecture
  - The type variables do not change during the lifecycle of the object

# Refactoring techniques

- Type field vs. states

```
class A {
  public static final int
    MODE_SIMPLE = 0;
 public static final int
    MODE_EXTENDED = 1;
  int mode;
}
```

```
class A {
  AMode mode;
}
class AMode{}
class SimpleAMode extends AMode {
}
class ExtendedAMode extends AMode {
}
```

- When (not) to use:
  - The type field is metadata, not a state variable of a class vs. simpler architecture
  - The type variables does change during the lifecycle of the object

# Refactoring techniques

- Extract vs. inline class

```
class Person {
  String firstName;
  String lastName;
  …
}
```

```
class Person {
  Name name;
  …
}
class Name {
  String firstName;
  String lastName;
}
```

- When (not) to use:
  - One class does the job of two vs. one class does nothing
  - Better readability vs. better performance (less memory)

# Refactoring techniques

- Push up vs. pull down

```
class A {
  A() {…}
}
class B extends A {
  int i;
  void myMethod() {…}
  B() { a(); }
}
class C extends A {
  int i;
  void myMethod() {…}
  C() { a(); }
}
```

```
class A {
  int i;
  void myMethod() {…}
  A() {a();}
}
class B extends A {
  B() { super(); }
}
class C extends A {
  C() { super(); }
}
```

- When (not) to use:
  - Duplication is subclasses vs. field or method not used in superclass

# Refactoring techniques

- Extract superclass, interface

```
class A {
  void myMethod() {…}
}
class B {
  void myMethod() {…}
}
```

```
interface I {
  void myMethod();
}
class S implements I{
  void myMethod() {…}
}
class A extends S{
  ~~void myMethod() {…}~~
}
class B extends S{
  ~~void myMethod() {…}~~
}
```

- When (not) to use:
  - Same or similar behaviour in two unrelated classes vs. different behaviour for unrelated classes
  - Common interface if the behaviour is only similar, superclass if it is the same

# Refactoring techniques

- Collapse hierarchy

```
class A {
  …
}
class B extends A {
  …
}
```

```
class A {
…
}
```

- When (not) to use:
  - Subclass has no added value
  - Architecture simplified

# Refactoring techniques

- Reference vs. value

```
class Order {
  final Customer customer;
}
```

```
class Order{
  Customer customer;
  …
}
class Customer{
  Address address;
}
```

- When (not) to use:
  - Frequently vs. slowly changing attributes

# Refactoring techniques

- Local objects vs. extracted class

```
void complexAlgorithm() {
  int myVariable1;
  float myVariable2;
  complexComputation
   (myVariable1, myVariable2);
}
```

```
void complexAlgorithm() {
  new MyClass().compute();
}
class MyClass {
  private int myVariable1;
  private float myVariable2;
  void compute() {…}
}
```

- When (not) to use:
  - Too long method and some local variables are related
  - Long method can be split up into multiple methods
  - Separation of responsibilities
  - Better readability vs. better performance

# Refactoring techniques

- Final classes vs. wrapper or subclass

```
final class Person {
  String firstName;
  String lastName;
  …
  void myMethod();
}
```

```
final class Person extends Named {
  Named name;
  …
}
class Named {
  String firstName;
  String lastName;
  void myMethod();
}
```

- When (not) to use:
  - Method needs to be added to a final class
  - Useful in GUI applications to separate data and view

# Refactoring techniques

- Delegate vs. bidirectional associations

```
class A {
  B b;
  B getB();
}
class B {
  A a;
  A getA() :
}
new A().getB().doSomething();
new B().doSomething();
```

```
class A {
  B b;
  B getB();
}
class B {
}
new A().getB().doSomething();
```

- When (not) to use:
  - Single way of accessing a method, it helps avoiding spaghetti code
  - Bidirectional associations are more difficult to maintain
  - Bidirectional associations make classes independent, hence useful in data models
  - Use delegate when an association is not used

# Refactoring techniques

- Inheritance vs. delegation

```
class B {
  int size();
}
class A extends B {
}
new A().size();
```

```
class A {
  B b;
  int size() { return this.b.size(); }
}
class B {
  int size() { … }
}
```

- When (not) to use:
  - When the subclass is not an extension or a partial of the superclass, so those can be split vs. reduced code because of the delegation

# Refactoring techniques

- Constructor vs. factory

```
class A {
  A() {…}
}
```

```
class A {
  private A() {…}
  static A create() {
    A = new A();
    …
    return A;
  }
}
```

- When (not) to use:
  - When it is not certain that an instance should be constructed
  - Factory can do additional operations
  - Factory can create an instance of a subclass

# Refactoring techniques

- Public field vs. encapsulated field

```
class A {
  public int i;
}
```

```
class A {
  protected int i;
  void setI(int i) { this.i = i; }
  int getI() { return this.i; }
}
```

- When (not) to use:
  - Additional operations are to be performed when a field is accessed
  - Single point of access, so easier maintenance

# Refactoring techniques

- Direct field access vs. setters/getters

```
class A {
  protected int i;
  A() { i = 1; }
}
```

```
class A {
  protected int i;
  A() { setI(1); }
  void setI(int i) { this.i = i; }
}
```

- When (not) to use:
  - Additional operations are to be performed when a field is accessed
  - Behaviour can be customized vs. better performance

# Refactoring techniques

- Modifiable vs. unmodifiable collection

```
class A {
  private Set s;
  Set getS() { return this.s; }
}
```

```
class A {
  private Set s;
  Set getS() {
    return Collections.
      unmodifiableSet(this.s);
  }
}
```

- When (not) to use:
  - Single point of access, so easier maintenance

# Refactoring techniques

- One vs. multiple temporal variables

```
tmp = 16;
…
tmp = 32;
```

```
width = 16;
…
height = 32;
```

- When (not) to use:
  - Temporal variable has a meaning
  - Better readability vs. better performance (less memory)

# Refactoring techniques

- Array (map) vs. object

```
myMap = {}
myMap['b'] = 'a'
myMap['a'] = 1
```

```
myClass = new MyClass()
myClass.b = 'a'
myClass.a = 1
class MyClass {
  int a;
  char b;
}
```

- When (not) to use:
  - Avoid errors of using a wrong key − type checking
  - Whitelisting keys
  - Class is easier to document

# Refactoring techniques

- Parameter list vs. object or map parameter

```
void myMethod(int i, float f);
myMethod(1,1.0)
```

```
void myMethod(A a);
class A {
  int i;
  float f;
}
myMethod(new A(1,1.0));
```

- When (not) to use:
  - Improve readability
  - Parameter list as class or map can be reused
  - Overloading vs. whitelisting map
  - Return value and parameter value are both objects

# Refactoring techniques

- Inout parameter vs. local variable

```
Float emToPixels(Float ems) {
   …
   return ems;
}
```

```
Float emToPixels(final Float ems) {
   Float result = ems;
   …
   return result;
}
```

- When (not) to use:
  - Make a function a black-box to remove accidental errors
  - Easier to debug

# Refactoring techniques

- Parameter check vs. exception

```
void myMethod(int i) {
  if (i == 0) return 1;
  …
}
if (myMethod(1) == 1) { … }
```

```
void myMethod(int i) {
  if (i == 0) throw
    new InvalidParameterException();
  …
}
try {
  myMethod(1);
} catch (Exception e) { … }
```

- When (not) to use:
  - Return value does not indicate invalid behaviour vs. exception handling overhead

# Refactoring techniques

- TODO comment vs. exception

```
void myMethod() {
  // TODO: not yet implemented
}
```

```
void myMethod() {
  throw new
    NotImplementedException();
}
```

- When (not) to use:
  - Always

# Refactoring techniques

- Conditional expression vs. polymorphism

```
class A {
  Mode m;
  void myMethod() {
    switch(m) {
      case SIMPLE: return a();
      case EXTENDED: return b();
    }
  }
}
```

```
class A {
  Mode m;
  myMethod();
}
class SimpleA extends A {
  void MyMethod() { return a(); }
}
class ExtendedA extends A {
  void MyMethod() { return b(); }
}
```

- When (not) to use:
  - Creating a new class vs. adding a new branch to existing code
  - Identical cases can be reflected in the class hierarchy

# Refactoring techniques

- Multiple branches vs. iterator and algorithm

```
if (a==1) { print(1); }
else if (a==2) { print(2); }
else if (a==3) { print(3); }
```

```
List<Integer> a =
    Arrays.asList(1,2,3);
for (Integer i : a) { print(i); }
```

- When (not) to use:
  - Same pattern appears in multiple branches
  - Usually standard library algorithms can be used

# Design patterns

- Solution for frequently recurring problems
- Good practice, but
  - YAGNI
  - Not necessary optimal for the actual state of the project
  - Code should converge to these with a sequence of refactoring changes

# Performance

- Readability and performance are in contradiction
  - In script languages refactoring causes performance loss
  - In C/C++ it may cause performance loss
- Performance optimization is sometimes reverting changes made during refactoring
  - Hence done at the end of the project
  - Code becomes much harder to read and debug

# Code optimization for performance

- Compiled languages
  - Bit level operations
  - Unfolding loops
  - Caching data
  - Inlining function
  - Macros instead of functions
- Interpreted languages
  - Optimize garbage collection
  - Memory allocation