BAD CODE SMELLS

FROM THE PERFORMANCE, RELIABILITY AND TESTABILITY PERSPECTIVE

Barbora Bühnová
buhnova@fi.muni.cz

LAB OF SOFTWARE ARCHITECTURES
AND INFORMATION SYSTEMS

FACULTY OF INFORMATICS
MASARYK UNIVERSITY, BRNO

lasaris

Bad code smells

- Violation of **CLEAN code** and **SOLID principles**
- Defined by **Martin Fowler** in *Refactoring: Improving the Design of Existing Code* [1]
- **Examples**
  - **Duplicated code**: identical or very similar code in more locations.
  - **Long method**: a method, function, or procedure grown too large.
  - **Large class, God class**: a class that has grown too large.
  - **Too many parameters**: a long list of parameters is hard to read, and makes calling and testing the function complicated.
  - **Feature envy**: a class that uses methods of another class excessively.
  - **Inappropriate intimacy**: a class that has dependencies on implementation details of another class.

Outline of the talk

- Performance
  - Bad code smells
  - Tactics
- Reliability
  - Bad code smells
  - Tactics
- Testability
  - Bad code smells
  - Tactics
- Conflicts between quality attributes

Outline of the talk

- Performance
  - Bad code smells
  - Tactics
- Reliability
  - Bad code smells
  - Tactics
- Testability
  - Bad code smells
  - Tactics
- Conflicts between quality attributes

Bad code smells for Performance

- Let's assume our code is perfectly **CLEAN**
- **What about performance?**
  Are there any performance code smells we could check for?

Let's discuss four **performance smells**:

- Smell #1: Redundant Work
- Smell #2: One by One Processing
- Smell #3: Long Critical Section
- Smell #4: Busy Waiting

Motivating example #1: Fibonacci Sequence

- 1, 1, 2, 3, 5, 8, 13, 21, …
- Fib(0) = Fib(1) = 1
  Fib(n+2) = Fib(n+1) + Fib(n)   where n≥0

In Java:

```
public int fibonacci(int n) {
  if(n <= 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Smell #1: Redundant Work

- Description
  - A time-consuming method computes the same many times in a single execution path
- Consequences
  - A slower execution time since the time-consuming operation is performed multiple times
- Solution
  - Call the heavy method only once and store the result for further reuse

Note: Applies also in more complex scenarios, such as caching of database results in distributed systems.

lasaris

Example #1: Fibonacci refactored

```
Map<Integer,Integer> cache1 = new HashMap<Integer,Integer>();

long fibonacci(int n) {
  if (cache1.containsKey(n))
    return cache1.get(n);
  if (n==0 || n==1) {
    int var1 = 1;
    cache1.put(n, var1);
    return var1;
  }
  int var2 = fibonacci(n-1) + fibonacci(n-2);
  cache1.put(n, var2);
  return var2;
}
```

Motivating example #2: Search

```
private ArrayList<Item> list = new ArrayList<Item>();

List<Item> findGreaterThan(int value) {
  List<Item> ret = new ArrayList<Item>();

  for (Item item : list) {
    if (item.isGreaterThan(value)) {
      ret.add(item);
    }
  }
  return ret;
}
```

Smell #2: One by One Processing

- Description
  - Overused linear search/processing
- Consequences
  - Slower performance
- Solution
  - Use smarter algorithms and/or data structures (binary search, sorted collections, map with precomputed search predicates)

Note: Become familiar with the performance of operations you execute on different types of data structures. And think about the complexity of your algorithms.

Example #2: Search refactored

```
private List<Item> list = new ArrayList<Item>();
private List<Item> var1 = new SortedList<Item>( ... );

...

List<Item> findGreaterThan(int value) {
  return subList(var1, value);
}
```

Motivating example #3: Password Cracking

```
static List<String> passwordsToCheck;

// launch 100 threads and FOR each thread
void run() {
  while (!passwordsToCheck.isEmpty()) {
    synchronized(passwordsToCheck) {
      if (!passwordsToCheck.isEmpty()) {
        String pwd = passwordsToCheck.remove(0);
        checkPassword(pwd);
      }
    }
  }
}
void checkPassword() { ... }
```

Smell #3: Long Critical Section

- Description
  - Unnecessary code performed in a critical section
- Consequences
  - More like single-threaded model
- Solution
  - Remove the code outside the critical section

Note: Sometimes it is favorable to use multiple locks within a class to enable partial locking of an object. See an example below.

Example #3: Password Cracking refactored

```
static List<String> passwordsToCheck;

// launch 100 threads and FOR each thread
void run() {
  while (!passwordsToCheck.isEmpty()) {
    synchronized(passwordsToCheck) {
      if (!passwordsToCheck.isEmpty()) {
        String pwd = passwordsToCheck.remove(0);
      }
    }
    checkPassword(pwd);
  }
}
void checkPassword() { ... }
```

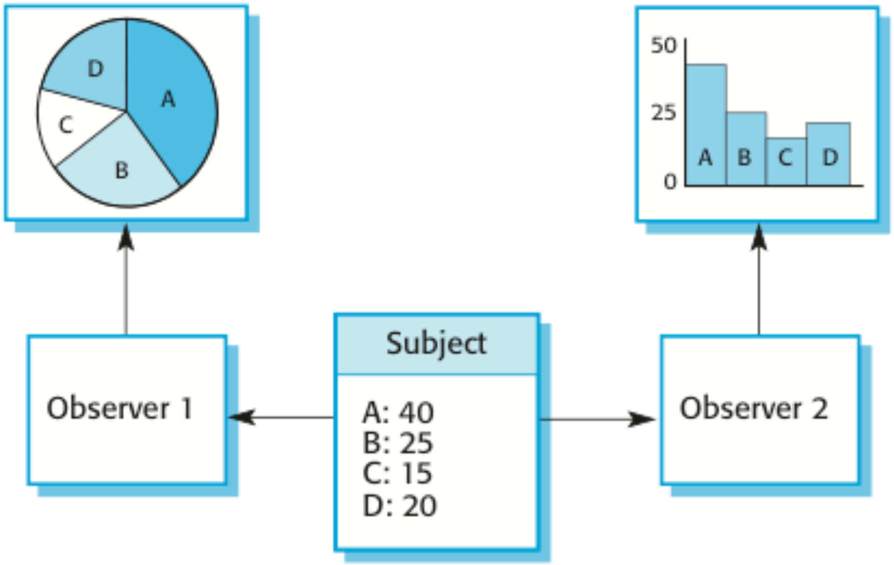
Example #3.b: Multiple locks within a class

```
public class MyUpdater {
  private long var1 = 0;
  private long var2 = 0;

 public void updateVar1() {
    synchronized(this) {
      // update var1
    }
  }
  public void updateVar2() {
    synchronized(this) {
      // update var2
    }
  }
}
```

```
...
private Object lock1 = new Object();
private Object lock2 = new Object();

public void updateVar1() {
  synchronized(lock1) {
    // update var1
  }
}
public void updateVar2() {
  synchronized(lock2) {
    // update var2
  }
}
...
```

Smell #4: Busy Waiting

- Description
  - Repeatedly checking if something interesting happened (e.g. value changed, user input arrived).
- Consequences
  - A lot of work with mostly no value, slowing down the system
- Solution
  - Hollywood principle: "Don't call us, we'll call you."
  - Observer pattern (Gang of Four book)



Tactics for Performance

- Tactic #1: Take a **profiler** into action
  - **Do not guess** where the performance problem is.
    Start your profiler and find the bottlenecks objectively.
  - It helps you to understand what is **happening in the background**.
- Tactic #2: Examine **complexity** and **frequency** of your computations
  - **Complexity** – Maybe you can do the thing **more efficiently**.
  - **Frequency** – Maybe you can do the thing **less often**.
- Tactic #3: **Concurrency**
  - Only if you **understand all aspects and consequences** of parallel execution.
- Tactic #4: Control the use of **resources**
  - Balance the load, control access, cache, replicate, etc.

Outline of the talk

- Performance
  - Bad code smells
  - Tactics
- Reliability
  - Bad code smells
  - Tactics
- Testability
  - Bad code smells
  - Tactics
- Conflicts between quality attributes

Bad code smells for Reliability

- Smell #1: Input Kludge
  - Check all **inputs for validity**! On all user interfaces and service interfaces.
- Smell #2: Blind Faith
  - Do **not trust others** (limit access to your code, check bug fixes), **nor yourself** (check the correctness of your results).
- Smell #3: Poorly Handled Exceptions
- Smell #4: Unguarded Sequential Coupling
  - Assumptions on the **right ordering of method calls** without control.
- Smell #5: Fashionable Coding
  - Usage of all the **new cool technologies** and constructs you do not really understand.

Tactics for Reliability

- Tactic #1: **Monitor** what is going on
  - **Acceptance checking** for individual methods and code fragments, **events** collection, **processing** and **logging**.
- Tactic #2: Handle **exceptions** carefully
  - Think twice about exception handling strategy and **responsibilities** inside the system.
- Tactic #3: Make your system **fault tolerant**
  - **Redundancy** and **self-healing**, e.g. seamless rebinding to a new service provider.
- Tactic #4: Implement **restart/recovery** capabilities
  - Redirection to a **filled-in form** when the form submission fails.
  - System **diagnostics** and **clean-up** after major failure.

Note 1: We only care about SW reliability (because this is a Software Quality course), not HW, although HW fault tolerance is a very interesting topic.

Note 2: We assume that we do not deal with an ultra-reliable system.
If so, other mechanisms would need to be in place (e.g. …).

Outline of the talk

- Performance
  - Bad code smells
  - Tactics
- Reliability
  - Bad code smells
  - Tactics
- Testability
  - Bad code smells
  - Tactics
- Conflicts between quality attributes

lasaris

Bad code smells for Testability

- Smell #1: **Global State**
  - Do not allow your objects to communicate secretly.
- Smell #2: Lack of **Dependency Injection**
  - Make your dependencies explicit.
- Smell #3: **Law of Demeter** violation
  - Only talk to your immediate friends.
- Smell #4: Misplaced and Hard Coded **new** Operator
  - Do not mix factory and service code.

Note: In over 90% of cases, Global State is the problem.

General advice: If your code is difficult to test, do not ask how to hack it, but what is wrong with that code!

Motivating example #1: Secret Communication

```
class X {
  ...
  X() { ... }

  public int doSomething() { ... }
}

int a = new X().doSomething();
int b = new X().doSomething();
```


lasaris

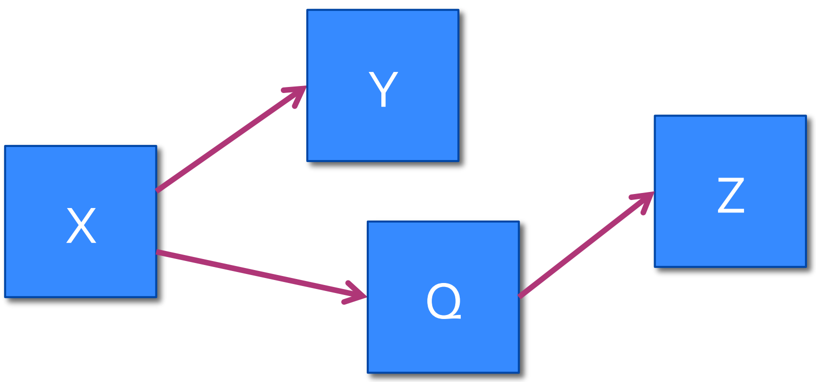
Motivating example #1: Secret Communication

```
class X {
  ...
  X() { ... }

  public int doSomething() { ... }
}

int a = new X().doSomething();
int b = new X().doSomething();
```

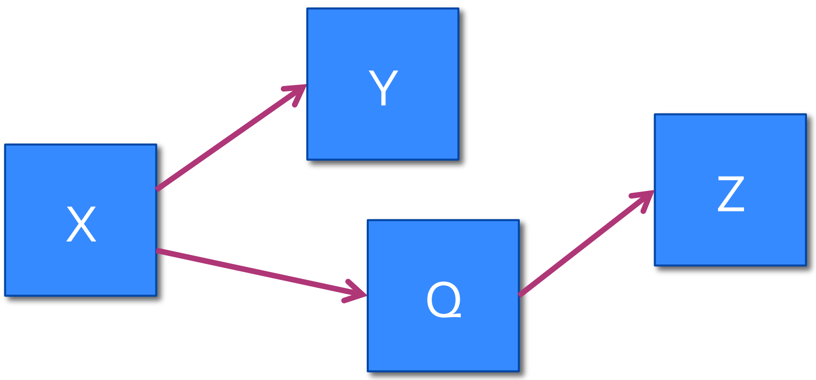
Does a==b ??

Motivating example #1: Secret Communication

```
a = new(X) →
```

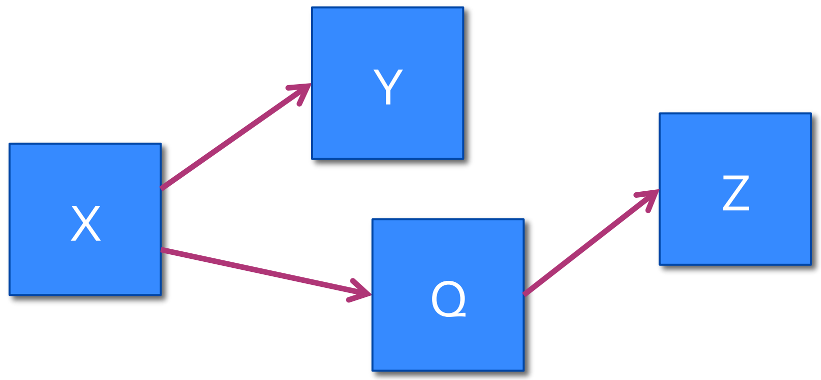



```
b = new(X) →
```



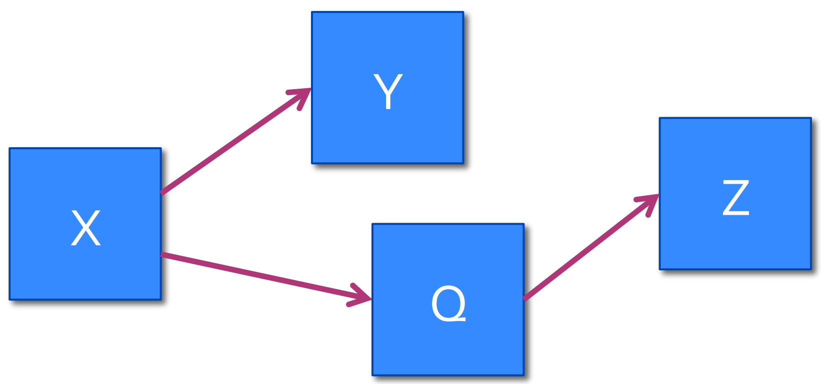
Motivating example #1: Secret Communication

```
a = new(X) →
a.doSomething()
```



a==b ✔

```
b = new(X) →
b.doSomething()
```

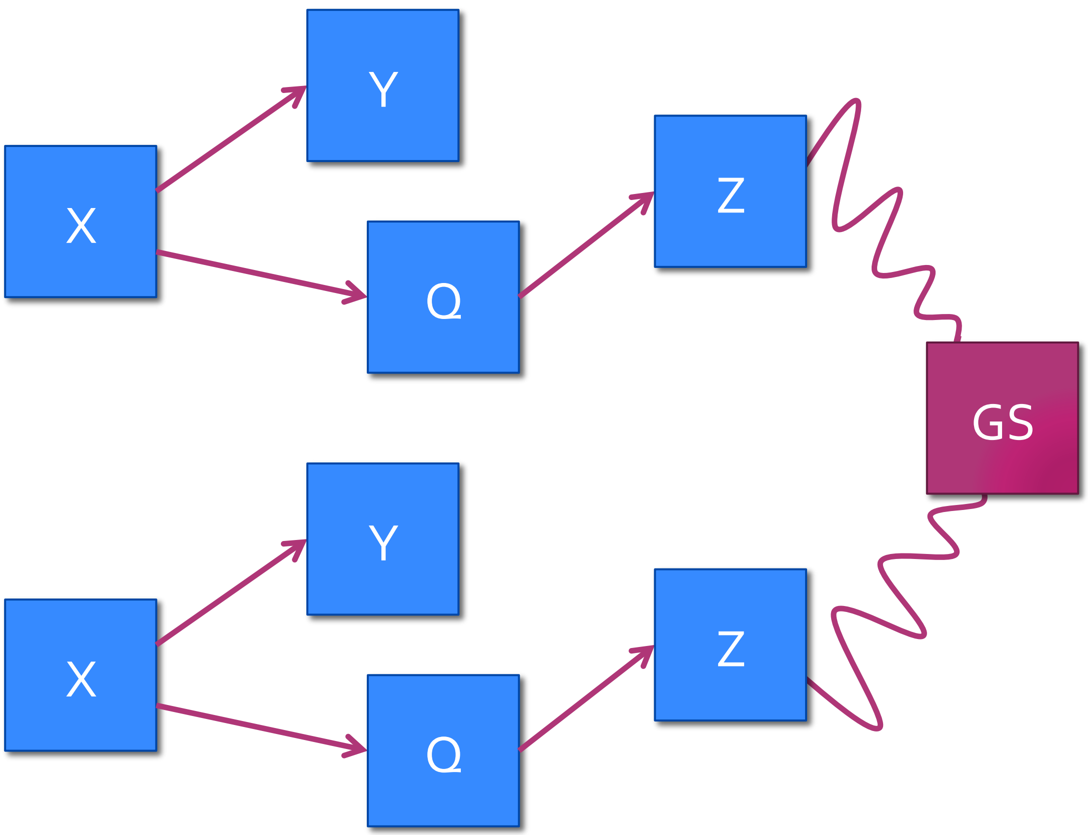


Motivating example #1: Secret Communication

```
a = new(X) →
a.doSomething()
```

a==b ✖

```
b = new(X) →
b.doSomething()
```



lasaris

Smell #1: Global State

- Multiple executions can produce different results
  - Test flakiness
  - Order of tests matters
  - Cannot run tests in parallel

What about Singletons?

- Unbounded location of state
  - Transitive dependencies
- Hidden Global State in JVM
  - System.currentTime()
  - new Date()
  - Math.random()

lasaris

Motivating example #2: Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Motivating example #2: Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

java.lang.NullPointerException
at talk3.CreditCard.charge(CredicCard.java:48)

Motivating example #2: Deceptive API

```
testCharge() {
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard(“1234567890121234”);
  cc.charge(100);
}
```

Motivating example #2: Deceptive API

```
testCharge() {
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

java.lang.NullPointerException
at talk3.CreditCardProcessor.init(CredicCardProcessor.java:146)

Motivating example #2: Deceptive API

```
testCharge() {
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Motivating example #2: Deceptive API

```
testCharge() {
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

java.lang.NullPointerException
at talk3.OfflineQueue.start(OfflineQueue.java:16)

Motivating example #2: Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Motivating example #2: Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard(“1234567890121234”);
  cc.charge(100);
}
```

- **CreditCard API lies**
  - It pretends to not need the CreditCardProcessor even though in reality it does.

Motivating example #2: Better API

```
testCharge() {
  ??
  CreditCard cc;
  cc = new CreditCard(“1234567890121234”, ccProc);
  cc.charge(100);
}
```

Motivating example #2: Better API

```
testCharge() {
  ??
  ccProc = new CreditCardProcessor(queue);
  CreditCard cc;
  cc = new CreditCard(“1234567890121234”, ccProc);
  cc.charge(100);
}
```

Motivating example #2: Better API

```
testCharge() {
  ??
  queue = new OfflineQueue(db);
  ccProc = new CreditCardProcessor(queue);
  CreditCard cc;
  cc = new CreditCard(“1234567890121234”, ccProc);
  cc.charge(100);
}
```

Motivating example #2: Better API

```
testCharge() {
  db = new Database(...);
  queue = new OfflineQueue(db);
  ccProc = new CreditCardProcessor(queue);
  CreditCard cc;
  cc = new CreditCard("1234567890121234", ccProc);
  cc.charge(100);
}
```

Motivating example #2: Better API

```
testCharge() {
  db = new Database(...);
  queue = new OfflineQueue(db);
  ccProc = new CreditCardProcessor(queue);
  CreditCard cc;
  cc = new CreditCard("1234567890121234", ccProc);
  cc.charge(100);
}
```

Dependency Injection

Smell #2: Lack of Dependency Injection

- **Dependency injection** makes your **dependencies explicit**
  - It does not make the dependencies in your code better or worse
  - It only makes them visible
- If there are **too many dependencies**, **do not blame DI**!
  - The dependencies have always been there, DI only showed them to you
- **Dependency injection** enforces the **order of initialization at compile time**
  - Compiler helps to prevent illegal test setup

Won't my system get flooded with arguments passed around?

Smell #2: Lack of Dependency Injection

- **Dependency injection** makes your **dependencies explicit**
  - It does not make the dependencies in your code better or worse
  - It only makes them visible
- If there are **too many dependencies**, **do not blame DI**!
  - The dependencies have always been there, DI only showed them to you
- **Dependency injection** enforces the **order of initialization at compile time**
  - Compiler helps to prevent illegal test setup

Won't my system get flooded with arguments passed around?

NO

```
testCharge() {
  db = new Database(...);
  queue = new OfflineQueue(db);
  ccProc = new CreditCardProcessor(queue);
  CreditCard cc;
  cc = new CreditCard("1234567890121234", ccProc);
  cc.charge(100);
}
```

© Miško Hevery [4]

lasaris

Smell #3: Law of Demeter violation

Law of Demeter: "Only talk to your immediate friends"

- If an object needs links to too many objects, there may be something wrong with the object
- Revealed by **Dependency Injection**

- "Our code often smells because we have a **few objects** doing **too much work**, which requires them to **know about too many other objects**." [Brandon Keepers]
  - A nice rule of thumb is to check if we are able to **describe the purpose** of each class and method without using AND and OR.

Single Responsibility Principle

lasaris

Smell #4: Misplaced and Hard Coded new Operator

To avoid misplace, clearly separate:

- "Code with a whole bunch of **`new`** operators and no **`if`** statement"
  = code responsible for **starting and wiring things**, i.e. Factories.
- "Code with a whole bunch of **`if`** statements and no **`new`** operator"
  = code that is actually **doing something**, i.e. Services.

To avoid hard coding, make sure that:

- Constructor only **constructs** the object and its dependencies.
  - Doing any **other work** in the constructor can significantly **hinder testing**.
  - You can end up doing unrelated work (e.g. sending emails) every time you need the object in your test.

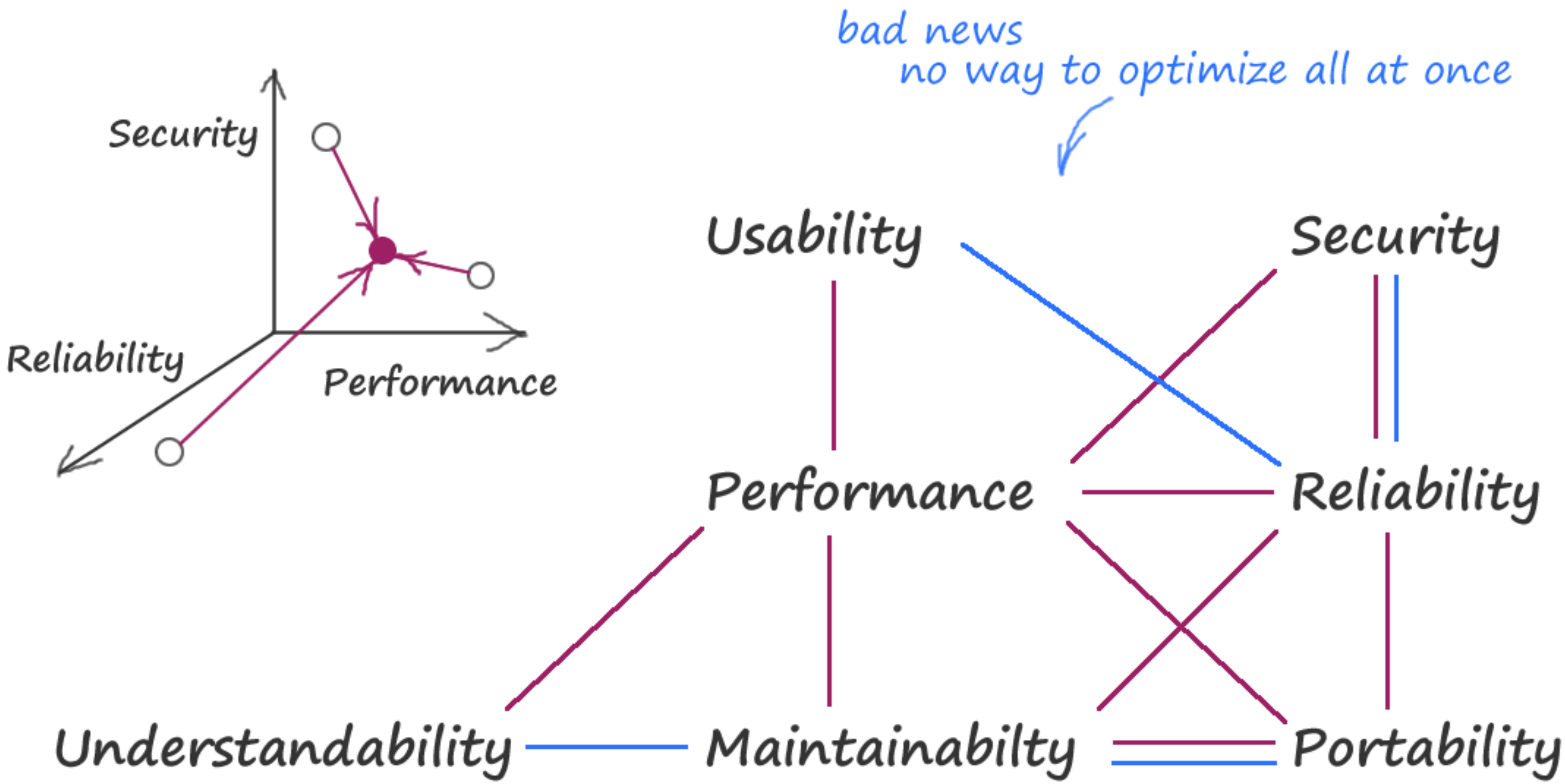
Tactics for Testability

- Tactic #1: Write CLEAN code
  - Simplicity matters.
- Tactic #2: Avoid global state
  - Including its **hidden** forms.
- Tactic #3: Separate interfaces from implementation
  - Make it possible to **exchange implementations** during testing.
- Tactic #4: Make your dependencies explicit
  - It makes the life of developers/testers easier, and then even **compiler can help** to inspect it.
- Tactic #5: Separate factories from business logic
  - During testing it is important to have **access to each of these parts** without mixing it with the other.

Outline of the talk

- Performance
  - Bad code smells
  - Tactics
- Reliability
  - Bad code smells
  - Tactics
- Testability
  - Bad code smells
  - Tactics
- Conflicts between quality attributes

Conflicts between quality attributes



Takeaways

- **Bad Code Smells** apply also to **quality attributes**.
  - They are just **not that easy to Google.**
- **Tactics** in comparison to **Bad Code Smells** are usually defined on a higher level of abstraction.
- Each **tactic for a specific quality** attribute can act as an **anti-pattern** for a different quality attribute.
  - That is where **conflicts** between quality attributes emerge.

thanks for listening

Barbora Bühnová, FI MU Brno
buhnova@fi.muni.cz
www.fi.muni.cz/~buhnova

contact me

References

- [1] Martin Fowler et al. Refactoring: Improving the Design of Existing Code, Addison-Wesley, Mar 2012. ISBN 978-0133065268.
- [2] Patrycja Wegrzynowicz. Automated Refactoring of Performance and Concurrency AntiPatterns. YouTube, Jan 2013. Available at https://www.youtube.com/watch?v=XLCbb6dcsJQ.
- [3] Brandon Keepers. Why Our Code Smells. YouTube, June 2012. Available at https://www.youtube.com/watch?v=JxPKljUkFQw.
- [4] Miško Hevery. The Clean Code Talks - Global State and Singletons. YouTube, Nov 2008. Available at https://www.youtube.com/watch?v=-FRm3VPhseI.
- [5] Miško Hevery. Guide: Writing Testable Code, Google, Nov 2008. Available in the int. syllabus in IS.
- [6] Slava Imeshev. Architecture for Scaling Java Applications to Multiple Servers. YouTube, Aug 2012. Available at https://www.youtube.com/watch?v=DhKpqGDXRCk.
- [7] Lars Lundberg et al. (editors). Software quality attributes and trade-offs, Blekinge Institute of Technology, June 2005.
- [8] Mikael Svahnberg et al. A Method for Understanding Quality Attributes in Software Architecture Structures. In Proc. of SEKE'02, pages 819-826. ACM New York, 2002. ISBN:1-58113-556-4.
- [9] Michael Feathers. Escaping the Technical Debt Cycle. YouTube, Oct 2014. Available at https://www.youtube.com/watch?v=7hL6g1aTGvo.