



High performance. Delivered.

# Java Coding Pitfalls

Experience from code review  
of 1,5M lines of code

tomi vanek

December 2013



consulting | technology | outsourcing

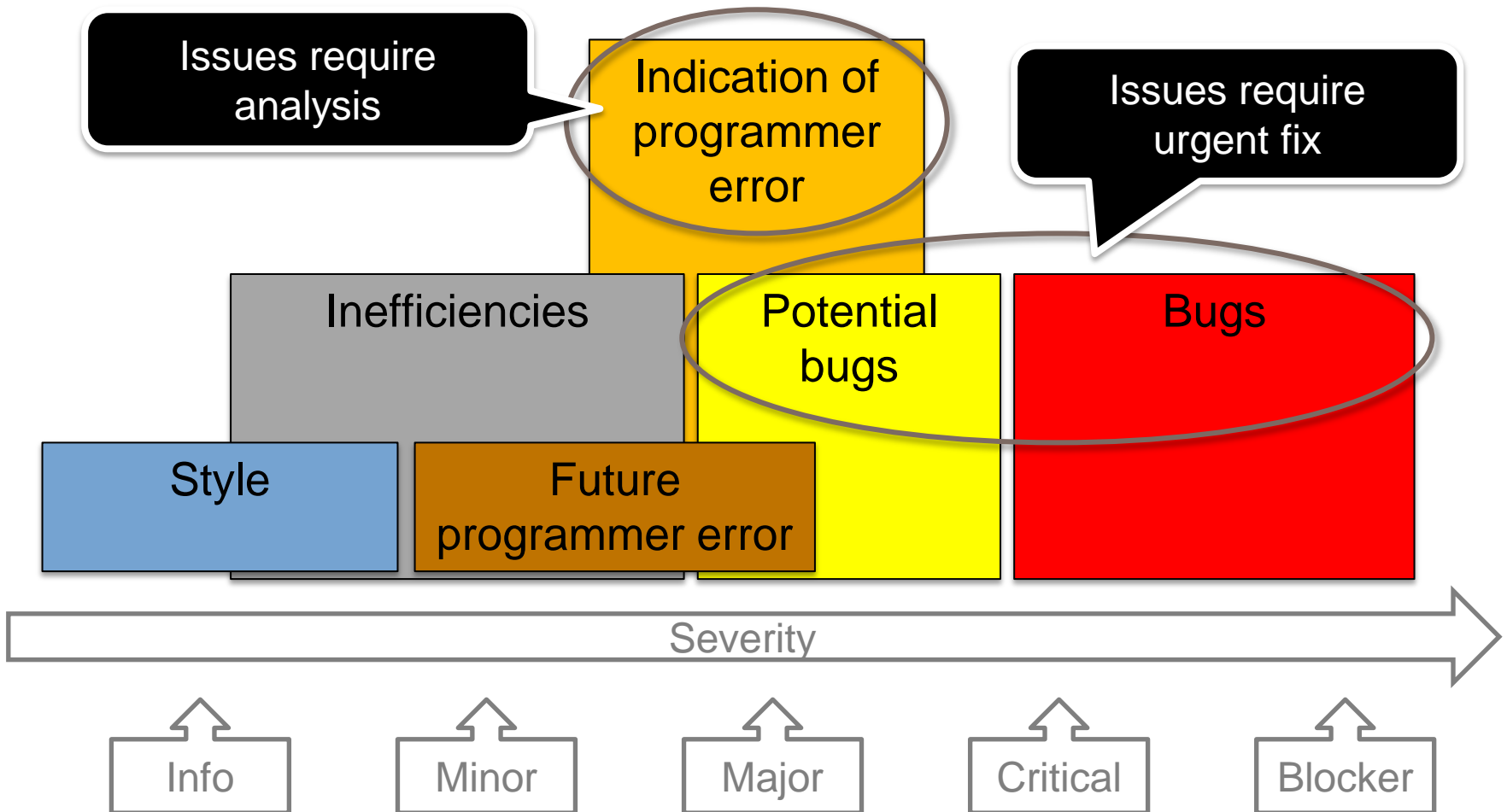
# Background

---

- Code review of project with 1,5M lines of code
- Static code analysis with SonarQube
  - PMD, CheckStyle, FindBugs
- Refactoring of code
- Unit test review and refactoring

Categories of code issues

# Categories of issues



# Examples for bugs

---

- Logic Errors that would lead to null pointer exception
- Failures to close file handlers, IO streams or database connections
- non thread safe behavior in multithreaded environment
- Methods designed to check equality that always return false (or true)
- Impossible class casts
- Missing overridden *hashCode()* if method *equals()* is overridden

# Examples for potential bugs

---

- Potential NPE, which happen only under certain conditions
- Null checks that dereference the items they are checking
- Math operations that use wrong precision or lose precision (i.e class ImmutableMoney uses double for money value, instead of BigDecimal)
- Comparing strings and objects with == and != instead of .equals() method
- Conditionals that assign instead of compare: if (foo = 1) { ...
- Catch block that swallow exceptions rather than logging or throwing further

# Examples for indication of programmer error

---

- Empty conditional body, empty catch or finally block
- Re-check for null (copy-paste without reading code)
- Commented out code
- Unused fields, local variables, private methods
- Inconsistency in JavaDoc and method declaration (parameters, return values, checked exceptions)
- Exception created and dropped rather than thrown
- Call to equals() comparing different types
- Self assignment of field
- Missing Break In Switch
- Code duplicity

Equality



# Comparison of String objects using == or !=

## Two string variables

```
if (foo == bar) {  
    // action  
}
```



```
if (foo != null  
    && foo.equals(bar)) {  
    // action  
}
```



## String variable and constant string

```
if (foo == "undefined") {  
    // action  
}
```



```
if ("undefined".equals(foo)) {  
    // action  
}
```



## Is string variable an empty string?

```
if (foo != "") {  
    // action  
}
```



```
if (StringUtils.isNotBlank(foo)) {  
    // action  
}
```



Never use == and != for string comparison

# Common errors in methods `compareTo`, `equals` and `hashCode`

---

- Class defines `compareTo(...)` and does not define `equals()`
  - Class uses inherited `Object.equals()`, that is incompatible with definition of equality in the overridden method `compareTo()`
- Class inherits `equals()`
  - Class uses inherited `Object.hashCode()`, that is incompatible with definition of equality in the overridden method `equals()`
- Use of class without a `hashCode()` method in a hashed data structure causes that the object may not be found


Whenever you implement `equals()`, you **MUST** also implement `hashCode()`

# Unintentional overloading of equals()


---

**If equals overloads the method with parameter type (i.e. own class type), the contains method in List can erroneously return false, even if the object is in the list**

```
public class Foo {  
    public boolean equals(Foo o) {  
        ...  
    }  
}
```



```
public class Foo {  
    @Override public boolean equals(Object o) {  
        ...  
    }  
}
```




**Never forget @Override annotation by equals()**

# Equals method should not assume anything about the type of its argument

---

**equals() - should expect null and any class type**



```
@Override public boolean equals(Object o) {  
    if (this == o) return true;  
    if (!(o instanceof Foo)) return false;  
    Foo that = (Foo) o;  
    if (!super.equals(that)) return false;  
    if (bar != null ? !bar.equals(that.bar) : that.bar != null) return false;  
    ...  
}
```

Use IDE code generation for creating equals() and hashCode() methods.

# Class cast exceptions

**instanceof will always return false – typically a copy-paste bug**

```
public void execute(Foo foo) {  
    if (foo instanceof Bar) {  
        ...  
    }  
}
```

Never executes body  
of if statement

**Impossible cast – typically a copy-paste bug**

```
if (foo instanceof Foo) {  
    Bar that = (Bar) foo;  
    ...  
}
```

By execution throws  
ClassCastException

**Class Cast Exception With To Array**

```
Integer[] a = (Integer [])c.toArray();
```

ClassCastException

```
Integer[] b = (Integer [])c.toArray(new Integer[c.size()]);
```

Avoid copy-paste by programming.

# Test for floating point equality, test for time equality

## Floating point number is NEVER precise

```
if (foo == 0.) {  
    // action  
}
```

```
double EPSILON = .0001;  
if (Math.abs(foo) < EPSILON) {  
    // action  
}
```

Initiation of 2 time-related variables with current time may or may not get equal value. Time has is typically also precision in the computation context.

```
Date foo = new Date();  
Date bar = new Date();  
...  
if (foo == bar) {  
    ...  
}
```

May be  
false, if GC  
is running  
between  
initialization  
statements

```
Long epsilon =  
    TimeUnit.HOURS.toMilliseconds(1);  
Date foo = new Date();  
Date bar = new Date();  
if (bar.getTime() - foo.getTime()  
    < EPSILON) {  
    ...  
}
```


Test imprecise value types with precision interval.

Exception handling

# Exception information and logging

## Log and Throw


Avoid logging and throwing – as this results in multiple log messages for the same problem. Exception should be logged at last resort error handler.



```
...  
} catch (FooException ex) {  
    LOG.error(ex);  
    throw new BarException(ex);  
}
```

## Re-throw include original exception

Exception must keep stack cause chain information, to provide complete information for the problem triage.




```
...  
} catch (FooException ex) {  
    LOG.error(ex);  
    throw new BarException();  
}
```

## Proper solution

**Not catch exception that cannot be handled** on the method level.

If additional details of conditions / state that caused the exception should be added to exception, throw wrapper exception with details and original exception.



```
...  
} catch (FooException ex) {  
    throw new  
        BarException("Details..", ex);  
}
```

If exception cannot be solved, it should not be caught.



# Catch

## Avoid catching Throwable

Catches also Error classes - fault situations of JVM, that cannot be recovered in the application business logic (i.e. OutOfMemoryException)

```
try {  
    // guarded block  
} catch (Throwable ex) {  
    // What to do with Error-s?  
}
```

## Swallowed exception

Catch block does not handle caught exception and continues in processing, or returns null

```
...  
} catch (FooException ex) {  
    return null;  
}
```

## Empty catch or finally block

Mostly programmer error – copy-paste error, forgotten code by refactoring or not finished code. If valid empty catch, need comment with valid explanation.


```
...  
} catch {  
    // no code here...  
}
```

Process “expected” exceptions, or throw to caller, if “no cure” on method level.

# Finally

## Return From Finally Block


If finally is called after throwing an exception, return in finally causes “swallow” of exception, and the method returns normally




```
...  
} finally {  
    return result;  
}
```

## Throw exception from finally block

If the cleanup logic in finally block throws an exception, the exception from the guarded block will get lost (will be replaced by exception thrown in finally)



```
...  
} finally {  
    cleanup();  
}
```



Problem if  
cleanup  
throws  
exception


Never end method processing (by return or exception) in finally block.

# Exception in catch block

---

## Exception without preserving stack trace

Mainly in case an exception is thrown in catch block. In this case the cause exception, that initiated catch block processing has to be connected to the new exception before throw with `initCause()` method.




```
...  
} catch (FooException fx) {  
    try {  
        // code can throw some exc.  
    } catch (BarException bx) {  
        bx.initCause(fx);  
        throw bx;  
    }  
}
```

If exception happens in catch block, preserve exception cause chain (stack trace).


# Resource not closed

## Failures to close file handlers or database connections

close resources (database connections, files, etc.) in finally block



```
try{
    Connection conn =
        getConnection();
    // If exception is thrown,
    // the following connection
    // close is not called,
    // and stays open.
    con.close();
} catch (SQLException ex) {
    LOG.error(ex);
}
```



```
Connection conn = null;
try{
    conn = getConnection();
    // work with DB connection
} catch (SQLException ex) {
    throw new RuntimeException(ex);
} finally {
    try {
        if (con != null) con.close();
    } catch (SQLException ex) {
        LOG.error(ex);
    }
}
```

Close in finally block

Use framework (Spring) for resource cleanup outside of business code.

Null

# Logic errors that would lead to null pointer exception


## Misplaced Null Check

Pay attention to order in logical operands - if first statement determines the result (true in OR, false in AND), the second is not executed


Split complex logic into simpler nested conditions and/or extract values to local variables

Always do null check against collection before using it in for-each statement

If collection processed in for each loop is null, the loop generates a `NullPointerException`



```
if (foo.getBar() > 0 && foo != null) {  
    ...  
}
```



```
if (foo != null && foo.getBar() > 0) {  
    ...  
}
```



```
if (fooList != null) {  
    for (Foo foo : fooList) {  
        // process foo  
    }  
}
```

`NullPointerException` typically occurs due to logical errors in code.

Threads and session

# Multithreaded access

---

## Common violations:


- Non-final static fields have non-thread-safe type
  - Calendar
  - DateFormat
- Initiation and update of static field from non-static (instance) method

Check for threadsafety in JavaDoc of classes, that are used as mutable field types.




# Mutable violation in Spring beans

- Singleton Spring beans (default configuration) must be designed as **stateless**, without any instance value fields
- In Spring beans instantiated in application context (singletons) should be only fields wired by Spring dependency injection with references to Spring beans



```
public class Foo {  
    private Calendar start;  
  
    public void setStart(  
        Calendar start_) {  
        start = start_;  
    }  
    ...  
}
```

Not thread safe



```
public class Foo {  
    private Bar bar;  
  
    public void setBar(Bar bar_) {  
        bar = bar_;  
    }  
    ...  
}
```

Only for dependency injection

Singleton Spring beans must be designed as stateless.

# HTTP session

---

- Each object stored into session **MUST** be serializable
  - To manage load, container can persist the session, or synchronize to other nodes in the container cluster
- Limit putting data into session
  - Session size has significant performance impact by increased load
- Use `setAttribute` by changes of object value in session
  - `setAttribute` can trigger the session synchronization logic of the container
- Store more fine-grained objects stored in session attributes, instead of single monolithic object
  - Some containers synchronize sessions in container by sending only the changed objects

**Application code should access session objects only through framework.**

Testability

# Test Initiation with non-constant values

- Symptom: Tests are not reproducible - test results are changing in time

```
public void setUp() throws Exception {  
    fileDate = new Date();  
    applicationDate = new Date();  
    ...  
}
```


Typically the 2 time values will be the same. But if the GC starts between these 2 calls, the initialized time will be different. This could cause different test results.

```
private static final String TEST_DAY = "2010-08-26";  
private static final Date DATE = new SimpleDateFormat("yyyy-MM-dd").parse(TEST_DAY);  
  
public void setUp() throws Exception {  
    fileDate = DATE;  
    applicationDate = DATE;  
    ..  
}
```

**NEVER** initiate the test starting values with current time.

# Testing implementation details instead of functionality


- Symptom: Tests fail by refactoring in implementation code, that does not change the results of the tested method. Tests do not check the results in detail (wrong



```
EasyMock
    .expect(repositoryMock.find(DOC_ID))
    .andReturn(testFile);
EasyMock.replay(repositoryMock);
testedFoo.setRepository(repositoryMock);

testedFoo.handle(requestMock, response);

EasyMock
    .verify(repositoryMock, requestMock);
```



```
EasyMock
    .expect(repositoryMock.find(DOC_ID))
    .andStubReturn(testFile);
EasyMock.replay(repositoryMock);
testedFoo.setRepository(repositoryMock);

testedFoo.handle(requestMock, response);


assertEquals(200, response.getStatus());
assertEquals(CONTENT_LENGTH,
    response.getContentLength());
// ... more assertions for result
```

Don't Replace Asserts with Verify.


Performance

# Maps and sets of URLs can be performance hogs

String representations of URLs are preferred as Map keys over URL Objects in order to avoid network call overhead.



```
final URL ACC = new URL("accenture.com");  
Map<URL, Boolean> urls = new HashMap<URL, Boolean>();  
  
urls.put(ACCENTURE, Boolean.TRUE);  
Boolean value = urls.get(ACC);
```



```
final String ACC = "accenture.com";  
Map<String, Boolean> urls = new HashMap<String, Boolean>();  
  
urls.put(ACCENTURE, Boolean.TRUE);  
Boolean value = urls.get(ACC);
```

**Use String representation of URL as key in hash containers.**

# Dead store to local variable, unused private methods and fields

---

- Instantiation of unused variables, unused private fields and unused private methods cause unnecessary increase of memory consumption and processing time.

```
Date date = new Date();
```

Unnecessary object initialization – in next statement thrown away and replaced with a new instance.

```
date = new SimpleDateFormat("yyyy-MM-dd").parse("2013-12-25");
```



```
final Date date = new SimpleDateFormat("yyyy-MM-dd").parse("2013-12-25");
```



**Avoid unnecessary instantiation of unused variables.**

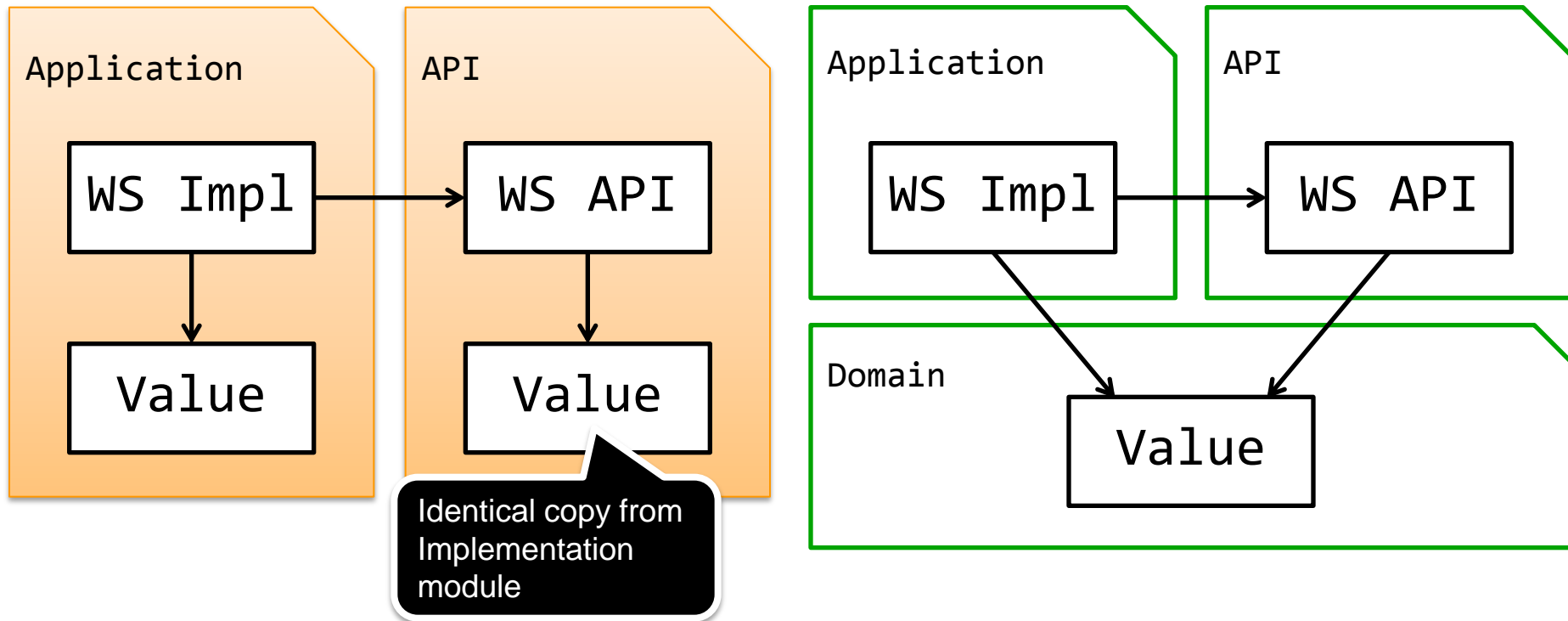


Design

# Cyclic module dependency

## Copy class into different module

Cyclic module dependencies solved with code copying - duplication  
In this example web service API module was externalized, but domain classes remained in the application. Circular dependency was solved by class duplication.



By modularization avoid code duplication.

# Same class name in different namespaces

---

- java.util.Date
- java.sql.Date
- Domain specific Date class

```
private Date startDate = new Date();
```



Which “Date” is  
in this code??!

Unique names avoid misunderstandings by code maintenance.

# Class size and complexity

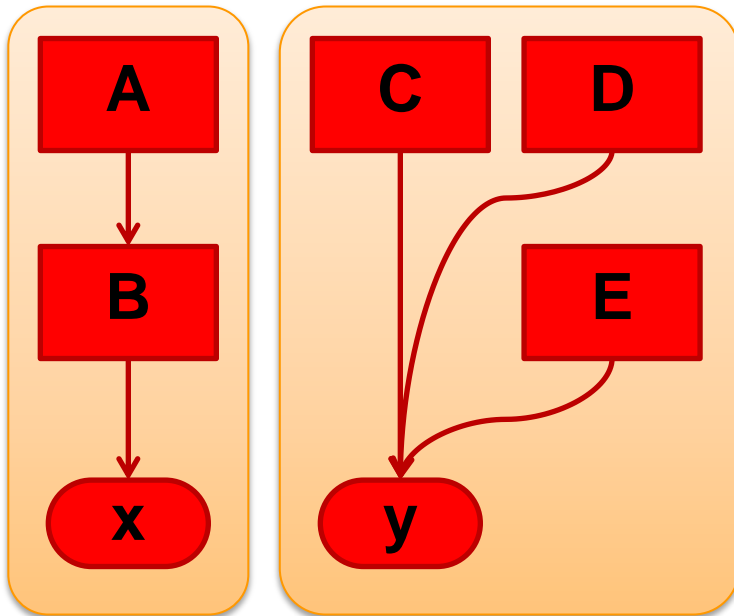
---

- Class size
  - number of fields and number of methods
- Public interface size
  - number of public methods and constants
- Size of methods
- Cyclomatic complexity
  - how many different paths can be performed during a block of code execution
  - calculation: Each method that not is “accessor” (getter) starts the counter with 1. For each keyword/statement (if, for, while, case, catch, throw, return, &&, ||, ? ) found, the counter is incremented by 1

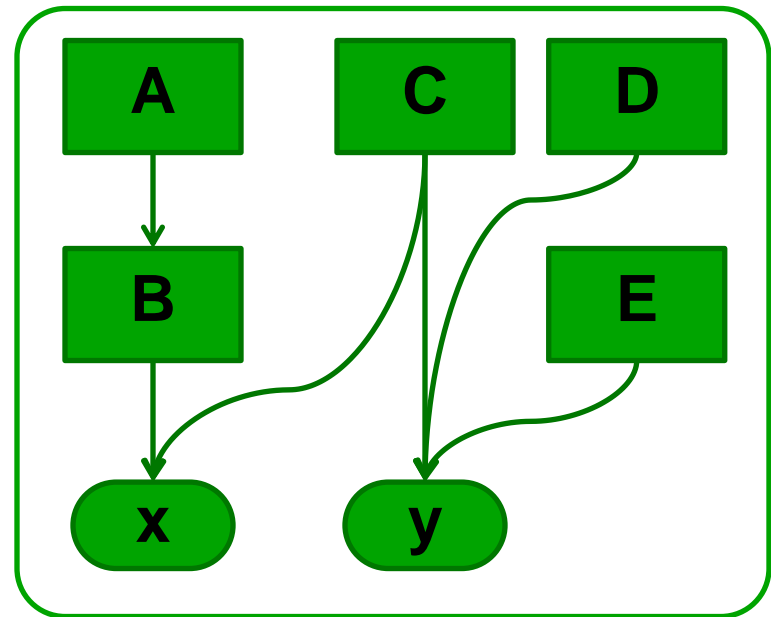
**Complex classes and methods are hard to read, understand and change.**

# Class cohesion

- LCOM4 (Lack of cohesion)
  - how focused the responsibilities of class are (Wikipedia)



LCOM4 = 2



LCOM4 = 1

Code with low cohesion is difficult to maintain, test, reuse and understand.

Discussion

# Reference

---

- Books
  - Joshua Bloch: Effective Java (2<sup>nd</sup> edition)
  - Campbell, Papapetrou: SonarQube in Action
  - Steve McConnell: Code Complete (2<sup>nd</sup> edition)
- Internet Articles
  - <http://www.odi.ch/prog/design/newbies.php>
  - <http://javaantipatterns.wordpress.com/>
  - <http://www.javapractices.com/home/HomeAction.do>
  - <https://today.java.net/article/2006/04/04/exception-handling-antipatterns>