



# Best practices in software development

**Jacek Panachida**

**[jacek.panachida@rec-global.com](mailto:jacek.panachida@rec-global.com)**

**Engine developer**

**Research & Engineering Center Sp. z o. o.**

**15/12/2010**



# Agenda

- Introduction
- Best practices
- Programming principles
- Programming rules of thumb



## What is best practice?

The most efficient (least amount of effort) and effective (best results) way of accomplishing a task, based on repeatable procedures that have proven themselves over time for significant number of people

## What is best practice? Cont.

There is, however, no practice that is best for everyone or in every situation, and no best practice remains best for very long as people keep on finding better ways of doing things.

- „Best practices” has implications of finality, obedience, authority, and universality
- Possible better terms are „better practices” or „current thinking”
- „Better practices” seems to seek better ways, which may even lead to tweaking the suggested practice to make it even better



# Software development process

- Requirements
- Specification
- Architecture
- Design
- **Implementation**
- **Testing**
- Deployment
- Maintenance



## Lazy initialization

- Avoid creating unnecessary objects and always prefer to do Lazy Initialization

```
public class Countries {  
    private List countries;  
    public List getCountries() {  
        //initialize only when required  
        if(null == countries) {  
            countries = new ArrayList();  
        }  
        return countries;  
    }  
}
```

- Never make an instance fields of class **public**

```
public class MyCalender {  
    public String[] weekdays =  
        {"Sun", "Mon", "Tue", "Thu", "Fri", "Sat", "Sun"};  
}
```

- Make it private and add getter

```
private String[] weekdays = {"Sun", "Mon", "Tue", "Thu", "Fri", "Sat", "Sun"};  
public String[] getWeekdays() {  
    return weekdays;  
}
```

- But array is still accessible. Return a clone of array instead of array itself

```
public String[] getWeekdays() {  
    return weekdays.clone();  
}
```

## Minimize mutability

- Always try to minimize mutability of a class
- To make a class immutable you can define its all constructors private and then create a public static method
- Immutable classes are simple, they are easy to manage. They are thread safe.

- Prefer composition over inheritance
- Try to use Interfaces instead of Abstract classes
- You can not inherit multiple classes in Java but you can definitely implements multiple interfaces

## Limit the scope of variables

- Always try to limit the scope of local variable
- Minimizing the scope of a local variable makes code more readable, less error prone and also improves the maintainability of the code
- Always initialize a local variable upon its declaration. If not possible at least make the local instance assigned null value

## Use standard libraries

- Try to use standard library instead of writing your own from scratch
- They are generally of high quality, often improve their performance over time.
- They usually form *de facto* standards

- This will save a lot of if else testing for null elements
- The following qualify as "empty" objects:
  - The empty String
  - Zero-length arrays
  - Collections containing 0 items
  - Maps containing 0 items

- Avoid floating point numbers
- Rounding errors will always occur when using these data types - they are unavoidable
- For calculations of monetary amounts it is better to use BigDecimal

## Be specific in throws clause

- Do not group together related exceptions in a generic exception class - that would represent a loss of possibly important information

- In general, when an exception occurs, it can be thrown up to the caller, or it can be caught in a catch block
- When catching an exception, some options include
  - inform the user (strongly recommended)
  - log the problem
  - send an email describing the problem to an administrator

- Such data is often critical for understanding and solving the problem, and can greatly reduce the time needed to find a solution
- In addition, if you are defining exception classes yourself, you may even design your constructors to *force* the caller to pass the pertinent data

## Always close streams

- Streams represent resources which you must always clean up explicitly, by calling the close method
- If multiple streams are chained together, then closing the one which was the last to be constructed, and is thus at the highest level of abstraction, will automatically close all the underlying streams.

- When reading and writing text files
  - It's almost always a good idea to use buffering
  - It's often possible to use references to abstract base classes, instead of references to specific concrete classes
  - There is always a need to pay attention to exceptions (e.g. IOException and FileNotFoundException)

- Classes or methods are too long
- Little or no Javadoc
- Inappropriate use of multiple return statements
- Excessive use of the `instanceof` operator
- Using exceptions to define regular program flow

- Indirection - named constants replacing "magic numbers"
- Minimizing visibility - private fields, package-private classes
- Generic references (polymorphism) - using high level references (interfaces or abstract classes) instead of low level references (concrete classes)



## Use source code control system

- It acts as the definitive version of the source
- It allows for automated daily backups
- It allows *time travel*, in the sense of letting you create a snapshot of the code as it appeared at some specific time in the past
- It lets you create *parallel universes*, by letting you define *branches* for your project

## Do not break portability

- Do not use hard coded file names and paths, use the `File` class, and allow file paths to be configured during deployment
- Remember that some systems have case sensitive file names (Unix), while others do not (Windows)
- Do not rely on thread scheduling and thread priorities to define program logic

- Do not hard code sizes and positions of GUI elements
- Do not rely on a specific screen resolution
- Do not hard code colors using numeric values, use the symbolic constants instead

- Do not hard code text sizes
- Use the `System.getProperty(String)` method to refer to items which depend on the system, such as line terminators and path separators

- The standard specifies a consistent style and format for source code, within the chosen programming language
- Ideally, you shouldn't be able to tell by looking at it who on the team has touched a specific piece of code.

- The first lines of a method are usually devoted to checking the validity of method arguments
- This is particularly important for constructors
- Use e.g. `IllegalArgumentException`, `NullPointerException` or `IllegalStateException`

## Validate user input

- Verify that all input from the user contains valid characters and represents a valid value before using that value in your application
- Be restrictive. You can always ease up on the restrictions

- Implementing any code that creates file paths, HTML, SQL statements or other strings that another subsystem parses requires care
- Unless you disallow all special characters in your input validation, you will need to make sure you properly escape or in some other way



## Never keep passwords in clear text

- Applications should not refer to the clear text versions of passwords - not in the database, not when logging, nor in any other area
- Use salt and hash function to encrypt the password



## Minimise error message information

- Provide useful error messages to your users, but keep the details in the log file



## Have someone review your code

- Having someone else read through your code almost always results in them asking you questions
- It provides an opportunity to step back and take a fresh look at your work

## Know your application

- Don't start testing without understanding the requirements
- If you test without knowledge of the requirements, you will not be able to determine if a program is functioning as designed

## No assumptions in testing

- Don't start testing with the assumption that there will be no errors
- As a tester, you should always be looking for errors



There is no bug free application

- No matter how much testing you perform, you can't guarantee a 100% bug free application
- Try to explore as many bugs as you can, but prioritize your efforts on basic and crucial functions



## Think like an end user

- Don't think only like a technical guy. Think like customers or end users
- Think how an end user will be using your application

## Don't repeat yourself

- „Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”
- When the DRY principle is applied successfully, a modification of any single element of a system does not change other logically-unrelated elements

- A fail-fast system is designed to immediately report at its interface any failure or condition that is likely to lead to failure
- Fail-fast components are often used in situations where failure in one component might not be visible until it leads to failure in another component.

- The PoLK is a specific case of loose coupling
- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit
- Each unit should only talk to its friends. Don't talk to strangers.
- Only talk to your immediate friends

- „Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”
- Code obeying the principle doesn't change when it is extended. No code review nor new tests are necessary.

## Principle of good enough

- It favours quick-and-simple (but potentially extensible) designs over elaborate systems designed by committees
- Once the quick-and-simple design is deployed, it can then evolve as needed, driven by user requirements
- Ethernet, the Internet protocol and the World Wide Web are good examples of this kind of design.

- Applies to user interface design, software design, and ergonomics
- When two elements of an interface conflict, or are ambiguous, the behaviour should be that which will least surprise the user

- SRP states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class
- The reason it is important to keep a class focused on a single concern is that it makes the class more robust

- SoC is the process of separating a computer program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program.
- Examples: MVC pattern, HTML and CSS.

- It states that you are allowed to copy and paste the code once, but that when the same code is replicated three times, it should be extracted into a new procedure.
- Duplication in programming is a bad practice because it makes the code harder to maintain.

- The rule states that, for many events, roughly 80% of the effects come from 20% of the causes
- Microsoft noted that by fixing the top 20% of the most reported bugs, 80% of the errors and crashes would be eliminated

- The rule suggests that arbitrary limits on the number of instances of a particular entity should not be allowed
- An entity should either be forbidden entirely, one should be allowed, or any number (presumably, to the limit of available storage) of them should be allowed



# References

- Wikipedia <http://en.wikipedia.org>
  - [http://en.wikipedia.org/wiki/Best\\_practice](http://en.wikipedia.org/wiki/Best_practice)
  - [http://en.wikipedia.org/wiki/Category:Programming\\_principles](http://en.wikipedia.org/wiki/Category:Programming_principles)
  - [http://en.wikipedia.org/wiki/Category:Programming\\_rules\\_of\\_thumb](http://en.wikipedia.org/wiki/Category:Programming_rules_of_thumb)
- Business Dictionary <http://www.businessdictionary.com>
- Virapatel <http://viralpatel.net/blogs/2010/02/most-useful-java-best-practice-quotes-java-developers.html>
- Collected Java Practices <http://www.javapractices.com>
- Software Testing Help <http://www.softwaretestinghelp.com/category/testing-best-practices>
- Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features  
<http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm>



**Thank you**