CLEAN CODE DEVELOPMENT - CHECKLISTE

Basics

- Kill Useless Code
- Delete Useless Comments
- Invest in a Precise Naming
- Avoid Magic Numbers (meaning of numbers)
- Prefer Polymorphism to if/else or switch/case
- Ensure that "Changes have Local Consequences"
- Fields shall define state
- Correct Exception Handling (as specifically as possible)
- Refactoring patterns: Know Refactoring Catalogue and current DIE

Coding Source

- Vertical Separation
- Explanatory Variables
- Nesting:
  Deep nested code should take on more specific and detailed tasks
- Separate Multi-Threading Code
- Encapsulate Conditionals - Example:
  if (this.ShouldBeDeleted(timer)) *liest sich einfacher als*
  if (timer.HasExpired && !timer.IsRecurrent)
- Avoid Negative Conditionals: not > 0 versus <= 0
- Encapsulate Boundary Conditions:
  loops like while / do-loop / loop-do should be marked and listed in one place

Coding Principles

- Don´t Repeat Yourself (DRY)
- Keep it simple, stupid (KISS)
- Code Communication, Simplicity, Flexibility (Beck)
- Avoid Early Optimizations
- Single Level of Abstraction (SLA)
  "*doing*" Code: d.h. Code der wirklich etwas tut (z.B. berechnet) und
  "*calling*" Code: dies ist Code, der konkreteren Code aufruft.
- Keep Configurable Data At High Levels
- Over-Configurability
  -Working with default values and subsequent adjustment
  -program can detect the optimal config
  (num. of cores, availability of data / files)
  -Remove configuration parameters, etc.
- Hidden Temporal CouplingRule, Example: kein read(); vor open();
- Don't Be Arbitrary: structure must be consistently communicated by the code

Architecture und Class Design

- Besser Instanziieren als Vererben
- Pro Komponente nur eine Aufgabe
- Aufgabentrennung
- Geheimnisprinzip durchsetzen
- Inversion of Control Container
- Verwende Dependency Injection (Framework!)
- Prüfe die Anwendung von Basispatterns
- Entwurf und Implementation überlappen nicht
- Design and Implementation must not overlap
- Implementation spiegelt Entwurf (UML2Wall)
- Visualisiere, Messe & Constraine die Architektur (Sonar, SonarJ, JDepend,..)
- Prüfe den Einsatz von Enterprise Patterns

Packages

- **Put together what belongs together**
  Common Closure + Common Reuse Principle (CCP, CRP)
  This principle means that a class should be designed for this:
  A) not to be changed B) but may indeed experience extensions
- **Visualize Package Dependencies (SonarJ, NDepend)**
- **The Dependencies of Packages mus be free of Cycles**
  Acyclic Dependencies Principle (ADP)

rACD *(Relative mean component dependency): The ACD metric is calculated by dividing the sum of all values by the number of nodes. This value is divided again by the number of nodes (or their square) and results in the rACD value.*

Look at the Ca and Ce metrics and calculate the values for the packages from them:
-Stable Packages:
Usually more abstract packets, i. e. A = AC/CC+AC is high.
-Unstable packages:
Are usually completely concretely implemented packages,
i. e. I = Ce / (Ce + Ca) is also high.

Code Quality

Use assertions! Pre / Post Conditions
Pre-conditions: Test whether all parameters contain correct values.
'Post-conditions:' Test whether all return parameters are correct.
'Assertions:' Test whether variables and objects in the function code are correct.
Pre-conditions: Test whether all parameters contain correct values.
Source Code Conventions: Check your Source Code Conventions;
tools for Java alone: Checkstyle, PMD, FindBugs

Automated Unit Tests: Automated tests offer double benefits:
Code Coverage Analysis - 3 Types of Measurement:
Statement Coverage: Number of statements that pass through a class
Conditional Coverage: How to run through conditional constructs (e. g. if/else). Ideally, the test should cover everything.
Method Coverage: How many methods of a class are called.
The entire coverage is derived from a more complex formula:
TPC = (CT + CF + SC + MC)/(2*C+S+M)
CT = conditionals that evaluated to „true“ at least once
CF = conditionals that evaluated to „false“ at least once
SC = statements covered MC = methods entered
Test first
Code-Kata: Kata means practice and is used in the context of karate. For a simple programming problem, a solution should be found within 30-60 min. Solution will then be presented. Goal is to train and improve own skills.
Mockups: Known mock tools for Java are: mockito, EasyMock, jmock
Funktionale Techniken - Important techniques are:
- Use of first-class /Higher Order Functions: e. g. func as argument or return value; generation of functions
- Pure Functions: Use of Recursion (or Tail-Recursion)
- Immutable Data: The use of data structures that cannot be changed.
- & many other concepts: lazy evaluation, strict evaluation, type systems, etc.

Split long methods and reviews
1. Create a class named exactly like the method: e.g. complexCalculation() becomes the class ComplexCalculator.
2. Create a field in the class for each parameter, var, and field used in the original method. Leave the names.
3. Build a constructor for these variables or parameters
4. Copy the old complexCalculation() method to this class. Parameters now become references to the fields!
5. Now replace the body of the original method with code that creates an instance of this class.
6. If there were fields in the old method that were used, set them after the call:

Reviews: Code reviews should be an integral part of a quality concept.
Measuring and Tracking Errors: Issue tracking systems
Static Code Analysis (Metrics) / Measure Complexity
- metrics: Basic metrics (lines, methods, classes, packages, files, duplicates, etc.), McCabe Cyclomatic Complexity, Halstead metric, Code Coverage (as pure' visit metrics'), Test coverage, Style Injuries, JDepend and coupling metrics
- belong to the latter: Pressman metrics, ACD / rARD metric

Technical Debt of a Product (Sonar)