Code Design Standards

Proactive Rules

Feature Envy 1

Do not use getter and setters except when serializing or deserializing. UI interactions are considered in the same category as serializing/deserializing.

Feature Envy 2

Avoid methods which uses only parameters and no fields and at least one parameter is stored in other object. You should consider moving that method where the parameter[s] is/are stored.

Feature Envy 3 – Law of Demeter

If the class is not a container, then it should not expose an internal dependency.

Only containers can expose a dependency.

Depend on Abstractions 1

Every dependency should be abstracted. That means that every field and method argument should have as type either a primitive type or an interface (or the equivalent).

Observation. Abstract classes are not abstractions.

Depend on Abstractions 2

A method argument could have a primitive type only if that primitive type is used in computations inside the method. If it is only used to call methods from other dependencies, then it should be abstracted.

Depend on Abstractions 3

Corollary: if two classes implement the same interface then those two classes are interchangeable at least for one client of that interface (a class that depends on that interface).

Interface Segregation

If a class depends on an interface, then it should use all methods from that interface.

Interface Inheritance

If an interface inherits another interface, then there it must be at least a scenario when an object having the child type is used in a method call as an argument requiring the parent type.

Class Cohesion 1

Two methods belong to the same class only if they share at least a common field. This rule is transitive, meaning that

- if a and b methods belong together and
- b and c belong together

then a and c belong to the same class (until b is removed).

If the interface is cohesive but the implementation is not, use composition.

Class Cohesion 2

If a set of data is shared between multiple instances of the same class, that set of data should be extracted into another class, together with the functionality using that data. The new class will be a dependency for the original class. All missing data for the new class functionality will be provided as call arguments by the original class. Values which could be considered constants are a sign for this situation. There are no constants in software.

Class Cohesion 3 – Avoid Temporary Fields

A temporary field is a field for which the first operation in all methods of the class is the initialization. Replace it with local variables in all the methods.

Methods should be Commands or Queries

A method should either change the state of the object or should return a value, not both.

Methods should use all parameters.

Self-explanatory.

Methods should have functional cohesion 1

A method should have a single purpose.

Methods should have functional cohesion 2

Look for empty lines inside a method as a hint. If the method computes a result, then it uses that result to do other things than the method should be split.

Methods should have functional cohesion 3

When implementing a formula look for hints on how the formula is described. If the formula is described as having multiple parts, implement each part in a separate method. However, if you missed that and a part becomes polymorphic, then it is time to extract that part.

Methods should have functional cohesion 4

Encapsulate conditions in methods.

Do Not Repeat Yourself

Avoid duplication.

Duplication is also when a change inside a class triggers a change inside at least another class. Keep each convention encapsulated in one place.

An exception for this rule is when an application is written in different languages and there is a justified need to duplicate some behavior in another language, for performance reasons.

Another exception is related to data, when data needs to be serialized and there is no framework which can be used, like ORMs, for justified reasons.

Do not use class inheritance

The only exception to the rule is when the framework forces the inheritance: it expects an object of a certain type and that type is not an interface(abstract). In that case, the inherited type should be used as an abstraction. Do not duplicate code. Use delegation to avoid duplication.

Do not use constants in conditions 1

Do not use constants as selectors for method behavior. If the selector is a method parameter, refactor each case in a separate method.

Observation. True, false, null are also constants.

Do not use constants in conditions 2

Do not use constants as selectors for method behavior. If the selector is a field assigned only once, refactor each case in a separate class.

Do not use constants in conditions 3

Do not use constants as selectors for method behavior. If the selector is a field which is changed during an object lifetime, use State pattern.

Do not use constants in conditions 4

Do not use constants as selectors for method behavior. If the selector is a value provided from outside your code, through a communication channel as a file, TCP socket, another framework or similar, then use the dispatcher pattern.

Observation. Be wary of constants, they are a very strong sign of polymorphism. This includes null constant.

Do not use class checking operator (such as instanceof, is, isinstance and so on)

Similar with using a field value as selector, this is a sign of polymorphism. The code guarded by the class type test should be moved to that class. Consider also if double dispatch is a solution.

Avoid class cast exception

Self explanatory. Pay attention to the previous rule also.

Default methods
You can use them only if they do not break any of the rules in this section.

Clarity

!Use this
Avoid negation operator.

Do not use “this.”.

Ottinger’ Rules – Avoid Encodings
This is valid for any type of encoding, including Hungarian notation.

Ottinger’s Rules – For a meaning use only one word
Avoid synonyms. Pick only one word for each meaning.

Ottinger’s Rules – Use a word only for one meaning
Also avoid using the same word for multiple concepts.

Ottinger’s Rules – Use problem domain names
When the concepts are part of the domain, you use those concepts when you discuss specifications, use the same names used in discussions.

Ottinger’s Rules – Use computer science names
When there is not direct correspondent in the domain you can use engineering terms. Try not to force the translation of domain concepts into engineering terms.

Ottinger’s Rules – Meaningful context
There are few names which are meaningful in and of themselves. Most, however are not. Instead, you need to place them in context. Methods, classes, packages/namespaces are context.

The term `tree' needs some disambiguation, for example if the application is a forestry application. You may have syntax trees, red-black or b-trees, and also elms, oaks, and pines. The word `tree' is a good word, and is not to be avoided, but it must be placed in context every place it is used.

Ottinger’s Rules – Don’t add artificial context
Do not duplicate context. Don’t add prefixes to a class if the package name (or namespace) is enough. The same is valid for method names, do not duplicate what the class name already expresses.

Glossary

New Feature – It is considered a new feature if it does not imply a change in the existing test cases.

Change Request – A change request involves changes in existing test cases. Functionalities involved in a change requests are all those for each at least a test case was changed.

Bug Fixing - Bug fixing should be treated exactly as a change request, i.e. a functionality is affected by a bug only if the bug affects a test case for that functionality.