Coding Rules

October 8, 2017

Coding Rule 1: Use singular nouns or noun phrases as names for classes. Use Pascal casing for their spelling.

Coding Rule 2: A class documentation comment starts with an enumeration of all the characteristics ascribed to the class itself, as well as all the characteristics ascribed to individual objects of that class.

Coding Rule 3: Include at least a version number and a list of author names in the class documentation comment.

Coding Rule 4: Use verbs or verb phrases in active tense as names for methods. Use camel casing for their spelling.

Coding Rule 5: Choose method names in such a way that their invocations read as commands or questions in a natural language.

Coding Rule 6: Start the definition of a class with a definition of all its constructors.

Coding Rule 7: Group all elements of a class related to a single property. Start a group with its public elements; finish with its private elements.

Coding Rule 8: Introduce a getter and a setter for each single-valued, mutable property. Introduce a getter for each single-valued, immutable property.

Coding Rule 9: Do not define a method that at the same time changes the state of some of the objects involved in it, and returns a result.

Coding Rule 10: Inspectors never change the observable state of any object.

Coding Rule 11: Mutators never return a result.

Coding Rule 12: In a group of elements related to a single property, inspectors precede mutators with the same access right.

Coding Rule 13: Prefer instance methods over static methods.

Coding Rule 14: Use a static method if only primitive values are involved in it.

Coding Rule 15: Use class methods if their behavior must at all times be the same for all the objects of a class. Use class methods among others to manipulate class properties.

Coding Rule 16: Use singular nouns or noun phrases as names for formal arguments and local variables. Use camel casing for their spelling.

Coding Rule 17: Start documentation comments attached to methods with a one line summary of the method's semantics.

Coding Rule 18: Clarify in a method's documentation the role of each formal argument in a clause headed by the predefined tag `@param`.

Coding Rule 19: Introduce basic inspectors to get the current value of each property ascribed to classes and their objects, and add the annotation `@Basic` to their definition.

Coding Rule 20: Add the annotation `@Immutable` to inspectors without formal arguments, whose result does not change as long as the application is executing.

Coding Rule 21: Specify the result of derived inspectors in one or more clauses headed by the predefined tag `@return`.

Coding Rule 22: If we choose to handle an exceptional case in a total way, deal with it in a return-clause in case of a derived inspector, and in a postcondition in case of a mutator or a constructor.

Coding Rule 23: Deal with all exceptional cases in the definition of a method, such that clients always know exactly how it behaves under all possible circumstances.

Coding Rule 24: Specify changes to the state of objects or classes involved in mutators and constructors in one or more clauses headed by the self-defined tag `@post`.

Coding Rule 25: The setter setα for a property a must deal with illegal values in a nominal way, in a total way or in a defensive way.

Coding Rule 26: Specify the effect of mutators and constructors in terms of other mutators, respectively constructors, if their effect must at all times be equivalent with the combined effect of the latter methods. Use one or more documentation clauses headed by the self-defined tag `@effect` for that purpose.

Coding Rule 27: Invoke class methods against their classes. Do not invoke class methods against objects of their class.

Coding Rule 28: In a group of elements related to a single property, definitions of methods precede declarations of variables.

Coding Rule 29: Choose names of variables in such a way that they reflect as well as possible the information they store. Use camel casing for their spelling.

Coding Rule 30: Encapsulate all technical details concerning the internal working of a class in its definition.

Coding Rule 31: Use private variables to encapsulate all aspects concerning the representation of classes and their objects.

Coding Rule 32: Explicitly initialize an instance variable or a class variable, if a proper initial value exists, even if that value coincides with the default value of its type.

Coding Rule 33: Add a final qualification to the declaration of all instance variables and class variables that serve to register immutable properties.

Coding Rule 34: Use all upper case in spelling identifiers naming symbolic constants. Separate successive words in such names by underscores.

Coding Rule 35: Use symbolic constants for timeless properties. Use immutable properties for values that are constant during a running application, but that may change over time.

Coding Rule 36: Use the conditional operators && and || to combine boolean subexpressions instead of the unconditional operators & and |.

Coding Rule 37: Prefer qualified selections of instance variables of objects over unqualified selections.

Coding Rule 38: Access non-final instance variables and class variables for storing single-valued properties only in the body of getters and setters. Initialize final instance variables in the body of constructors.

Coding Rule 39: Invoke instance methods against the prime object in an unqualified way, unless you want to emphasize that they are invoked against `this`.

Coding Rule 40: Do not access static variables or invoke static methods in a qualified way in the body of their class, unless their names conflict with names of other entities in nested scopes.

Coding Rule 41: Invoke a more extended constructor of the class in the body of a less extended constructor.

Coding Rule 42: Work out a separate test for each case you can distinguish in the specification of a method, if you opt for black box testing.

Coding Rule 43: Develop a JUnit test case for each class in a software system that will be tested.

Coding Rule 44: Complement a test case with an immutable test fixture to collect objects whose state does not change during the entire test case.

Coding Rule 45: Complement a test case with a mutable test fixture to collect objects whose state changes during at least one individual test.

Coding Rule 46: Name a test method after the method under test complemented with a brief description of the actual case the test aims at.

Coding Rule 47: Encapsulate the class invariant that applies to a class property α in a static checker named `isValid`α.

Coding Rule 48: Encapsulate the class invariant that applies to an object property β in a non-static checker named `canHaveAs`β if the conditions imposed on that property are object-dependent. Use a static checker named `isValid`β if the conditions are object-independent.

Coding Rule 49: Annotate an instance method `@Raw`, if it must be possible to invoke that method against an object that does not satisfy all its invariants.

Coding Rule 50: Annotate a constructor `@Raw`, if the constructor cannot guarantee that all the characteristics ascribed to the newly created object (including characteristics introduced at the level of subclasses) satisfy their invariants.

Coding Rule 51: Annotate formal arguments of class type `@Raw`, if it must be possible to supply actual objects not satisfying all their invariants.

Coding Rule 52; If you prefer to handle an exceptional case in a nominal way, deal with it in a precondition.

Coding Rule 53: Specify preconditions imposed on objects and values involved in a method in one or more clauses headed by the self-defined tag `@pre`.

Coding Rule 54: Do not add assert statements to dynamically verify the full contract of a method. Only use them to verify more complex preconditions imposed on methods.

Coding Rule 55: Use arrays to represent fixed-sized multi-valued properties of positionally ordered values.

Coding Rule 56: Complement a multi-valued property α represented in terms of an array with a basic inspector `getNb`α returning the number of elements collected by that property, and with an instance checker `canHaveAsNb`α`s` or a static checker `isValidNb`α`s` imposing restrictions on that number.

Coding Rule 57: Complement a multi-valued property α represented in terms of an array with (1) a basic inspector `get`α`At` returning the element at a given index, (2) a mutator `set`α`At` registering a given element at a given index if it must be possible to change individual elements, and (3) an instance checker `canHaveAs`α`At` or a static checker `isValid`α`At` imposing restrictions on the element registered at a given index.

Coding Rule 58: Number elements in sequential multi-valued properties starting from 1.

Coding Rule 59: Use while statements and do-while statements for dynamically bounded iterations. Use do-while statements in case at least one iteration is needed; use while statements in all other cases.

Coding Rule 60: Use for statements and enhanced for statements for statically bounded iterations. Use enhanced for statements to encode iterations over all elements of an iterable collection; use ordinary for statements in all other cases.

Coding Rule 61: Complement the implementation of complex iterative constructs with loop invariants and a loop variant.

Coding Rule 62: Introduce each self-defined exception class as a direct or indirect subclass of the predefined class of runtime exceptions.

Coding Rule 63: Terminate names of self-defined exception classes with the suffix `Exception`.

Coding Rule 64: Specify conditions under which methods must throw exceptions in clauses headed by the predefined tag `@throws`.

Coding Rule 65: Enumerate in the throws clause of a method all exceptions that can result from the execution of the method, and that directly or indirectly belong to the predefined exception class Exception, including unchecked exceptions of type `RuntimeException`.

Coding Rule 66: The set of exceptions enumerated in the throws clause of a method must be identical to the set of exceptions specified in its documentation.

Coding Rule 67: Isolate common aspects in conditions imposed on dependent properties $\alpha_1, \alpha_2, ..., \alpha_n$ in a boolean inspector `matches`$\alpha_1\alpha_2...\alpha_n$. Invoke that method in the checker for each property involving the value to be investigated and the current value of all other dependent properties.

Coding Rule 68: Annotate value classes `@Value` in their heading.

Coding Rule 69: Value classes will offer a series of symbolic constants initialized with frequently used values. Value classes will never offer a default constructor.

Coding Rule 70: Override the method equals inherited from the root class Object in each value class, such that objects with equal characteristics are considered equal.

Coding Rule 71: Override the method `hashCode` inherited from the root class `Object` in each value class, such that equal objects have identical hash codes.

Coding Rule 72: Override the method `toString` inherited from the root class `Object` in each class, such that it returns a human-readable textual representation.

Coding Rule 73: Do not override the method `clone` inherited from the root class `Object` in any value class.

Coding Rule 74: Ordinary classes, i.e. classes collecting true objects, will never override the methods `equals` nor `hashCode` they inherit from the root class `Object`.

Coding Rule 75: Make sure that each object is an instance of the same class for its entire lifetime.

Coding Rule 76: Complement a unidirectional association with restricted multiplicity in the definition of the referring class with a private instance variable referencing the associated object of the referred class.

Coding Rule 77: Complement a unidirectional association α with restricted multiplicity in the definition of the referring class with (1) a basic inspector `get`α returning the associated object, (2) a mutator `set`α registering a given object as the newly referred object, and (3) an instance checker `canHaveAs`α or a static checker `isValid`α imposing restrictions on the referenced object.

Coding Rule 78: Complement a bidirectional association with restricted multiplicity in the definition of both participating classes with a private instance variable referencing the associated object from the other class.

Coding Rule 79: Complement a bidirectional association α with restricted multiplicity in the definition of controlling class with (1) a basic inspector `get`α returning the associated object, (2) a mutator `set`α registering mutual references between its prime object and a given object, and (3) an instance checker `canHaveAs`α or a static checker `isValid`α imposing restrictions on the referenced object.

Coding Rule 80: Complement a bidirectional association α with restricted multiplicity in the definition of non-controlling class with (1) a basic inspector `get`α returning the associated object and (2) a auxiliary method `set`α

registering an association from its prime object to a given object.

Coding Rule 81: Encapsulate the class invariant for a bidirectional association with restricted multiplicity in a boolean inspector `hasProper`α, where α is the role of the other class in the association.

Coding Rule 82: Isolate all restrictions on bidirectional associations except for consistency of mutual references in a boolean inspector `canHaveAs`α, where α is the role of the other class in the association.

Coding Rule 83: If an association α with unrestricted multiplicity is presented as a list, introduce a basic inspector `getNb`α`s` returning the number of associated objects and a hidden setter `setNb`α`s` to change that number. Only introduce a checker if an upper bound must be imposed on the number of associated objects.

Coding Rule 84: If an association α with unrestricted multiplicity is presented as a list, introduce (1) a basic inspector `get`α`At` returning the associated object at a given index, (2) an instance checker `canHaveAs`α`At` or a static checker `isValid`α`At` imposing restrictions on the associated object registered at a given index, and (3) an instance checker `hasProper`α`s` encapsulating the class invariant.

Coding Rule 85: If an association a with unrestricted multiplicity is presented as a list, introduce mutators `add`α`At` and `remove`α`At` to add and remove elements. Often, simpler methods such as `addAs`α and `removeAs`α are sufficient.

Coding Rule 86: If a class is involved in an association with unrestricted multiplicity, introduce at least a constructor initializing new objects that are associated with the lowest possible number of objects of the other class

Coding Rule 87: If a class is involved in an association with unrestricted multiplicity, introduce at least a destructor that disconnects the object to be terminated from all objects it is still associated with.

Coding Rule 88: If an association a with unrestricted multiplicity is presented as a set, introduce a basic inspector `hasAs`α checking whether the given object can be included in the set of associated objects.

Coding Rule 89: If an association α with unrestricted multiplicity is presented as a set, introduce (1) an instance checker `canHaveAs`α or a static checker `isValid`α imposing restrictions on associated objects, and (2) an instance method `hasProper`α`s` encapsulating the class invariant

Coding Rule 90: If an association with unrestricted multiplicity α is presented as a set, introduce mutators `addAs`α and `removeAs`α to add and remove elements.

Coding Rule 91: Declare instance variables for storing references to lists, sets or maps of the interface type `List<T>`, `Set<T>` or `Map<K,V>`, and assign them instances of concrete classes implementing those interfaces.

Coding Rule 92: Access instance variables referencing data structures directly in the body of their class, and complement their declaration with representation invariants imposing restrictions on their contents

Coding Rule 93: Work out conditions imposed on bidirectional associations in the class controlling the association, and invoke that checker in the definition of the checker in the non-controlling class.

Coding Rule 94: Invoke the most extended constructor of the superclass in the most extended constructor of a subclass.

Coding Rule 95: Grant subclasses access to getters and setters defined at the level of their superclass by qualifying them at least protected.

Coding Rule 96: Do not introduce public constructors in abstract classes. Qualify them protected, package accessible or private. Add the annotation `@Model` to their definition if it must be possible to use them in specifications of public constructors.

Coding Rule 97: Work out a consistent set of tests for a method as part of the test suite for the class in which that method is defined or redefined.

Coding Rule 98: Develop class hierarchies in such a way that any extension with new subclasses has no impact at all on the definition of existing classes in the hierarchy.

Coding Rule 99: Use the annotation `@Override` consistently in overriding inherited instance methods.

Coding Rule 100: Use final qualifications only for methods whose specification and implementation has degenerated to return a constant value.

Coding Rule 101: Test suites for superclasses will verify the correctness of all methods defined at that level. Test suites for subclasses will only include tests for newly defined methods and for inherited methods they override.

Coding Rule 102: Objects of a subclass must honor all the class invariants imposed on objects of its superclass. A subclass may impose additional class invariants on its objects or strengthen class invariants it inherits from its superclass.

Coding Rule 103: If a subclass overrides an instance method, it must honor all the postconditions imposed on that method at the level of its superclass. A subclass may impose additional postconditions or strengthen inherited postconditions in overriding instance methods.

Coding Rule 104: If a subclass overrides an instance method, it must honor all the preconditions imposed on that method at the level of its superclass. A subclass may leave out inherited preconditions or it may weaken their

definition at the level of the overriding method.

Coding Rule 105: If a subclass overrides an instance inspector, it must honor the return clause imposed on it at the level of its superclass. A subclass may strengthen return clauses in overriding instance methods.

Coding Rule 106: Isolate differences between different types of objects in small methods that are overridden at the level of corresponding subclasses. Use these small methods in final definitions of more complex methods at the level of their superclass.

Coding Rule 107: Isolate conditions under which methods must achieve their regular effect in boolean inspectors. Override these so-called abstract preconditions at the level of subclasses if conditions must vary from type to type.

Coding Rule 108: Never extend the list of (unchecked) exceptions in overriding a method. Remove from the list all exceptions that can no longer be thrown, if the conditions under which they must be thrown at the level of the overriding method have reduced to false.

Coding Rule 109: If a subclass overrides an instance method, it must honor conditions under which that method must throw exceptions. A subclass is not allowed to extend nor to reduce conditions under which an inherited method must throw exceptions.

Coding Rule 110: Specify conditions under which methods throw exceptions in clauses headed by the predefined tag `@throws`.The formal specification starts with an expression under which the stated exception must be thrown, optionally followed by an expression under which the state exception can be thrown.

Coding Rule 111: If a subclass overrides an inspector, it must honor the return type as specified at the level of its superclass. A subclass may strengthen inherited return types to subtypes.

Coding Rule 112: If a subclass overrides an instance method, it must honor access rights imposed on that method at the level of its superclass. A subclass may widen access rights at the level of the overriding method.

Coding Rule 113: If a subclass overrides an instance method, it must honor raw annotations imposed on that method at the level of its superclass. A subclass may not leave out a raw annotation in overriding a method, nor may it add that annotation to the method itself or to one of its arguments.

Coding Rule 114: If a subclass overrides an instance inspector, it must honor immutable annotations imposed on that method at the level of its superclass. A subclass may not leave out that annotation in overriding a method. It may, however, impose that annotation to the overriding method.