International School

Capstone Project 1

CMU-SE 450

Code Standard

Version: 1.0

Date: 30/11/2021

School Connect Application

Submitted by:

Nguyen Thanh Phu

Nguyen Trung Hieu

Dang Nguyen Bao Hoai

Nguyen Van Thang

Approved by

Nguyen Minh Nhat

Code Standard Review Panel Representative:

Name	Signature	Date

Capstone Project 1- Mentor:

Name	Signature	Date

Project Information

Project acronym	SConA		
Project Title	School Connect Application		
Start Date	22-Aug-2021	**End Date**	18-Dec-2021
Lead Institution	International School, Duy Tan University		
Project Mentor & contact details	Name: Nguyen Minh Nhat Email: nhatnm2010@gmail.com Tel: +84 905 125 143		
Scrum Master & contact details	Name: Nguyen Thanh Phu Email: thangphu104@gmai.com Tel: +84 772 492 301		
Team members	**Name**	**Email**	**Tel**
	Nguyen Trung Hieu	hnguyentrung20@gmail.com	+84 975 299 149
	Dang Nguyen Bao Hoai	dangbhoai@gmail.com	+84 773 305 395

DOCUMENT INFORMATION			
Document Title	Project plan		
Author(s)	Team C1SE.44		
Role	C1SE.44-Code Standard -SConA-v1.0		
Date	30-Nov-2021	**Filename**	C1SE.44-Code Standard -SConA-v1.0
URL	**https://github.com/PhuNguyenThang/Capstone1**		
Access	Project and CMU Program		

DOCUMENT APPROVALS

The following signatures are required for approval of this document.

Nguyen Minh Nhat *Mentor*	Signature	Date
Nguyen Thanh Phu *Scrum Master*	Signature	Date
Nguyen Trung Hieu *Product owner*	Signature	Date
Dang Nguyen Bao Hoai *Product owner*	Signature	Date

REVISION HISTORY

Version	Date	Comments	Author	Approval
1.0	30-Nov-2021	Create Code Standard Document	Nguyen Thanh Phu	

TABLE OF CONTENTS

1 Introduction

1.1 Preface

This document has taken the Herschel coding standards [1]. This is essentially a self-standing document but it is heavily based on the Sun Java Guidelines.

1.2 Comments

For comments on this document please use the Gaia Wiki at the following url (all on one line):
http://www.rssd.esa.int/SA-general/Projects/GAIA/wiki/index.php?
title=Talk:CU1: Coding standard comments

1.3 Scope

This technical note defines a set of Java coding conventions and guidelines to be applied through-out the DPAC development. It addresses the following:

- stylistic conventions to produce easily readable, consistent code
- naming conventions
- conventions to achieve robust, maintainable code
- conventions to assist software portability
- Java design guidelines
- GUI design guidelines

1.4 Applicability

This standard applies to all Java software that will be run in any DPAC operational centers.

1.5 Purpose

The aim has been to capture, in a single document, a set of practical guidelines, based on common practice in the Java community. The guidelines are based on information from the following sources:

- Code Conventions for the Java Programming Language [2]
- How to Write Doc Comments for the Javadoc Tool [3]
- Writing Effective Java [4]
- ESA Java Coding Standards [5]
- Common practice in the Java community [6]
- Experience of team members [7]

In cases where the standards do not address a particular issue, the code should follow common Java usage. The Java class library provides a good source of examples.

1.6 Java Version

The Java Software Developer's Kit (SDK) consists of a programming language, a set of development tools, a runtime environment and a comprehensive class library. The Java language is defined in the Java Language Specification [6]. The suggested target platform is the 64-bit JDK 1.8 at least Java (TM) 2 Runtime Environment, Standard Edition (build 1.5.0 06-b05). Problems have been noted with large arrays in earlier versions.

2 Naming Conventions

The following guidelines are based on *Sun's Java Coding Conventions* [2], with additional ideas from *Effective Java* [4].

2.1General

CS-2-1 Source code should use the ASCII character set. Unicode characters are permissible in strings, using the Unicode escape character" *u*".

CS-2-2 The names of variables, methods, classes and interfaces should use US spelling (e.g., Color, initialize, Serializable). Rationale: Mixed US/UK spelling can result in methods being overloaded, when the intention is to override them. US spelling is the convention adopted by the JDK class library.

CS-2-3 Comments should be in English.

2.2 Packages

CS-2-4 Packages that form part of the Gaia data processing system shall have the following prefix, irrespective of where they are developed: gaia.

CS-2-5 Package names shall be all-lower-case ASCII.

CS-2-6 Package names should be short (e.g. util rather than utility), so that the fully qualified package name does not become excessively long.

2.3 Classes

CS-2-7 Class names are nouns, in mixed-case, with an initial upper-case letter and the first letter of each subsequent word capitalized (e.g., Core Factory).

CS-2-8 Class names should be self-explanatory and descriptive.

CS-2-9 Abbreviations and acronyms should be avoided in class names, unless these are more widely used than the full form (e.g., HTML). However, abbreviations and acronyms are acceptable if the resulting class name would be excessively long.

CS-2-10 Class names should not normally be longer than 32 characters, to avoid the need for excessive wrapping of source lines.

CS-2-11 Acronyms should have only the first letter capitalize ed. For example, CusUrlFactory instead of CUSURLFactory.

CS-2-12 The names of exception classes should end with Exception (e.g., GaiaException).

CS-2-13 Avoid using class names that appear in the JDK library, or within other Gaia sub-systems. Rationale: The use of fully-qualified class names i n the code, to resolve ambiguities, is inconvenient. Even if they can be disambiguated with import statements, it hinders clarity.

2.4 Interfaces

CS-2-14 The same rules apply as for classes, except that the names may be adjectives or nouns (e.g., Serializable or Source).

CS-2-15 The interface should have the generic name (e.g. Source), whereas implementations should have a suffix (e.g. SourceImpl) or a prefix (e.g. OraSource or Abstract-Source).

The rationale is that the client sees the interface, rather than the implementation. Also, there may be many implementations; a suitable prefix or suffix describes the particular type of implementation. Where there is only a single implementation, the suffix Impl is preferred (This abbreviation is acceptable as it is commonly used).

2.5 Methods

CS-2-16 Method names are verbs or nouns in mixed-case, starting with a lower-case letter (e.g. size or addElementary).

CS-2-17 Acronyms should have only the first letter capitalized ed (e.g. getHtml).

CS-2-18 The name is normally a verb if it performs some action (e.g. a mutator method). For example, schedule, addItem or setName. The name is normally a noun if it returns a non-boolean attribute or value (e.g. an accessor method). For example, length. Methods that return a boolean start with is, has, or similar (e.g. isValid, hasChild).

CS-2-19 Methods that return a value may take the form size or getSize. The get form is preferred if:

(a) the class is to be used as a Java" Bean",

(b) there is a corresponding set method, or

(c) the result is an attribute

CS-2-20 The following method naming conventions are preferred:

(a) Methods that convert the type, returning a new object, begin with to (e.g. toString).

(b) Methods that return a view are called asType (e.g. asList).

(c) Methods that return a primitive type xxx representation are called xxxValue (e.g. intValue).

(d) Methods that return a singleton instance are called getInstance.

(e) CS-2-21 Methods that return an iterator over type Foo, should be called fooIterator. In simple cases, such as a collection class, iterator may be sufficient.

Rationale: The prefix is useful where there is multiple one- to-many associations.

2.6 Variables

CS-2-22 The names of 'long-scoped' variables should be meaningful. Variable names for 'short-scoped' variables (i.e., index variables in short loops) need not be meaningful.

CS-2-23 The names of local variables should be in mixed case, starting with a lower-case letter (e.g. packetSize). This also applies to the formal parameters of methods. We recommend not to use names starting with underscore. This is against the Sun Java Code Conventions [2]. Moreover, in a language with compiler-enforced visibility constraints and smart editors that keep you advised of them, there is no reason for marking local variables with underscores.

CS-2-24 The names of static final constants should be all upper-case words, separated by underscores (e.g. MIN WIDTH).

CS-2-25 Avoid names that differ only in case. Never use in the same namespace identifiers that have the same letters, but that are capitalized in different ways. As an additional recommendation, generally avoid putting similar identifiers in the same namespace.

CS-2-26 Do not declare names in one scope that hide names declared in a wider scope. Pick different names as necessary.

2.7 Source Files

CS-2-27 The name of a source file should be the same as the name o f the public class or interface that it contains, with the suffix .java. For exampl e, the public interface AstroElementary should be in a file called AstroElementary. java.

2.8 Properties

CS-2-28 All packages should use the gaia.tools.PropertyLoader to load properties. This re-quires the main loop to call gaia.tools.ProprtyLoader.load();

CS-2-29 Java properties should have the prefix gaia.xxx, where xxx is the name of the java package where the property is used. For example, the subsystem gaia.tools.dal.Store might have the following value:
gaia.tools.dal.Store = gaia.tools.dal.FileStore

This convention may be extended hierarchically to sub-packages, if required.

Rationale: This naming convention gives each subsystem its own name-space and avoids clashes with properties used by other third-party libraries.

2.9 Example

The following example illustrates use of the naming conventions:

```
class FooBar {                                          // class name
    private static final int MAX_PERSONS = 100; // static constant
    private static int count;                           // static variable
    private String name;                                // instance variable

    void setName(String name) {                         // method 'setName'
                                                        // parameter 'name'
        this.name = name;
        int n = count++;                                // local variable 'n'
    }
}
```

2.10 Constants

CS-2-30 The gaia parameter database should be used for constants where possible.

3 Source Structure

The Java source layout should adopt the following conventions, which are based on the Sun Java Code Conventions [2]. Although some of the points may seem pedantic, experience shows that following a uniform style makes it easier for others to understand your code and vice versa.

3.1 Package Structure

CS-3-1 See Section 2 for package naming conventions.

CS-3-2 The Gaia processing software is organized into a set of Java packages, with the com-mon prefix:

gaia.

The next level in the hierarchy typically corresponds to algorithms and infrastructure.

e.g:

gaia.algo.gis.attitude
gaia.infra

In some cases, there is an intermediate package, to group together a set of related subsystems:

gaia.ora.infra
gaia.ora.update

In many instances we have interfaces and implementations these should preferably be split in two packages thus:

gaia.dm
gaia.dmimpl

CS-3-3 The directory structure of the source tree must correspond exactly with that of the package tree. For example, the file:

gaia/foo/bar/MyClass.java

contains the Java source for the class:

gaia.foo.bar.MyClass

CS-3-4 Each package must have a JavaDoc package-info.java file to describe the package (see Section 5 for further details).

3.2 File Structure

CS-3-5 Each Java source file may contain only one public class or interface [1] The file may also contain non-public classes and interfaces, which are only used by the public class. In this case, it is usually better to make them nested classes and interfaces.

CS-3-6 The source file should be organized in the following or der:

- beginning comment (see section on comments)
- package statement
- import statements
- public class or interface
- associated non-public classes and interfaces

CS-3-7 Each class should be organized in the following order:

- Javadoc class comment (including @author and @version tag for the public class)
- class statement
- constant and variable declarations
- Constructors (including clone)
- Methods

CS-3-8 Constant and variable declarations should be in the following order:

- static constants
- static variables
- instance variables

Within each of these groups, the declarations should be in the following order:

- public
- protected
- default
- private

CS-3-9 Methods should be grouped by functionality rather than access rights.

CS-3-10 Each method should be preceded by a Javadoc comment.

CS-3-11 When multiple modifiers are used together, they shall l appear in the following order:

- public
- static
- abstract
- synchronized
- final

3.3 Indentation and Layout

CS-3-12 Source layout should follow the following rules:

- Indentation should be 4 spaces long and should not be done using tabs.
- Avoid lines longer than 80 characters.
- Use a space before '(' for statements but not methods.
- Follow commas in argument lists with a space.

CS-3-13 Use Sun's preferred line-wrapping style (see [2] for details):

- break after a comma
- break before an operator
- prefer higher-level breaks
- align with expression at same level on previous line

CS-3-14 Use blank lines as follows:

- 1 between methods, before (block or single line) comment
- 1 between logical sections of a method
- 2 between sections of a source file

Rationale: Without the braces, it is easy to make a mistake by adding an additional statement at the nested indent level, expecting it to be executed as part of the block.

CS-3-17 Only one declaration should appear on each line for both public and private attributes.

CS-3-18 Avoid multiple assignments, e.g. x = y = 42;

CS-3-19 Avoid embedded assignments, e.g. x = y + (z = getNext());

CS-3-20 Use parentheses to make precedence clear, with mixed operators.

CS-3-21 Omit parentheses around return argument, unless needed for clarity.

3.4 Comments

CS-3-22 All Java source code shall include JavaDoc comments. Section 5 of this document gives specific guidelines on the use of JavaDoc.

CS-3-23 The beginning comment at the start of each source file shall contain a heading with the institutes name and/or copyright and licensing model, e.g.:

```
/*------------------------------------------------------------
 *
 *              Gaia Science Operations Centre
 *              European Space Astronomy Centre
 *
 *              (c) 2005-2020 European Space Agency
 *
 *------------------------------------------------------------
 */
```

Note: This is not a JavaDoc comment (i.e., it uses '/*' instead of '/**').

CS-3-24 The JavaDoc comment on the class should contain the author of the class and SVN id using the JavaDoc tags @author and @version, e.g.

@author John Smith
@version Id

The Id field will be filled in automatically when the code is fi rst submitted to SVN and will be updated at each change.

CS-3-25 Use 'TODO' in a comment where something is broken. Use 'XXX' in a comment to flag something that is bogus but works.

3.5 Import Statements

CS-3-26 Import statements may NOT use the * wild cards for any package. Tools such as Eclipse will organize all imports explicitly. Explicit imports of each class lead to much less confusion later when looking at code e.g., this makes the dependencies between subsystems clearer.

4 Libraries and Portability

This section contains rules on the use of third-party libraries. It also gives guidelines on software portability.

4.1 Class Libraries

CS-4-1 Only approved third-party libraries and packages may be used[2]. These will be included in the gaia common library or agreed with the Data Processing Centre.

Rationale: Uncontrolled use of third-party libraries can lead to problems if those packages:

- are not maintained throughout the life of the project
- have licensing / legal restrictions
- are of poor quality (not fit for purpose)
- have overlapping functionality

CS-4-2 Programs may use any classes from the Java2 SDK Standard Edition library. Sun has structured Java as a set of standard classes, plus a number of extension libraries (e.g. Java-3D). There is no problem, in principle, in using any of these Sun extensions, but they must first be approved.

CS-4-3 Java Enterprise Edition may be used where appropriate.

CS-4-4 Classes from the com.sun package should not be used. Rationale: This is not part of the supported API and may not exist in other implementations of the Java platform.

4.2 Platform Portability

Platform independence is one of the significant advantages of the Java programming language. The end platform for Gaia processing may not be known for some years and may be heterogeneous, it is desirable to write portable code. Such code is more likely to survive the evolution of a single platform (such as Solaris), given the lifetime of the mission.

The steps described below will be taken to maximize the portability of the code.

CS-4-5 Java code should not make operating-specific assumption ions. Java provides appropriate mechanisms for abstracting such features.

CS-4-6 Don’t embed newlines in strings, unless you are sure it is portable.

CS-4-7 Programs should not rely on vendor-specific features or behaviour of a JDK imple-mentation (e.g., Sun or IBM). For example, garbage collection strategies may be different.

CS-4-8 The Java code shall not include hard-coded file paths that are specific to a particular site or host computer. In general, such configuration inform action should be obtained from Java property files. These may, in turn, specify the location of files, databases, etc.

CS-4-9 JNI (Java Native Interface) may not be used. Some exceptions might be approved, but should be cleanly decoupled, perhaps within a separate server process.

4.3 Internationalization

There is no requirement to support internationalization of languages (i.e. switching the language used for menus, messages, on-line help, etc).

CS-4-10 It should not be assumed that the Java locale is set to any particular region. For example, the default locale should not be relied upon for formatting of dates, etc, when a specific format is required (e.g. for exported files).

5 JavaDoc

The following guidelines explain how JavaDoc should be used on the project. They are based on Sun's *How to write doc comments for JavaDoc* [3] . Reference should be made to the original document for further information on JavaDoc.

5.1 Introduction

The aim, when writing JavaDoc, is to produce an API specific at ion that should ideally be sufficient to write a compliant implementation [3]. The JavaDoc should therefore be implementation independent.

The JavaDoc tool automatically generates API documentation for public and protected classes and their methods, from special comments in the source code. Additional package-level documentation is written in HTML, to explain the overall structure and purpose of the package. It is possible to link in additional HTML files, where more extended d documentation is required.

Because it is an API specification, the JavaDoc should not contain information that does not form part of the specification. However, the package documentation may contain links to related documentation, such as user guides, design notes, tutorials (see below).

5.2 General

CS-5-1 Java source code shall include doc comments so that documentation may be generated automatically using the JavaDoc tool.

CS-5-2 Write doc comments as an implementation-independent API specification. Make it clear in what ways conforming implementations are allowed to differ.

Note: It should be clearly indicated when implementation-specific c remarks are made.

CS-5-3 JavaDoc comments are required for all public and protected classes, methods, inter-faces and fields, which form part of the package's API.

CS-5-4 For uniformity, private and package-scope classes, interfaces and methods, should have comments in the JavaDoc style, but the use of JavaDoc"@" tags is not re-quired. Rationale: JavaDoc comments are primarily for the public API, but its help-ful to adopt a uniform style. It is not necessary to document the methods of simple anonymous inner classes.

CS-5-5 The first sentence of each JavaDoc comment should be a complete concise description of the item.

The JavaDoc tool copies these sentences to the appropriate summary section. They must there-fore be able to stand on their own.

Note: See the Sun guide [3] for a way of embedding a period in the sentence.

5.3 Tags

The usage of tags is described in the Sun JavaDoc guidelines [3].

CS-5-6 Tags should appear in the following order:

@author
@version
@param
@return
@throws
@see
@since
@serial
@deprecated

CS-5-7 The comment preceding a class shall contain the following after the textual description (where John Smith is the author's name):

@author John Smith
@version Id

If a source file contains more than one class, this applies to the class which has the same name as the source file (without the .java extension).

CS-5-8 Mandatory tags:

- @author for the class/interface with the same name as file
- @param include units where appropriate
- @return (if non-void) include units where appropriate
- @throws for all checked and unchecked exceptions

Note: @throws is synonymous with @exception. The former should be used through-out, for consistency and because it is more concise

5.4 HTML

CS-5-9 JavaDoc comments are in HTML and must use appropriate HTML tags for format-ting.

Note: Blank lines in the source comment do not lead to blank lines in the formatted JavaDoc, unless the <p> or the <br> tags are used. Blank lines

should be included, so that the comments are easy to read both in the Java source fil e and in the formatted HTML.

Other commonly used tags include:

- <code>..</code> for names of classes, methods, etc
- <pre>..</pre> tag for code examples
- ..
.. to format lists

The source code of the JDK library is a good source of examples.

5.5 Exceptions

CS-5-10 Use the JavaDoc @throws tag to document checked and unchecked exceptions. Unchecked exceptions are used for conditions that violate a precondition. The @throws tag may therefore be used to document (the complement of) preconditions.

CS-5-11 See the Exception and Error Handling section of this document for further details on exceptions.

5.6 Package Documentation

CS-5-12 Each Java package should have a package-info.java fi le that provides overall documentation for the package. This provides top-level documentation for the package as a whole, whereas the JavaDoc comments document individual classes. This is now the preferred way of doing package level documentation[4].

CS-5-13 The following is an example of a package-info.java file :

/**

The data access layer consists of two methods for accessing data. The Store interface always returns tabular data in the form of GaiaTable while the Object Factory provides Objects implementing the Data Model interfaces.

<img src="doc-files/package overview.jpg"> **/

package gaia.tools.dal.jdbc;

Here the <img> HTML tag is used to include a jpg UML view of the package itself. This is not mandatory. The description of the package should possibly contain the following information:

- a summary sentence describing the package;

- a description of the package;
- additional parts of the API specification that are not included in the doc comments (or links to the specification).
- related documentation
- references to documents that do not form part of the specification (e.g. tutorials, user guides).

5.7 Images and Diagrams

CS-5-14 See the Sun guidelines [3] for information on including images, diagrams, etc in the JavaDoc.

6 Java Design Guidelines

The following guidelines are aimed at producing robust, easily-maintainable code. The book Effective Java [4] is recommended for further reading.

6.1 Visibility

The public API of a package is a contract between the package developer and package user. It implies a commitment, on the part of the developer, to keep the API stable.

CS-6-1 Ideally, classes, methods, etc, should not be made public unless they are an official part of the supported API. In practice, this is not always possible because Java does not allow subsystems to have local packages that are only visible to that subsystem.

Where possible, minimize accessibility by exposing only what is an official part of the API. In particular:

- Classes should only be public/protected where they are part of the package API.
- All attributes should be private where possible.
- Consider using Java interfaces, instead of classes, for the public API. This makes it easier to change the implementation of the package.
- Don't provide get/set methods for every field, unless there i s a reason.
- If a class has to be public but it is not intended for general use, then exclude it from the JavaDoc and/or add a suitable comment, such as the following:" Warning: This class does not form part of the supported public API of this package."

CS-6-2 **Sub-classing** Extending a class can led to some subtle problems. For example, changes to the implementation of a class can break sub-classes, even though the API is unchanged. Making a class protected can break the encapsulation, unless the class is carefully designed to avoid such problems (see [4] for examples). Sub-classing within a package is alright, because the same developer is responsible for all the classes. Avoid protected classes and

attributes (it exposes the implementation). Prefer composition to sub-classing, if the functionality of the class is to be extended outside the package, unless the class is properly designed (and documented) for sub-classing. Declare all public classes final unless they are designed to be sub-classed. Protected classes must document the effects of overriding methods. Constructors must not invoke overridable methods of the class.

CS-6-3 **Serialization** Making a class Serializable breaks encapsulation by exposing its inter-nal fields. This problem can be addressed as follows: Only imp lement Serializable if really needed. Use a custom serialized form to separate the logical and physical structure. Unless the serialized format is guaranteed to be stable, include a warning, such as the following, in the JavaDoc:" Warning: Serialized objects of this class may not be compatible with future releases. The current serialization support is appropriate for short term storage or RMI between applications running the same version of the software."

CS-6-4 Non-instantiable classes, with only static methods, should have a dummy private constructor.

CS-6-5 Make defensive copies of arguments or return values where necessary. If a mutable object field is set or returned by a method, the caller can change it and may violate the class invariants.

6.2 Inheritance

CS-6-6 See 'Sub-classing' in the above section on Visibility.

Inheritance should satisfy the well-known Liskov Substitution Principle. That
CS-6-7 is, an
instance of the subclass should always be usable where an instance of the superclass
can be used.

Note: this applies to the behaviour of the class and not just its type signature. In particular:

- The subclass may not strengthen the preconditions of its superclass.
- The subclass may not weaken the postconditions of its superclass.
- The subclass should not restrict the visibility of methods or attributes of its superclass.

CS-6-8 A non-abstract method should not be overridden by an abstract method in a subclass.

CS-6-9 Attributes declared in child classes should not hide (i.e., have the same name as) inherited attributes.

CS-6-10 Child classes should not hide inherited static methods. Rationale: Redefining a static method in a subclass is confusing, as static methods cannot be overridden.

CS-6-11 A superclass method should not be overloaded by a subclass, unless all overloading's in the superclass are also overridden in the subclass. Rationale: When a method is overridden by a subclass, it is usually required that invocations for all argument types are overridden. Overriding only some of the superclass' overloaded methods is usually an error, but may happen accidentally if a method is overloaded, when the intention is to override it.

Note: It is possible to violate this rule by adding a new overloading to a superclass, without changing the subclass. This illustrates one reason why is dangerous to allow classes to be extended outside the package in which they are defined (See 'Sub-classing' in the above section on Visibility).

CS-6-12 A subclass should not contain a method with the same name and signature as a superclass if these methods are declared to be private.

Rationale: Redefinition in a subclass is confusing as privat e methods cannot be over-ridden to provide polymorphism.

6.3 Packages / Subsystem

CS-6-14 The API should be clearly identifiable (Java visibil ity rules are not hierarchical). The API typically consists of interfaces, factories, exceptions and facade classes. Keep these together.

CS-6-15 Use a class comment to say where public interfaces and classes are not part of the API.

CS-6-16 Subsystems may not have mutual (bi-directional) dependencies.

6.4 Equality

CS-6-17 Consider whether a class should have an equal's method: equals is typically needed for 'value' classes. Is the default (reference equality) correct?

CS-6-18 The equals method must adhere to the contract specified in the documentation of java.lang.Object:

- equals must be reflexive, symmetric, transitive and consistent;
- x.equals(null) must return false.

CS-6-19 Throw ClassCastException if the type of argument cannot be compared.

CS-6-20 Be aware that it is not possible to extend an instantiable class and add a new attribute, whilst preserving the contract for equals (see [4]).

CS-6-21 The hash Code method must be implemented when equals is implemented.

CS-6-22 Be careful comparing float/double values involving -0.0f and NaN, e.g. 0.0 == -0.0 and NaN != NaN

CS-6-23 Should Comparable be implemented (Does the class have a natural ordering)?

- sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
- compareTo must be transitive.
- x.compareTo(y) == 0
  ⇒ ∀z sgn(x.compareTo(z)) == sgn(y.compareTo(z))
- if (x.compareTo(y) == 0) != x.equals(y), document that the ”class has a natural ordering that is inconsistent with equals”.
- There is a problem if an instantiable superclass also implements compareTo.

6.5 toString

CS-6-24 Most top-level classes should implement the toString method.
toString should appear in interfaces definitions, even though it is in herited
CS-6-25 from Object. This allows it to be documented (see the following items).

CS-6-26 The documentation of the toString method should normally state that the format is subject to change. Applications should not rely on the format of the string returned by toString.
CS-6-27 For simple value classes, it may be appropriate to define the format in the documen- tation, so that applications can rely on it. In such cases, it may be useful to have a constructor that parses the same representation.

6.6 Immutability

CS-6-28 Consider whether classes should be immutable. This is often appropriate for a 'value' class (e.g. Complex number).

CS-6-29 Immutability should be enforced. Does the API expose mutable fields?

CS-6-30 Don't provide clone or a copy constructor for an immutable class.

6.7 Persistence

CS-6-31 Do not use any Database API directly in client applications. Instead, use the interfaces in the package gaia.tools.dal. Implementations of these interfaces may then use Database specific calls.

CS-6-32 All data classes should be defined in terms of interfaces which all clients use. Implementations of these classes may be specific to one persist ence type. For example consider gaia.tools.dal.GaiaTable which has implementations for Oracle, JDBC, FITS and ASCII.

6.8 Loggin

CS-6-33 Use the interface gaia.tools.util.Logger for logging.

CS-6-35 Create a logger in each class as follows:

```
private static Logger logger = GaiaFactory.getLogger(MyClass.class);
```

CS-6-36 Use logger.trace for all cases where some information is required. The logger allows tracing to be turned on per class and is a very small overhead.

CS-6-37 Always log caught exceptions.

6.9 Enumerations

Enumerations were added in Version 1.5 of the language and usage is encouraged as it leads to cleaner code.

CS-6-38 Use enums and associated functionality instead of integer constants whenever possi-ble, e.g.

```
private enum Color { RED, BLUE, YELLOW
}; int k = Color.RED.ordinal(); String s =
Color.RED.toString();
Color c = Color.values()[k];
```

6.10 Foreach loop

The foreach loop construct was added in Version 1.5 of the language and usage is encouraged.

CS-6-39 Use the foreach looping construct for iterating over collections and arrays whenever feasible, e.g.

```
ArrayList<Double> list = new ArrayList<Double>(); ...

// iterate over all array elements for
(Double d : list)
// use element d here
    ...
```

6.11 Singleton Classes

Singleton classes are sometimes useful, for example for a logger that must be accessed from anywhere within a program. However, excessive use should be avoided. Singleton classes can lead to problems, for example, because they are not reinitialized when running a sequence of JUnit test harnesses.

CS-6-40 Avoid excessive use of singleton classes.

CS-6-41 Consider using a factory to create single instances of a class rather than using Singletons directly. Define the interface separately and use th e Interface in all client code.

CS-6-42 Enforce singleton classes with a private constructor.

6.12 Miscellaneous

CS-6-43 Don’t access static methods via an object.

CS-6-44 Numerical constants should not be coded in-line, except small values (e.g. 1,0,-1).

CS-6-45 Constants should be declared static final.

CS-6-46 Avoid finalizers (except in very special situations). It cannot guarantee when (or even if) they will be executed. This can lead to very obscure platform-specific bugs.

CS-6-47 Do not rely on the use of delays to avoid race conditions. They may not the same on every platform, under every condition.

CS-6-48 Use StringBuilder for repeated concatenation of strings. It is more efficient than concatenation with '+'. For code which will not be executed often '+' may be acceptable.

CS-6-49 Refer to objects by their interfaces, not their implementation classes. e.g. Declare a variable as a List, not a Vector.

CS-6-50 Use generics for all List and Map implementations i.e. List<AstroElementary> rather than simply List. Also use them instead of casting when navigating through collections.

CS-6-51 If a class is too big or complicated, consider how to split it up based on responsibili-ties (i.e. separation of concerns).

CS-6-52 Accessor (e.g. get) methods should not change the logical state of an object.

Note: The physical state may change, for example in the case of an object that uses internal caching to improve performance. However, the logical state, provided by the abstraction, should not change. If the object is Comparable, it should be equal before and after the get operation.

CS-6-53 It is a good idea to make a distinction between accessor and mutator methods. How-ever, it is sometimes appropriate for a mutator to return a value, such as:

- this
- the object added or removed by the method
- the next value for an iterator
- a status values

CS-6-54 Avoid having overloaded methods (or constructors) that can match the same call, if they have different visibility.

Rationale: Different methods may be invoked if the same call is made from within and outside the package. This is confusing.

CS-6-55 Multiple declarations with the same name should not be visible simultaneously, ex-cept for overloaded methods. For example, it is confusing to have a method and a variable with the same name visible in the same scope.

7 Test Harnesses

Each package that contains code should have a test harness. Where possible, these should pro-vide a go/no-go indication, so that the system may be tested automatically as part of the build procedure.

7.1 General

CS-7-56 Each sub-system should have a test harness in the form of JUnit test cases.

CS-7-57 Tests must be repeatable (i.e. initialize random number generators to known seed).

CS-7-58 Test data must be platform independent (i.e. no binary files).

CS-7-59 Tests must be robust. Any compliant implementation should pass the test, even if it gives slightly different results due to rounding errors. Tests involving floating-point

equality should normally allow a very small difference from the expected value - JUnit assert Equals allows for this.

CS-7-60 Test harnesses should cover both normal operation (e.g. derived from use-cases) and abnormal conditions.

Note: Java exceptions allow a single test harness to contain multiple tests that involve abnormal conditions.

CS-7-61 Where the API and implementation are provided by separate packages, the test har-nesses belong with the API. They are a test of any compliant implementation. They can be thought of as part of the semantic specification of the A PI. It is also possible to have white-box test harnesses associated with a specific implementation. Packages should have a test harness that tests the package at the level of its public interfaces (i.e. black box tests). Individual classes may also have test harnesses, which test their public interfaces (black box) and/or private classes and methods (white box).

CS-7-62 The developer should identify all requirements to be implemented by the package or sub-system under test (use-cases, performance, etc) and design the test harnesses to ensure that the code fulfills those requirements.

CS-7-63 Long running Integration level tests should be named XXXIntegrationTest.

7.2 Unit Tests

CS-7-64 Unit tests shall be implemented using the JUnit framework.

CS-7-65 Black-box testing has to be preferred against white-box testing[5].

CS-7-66 Black-box test harnesses may be placed in 'test' sub-directories of the packages being tested.

CS-7-67 The classes that implement test cases must have names that end with Test. For example, the test harness for class Foo would be called FooTest.

7.3 GUI Tests

Graphical User Interfaces (GUIs) may require user interaction to test them. The following guide-lines are designed to make this as easy as possible.

CS-7-68 GUIs tests shall be implemented using the JFCUnit framework.

CS-7-69 The design of a GUI should attempt to cleanly decouple presentation and user inter-action from the underlying functionality, so that the two can be tested independently. For example, a mission planning subsystem might have a package that implements a scheduler and another package that provides the GUI. The scheduler can then be tested in isolation using an automatic (JUnit) test harness. The GUI is tested manually, but the test only involves checking the interactive aspects and not the underlying scheduling functionality.

CS-7-70 GUI components, which are used as building blocks, should have a manual test pro-gram so that they can be tested in isolation. For example, a set of widgets for entering numbers, with range checking, should have a simple test program that allows the widgets to be exercised and tested on their own. When these widgets are used in an application, the test harness of the application can assume that the widgets work correctly.

8 Exception and Error Handling

The following guidelines cover the handling of errors and exceptions within Java programs.

8.1 Exceptions

CS-8-1 No System. Exit calls should be made, except in the top-level class that contains the ”main” method. Other classes should simply throw an exception.

CS-8-2 Don't use exceptions to handle normal control flow. Th ey should only be used for exceptional conditions.

CS-8-3 The various types of Throwable should be used for the following purposes:

- (a) Checked exceptions should be used for conditions from which the caller might reasonably attempt to recover. This may include cases where the data might assume unexpected values. In such cases the presence of a checked exception will inform the user that something can be done in this case to prevent the whole process from crashing, for instance skipping this set of data and proceed with the next chunk.

- (b) Unchecked exceptions should be used for conditions that indicate a precondition violation (e.g. NullPointerException). These are not intended to be caught by the caller.
- (c) Errors should be reserved for use by the Java Virtual Machine (JVM) or to indicate invariant failures (e.g. thrown by an assertion).

Note: Some third-party libraries throw an unchecked exception that the user may need to catch.

CS-8-4 Don't catch an Error. Errors normally indicate a problem in the Java Virtual Machine, which is not recoverable.

CS-8-5 Don't throw an Error. These are normally reserved for the JVM and assertion mechanism.

CS-8-6 Throw unchecked exceptions to indicate a pre-condition violation. This is typi-cally an IllegalArgumentException, IllegalStateException, NullPointerException or IndexOutOfBoundsException.

CS-8-7 Use the JavaDoc @throws tag to document each exception that the method throws, in the form (see example below for more details):

```
@throws IllegalStateException if setSample() has not been called
  yet
```

Note: It is not possible to know all the exceptions that a method may throw, since it may propagate unchecked exceptions from other methods that it calls. However, the JavaDoc should document all the checked exceptions and all the unchecked exceptions that the caller might reasonably want to catch.

CS-8-8 Declare unchecked exceptions in the method signature. This should include all the unchecked exceptions that the caller might reasonably want to catch.

Note: Declaring the exception does not make it a checked exception; this is deter-mined solely by inheritance from RuntimeException.

CS-8-9 Use exception translation so that a class throws an exception appropriate to the abstraction that it provides. For example, different implementations of an interface might use different data structures, which throw different exceptions in similar situations. An appropriate exception should be chosen, when designing the interface (i.e. API), which matches the conceptual abstraction. Implementations may then catch and re-throw the underlying exception, to translate it to the correct exception for the API.

CS-8-10 Failures should be atomic. Either test before changing the state of the object (e.g. precondition) or repair the object afterwards. Otherwise, document the behavior.

CS-8-11 If a method's preconditions are satisfied, it should either return a valid result (that satisfies its postconditions if present) or it should throw a n exception specified by an @throws tag. See example below.

CS-8-12 Ensure invariants are satisfied by all constructors and methods. Constructors should create fully initialized objects which satisfy their class invariants.

8.2 Examples

The following is an example of the occurrences of both a checked and an unchecked exception in one method. This is the getValue(double xp) method of the class LinearInterpolator (in package gaia.tools.numeric.interpolation contained in the library GaiaTools). The method returns the linear interpolation at xp of the sampled function previously set by calling setSample(). Two things can happen:

- either the sampled function has not been set yet and thus the interpolation cannot be carried on;
- or the xp value at which the interpolation is required is out of the range of the sample function.

In the first case, the call to the getValue() method will always fail. This is clearly a program-ming error and should throw an unchecked exception.

However, in the second case the problem might be due to the specific value assumed by xp. If we imagine applying this method to several sets of data, it might be that for one set the value of xp is out of range for some reason. The user obviously wants to know and be warned about this but probably does not want the program to crash and may instead decide to go on with the next set of data logging the exception message. In this case thus it is correct to throw a checked exception.

The resulting code will look as follows:

```
/**
 * If x[] and y[] have been set, this method will look for the
 * optimal interval and will calculate the linear interpolation.
 *
```

```
     @para
     * m        xp
     *              point  where the function  has to be interpolated.
     *
     * @return f(xp) estimated double      value  at  xp
     * @throwsIllegalStateException if setSample() has not been called
     *              yet
     @throw
     * s         GaiaInvalidDataException if xp lies out of the range where
     *              the sampled functionhas   been  defined
     */
   public double getValue(double          xp ) throws IllegalStateExcep tion,
            GaiaInvalidDataException {

        if (!existSample) {
            //  setSample() has not been called yet
            throw new IllegalStateException
            (
                 "No valid sampled function set!");
        }

        if ( xp < x[0] || xp > x[ size - 1]) {
            //  xp lies out of the range where the sampled function has
            //  been defined
            throw  new GaiaInvalidDataException(
                 " Input  value  out of bounds ! " );
         }
    ...
}
```

9 Graphical User Interfaces

9.1 Design

CS-9-1 Where possible, the GUI should be structured so that the user interface is cleanly de-coupled from the underlying functionality. This makes it possible to write automatic test harnesses for the functional aspects. (See section on testing.)

CS-9-2 Java GUIs should use the Swing components in preference to the older AWT com-ponents, where possible. For example, javax.swing.JTextField should be

used in preference to java.awt.TextField. This may require a suitable browser plug-in for applets.

CS-9-3 Don't extend one of the window classes (e.g. JFrame) if you want your component to be usable in more than one circumstance. Subclass JComponent or JPanel instead and let the application decide whether to put it in another panel, a frame, internal frame, dialog, applet or whatever.

CS-9-4 If the previous two points are adhered to, it allows the look and feel to be completely pluggable, in two ways: between the native platform L&F and the cross-platform (Metal) L&F between independent windows and internal frames of a JDesktopPane

CS-9-5 Don't declare the component classes to be final as this will prevent it from being customized and consequently it will not be reusable.

CS-9-6 Provide a public no-argument constructor. This is a standard Swing convention and will allow it to be used with automatic tools.

CS-9-7 Perform operations that take a significant amount of time (e.g. ¿ 1 sec) in a separate thread so that the user remains in control.

9.2 Constructors and Object Lifecycle

Constructors are needed to create new objects and initialize them to a valid state. Constructors have a large impact on performance. Good coding rules, especially to minimize their use, are important for well performing applications.

CS-9-8 A class shall define at least one constructor. Rationale: The default constructor, i.e., the one with no arguments, is automatically provided by the compiler if no other constructors are explicitly declared. The default constructor supplied by the compiler will initialize data members to null or equivalent values. For many classes, this may not be an acceptable behavior.

CS-9-9 Avoid creating unnecessary objects. Rationale: Because of the complex memory management operations it involves, object creation is an expensive process. Creating an object not only implies allocating its memory, but also taking care of releasing it later. Since the Java programming language provides an automatic garbage collector that operates transparently, Java programmers often forget that garbage collection cycles are expensive, and that they can seriously increase the overall load of a system.

CS-9-10 Consider alternatives to the use of new when creating Objects. Rationale: From an object-oriented design point of view, the new keyword has two serious disadvantages:

- It is not polymorphic. Using new implies referencing a concrete class explicitly by name. New cannot deal with situations where objects of different classes must be created depending on dynamic conditions.
- It fails to encapsulate object creation. Situations where objects could be created only optionally cannot be properly handled with new. Properly handling these drawbacks is, however, possible. A number of design patterns, like static fac-tory methods, abstract factories, and prototypes [7] can be used to encapsulate object creation to make it more flexible and scalable.

Critical systems have special rules for memory allocation, that may supersede this recommendation.

CS-9-11 Consider the use of static factory methods instead of constructors. Rationale: static factory methods are static class methods that return new objects of the class. Among their advantages, with respect to constructors, are:

- They can have descriptive names. This is particularly useful when you need to have a variety of different constructors that differ only in the parameter list.
- They do not need to create a new object every time they are invoked (that is, they encapsulate object creation).
- They can return an object of any subtype of their return type (that is, they are polymorphic).

CS-9-12 Consider the use of lazy initialization. Do not build something until you need it. If an object may not be needed during the normal course of the program execution, then do not build the object until it is required. Use an accessor method to gain access to the object. All users of that object, including within the same class, must use the accessor to get a reference to the object.

9.3 Look and Feel

CS-9-13 Java GUIs should follow the Sun Java Look and Feel Design Guidelines [8].

CS-9-14 The choice of native or cross-platform Look and Feel should be controlled by a system-wide configuration parameter.

CS-9-15 Provide keyboard shortcuts for common operations that are awkward with the GUI. The same shortcut keys should be used for the same sort of operations across the system.

CS-9-16 It's a good idea to use tool tips liberally. However, some users don't like them and they can get annoying, so provide a means of turning them off. You can do this with:

ToolTipManager.sharedInstance().setEnabled(false);

Note: that a number of Swing classes (including JLabel, JButton, JToolTip) support HTML formatting. This can be used to improve appearance and readability.

CS-9-17 It is a good idea to make the font size of text edit panes configurable. This makes it easier to give demonstrations of the software (e.g. using a projector).

9.4 Testing

CS-9-18 See ”Gui Tests” in the section on Test Harnesses.

CS-9-19 Ensure that resizing a component with the mouse has the desired effect.

CS-9-20 Ensure that the focus is set to where the user expects it.

10 VueJS

10.1 Always use:key inside v-for

Using the key attribute with the v-for directive helps your application be constant and predictable whenever you want to manipulate the data.
This is necessary so that Vue can track your component state as well as have a constant reference to your different elements. An example where keys are extremely useful is when using animations or Vue transitions.
Without keys, Vue will just try to make the DOM has efficient as possible. This may mean elements in the v-for may appear out of order or their behavior will be less predictable. If we have a unique key reference to each element, then we can better predict how exactly our Vue application will handle DOM manipulation.

```
<template>
  <!-- BAD -->
        <div v-for="product in products">{{ product }}</div>

  <!-- GOOD! -->
        <div v-for="product in products" :key="product.id">{{ product }}</div>
</template>
```

10.2 Use kebab-case for events

When it comes to emitting custom events, it's always best to use kebab-case. This is because in the parent component, that's the same syntax we use to listen to that event.
So for consistency across our components, and to make your code more readable, stick to using kebab-case in both places.

PopupWindow.vue

```
this.$emit("close-window");
```

ParentComponent.vue

```
<popup-window @close-window='handleEvent()' />
```

10.3 Declare props with camelCase and use kebab-case in templates

This best practice simply just follows the conventions for each language. In Javascript, camelCase is the standard and in HTML, it's kebab-case Therefore, we use them accordingly. VueJS converts between kebab-case and camelCase for us so we don't have to worry about anything besides actually declaring them.
In Javascript, camelCase is the standard and in HTML, it's kebab-case Therefore, we use them accordingly.

BAD!

```
<template>
        <PopupWindow titleText="hello world" />
</template>
<script>
        export default {
                props: {
                        'title-text': String
                }
         }
</script>
```

GOOD!

```
<template>
        <PopupWindow title-text="hello world" />
</template>

<script>
        export default {
                props: {
                        titleText: String
                }
        }
</script>
```

10.4 Data should always return a function

When declaring component data, the data option should always return a function. If it does not, and we just simply return an object, then that data will be shared across all instances of the component.

BAD!

```
export default {
  data: {
    name: "My Window",
    articles: [],
  },
};
```

However, most of the time, the goal is to build reusable components, so we want each object to return a unique object. We accomplish this by returning our data object inside a function.

GOOD!

```
export default {
  data() {
    return {
      name: "My Window",
      articles: [],
    };
  },
};
```

10.5 Don’t use v-if with v-for elements

It’s super tempting to want to use v-if with v-for in order to filter elements of an array.

BAD!

```
<div
  v-for='product in products'
  v-if='product.price < 500>
```

The problem with this is that VueJS prioritizes the v-for directive over the v-if directive. So under the hood, it loops through every element and THEN checks the v-if conditional.

This means that even if we only want to render a few elements from a list, we’ll have to loop through the entire array.

This is no good.

A smarter solution would be to iterate over a computed property. The above example would look something like this.

This means that even if we only want to render a few elements from a list, we’ll have to loop through the entire array.

GOOD!

vue

```
<template>
  <div v-for="product in cheapProducts">{{ product }}</div>
```

```
</template>

<script>
 export default {
        computed: {
                cheapProducts: () => {
                        return this.products.filter(function (product) {
                                return product.price < 100
                        })
                }
        }
 }
</script>
```

This is good for a few reasons.

- Rendering is much more efficient because we don't loop over every item
- The filtered list will only be re-evaluated when a dependency changes
- It helps separate our component logic from the template, making our component more readable

10.6 Validate your props with good definitions

This is arguably the most important best practice to follow.
Well. It basically saves future you from current you. When designing a large scale project, it's easy to forget exactly the exact format, type, and other conventions you used for a prop.

```
export default {
  props: {
    status: {
      type: String,
      required: true,
      validator: function (value) {
        return (
          ["syncing", "synced", "version-conflict", "error"].indexOf(value) !==
          -1
        );
      },
    },
  },
};
```

10.7 Use PascalCase or kebab-case for components

A common naming convention for components is to use PascalCase or kebab-case. No matter which one you choose for your project, it's most important that you stay consistent all the time. PascalCase works best because it is supported by most IDE autocomplete features.

```
# BAD
mycomponent.vue
myComponent.vue
```

Mycomponent.vue
 # GOOD
MyComponent.vue

10.8 Base components should be prefixed accordingly

Another naming convention is focused around naming base components – or components that are purely presentational and help setup common styles across your app.
According to the Vue style guides, base components are components that only contain…

- HTML elements
- Additional base components
- 3rd party UI components

The best practice for naming these components is to give them the prefix "Base", "V", or "App". Once again, it's alright to use either of these as long as you stay consistent throughout your project.

BaseButton.vue
BaseIcon.vue
BaseHeading.vue

The purpose of this naming convention is that it keeps your base components together in your file system. Also, using a webpack import function, you can search for components matching your naming convention pattern and automatically import all of them as globals in your Vue project.

10.9 Components declared and used ONCE should have the prefix "The"

Similar to base components, single instance components (ones used once per page and does not accept props) have their own naming convention.
These components are specific to your app and are normally things like a header, sidebar, or footer. There should only ever be one active instance of this component.

TheHeader.vue
TheFooter.vue
TheSidebar.vue
ThePopup.vue

10.10 Stay consistent with your directive shorthand

A common technique among Vue developers is to use shorthand for directives. For example,

- @ is short for v-on:
- : is short for v-bind
- # is short for v-slot

It is great to use these shorthands in your Vue project. But to create some sort of convention across your project, you should either always use them or never use them. This will make your project more cohesive and readable**.**

10.11 Don’t call a method on created AND watch

A common mistake Vue developers make (or maybe it was just me) is they unnecessarily call a method in created and watch. The thought behind this is that we want to run the watch hook as soon as a component is initialized.

BAD!

```
<script>
  export default {
        created: () {
              this.handleChange()
        },
        methods: {
              handleChange() {
                    // stuff happens
              }
        },
        watch () {
              property() {
                    this.handleChange()
              }
        }
  }
</script>
```

However, there Vue has a built in solution for this. And it’s a property of Vue watchers that we often forget.
All we have to do is restructure our watcher a little bit and declare two properties:

- handler (newVal, oldVal) – this is our watcher method itself
- immediate: true – this makes our handler run when our instance is created

GOOD!

```
<script>
  export default {
        methods: {
              handleChange() {
                    // stuff happens
              }
        },
        watch () {
              property {
                    immediate: true
                    handler() {
                          this.handleChange()
                    }
              }
```

```
        }
 }
</script>
```

10.12 Template expressions should only have basic Javascript expressions

It's natural to want to add as much inline functionality in your templates as possible. But this makes our template less declarative and more complex. Meaning that our template just gets extremely cluttered.
For this, let's check out another example from the Vue style guide.Look how confusing it is.

BAD!

```
<template>
 {{ fullName.split(' ').map(function (word) { return word[0].toUpperCase() +
 word.slice(1) }).join(' ') }}
</template>
```

Basically, we want everything in our template to be intuitive as to what it is displaying. To keep this, we should refactor complex expressions into appropriately named component options. Another benefit of separating out complex expressions is that it means these values can be reused.

GOOD!

```
<template> {{ normalizedFullName }} </template>
<script>
 export default {
        // The complex expression has been moved to a computed property
        computed: {
                normalizedFullName: function () {
                        return this.fullName.split(' ').map(function (word) {
                                return word[0].toUpperCase() + word.slice(1)
                        }).join(' ')
                }
        }
 }
</script>
```

References

[1] Jon Brumfit. Java coding standard and guidelines for the herschel common science system. Technical report, ESTEC, October 2002. HSCDT/TN009.

[2] Sun Microsystems. Code conventions for the java programming language. Technical report, Sun, December 1999. http://java.sun.com/docs/codeconv.

[3] Sun Microsystems. How to write doc comments for javadoc. Technical report, Sun, 2000. http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html.

[4] J. Bloch. *Writing Effective Java*. Addison-Wesley, 1st edition, 2001.

[5] ESA Board for Software Standardization and Control. Java coding standards. Technical report, ESA, 2004.

[6] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[7] Sun Microsystems. Java look and feel design guidelines. Technical report, Sun, October 1999. http://java.sun.com/products/jlf/dg/index.htm.

[8] Vuejs best practices for pro developers, Matt Maribojoc https://learnvue.co/2020/01/12-vuejs-best-practices-for-pro-developers/