

Jutil.org

Jan Dockx
Marko van Dooren
Eric Steegmans

Report CW 342, June 2002



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Jutil.org

Jan Dockx
Marko van Dooren
Eric Steegmans

Report CW 342, June 2002

Department of Computer Science, K.U.Leuven

Abstract

Although reuse is one of the key advantages of object-oriented software, there is still a lot of general purpose code that is written over and over again. In this paper we take a look at the practical problems that prohibit the reuse of such code, and present the Jutil.org project as a solution to such problems.

Keywords : library, object-oriented, reusable.

CR Subject Classification : D.2.13, D.1.5, D.2.4, D.2.7, D.2.9

Jutil.org

Jan Dockx
Marko van Dooren
Eric Steegmans

K.U.Leuven

Abstract. Although reuse is one of the key advantages of object-oriented software, there is still a lot of general purpose code that is written over and over again. In this paper we take a look at the practical problems that prohibit the reuse of such code, and present the Jutil.org project as a solution to such problems.

1 Introduction

One of the major advantages of object-oriented software is reuse. However, we often find ourselves writing the same general purpose code over and over again in different projects. In this paper, we take a look at some practical issues that are responsible for this situation, and we present the Jutil.org project as a solution to the problem.

In the next section we will look at some practical problems that prohibit the reuse of general purpose code. In section 3, we briefly present the Jutil.org project. Section 4 introduces quality labels and quality levels we are planning to use to indicate the quality of the code. In section 5, we take a look at the measures that will be taken in order to obtain a high quality library. Section 6 mentions the license under which the source code will be made available, and section 7 presents the features currently present in the Jutil.org project. Finally, section 8 invites you to participate in the project.

2 Why we don't reuse enough

Why is there so much code that is not reused, but implemented time and time again in different projects ? Is it because general purpose code is difficult to reuse in object-oriented programming? We do not believe so, the standard Java API[5] and EiffelBase[2] are reasonable counterexamples. If the theory, in this case object-oriented programming, is not the cause of the problem, chances are good that nature of the problem is a practical one. We can distinguish two kinds of practical problems:

1. Management problems
2. Quality problems

2.1 Management problems

The user's point of view

As programmers, we have some requirements concerning management when we are looking for general purpose code to reuse.

- We want to be able to submit a bug report when we find a bug, so it is not sufficient that we can only download the library.
- When the problem that is solved by the code is a moving target, for example reading files in a certain file format, or when the code is not complete yet but has the functionality we need at this moment, we want to be reasonably sure that the code will be maintained. After all, we do not want to be stuck with an old library while the problem domain evolves, or when we need additional functionality. Of course we can enhance the library ourselves, but it is not appealing.
- The library must do a reasonable amount of work for us. We do not want to use enormous amounts of small libraries, because it takes some trouble integrating them into our own project, and mainly because we will probably want to keep them up to date.

The developer's point of view

From a developer's point of view, there are also some requirements, knowing that just dumping the code on a web site will not do much good.

- Creating a project is a significant amount of work, so creating one for only a small piece of code seems not attractive.
- The same goes for incomplete code. We often develop only the functionality we need, and want to get on with whatever we were using that code for.

2.2 Quality problems

Whatever general purpose code we reuse, has got to be worth reusing. After all, we will have to invest time in it, and we want to win that time back by not writing the code ourselves. Unfortunately, many of the quality aspects that are an added value in general are often neglected because there is no immediate need for them for the author of the code, who is using it in a specific situation.

Next, we will briefly discuss some general quality requirements.

Documentation

Good documentation is crucial to reusable code. If a developer has to go through a lot of trouble to guess what the code should do, and how to use it, he will probably go and search elsewhere. And if he can not find something well documented in a reasonable amount of time, he will most likely decide to write the code again.

Design

In the same category, there is the impact of the design. If I want to add some features to an existing library, but that library does not allow me to do so in a convenient way, I risk spending a lot more time in studying and modifying the library, with the possibility that I will fail, than when I would implement it myself. Being a human, I probably will not try that if I see signs of a bad design all over the place.

Implementation quality

Of course, the implementation of a library must be of a high quality if we want others to reuse it. A user of the library will feel more comfortable if he knows that certain measures have been taken to assure the correctness of the code. Examples are unit testing, integrity testing and proofs of correctness.

Ease of use

Finally, the influence of ease of use must not be underestimated¹. If the interface to the user of the code does not allow it to be used conveniently, programmers are unlikely to use it unless there is no better alternative.

3 Jutil.org

The Jutil.org project is an attempt to address these problems by creating an open source, high quality, general purpose API to complement the closed Java API. This way, we alleviate programmers from the burden of creating a new project, and provide a single source for finding and submitting general purpose code.

In the next section, we will introduce the notion of quality labels and quality levels, and in section 5, we will discuss the measures that are taken, or will be taken in the future in order to obtain a high quality library. Note that not all things are worked out completely yet since it requires a lot of work, and enough experience with the project. The specifics will be filled in when the need for them arises.

4 Quality levels

Different users will have different requirements. Some users will need the latest features, while for others it is important to have a very high quality library. In traditional open source projects, this is reflected by a stable release, with

¹ An example of the impact of a ridiculous number of arguments for a method can be found in [4], section 23.2

greater stability but less features, and an unstable release which has all the latest features, but is not ready for use in a product. We extend this approach to the tradeoff between features and quality with more levels.

4.1 The quality labels

With respect to stability, we think that the two most important quality aspects are the quality of the implementation, and the quality of the specification.

The implementation quality can be one of three labels. *untested* means that the implementation has not been tested. A method gets the quality label *tested* when it passes a unit test. For a class to be *tested*, additional integrity tests may be added. See section 5.3 for more information about the tests. Finally, a method is *proven* when its implementation has been proven, and a class is *proven* when all of its non-private methods have been proven.

The specification quality can also be one of three labels. An *informal* specification is a vague description of what the method is supposed to do. A *structured* specification is a more precise description, written in common language, which contains preconditions, postconditions, And finally, a *formal* specification is a very precise description written in a formal language such as JML[3].

4.2 The quality levels

A combination of a label indicating the quality of the implementation, and a label indicating the quality of the specification is called a *quality level*. Table 1 shows the possible combinations. Note that the combination *informal-proven* is not allowed because an informal specification will leave too much room for interpretation.

	Untested	Tested	Proven
Informal	✓	✓	✗
Structured	✓	✓	✓
Formal	✓	✓	✓

Table 1. The different possible quality levels.

4.3 Names for the quality levels

The combinations that make up the quality levels, are not really suited as names for them, they are too long.

We did not want to choose level 1, level 2, . . . because such a naming is confusing. Is level 1 the highest quality or the lowest quality ? We could have used it if we did not find better names, but fortunately we did.

Programs are combinations of small elements, the language elements. The better the structure that holds them together, the higher the quality of the program². The same is true for minerals of carbon. They are formed from C atoms³, and the structure determines the quality of the mineral. *Coal* is highly unstructured and contains a lot of impurities. Then comes *oil*, which has more structure and far less impurities. Adding more structure could give us *graphite*, widely used for light composites. And finally, the highest quality is of course *diamond*.

At this moment only two names have been chosen: *coal* represents {*Informal-Untested*}, and *diamond* represents {*Formal-Proven*}.

4.4 Using quality labels

At this moment the code in Jutil.org does not contain any quality labels yet, but is *coal* by default(although a lot of code is of a higher quality).

In the worst case scenario, we will have to add and maintain them manually, which is tedious and error-prone. Writing a tool that can create the labels itself using only the java files is virtually impossible⁴, but we could make it feasible by making some conventions, like using JML informal assertions for structured specifications, JML formal assertions for formal specifications, and for example a new tag in the javadoc comment block for writing informal specifications. Checking whether or not a method is tested could be done by adding tags to the test classes, and including the version of the original class that has been tested by that test class, so a tool can check whether or the class has changed, and tests possibly need to be updated.

None of these measures has been carried out yet. The first move to a higher quality level will probably happen when the packages *org.jutil.event* and *org.jutil.math.matrices* are finished.

5 How to obtain high quality

This section describes the measure we take, or hope to take in order to obtain a high quality API.

5.1 Guidelines

A first way to obtain high quality code, is to use a consistent design, coding and documentation style. The specific guidelines can be found on the web site of the project[8]. There is little use for repeating them here.

² We consider the specification to be a part of the program.

³ This would have fitted better in a C/C++ context, but the mapping from names to quality does not depend as much on taste when using these names as when using different types of coffee.

⁴ For example, it would need to detect the difference between an informal and structured description.

5.2 Documentation

Every class should ideally come with formal specifications, a class diagram, a general description, and possibly an example of how to use it. Each package should provide documentation briefly describing the classes in that package, and provide class diagrams. The next sections describe the different parts of the documentation.

Currently all this information is present in the javadoc documentation of a class, although it may be better to split it up in separate files so that once you know how a class works, you are not bothered anymore with information you already know.

Class diagrams

A picture can say more than a thousand words, and so can a class diagram. A class diagram is a very effective tool for presenting the structure of a class, meaning its superclasses and the associations it has, and for giving a quick overview of the operations that are applicable to an object of that class. Such a class diagram will be present in the javadoc of each class. More general class diagrams will be put in the package documentation, where entire hierarchies of classes can be shown. The class diagrams are currently generated from Together[10] by Batik[6].

Class documentation

The class documentation will consist of a general description of the class. If appropriate, a short example of how to use the code will be given.

Specifications

An extremely important piece of the API is its specification. The specifications are currently written in a mix of JML[3] and standard javadoc. The description of the method and its parameters is written in the javadoc block. That block is followed by a JML specification in which all specifications regarding preconditions, postconditions and exceptions are written.

5.3 Testing

For each class, there will be a separate test class. The API classes reside in the *org.jutil* package, while the test classes reside in a parallel structure in the *org.jutiltest* package. A test class should perform thorough unit tests on all methods, and integrity tests for the class it is testing. Currently, all test classes use the Junit[1] framework.

5.4 Proofs of correctness

Ideally, every method and class should be proven correct. The specifics of what syntax to use for the proof, and where to put the proof have not been worked out yet.

6 License

The code is licensed under an Apache[7] style license. The license can be found at [9]. The difference with the Apache license is that the obligation to mention the use of Jutil.org code in the end-user documentation is turned into a request.

7 Current status

Next, we will briefly describe the main features currently present in the Jutil.org project. The full set of features can be seen at the project web site[8]. The full documentation can be found in the javadocs.

7.1 Event system

The package *org.jutil.event* contains code that can be used to aid when implementing a system that sends events. It contains a class *EventSourceSupport* that keeps track of all listeners, and a class *Notifier* that is used by *EventSourceSupport* for actually delivering the events.

7.2 Collection operators

The package *org.jutil.java.collections* contains mainly collection and map operators. They can be used for applying certain actions on all elements of a collection, with or without support for roll-back in case of an exception, performing accumulations (e.g. forall and exists quantifier), filtering collections and calculating transitive closures over an object structure.

7.3 Matrices

The *org.jutil.math.matrix* package contains matrices, and support for several operations by means of a Strategy pattern in order to let the user choose between speed and precision. Currently supported operations are QR, LU, Cholesky, Hessenberg, Schur and eigenvalue decomposition, solving systems of linear equations and least square problems. The current package is not yet suited for high-precision calculations (although the results differ very little from the results of commercial products). At this moment the creation of a good design is more important.

8 Joining the project

Anybody who has general purpose code lying around or wants to help us can join the project. More details can be found on the project web site[8].

References

1. *Junit*. <http://www.junit.org>.
2. INTERACTIVE SOFTWARE ENGINEERING, *Eiffelbase*.
<http://www.eiffel.com/products/base.html>.
3. G. T. LEAVENS, *Java modelling language*. <http://www.jmlspecs.org>.
4. B. MEYER, *Object-Oriented Software Construction, Second Edition*, Prentice Hall, 1997.
5. SUN MICROSYSTEMS, *JavaTM 2 platform standard edition*.
<http://java.sun.com/j2se/1.4/>.
6. THE APACHE SOFTWARE FOUNDATION, *Batik svg toolkit*.
<http://xml.apache.org/batik/>.
7. ———, *The Apache Software Foundation*. <http://www.apache.org>.
8. THE JUTIL.ORG PROJECT, *Jutil.org*. <http://org-jutil.sourceforge.net>.
9. ———, *Jutil.org license*. <http://org-jutil.sourceforge.net/LICENSE>.
10. TOGETHERSOFT, *Together controlcenter*. <http://www.togethersoft.com>.