

Surveys on Component-Based Development

Jyrki Katajainen (Editor)

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk*

Contributors

Jesper Alf Dam
Toke Boye Riis
Jesper Rude Selknæs

Preface

This volume contains a selection of surveys written as part of the course “Selected topics in software construction” held in spring 2009 at the Department of Computer Science at the University of Copenhagen. The course components were lectures (8 á 2 hours), discussion sessions (5 á 2 hours), inspection meetings (2 hours per group), three assignments, a 3-week project, a one-day workshop, and an oral exam. Survey writing was one of the assignments; all the surveys were presented at the workshop held at the end of the course. Of the 16 surveys handed in, three were selected to this volume.

The course home page can be found at <https://isis.ku.dk/kurser/index.aspx?xslt=default&kursusid=28065>. In the lectures given by Claus Jensen, Bo Simonsen, Lars Yde, and myself the following topics were covered:

- design patterns,
- refactoring,
- development practices,
- software testing,
- programming style,
- development process models, and
- programming psychology.

The surveys included in this collection deal with component-based development, software-library development in particular, which was the main theme in the assignments.

This course was a collective learning experience for us all, including the teachers, and I want to thank all the participants for being willing to share their knowledge with others. Finally, I would like to thank the contributors to this volume for permitting us to publish their surveys.

October 2009

Jyrki Katajainen

Table of contents

Preface	i
Assignment formulation	1
Design patterns and good software: Cause or effect?	3
<i>Jesper A. Dam</i>	
Usability in program libraries	13
<i>Toke B. Riis</i>	
A survey on usability of parallel programming systems	24
<i>Jesper Rude Selknæs</i>	
Author index	34

Assignment formulation

In component-based development the idea is to build software systems using pre-fabricated components. Today most development falls into this category. As a historical remark, the idea of component-based development was first discussed in [M. Douglas McIlroy, *Mass-produced software components*] in 1968. In this assignment, your task is to write a survey on a topic related to component-based development. You can select the topic by yourself, but of course you are welcome to draw inspiration from the teachers and your fellow students.

For example, when having the CPH STL in mind, possible topics include, but are not limited to:

- client-usage experience of program libraries
- development experience of program libraries
- usability of program libraries
- design of program libraries
- evaluation of program libraries
- implementation of program libraries
- testing of program libraries.

The design of the CPH STL is presented in [Jyrki Katajainen and Bo Simonsen, *Applying design patterns to specify the architecture of a generic program library*]. In the referee reports to this paper, referees posed among other things the following questions, any of which could be elaborated in a survey:

- Compare the library to what can be done in other languages, such as C#, Java, Smalltalk, etc.
- Validate the quality of the design with experimental results.
- Evaluate the quality of the library in terms of run-time benefit.
- Describe the experience of writing a program that uses the components of the library extensively.
- Describe the experience of students who have benefited from the safe versions of the containers.
- Describe ways of configuring the library automatically so that it would potentially adapt depending on usage scenarios.

The assignment has three parts.

1) Survey writing. This part is individual. The surveys should be handed in via ISIS by 23rd March 2009 in PDF format. Please, share your survey with others to facilitate the later refereeing process. The survey should be written using the `DIKU-article` L^AT_EX style (see the CPH STL home page under Tools), and its length should be at most 12 pages. Be careful you mention your main sources in the list of references. In particular, try

to explain the topic using your own words and your own examples; avoid plagiarism!

2) Refereeing. Each survey should be reviewed by two other students. That is, in addition to writing a survey, you should comment two other surveys. (The course manager will distribute the surveys for refereeing. If you have any preferences, let him know.) You should provide your brief feedback by e-mail (before the workshop; cc: jyrki@diku.dk). This refereeing is an obligatory part of the assignment. Two members of the course team will read and evaluate all surveys. Recall that the weight of this assignment is 30% of the final grade. However, we will not publish the actual grade points achieved since grading will be done as an integrated whole.

3) Workshop. The philosophy of this course has been that “we learn much faster and much better with the active cooperation of others” [Gerald M. Weinberg, *The psychology of computer programming*, p. 5.i]. In compliance with this, we arrange a workshop on the 30th March (at **9.00-15.00**) where each student presents his survey to others (15 minutes per student). This presentation is an obligatory part of the assignment. In the middle of the day we have a joint lunch which is sponsored by the PE-lab.

The three best surveys—selected on subjective criteria by the members of the course team—will be invited to a survey collection which will be published as a CPH STL report. Additionally, each author will get a CPH STL T-shirt so, please, give your size (XXXL, XXL, XL, L, M, S) in a comment on the ISIS page when handing in the survey. After the course, you can use the survey as a starting point for a written project, a master’s thesis, or a scientific study.

Design patterns and good software: Cause or effect?

Jesper A. Dam

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jalf@diku.dk*

Abstract. The CPH STL is currently in the process of being refactored to use the familiar design patterns defined by Gamma et al. [10]. Design patterns are commonly seen as vital ingredients of well-designed object-oriented code, and are used liberally in many software projects. However, there are a number of different viewpoints on the value of design patterns, and how they should be allowed to influence the design of our code. In this survey, I will try to outline some of these perspectives on design patterns, and their value as tools for component-based development.

1. Introduction

Since Gamma et al., popularly known as the Gang of Four, published their book on design patterns [10] in 1995, design patterns have become a major part of most developer's vocabulary, and the patterns they proposed have been widely accepted as good design, and effectively synonymous with writing good code; to write good, robust, flexible and extensible code, we must use design patterns, and if don't use design patterns, our code is inherently flawed.

However, as with most things, there are dissenters, developers who dislike this approach—some who believe the entire *idea* of recurring design patterns is flawed and does more harm than good, as well as more moderate views claiming that there is value to the Gang of Four's book, if it is just used correctly, which it isn't.

Perhaps, rather than an encouragement to actively implement these design patterns in our code, we should really read the book as a catalog of the design patterns that tend to show up *in any case*, and simply use the book to build a common vocabulary to discuss our code, rather than as a set of predefined building blocks to use in the design phase.

Perhaps design patterns are not the *cause* of good code, but a *side-effect*; something that shows up by itself in any high-quality code-base, but isn't a goal in itself.

The following sections will present these different views on design patterns, and categorize them into a few broad classes.

2. Design patterns are there to be used

There are plenty of sources indicating that the use of design patterns makes code more readable and maintainable, and make programmers more productive. They offer tried and tested solutions to common recurring problems; the programmer doesn't have to come up with a solution from scratch, he can simply look one up in the GoF book.

And because everyone knows these common building blocks, they also make it easier for others to understand the code as well. The maintainer simply needs to know that “the code uses the bridge pattern to implement class X, the visitor pattern to traverse Y, and the template-method pattern to ensure that invariant Z is always maintained”, and then he understands the code base.

In other words, design patterns should be actively used. The programmer should consciously decide “I will use the visitor design pattern to solve this problem. I will use the template-method design pattern for that problem”.

It seems common to think of patterns as best practices: If you have a problem that *can* be solved by using a design pattern, then using that design pattern must necessarily be the *best* solution. It is designed by experts, after all. It is well understood, commonly known, flexible, robust, extensible, and everything else we require from high-quality code. Various degrees of this point of view are common, and there is little need to examine it further. What all proponents of this view have in common, is the belief that we should *write* design patterns. It is not enough to notice them when they turn up in our code. We only reap the benefits of design patterns by consciously *implementing* them as solutions to the problems we're faced with.

Ng et al. [13] have even shown that knowledge of design patterns may outweigh work experience as a factor determining the productivity of a programmer. However, on closer examination, their study seems biased because ultimately, what they seem to measure is not “how much faster does work go in a well-structured code base using known design patterns”, but rather “how much faster does work go if expert programmers who are *already* intimately familiar with the code base do all the initial work, preparing for the implementation of the new features”. And in this case, we would obviously expect a significant increase in productivity, whether or not they used known design patterns.

It is clear that there are some good points in this point of view: having a common terminology certainly makes it easier for us to communicate. If everyone agrees on what a visitor looks like, we don't need to describe the exact implementation of it, in order to describe the software system. And the more we use the patterns we have names for, the easier it will be to describe and explain our code to anyone who is familiar with the same names.

But other implications are harder to accept as universal truths. If a pattern *can* be used to solve a problem, does that mean it is the best choice? Should we automatically use a design pattern simply because it is able to solve the problem? Are there no downsides to the use of design patterns?

Is there no trade-off? The Gang of Four's book is silent on the subject of when *not* to use a design pattern, which seems to have led to a common belief that there *are* no reasons not to use them.

3. Design patterns are harmful

At the other extreme, some software developers believe that design patterns are harmful; that the entire field of software development would have been better off if the Gang of Four's book had never been published; simply *knowing* about design patterns is harmful, or as Ericson, a lead programmer in one of Sony's game divisions says on his personal blog [9], they cause "mind rot". He points out the tendency to use patterns everywhere, instead of thinking about the problem. Once a programmer has read about the factory pattern, he will try to find places where it can be used. Every time he has to instantiate an object, he will try to create a factory for it. Ultimately, this leads to over-complicated code that is almost impossible to read, inefficient to execute, and takes 500 lines of code to implement something that could have been done in 30.

Another danger is that the programmer may *think* he recognizes the problem, applies a pattern to solve it, and only then realizes that the problem was actually subtly different from what the pattern was designed for; the pattern wasn't the solution at all, but it looked similar. As an example of this, he offers an anecdote from his own time as a student [9]. In trying to do the least possible amount of work to pass a statistics exam, he taught himself to recognize a small number of recurring "patterns" in the exam questions, assigned names to each, and memorized the solutions to these. This approach backfired as he mistakenly believed a question in the exam could be solved by the pattern he calls "*the plank*".

The conclusion he draws from this is that in trying to turn everything into patterns, rather than just *thinking* about the problem, it is easy to blind oneself with patterns, and subsequently turn every problem-solving process into a quest to find the pattern that matches, even if that means changing (or misunderstanding) the problem to fit the solution (and thus no longer solving the required problem at all).

Of course, one personal anecdote hardly proves that design patterns do more harm than good, and his failure could just as easily be attributed to his attempt to cram a semester's worth of learning into a single week. Or we could point out that just because *he* apparently tried to rely blindly on predefined patterns to solve problems he didn't understand, that doesn't mean others will do the same.

While he may draw some extreme conclusions based on this, he may still have a valid point. Treating patterns as ready-to-use prepackaged solutions is dangerous. We should be wary of anything that encourages us to mindlessly insert pre-generated code, rather than *thinking* about the proper solution to the problem.

The other side of his objection is that design patterns offer nothing new—even when they are not misused, there is simply nothing to be gained: the design patterns used are either too trivial and obvious to deserve a name, or have been known for decades under other names, or are exclusively relevant to a few flawed languages that (he claims) should not be used in any case.

While Dominus [7, 8] does not share this extremely hostile view of design patterns, and in general believes that they are useful in some limited scope, (although again, only as a crutch for developers who are forced to work in “the wrong” languages), he does point out a very specific way in which the Gang of Four’s book may have caused harm: although the book is based on the ideas of the architect Christopher Alexander [1], they use the word “pattern” to mean different things, and in doing so, the Gang of Four’s book effectively obscures Alexander’s much more profound ideas.

Because people already *think* they know what is meant by patterns, they will never even look at Alexander’s books or his idea of a pattern language. Alexander wanted to develop a “pattern language” which can be used by both users and developers to express and discuss design. Rather than specifying *how* to solve specific problems, it helps determine *what* should be designed, and thus create your own patterns for each individual situation.

But once people have become used to the Gang of Four’s definition of patterns as pre-defined standard building blocks that simply have to be glued into a code base in order to work their magic, they will never discover Alexander’s much more interesting idea, and in this sense, the mere existence of the design-pattern book has harmed the field of software development.

Another real problem with design patterns may be that not all of them are actually *good*. In particular, the singleton pattern has often been “re-classified” as an anti-pattern, for example, in [5, 11, 15, 16]. The argument can be summarized by pointing out the two traits a singleton guarantees:

- It guarantees global access.
- It guarantees that only a single instance of the class is instantiated.

The first property is typically considered a bad thing. Most modern programming paradigms try to eliminate shared global data. Global mutable data causes many problems, so we should be skeptical of a design pattern which *forces* data to be global.

The problem with the second property is more subtle, but essentially, it offers a guarantee that we don’t need, and in doing so, it needlessly removes flexibility from our code. In the early design stage of a project, it is common to assume that only one instance of some component will be needed: we only need one logger, one database, or one configuration object. It wouldn’t make *sense* to have more than one. And just as often, it turns out later on that we were wrong. We do need multiple loggers, because we want to log multiple types of messages to be used for different purposes. We need to access a second database as well. Or while only one configuration is ever *active* at a time, multiple configuration objects may exist. The user will want to modify the configuration from the option screen without having them applied until he presses OK.

So the singleton seems to combine two traits we often go out of our way to *avoid*. It removes flexibility, and it forces data to be global. In the rare cases where one of these traits is needed, we should attempt to provide that, without further burdening ourselves with the other trait.

Overall though, it seems that Ericson [9] is overstating his case a bit. A personal story of how *he* misused the idea of patterns, and a claim that they're generally too simple to read about does not mean that the publication of the book has set the field of software development back by decades, as he claims. However, he does show how easy it is to abuse or be misled by the concept of design patterns, and as a consequence, fail to solve the required problem, and this point should not be ignored.

4. Design patterns are not “used”, they are found

Somewhere between these two extremes, there are developers who primarily see the Gang of Four's book as an observation—a neutral statement that these types of patterns tend to occur a lot in our code. In that sense, the book should be seen as a taxonomy of the patterns that we have all been writing intuitively; the only contribution of the book is to put names to these patterns to aid in communication as well as further studies. A zoologist may put names to newly encountered species, but he doesn't pretend that they have a value, that they are “good” or “bad”. They just are there. And some developers suggest that we should treat design patterns the same way.

Rather than being a recipe for building successful software, design patterns should be used to analyze existing code, discovering and identifying common traits. Braithwaite [2] is skeptical that every problem can, or should, be decomposed into a small number of known patterns, but also argues that when *discovered* in code, they ease communication; because we all know what the flyweight pattern is, we are able to recognize it when we see it, put a name to it, and use that name in explaining the code to others.

Another blog by Cunningham [4] sees the Gang of Four's book as little more than a collection of data, but lacking a conclusion of any kind. It does not tell programmers what they should do with these patterns, it simply provides some data points that others may analyze and build on to in subsequent studies.

While the book does largely seem to be a collection of data, a list of patterns that the authors have encountered often in their own code with no clear conclusion, it seems unlikely to me that their intents were solely to provide a neutral observation for others to base their research on. In the introduction of their book it is stated that “Our goal is to capture design experience in a form that people can use effectively.” As such, it seems clear that they intend for these patterns to be *used*—programmers should read the book, and then go forth and write visitors and bridges and factories into their code.

But regardless of what the Gang of Four themselves intended, perhaps

the book *is* more valuable as an objective data point, as an index of code that shows up often, for better or for worse, in successful projects. While the Gang of Four apparently intended us to read the patterns *they* use, and copy them in our own code, they fail to put forth a very compelling argument. They simply state that “these are patterns we see a lot in (our) good code”, and jump to the conclusion that therefore the patterns themselves must be one of the *causes* of their code’s quality. Of course, as anyone who has even a passing knowledge of statistics knows, correlation does not equal causation. It may be true that these patterns are often seen in high-quality code, but that does not mean they *cause* high quality code.

5. The GoF simply missed the point

In [6], Dominus looks along a completely different axis than most of the above. Whether design patterns are good or bad is irrelevant. What’s interesting is that the Gang of Four drew the wrong conclusion. They proposed solution to the problem of designing code is to turn the programmer into a “human compiler”. The programmer is “programmed” to perform a simple macro substitution when faced with certain problems. “This looks like a singleton. Copy and paste the singleton code from the Gang of Four”. No further thought involved. This is copy/paste code at its worst. Rather than asking “how can I solve this problem”, the book trains us to ask “does this look like one of the patterns defined by the Gang of Four?” As a consequence, we get the same logic duplicated over and over, so there is a high potential for bugs in the repetitive code that no one bothered to think about, because they just had to copy it from a book.

Instead, he suggests, the book should be considered valuable data for language designers. These are the common recurring problems that need to be built into the language. As he points out, simple function calls were once a design pattern. In assembly language, the programmer had to implement this mechanism himself. And today, we know that the “correct” solution was not to formalize this design pattern and give it a name, but to build it into our programming language. We no longer have to think about implementing function calls. The compiler generates them for us at our request.

Similarly, object-oriented programming in C can be considered a design pattern. To get polymorphic behaviour, we can add function pointers to a `struct` to emulate a `vtable`. And again, higher-level languages such as C++ took this design pattern, and built it into the language so this is done automatically. A programmer simply has to type a few keywords, like `class` or `virtual`, and the compiler does all the tedious implementation work.

Design patterns should be seen as red flags, indicators that the language they occur in is flawed. Programmers should try to move to a more powerful and expressive language, and language designers should get to work *fixing* these shortcomings. Many others have observed that design patterns seem to be largely a phenomenon in object-oriented or imperative languages. The

factory pattern becomes redundant in a language with higher-order functions and currying. The visitor pattern is a workaround because the languages in question do not support multiple dispatch directly, and the iterator pattern becomes largely redundant with the `map` and `fold` functions from functional programming. The C2 Wiki [3] contains an interesting list of design patterns, and the language features that make them redundant.

And as previously mentioned, some of the design patterns seem to overshadow more general and universal ideas, such as Alexander’s notion of a “pattern language”, where patterns are not pre-designed building blocks to be plugged into the project, but a process for coming up with *new* designs as needed.

Norvig [14] classifies patterns into three groups: invisible, formal, and informal. An *invisible* pattern is one that is so well integrated into the language that the programmer can use it without thinking. Classes in C++ are an example of this; the “object-oriented” `struct` design pattern in C is no longer necessary. The procedure-call pattern is another example. These concepts are so common and well-integrated into our language that we often don’t realize that they were once “patterns” the programmer had to implement over and over.

Formal patterns are implemented in the language, or its standard library, so that it can be invoked or instantiated when required. The C++ STL iterators may be an example of this level of integration. All the hard work has been done, but we still have to invoke the code to use it. The pattern is still visible, it has simply become easier to use.

Finally, the “informal” level is the one the Gang of Four seems to aim for. They only wish to describe each pattern so that a programmer can implement it from scratch at every use.

Whatever the value of the design patterns that the Gang of Four identified, they do seem to focus on low hanging fruit. They correctly identified the problem that “design is hard”. In their ideal world, it seems that ordinary programmers simply should not design code at all, but simply copy endlessly from what “experts” have already written. Rather than trying to make design easier, by coming up with a more reliable process for it, like Alexander suggested (although his domain was architecture, rather than programming), they take the easy way out, trying to suggest that programmers should simply reimplement the designs made by “experts”.

Having correctly identified common sources of duplicate and repetitive code, they are satisfied to simply canonize it as “the right way to do it”, teaching programmers to duplicate it *further*, instead of actually trying to *avoid* the duplication. In this sense, the Gang of Four’s largest flaw was simply their lack of ambition. In Norvig’s terminology, they only aspired to create *informal* patterns. The formal or invisible levels of implementation would seem much more valuable.

6. Conclusion

It is clear that many design patterns are simply there to cover up deficiencies in individual languages. They invoke the “human compiler” to patch over the problems that the C++ or Java compiler is unable to cope with. These are patterns that simply don’t exist in other languages—there is no need for them because the problem they solve is solved directly through the language instead, in the same way that “function call” is no longer a design pattern the way it was in the days of assembly programming.

Any Python, Ruby, or Haskell programmer can point out many examples of this. Many of the problems that require the object-oriented programmer to code the same things again and again simply go away in other languages. If the problem doesn’t go away entirely in these languages (like the problem of applying an operation to a sequence of values does in functional programs), the pattern for solving it either becomes a simple library function that can be applied at will, a truly reusable component, or a built-in language feature.

The correct response to many of the Gang of Four’s design patterns should be to switch to (or, if necessarily, design) a language where they become irrelevant.

Then there are patterns that simply seem like mistakes. It is hard to imagine what the justification was for including the singleton pattern in their book. Perhaps it was simply pragmatism—they feared that people would never adopt this new “OOP” thing if it didn’t allow them *some* way to do what they’d always done, i.e. used global data. The singleton is often considered an “object-oriented way to do globals”. Another possibility is that they simply got carried away, trying to identify as many patterns as possible, and while they *did* frequently encounter singletons in their code, they failed to realize that it was a design flaw, perhaps an old habit from older imperative languages where global data was widely used.

Finally, there are the design patterns that seem like genuinely good ideas. The template-method pattern is useful—a class where only a few customization hooks are exposed to the programmer, so that class invariants can be guaranteed to be maintained may be obvious to some, but formalizing it and making people aware of it can only be beneficial. Similarly, the bridge pattern may seem obvious to many, but it is a good example of separation of concerns. It allows the CPH STL to decompose common container classes into small internal classes having a minimal interface, and then wrapper classes which build on them to add extra convenience functionality.

These are common-sense designs that everyone can agree on. It is good to keep a clean separation of concerns between classes, just like every class should be able to guarantee its own consistency and validity at all times. These patterns are also much more general—they can be applied with little modification to Python or SML or any other language.

No matter the language, and even in languages that don’t have classes, there is always a desire to keep different responsibilities separated so that

they can be analyzed and understood in isolation. As McConnell mentions throughout his book [12], we will always need abstractions that let us focus on a small part of our code at a time, and some of the Gang of Four's design patterns facilitate this.

In general, it seems that design patterns, at least in the Gang of Four's terminology, are not entirely compatible with the goals of component-based development. Rather than trying to abstract common and recurring code structures out into reusable components, as has happened on many levels in the evolution of programming languages (ranging from the primitive support for function calls in the compiler, to abstract high-level features such as the adoption of lambda-calculus or closures to many languages, or the many helpful libraries in Python or even the STL), they seem to be satisfied with manually rewriting the same code over and over, every time the problem is encountered. They merely hope to make this process slightly more painless by teaching "the human programmer" which code to insert when faced with these known, recurring problems.

However, it may also be worth recognizing that we will always use patterns. Experienced programmers will always remember how they solved previous problems, so that they can apply similar solutions to problems in the future. So even if we were to assume that the most negative views were correct, and that design patterns are downright evil, what is the alternative? Rather than everyone using the same patterns, each programmer would come up with their own, which they were unable to communicate to others, and which may not be elegant solutions to the problem at hand. Trying to start completely from scratch every time you solve a problem is hardly a solution either. We need to be able to build on and share our prior knowledge to be productive.

Perhaps the conclusion should simply be that there is value in the Gang of Four's book, as long as the reader remembers that not all patterns are born equal. Some patterns should generally be considered anti-patterns, others are useful in specific languages, and some represent almost universal "best practices", qualities a programmer should always strive for in his code.

Perhaps the Gang of Four's book should be thought of as basic research—they gathered some interesting data, but have not drawn a meaningful conclusion. They have identified patterns that often occur in our code, and it is left for future work to consider the value of each of these patterns, and whether it is one that should be actively implemented, merely named and recognized when it occurs, avoided altogether, or perhaps fixed "at the language level".

As such, it seems remarkable that I have been unable to find any such studies. It seems that everyone has simply taken the book at face value, without further analyzing the data the Gang of Four presented. As this survey has shown, many blog posts and other informal discussions can be found, but no formal study or discussion of the individual patterns seems to exist, although I am sure such a work would reveal valuable information.

References

- [1] C. Alexander, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press (1977).
- [2] R. Braithwaite, *Pouring water back into the flask*, <http://weblog.raganwald.com/2007/12/pouring-water-back-into-flask.html>.
- [3] *C2 Wiki: Are design patterns missing language features*, <http://www.c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures>.
- [4] B. Cunningham, *Design patterns aren't from hell: They were driven there by developers*, <http://bc-squared.blogspot.com/2008/09/design-patterns-arent-from-hell-they.html>.
- [5] S. Densmore, *Why singletons are evil*, <http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx>.
- [6] M. Dominus, *Design patterns of 1972*, <http://blog.plover.com/prog/design-patterns.html>.
- [7] M. Dominus, *"Design patterns" aren't*, <http://perl.plover.com/yak/design/>.
- [8] M. Dominus, *"Design patterns" aren't: Postscript*, <http://perl.plover.com/yak/design/samples/note.html>.
- [9] C. Ericson, *Design patterns are from hell*, <http://realtimecollisiondetection.net/blog/?p=44>.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
- [11] A. Latham, *Singletonitis*, <http://pragmaticintegration.blogspot.com/2006/02/singletonitis.html>.
- [12] S. McConnell, *Code Complete*, 2nd edition, Microsoft Press (2004).
- [13] T.H. Ng, S.C. Cheung, W.K. Chan, and Y.T. Yu, *Work experience versus refactoring to design patterns: A controlled experiment*.
- [14] P. Norvig, *Design patterns in dynamic programming*. <http://norvig.com/design-patterns/>.
- [15] J. Petrie, *Performant singletons*, <http://scientificninja.com/advice/performant-singletons>.
- [16] A. Vieiro, *Singletonitis*, <http://www.antonioshome.net/blog/2006/20060906-1.php>.

Usability in program libraries

Toke B. Riis

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
tokeboy@diku.dk*

Abstract. This is the report for the S-assignment on the course **Selected topics in software construction**. It contains a brief usability evaluation of three program libraries as well as recommendations for the CPH STL.

1. Preface

In this assignment I will evaluate three existing program libraries for the following programming languages; C# [3], Java [6], and PHP [1]. I’ve chosen the three since these are those I have the most experience in using.

Since usability evaluation of program libraries is uncharted territory, I’ll base my evaluation on the online documentation as well as the programming environments used with the program library. This is done mainly because they—in my opinion—contribute immensely to the usability.

The reasoning is that, if you have a large amount of building material and don’t know anything about construction of buildings, a blueprint will have great value—probably even indispensable. Hence documentation is important. Using the building material metaphor once more, you do need some basic tools (e.g. a hammer and a saw) in order to put the building materials together. Hence, tools—or in our case programming environments—are important.

Since the reader isn’t necessarily a usability expert I’ll start by introducing the usability concepts I will use throughout this document. Afterwards evaluation of the program libraries is described. Finally, recommendations for the CPH STL [2] will be presented.

1.1 Usability concepts

I understand the word “usability” as a concept for analysing how “usable” some artifact is. This is of course a very broad definition so more clarification is obviously needed. I have only worked with usability in correlation with graphical interfaces during classes in Human Computer Interaction. Although the concepts from this research area don’t fully apply to the scope and domain of this assignment, they do have similarities which I will use.

Since usability, as mentioned, is a very broad term, other more specific concepts exist. Ensuring that these concepts are fulfilled will automatically make the artifact more usable. The following list is some of the more widely accepted terms—more probably exist.

Learnability. Is it easy to learn? Is it easy to learn the structure?

Efficiency Is it memory efficient? Fast? Does it use little hard-disc space?

Does the programmer work faster?

Accessibility. How accessible is it? Is it open source? Is it well documented?

Does it run on all operating systems?

Reliability. Does it always work? Does it compile? Does it work in all environments?

Memorability Is the program library easy to remember?

Correctness. Does it have errors? Does it do what it's supposed to? Does it prevent errors?

Satisfaction. Are users of the program library happy? Excited?

These concepts do have some correlation. For instance, if a program library is easy to learn it's *probably* also easy to remember and vice versa. If a program library is reliable it's *probably* also correct and vice versa. If the framework is reliable, correct, easy to remember and learn, then the programmer can *probably* produce code faster than if the program library was the opposite (efficiency). Satisfaction of users surely relates to the general usability concept; if the program library is usable, then the users tend to be happy—and actually vice versa.

Bottom line is that if all the concepts are fulfilled, then the program library will be highly usable!

2. C#

In this section I will evaluate the C# standard library. First I will evaluate the documentation and then the programming environment identifying good and bad usability.

2.1 Documentation

The .NET framework is a huge program library. Due to this fact the learnability is very low. There is no point of entry, if you want to start using it, thereby making it very hard for novice users. For the same exact reason the memorability of the program library is also low. Novice users—and even experts—will find it virtually impossible to remember all the details of the program library. It is simply too big for one person to know everything.

For some people this would have an impact on their satisfaction of use, but the web-based documentation does support users in locating functionality; they have a search feature. Furthermore, some documentation pages have small code examples, which make it easy to figure out how to use the described functionality.

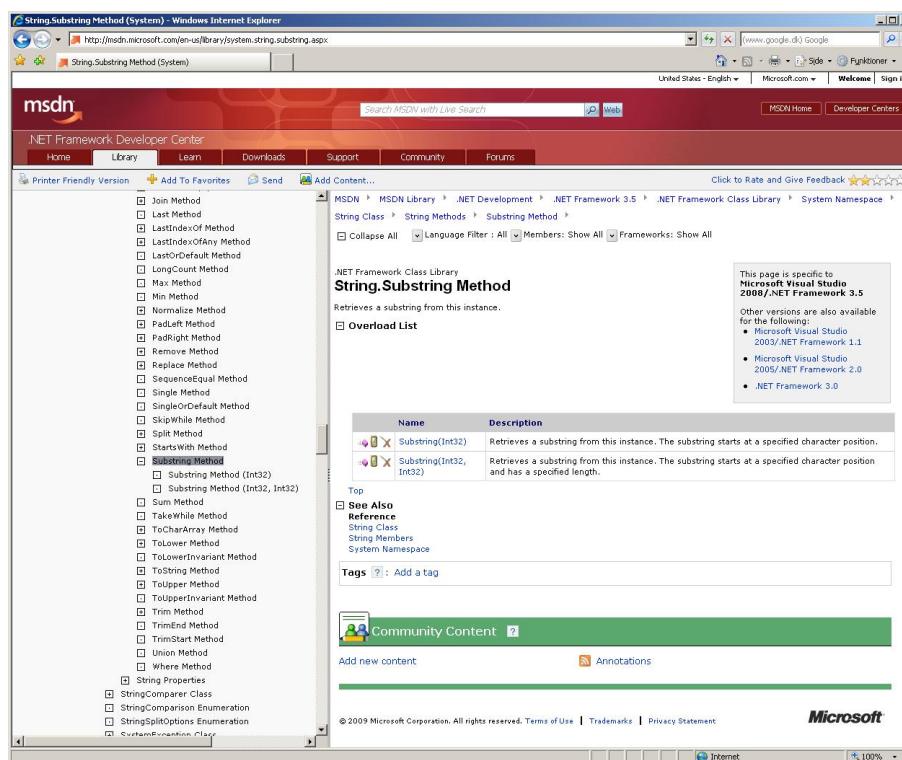


Figure 1. Microsoft .NET framework 3.5.

The .NET framework isn't that accessible. It only runs on Windows platforms and it isn't open source. This is changing as the mono project¹ is becoming better.

The online documentation is very efficient to use, if you know what you're looking for. If you don't, you'll have a hard time locating the correct functionality, since the search feature isn't that great.

Is the documentation reliable? In short: yes. I for once, haven't found any flaws in the documentation. Some functionality, though, should have been described a bit more. For instance a search for "default proxy" produces a link to a page with deprecated functionality. Here it would have been preferable if the page at least could point to a new functionality.

2.2 Environment

The most used editor for C# .NET is obviously the Visual Studio programming environment. This environment supports the user with the basic editor features like syntax highlighting. Furthermore it includes the "IntelliSense"-feature which reduces the user's memory load. It also includes some neat

¹ For an open source implementation of .NET, see <http://www.mono-project.com>.

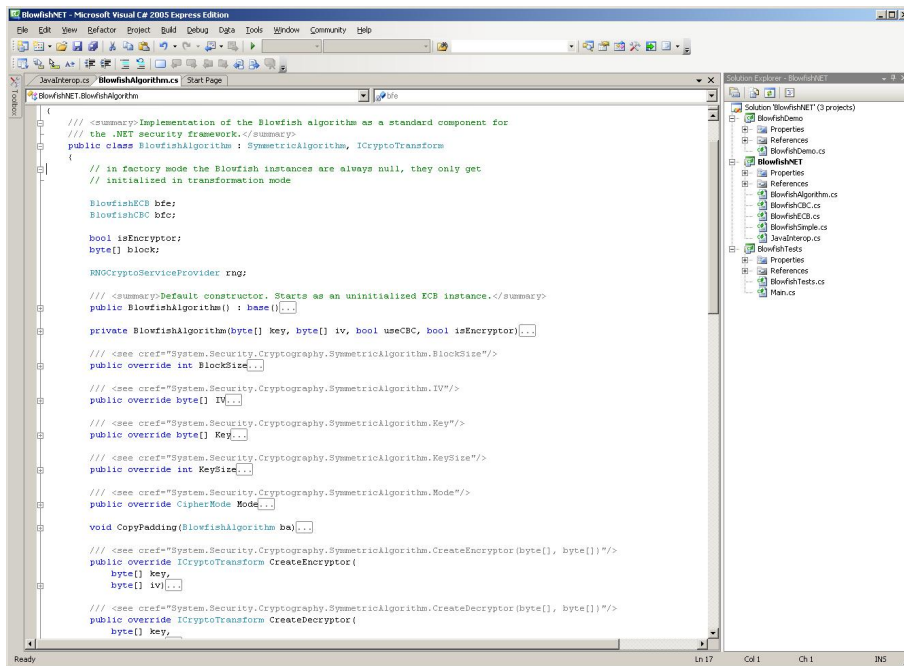


Figure 2. Microsoft Visual Studio for C# 2005.

debugging features, which reduces the complexity and time consumption, thereby increasing the efficiency of the programmer.

The compiler offers excellent feedback, e.g. it states what is wrong and on which line it is. Furthermore one can simply click the error and be forwarded to the specific line. The compiler prevents some errors, as it gives warnings when you for instance have forgotten to assign a variable a default value. This also relates to the efficiency of the programmer.

The learnability of the environment is good. At first glance it might seem very complex, but when you start to use it, it turns out to be very simple to use, for instance, by offering guides for novices. But this might be a matter of opinion. In my opinion it is easier to use one environment instead of a separate editor, compiler, and debugger.

All the above features give the user a satisfactory feeling when using the environment and thereby also the program library.

2.3 General usability features

The accessibility and reliability of the .NET framework on Windows machines is very good. Most PCs have the .NET framework installed, which allows for easy distribution of programs written in C#. Since this is the case, your code will run on any type of hardware, since the result of a compilation is .NET byte code.

Since the result is byte code, the efficiency is not as good as a binary executable, but still it is faster than e.g. Java and PHP, which will be evaluated in later sections.

Since the program library itself is closed source, the programmer doesn't know what happens "behind the scenes", which for some user types might result in some frustration, making them less satisfied. Furthermore, memory management is handled by the program library itself, making it difficult for a programmer to control when memory clean-up is performed.

The program library is, as far as I can tell, very consistent. There is a strong hierarchical design structure which supports the learnability.

3. Java

In this section I will evaluate the Java standard library. First I will evaluate the documentation and then the programming environment identifying good and bad usability.

3.1 Documentation

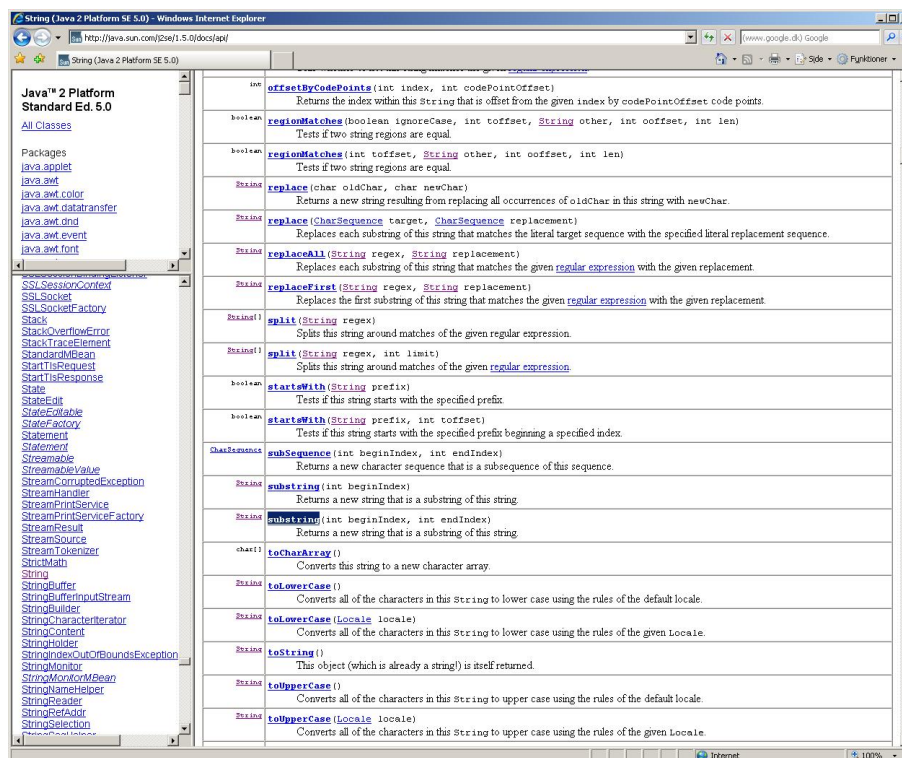


Figure 3. Java 2 platform standard edition 5.0 API specification.

The Java API is an example on how *not* to make an online documentation. In comparison to the .NET, the Java documentation has some problems.

Firstly it doesn't have search functionality. This will prevent efficient look-ups into the documentation. Obviously Google² can resolve this. The documentation is comprehensive as well. The learnability of the program library is in the light of this, low and the human memory consumption high.

On the other hand, the documentation is consistent and the descriptions of the functionality are, although a bit extensive, well made, which adds to the learnability. Notably, the program library has a highly hierarchical structure, but the online documentation doesn't show this.

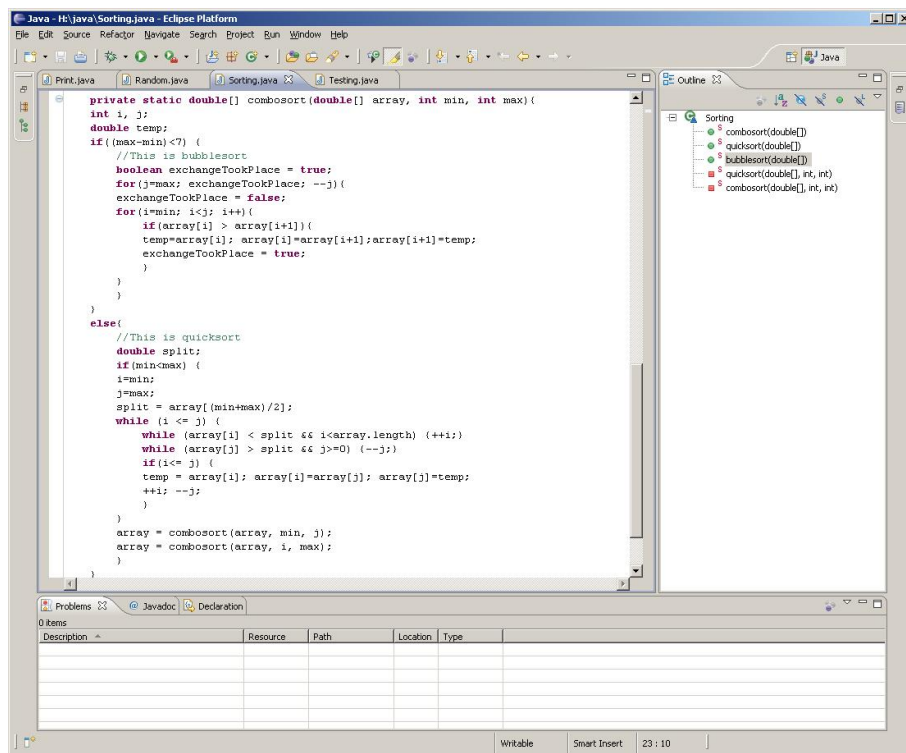


Figure 4. Eclipse platform.

3.2 Environment

The most widely used environment for programming Java is the Eclipse platform³. Similar to the Visual Studio environment, Eclipse offers syntax highlighting, a debugger, and an informative compiler. In my opinion they are, besides the graphical look'n'feel, very much alike. Therefore the same

² <http://www.google.com>

³ <http://www.eclipse.org>

strengths apply. The main difference is that Eclipse has a built-in Subversion client, which enables reversal of changes even though the editor has been closed.

Java doesn't solely rely on one environment; you can choose your own editor. Examples of such could be Emacs⁴ and JEdit⁵. These editors only support syntax highlighting and rely on different tools for compilation and debugging. So a greater versatility is present here than in .NET⁶.

3.3 General usability features

The accessibility of Java is sublime. This is due to the fact that Java "runs on everything". As you might know, Java utilizes a virtual machine, so that despite hardware and operating system your program will run. Although improvements to the efficiency of this virtual machine have been made, the code doesn't run that fast. Especially the load of the virtual machine is time consuming. But the accessibility depends on whether, or not, the users have Java installed just as it was the case with .NET.

Depending on what you're coding, the satisfaction differs. For instance, if you want to make some real-time tracking, you might get dissatisfied. If you on the other hand are programming a banking system where your customers are using different operating systems and hardware platforms, you might be glad you've chosen Java as your platform. Bottom line is that satisfaction depends on what one is programming. It should be noted that Java has the advantage of being open source. This will enable expert users to figure out what is inside the "black box".

Reliability of Java is high. This is mainly due the large amount of contributors and users. Errors are found quickly and corrections are made continuously.

4. PHP

In this section I will evaluate the PHP standard library. First I will evaluate the documentation and then the programming environment identifying good and bad usability.

4.1 Documentation

In my opinion this documentation is by far the best one. The search result gives you options if you've misspelled the search keywords. Furthermore, there are many examples. These are not only limited to be contributed by website administrators, you as a private person can contribute too. This enables the program library to be easy to use for novices, since there are so

⁴ <http://www.gnu.org/software/emacs/>

⁵ <http://www.jedit.org>

⁶ Notably .NET has a command-line compiler, which is practically never used in coherence with a third-party editor.

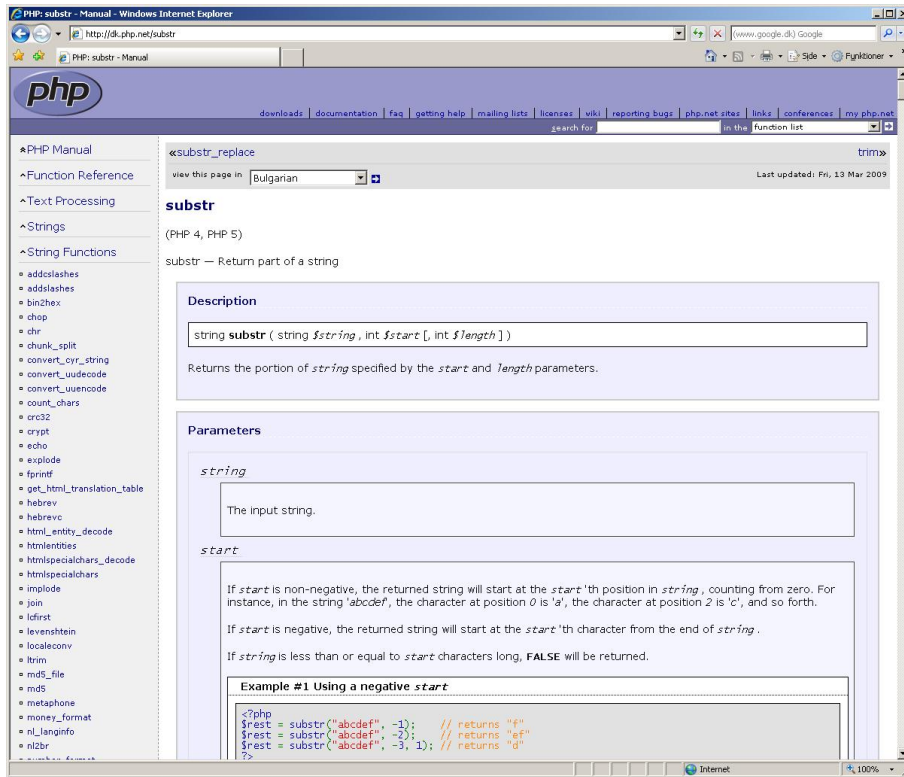


Figure 5. PHP manual.

many examples. Expert users can simply ignore the examples, since these are located at the bottom of each page.

Again, this is a huge program library, and that's not even including the Pear project⁷. This will, again, make it impossible to memorize and learn the components of the entire program library.

4.2 Environment

There exists no “de facto” standard when programming PHP. The environment could be a Linux or a Windows distribution of Apache. The editor could be the Crimson Editor⁸, JEdit, or even Eclipse⁹. These all offer the basic syntax highlighting. All this enables the programmer to use the editor and environment he/she prefers, again giving the programmer some sort of satisfaction, since one is not “forced” to use a specific tool.

Since PHP is a scripting language, compilation is usually done at run-

⁷ PHP Extension and Application Repository, see <http://pear.php.net/>.

⁸ <http://www.crimsoneditor.com>

⁹ <http://www.eclipse.org/pdt/>

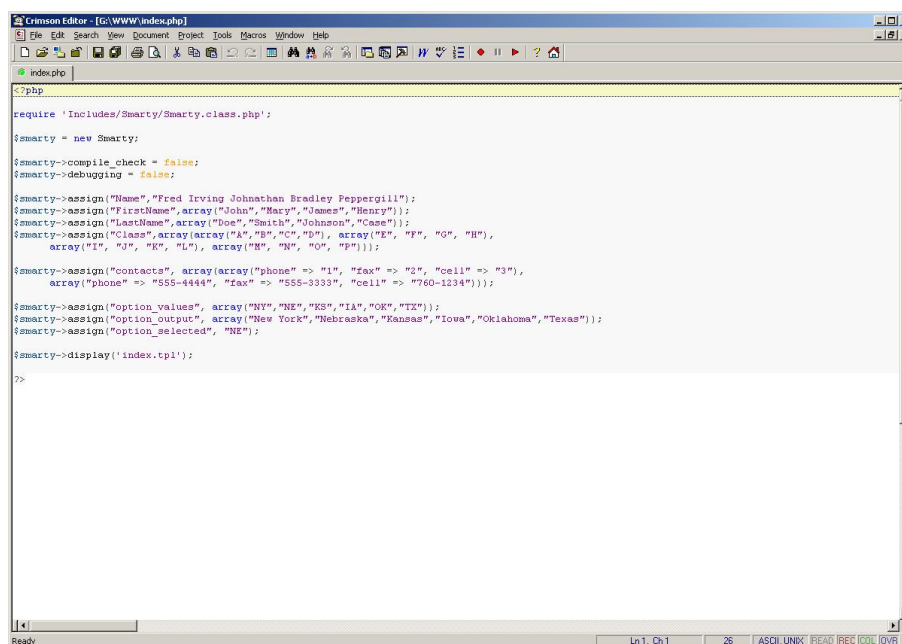


Figure 6. Crimson editor.

time¹⁰, usually as a request to a web page. The error messages supplied from this parsing and compilation are fairly informative, but it doesn't allow for redirection to where the error is located in the code. This is not as efficient as e.g. in the .NET environment.

4.3 General usability features

Programming PHP will make you capable of producing complex websites. PHP combined with the MySQL database management system¹¹ is in my opinion a good and powerful combination. But if you want to do hard computing or standalone applications, you're better off programming Java or C#. The efficiency of the PHP code is not that high.

Accessibility on the other hand is excellent for PHP. If you have a browser, you can run a PHP program (admittedly it runs on a server somewhere).

As with Java, PHP is open source and the reliability of the components is high, again caused by a mass of contributors and users.

It is one of the most used languages for web-based programming, which shows the high usability—and the overall satisfaction of the users.

¹⁰ Binary compilers do exist, see <http://www.roadsend.com/>.

¹¹ <http://www.mysql.com>

5. Recommendations for the CPH STL

So what is good usability for program libraries? I would say a good documentation and a good environment for using it. Since the documentation of the CPH STL is scattered over a large amount of different reports (which are of diverse quality) I would recommend producing an online documentation when a proper release has been completed. Preferably it should hold the same standard as the PHP manual, with loads of examples and the option for all users to contribute and ask questions. As mentioned previously, this will enable novice users to easily get acquainted with the library.

Since the CPH STL is a C++ library, the need for a specific programming environment is not crucial. For instance, both the Microsoft Visual Studio¹² as well as Eclipse support C++ compilation and syntax. Therefore sufficient editing and debugging tools exist, enabling users to have a high efficiency as well as satisfaction. Accessibility is not really an issue, since C++ compilers exist for all platforms. Furthermore, the strong advantage of the CPH STL is the efficiency of which the produced programs run. Before an actual release, the library should be thoroughly tested ensuring high reliability.

The people who at the present time are working on the project are more or less all students from DIKU¹³. I stumbled across the following statement while searching the Internet for literature on usability in coherence with component based libraries [4]: "... 'hard' algorithmic problems have a greater value in the 'reputation market' than issues of usability". This seems to be true in the case of the CPH STL too. At the current stage the program library contains a lot of code, but it isn't really usable for the mainstream public. A major clean-up and refactoring is inevitably necessary. This could be done by making the project completely open source, thereby enabling foreigners and students at other universities to contribute to the library. This would of course require some work from some sort of "leader" to keep the project together. But the benefits would be a working and usable library.

The CPH STL will most likely be a large program library as well. Why not do what none of the evaluated program libraries do? That is, make a small release. This would support novice users by making it less overwhelming. This could also be supported by a tutorial.

In order to maintain a high degree of usability in the program library, there has to be some sort of guidelines which should be followed. I suggest using "the eight golden rules" [5, Section 2.3.4]. They are some of the most commonly accepted principles for good usability in interactive designs. In the following I'll try to describe why five of them extend to program libraries:

1. Strive for consistency
 - + Variable naming
 - + Function naming
 - + Programming style

¹² <http://msdn.microsoft.com/en-us/visualc/>

¹³ The department deserves a link as well: <http://www.diku.dk/>.

- + Commenting
- + Structure
- 2. Cater to universal usability
 - + Novice vs. intermediate vs. expert users
 - + Difference between coding skills
 - + Difference in education
 - + Difference in experience
- 3. Offer informative feedback
 - + Understandable and distinct error messages
 - + The above applied to exceptions
 - + Descriptions of functionality and parameters
- 4. Prevent errors
 - + Error and exception handling
 - + Sanity check of method/function parameters (e.g. assertions)
- 5. Reduce short-term memory load
 - + Keep template arguments simple
 - + Reduce the functionality in a class to a minimum
 - + Reduce the number of overloaded methods (e.g. six similar versions of `insert`)
 - + Reduce number of method or function arguments.

To ensure that the five rules are followed, these obviously have to be outlined. This however deserves a survey of its own.

References

- [1] Mehdi Achour et al., PHP manual, <http://www.php.net/manual/en/> (1997–2009).
- [2] Department of Computer Science, University of Copenhagen, The CPH STL, <http://cphstl.dk> (2000–2009).
- [3] Microsoft Corporation, MSDN library, <http://msdn.microsoft.com/en-us/library/> (2009).
- [4] D.M. Nichols, K. Thomson, and S.A. Yeates, Usability and open-source software development, <http://www.cs.waikato.ac.nz/~daven/docs/oss.pdf> (2001).
- [5] B. Schneiderman and C. Plaisant, *Designing the user interface*, 4th edition, Addison Wesley (2005).
- [6] Sun Microsystems, Inc., Java 2 platform standard edition 5.0: API specification, <http://java.sun.com/j2se/1.5.0/docs/api/> (1994–2009).

A survey on usability of parallel programming systems

Jesper Rude Selknæs

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jaesper@diku.dk*

Abstract. In this survey the usability of parallel programming systems is investigated. The focus is on the methods used, or not used, to ensure and evaluate usability. Current efforts in the field of usability of such systems are surveyed, an experimental study to evaluating the usability of parallel programming systems is described, and the approach used is discussed.

1. Introduction

In recent years, the development of high performance computers has moved from sequential machines to parallel machines. This development is driven by two factors:

1. Hardware producers have reached some psychical limits in the field of pumping up the performance of individual processors.
2. The price of a single-machine supercomputer is incomprehensible high¹, compared to parallel systems, such as a network of workstations (NOW).

To support this move to parallel architectures, a vast number of software-libraries has been developed. This has especially been necessary because parallel systems do not share memory. However, little effort has been put into ensuring and assessing the usability of such systems. The lack of focus on usability of parallel programming systems originates from the tradition in high performance computing societies, where focus has been on performance alone [2, 5, 8].

The lack of focus on usability of parallel programming systems has given rise to a vast amount of scientific software that is badly composed. According to [11], badly composed code induces errors and decreases maintainability. Labour-intensive maintainability has moved the focus of some developers a bit more to usability [3, 10, 16]. However, as we shall see, there is still a long way to go, as methods, normally considered to be ‘state of the art’ when evaluating usability, are seldom used in the context of parallel programming systems, even though usability *is* known to be an important property of any system (see, for example, [11, p. 463]).

¹ http://en.wikipedia.org/wiki/Computer_cluster

2. Definition of usability

When considering usability of a parallel programming system, it is interesting to investigate what exactly is meant by ‘usability’. In 2008, I followed the course ‘Cluster-architecture and computing’ at DIKU. From the experience made during this course, it seems clear to me that the field can be broken into two parts:

1. Usability of the libraries themselves.
2. Understandability of the abstract parallel paradigm in question, and its use in the implementation of a given parallel system.

Both parts are equally important, but they are of such different nature that it seems probable that different techniques are to be used in assessing them.

In regard to the usability of the libraries themselves, techniques for measuring the usability of normal programming libraries and components can be applied. Unfortunately, such techniques are quite sparse, and the investigation into these techniques has apparently not been the object of many studies. Typically some heuristics are used, such as the number of program lines needed to complete some specific task [16].

The assessment of how easy a given parallel paradigm is to understand and how easy it is to implement a parallel algorithm using the paradigm in question is seldom considered. There is, however, some vague attempts (see, for example, [3]). The methods in this category are even more sketchy than those used to explore the usability of the libraries. In my opinion, techniques such as heuristics, interview of the users and observation studies must all be brought into action, to get a clear picture. This is in line with [15], which describes such methods, but with the intend that they are used on traditional user interfaces.

3. Current efforts

Up until now, developers of parallel programming systems have been concerned primarily with performance. Typically, a publication on a new parallel programming system contains a lot of performance graphs on speed-ups and CPU utilization for a number of standard benchmarks². Little effort has been made to assess the usability of the overall system. This is a consequence of the fact that the field of parallel programming is a continuation of the field of high performance computing, meaning that the people involved are interested in performance [17].

In this section, some of the relevant studies on usability of parallel programming systems will be discussed. As there is not a lot of literature in this field, literature on the usability of programming systems in general will be used to show what one could expect when considering usability in the field of parallel programming systems.

² Such as matrix multiplication, Monte-Carlo simulation, and so on.

3.1 Moving from message passing to distributed shared memory

Ever since the development of the first parallel computer systems, developers have been speculating in making a distributed shared memory (DSM) system. A DSM is a system that conceals the distributed nature of the memory spaces of parallel processors, meaning that there exists some distributed shared memory space to which all processors have access, as if it was part of their own memory space (see Figure 1).

Today there exist many such systems. Some are byte oriented, meaning that the distributed shared memory is accessible to programmers as normal memory and can be allocated and manipulated as normal memory. In this category we find systems such as ORCA [1], Munin [2], and Mirage [8]. These systems often require customized operating-system kernels or special-purpose programming languages, which means that they are in little use.

Some DSM systems are object oriented. These systems operate at a higher level, and only objects are shared. In this category we find system such as T++ [6] and CORBA [5].

Finally, some systems use an immutable data model. Here we find systems such as LINDA, PastSet, and TSPACE [5, Chapter 18]. In this category, tuples are shared. Tuples are basically objects without methods. Tuples are shared in such a way that when a process has a tuple, the tuple is not shared, but when the processor releases the tuple, other processors can gain access to that tuple.

Common to all DSM systems is that, when a DSM system is published, the authors make a great effort in describing why the system is very easy to use, but no empirical work to support these claims has been done. Of course, it is common sense that it should be easier to write parallel programs, if one can assume that the parallel processors share memory at some level.

DSM systems, in general, attempt to move from a hard-to-understand message-passing model to a more understandable DSM-based model. In other words, DSM systems do not attempt to make libraries more usable, but to move to a more understandable paradigm.

3.2 Wrapping message-passing code

A number of projects has been done in the field of wrapping message-passing code in different kind of disguises. This is done in order to hide all the difficult things of message passing, leaving the programmer with a framework that is more usable.

One such project is described in [4]. Here the goal is to design a system that allows the programmer to specify a parallel algorithm (or indeed any other algorithm) in a generic way, using skeleton functions. The system then translates this specification into appropriate parallel code, effectively hiding the message passing from the programmer. The project aims for, what the article calls, language-less programming, meaning that all handling of sharing and communication between nodes should be moved away from the

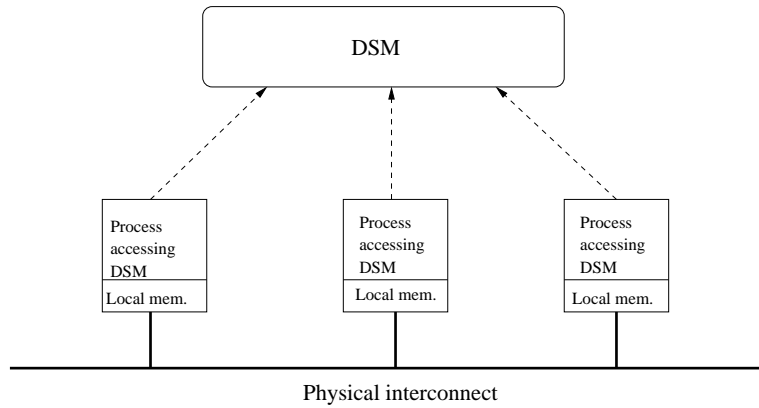


Figure 1. DSM abstraction.

programmer into the system. This would facilitate the development of GUI interfaces to the parallel programming system. This means that the system would be usable to a wide range of people, where the most obvious are the scientists in need of simulation results, or the like. Unfortunately, the project is nowhere near this state, but it seems like an interesting attempt to move away from traditional programming to a WYSIWYG environment of development. It is a pity that the articles do not contain any empirical evaluation of the usability of the system in its present state, which means that all we can conclude is that the project sounds interesting and has a potential to evolve into something very useful. Thus, this project is a real attempt to make the libraries more usable, keeping the paradigm intact.

Another project in the field of wrapping message-passing code is described in [10]. In this project, MPI-based legacy code is wrapped as Java/CORBA components. It is worth noting that this project is concerned with wrapping a specific implementation of a parallel system in such a way that it can be easily parameterized by the user. In the concrete example, a Lennard-Jones fluid simulation is wrapped, and the performance of the wrapped code is evaluated. One of the main points in this project was that we should not think of parallel systems, but problem-solving environments in general, more precisely component-based problem-solving environments (CB-PSE) as they are called in [10]. The idea is that, if we want to solve problems, for instance in the field of fluid simulations, we should develop a CB-PSE for fluid simulation that will allow the users to compose their own fluid simulation software, without concerning themselves with parallel programming, and indeed they should, ultimately, not be bordered with programming at all, as graphical user interfaces should be used to improve the programmer productivity. Having the wrapped code in Java, the development of user interfaces to the CB-PSEs should be quite easy, and this would indeed facilitate the usability of such environments. Once again, it is striking that an

article that is concerned about making systems that are usable by completely untrained users does not even mention the need for a usability evaluation. This project is a mix of making the libraries more usable, and switching to a more understandable paradigm.

3.3 The XtremeWeb project

The XtremeWeb project is a global computing system, based on the Internet [7]. This is not a traditional parallel programming system, but a grid system along the same lines as Minimum Intrusion Grid [9], in the development of which DIKU is heavily involved.

Not being a parallel programming language per se, one could ask what the system has to do in this survey? The answer to this is that it goes to show that the effort in making parallel programming systems more usable, may not evolve so much around the systems themselves, but more around the task of developing utility systems that utilize traditional libraries.

Once again we are faced with the fact that the article emphasizes the need of usability, but no empirical work is done to evaluate the usability of the system, in spite of the guidelines given in [15]. This project is a mix of making the programming interfaces more usable, and moving to a more understandable paradigm.

3.4 Use of HCI techniques

A study on developing a programming system using HCI techniques is found in [13]. The HANDS system described in [13] is not an attempt to make a parallel programming system, but the study shows that it is indeed possible to use well-known HCI techniques to improve the usability of programming systems in general.

In the HANDS system, the programming language and program libraries are considered to be the user interface to the computer. This approach is most useful to facilitate the use of traditional HCI techniques in designing the programming environment, as it is, in principle, no different from designing any other user interface.

The study even goes as far as doing an empirical experiment, and even though we are not given any quantitative results, it seems that this approach can produce programming environments with a high degree of usability, and it seems clear that it must be possible to design parallel programming systems in the same way.

Similar results can be found in [12]. This is a technical report, surveying the field of usability of programming systems for novices. The point in [12] is supported by empirical work, and basically the results in [12] coincide with the results found in [13]. So it seems safe to conclude that it should be feasible to obtain better usability for parallel programming systems, by perceiving the system as a user interface to the parallel computer, and then use HCI techniques to improve this interface.

4. An experimental approach

During the literature search for this assignment, one publication singled out, because it was the only one found that takes an experimental approach to assessing the usability of a parallel programming system [16]. This section will describe the work done in [16], and evaluate the methods used, and the results obtained.

The study [16] aims to evaluate the usability of two parallel programming systems, namely a PVM³ like system, called NMP⁴, and a more high-level, DSM-like system called Enterprise. NMP is a classical message-passing system, whereas Enterprise is a template library that automatically inserts the necessary code for communication between nodes in a distributed memory machine. Enterprise aims to use semantics similar to the one used in business organizations, to facilitate the use of parallel computing in the business world. Enterprise does not come with a specific library, users should extend template classes to implement their own classes. It seems clear that in the design of Enterprise some effort has been put into user-friendliness, whereas NMP⁵ is a message-passing system designed with performance in mind.

4.1 The experiment

The study consisted of two parts: one that examined how easy novice programmers found the use of the two systems, and one that examined how productive experienced programmers were when using the systems. In both experiments, one group using the NMP system, and one group using the Enterprise system was given the task of writing a parallel version of a computationally heavy application.

The study evaluated usability by considering five issues:

Learning curve. The speed of which a user can adapt to using the system.

Programming error. Measure how easy it is to make a correct program, i.e. how does the system support the programmer in his work?

Deterministic performance. How much overhead does the system introduce? Of course the most effective, in terms of performance, is some low-level message-passing scheme, but this may contradict productivity [5, Chapter 18].

Compatibility with existing software. How well does the system in question connect with legacy code? A formidable example on a successful project on this point is the sciPY project (see <http://scipy.org>).

Integration with other tools. How well does the system interact with other development tools, such as debuggers, editors, automatic refactoring tools, and performance-measurement tools.

Only the experiment with the inexperienced programmers was conducted in practice. A class of 15 computer-science students, at the University of

³ Parallel Virtual Machine (see [5, Chapter 18])

⁴ Network Multiprocessor Package

⁵ As all PVM-like systems.

Alberta, was divided into two groups, one solving the assignment with the NMP system, and one solving the same assignment with Enterprise.

The task was to write a parallel algorithm for computing the transitive closure of a given graph. The problem can be parallelized by letting each node in the distributed system work on a subgraph. Even though it seems that this problem is one of the ‘embarrassingly parallel problems’ many of the students had difficulties in understanding it, so a more complex problem was not a realistic idea.

The participants were given some lectures on parallel programming, and a 20-minute instruction on how to use the NMP and Enterprise systems. Also, the students were told what the experiment was about, and what was being measured. This orientation to the purpose of the experiment addressed the ethic considerations on human behaviour research, described in [14].

After the students completed their assignments, they were asked to write a two-page document to comment on the system they were assigned to.

4.2 Results

The results of the study were that, except for hours spent on the work⁶, Enterprise gave better support for the development process. As to five aspects described above, Enterprise was better than NMP. However, the variation of the results was quite large, especially the number of compilations, where some of the NMP students compiled over 200 times, excluding compilations that failed due to syntax errors.

The study showed that it was possible to measure the usability of parallel programming systems objectively. The experiment revealed that there are some fundamental concepts of parallel programming that must be understood, independently of the systems used for such programming. The experiment also supported the claim that a high-level programming system can be more usable than a low-level programming system. Finally, the student comments gave some useful feedback for the developers of Enterprise.

Finally, the study concluded that usability studies of parallel programming systems should be conducted—much more than they are done today because the results indicated that the cost of carrying out such studies can be much less than the cost of not doing them, as the latter can result in the release of a low-level programming system, which by this experiment did not yield as good results as a high-level programming system.

4.3 Soar point in the study

The study is obviously incomplete as the experiment for experienced programmers seemed to be cancelled. It would have been interesting to compare the results for the inexperienced programmers with those for the experienced ones.

⁶ This can be explained by longer compile times for Enterprise programs.

The experiment used some heuristics to measure usability. This method was properly chosen to ensure some degree of objectivity of the study, but it is a known fact that such heuristics generally are not enough to assess usability [15]. According to [15], heuristic methods must be supplemented by methods such as interviews, observational studies, and so on. The only attempt made in that direction was the comment document that the students wrote after the experiment. Such a report has the advantage of being effortless for the researchers, but it does not catch all problems encountered, and there are obvious bias problems, as the participants in such a study have a tendency to wanting to appear more clever than they really are [14].

The systems compared in this study were quite different in nature. It would have been nice if the systems compared had been more similar, so that there could have been no doubt that the systems were actually comparable. In the case of NMP versus Enterprise, the differences in the results for the two systems could appear because apples are compared to pineapples, in the sense that the two systems simply were too different to be the subject of a meaningful comparison.

The study didn't discuss the fact that an important issue in parallel programming is how well the parallel model in question is understood by the users. This aspect was neglected totally, as no data on the quality of the design of the parallel programs developed was presented. This means that we cannot tell whether the different results for the two systems originated from a better understanding of the parallel model, or from a more usable programming environment. This is especially hard, because Enterprise apparently came with a GUI-based development tool, whereas NMP was just a program library.

The size of the study, and the composition of the test group was also problematic. It seems clear that the groups were obviously biased, because all the participants shared, at least to some degree, an interest in parallel programming, and they all had roughly the same educational background, i.e. they thought alike. It also seems clear that when only 15 students participates, it is questionable whether the results say something about the participants or about the software being tested.

The results of the study were evaluated statistically, which is, of course, a nice scientific approach. However, the study said nothing what so ever about the statistical inference of the the results from the Enterprise group versus the NMP group. Confidence intervals for the differences found are presented for both groups, giving us a measure of uncertainty for the differences of the two groups. However, we really do not know if the differences shown are statistically significant or not. Such an analysis would be quite easy to perform, if we had access to the raw data, as [14, Chapter 13] tells us that the significance p can be found by calculating the value t , and then looking up t in a table. To calculate the value t , the t-test can be used, given that μ_1 and μ_2 are the mean values of the populations, n_1 and n_2 are the number of participants in each population, and S is the pooled standard deviation of the populations, we have that

$$t = \frac{\mu_1 - \mu_2}{\sqrt{(\frac{1}{n_1} + \frac{1}{n_2})S^2}} \quad (1)$$

Of course we know the means μ_1 and μ_2 , and n_1 and n_2 , because they were given in the results section of the study. The pooled standard deviation of the two distributions, S , can be computed by the formula

$$S^2 = \frac{\sum(X_1 - \mu_1)^2 + \sum(X_2 - \mu_2)^2}{n_1 + n_2 - 2}, \quad (2)$$

but the problem is that we don't know X_1 and X_2 as they were only presented graphically. So we cannot compute the significance value, based on the informations at hand. Therefore, we must conclude that it cannot be said whether the two distributions give statistically significant results. The size of the study is also problematic in this perspective, as it is questionable whether a significance analysis would make sense for such small populations.

It is a major problem that the study provided no explanation what so ever for the large variations. From a scientific point of view, it is exactly such 'strange' results that are the most interesting and literally cry out for an explanation or further studies. The authors claimed that their inexperience in conducting such studies disqualified them in coming up with an explanation, but one must ask oneself the question: If the results of a study couldn't be explained by the authors themselves, to some reasonable degree, what can we actually conclude from their results?

5. Conclusion

From the preceding sections, we can conclude the following:

- Many developers of parallel programming systems claim that they aim for a high degree of usability, but little effort is put into supporting such claims.
- It has been demonstrated that normal HCI techniques have been applied in the field of sequential programming systems, which of course indicates that such techniques can also be applied in the field of parallel programming systems.
- It has been shown that the main goal in usability design of parallel programming systems has been to make the distributed-memory model more similar to the, more normal, shared-memory model.
- Also, efforts have been made into wrapping systems, to conceal message passing from the users. Unfortunately, no real usability studies of these activities have been conducted.
- It has been shown that it is possible to do objective usability studies of parallel programming systems, even though the presented example had some shortcomings.

- It seems that the effort in the field of making parallel systems more usable is primarily concerned with making the abstract parallel system more understandable, or indeed to hide the parallel nature of such a system entirely. Not much effort has been put into the usability of the systems themselves.

References

- [1] H. Bal, M. Kaashoek, and A. Tanenbaum, Orca: A language for parallel programming of distributed systems, *IEEE Transactions on Software Engineering* **18**, 3 (1992), 190–205.
- [2] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, Munin: Distributed shared memory based on type-specific memory coherence, *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, ACM (1990), 168–176.
- [3] J. M. Bjørndalen, B. Vinter, and O. J. Anshus, PyCSP: Communicating sequential processes for Python, *Communicating Process Architectures 2007* (2007), 229–248.
- [4] J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, Parallel programming using skeleton functions, *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe, Lecture Notes in Computer Science* **694**, Springer-Verlag (1993), 146–160.
- [5] J. Dollimore, T. Kindberg, and G. Coulouris, *Distributed Systems: Concepts and Design*, 4th Edition, Addison Wesley (2005).
- [6] H. Essafi, M. Pic, M. Viala, and L. Nicolas, T++: A parallel object oriented language for a task and data parallel programming, *Proceedings of the International Workshop on Computer Architectures for Machine Perception*, IEEE Computer Society (1995), 216–220.
- [7] G. Fedak, C. Germain, V. Néri, and F. Cappello, XtremWeb: A generic global computing system, *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid*, IEEE Computer Society (2001), 582–593.
- [8] B. Fleisch and G. Popek, Mirage: A coherent distributed shared memory design, *SIGOPS Oper. Syst. Rev.* **23**, 5 (1989), 211–223.
- [9] H. H. Karlsen and B. Vinter, Minimum intrusion Grid: The simple model, IEEE Computer Society (2005), 305–310.
- [10] M. Li, O. F. Rana, and D. W. Walker, Wrapping mpi-based legacy codes as java/corba components, *Future Gener. Comput. Syst.* **18**, 2 (2001), 213–223.
- [11] S. McConnell, *Code Complete*, 2nd Edition, Microsoft Press (2004).
- [12] J. F. Pane and B. A. Myers, Usability issues in the design of novice programming systems, Technical report, Carnegie Mellon University, School of Computer Science (1996).
- [13] J. F. Pane, B. A. Myers, and L. B. Miller, Using HCI techniques to design a more usable programming system, *Proceedings of the 2002 IEEE Symposium on Human-Centric Computing Languages and Environments*, IEEE Computer Society (2002), 198–206.
- [14] R. L. Rosnow and R. Rosenthal, *Beginning Behavioral Research: A Conceptual Primer*, 5th Edition, Prentice Hall, Inc. (2005).
- [15] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd Edition, Addison Wesley Longman (1998).
- [16] D. Szafron and J. Schaeffer, An experiment to measure the usability of parallel programming systems, *Concurrency—Practice and Experience* **8**, 2 (1996), 147–166.
- [17] G. V. Wilson, Assessing the usability of parallel programming systems: The Cowichan problems, *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, Birkhäuser Verlag Basel (1994), 183–193.

Author index

Dam, Jesper A. 3

Katajainen, Jyrki i, 1

Riis, Toke B. 13

Selknæs, Jesper Rude 24