Building Sustainable Collaborative Research Software

E.S. Peer\textsuperscript{a} A.P. Engelbrecht\textsuperscript{b} F. van den Bergh\textsuperscript{c}

\textsuperscript{a}University of Pretoria, Gauteng, South Africa, espeer@cs.up.ac.za
\textsuperscript{b}University of Pretoria, Gauteng, South Africa, engel@cs.up.ac.za
\textsuperscript{c}Council for Scientific and Industrial Research, Gauteng, South Africa, fvdbergh@csir.co.za

Abstract

Building successful research software in a university environment is a challenging task which needs to address the problems of throw away research code and the short term involvement of students in any collaborative effort. This paper identifies four critical success factors for producing useful and efficient research software, namely: collaboration, sustainability, high quality and a low barrier to entry. A case study in the field of computational intelligence is used to illustrate how these success factors are satisfied by means of a multi-disciplinary approach involving open source software licensing, design patterns, aspect oriented programming, agile development techniques, the correct choice of implementation language and the identification and use of freely available software development tools and frameworks.

Keywords: Collaborative Research, Open Source Software, Design Patterns, Agile Techniques, Aspect Oriented Programming, Java, Computational Intelligence

Computing Review Categories: I.2.5, K.5.1, K.6.3

1 Introduction

University research environments pose unique constraints on the software development process. Most research projects are short lived, spanning only the duration of an undergraduate project or post graduate degree, after which, any software written for the purposes of the research is typically confined to the researcher's personal files, never to be looked at again. This phenomenon, where software is written to perform a specific once off task and then discarded, is referred to as throw away code. Unfortunately, due to its short life span and singular purpose, throw away code is usually undocumented, poorly engineered and as a result virtually impossible to maintain or extend, even if it were to find its way to a larger audience. Furthermore, traditional software development methodologies \cite{44} break down because the exploratory nature of scientific research makes it difficult, if not impossible, to define software requirements up front.

Throw away research code is both risky and inefficient. Such software does not usually undergo the same peer review process that would normally be imposed on a publication based on the results that the software produces, increasing the risk of software defects and, by extension, invalid results from going undetected. Further, validating these results is made impractical for reviewers, since they would have to reimplement equivalent software themselves. Ultimately, results are untenable and go unvalidated in a scenario where the risk of software defects is already high. Building on the existing work of others is also made inefficient by throw away code, since others must expend time reinventing the wheel instead of advancing the science. Fortunately, these problems can be addressed by having sustainable collaborative research software, shared by all parties concerned.

The following four interrelated and overlapping goals for building successful research software are identified and addressed by this paper:

- **Collaboration:** Collaborative projects create opportunities to realise efficiencies through shared resources, such as a shared software platform. Further, a collaborative project has the potential to generate higher visibility than stand-alone projects, drawing more people and resources to its cause.
- **High Quality:** It is of paramount importance that collaborative research software be reliable, in order to ensure the accuracy of academic results and the ongoing participation of collaborators. Quality is linked to collaboration in that good quality software is likely to draw more participants, due to the utility of a platform that works reliably.
- **Sustainability:** The longevity of a collaborative project should not be adversely affected by the short term involvement of student participants who are likely to only have an interest in the project for the duration of their studies. Mechanisms need to be in
place to ensure the project survives a high cycle rate of participants and thrives into the future. Long term participants are unlikely to invest time into a project that has an uncertain future.

- **Low barrier to entry:** More obstacles preventing a potential collaborator from participating directly translates into fewer active participants. Further, students need to become productive participants very rapidly, they cannot afford to waste a significant portion of their available study time just to familiarise themselves with the platform.

A case study of a research software platform, believed to satisfy the above goals, is introduced in Section 2. The remainder of the paper is dedicated to the attributes of this case study that enable it to satisfy these goals. Section 3 addresses the role that open source software licensing plays as such an enabler. Best practices employed by the developers of the software are described in Section 4. The choice of implementation language can also have an influence and is discussed in Section 5. Section 6 presents the key tools that aided in the development of the case study. Finally, this paper concludes in Section 7.

## 2 Case Study

CILib1 (Computational Intelligence Library) [35] is a collaborative effort, founded by the Computational Intelligence Research Group (CIRG)2 at the University of Pretoria3, to build research software for studying the field of computational intelligence [12, 13, 34, 50]. Specifically, CILib is intended to be a flexible framework for conducting research and experimentation within all five primary paradigms of computational intelligence, namely:

- **Swarm intelligence:** This paradigm consists of particle swarms [24, 11], ant colony optimisation [10] and related processes, which are efficient optimisers based on observations of the choreography of birds flocking and the social behaviour of ants, bees and other social organisms in nature.

- **Evolutionary computing:** The evolutionary computing paradigm consists of optimisation processes that model evolution in nature, such as: genetic algorithms [22, 19], genetic programming [26], evolutionary programming [15, 16], evolutionary strategies [37, 38], cultural evolution [40] and coevolutionary strategies [45, 21].

- **Neural networks:** By modelling neurons in the human nervous system, neural networks are able to solve unsupervised and supervised problems by clustering and modelling nonlinear multivariate relationships in data [4].

- **Artificial immune systems:** An artificial immune system [8], which is modelled on organic immune systems, is a special classifier that is particularly adept to recognising abnormal patterns in data.

- **Fuzzy logic:** Fuzzy systems enable approximate reasoning and computing using an algebra of linguistic variables, making them highly suited to deal with imprecise, or fuzzy, data [46, 47, 48, 49].

These paradigms are related in that hybrids are possible, eg. neural networks can be trained using particle swarms [43], and algorithms in the various paradigms can be used to solve overlapping problems, eg. both genetic algorithms and particle swarms can be used to solve optimisation problems [35]. Further, research in the field may require composing and extending existing algorithms in new and interesting ways. Thus, in addition to satisfying the goals for successful research software, a platform for conducting research into computational intelligence must be flexible in order to be adapted for many purposes and accommodate new requirements. The authors believe that CILib is such a platform, enabling efficient scientific experimentation with algorithms and problems, by composing them in various combinations at run time using a flexible XML (eXtensible Markup Language) [20] based configuration syntax.

In order to facilitate collaboration with third parties outside of the University of Pretoria, CIRG has chosen to license CILib under the GNU4 GPL (General Public License, refer to Section 3), the most prominent open source license in the software industry. As a result of the cooperative efforts of fourteen members5 of CIRG, CILib sports mature implementations of many swarm intelligence algorithms, an early implementation of an evolutionary computing framework, benchmark problems to test optimisation algorithms and a simulation environment for performing experiments. Various postgraduate students are currently working on improving the evolutionary computing framework and introducing a neural network framework and artificial immune systems. Nobody in CIRG is currently working with fuzzy logic. It is hoped that this paper will have the side effect of enticing third parties to get involved and fill in the gaps.

In tandem with CILib, CIRG is developing a secondary project called CiClops (Computational Intelligence Collaborative Laboratory Of Pantological Software) [35] which adds value to CILib by providing graphical tools for the set up of CILib experiments, management of parallel CILib simulations on a cluster of workstations, a repository of CILib simulation data for various configurations of algorithms and problems, and statistical analysis tools to interpret those results.

---

1http://cilib.sourceforge.net
2http://cirg.cs.up.ac.za
3http://www.up.ac.za
4GNU’s Not Unix: The operating system developed by the FSF.
5http://sourceforge.net/project/memberlist.php?group_id=72233
3 Open Source Software

Open Source Software (OSS) [36], also known as free software [42], is any software distributed under a license conforming to the open source definition as published by the Open Source Initiative (OSI)\(^6\). Unlike the OSI, which approaches OSS from a more pragmatic perspective, the Free Software Foundation (FSF)\(^7\) approaches OSS from an ethical standpoint concerning the civil liberties of software users and programmers. Essentially, free software licenses are designed to protect four basic freedoms:

- **Freedom of use:** Recipients of OSS are granted the right to use the software for any purpose.
- **Freedom to source:** Recipients of OSS are provided free access to the source code.
- **Freedom to modify:** Recipients of OSS are granted rights to prepare derivative works.
- **Freedom to distribute:** Recipients of OSS are granted rights to distribute the software, in original or modified form, either for free or for a fee.

While the OSI and FSF have somewhat different motives and are in disagreement about whether OSS should properly be called free software or *vise versa*, a common ground lies in the terms of the software licenses that they both advocate, all of which uphold the four freedoms mentioned above.

There are two primary flavours of OSS licenses: those that are copyleft\(^8\), or GPL style; and those that are not. Copyleft licenses place an additional restriction on the software, so they are less permissive and are therefore arguably less free licenses, requiring that any modifications, if distributed, must be made available under free terms again. A copyleft clause in a license essentially prevents free software from becoming non-free, which benefits the free software community as a whole even though the rights of any given individual within that community are curtailed. Copyleft licenses such as the GPL, which CILib is licensed under, level the playing field. Developers may be more willing to participate in a project licensed under copyleft terms because their efforts cannot be made non-free or exploited by any single member of the community.

OSS provides many benefits, which are aligned with the success factors identified in Section 1, to both developers and users of the software:

- **Gratis software:** Provided the software is useful, which CIRG believes CILib is, then making the software more available by offering it without fee lowers the barrier to entry for users. Developers can leverage the resulting large user base to gather and refine requirements, obtain bug reports and results of user testing as well as benefit from the reputation that accompanies a well known and successful project. Thus, by making users part of the development process, the overall quality of the software can be improved. On the sustainability front, a large user base means that there are many stakeholders in the project's success, some of which may wish to participate more actively, bringing more resources to the table to ensure its future. OSS projects with a large community are unlikely to become abandoned, particularly since the software ownership is shared by all users.

- **Peer review:** To many eyes, all bugs are shallow [36]. Since the source code is the primary means of communication between open source developers, the chances of a bug going by undetected is significantly reduced. Further, once a bug has been identified, the changes are good that the solution to the problem will be obvious to at least one of the project developers. The increased levels of peer review have an obvious positive effect on the overall quality of the code. For CILib, transparent source code access engenders the trust of the research community in the results generated by the software and lowers the barrier to entry for developers who are able to learn by example from the existing code base.

- **Cost sharing:** The ability to share resources is a strong incentive for collaborative efforts. Reuse of existing shared code which requires only minor modifications is far cheaper than developing a platform like CILib from scratch, lowering the barrier to entry for new research. Major additions to the project benefit from the existing code base to a lesser extent, however, they still derive the full benefits that an existing large community brings to the table, including peer review and user feedback.

- **Reputation rewards:** Being involved in a prestigious project is a strong incentive for participation. Although the sustainability of a successful project is catered to by other aspects of OSS, the pride associated with being on the project team may play a role in motivating some developers to remain or become involved with the project, even if they are not direct stakeholders in the success of the project. From a university perspective, a highly regarded collaborative project may be able to draw research funding from investors.

- **Longevity:** The economics of OSS is an interesting field [36, 28], even though the software itself is distributed free of charge, there are still opportunities to profit from secondary markets, such as training, support and other services related to the software. In fact, these secondary markets typically would not exist without the software being freely available in the first place, or at least, they would be significantly smaller markets. Specifically, CIRG has the opportunity to license its copyrighted portions of CILib to commercial entities under non-free terms in a strategy known as dual licensing. In addition, CIRG may wish to provide access to CiClops as a service to third parties for

\(^6\)http://www.opensource.org
\(^7\)http://www.fsf.org
\(^8\)http://www.fsf.org/licenses/licenses.html#WhatIsCopyleft
a fee, enabling paid for resources to be allocated to advancing the project into the future. Further, students can legally make use of CILib in industry and benefit from any secondary markets it introduces, since they retain their rights to CILib granted them under the GPL, meaning they may remain stakeholders in the project for a longer term than the duration of their studies. As a side effect, if students intend to make use of the software later in their careers, then they have an incentive to produce better quality work. Of course, any other stakeholders may also benefit from these secondary markets. Thus, the sustainability of a large OSS project with many stakeholders is reasonably assured.

4 Methodology

OSS is probably the most important factor contributing to CILib's suitability as a sustainable collaborative research platform, however, the methodology and best practices employed by the developers undeniably plays a non-trivial role too. This section discusses the role design patterns, aspect oriented programming, agile techniques and living documentation play in reducing the complexity of the project, implicitly improving quality and driving the barrier to entry for developers lower.

Design patterns

Good software design is difficult to accomplish, particularly for novice programmers, usually requiring a number of redesign iterations. Design patterns [18] alleviate this problem, encapsulating the knowledge of experienced programmers in the form of pattern catalogues, by specifying proven solutions to regularly occurring software design scenarios. These patterns are not specifically planned or invented, rather, they are discovered by observing best practices and recurring design solutions that have proven to be useful, efficient, and extensible in existing software. Thus, software programmers are able to draw upon the mature and successful designs of others with more experience.

The Gang of Four [18], as the pioneers of the field are usually referred to, presented a catalogue identifying a number of general object oriented programming patterns. In addition, specific catalogues pertaining to various scenarios have since been published, including high level software architecture design [6, 17], distributed systems and concurrency [41], database programming [33], as well as catalogues specific to certain programming languages and frameworks [2, 32].

Each entry in such a catalogue consists of four essential components. Firstly, a short descriptive pattern name, which collectively define a vocabulary for communicating about entire designs at a high level of abstraction. Second, an outline of the problem context specifies when it is appropriate to apply the pattern. Thirdly and most importantly, the solution to the problem is described, usually along with an example demonstrating the use of the pattern. Finally, the impact and any known consequences of the pattern are listed.

Software implementing design patterns not only benefits from the expert experience derived from the patterns, but the patterns themselves also implicitly serve as documentation. Scholars of design patterns should be able to understand the design of such software with little more documentation than a reference to the applicable pattern and a brief explanation of any unusual implementation details. Furthermore, programmers unfamiliar with design patterns can simply refer to the catalogue where the design is discussed in detail.

The self documenting nature of design patterns reduces the barrier to entry for developers and increases the chances for collaboration. Design patterns can be taught to students during their undergraduate studies in order to further reduce the learning curve required to participate during postgraduate studies. Finally, because the patterns are a reflection of good design choices by experienced programmers, the quality of software based on appropriate design patterns benefits as an obvious side effect.

Aspect oriented programming

Object oriented programming can reduce code complexity by localising inter-dependent code, however, some types of related functionality, known as aspects, cannot be cleanly encapsulation into a single component. For example, synchronisation of concurrent objects, database persistence, transaction management, logging and security are all aspects that cross-cut unrelated functionality of a system, resulting in tangled code that is unnecessarily complex and difficult to maintain. Aspect Oriented Programming (AOP) [25] strips these aspects out of the code, localising related functionality, so that it can be woven into the code externally using an aspect weaving language.

For example, using an aspect weaving language such as XDoclet (refer to Section 6), authorisation checks can be applied to code after the fact, so that a security aspect is automatically executed before each method call by simply associating the aspect with the appropriate methods by means of a source code annotation. Thus, the code is kept clean of any security functionality, only concerning itself with the business logic at hand, resulting in reduced complexity and as a direct consequence a lower barrier to entry. Further, since all code related to an aspect is localised in a single place, aspects are easier to test and maintain, ultimately resulting in higher quality code when used appropriately. For example, CiClops makes use of the database persistence, security and transaction management aspects provided by a J2EE (refer to Section 6 container, which are provided by a third party, well tested and of a high quality. Most importantly, the CiClops code base is not littered with any database API calls, security checks or complex transaction management code.
Agile techniques

The CILib developers have adopted a number of agile development practices from eXtreme Programming (XP) [3], including:

- **Simplicity and refactoring**: Researchers are not always able to articulate system requirements up front, needing time to experiment and explore with different implementations until something that works is discovered. Thus, the XP axiom of keeping the code as simple as possible to satisfy the requirements of the day is applicable. Over designing for something that may or may not prove useful in the future is a waste of development resources, however, this approach requires regular refactoring of the code to be performed as commonalities and better ways to do things are discovered. Keeping the system simple for as long as possible reduced the barrier to entry for new participants. Also, there is less to go wrong in a simple design, making higher quality software more likely.

- **Testing**: Automated unit testing undeniably improves the quality of the code base, since defects are likely to be detected and corrected. However, the exploratory nature of scientific research presents interesting challenges. For example, if a new optimisation algorithm is being developed, where the performance of the algorithm is the subject of an empirical study, how does one go about defining a suitable test? The correct performance of the algorithm is not known and the only way to determine its performance in to execute the untested code. Of course, once the performance of the algorithm is known, then regression tests can be written to check that the implementation does not get broken during routine code maintenance. Regression testing increases developer confidence, since the tests can be executed at any time to determine whether a recent change breaks the system. Further, the tests themselves constitute documentation for the system, since they define the expected behaviour of the code, again lowering the barrier to entry.

- **Pair programming**: Pair programming, which provides immediate peer review of code and enables developers to learn from others with more experience, is not always applicable in a research environment where students are working on their own projects. However, CILib has benefited from a variant of pair programming during regular developer workshops, where a group of developers have accompanied a developer in a hot seat around data projector. The developer in the hot seat is swapped out from time to time, typically when one of the other programmers around the table figures out a better method to solve the problem at hand or if the developer in the hot seat requires a break. This technique improves team morale, catches defects early, enables new participants to learn from others and generates new and inspiring ideas.

Living documentation

Static documentation can very easily become out of synchronisation with a constantly changing code base and incorrect documentation can be worse than having no documentation at all.

The documentation implicitly encapsulated by unit tests is by definition always up to date, since the moment the code deviates from the encapsulated requirements, the automated tests fail and something must be done to rectify the situation.

Tools such as Javadoc (refer to Section 6) extract documentation from the structure of the code itself, augmenting it with the comments in the source. The proximity of the comments to the code improves the chances that comments will be maintained compared to external documentation, however, care must be taken to keep comments in synchronisation with the code. At very least, the structure of the code is always correctly captured by Javadoc.

Although existing documentation encompassed by research papers and dissertations based on the project are static, any changes to the code necessitated for the research should be documented. Thus, the body of documentation will constantly increase in size, in effect becoming a living body of documentation as a whole. Even if some of the documentation becomes outdated, more recent documents should be available for consultation.

Good and up to date documentation has an obvious effect of lowering the barrier to entry. Further, the better a developer understands the platform, the lower the probability of making a mistake that adversely affects the quality of the software.

5 Language Choice

For reasons that will become apparent in this section, Java was chosen as the implementation language for CILib.

Java is a modern, high level, general purpose, object oriented programming language [14, 23]. Programs written in Java are compiled into an intermediate language, known as byte code, which is interpreted at run time by a Java Virtual Machine (JVM).

The Java platform is not OSS and the methodologies discussed in the previous section are agnostic to the choice of language, however, Java does present a unique set of benefits over other language choices in the context of building open source collaborative research software, including platform and vendor independence, an extensive reusable API (Application Programming Interface), garbage collection, free availability of high quality tools as well as excellent run time performance.

Platform and vendor independence

A cornerstone of Java has always been the concept of write once, run anywhere. This goal has been achieved by virtue of the JVM, since only the underlying virtual machine need be ported to each platform where Java is supported.
Supported platforms include Windows\textsuperscript{9}, Linux\textsuperscript{10} and MacOS\textsuperscript{11}, reducing the barrier to entry since any user can be accommodated. Further, the Java specification is guided by the Java Community Process (JCP)\textsuperscript{12}, giving multiple vendors the opportunity to contribute and participate in the decisions that dictate the future direction of Java. Competing JVM implementations are available from multiple Java vendors, including Sun Microsystems\textsuperscript{13}, IBM\textsuperscript{14}, BEA\textsuperscript{15} and the Blackdown project\textsuperscript{16}, ensuring diversity in the market place and the future longevity of the Java platform. A completely free JVM implementation is also being worked on by the GNU Classpath community\textsuperscript{17}, along with a native Java compiler as part of the GNU Compiler Collection (GCC).

Java foundation classes

The Java platform, which is guaranteed to be available on any compliant JVM, is defined in terms of the Java Foundation Classes (JFC). The JFC, or Java APIs, provide XML processing, Input/Output (I/O), Graphical User Interface (GUI), networking and numerous other services to applications. Further, with the introduction of generics in Java version 1.5, a type safe collections framework using templates has been provided [5]. The JFC have been through many revisions, gradually improving their design, which is heavily based on design patterns. Specifically, Java’s reflection API, which is required for implementing the Inversion of Control (IoC) design pattern [31], is a fundamental reason Java was chosen over C++ as the implementation language for CILib. The IoC pattern is a key enabler for building flexible component based software. Finally, the Java APIs are well tested, improving the reliability of code that uses them.

Garbage collection

Garbage Collection (GC) relieves a programmer from having to explicitly manage memory deallocation, resulting in safer code due to the reduced risk of introducing difficult to find memory leaks. GC is associated with at least some performance overhead, since an additional process must be executed from time to time to recycle unreferenced memory. Counter intuitively, in spite of this overhead, GC can have a net increase in the performance of an application\textsuperscript{18}. For example, heap compaction performed by GC increases the likelihood of cache hits. Further, since GC only executes when memory is tight, programmes with a low memory footprint may never need to run a GC cycle. Another factor to consider is that smart pointer based reference counting techniques, which are typically employed to simplify memory deallocation in non-GC languages, can carry a much higher overhead than GC, since counters need to be updated for every assignment. Worse, reference counting techniques are dangerous because they cannot deal with circular references or anonymous objects. Finally, explicit destructors can be a significant performance overhead for stack allocated resources.

Tool support

There are many high quality development tools and frameworks freely available (refer to Section 6) which can dramatically reduce the barrier to entry for developing Java software. Further, the proper use of good tools can improve the quality of the software, by imposing certain best practices on a programmer.

Performance

Java is still plagued by the stigmatism of poor performance due to early and immature implementations of the JVM. This situation is further exacerbated by the intuition that interpreted languages with additional GC overheads must have inferior performance to natively compiled languages. Modern HotSpot [1] JVMs, however, have dramatically improved the performance of Java, to the point where it is comparable and in certain circumstances superior in performance to natively compiled languages such as C/C++ [9, 39]. HotSpot JVMs sport state of the art generational GC algorithms, speculative run time optimisation using dynamic profiling, and Just In Time (JIT) compiling of critical code, known as hot spots, to instructions optimised for the local processor. Numerous microbenchmarks [29, 30, 7, 27] have been conducted, which show Java performance to be on par with other languages.

In the context of computational intelligence, high performance is an essential aspect of the quality of software, since simulations are typically computationally complex, sometimes requiring hours or even days of processing time. For this reason, when the initial author of CILib contemplated implementing the code in Java, testing was performed using a particle swarm optimiser as a benchmark. This benchmark, originally implemented as throw away code, later became known as NastyPSO\textsuperscript{19}, by virtue of the fact that it was a quick and dirty hack at the time to determine how Java measured up. Table 1 illustrates the performance of the NastyPSO benchmark implemented in three programming languages. The three implementations are absolutely identical, barring syntactic differences in the languages. Time was measured using the Unix time command which measures the actual amount of time that each process was scheduled and is thus invariant to the load on the test machine, which was an Intel Pentium4 architecture. Various compiler optimisations

\begin{table}[ht]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Language & Time (s) \\
\hline
C/C++ & 1.37 \\
Java & 1.49 \\
Java HotSpot & 1.10 \\
\hline
\end{tabular}
\caption{NastyPSO Benchmark Results}
\end{table}

\footnotesize
\textsuperscript{9}http://www.microsoft.com/windows/default.mspx
\textsuperscript{10}http://www.linux.org
\textsuperscript{11}http://www.apple.com/macosx/
\textsuperscript{12}http://www.jcp.org
\textsuperscript{13}http://www.sun.com
\textsuperscript{14}http://www.ibm.com
\textsuperscript{15}http://www.bea.com
\textsuperscript{16}http://www.blackdown.org
\textsuperscript{17}http://www.gnu.org/software/classpath/classpath.html
\textsuperscript{18}http://www.digitalmars.com/d/garbage.html
\textsuperscript{19}http://cilib.sourceforge.net/NastyPSO


\normalsize
Table 1: NastyPSO Performance

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler / VM</th>
<th>Time (seconds)</th>
<th>Time (relative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>Intel Compiler 7.1.006 (-O3 -march=pentium4)</td>
<td>71.1</td>
<td>0.69</td>
</tr>
<tr>
<td>C++</td>
<td>Intel Compiler 7.1.006 (-O3 -march=pentium4 -mp)</td>
<td>102.4</td>
<td>1.00</td>
</tr>
<tr>
<td>Java</td>
<td>Sun HotSpot VM 1.4.2.06 (-O, -server)</td>
<td>104.3</td>
<td>1.02</td>
</tr>
<tr>
<td>Java</td>
<td>Sun HotSpot VM 1.5.0.01 (-O, -server)</td>
<td>105.0</td>
<td>1.03</td>
</tr>
<tr>
<td>Java</td>
<td>Blackdown HotSpot VM 1.4.2.01 (-O, -server)</td>
<td>105.3</td>
<td>1.03</td>
</tr>
<tr>
<td>Java</td>
<td>IBM VM 1.4.1 (-O)</td>
<td>110.3</td>
<td>1.08</td>
</tr>
<tr>
<td>C++</td>
<td>GNU Compiler 3.3.5 (-O3 -march=pentium4)</td>
<td>137.7</td>
<td>1.34</td>
</tr>
<tr>
<td>C#</td>
<td>Mono 1.0.1</td>
<td>188.0</td>
<td>1.84</td>
</tr>
<tr>
<td>Java</td>
<td>Sun HotSpot VM 1.5.0.01 (-O, -client)</td>
<td>203.1</td>
<td>1.98</td>
</tr>
<tr>
<td>Java</td>
<td>Sun HotSpot VM 1.4.2.06 (-O, -client)</td>
<td>203.3</td>
<td>1.99</td>
</tr>
<tr>
<td>Java</td>
<td>Blackdown HotSpot VM 1.4.2.01 (-O, -client)</td>
<td>204.4</td>
<td>2.00</td>
</tr>
<tr>
<td>C++</td>
<td>GNU Compiler 3.3.5</td>
<td>208.8</td>
<td>2.04</td>
</tr>
<tr>
<td>Java</td>
<td>BEA WebLogic JRockit 1.5.0 (-O)</td>
<td>221.3</td>
<td>2.16</td>
</tr>
</tbody>
</table>

and virtual machines were tested. Accuracy is another important aspect of the quality of computational intelligence software. Thus, the relative time column is measured relative to the simulation compiled with the Intel compiler’s maintain precision (-mp) option, forcing the compiler omit optimisations that could produce inaccurate results. The server HotSpot JVMs, which perform more aggressive optimisation than the client HotSpot JVM for long running processes, all perform roughly on par with the Intel compiled code, provided it was forced to maintain precision. In Java’s favour, the Intel compiler is not free software, making it less accessible and increasing its barrier to entry. Finally, Java significantly outperforms C++ code compiled using the free GNU compiler, even when aggressive optimisations are used.

The source code for the NastyPSO benchmark is available for peer review under GPL from the CILib project page at SourceForge\[^{20}\].

## 6 Tools

This section briefly mentions some of the tools and frameworks utilised by CIRG developers. Barrier to entry is reduced through freely available and well documented tools. Further, overall quality is improved by virtue of the tools implicitly guiding their users into utilising best practices. The level of integration and cooperation between these tools and frameworks is astonishing, however not altogether surprising, since many of them boast the hallmarks of successful collaborative open source projects themselves.

### Ant

Ant\[^{21}\] enables the scripting and automation of the Java build process. Tasks and their dependencies can be configured in an easy to use XML document and may include compiling sources, generating documentation, weaving aspects, executing automated unit tests and synchronising with a version control repository amongst others.

### JUnit

JUnit\[^{22}\] is a Java unit testing framework that integrates tightly with the Ant build process. JUnit makes testing trivial while supporting features such as fixtures, which enable multiple tests to be performed on the same set of classes while being completely isolated from each other.

### Javadoc

Javadoc is distributed as part of the standard Java SDK (Software Development Kit), making it accessible to all Java developers. Browseable HTML (HyperText Markup Language) documentation is generated by Javadoc from the structure of Java classes and special comments written in the code.

### XDoclet

XDoclet\[^{23}\] also integrates with Ant, enabling AOP through attributes embedded as special Javadoc comments in the source code. Later versions of XDoclet are likely to support the new source code annotation features in Java version 1.5.

### J2EE

J2EE\[^{24}\] (Java 2 Enterprise Edition) enables software to utilise enterprise features such as managed persistence and transactions. High quality open source implementations of J2EE are available, including JBoss\[^{25}\], JonAS\[^{26}\] and

\[^{20}\]http://www.sourceforge.net
\[^{21}\]http://ant.apache.org
\[^{22}\]http://www.junit.org
\[^{23}\]http://xdoclet.sourceforge.net
\[^{24}\]http://java.sun.com/j2ee/
\[^{25}\]http://www.jboss.org
\[^{26}\]http://jonas.objectweb.org/
Geronimo\[^{27}\].

**CVS**

CVS\[^{28}\] (Concurrent Versions System) enables multiple developers to cooperate on version controlled source code repository. While CVS is tried and tested, more feature rich version control tools are now freely available, such as Subversion\[^{29}\], however, they are still not supported by SourceForge.

**Eclipse**

Eclipse\[^{30}\] is a fully featured Java IDE (Integrated Development Environment) which supports folding, syntax highlighting, code completion and powerful refactoring tools. Since Eclipse is itself written in Java, it is also platform independent.

**SourceForge**

SourceForge\[^{31}\] facilitates the development of open source projects by providing a community based collaborative environment. Project services, such as a central CVS repository, bug tracking and mailing lists, are provided free of charge by SourceForge to open source developers. Finally, the visibility created by having a project hosted by SourceForge may help to attract third party collaborators.

7 Conclusion

In this day and age of limited resources and high expectations of those resources, leveraging the efficiencies of collaborative research is of paramount importance. Four critical success factors were identified for building successful research software in an environment of throw away code and high cycle rate of student developers, namely: collaboration, high quality, sustainability and a low barrier to entry.

OSS licensing is a key enabler that directly addresses all four of these goals. Collaboration is an inherent feature of OSS, with high quality being achieved through a superior peer review process. Community ownership of the source code amongst many stakeholders ensures an OSS project’s longevity and barriers to entry are broken down because the software is freely available to anyone.

Design patterns, aspect oriented programming, agile techniques and living documentation further reduce the barrier to entry by reducing complexity, while simultaneously improving the quality of the software.

While the choice of language is certainly not a prime factor in the success or failure of a research software platform, many features of Java play into the identified critical success factors. Platform independence reduces the barrier to entry, allowing users of any operating system to participate. Further, the future of the Java platform is not tied to the success or failure of a single vendor. The JFC play a role in increasing software quality and reducing the barrier to entry for developers, by providing high quality APIs. Garbage collection further reduces the risk of software defects while reducing responsibilities of developers, and hence the barrier to entry. Open source development tools and frameworks further contribute to lowering barriers to entry and improving the overall quality of the software. Finally, if performance is taken as one of the metrics used to measure the quality of software, then Java measures up exceptionally well to its competitors.

Acknowledgement

The financial assistance of the National Research Foundation towards this research is hereby acknowledged. Opinions expressed in this paper and conclusions arrived at, are those of the authors and are not necessarily attributed to the National Research Foundation.

References


\[^{27}\]http://geronimo.apache.org/

\[^{28}\]https://www.cvshome.org/

\[^{29}\]http://subversion.tigris.org/

\[^{30}\]http://www.eclipse.org

\[^{31}\]http://www.sourceforge.net


[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


