Multiscale Spatiotemporal Visualisation

STREP # FP7-248032

MSV Deliverable

D5.1 – Review of available collaborative software tools
Work package 5: Shared implementation

Stephen Aylward (KIT)
Julien Finet (KIT)

31/01/2012

SEVENTH FRAMEWORK PROGRAMME
Multiscale Spatiotemporal Visualisation
Collaborative software tools review

DOCUMENT INFORMATION

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>Acronym</th>
<th>MSV</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP7-248032</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full title</th>
<th>Multiscale Spatiotemporal Visualisation: development of an open-source software library for the interactive visualisation of multiscale data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project URL</td>
<td><a href="http://www.msv-project.eu">http://www.msv-project.eu</a></td>
</tr>
<tr>
<td>EU Project officer</td>
<td>Frank CUNNINGHAM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Review of available collaborative software tools</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Work package Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Shared Implementation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of delivery</th>
<th>Planned</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12-2011</td>
<td>31-01-2012</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status</th>
<th>Version v3</th>
<th>final</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Nature</th>
<th>Prototype</th>
<th>Report</th>
<th>Demonstrator</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
<th>Consortium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors (Partner)</th>
<th>Stephen Aylward (KIT), Julien Finet (KIT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsible Author</td>
<td>Stephen Aylward</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:Stephen.aylward@kitware.com">Stephen.aylward@kitware.com</a></td>
</tr>
<tr>
<td>Partner</td>
<td>KIT</td>
</tr>
<tr>
<td>Phone</td>
<td>+01-919-969-6990</td>
</tr>
</tbody>
</table>

Abstract (for dissemination)
The product of this task is a summary of the best-in-class software infrastructures of other successful collaborative development efforts. Other efforts to be considered include KIT’s VTK, ITK, NA-MIC’s Slicer, Osirix, University of Heidelberg’s MITK, BrainLab’s VVLink, OpenMAF and GIMIAS.

Keywords
Open-source, Testing, Documentation, Software Engineering

<table>
<thead>
<tr>
<th>Version Log</th>
<th>Rev No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Date</td>
<td></td>
<td>Author</td>
<td>Change</td>
</tr>
<tr>
<td>21/12/2011</td>
<td>V1</td>
<td>Stephen Aylward</td>
<td>First draft with all contents</td>
</tr>
<tr>
<td>26/01/2012</td>
<td>V2</td>
<td>Stephen Aylward</td>
<td>Suggestions from Dr. Testi</td>
</tr>
<tr>
<td>30/01/2012</td>
<td>V3</td>
<td>Stephen Aylward</td>
<td>Details from NA-MIC added</td>
</tr>
</tbody>
</table>
Multiscale Spatiotemporal Visualisation
Collaborative software tools review

Project Consortium Information

Disclaimer: This document is property of the MSV Consortium. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.
List of definitions

GUI     Graphical user interface
API     Application programmers interface
MSV     Multi-scale spatio-temporal visualization
# Table of Contents

1. Introduction .................................................................................................................. 6

   1.1. Overview of WP5 ........................................................................................................ 6

       1.1.1. T5.1: Review tools for collaborative software development infrastructure .......... 6

       1.1.2. T5.2: Demonstrate the infrastructure to ensure interoperability ...................... 6

       1.1.3. T5.3: Report on the collaborative software development infrastructure of the MSV .... 7

1.2. Objective and organization of this document ................................................................ 7

2. Software license as a tool for collaboration ..................................................................... 8

3. Defining a toolkit’s scope .................................................................................................. 9

4. Software Repository ........................................................................................................ 11

5. Community quality standards .......................................................................................... 13

6. Assessing contribution quality .......................................................................................... 14

   6.1. Algorithm quality ........................................................................................................ 14

   6.2. Implementation quality ................................................................................................ 15

       6.2.1. Coding style ........................................................................................................... 15

       6.2.2. Cross-platform building and packaging ............................................................... 16

   6.3. Documentation quality ............................................................................................... 17

   6.4. Backward compatibility ............................................................................................. 18

       6.4.1. Backward compatibility policies ........................................................................ 18

   6.5. Tests and Coverage ..................................................................................................... 18

       6.5.1. Regression tests ...................................................................................................... 19

       6.5.2. Dynamic tests ........................................................................................................ 19

       6.5.3. Static Tests ............................................................................................................... 20

       6.5.4. Code coverage ........................................................................................................ 20

   6.6. Reporting ...................................................................................................................... 21

7. Software process workflows .............................................................................................. 23

8. Conclusion ........................................................................................................................ 24
1. Introduction

1.1. Overview of WP5

The objective of WP5 is to establish the collaborative software development infrastructure needed by the open-source MSV toolkit. The process will be driven by the need for the toolkit to provide a reference implementation of the technical foundations of MSV as described in WP2 (shared vision) and WP4 (best practice) and to include demonstration applications to address the common, critical needs from the use-cases described in WP3 (exemplary problems). The software infrastructure of WP5 will integrate those guidelines into leading collaborative development tools to establish a cohesive environment that facilitates the development, documentation, validation, maintenance, and distribution of the MSV toolkit while ensuring adherence to those guidelines.

The practices of other successful collaborative software development efforts will be reviewed, documented, and selectively demonstrated to produce a report that defines the collaborative software development infrastructure to be used by the MSV.

Emphasis will be placed on establishing a software development workflow that suits the parties involved and ensures constant progress towards the goals, as documented in WP2, WP3 and WP4, of the MSV toolkit.

Additional emphasis will be placed on establishing an infrastructure that supports software development transparency that, in turn, fosters community acceptance and support.

1.1.1. T5.1: Review tools for collaborative software development infrastructure

The product of this task will be a summary of the best-in-class software infrastructures of other successful collaborative development efforts. Other efforts to be considered will include KIT’s VTK, ITK, NAMIC’s Slicer, Osirix, University of Heidelberg’s MITK, BrainLab’s VVLink, MAF, and GIMIAS.

The collaborative software development tools that we anticipate using include the following: CMake for cross-platform building, CDash for regression testing and algorithm validation, Cov for code coverage monitoring, Doxygen for documentation, and wrapping for use from Java, TCL, and Python.

A report (D5.1) on the best-in-class tools will be circulated to the MSV partners. The report will conclude with a recommendation of the tools to be employed. Feedback will be gathered, and consensus on the infrastructure will be resolved.

1.1.2. T5.2: Demonstrate the infrastructure to ensure interoperation

The infrastructure chosen from T5.1 will be established, in prototype, to ensure its practicality (M5.1 and D5.3). One important consideration is the interoperation of collaborative tools, and experimentation via a prototype is paramount to ensuring the feasibility and interoperation of the components of the recommended approach. Once implemented, the prototype will be reviewed for transparency. Process transparency within software development facilitates community acceptance and encourages others to contribute back to the toolkit. Via transparency and subsequent community involvement, maintenance costs are nearly eliminated, and the toolkit inherently maintains its status as being state-of-the-art.

Lessons learned during the installation, integration, configuration and use of these tools will be documented throughout this process. That documentation will feed into T5.3.
1.1.3. T5.3: Report on the collaborative software development infrastructure of the MSV

A final report (D5.2) will be generated that defines the MSV’s software development infrastructure, documents its installation and configuration procedures, and recommends a course for the use and maintenance of that infrastructure. A gap-analysis will be included to recommend further infrastructure development needs and courses for meeting those needs.

1.2. Objective and organization of this document

This document (T5.1) describes the issues and options that must be considered when establishing a collaborative software development environment. These topics range in scope from broad (e.g., selecting a license at the start of the project, Section 2) to detailed (e.g., the automated assessment of what portion of the lines of code are covered by unit tests, Section 3.3). Best practices are given via examples from other collaborative communities. The document begins by considering how the licensing terms and scope of a toolkit define its potential collaborative environment. The document then discusses the issues and options for handling contributions while maintaining community standards via collaborative tools and software processes.
2. Software license as a tool for collaboration

The software license is perhaps the most influential “tool” that software developers have for declaring who they would like to participate as collaborators.

- “Closed source” licenses impede collaboration by limiting the development of derivations and the exchange of knowledge to the outer layer (GUI and API) of the toolkit.
- “Copyleft” open-source licenses restrict collaborations to those who are also willing to make their derived efforts open source. One of the most well-known copyleft license is the GPL (Gnu Public License).
- “Dual” open-source licenses specify who may use the software for free and who has to pay to use the software. A popular instance of this license is the QPL (Qt Public License). It is no longer used by Qt, but it is still used by others. QPL allows for free use in academia and requires the purchase of a license if the software is used in a commercial product. One challenge with such licenses is that private universities (e.g., Duke, Harvard, Yale) are commercial, for-profit enterprises. It has been conjectured that, if brought before a court of law, conducting grant funded research at these private universities may be considered a commercial activity and require commercial software licenses.
- “Permissive” open-source licenses allow for the unrestricted use of the software in commercial and non-commercial applications. Popular examples of this type of license include “BSD-Style” licenses and the Apache 2.0 license.

Licenses also provide an opportunity for collaborators to take steps towards reducing their liability regarding

1) using the software in a manner that causes harm to people or property
   - The license should declare that the software is offered “AS-IS” and without guarantee of suitability for any use.
2) enabling the infringement on patented technologies
   - The license should include a statement that no effort has been made to ensure that the software does not infringe on patents
3) incorporating contributions from others without fear of integration of patented technologies or incompatible licenses.
   - The license should state that anyone who makes a contribution back to the library is providing a free and unrestricted license to use their contribution for any purpose and providing a free and unrestricted license to any patents that they have knowledge of and that are embodied by their contributions.

One of the most comprehensive and yet easy-to-understand licenses which covers the above liability and contribution considerations as well as provides permissive open-source use is the Apache 2.0 license. The Apache 2.0 license has been adopted by VTK and ITK. Historically, the ITK and VTK developers have reported that many of the major code bases and challenging bug fixes contributed to their toolkits came from commercial collaborators. Commercial collaborators tend to provide a level of development, documentation, and testing rigor that goes beyond what can be afforded by most academic groups. That rigor benefits the quality and longevity of the toolkit. More information on licenses is provided in the D6.1 document of the MSV consortium.
3. Defining a toolkit’s scope

Along with licensing, the declared scope of a toolkit is another major influence on the potential collaborators for a toolkit. Several excellent collaborative tools have been developed for forming, communicating, and updating that scope.

The declaration of scope typically begins with the specification of use-cases and taxonomies, and Wikis are an excellent tool for creating and maintaining such documents. New versions of wikis are quite powerful, with the ability to post mathematical equations, images, and rich formatting. “Watching” a wiki page allows a user to automatically receive an email when that page is updated, thereby maintaining progress on those documents. Wikis are preferred over GoogleDocs and Microsoft Word documents because they offer the greatest transparency to others and also enable the broadest participation from the community.

After or simultaneous with the identification of use-cases and taxonomies, it is necessary to define the tests which will be used to define and demonstrate the success of the toolkit. Typically these cases should represent the depth and breadth of the problem domain for users as well as the range of technical hurdles that the developers must address. Furthermore, as the toolkit is adopted by others and its failings and successes are realized, it is necessary to maintain the data associated with those cases. That is, data is the driving force behind the toolkit and the tangible evidence of the utility of the toolkit – it is one of the most important resources throughout the life of a toolkit. Three popular, open-source solutions for maintaining data online are:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XNAT</td>
<td>XNAT is an open source imaging informatics platform, developed by the Neuroinformatics Research Group at Washington University. It facilitates common management, productivity, and quality assurance tasks for imaging and associated data. Thanks to its extensibility, XNAT can be used to support a wide range of imaging-based projects. XNAT is Fully Customizable: The &quot;extensibility&quot; of XNAT does more than give it its name, it is the application's reason for being. You can configure XNAT in any number of different ways to support your data management and project management needs. You can also access and control the application in any number of ways by using our REST API.</td>
</tr>
<tr>
<td>MIDAS</td>
<td>MIDAS integrates multimedia server technology with Kitware's open-source data analysis and visualization clients. The server follows open standards for data storage, access and harvesting. MIDAS has been optimized for storing massive collections of scientific data and related metadata and reports. MIDAS is available under a non-restrictive (BSD) open-source license. A variety of data access methods are provided including web, file system, and DICOM server interfaces. Local, centralized and distributed batch processing is provided via an extensible scripting language that has been specialized for statistical hypothesis testing. Kitware can customize and install MIDAS at your site or host your data and provide computational resources using our own MIDAS installation.</td>
</tr>
</tbody>
</table>
The IJ is an Open Access on-line publication covering the domain of medical image processing and visualization. The unique characteristics of the Insight Journal include:

- Open-access to articles, data, code, and reviews
- Open peer-review that invites discussion between reviewers and authors
- Emphasis on reproducible science via automated code compilation and testing
- Support for continuous revision of articles, code, and reviews

Once the data, taxonomies, and use-case of a toolkit have been, a derived set of requirements must be specified. These requirements can influence the development process via a variety of tools. Which tools is most appropriate will depending on the size of the collaborative community, the level of accountability needed (e.g., is the software development process going to produce a product that will be subject to regulatory review such as FDA approval), and the level of external participation that is anticipated/desired early in the development process. One technique that is growing in popularity is to use Mantis or JIRA trackers to publicize the requirements. “Commit hooks” for git and svn repositories that gather contributions can query the contributor to determine which requirements their code addresses. It is even possible to define the hooks such that any contribution that does not address a requirement is rejected. This approach may seem burdensome, but it maintains toolkit focus, avoiding feature creep and bloat.

MantisBT is a free popular web-based bug tracking system. It is written in the PHP scripting language and works with MySQL, MS SQL, and PostgreSQL databases and a webserver. MantisBT has been installed on Windows, Linux, Mac OS, OS/2, and others. Almost any web browser should be able to function as a client. It is released under the terms of the GNU General Public License (GPL).

(WARNING: JIRA is NOT open source or freely available. Only certain projects qualify for free use of JIRA. It is mentioned here because of its popularity, but generally we cannot recommend its continued use in open-source environments.)

IRA is a project tracking tool for teams building software. JIRA sits at the center of your development team, connecting the team and the work being done. Track bugs and defects, link issues to related source code, plan agile development, monitor activity, report on project status, and more.
### 4. Software Repository

The community software development process then begins with the first unit of code is shared. Software repositories are responsible for hosting the code. Section 7 of this document discusses how the posting of code to a repository, managing experimental and release versions of the code, and other software processes are tightly integrated with repository policies. In this section we merely describe the two most popular types of repositories for hosting code.

<table>
<thead>
<tr>
<th>Software</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Git</strong></td>
<td>Git is a free &amp; open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Its highlights include:</td>
</tr>
<tr>
<td></td>
<td>• Distributed development. Like most other modern version control systems, Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch. Repositories can be easily accessed via the efficient Git protocol (optionally wrapped in ssh for authentication and security) or simply using HTTP - you can publish your repository anywhere without any special webserver configuration required.</td>
</tr>
<tr>
<td></td>
<td>• Strong support for non-linear development. Git supports rapid and convenient branching and merging, and includes powerful tools for visualizing and navigating a non-linear development history.</td>
</tr>
<tr>
<td></td>
<td>• Efficient handling of large projects. Git is very fast and scales well even when working with large projects and long histories. It is commonly an order of magnitude faster than most other version control systems, and several orders of magnitude faster on some operations. It also uses an extremely efficient packed format for long-term revision storage that currently tops any other open source version control system.</td>
</tr>
<tr>
<td></td>
<td>• Cryptographic authentication of history. The Git history is stored in such a way that the name of a particular revision (a &quot;commit&quot; in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed. Also, tags can be cryptographically signed.</td>
</tr>
<tr>
<td></td>
<td>• Toolkit design. Following the Unix tradition, Git is a collection of many small tools written in C, and a number of scripts that provide convenient wrappers. Git provides tools for both easy human usage and easy scripting to perform new clever operations.</td>
</tr>
<tr>
<td><strong>Subversion (SVN)</strong></td>
<td>Subversion is an open source version control system. Founded in 2000 by CollabNet, Inc., the Subversion project and software have seen incredible success over the past decade. Subversion has enjoyed and continues to enjoy widespread adoption in both the open source arena and the corporate world.</td>
</tr>
<tr>
<td></td>
<td>• Most CVS features: CVS is a relatively basic version control system. For the most part, Subversion has matched or exceeded CVS's feature set where those features continue to apply in Subversion's particular design.</td>
</tr>
<tr>
<td></td>
<td>• Copying, deleting, and renaming are versioned.</td>
</tr>
<tr>
<td></td>
<td>• Atomic commits: No part of a commit takes effect until the entire commit...</td>
</tr>
</tbody>
</table>
Multiscale Spatiotemporal Visualisation
Collaborative software tools review

<table>
<thead>
<tr>
<th>has succeeded. Revision numbers are per-commit, not per-file, and commit's log message is attached to its revision, not stored redundantly in all the files affected by that commit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Branches and tags are both implemented in terms of an underlying &quot;copy&quot; operation. A copy takes up a small, constant amount of space. Any copy is a tag; and if you start committing on a copy, then it's a branch as well. (This does away with CVS's &quot;branch-point tagging&quot;, by removing the distinction that made branch-point tags necessary in the first place.)</td>
</tr>
<tr>
<td>- Merge tracking and interactive conflict resolution: The Subversion command-line client (svn) offers various ways to resolve conflicting changes, include interactive resolution prompting. This mechanism is also made available via APIs, so that other clients (such as graphical clients) can offer interactive conflict resolution appropriate to their interfaces.</td>
</tr>
</tbody>
</table>
5. Community quality standards

Whereas licensing and scope are generally tools for determining who will collaborate and what the toolkit should ideally look like, the remaining of this document the tools of collaborative software development that define how the collaborations will proceed day-to-day, to build the toolkit. In particular, the tools for collaborative software development define how a project will balance two somewhat-opposing priorities:

1) Enforcing the quality standards set by the community regarding the appearance, ease-of-use, longevity, and utility of the toolkit
2) Enabling collaborators to more quickly and more effectively make use of and contribute code back to the toolkit

Consider that in precision woodworking, much of the effort goes into making jigs such so when a cut is finally made, it is constrained to be made exactly as expected and with as little effort as possible. Similarly, collaborative software development tools are the jigs that both guide the actions of the developers for the good of the toolkit and yet allow the developers’ efforts to have maximum impact.

If a collaborative development tool is too restrictive or cumbersome, the contributions to the toolkit will be limited. If a collaborative development tool is too permissive, then the toolkit becomes “sloppy” and its utility and lifespan are limited.

In this section we look at the quality standards that are important to the utility and longevity of a toolkit and how collaborative software development tools help maintain those standards. In the next section we then look at how the collaborative software development tools facilitate the development of contributions and the toolkit. We begin by looking at the anatomy of a contribution.
6. Assessing contribution quality

Contributions generally have six components, these are each address in this section:

(6.1) Algorithm
(6.2) Implementation
(6.3) Documentation
(6.4) Backward compatibility
(6.5) Testing
(6.6) Reporting

6.1. Algorithm quality

Regarding the algorithm embodied by a contribution, the primary task of quality assessment is determining the utility of the algorithm with respect to the scope of the toolkit. In many ways, this is the most challenging of all of the quality assessment topics because it can be a highly subjective process. It requires an understanding of the application domain of the toolkit, the state-of-the-art within that domain, and the algorithms and implementations already available in the toolkit.

Algorithm quality assessment, as a subjective assessment, was historically left to the lead architects of a toolkit. They would decide what methods are added and/or removed. However, this task often alienated the lead architects from the rest of the community. If handled in a non-transparent manner, the rejection or inclusion of an algorithm could be viewed as arbitrary and have broad and extremely negative impact on the community, the developers, and the toolkit. Consider Apple’s AppsStore for the iPhone. Those who follow tech news are all too familiar with the multiple public relations nightmares that Apple has endured for its apparently arbitrary and self-serving acceptance/rejection of contributed apps into its AppsStore. On the other hand, consider a toolkit which accepted all contributions: multiple instances of the same functionality might be present, strong methods would become obscured by weaker ones, and maintaining such a broad code based would become taxing.

There appear to be two acceptable solutions to Algorithm Quality Assessment:

1) Community Review: For community review a popular type of tool is exemplified by the Insight Journal (www.insight-journal.org). Each publication represents code that is being offered for consideration for contribution to ITK (a similar journal exists for VTK and one is in-the-works for Slicer). A publication consists of the code of the contribution as well as a written description of the algorithm and its utility. The community members can download and try out the code, read the descriptions, and write reviews. Those articles that receive the most positive reviews become the primary candidates for inclusion in the toolkit. In that manner, the community drives the contribution selection process, not the lead architect. The process is transparent. Furthermore, developers are motivated to create high quality contributions because of the public nature of the process.

2) Quantitative Review: An outstanding example of quantitative review is the Retrospective Image Registration Experiment (RIRE Project, Vanderbilt University, http://www.insight-journal.org/rire/). In this project, a set of data was engineered for quantitatively evaluating rigid registration algorithms. Potential contributors would submit the results from their algorithms on this data, their algorithms would be scored, and those scores published on the web. The best
performing algorithm could thereby be easily identified and integrated into quality toolkits (such as ITK). Other examples of such quantitative evaluations include the multitude of “grand challenges” that have become popular at workshops at international conferences such as the annual Medical Image Computing and Computer-Assisted Interventions (MICCAI) conference. For a list of past and ongoing grand challenges, see http://www.grand-challenge.org.

Even the best algorithm, however, can be made unappealing to a toolkit or its community if its implementation is not consistent with the rest of the toolkit. This is a matter of assessing implementation quality, as explained next.

6.2. Implementation quality

There are two factors that should be considered during most implementation assessments.

1) Coding style
2) Cross-platform building and packaging

6.2.1. Coding style

Coding style refers to the appearance of the code when viewed in an editor (VIM, EMacs) or development environment (Eclipse, Qt Creator, or MS Visual Studio). We cannot emphasize enough how important it is for a toolkit to adopt a reasonable and consistent coding style. A well-established coding style makes all of the code in the toolkit appear as though it was written by a single person, with the primary intention being ease-of-comprehension by others. In general, new toolkits should follow the coding style standards set forth by other toolkits. Ease-of-comprehension is a general philosophy whereby the code is written with “others” in mind – not merely to solve a problem, but also to educate / convey how that problem was solved. The coding style should be the first thing a new toolkit developer reads and should be carefully applied to every code contribution

Contents of a coding style guideline will vary, but the guidelines should be published on the toolkit’s wiki or other readily available document. ITK’s Coding style is online at http://www.vtk.org/Wiki/ITK_Coding_Style_Guide. Its sections include:

- System Overview & Philosophy --- coding methodologies and motivation for the resulting style.
- Copyright --- the copyright header to be included in all files and other copyright issues.
- File organization --- how to organize source code.
- Naming conventions --- patterns used to name classes, variables, template parameters, and instance variables.
- Namespaces --- the use of namespaces to limit scope
- Code Layout and Indentation --- accepted standards for arranging code including indentation style.
- Exception Handling --- how to add exception handling to the system.
- Documentation Style --- a brief section describing the documentation philosophy of ITK

That is, it offers a detailed explanation of how brackets should be indented at the start and end of for-loops, how comments should be formatted, how variable names should be chosen, how try-catch exceptions should/shouldn’t be used, how spaces should be places around mathematical operators, and much, much more.
A variety of open-source tools have been developed to verify that a project’s coding style has been followed. Configuration files are used to encode the coding style for a specific project, and the tools use those configuration files when parsing contributed code or to continuously monitor code as it is written in a compliant editor, e.g., vim or emacs. Two popular code-checking tools are cpplint and KWSstyle.

<table>
<thead>
<tr>
<th>CPPLint (aka Google-StyleGuide)</th>
<th>CPPLint is a python script developed at Google for checking code styles. It is a minimalist yet effective tool for verifying python, Objective-C, C++, and java coding styles.</th>
</tr>
</thead>
<tbody>
<tr>
<td>KWStyle</td>
<td>KWStyle is primarily checking C/C++ source code but can be easily extended to other languages. It assumes that the code is syntactically correct, i.e. it compiles on a standard compiler. It then provides a multitude of checks. It can also output errors in a format that can be read by the CDash automated testing system for web-based report (more detail is given on CDash later).</td>
</tr>
</tbody>
</table>

### 6.2.2. Cross-platform building and packaging

Beyond the appearance of the code, it also needs (at a minimum) to be compiled, run, and packaged for redistribution on other machines (e.g., on difference Windows installations) and potentially on other platforms (e.g., Windows, MacOS, and Linux). Even when writing code for a single platform (e.g., Windows), building and packaging can be extremely daunting and frustrating tasks because of the variety of Windows installations, compiler versions, and systems configurations that are in-use. An open-source and collaborative project should attempt to maximize the number of machines, compilers, and operating systems that are supported by their code, so as to foster the broadest community of collaborators and users as possible. Furthermore, compiler variations result in difference checks being performed on the code, and major errors can be incorrectly overlooked by one compiler and reported by another. However, the balance between burden and benefit must be maintained – and writing makefiles for linux, VS project files, VS solution files, and makefiles for MacOS would be an unreasonable burden on most developers. Luckily, some outstanding open-source tools have been developed for cross-platform building and packaging.

<table>
<thead>
<tr>
<th>CMake</th>
<th>CMake is a cross-platform, open-source build system. CMake is a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is downloaded over 1800 times a day, and it used to build KDE, World of Warcraft, and 1,000s of other commercial and open-source projects.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCons</td>
<td>SCons is an Open Source software construction tool—that is, a next-generation build tool. Think of SCons as an improved, cross-platform substitute for the classic Make utility with integrated functionality similar to autoconf/automake and compiler caches such as ccache. In short, SCons is an easier, more reliable and faster way to build software.</td>
</tr>
</tbody>
</table>
qmake is a tool that helps simplify the build process for development project across different platforms. qmake automates the generation of Makefiles so that only a few lines of information are needed to create each Makefile. qmake can be used for any software project, whether it is written in Qt or not.

qmake generates a Makefile based on the information in a project file. Project files are created by the developer, and are usually simple, but more sophisticated project files can be created for complex projects. qmake contains additional features to support development with Qt, automatically including build rules for moc and uic. qmake can also generate projects for Microsoft Visual studio without requiring the developer to change the project file.

6.3. Documentation quality

Companion to the release and distribution of code that is well written and cross-platform is the need to document the API and operation of the code. Note that this documentation is quite different from the documentation of the algorithm. Both are important, however, the tools for collaborative software development are typically (for better or worse) focused on documenting the code.

Different programming languages are served by different code documentation systems. JavaDoc is extremely popular and well designed for converting java comments and code into effective documentation. Python is well supported by a variety of packages, such as Sphinx which produces documentation in PDF, HTML, and Latex and several other formats. For C++, the most popular documentation system is Doxygen.

For its website:

*It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.*

*You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.*

*You can even `abuse' doxygen for creating normal documentation.*

*Doxygen is developed under Linux and Mac OS X, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available.*

One of the most famous use cases for Doxygen in the open-source medical image analysis field is with ITK. The 800+ page Insight Software Guide, that is well illustrated with data, examples, code, graphs, tables, latex-encoded equations and more is actually 100% generated by doxygen. The comments in ITK’s extensive collection of examples become the Insight Software Guide. The beauty of this
approach is that as examples are added or modified, the Insight Software Guide is automatically updated to reflect those changes. While the burden on the code writing may at times seem extreme, by integrating documentation generation with code writing, the developer is less likely to postpone documentation generation to a point in the future which often results in the documentation never being generated. ITK’s software guide is available in PDF at http://www.itk.org/ItkSoftwareGuide.pdf and ITK’s doxygen-generate HTML documentation is online at http://www.itk.org/Doxygen40/html/index.html

6.4. Backward compatibility

While often overlooked by open-source toolkits, a well thought-out backward compatibility policy is critical to ensuring that the code has a broad and lasting impact on its community.

Note that creating a backward compatibility policy does not mean that backward compatibility needs to be maintained throughout the entire lifetime of the code. The critical part is ensuring that the community knows what the policy is and that the developers follow it.

Note also that code testing can also drive the development process: test-driven code development. This is a software process that guides the creation of code by having developers first write tests that would pass if they had code that did what they wanted, and then working to create the code that makes it pass. Such software processes are discussed in more detail in the next section.

6.4.1. Backward compatibility policies

One of the most stringent backward compatibility policies for open-source software is the Insight Toolkit’s backward compatibility policy. The Insight Software Consortium, which holds the copyright to ITK and sets the policies and processes of ITK, published several position statements on backward compatibility from its leaders. Those statements are online at:

http://www.itk.org/Wiki/ITK_Backward_Compatibility_Open_Discussion

The outcome of those and other discussions led to the adoption of ITK’s backward compatibility policy which is available online at:

http://www.itk.org/Wiki/images/2/2d/ITKAPIChangePolicy.pdf

It can be argued that stricter backward compatibility policies are kinder to users in that those policies allow the users to build upon a toolkit and make use of updates with little change to the users code base. However, it can also be argued that too strict of a backward compatibility policy can stifle innovation. The challenge are (1) to choose the right balance, based on the user base; and (2) to setup regression tests so that it is when backward compatibility is broken by a code modification. This leads to our next topic, testing.

6.5. Tests and Coverage

Three types of tests are typically employed to ensure the proper and consistent (i.e., backward compatible) operation of a toolkit: regression tests, dynamic tests, and static tests.
6.5.1. Regression tests

As mentioned, regression testing and backward compatibility are nearly synonymous. Regression tests are “unit tests” that repeatedly exercise a specific component of the code. The proper operation of the code for a given test is assured once (when the test is created) and from that point on only changes in regression test results need to be detected – to ensure that on-going changes to the code do not change the API or lead to different results.

There is an “art” to crafting effective regression tests. Rather than attempting to assess the expected performance of an algorithm for a problem domain, the tests should check the “corner cases”, i.e., the cases that represent the extremes or unusual cases that may be encountered. Furthermore, there is no need to ensure an optimal answer is achieved by a regression test. Instead, the focus should be on developing small tests that run quickly (to encourage writing and running many, diverse tests) and that give consistent results when the unit is operating correctly. For example, it may not be necessary to test the final answer achieved by an optimizer, but instead it may be sufficient to test it for convergence over only a few iterations.

6.5.2. Dynamic tests

Beyond testing the numerics associated with an algorithm, it is necessary to test the memory and cpu usage of an implementation. Memory and CPU time are precious commodities, and minimizing their use will enable a toolkit to be applied to larger and larger problems. Three popular tools for conducting memory and profiling (performance) tests are discussed below.

<table>
<thead>
<tr>
<th>Valgrind</th>
<th>Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools. The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a heap/stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, ARM/Android (2.3.x), X86/Darwin and AMD64/Darwin (Mac OS X 10.6 and 10.7).</th>
</tr>
</thead>
</table>
| LTPProf | TProf is a small but powerful CPU profiling tool for Visual C++, Borland Cbuilder, Delphi and VB applications. Although small, it has a combination of features difficult to find in even the most expensive tools available today:  
- Line level resolution  
- No instrumentation, no rebuild, no time wasted  
- Profiled program runs at its normal speed  
- Profile Visual C++, C Builder, Delphi and Visual Basic code  
- Price: EUR 49.95 |

(for Windows)
IBM’s Rational Purify

Rational® Purify® is a dynamic software analysis tool designed to help developers write more reliable code. It includes two capabilities: 1) Memory debugging (pinpoints hard to find memory errors such as uninitialized memory access, buffer overflow and improper freeing of memory), 2) Memory leak detection (identifies memory blocks that no longer have a valid pointer). Purify is supported on Windows®, Linux®, Solaris®, and AIX®.

6.5.3. Static Tests

Certain code deficiencies can be detected most effectively by looking for certain code usage patterns. This is the concept behind “static testing.” Errors detected via static tests can include the following:

- Out of bounds checking
- Check the code for each class
- Checking exception safety
- Memory leaks checking
- Warn if obsolete functions are used
- Check for invalid usage of STL
- Check for uninitialized variables and unused functions

For C++, one of the most popular static checking applications is Cppcheck.

6.5.4. Code coverage

An important, but often overlooked component of testing is ensuring that the tests cover the code. Coverage is roughly quantified by portion of the lines of code that are run in the unit when its tests are run.

It should be noted that for ITK the target coverage is 85% - that is, the combined tests of ITK should cause the execution of 85% or more of the code. This number may seem low, given that ITK is being applied in critical conditions of medical care; however, the remaining 15% of code is typically associated with conditional statements that should only be executed when errors are encountered. This makes testing those conditionals difficult and tedious. Nevertheless, if a medical device employs ITK, more thorough coverage tests would be needed for that particular application.

Two popular systems for coverage computation are gcov and bullseye.
your optimization efforts will best affect your code. You can also use gcov along with the other profiling tool, gprof, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help you analyze your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

**BullseyeCoverage**

BullseyeCoverage is a code coverage analyzer for C++ and C that tells you how much of your source code was tested. You can use this information to quickly focus your testing effort and pinpoint areas that need to be reviewed. Code coverage analysis is useful during unit testing, integration testing, and final release. BullseyeCoverage enables you to create more reliable code and save time.

- Function coverage gives you a quick overview and condition/decision coverage gives you high precision
- Works with everything you can write in C++ and C, including system-level and kernel mode
- Short learning curve
- Include or exclude any portion of your project code
- Run-time source code included, for custom environments
- Simple licensing
- Merge results from distributed testing
- Integration with Microsoft Visual Studio

### 6.6. Reporting

The final piece of the puzzle of tools for Collaborative Software Development is reporting. Reporting involves rapidly and concisely conveying and tracking adherence to coding style, cross-platform compilation, and the results of the tests, for the community to monitor and the developers to manage.

**Quality Assurance Toolkit (QAT)**

Quality Assurance Toolkit (QAT) is an Open Source project written in Python with BSD license with the aim of gathering different aspects of Quality Assurance like Coding Style, Coverage, Code Analysis, Statistics using an extensible mechanism based on Rules. Beyond this basic functionality which helps the developer to implement elegant code, it is possible to integrate external applications for different calculation of code metrics. An html report is provided, and a history is maintained for graphs generation. An example applied to MAF3 can be found here: [http://qa.openmaf.org/](http://qa.openmaf.org/)

**CDash**

CDash is an open source, web-based software testing server. CDash aggregates, analyzes and displays the results of software testing processes submitted from clients located around the world. Developers depend on CDash to convey the state of a software system, and to continually improve its quality. CDash is a part of a larger software process that integrates Kitware's CMake, CTest, and CPack tools, as well as other external packages used to design, manage and maintain large-scale software
Good examples of a CDash are the [CMake quality dashboard](https://cmake.org/cdash) and the [VTK quality dashboard](https://www.vtk.org).
7. Software process workflows

In conjunction with the above collaborative software development tools, a variety of workflows for collaborative software development have also been developed. These workflows have been detailed and studied extensively, and they continue to evolve as new tools are developed and new lessons learned. Herein we provide a brief introduction to the most popular software development method: Agile Development. We also present a common specific case of Agile Development that is called “Scrum.”

From Wikipedia: **Agile software development** is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen interactions throughout the development cycle. The Agile Manifesto introduced the term in 2001.

Also from Wikipedia: **Scrum** is a form of agile project management. Scrum contains sets of methods and predefined roles. The main roles in Scrum are:

- the "ScrumMaster", who ensures the process is followed and removes impediments
- the "Product Owner", who represents the stakeholders and the business
- the "Development Team", a group who does the coding, implementation, testing, etc.

A sprint is the basic unit in Scrum. Sprints last between one week and one month and are a "timeboxed" (i.e. restricted to a specific duration) effort of a constant length.
Each sprint is preceded by a meeting, where the talks for the sprint are initiated for the sprint goal, and followed by a retrospective meeting where the progress is reviewed and lessons for the next sprint are identified.

During each sprint, the group creates an unfinished portion of product. The work items that go into a sprint come from the product “backlog”, which is a prioritized requirements of work to be done. Which backlog items go into the sprint is determined during the sprint planning meeting. During this meeting, the Product Owner informs the group of the items in the product backlog that he or she wants completed. The group then determines how much of this they can commit to complete during the next sprint, and records this in the sprint backlog.[6] During a sprint, no one is allowed to change the sprint backlog, which means that the requirements are frozen for that sprint. Management is timeboxed such that the sprint must end on time; if requirements are not completed for any reason they are left out and returned to the product backlog.

Scrum enables the creation of self-organizing groups by encouraging co-location of all group members, and verbal communication between all group members and disciplines in the project.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need (often called requirements churn), and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the group’s ability to deliver quickly and respond to emerging requirements.

8. Conclusion

A rich set of tools and processes exist for managing a collaborative software development effort. The main goal of these conventions is to enable developers to rapidly produce software that implements a particular algorithm while maintaining the quality standards of the community regarding implementation, documentation, backward compatibility, and testing.

Herein we presented several leading tools and process, and provided a context for understanding how those tools and processes contribute to achieving high quality software.

Future documentation from the MSV Consortium will detail the specific tools and processes chosen for the MSV project.