

A Middleweight Requirements Management Framework

Steve W. Bannerman
Simplify and Oxford University
steve.bannerman@comlab.ox.ac.uk

Abstract

Software developers need to manage requirements. To do so, they can choose between “ultra-lightweight,” “lightweight,” and “heavyweight” requirements management tools. Ultra-lightweight tools are simple, flexible, and cheap but do not scale very well. Lightweight tools are also simple, flexible, and cheap but do not scale well. Heavyweight tools scale well, but are complex, inflexible, and costly. Problems with these tools are decreasing the quality and/or increasing the cost of non-trivial software development efforts. This paper presents a requirements management framework that is simple, flexible, and cheap in addition to scaling well: a middleweight requirements management tool.

Introduction

Organizations need to manage requirements for non-trivial systems they develop. It is an essential activity of both the system development lifecycle and the software development lifecycle [8]. But what are requirements and what does it mean to manage them?

A requirement is a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents [1]. A requirement is categorized as “functional” if it specifies what the system needs to do. Otherwise, it is categorized as “non-functional.”

A requirement can be embodied by a group of individual statements. For example: the requirement that a system be highly reliable can be embodied as: “The system shall be 99.997% reliable.” This embodiment, or artifact, can also have attributes associated with it. Common attributes are a name, a priority, and an estimated effort. Additionally, the embodiment can be associated with other artifacts. Common associations link a requirement to the implementation artifacts that support it or to the artifacts that describe how to test those implementation artifacts in order to meet the requirement.

Although the distinction between a requirement and its embodiment is important in some contexts, it is not important here. For the duration of the paper, the term requirement refers to both the requirement and its embodiment.

Managing requirements means at least two activities: managing the requirements artifacts themselves (elicitation and analysis) and using information from the requirements artifacts as an input into the project management process (scoping and prioritizing).

Software developers manage requirement artifacts by: (1) adding a requirement to a requirement set; (2) changing a requirement in a requirement set; and (3) deleting a requirement from a requirement set. Changing a requirement may involve adding/changing/deleting attributes of a requirement or forming/breaking an association between a requirement and another artifact.

Software developers use information from requirement artifacts in a variety of ways, ways that vary from simple to sophisticated. Consider a simple case. Assume that all current requirements for a release have been elicited and analyzed with respect to effort and priority. Further assume that the software developer playing the project manager role has been given an end date for the release. He or she will use the information associated with the requirements to help define the requirement set that is most desirable (priority) and achievable (effort) in the time allotted. Alternatively, they can negotiate for more time based on this information.

Many software developers use tools to help them manage requirements more effectively. Some tools are not software applications at all – “ultra-lightweight” tools such as pen and paper. Some tools are general software applications – “lightweight” tools such as Word and Excel - according a survey by Rational, approximately 50% of all requirements are captured with this type of tool [7]. Finally, some tools are commercial off-the-shelf software applications – “heavyweight” tools such as DOORS [13] and RequisitePro [9].

Each of these types of requirements management tools has its problems. In the next sections we identify the major strengths and weaknesses with each type. We then propose a “middleweight” solution which reduces or removes the weaknesses while maintaining the strengths. Following that, we discuss potential problems of the middleweight solution itself, experiences in implementing the framework, and future work of the middleweight project.

Strengths and Weaknesses

Ultra-lightweight requirements management tools are simple, flexible, and cheap. Every software developer has access to a pen and a paper and knows how to use them. Therefore, they are simple. Software developers can write the requirement anyway they want and they can change and order the requirement set at will by adding, changing, or removing papers. Therefore, they are flexible. And, obviously, they are cheap when compared to software development costs.

However, ultra-lightweight tools do not scale very well, at least in a couple of ways. First, they are physically constrained. If the project managers and the software developers wish to look at the information contained in the requirement set, they have to access the same instance of the requirement set (unless copies are made). The problem is exacerbated for geographically distributed projects. Second, when a requirement set gets large, it becomes time consuming and error prone to filter and/or sort it. Third, it is difficult and time consuming to make a backup copy, although it can be done. And finally, it is difficult (or impossible without a versioned copy) to determine what has changed in a requirement set.

Lightweight requirement management tools are simple, flexible, and cheap. Nearly every software developer has access and practice using Word and Excel. Because software developers already know how to use these tools, managing requirements with them is simple. Since Word and Excel are general document and spreadsheet applications, respectively, they are flexible. And software developers already have access to them for other reasons; therefore, they are cheap.

But lightweight tools do not scale very well either, but for a different reason: lightweight tools operate at the wrong level of granularity. Whereas ultra-lightweight tools allocate a requirement to a single physical object (piece of paper or Index Card [6]), lightweight tools allocate a requirement set to a single logical object (a Word or Excel file). This causes concurrency problems as only one software developer can change the requirement set at a time. It also makes it difficult to determine what has changed between one version of a requirement set and another. In fairness, Word and Excel provide tools to facilitate viewing changes between versions of documents; however, it is still difficult and time consuming to determine changes to individual requirements within a requirement set.

A possible, but seldom used variant of a lightweight tool operates at the correct level of granularity. That is, one Word or Excel file per requirement. This removes the scaling problem of concurrency. Unfortunately, it introduces other scaling problems. The proprietary format prevents filtering and sorting based on information within the file. This makes it difficult to find the “right” set of requirements to work on. Also, the proprietary format imposes a file overhead (even with no requirement content) of approximately 10KB; although size is not as important as it once was this might be a scaling consideration on some projects.

Heavyweight requirement management tools scale well. They operate at the correct level of granularity (the individual requirement level) and use commercial databases to store the requirement set. They provide custom or generic applications that allow software developers to maintain the requirement set. Many

software developers can edit the requirement set at one time and differences between versions of requirements can be viewed. Therefore, these tools scale.

Unfortunately, heavyweight tools are complex, inflexible, and costly. Tools that involve application software and database software, as well as database administration, can be complex because of heterogeneous environments and different versions of software. Additionally, tools with a lot of features are complex because it takes time and effort to learn how to use them. These tools are inflexible because the application software and the database schemas are proprietary; this makes it difficult or impossible to customize. Finally, these tools are costly in price (hundreds or thousands of dollars per license), training, and maintenance.

Figure 1 summarizes the strengths (S) and weakness (W) of the tool types.

Tool type	Simple	Flexible	Cheap	Scales			
				Backup	Concurrency	Filter/Sort	Compare
Ultra-lightweight	S	S	S	W	W	W	W
Lightweight	S	S	S	S	W	W	-
Heavyweight	W	W	W	S	S	S	S

Figure 1 – Strengths and weaknesses by tool type

Solution

Let’s consider source code management for a moment. Few would dispute that it is more mature than requirements management. In general, programmers organize their source code files within a set of source folders. Further, they organize the source code files (within a particular source folder) into source packages. Source code developers use their favorite tools (anywhere from “lightweight” to “heavyweight”) to browse and transform the source code files. They also use their favorite tools to persist their changes to a configuration management repository (which allows them to share their source files with their team and have their changes backed up). Finally, they use their favorite tools to filter, sort, and compare their source files.

So can requirement developers benefit from this “paradigm?” They can if it allows them to reap the strengths afforded by the ultra-lightweight, lightweight, and heavyweight solutions and simultaneously avoid their weaknesses. That is, a simple, flexible, cheap solution that also scales: a “middleweight” requirements management solution.

Core Solution - RML

Again, a requirement (or its embodiment) is a set of statements, attributes, and associations that convey a condition or a capability that must be met by a system. Such embodiments can be organized in a tree structure, with the requirement itself as the root and the attributes and associations as branches of the tree.

And since Extensible Markup Language (XML) is both well-suited for representing tree structures and has so much industry support, an XML language seems the natural choice for embodying requirements in a file. Therefore, the Requirements Markup Language (RML) is proposed.

A first approximation of the RML is described with an example requirement (Figure 2). The only universal attribute of a requirement is its description. This requirement has two attributes: its owner and the date of its creation. It also has two associations: a relative reference to its refining use case file and an absolute reference to its related test case file. Note that the attributes and associations are typed, implying some flexibility in reading and writing them.

```

<requirement>
  <description>Create a switch in the context of a site</description>
  <attributes>
    <attribute type="string" name="owner" value="John Smith"></attribute>
    <attribute type="date" name="created" value="01/01/2003"></attribute>
  </attributes>
  <associations>
    <association type="relative" name="refining use case" value="file1"></association>
    <association type="absolute" name="related test case" value="file2"></association>
  </associations>
</requirement>

```

Figure 2 – First approximation for RML

If RML is used to store individual requirements in a file, several of the “middleweight” requirements tool’s “requirements” are solved. The structure of the RML file is simple: just attribute and association tags. It is also flexible: any number of attributes or associations can be present. Finally, it is cheap (assuming everyone has the ability to freely create and edit files on their computers). These qualities match those of ultra-lightweight and lightweight requirements management tools.

But the benefits don’t stop there; the scaling qualities of heavyweight requirement management tools are also matched. Software developers can check requirement files into a repository that is backed up. Different individuals can edit different requirements at the same time because they’re embodied in different files. And all of the existing filtering, sorting, and comparison tools for files can be leveraged on the requirement files.

Extended Solution – A Requirements Development Environment

With a simple, flexible, and cheap requirements language, developers can browse and transform the requirement set by using their favorite “file explorer” and directly editing requirement files. They can place the requirement files under configuration management, thus sharing them amongst their team members and having them backed up. Strictly, no extension is necessary.

However, XML files are not easy to read or write for most, and especially those who focus on “business” rather than “technology.” Nor are they intended to be so. Therefore, we should extend the solution to ease browsing and transformation of the requirement files.

A look at Java technology is instructive. Java is an object-oriented language from Sun. A tree of constructs such as classes, fields, and methods are stored in files with a “.java” extension. At a minimum, a Java developer needs a text editor (to edit the “.java” files), a compiler (to transform the “.java” files into “.class” files) and a runtime environment (to execute the “.class” files). If they need to share their source files, they also need a tool to place them in a repository.

But many Java developers utilize tools that extend this basic toolkit to make them more effective. The Eclipse project [4] is an example of such a tool set. It provides a graphical interface that allows developers to browse their source files (with helpful highlighting). It also provides access to a command-line interface that allows developers to filter, sort, and transform their source files (via ant described in XML). Finally, it allows Java developers to synchronize their local source files with a repository.

Paralleling these Java technology extensions, two broad types of extension are fitting for the core “middleweight” solution: (1) graphical interfaces to support browsing and transformation of requirement files; and (2) command-line tools to support the filtering, sorting, and transformation of requirement files.

The extensions are explained below with the aid of design patterns. Their aim is to make browsing and transformation of RML simpler and more flexible for the same price (free). They have been implemented in Java and are part of the Simplify project [12].

Reading and Writing

Both graphical and command-line extensions are supported by transforming a requirement from the file system into a Java object and vice-versa. The Memento design pattern [5] has explanatory value here. Requirement readers and writers (originators) have the responsibility to transform from and to requirement files (mementos). In this case, the file system can be thought of as the caretaker of the mementos. Figure 3 identifies the syntax of their responsibilities; the semantics of their responsibilities are outside the scope of this framework description because of space constraints (but, hopefully, are somewhat obvious).

```
public interface RequirementWriterI {
    public void writeRequirements(RequirementFolderI folder) throws WriteException;
    public void writeRequirements(RequirementPackageI reqPackage) throws WriteException;
    public void writeRequirement(RequirementI requirement) throws WriteException;
}

public interface RequirementReaderI {
    public void readRequirements(RequirementFolderI folder) throws ReadException;
    public void readRequirements(RequirementPackageI reqPackage) throws ReadException;
    public void readRequirement(RequirementI requirement) throws ReadException;
    public RequirementI readRequirement(File reqFile) throws ReadException;
}
```

Figure 3 – Readers and Writers

As mentioned in the core solution section, the “types” of the attributes and associations need to be flexible. One way to facilitate this is to establish a mapping between type names and prototypes at startup. Then, as a reader reads the requirement from its embodiment (in RML), it can build attributes and associations from the identified prototypes.

Let’s assume two factories: an attribute factory and an association factory. Both are Singletons [5] in the sense that there is only one of each in any object environment. While configuring the factories, the goal is to associate a “type” name with a Prototype [5], or prototypical instance, which can be copied and validated when a particular requirement is read. Figure 4 shows a sample configuration and the two main responsibilities associated with a factory: initialization from a configuration and creating a product (an attribute in this case).

```
string=ca.simplify.reqs.attribute.StringAttribute
integer=ca.simplify.reqs.attribute.IntegerAttribute
float=ca.simplify.reqs.attribute.FloatAttribute
date=ca.simplify.reqs.attribute.DateAttribute

public interface AttributeFactoryI {
    public void initialize(Map config) throws InitializeException;
    public AttributeI createAttribute(String type, String name, String value);
}
```

Figure 4 – Attribute Factory

In this example, a team has decided upon the following attribute types: {string, integer, float, date}. Similarly, they might settle on the following association types: {absolute file reference, relative file reference}.

Once the type factories are initialized, the requirement reader(s) can build requirements from RML files. The Builder [5] design pattern illustrates this collaboration: the requirement reader (director) interprets an RML file and delegates to the attribute and association factories (builders) to create the appropriate attributes and associations (products). Also, according to a requirement's position within a requirements folder, the reader (director) constructs an appropriate requirement package and places the requirement in it.

Writing a requirement into an RML file is straightforward. The requirement writer must simply traverse the requirement tree and generate attribute and association tags according to the components of the requirement. The directory and the file name are derived from the requirement folder, the requirement package that contains the requirement, and the name of the requirement.

Filtering and Sorting

In addition, both graphical and command-line extensions need the ability to filter and sort requirements. With respect to graphical tools, developers need to filter out those requirements that are of interest (for example: those requirements that are so far unsupported). They also need to sort those requirements that pass through the filter (for example: in order of user priority).

Similarly, with command-line tools, developers need to filter out a set of requirements that they want to process (for example: generating a document with all of the requirements added since the beginning of the iteration). They also need to sort the requirements that pass through the filter prior to processing them (for example: in named order). Figure 5 identifies the responsibilities of filters and sorters.

```
public interface FilterI {
    public boolean filter(RequirementI input);
    public RequirementI[] filter(RequirementI[] inputArray);
}

public interface SorterI {
    public RequirementI[] sort(RequirementI[] inputArray);
}
```

Figure 5 – Filters and Sorters

Browsing and Transforming

Once developers have the tools to read, write, filter, and sort requirements, they need to be able to browse and transform them through a graphical user interface. Since we're borrowing from the source code "paradigm," a workbench which contains requirements folders is fitting. The workbench provides two main navigational views on the set of requirements: a tree and a list. A filter is appropriate for tree views while both a filter and a sorter are appropriate for list views.

Since filters and sorters are also applicable for command-line tools, their definitions are also embodied in files. The simplest way to associate filters and sorters with a view is to select a filter description file and/or a sorter description file. This leverages the existing mechanism instead of making developers describe filters or sorters twice (once in a file and once graphically).

By default, a tree view has a "null filter," an example of the Null Object design pattern [12]. A "null filter" lets all requirements pass through it. Similarly, by default a list view has a "null filter" and a "null sorter." The "null sorter" is also an example of the Null Object design pattern. A "null sorter" keeps the same order as it is given.

Once a user has navigated to a particular requirement, the information associated with the requirement is organized and presented in four main groupings: details, attributes, associations, and problems. Each main grouping is presented in a notebook. Within each notebook (except the problems notebook), values can be browsed and transformed. The problems notebook indicates the problems with attributes or associations. When changes are made, the user can have a requirement writer save those changes to the file system or the user can revert by having a requirement reader re-read and replace the requirement(s).

Within a navigational view (tree or list), Decorators [5] are used to extend the user interface behavior of requirements folders, requirements packages, requirements, attributes, and associations. They enable presentation of meaningful icons and meaningful menus without cluttering these core classes. For example, a requirement that has an association to a file that doesn't exist has a problem. Decorators provide icons that differentiate between requirements that have problems and those that don't. Also, decorators provide menus that are appropriate for the state of the object it is decorating; a "Save" action is only provided if the object has been changed.

Graphical user interfaces are great for some activities. However, they are not very useful for other activities. For instance, if a project team wants to begin tracking a new attribute for their requirements, it would be very tedious and error-prone to add the new attribute by iterating through and transforming all of the requirements with a graphical user interface. Similarly, if a project team wants to know how many requirements it supported during its previous iteration, counting in the context of a graphical user interface is tedious and error-prone.

Therefore, the following core command-line tools can be combined together and extended to aid software developers in browsing and transforming requirements:

1. Take a set of requirement folders and generate a list of requirement files;
2. Take a list of requirement files and a filter definition and generate a list of requirements that pass through the filter;
3. Take a list of requirement files and a sorter definition and generate a sorted list of requirements;
4. Take a list of requirement files and process them in some custom fashion.

A utility that processes a list of requirement files needs to be configurable so that custom implementations of processors can be invoked. By identifying a class that implements the processor interface (Figure 6) and ensuring that it can be loaded, developers can extend the framework. Some processor implementations will be useful to the requirements community at large (such as generating a .pdf document from a list of requirements) whereas others will be throw-away (such as changing a specific word in the title of a list of requirements). The robustness of the processors should correspond to their scope of use.

```
public interface ProcessorI {  
    public void process(RequirementI[] inputArray);  
}
```

Figure 6 - Processors

These are the core components of a middleweight requirements framework that maintains the strengths and improves upon the weaknesses of the ultra-lightweight, lightweight, and heavyweight requirements management solutions.

Figure 7 shows an update of the strengths and weakness of the tool types.

Tool type	Simple	Flexible	Cheap	Scales			
				Backup	Concurrency	Filter/Sort	Compare
Ultra-lightweight	S	S	S	W	W	W	W
Lightweight	S	S	S	S	W	W	-
Middleweight	S	S	S	S	S	S	S
Heavyweight	W	W	W	S	S	S	S

Figure 7 – Strengths and weaknesses by tool type

Discussion

Potential Problems

Since we know there is no “silver-bullet,” [3] an obvious first question is: “what are the problems with a middleweight requirements management solution.” There are at least a few potential problems: (1) it is not quite as simple as “pen and paper;” (2) because it is so flexible, there is some inherent overhead in maintaining it; (3) because it is free, it is not supported; and (4) because it is new, it does not provide all of the “bells and whistles” of heavyweight tools.

Nothing is as simple as “pen and paper” – agreed. However, a middleweight solution need not replace a “pen and paper” solution to be useful; rather they are complementary solutions. “Pen and paper” solutions like Index Cards [6] are perfectly valid for activities where a level of stability has not yet been reached and/or personal interaction is desired.

Programmers often use a pen and a paper to write out a high level algorithm or draw a picture to help them write their code. However, they don’t stop there! They then formalize the code in a file and then share it with their team and have it backed up. Granted, code “executes” and requirements do not – but, “execution” is not the only reason that programmers write their code in files. Punch cards could still be used in place of files but they suffer from some of the same weaknesses as ultra-lightweight requirement tools.

Therefore, ultra-lightweight requirement tools should be used in conjunction with a middleweight requirements tool. Once requirements stabilize to the point where they can be shared with a team, they should be embodied in requirement files and placed under configuration management. In iterative efforts, the team will need to extract subsets of those requirements in the form of ultra-lightweight tools (such as Index Cards) so that the team can pass them around, write on them, sort them, etc. This can be facilitated by implementing a “requirement processor” that writes a document appropriately (for example: write 2 columns of requirements on a page, instead of the typical 1).

Another potential problem is that with flexibility comes complexity. As such, some complexity is introduced by the flexibility of a middleweight requirement tool. For example, “what types of attributes should we track on our project?” Or, “what set of requirement processors” do we need? Or, in the extreme, “how do I edit a file?”

Answering these questions and ensuring that the team has the appropriate code base should be the responsibility of a “requirements administrator.” This will ensure that the project wide decisions are made by an individual or a centralized group. In addition, it will isolate those who are not as familiar with software and files and free them to browse and transform the requirements.

Establishing such an administrator is a tradeoff that enables the requirements to be “scalable” artifacts and protects non-developers from complexity. Of course, this tradeoff has an associated cost. But so do tradeoffs related to ultra-lightweight or lightweight tools: costs of finding, changing, filtering, sorting, and

comparing requirements. And, obviously, heavyweight tools have associated costs. Thus, each project has to evaluate which type of requirements tool is appropriate for it.

“You get what you pay for,” or so the saying goes. An open source middleweight requirements tool has no guaranteed project specific support. Again, for some projects this may be a non-negotiable and disallow a non-heavyweight solution. However, open source project development tools are commonly used by software developers. And, depending on the sophistication of the development team, patches to open source products can be completed more quickly than patches to a supported product. Since there is a lot of overhead for supporting companies if they provide project specific support, we often have to wait for the next release of the product anyway. If we have the source code, we can change it and get our project going again.

Finally, a middleweight requirements tool is not as “feature rich” [10] as a heavyweight tool. That is, not yet. As the software community contributes to the effort supporting middleweight tools, the “features” will come (and without the cost or complexity of heavyweight products). But for now, it provides a balance between the ultra-lightweight and heavyweight tools, allowing requirements to be embodied in a simple, flexible, and cheap way that scales.

Experience

Although the survey mentioned in the introduction indicates that approximately 50% of requirements are captured in Word and Excel, during the 10+ years I’ve worked in the software development industry, my estimate is closer to 90%. That is, when the requirements are managed at all. For the reasons set forth in the strengths and weaknesses section, as well as my personal frustration with the alternatives, I started this effort.

In developing the middleweight framework, the only tools I used were the Eclipse development environment and the VisualAge for Java development environment. I developed the automated testing and domain code with Eclipse. I defined the user interface specifications for the standalone Swing client with VisualAge for Java. I followed an XP-like process [2].

At a high level, I recorded my requirements (in RML files rather than on index cards) and then I implemented my automated testing code and my domain code. As I thought of more requirements, I entered them in RML files – initially with a text editor and later with a Swing client. I used the requirements to plan my work by ensuring that each requirement had an associated state attribute (one of {unsupported, partially supported, supported}) and a priority attribute (an integer). I implemented filtering so I could see all of the unsupported requirements. I implemented sorting so I could see them in order of priority.

In this way, the tool helped me plan my work – it showed me the “user stories” to work on next. It also helped me keep track of what I had done and what I had left to do. In the same way that I used my automated component and unit tests to indicate what code I needed to change, I used my requirements to tell me what “behavior” I needed to design and add. And I did so without needing any additional tools on my computer.

So, based on my experience, a middleweight requirements tool is appropriate for a small scale effort (10s of requirements). For the reasons set forth in the paper, I believe it is even more appropriate for medium and large scale efforts (100s and 1000s of requirements).

Future Work

There are several main areas of future work related to the maturation of the middleweight framework and to research based on it. Briefly, they are:

In order to mature the middleweight framework, project teams on small, medium, and large scale projects must begin to use it. Experience reports from these projects will be the strongest indicator as to whether or

not middleweight requirements management tools can and do improve requirements management in the software development industry.

Next, needs derived from experiences using the middleweight framework will be used to drive the extension and refinement of the framework itself. Also, in order to provide a framework development perspective, a stable version of the framework will be used to capture and maintain these “needs” (requirements); one version of the tool will be used to drive the development of the next version.

In particular, the following extensions are expected: (1) integration into the Eclipse development environment so that programmers using it can have a “seamless” development environment; (2) a C# implementation of the framework so that requirements for C# projects can be managed by a tool written in the “native” language; (3) a web application to support browsing, transformation, and synchronization of the underlying files; and (4) integration of defect information such that defects are also stored in files and are associated with the appropriate requirements.

Finally, the middleweight requirements framework will act as a platform for requirements management research. Analysis tools will be developed that make measurements on sets of requirement files. The results of analyses will be used to prove (and disprove) hypotheses related to software development.

For example, consider the project velocity measurement derived in the XP process. Wouldn't it be nice to be able to have a computer derive this value (accurately and quickly) and compare it with values from similar projects? Or consider the backlog in the Scrum process [11]. Wouldn't it be nice to know what (if anything) corresponded to the “spikes” in the backlog last month? By placing the requirement data in an “open database,” (the file system), we should be able to innovate and gain a much better understanding of what is really important when we manage requirements on “real-world” projects.

Conclusion

This paper makes three main contributions to the requirements development community and the larger software development community:

- (1) It outlines the need for a middleweight requirements management solution – one that is simple, flexible, cheap in addition to scaling well;
- (2) It proposes an XML language for describing requirements, RML. RML enables a middleweight requirements management solution; and
- (3) It describes an extensible framework to make browsing and transformation of requirement files more effective.

References

- [1] IEEE Std 610.12-1990
- [2] Beck, Kent, Extreme Programming Explained, Addison-Wesley, 2000. ISBN 201-61641-6
- [3] Brooks, Fred, No Silver Bullet - Essence and Accidents of Software Engineering. IEEE Computer, 20(4):10-19, Apr. 1987
- [4] Eclipse: www.eclipse.org
- [5] Gamma, Helm, Johnson, Vlissides, Design Pattern-Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995, ISBN: 0201633612
- [6] Index Cards: <http://c2.com/cgi/wiki/IndexCards>
- [7] Managing Requirements Survey: <http://www.rational.com/products/reqpro/forms/survey.jsp>
- [8] Pressman, Roger, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1997, ISBN: 0-07-52182-4
- [9] Rational Requisite Pro, <http://www.rational.com/products/reqpro/>
- [10] Requirements Management Tool Survey, <http://www.incose.org/tools/tooltax.html>

[11] Scrum Development Process, <http://www.controlchaos.com/>

[12] Simplify: www.comlab.ox.ac.uk/simplify/index.html

[13] Telelogic DOORS, <http://www.telelogic.com/products/doorsers/doors/>