arXiv:0808.0347v1 [cs.SE] 3 Aug 2008

Towards a Process for Developing Maintenance Tools in Academia

Holger M. Kienle and Hausi A. Müller
University of Victoria
Victoria, Canada
{kienle,hausi}@cs.uvic.ca

Abstract

Building of tools—from simple prototypes to industrial-strength applications—is a pervasive activity in academic research. When proposing a new technique for software maintenance, effective tool support is typically required to demonstrate the feasibility and effectiveness of the approach. However, even though tool building is both pervasive and requiring significant time and effort, it is still pursued in an ad hoc manner. In fact, little research has addressed the question how to make tool building in academia more disciplined, predictable and efficient. In this paper, we address these issues by proposing a dedicated development process for tool building that takes the unique characteristics of an academic research environment into account. We first identify process requirements based on a review of the literature and our extensive tool building experience in the domain of maintenance tools. We then outline a process framework based on work products that accommodates the requirements while providing needed flexibility for tailoring the process to account for specific tool building approaches and project constraints. The work products are concrete milestones of the process, tracking progress, rationalizing (design) decisions, and documenting the current state of the tool building project. Thus, the work products provide important input for strategic project decisions and rapid initiation of new team members. Leveraging a dedicated tool building process promises tools that are designed, build, and maintained in a more disciplined, predictable and efficient manner.

1. Introduction

Typical research in the domains of software maintenance and reverse engineering proposes techniques to improve the comprehension of software systems. In order to evaluate a proposed technique and to demonstrate its feasibility many researchers implement a tool. Such a tool can range from a simple proof-of-concept prototype to a fully-fledged, industrial-strength application.

This paper explores the current state of tool building practices in academia with the aim to improve upon the state-of-the-art. This topic is worthwhile to address because even though tool building is a popular technique to validate research in software engineering, it is neither simple nor cheap to accomplish. For example, tools such as the Rigi and Moose reverse engineering environments require significant resources to develop and maintain. Rigi is now being maintained for over a decade and this effort has been accomplished by a number of students at the Master and Ph.D. level as well as occasionally by dedicated staff members. In this context the resulting constant turn-over of (unexperienced) developers is a concern. Nierstrasz et al., who have developed the Moose reengineering tool, observe that "crafting a tool requires engineering expertise and effort, which consumes valuable research resources" [30]. The transfer of engineering expertise in the domain of tool building has to be addressed within each development project. Furthermore, it also should be addressed within the tool building community in order to communicate the state-of-the-art and to further improve upon it. Currently, the communication of engineering expertise to the community is not sufficiently rewarded: research contributions in conferences and journals are measured by novelty, not by synthesis of existing work and experiences.

Having a process when building a maintenance tool is especially desirable if it is a long-running project that has a significant turnover of (student) developers. For longer-term research this is often the case. Furthermore, to show the effectiveness of these tools they should be sufficiently stable and mature to serve in larger-scale (industrial) user studies. Building a high-quality tool without following a process exposes the research project to unnecessary risks.

Improving on the current tool building practice promises tools that are designed, build, and maintained in a more disciplined, predictable and efficient manner. Furthermore, practices can emphasize certain non-functional tool requirements such as usability and adoptability, or a certain approach to tool building such as component-based development. In this paper we advocate to employ an explicit tool building process to raise the state-of-the-art.

This paper is organized as follows. Section 2 gives further background on tool building in academia, concluding that it is executed in an ad hoc manner and that a suitable process can improve upon the state-of-the-art. We explore then two more disciplined approaches to tool building that have been pursued in the construction and maintenance of SHriMP and TkSee. Section 3 presents the requirements that are desirable for a tool building process in an academic environment. These requirements have been distilled with the help of a literature survey as well as from our own tool building experiences. The identified requirements can, on the one hand, provide important background information and constraints for developing an appropriate tool-building method, and, on the other hand, serve as evaluation criteria to judge the efficacy of a proposed tool-building method. Section 4 is a first step towards a dedicated process for building tools. We advocate a flexible process framework based on work products that can be easily tailored to accommodate different needs. Importantly, the work products that we are proposing satisfy our identified process requirements. Section 5 closes the paper with conclusions and future work.

2. Background and Related Work

Research in tool support for software maintenance is actively pursued by many academics in software engineering. In the following, we view such tools as *maintenance tools* that offer functionalities for the comprehension and analysis of a target system with the goal to support engineers in the performance of maintenance tasks. Examples of such tools are software visualizers, bug trackers and slicers as well as recommender systems, search and metrics engines. In this paper we focus mostly on reverse engineering and graph-based software visualization tools because it is our main expertise and interest. Given that the construction of maintenance tools is a pervasive activity in academic research, surprisingly little work has focused on how tools are built and how to improve upon the current practice.

Some researchers have published experiences about tool building. For example, Lanza describes his experiences with the CodeCrawler software visualizer [22], discussing CodeCrawler's architecture (composed of three subsystems: core, metamodel, and visualization engine), the visualization engine (realized by extending the HotDraw framework), and desirable interactive mechanisms for usability. Furthermore, he distills lessons learned for all of the discussed issues. He observes that "to our knowledge there is no explicit work about the implementation and architecture of reverse engineering tools, and more specifically about software visualization tools." Guéhéneuc describes his use of design patterns and Java language idioms when constructing the Ptidej tool suite [12]. The special issue on Experimental Software and Toolkits (EST) [45] of Elsevier's *Science in Computer Programming* journal is devoted to the description of academic research tools. One tool of the special issue, g^{4}re, falls within the maintenance domain [20].

Besides the published tool building experiences discussed above, researchers have also identified requirements for maintenance and reverse engineering tools [19, sec. 3.2]. For example, Tichelaar discusses six requirements for reengineering environments [43, sec. 5.1]. Wong has distilled 23 high-level and 13 data requirements for software understanding and reverse engineering tools [46]. He also recommends to "summarize and distill lessons learned from reverse engineering experience to derive requirements for the next generation of tools."

The growing interest of researchers to report their tool building experiences is a positive development that should be further strengthened within the research community. However, so far the focus is almost exclusively on documenting engineering expertise, but not on process. Unfortunately, most researchers do not report at all about their tool development process. It seems that researchers often develop their tools by themselves and are the only users of the tool during and after development. Tools are then evaluated with a case study or anecdotal evidence. As a result, the building of tools resembles a *craft* rather than *professional engineering* [34].

One approach to professionalize tool building is to follow a process. Indeed, all software development projects should use a well-defined process—tool-building in an academic environment is no exception to this rule. According to Kruchten, without such a process the "development team will develop in an ad hoc manner, with successes relying on the heroic efforts of a few dedicated individual contributors. This is not a sustainable condition" [21, p. 15]. A process should provide guidance on what work products should be produced when, how, and by whom. A well-defined process enables a repeatable and predictable way of building software systems. There are a few examples of researchers that touch on process aspects when relating their experiences. The following sections give examples of two approaches to tool building—SHriMP and TkSee—that can provide valuable input towards identifying process requirements. These can in turn be used when defining a suitable process for the domain of academic tool building. However, to our knowledge, no full process has been proposed so far. The closest research that we are aware of is Chirouze et al.'s work on Extreme Researching (XR), which is a process specifically designed for applied research and development in a distributed environment that has to cope with changing human resources and rapid prototyping [5]. It is an adaptation of Extreme Programming (XP) and has been developed by Ericsson Applied Research Laboratories to support distributed telecommunications research. XR is based on several core principles taken from XP (short development cycles, test-

driven development, collective ownership, and discipline) and encodes them in a set of activities (e.g., remote pair programming, unit testing, collective knowledge, coding standards, frequent integration, and metaphor). Dedicated tool support is available for XR with a web-based portal. Based on three projects, the authors of XR estimate that their process has yielded an increase of output of around 24% and reduced project-overrun time on average by half.

2.1. Tool Building in SHriMP and TkSee

The development history of the **SHriMP** program comprehension tool is one of the few more comprehensive sources that allow us to infer requirements for tool-building.

The SHriMP project is an example of building a tool that has been continuously refined and evaluated, following an iterative approach of designing and implementing tools, which has been proposed by the team's leading professor [38, sec. 11.1.1]. Evolving a tool such as SHriMP is a major research commitment, involving several students at the Ph.D. and master level at any given time. SHriMP had several major implementations. The first implementation was based on Rigi (with Tcl/Tk and C/C++), followed by a Java port. The Java version of SHriMP was then re-architected in a component-based technology, JavaBeans, to facilitate closer collaboration between reverse engineering research groups.

The Rigi-based version of SHriMP was evaluated and improved based on two user studies with 12 and 30 participants, respectively [41] [42]. The results of both user studies helped to improve SHriMP by identifying several shortcoming. Wu and Storey have also published the results of their first Java port of SHriMP; they state that "during the development of this prototype, we took into consideration the knowledge from previous prototypes and empirical evaluations" [47]. Even though they do not provide further details, we can infer that their development process is (1) prototype-based, (2) iterative, and (3) based on feedback from empirical evaluations.

The **TkSee** search tool (written in Tcl/Tk and C++) is an example of a tool that has been improved based on the feedback obtained from developers in industry. Early versions of TkSee were delivered to users at Mitel in 1996. At the time, the tool was used by relatively few user, which used only a small subset of its features. Generally, users were reluctant to adopt new tools. Lethbridge and Herrera describe TkSee's development process as follows [25, p. 89]:

> "TkSee had been developed in a university research environment following an informal and opportunistic development process. Features had been added by students and researchers when they had had bright ideas they wished to experiment with, hence it lacked complete documents describing its requirements, its design (except that of its database architecture [24]) and how to use it. There was considerable staff turnover among TkSee developers because many were students. ... The newer staff was often not able to understand the tool or the purpose of certain features.

To improve TkSee, feedback was obtained from field observations as well as formal user studies using think-aloud protocol and videotaping. The results were then communicated to the tool developers in a report and sessions with video clips to "emphasize certain points and convince them of the seriousness" of the (usability) problems. Similar to SHriMP, we can conclude that tool development had at least one iteration that improved the tool based on feedback from a user study. Furthermore, the use of Tcl/Tk would facilitate rapid prototyping of TkSee.

The approaches to tool building of both SHriMP and TkSee provide valuable input to identify requirements that a dedicated process for tool building should exhibit.

3. Process Requirements

This section introduces requirements for a software process to develop maintenance tools. A dedicated development process has to accommodate the particular characteristics and constraints of the target domain. For instance, tool development in an academic research environment often is ad hoc and unstructured. Tools are often constructed by students that have only a few years of programming experience, and that typically work alone or in small teams with informal communication flow and without an explicit process. Furthermore, they often work not closely supervised and are evaluated based on their finished product, not on how they have constructed it.

Researchers in the domain have reported some experiences that allow to distill properties that an appropriate development process should probably possess. We also draw from informal discussions with other researchers in the maintenance domain, and from our own experiences of developing software visualization and reverse engineering tools. Our own experiences include the development and maintenance of the Rigi and Bauhaus tools as well as the tools constructed in the context of the Adoption-Centric Software Engineering (ACSE) project [19].

The following sections discuss the requirements that we have been able to identify for a tool building process in an academic research setting.

3.1. Feedback-Based

Many ideas for improvements of software systems originate from their users. There is evidence that software development projects are the more successful, the more *user-developer links* they have [18]. These links are defined as the techniques or channels that allow users and developers to exchange information; examples are surveys, user-interface prototypes, requirements prototypes, interviews, bulletin boards, usability labs, and observational studies.

As with many other software development projects, often the researchers developing a research tool have a poor initial understanding of its requirements. Requirements

elicitation is difficult, for instance, because the target users are ill-defined and the tool's functionality might be not fully understood yet. To alleviate this problem and to bootstrap the first release, Singer and Lethbridge propose to first involve maintenance engineers (who will later use the tool) to understand their particular needs and problem domain before starting the design and implementation of the tool [36]. This can be achieved with questionnaires, (structured) interviews, observation sessions, and (automated) logging of tool use. For instance, Lethbridge and Singer have used a simple, informal approach to elicit initial tool requirements from future users [35]:

> "For the first release, we brainstormed a group of software engineers for their needs, and then designed, with their continued involvement, a tool called SEE (Software Exploration Environment)."

Once a research tool has reached a first beta release, feedback should be obtained. Lethbridge and Singer follow a process with two main phases: (1) study a significant number of target users to thoroughly understand the nature of their work, and then (2) develop and evaluate tools to help the target users work better. Importantly, the second phase involves "ongoing involvement with [target users]" [26]. User feedback can be obtained, for instance, with (longitudinal) work practice studies of individuals or groups. Such studies observe and record activities of maintainers in their normal work environment, working on real tasks. The designers of the Augur software exploration tool, for instance, report that "to gain some initial feedback on Augur's effectiveness, we have conducted informal evaluations with developers engaged in active development" [9]. Hundhausen and Douglas have used the finding of an ethnographic study with students in a course (involving techniques such as participant observation, semi-structured interviews, and videotape analysis) to redefine the requirements for their algorithm visualization tool [16].

Wong states that "case studies, user experiments, and surveys are all necessary to assess the effectiveness of a reverse engineering tool" [46, p. 93]. Many researchers conduct informal case studies of their tools by themselves,[1] considering themselves as typical users [44]. However, it is not clear whether this approach can generate the necessary feedback to further improve a tool. A more effective approach is user studies. In the best case, feedback is provided by the actual or potential future users of the tool; however, obtaining feedback from this type of user is often impossible. As an alternative, researchers use other—presumably "similar"—subjects such as computer science students. For example, Storey conducted a study with 12 students to assess a new visualization technique introduced with the SHriMP tool. Observations and user comments resulting from the study generated several improvements [41, sec. 5.3]. Storey explains her strategy as follows:

> "We are currently planning further user studies to evaluate the SHriMP and Rigi interfaces. Observations from these studies will be used to refine and strengthen the framework of cognitive design issues which will, in turn, be used to improve subsequent redesigns of the SHriMP interface" [40].

In another user study involving students, researchers did ask 13 questions about their visualization tool, sv3D, with the goal "to gather feedback from first time users about some of the sv3D features" [27]. They conclude that the users' "answers provided us with valuable information that supports the implementation of new features in sv3D."

To summarize, researchers have tried to obtain feedback from different types of subjects (e.g., professionals who will use the tool, students who substitute for "real" users of the tool, and, last but not least, themselves) as well as with different methods (e.g., case studies, field studies, surveys, and formal experiments).

3.2. Iterative

Iterative software development processes—as opposed to processes that follow a waterfall or sequential strategy—are an approach to building software in which the overall life-cycle is composed of a sequence of iterations. Boehm's spiral model, the Rational Unified Process (RUP), and Extreme Programming are examples of iterative software development processes. Developing software iteratively is among the six best practices identified by Kruchten [21].

The waterfall model provides minimal opportunity for prototyping and iterative design [11], but "is fine for small projects that have few risks and use a well-known technology and domain, but it cannot be stretched to fit projects that are long or involve a high degree of novelty or risk" [21, p. 75]. As illustrated by Rigi and SHriMP, tool-building projects can run for several years. Since research tools typically explore new ideas and strive to advance the state-of-the-art, they are risky, potentially involving new algorithms and techniques. Thus, the waterfall approach is not suitable for the development of research tools.

Iterative software development has several benefits compared to the waterfall model; especially important benefits in the context of tool building are that user feedback is encouraged, making it possible to elicit the tool's real requirements, and that each iteration results in an executable tool release that can be evaluated. Generally, iterative development is most suitable to cope with requirements changes [23]. As in other software projects, requirements in tool-building can change frequently (e.g., because of user-feedback and modified hypothesis).

A particular example of the iterative nature of tool development is provided by the development of a schema and corresponding fact extractor. A schema embodies a certain domain (e.g., C++ code or software architecture), which has to be iteratively refined. Changes in the schema trigger changes in the fact extractor, and vice versa [29].

[1] For instance, in a survey of twelve visualization tools, two were evaluated with user studies and eleven with a case study [39].

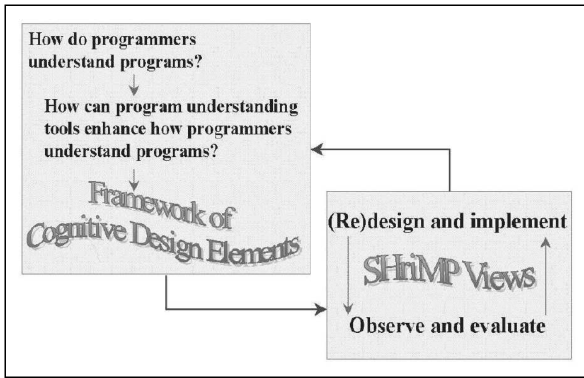

Figure 1. Storey's tool-building process [38]

As an outcome of the SHriMP tool, Storey proposes an iterative approach for designing and implementing tools [38]. It is a process consisting of "several iterative phases of design, development and evaluation." Figure 1 depicts a rendering of the process iterations, illustrated with SHriMP as the subject. There is an "iterative cycle of design and test" that aims at improving a tool [40]. The (initial) design of the tool is guided by the cognitive design elements framework, which provides a catalog of issues that should be addressed by software exploration tools. Testing of tool design and implementation can be accomplished by user studies. This cycle is embedded in a larger iterative cycle for improving the framework. When adopting Storey's process, the larger cycle could be omitted or replaced with a different framework.

Similar to Storey, Wong argues for an iterative approach when building reverse engineering tools. He explains the process with his proposed Reverse Engineering Notebook:

> "The Notebook research itself undergoes continuous evolution. The evolution follows a spiral model ... Each iteration of the Notebook involves several states, including studies with users, evaluating alternative technologies, considering compatibility constraints and adoption risks, validating the product, and planning for the next phase" [46, p. 99].

3.3. Prototype-Based

The construction of prototypes is common in engineering fields (e.g., scaled-down models in civil engineering). For software construction, prototyping is the "development of a preliminary version of a software system in order to allow certain aspects of that system to be investigated" [33]. The prototype typically lacks functionality and does not meet all non-functional requirements. Rapid prototyping is the use of tools or environments that support prototype construction. Examples of technologies that facilitate rapid prototyping are weakly-typed (scripting) languages with support for GUI construction (e.g., Tcl/Tk and Smalltalk), tools for rapid application development (e.g., IBM's Lotus Notes), and domain-specific prototyping environments and languages (e.g., Matlab).

A throwaway prototype produces a cheap and rough version of the envisioned system early on in the project—it need not be executable and can be a paper simulation or storyboard [4]. Such a prototype is useful to obtain and validate user requirements and is then discarded once the initial requirements have been established. Exploratory prototypes are typically throwaway prototypes that are "designed to be like a small experiment to test a key assumption involving functionality or technology or both" [21, p. 161]. As opposed to the construction of a throwaway prototype, an evolutionary approach to prototyping continuously evolves the prototype into the final product.

Using prototypes in software development has a number of benefits. They are effective for defect and risk avoidance, and for uncovering potential problems (e.g., verification of the proposed system architecture, trying out a new technology or algorithm, identification of performance bottlenecks, or exploration of alternative designs) [3]. Yang has used rapid prototyping to develop a tool called the Maintainer's Assistant; he describes several benefits:

> "There are several advantages of rapid prototyping which can be taken to develop the Maintainer's Assistant itself. For instance, the system can be developed much faster by rapid prototyping, so that it can speed up the implementation. The user is involved in the process, so that he or she (mainly the builder in our system) can determine if the development is moving in the right direction. Rapid prototyping produces a model of the final system, so that the user, as well as builder, can see the final system in the early development stage" [48].

Prototyping has potential drawbacks also. In contrast to a description of a system's behavior, executable prototypes are well suited to represent what a system does. However, they are less well suited to capture design rationale (i.e., why was the system built and why in this way)—this should be documented explicitly [32]. There is also the threat that a prototype that was originally planned to be thrown away is evolved into the final product [10]. Evolutionary prototyping is then "merely the official name given for poor development practices where initial attempts at development are kept rather than thrown away and restarted" [23].

Prototypes are well suited for applied research—Chirouze et al. go as far as stating that "the idea of rapid prototyping and researching is intrinsically linked" [5]. Wong proposes an iterative development strategy for the building of maintenance tools. In this approach, "each iteration builds an evolutionary prototype that is intended to be a solid foundation for the next iteration" [46, p. 99].

Prototypes have been successfully used in the building of software engineering tools (e.g., using Visual Basic [15], and Tcl/Tk [41]). A tool prototype can serve as an initial proof-of-concept. Such a prototype does not need to fulfill certain requirements. For instance, a prototype can demonstrate the usefulness of a novel algorithm without meeting

the scalability requirements of a production tool. However, if user studies are based on the prototype, it might have to meet much higher quality standards. For example, Storey et al. report that

> "The first prototype of the SHriMP interface was implemented in Tcl/Tk. Tcl/Tk is a scripting language and user interface library useful for rapidly prototyping graphical interfaces. However, its graphics capabilities are not optimized for efficiently displaying the large graphs typical of software systems. The second prototype has been implemented using Pad++, a graphics extension for Tcl/Tk. Pad++ is highly optimized for efficiently displaying large numbers of objects and smoothly animating the motions of panning and zooming" [40].

3.4. Other Requirements

Based on our experiences, we believe that the following requirements should be considered for a tool-building process as well:

lightweight: The notion of a lightweight process is not clearly defined. However, evidence of a lightweight process is that it strives to minimize (intermediate) artifacts such as vision statement, use-case model, and status assessment, recognizing that these artifacts are not the goal of a process, but a means to an end. Martin says, "a good process will decrease the demand for such intermediate artifacts to the barest possible minimum" [28]. Extreme Programming has four core values from which twelve practices are derived; one of these values is simplicity [2]. The SEI's Personal Software Process Body of Knowledge states that "a process description should be brief and succinct" [31, Key Concept 1.1.1]. A goal of configuring RUP is that the process "must be made as lean as possible while still fulfilling its mission to rapidly produce predictably high-quality software" [21, p. 31]. Cockburn introduces two factors that determine a method's weight: *method size* (i.e., "number of control elements, including deliverables, standards, activities, milestones, quality measures, and so on") and *method density* (i.e., "the detail and consistency required in the elements") [6]. He says, "a relatively small increase in methodology size or density adds a relatively large amount to the project cost" [6]. Agile methods are typically more lightweight than plan-driven approaches.

component-based: Component-based software development (CBSD) promises lower development costs and higher productivity compared to implementing systems from scratch [14]. Researchers often rely on (off-the-shelf) components and products when constructing tools. For example, fact extraction of source code has been realized by leveraging Reasoning Refine, GNU GCC, Eclipse, SNIFF+ and Source Navigator, and visualizations of software structures have been implemented on top of AT&T gaphviz, Rational Rose, Adobe FrameMaker, Microsoft Visio, and Web browsers. However, following a component-based approach for tool building has unique challenges. It is necessary, for instance, to evaluate candidate components in terms of functionality, interoperability, and customizability. When realizing software visualization functionality with components, Lanza cautions that "reusing graph visualization tools and libraries like [graphviz] can break down the implementation time, but it can also introduce new problems like lack of control, interactivity and customizability. . . . The first experiments we did with external engines soon reached a limit, because they were not customizable and flexible enough for our needs" [22]. In order to meet such challenges, a process should explicitly address CBSD issues.

adaptive: A process should be flexible enough to accommodate changing requirements of the system under construction. However, evolving requirements—or other changes in business, customer, or technological needs—might make it necessary to adapt the process itself during the development effort. The SEI's Personal Software Process Body of Knowledge states that "a quality process must be easy to learn and adapt to new circumstances" [31, Key Concept 5.1.4]. Fowler describes the adaptive nature of a process as follows, "A project that begins using an adaptive process won't have the same process a year later. Over time, the team will find what works for them, and alter the process to fit" [8]. A process can be adapted after an iteration as the result of a process review.

Even though we could not find explicit evidence for these process requirements in the literature, we believe that they should be part of an effective tool-building approach in academia.

4. Process Framework for Tool Building

To advance the goal of an effective tool building approach for academia, it is necessary to provide developers with a suitable development process (i.e., it has to meet the process requirements of the previous section). This process needs to encode guidelines on how to build tools in a repeatable and predictable way.

The dilemma with proposing a process is as follows: On the one hand, we need a development process for tool building; on the other hand, the individual projects seem too diverse to be accommodated by a single, one-size-fits-all process. Each individual tool-building project has its own unique characteristics such as the tool's requirements and functionality, the degree of technical uncertainty, the complexity and size of the problem, the number of developers, the background of the development team, and so on. This is especially the case for tool-building projects in academia, which can differ significantly from each other.

To resolve this dilemma, we do not define and mandate a full process; instead we are proposing a *process framework*. This framework addresses the tool-building requirements discussed in Section 3, but needs to be instantiated by researchers to account for the unique characteristics of their own development projects. The process framework is composed of a set of work products (WPs). A WP is "any planned, concrete result of the development process" [17].

The process framework's focus on WPs is inspired by IBM OOTC's process, which is described in the book *Developing Object-Oriented Software: An Experience-Based Approach* [17]. It focuses on WPs because often there is agreement on *what* should be produced, but not on *how* it should be produced. The developers of the process say, "it was decided that an approach that standardized on work products but allowed for individual choices on particular techniques used to produce work products was the best fit" [17, p. 4]. As a result, concrete WPs provide necessary guidance for tool builders without unnecessarily constraining them.

4.1. Work Products

We define seven core WPs that reflect important stages in tool development, ranging from requirements elicitation, over architecture, to prototype construction. The core WPs of the process framework address specific issues of the domain. We purposely omit more generic WPs that address issues such as implementation and testing.

Because of limited space we can only briefly describe each WP; a more detailed description of each WP is available in the first author's dissertation [19, sec. 7.2]. We also give example of interactions between WPs to illustrate that WPs support each other: one WP can provide valuable input for another WP.

Intended Development Process: This WP instantiates a suitable tool development process, which should meet our identified process requirements and account for project-specific characteristics. For example, a project that wants to employ components should reflect this practice by accommodating CBSD principles (such as guidance in selecting and adapting suitable components, and in assembling the component-based system). To support these tasks, the process framework already offers two WPs: Candidate Components and Tool Architecture.

Functional Requirements: This work product captures the users' expectations of the tool's functionality, providing a basis for communication between users and developers, and enabling to estimate development effort. For maintenance tools, functionality will entail fact extraction, analysis, and visualization. These functional units are typically exposed in the tool's architecture (see Tool Architecture WP). This WP can also contain a feature list, which can be bootstrapped from tool comparisons published in the literature (e.g., [1]).

Non-functional Requirements: This WP describes the tool's quality attributes. Tools should address important domain requirements such as scalability, interoperability, customizability, usability, and adoptability. Such requirements are often difficult to formalize and validate. There should be a rationalization for each of these quality attributes on how the tool will meet them. Obtaining user feedback with prototypes (see Technical and UI Prototype WPs) can be used in such cases to clarify quality attributes.

Candidate Components: This WP addresses the identification, assessment, and selection of (off-the-shelf) components and products that can be leveraged when building the tool. To enable a comparison among candidates, assessment criteria (reflecting the Functional and Non-functional Requirements WPs) have to be defined first. Documenting the rationale for selecting a certain component increases the confidence of the tool developers into the assessment and selection. Note that the selection decision may come to the conclusion that there is no suitable candidate component in which case the required functionality has to be implemented from scratch.

User Interface Prototype: This WP mandates the construction of a UI prototype. A typical objective for a prototype is that it should be good enough to enable user interaction and feedback; and detailed and complete enough to support some kind of evaluation. The Functional and Non-functional Requirements WPs can be used to identify the tool's "main line" functionalities that the prototype should focus on. If a candidate component is available and this component offers scripting support then it can be used as a rapid prototyping environment. The development of the prototype can give the developers valuable first insights for the actual tool development.

Technical Prototype: This WP is concerned with the construction of a prototype to explore issues related to the design, implementation, and architecture of the tool under construction. This is in contrast to the UI Prototype WP, which is exclusively concerned with user-interface design. Prototyping can be used as a risk mitigation technique to resolve or explore uncertainties of tool development that cannot be addressed by theoretical analysis or research alone. In the context of CBSD, exploratory prototypes are especially useful to provide input for the Candidate Components WP because public information about components can be inaccurate, misleading, or outdated.

Tool Architecture: This WP documents the tool's high-level architecture. It is discussed in more detail in the next section.

The full description of each WP follows a template to provide a common structure. This template consists of description, purpose, technique, advice and guidance, and references to related work. In the following section, a more complete and detailed description following the template is given.

4.2. Sample Work Product: Tool Architecture

description: The Tool Architecture WP captures the system architecture of the tool under construction. It can be seen as the set of early, global design decisions. The architecture places constraints on the tool's design and implementation. The architecture is often visualized with diagrams, but it can also take the form of a textual description of non-functional requirements and derived architectural decisions.

purpose: An architecture can be used to show the partitioning of the system into components; the communication patterns and mechanisms between components; the nature and services of the used components; etc. Without an architecture, it may be difficult to reason about tool properties, to communicate its design principles to new project members, and to maintain its (conceptual) integrity. The architecture can be used to reason about certain Non-functional Requirements of the system (e.g., performance, modifiability, and reusability).

technique: The tool's architecture can be described as a conceptual architecture diagram that shows the tool's main components. Often a single customized candidate component implements the functionality associated with a certain tool functionality (i.e., extractor, analyzer, visualizer, or repository). In the early project stage, a component in the diagram may indicate its intended functionality without revealing its realization. For example, there may be a component that is meant to implement repository functionality. Only later on in the project a decision will be made about the nature of the repository (e.g., local file system, relational database, or XML database). Still later on, the actual component may be chosen.

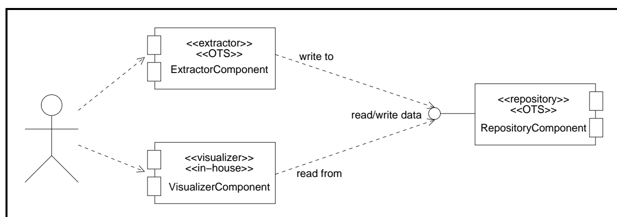

Figure 2. A tool architecture in UML

Architectural diagrams are typically visualized with UML. UML stereotypes can be used to convey additional meaning. For example, Egyed et al. use stereotypes to distinguish between `<<in-house>>` and `<<COTS>>` components [7]. Similarly, stereotypes can be used to identify the tool component functionality. Figure 2 shows an example of a tool's architecture modeled in UML. The tool is composed of three components. The repository component exposes an interface (shown in UML's shorthand ("lollipop") interface notation) to read and write data. The extractor and visualizer components use this interface to write and read data, respectively. Stereotypes are used to indicate the tool component types, and to distinguish between third-party and custom components.

advice and guidance: The tool architecture can be effective to assess whether the resulting tool meets established design principles. For example, a component-based system will be more maintainable if it minimizes coupling among components and relies on open standards. Boehm et al.'s MBASE approach identifies issues—grouped into *simplifiers* and *complicators*—that make the development of component-based systems easier or harder [37]. Simplifiers that are reflected in the architecture include "clean, well-defined APIs" and "a single COTS package;" complicators include "multiple, incompatible COTS packages."

To come up with an architecture it is often helpful to reuse existing expertise in the form of reference models and architectures, architectural patterns, or architectural mechanisms [21]. Several researchers have proposed reference models and architectures for reverse engineering tools (e.g., [24] [13]), which can be used as a starting point to define a tool's architecture.

references: The OOTC process defines System Architecture and Subsystem Model WPs; there is also a Rejected Design Alternatives work product that can be used to record why a certain architecture was not chosen. RUP defines two artifacts related to architecture: Software Architecture Document and Architectural Prototype [21, p. 86].

4.3. Discussion

The WP oriented approach of the proposed process framework has been inspired by OOTC. However, the framework's individual WPs are not the same as the ones described by OOTC because they are motivated by the process constraints and characteristics of our domain.

The process framework describes a minimum set of WPs. A tailored version of the framework is free to introduce additional ones, as appropriate. Specifically, a tool-building project that experiments with a novel or unproven technology could introduce dedicated WPs to make sure this risk is addressed adequately. For example, if a new tool wanted to employ XML databases as a repository—an unfamiliar approach to the project members—a Repository

Technical Prototype WP could be defined. This WP could prescribe tasks such as creation of a sample schema, and benchmarking with realistic data sets. The process framework has a number of desirable characteristics:

lightweight: We have argued before that a tool-building process should be lightweight. In keeping with the requirement of a lightweight process, the process framework itself defines only a minimal set of WPs, and each WP should be as concise as possible (i.e., rather 2–3 pages than 10–20 pages). The descriptions of the WPs in the process framework are also kept concise (around 1000 words each) to foster a rapid deployment of the process.

adaptive: Adaptive processes are desirable for tool-building projects because they often have an unstable environment (e.g., the composition and size of the development team can change considerably during its life-time). The WP orientation and separation of concerns of the process framework "addresses the risk of losing ground when tools, notations, techniques, method, or process need to be adjusted by allowing us to vary them while maintaining the essence and value of completed work" [17, p. 15].

tailorable: The framework is tailorable in the sense that researchers are free to extend the process framework with their own WPs. Researchers can also omit WPs if their tool-building approach has different process constraints. For instance, a research project may wish to incorporate an explicit traceability requirement for its process, or to eliminate the Technical Prototype WP if they have sufficient understanding about the technology.

separation of concerns: The WP oriented approach leads to a separation of concerns: WPs are independent from tools and notation as well as development process and techniques.

reuse: The process framework defines a reusable set of WPs. These can be reused by new team members to rapidly understand the current state of a tool project, and by other researchers to jump-start their own tool-building projects—thus, WPs can be seen as preserving valuable tool building knowledge across projects and people. In the best case, WPs lead to a situation where existing WPs are reused whenever possible, reserving the definition of new WPs for genuinely new contexts [4].

5. Conclusions and Future Work

In this paper we strive to advance academic tool building beyond the craft stage. What is needed is a more predicable approach that provides processes, techniques, and guidance to tool builders. The identified process requirements (i.e., feedback-based, iterative, prototype-based, lightweight, component-based, and adaptive) and the outlined process framework (consisting of seven core work products) are a first step in the direction towards the professional engineering stage. The process framework specifically addresses the process requirements of the domain of maintenance tool building, and an academic environment. Importantly, we have grounded the process requirements by analyzing the domain with a literature review of relevant tool building experiences.

It is an open question whether the proposed process framework is suitable for similar domains. It seems likely that the process framework generalizes to the whole domain of software engineering tool building and to environments that are similar to academic research such as research labs. However, one should be careful to jump to conclusions without a thorough investigation of the target domain. We hope that other researchers will investigate the suitability of our process framework in their domains.

In future work we want to apply our process framework in the construction of maintenance tools. It seems especially promising to first use the process on Master students implementing a relatively small tool with well defined scope. The work products that are created by the students during the implementation effort can be used to document progress, record constraints, and rationalize decision. Furthermore, the work products can be used to record tool-building experiences as an effective means to communicate important lesson's learned to subsequent generations of students that begin to develop tools under similar constraints. Lastly, we want to explore tool support for defining and evolving the process framework and its work products. Such tool support could be implemented on top of IBM Rational Method Composer.

References

[1] S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. *9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 7–17, May 2001.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[3] B. Boehm. Making RAD work for your project. *IEEE Computer*, 32(3):113–115, Mar. 1999.

[4] J. Cameron. Configurable development processes. *Communications of the ACM*, 45(3):72–77, Mar. 2002.

[5] O. Chirouze, D. Cleary, and G. G. Mitchell. A software methodology for applied research: eXtreme Researching. *Software—Practice and Experience*, 35(15):1441–1454, Dec. 2005.

[6] A. Cockburn. Selecting a project's methodology. *IEEE Software*, 17(4):64–71, July/Aug. 2000.

[7] A. Egyed, S. Johann, and R. Balzer. Data and state synchronicity problems while integrating COTS software into systems. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 69–74, May 2004.

[8] M. Fowler. The new methodology. *MartinFowler.com*, Apr. 2003. http://www.martinfowler.com/articles/newMethodology.html.

[9] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 387–396, May 2004.

[10] V. S. Gordon and J. M. Bieman. Rapid prototyping: Lessons learned. *IEEE Software*, 12(1):85–95, Jan. 1995.
[11] J. Grudin. Interactive systems: Bridging the gaps between developers and users. *IEEE Computer*, 24(4):59–69, Apr. 1991.
[12] Y.-G. Guéhéneuc. Ptidej: Promoting patterns with patterns. *1st ECOOP workshop on Building a System using Patterns*, July 2005. http://www.iro.umontreal.ca/~ptidej/Publications/Documents/ECOOP05BSUP.doc.pdf.
[13] J. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Requirements for information system reverse engineering support. *2nd IEEE Working Conference on Reverse Engineering (WCRE'95)*, pages 136–145, July 1995.
[14] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
[15] M. E. C. Hull, P. N. Nicholl, P. Houston, and N. Rooney. Towards a visual approach for component-based software development. *Software – Concepts & Tools*, 19(4):154–160, Aug. 2000.
[16] C. Hundhausen and S. Douglas. A language and system for constructing and presenting low fidelity algorithm visualizations. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 227–240. Springer-Verlag, 2002.
[17] IBM Object-Oriented Technology Center. *Developing Object-Oriented Software: An Experience-Based Approach*. Prentice Hall, 1997.
[18] M. Keil and E. Carmel. Customer-developer links in software development. *Communications of the ACM*, 38(5):33–44, May 1995.
[19] H. M. Kienle. *Building Reverse Engineering Tools with Software Components*. PhD thesis, Department of Computer Science, University of Victoria, Nov. 2006. https://dspace.library.uvic.ca:8443/dspace/handle/1828/115.
[20] N. A. Kraft, B. A. Malloy, and J. F. Power. A tool chain for reverse engineering c++ applications. *Science of Computer Programming*, 69(1–3):3–13, Dec. 2007.
[21] P. Kruchten. *The Rational Unified Process: an introduction*. Object Technology Series. Addison-Wesley, 1999.
[22] M. Lanza. Codecrawler—lessons learned in building a software visualization tool. *7th IEEE European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 1–10, Mar. 2003.
[23] P. A. Laplante and C. J. Neill. Opinion: The demise of the waterfall model is imminent. *ACM Queue*, 1(10):10–15, Feb. 2004.
[24] T. C. Lethbridge and N. Anquetil. Architecture of a source code exploration tool: A software engineering case study. Technical Report TR-97-07, University of Ottawa, Computer Science, 1997.
[25] T. C. Lethbridge and F. Herrera. Assessing the usefulness of the TKSee software exploration tool. In H. Erdogmus and O. Tanir, editors, *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation*, chapter 11, pages 73–93. Springer-Verlag, Dec. 2001.
[26] T. C. Lethbridge and J. Singer. Strategies for studying maintenance. *2nd Workshop on Empirical Studies of Software Maintenance (WESS'96)*, pages 79–83, Nov. 1996.
[27] A. Marcus, D. Comorski, and A. Sergeyev. Supporting the evolution of a software visualization tool through usability studies. *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 307–316, May 2005.
[28] R. C. Martin. RUP vs. XP. *objectmentor.com*, 2001. http://www.objectmentor.com/resources/articles/RUPvsXP.pdf.
[29] D. L. Moise and K. Wong. Issues in integrating schemas for reverse engineering. In J.-M. Favre, M. Godfrey, and A. Winter, editors, *International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM'03)*, volume 94, pages 81–91. Elsevier, May 2004.
[30] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. *10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 1–10, Sept. 2005.
[31] M. Pomeroy-Huff, J. Mullaney, R. Cannon, and M. Sebern. The personal software process(psp) body of knowledge. Special Report CMU/SEI-2005-SR-003, Software Engineering Institute, Carnegie Mellon University, Aug. 2005. http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05sr003.pdf.
[32] K. Schneider. Prototypes as assets, not toys: Why and how to extract knowledge from prototypes. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 522–531, May 1996.
[33] R. Schwaninger. Rapid prototyping with Tcl/Tk. *Linux Journal*, May 1998. http://www.linuxjournal.com/article/2172.
[34] M. Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, Nov. 1990.
[35] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. *6th IEEE International Workshop on Program Comprehension (IWPC'98)*, pages 173–179, June 1998.
[36] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, pages 209–223, Nov. 1997.
[37] I. Sommerville. Integrated requirements engineering: A tutorial. *IEEE Software*, 22(1):16–23, Jan./Feb. 2005.
[38] M.-A. D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, Dec. 1998.
[39] M. D. Storey, D. Cubranic, and D. M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. *ACM Symposium on Software visualization (SoftVis'05)*, pages 193–202, May 2005.
[40] M. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, Jan. 1999.
[41] M. D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On designing an experiment to evaluate a reverse engineering tool. *3rd IEEE Working Conference on Reverse Engineering (WCRE'96)*, pages 31–40, Nov. 1996.
[42] M. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs. *4th IEEE Working Conference on Reverse Engineering (WCRE'97)*, pages 12–21, Oct. 1997.
[43] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Universität Bern, Dec. 2001.
[44] M. A. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software – Concepts & Tools*, 19(3):109–121, 1998.
[45] M. van den Brand. Guest editor s introduction: Experimental software and toolkits (EST). *Science of Computer Programming*, 69(1–3):1–2, Dec. 2007.
[46] K. Wong. *The Reverse Engineering Notebook*. PhD thesis, Department of Computer Science, University of Victoria, 1999.
[47] J. Wu and M. D. Storey. A multi-perspective software visualization environment. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'00)*, pages 15–24, Oct. 2000.
[48] H. Yang. The supporting environment for a reverse engineering system—the maintainer's assistant. *Conference on Software Maintenance (CMS'91)*, pages 13–22, Oct. 1991.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. The license is available here: http://creativecommons.org/licenses/by-nc-sa/3.0/us/.