Component-Based Tool Development

Holger M. Kienle
University of Victoria, Canada
hkienle@acm.org

Abstract

This paper describes an emerging approach for the construction of software-engineering research tools, which is characterized by the use of preexisting components (e.g., off-the-shelf products, integrated development environments, and domain-specific tools) to realize tool functionalities. This approach can be seen as an instantiation of component-based development for the domain of tool building in academia. Building of tools in this manner is already pursued by many researchers, presumable because this approach promises to be more effective compared to coding a tool from scratch. For example, tools can be quickly prototyped and evolved in response to user input or new research directions. Also, tools can be more usable and adoption friendly.

I have explored issues surrounding tool building with components in my dissertation and have found that there is little work that evaluates and reflects on the impact of this approach. However, since this approach has its unique benefits and drawbacks and since it is increasingly employed by researchers, it becomes more and more important to improve upon the current practice, which can be characterized as ad hoc. This paper is a first step in this direction. It introduces and reflects on component-based tool building, identifying suitable components to construct maintenance tools, providing a catalog of tool examples that leverage the identified components, and discussing issues that need to be addressed to advance the state-of-the-art.

1. Introduction and Motivation

> "*Programs these days are like any other assemblage—films, language, music, art, architecture, writing, academic papers even—a careful collection of preexisting and new components.*"
> – Biddle, Martin, and Noble [6]

Component-based development (CBD) is a widely applied and highly successful approach for developing software systems. Consequently, researchers have started to adopt the idea of CBD for developing their research tools. I refer to this approach to tool building—which reuses existing, pre-packaged functionality—as *component-based tool development* (CBTD). As opposed to traditional tool building, which is characterized by a high degree of custom code and little reuse, CBTD leverages software components as building blocks.

There is a desire among researchers to build upon existing functionality and infrastructure. Participants of a tool building workshop for reverse engineering articulated that they "were tired of writing parsers/analyzers and wanted to avoid writing another one, in particular a C++ parser" [71]. This desire is driven by the realization that in tool building comparably little effort is spent on the research contribution, and that a significant effort is needed for the supporting infrastructure. Researchers would rather work on core activities that advance research than being tied up in lower-level plumbing. This observation is also articulated by Shaw [69]:

> "Most applications devote less than 10% of their code to the overt function of the system; the other 90% goes into system or administrative code: input and output; user interfaces, text editing, basic graphics, and standard dialogs; communication; data validation and audit trails; basic definitions for the domain such as mathematical or statistical libraries; and so on. It would be very desirable to compose the 90% from standard parts."

Consequently, researchers have turned to components (e.g., off-the-shelf products, integrated development environments, and domain-specific tools) to implement tool functionalities. For instance, parsers for reverse engineering have been implemented on top of GNU GCC, Eclipse, and parser generators such as Yacc.

Traditional tool building offers the most flexibility since almost all functionality is implemented from scratch and under the full control of the developer. On the downside, this approach is costly (e.g., in terms of longer development time) and can result in idiosyncratic tools that are difficult to learn and use. In contrast, CBTD promises faster development and tools that are more adoption friendly.

As Jacobson observed, "the use of components to build software systems is not a new idea" [31, p. xiii]. Similarly, the use of components to build research tools is not a new idea. However, the use of components fundamentally changes the development of tools and has unique benefits and drawbacks. This fact is often not realized by tool builders. Important questions for CBTD that have to be addressed by researchers are: What are good candidate components for CBTD given a tool's application domain, its required functionality, its desired quality attributes, and its envisioned users? What impact has a certain component on the overall tool architecture, on the ramp-up time and implementation effort for the tool, and on the further maintenance and evolution of the tool? What characteristics of the components and the architecture minimize risks and maximize effects?—Put differently, what are the design "sweet spots" for tools? And so on. Since CBTD is increasingly employed by researchers, I belive that the time is ripe to reflect upon this development practice with the goal to characterize the state-of-the-art, to identify shortcomings and challenges, and to propose directions to advance the state-of-the-art. Specifically, I advocate to collect case studies of tools that have used CBTD, to work on a catalog of suitable components, to distill lesson's learned (i.e., what works and what does not) that generalize across tools, and to formalize the process of CBTD.

Still, is it desirable or useful to reflect upon the way that we build tools in the academic domain? Should we not exclusively devote our attention to the finished product—that is, the tool itself (e.g., evaluating its features and how they compare to the ones of other tools)—instead of reflecting upon *how* the tool was built? I believe that this kind of reflection is not a distraction, but a necessary activity to advance our research field. This is illustrated by the points of tool adoptability and development time. Imagine a visualization that advances the current state of research with a novel rendering algorithm. From a pure research perspective the algorithm is the research contribution, and the tool is only a vehicle to demonstrate its feasibility. From an applied research perspective, it matters whether users will find the visualizer useful and usable. These factors, however, are largely determined by the tool development strategy; for instance whether the visualization is presented in a stand-alone tool, or integrated into a popular development environment such as Emacs or Eclipse. Similar points can be made for other important quality attributes of tools such as interoperability, customizability, and scalability. Also, as a research contribution the visualization algorithm should not only have novel features, but it should also be useful. This can be shown with user studies. But user studies typically lead to iterative tool development, in which user feedback is taken into account to improve upon the current tool implementation. Thus, a tool development strategy that enables faster tool evolution leads to more productive research and faster generation of research results. Less time spent developing tools translates into more time for creative and evaluative research activities.

In the following discussion of CBTD, I focus on components and common tool functionalities that are expecially relevant for software maintenance research. Software maintenance encompasses a broad spectrum of tools with diverse functionalities. However, many tools exhibit a similar conceptual architecture, which includes (1) a *fact extractor* to obtain information from certain sources (e.g., source code or machine code), and (2) a *visualizer* that renders information derived from these sources in some textual or graphical form. In the following discussion, I restrict myself to static fact extractors of high-level programming languages and visualizers for graphs. Software visualization tools are highly relevant for CBTD because they "are a major investment in infrastructure" [40]; the same is true for fact extractors.

The rest of this paper is organized as follows. Section 2 provides evidence that reflecting on tool building is worthwhile to pursue and important to advance applied research in the domain of software maintenance. Section 3 shows the scope of CBTD by identifying the types of applicable components. Section 4 then identifies concrete examples of components that are promising candidates for building component-based maintenance tools. The catalog also gives examples of tools that leverage these components, testifying to the fact that CBTD is already widely applied. Section 5 distills some experiences and lessons learned of the impact that CBTD has on tool building and quality attributes of tools. The experiences show that some researchers have published recommendations that can be useful for other researchers, but these experiences are unstructured, unfocused, and disconnected. Section 6 briefly outlines future directions to make CBTD more formal and less ad hoc by conducting case studies, synthesizing lessons learned, and defining a process that is tailored for CBTD in academia. Section 7 concludes the paper.

2. Context and Related Work

The software engineering community has a tradition of reflecting upon its own research methods and paradigms. Shaw has analyzed software engineering research methods and found several popular techniques that researchers use to validate convincingly their results [70]. Among those techniques is the implementation of a (prototype) system. The character of this validation is: "Here is a prototype of a system that ... exists in code or other concrete form." A tool prototype serves, for example, to prove the feasibility of a certain concept, or as the basis for user studies to further enhance the tool. Glass et al. provide a literature analysis of the software engineering field based on six leading research journals [22]. They found that 17.1% of the publications employ proof-of-concept implementations as a research method, placing it second only after conceptual analysis with 43.5%. Kitchenham et al. propose evidence-based software engineering to structure the research process with the goal to make it more result-oriented (e.g., more applicable to practitioners) [39]. Tichy argues that computer scientists should experiment more (e.g., to "reduce uncertainty about which theories, methods, and tools are adequate") [75]. These ideas have helped to improve upon the state-of-the-art of experimental software engineering by providing a framework or methodology to follow. Since research results in software maintenance are often embedded in tools and since experiments in this area often involve the evaluation of tools, following the above ideas make it necessary to address tool building issues as part of the research methodology.

The software engineering community understands that tool building is an essential activity of applied research. This seems especially pronounced in the areas of reverse engineering, software visualization, and program comprehension. Researchers also understand that tool building is a major investment. For instance, Nierstrasz et al., who have developed the well-known Moose tool, say that

> "in the end, the research process is not about building tools, but about exploring ideas. In the context of reengineering research, however, one must build tools to explore ideas. Crafting a tool requires engineering expertise and effort, which consumes valuable research resources" [58].

In order to make tool building more economical, researchers have pursued several ideas. For example, the reverse engineering community has advocated a common exchange format to increase tool interoperability and to share results. Researchers have been willing to collaborate on this issues and to expend efforts as testified by the *Workshop on Standard Exchange Format* (WoSEF) [71] and the Dagstuhl seminar 01041 on *Interoperability of Reengineering Tools*. The latter has resulted in the GXL exchange format [32]. Other examples of workshops that have addressed tool building issues are: *International Workshop on Adoption-Centric Software Engineering* (ACSE) held annually from 2001 to 2004; *International Workshop on Incorporating COTS Software into Software Systems* (IWICSS) held in 2004 and 2007; *Workshop on Design Issues for Software Analysis and Maintenance Tools* in 2005; and *International Workshop on Advanced Software Development Tools and Techniques* (WASDeTT) held twice in 2008.

Besides community efforts, individual researchers have reflected on their approach to tool building from various angles. For example, Lanza describes his experiences with the CodeCrawler software visualizer [43], 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 [26]. The special issues on Experimental Software and Toolkits (EST) [76] are devoted to the description of academic research tools. One tool in the first special issue, g^4re, falls within the maintenance domain [41]. The work that is perhaps closest to the spirit of CBTD is package-oriented programming (POP) proposed by Coppit and Sullivan [13]. POP is based on "the use of multiple, architecturally compatible, mass-market packages as large components" [13]. The use of multiple components is motivated by the observation that many tools require functionality that needs to be drawn from several independent domains such as text editing (e.g., provided by Word) and graph editing (e.g., provided by Visio). POP proposes to use components that are architecturally compatible to simplify integration and to minimize architectural mismatch.

3. Components for Tool Building

The most central notion of CBTD is *component*. As a working definition, we follow Czarnecki and Eisenecker, who define (software) components as "building blocks from which different software systems can be composed" [15]. I deliberately introduce a definition that is rather broad because CBTD draws from a diverse spectrum of applicable components.

In the following, I briefly identify several types of components that are applicable for CBTD:

off-the-shelf products: Examples of such products are large commercial applications such as Microsoft Office and Internet Explorer. These products can be purchased and then used as-is. This type also includes open source counterparts with similar functionalities such as OpenOffice and Firefox. Operating systems are also off-the-shelf products. Although often the case, these products are not necessarily GUI-based or intended for end users. An example of off-the-shelf products that are not targeted towards end users are database management systems.

integrated development environments: Examples of integrated development environments (IDEs) are Eclipse, Microsoft Visual Studio, IBM Visual Age, TogetherSoft Together, and Rational Rose. Examples from academia are Reiss' Field and Desert environments [65] as well as CWI's ASF+SDF Meta-Environment [77]. Many of these are also off-the-shelf products, but intended for software development activities.

domain-specific tools: There are many small to mid-size programs that handle specific tasks. For example, Unix has various small programs for effective manipulation of textual data. Other examples are text editors such as Emacs. However, the Emacs editor includes functionalities typically also found in integrated development environments such as syntax highlighting, pretty printing, spell checking, compilation, and debugging. Of specific interest are graph visualizers and graph editors such as the AT&T Graphviz package.

application generators: Application generators generate code for (parts of) a software system based on a (declarative) specification language [42]. Ideas from application generators can be also found in the OMG's Model Driven Architecture (MDA). Typical examples of this component type are scanner and parser generators such as `lex` and `yacc`, respectively. An example of more advanced generator technology is the the ASF+SDF Meta-Environment that can generate compilers and interpreters based on algebraic specifications.

(off-the-shelf) components: In contrast to full products, (off-the-shelf) components provide more limited functionality and cannot be directly executed. Instead, a number of such components are integrated to form a component-based system. Examples of popular component technology are Sun's JavaBeans and Java Enterprise Beans (EJB), and Microsoft's Component Object Model (COM) and Distributed COM (DCOM) [73]. Plug-ins for particular (off-the-shelf) products and IDEs are another example of components [7]. Plug-ins typically provide a low overhead of adding a smaller piece of well-defined functionality to the base application. Examples of products that support plug-ins are Firefox, Photoshop, and the Apache Web server. Eclipse is a popular IDE based on a multi-level plug-in architecture.

frameworks: Object-oriented frameworks provide a set of abstract and concrete classes with explicit variation points, which can be instantiated to create a complete application. There are many well-known examples of frameworks for the GUI domain [19], for instance, Java's Abstract Windows Toolkit (AWT) and Swing, Eclipse's Standard Widget Toolkit (SWT), and the Microsoft Foundation Classes (MFC). Frameworks are often used as a technique to support component models. For example, MFC provides functionality to simplify the implementation of OLE and COM components.

libraries: Libraries provide a coherent collection of functionalities in the form of procedures or classes. There are many libraries for data structures and algorithms such as the C++ Standard Template Library (STL). Another example is the LEDA library, which also has a large variety of graph data structures.

others: A component can be any other reusable element of software with a coherent set of functionalities (e.g., functions and classes, possibly aspects and concerns [15, p. 125f]). However, practical experience has shown that it is often difficult to reuse fine-grained assets such as individual classes because of their tight dependencies on other assets.

4. Catalog of Components

This section presents a catalog of components that have been used for realizing software maintenance tools. The catalog testifies to the fact that researchers indeed use CBTD. It should also give researchers that want to apply CBTD a good starting point on applicable components and reported experiences.

We first discuss components for graph visualizers (cf. Section 4.1), followed by components to realize fact extractors (cf. Section 4.2). Because of space constraints, only a few of the tools are mentioned; the full catalog is available in my dissertation [35, sec. 5.2].

4.1. Components for Graph Visualization

Table 1 provides a summary of the components that are suitable to build graph-based visualization tools.

Among the OTS products, Microsoft Office/Visio is probably the most popular component for CBTD. Both PowerPoint [82] and Visio [83] can be used to realize interactive graph editors for

Types	Host component	Tool-building examples
OTS products	Office/Visio	REOffice [82]; REVisio [83]; Huang et al. [30]; Nimeta [67]; Nova [14]
	FrameMaker	SLEUTH [61]; Desert [64]
	Web browsers	REPortal [52]; Software Bookshelf [20]; TypeExplorer [78]
IDEs	Eclipse	MARPLE [3]
	Rational Rose	Rose/Architect [18]; Berenbach [5]; Lakshminarayana et al. [72]
	Together	JaVis [54]; GoVisual [29]
tools	AT&T Graphviz	Reflexion model viewer [57]; ReWeb [66]; CANTO [1]
	SVG	SVG graph editor [46]; SPO [48]
libs	OpenGL	Extravis [33]; CodeCity [80]

Table 1. Examples of components to build graph-based visualizers

Types	Host component	Tool-building examples
IDEs	Eclipse	Creole [47]; Xia [81]
	IBM VA C++	Rigi C++ extractor [53]; ISME [51]
	GoLive	REGoLive [28]
tools	GNU GCC	CPPX [16]; XOgastan [2]; g^{4}re [41]
	EDG	FORTRESS [25]; Xrefactory [79]
	SNiFF+	`sniff2cdif` [74]; CrocoCosmos [45]; Dali [10]
	SN	Moise and Wong [56]; Pinzger et al. [60]

Table 2. Examples of components to build static fact extractors

Rigi-like software graphs. The graph editors are typically coded in VisualBasic using the application's native drawing capabilites (i.e., shapes and connectors). Graphs can have several hundreds of nodes represented with simple shapes such as squares or circles. Nova is a fault tree analyzer that leverages Visio to implement a graph-based editor [14]. In this case there are relatively fewer nodes but they have more complex shapes. Researchers have also used Visio to render static software graphs [30] and architectural UML diagrams [67]. Visio can be also used as a converter to export UML diagrams to HTML [67].

Adobe FrameMaker is another OTS product that has been used to render graphs in software documentation (SLEUTH [61]) and code views (Desert [65] [64]). Researchers can built upon a fully-featured word processor instead of implementing this functionality from scratch. However, since FrameMaker has no support for graphical objects, the graphs can only be rendered as static images.

There are myriad applications that use Web browsers to visualize information. Web browsers are attractive targets because in contrast to other OTS products they represent a whole class of products that can be easily exchanged for one another. Web browsers can be easily used to combine text with static pictures [52] [78]. To realize interactive graphs, plug-ins can be used. For example, the Software Bookshelf is a reverse engineering and documentation environment that renders architectural diagrams with a Java applet [20].

Eclipse has recently gained popularity among researchers as a platform for tool development. Researchers can extend Eclipse by writing plug-ins in Java. The published papers in the *eclipse Technology Exchange* (eTX) show the broad range of tools that leverage Eclipse. Eclipse supports graphs via the Graphical Editing Framework (GEF). The MARPLE tool uses GEF to provide different visualization of software structures such as Lanza's class blueprint [3].

Researchers can leverage Rational Rose to realize UML editors and visualizers. This is especially valuable if the research tool has to provide full UML editing capabilites. Similar to Microsoft Office, Rose is programmable with Rose Script or a COM-enabled API. The Rose Script Editor provides access to the scripting environment so that users can interactively edit and execute scripts. Egyed and Kruchten have developed an add-on to Rose called Rose/Architect, which analyses the current Rose model (i.e., UML class and object diagrams) and then proposes an abstraction of this model in the form of another UML model [18]. Berenbach has used Rose to realize a tool to associate use cases with requirements information by defining new symbols and relationships [5]. Lakshminarayana et al. use scripting to extract class information from UML diagrams in Rose [72]. This information is then used to compute (semantic) metrics to assess the software design.

TogetherSoft Together is a UML editor and an IDE for Java and C++. The Together Open API can be programmed in Java and consists of three different kinds of interfaces. A low-level interface allows to manipulate Java sources at the byte code level. A high-level interface allows read-only access of Together's state (e.g., model information). The mid-level interfaces allows modification of the model and customization of Together. The JaVis tools uses Together to visualize animated trace data for concurrent Java programs as UML sequence diagrams [54]. The GoVisual plug-in enhances Together with sophisticated new layout algorithms for UML class diagrams [29].

AT&T Graphviz is a package for graph drawing and editing. Its most popular tools are `dot` for static graphs, and `dotty` for interactive ones. The `dotty` graph editor can be customized with a dedicated scripting language. Graphviz is extremely popular among researchers to visualize static software graphs. There are also examples of interactive graphs [66] [57] [1].

Scalable Vector Graphics (SVG) is a Web standard introduced by the W3C for (interactive) vector graphics. Interactivity is supported via JavaScript or declarative animation. SVG makes it possible to realize sophisticated interactive graph visualizations in web browsers. There are several implementations of the SVG standard (e.g., the Adobe SVG plug-in). The SVG graph editor shows that it is possible to realize Rigi-like graphs with SVG [46]. It is implemented with 10K lines of JavaScript. SVG has been also used for chart-like visualizations of the evolution of repository data [48].

An example of a library component is the Open Graphics Library (OpenGL), which defines an API for advanced (3D) visualizations. There are different implementations of OpenGL depending on the platform and programming language. For example, CodeCity use the Jun library for Smalltalk [80], whereas Extravis uses Delphi's native OpenGL support [33].

4.2. Components for Static Fact Extraction

Table 2 provides a summary of the components that are suitable to build static fact extractors. Note that there are components that can be used for both visualization and extraction (e.g., Eclipse); this is typically the case for IDEs. Researchers have a keen interest in reusing existing extractors because many languages are difficult to parse. Also, fact extraction is for most researchers a tedious prerequisite that needs to be done before the actual research can take place.

Part of the Eclipse Java IDE is an incremental compiler that can be used for fact extraction. The compiler provides a Java API that allows access to its AST; there is also the Java Development Tools (JDT) model, which provides a simplified, high-level view of the AST. The Creole tool, which integrates SHriMP with Eclipse, makes use of the Java API [47]. Eclipse has extensions in the form of plug-ins; some of these can be used as a source for fact extraction. The Xia tool uses Eclipse's CVS plug-in to extract and visualize version information of a software system [81].

IBM VisualAge C++ is an example of a proprietary IDE that offers a C++ API to interface with it; however since the API is not publicly available, there are few documented uses. VisualAge C++ can be executed headless so that fact extractors can use it without starting up the GUI. The Rigi C++ extractor uses VisualAge's API to generate facts about the compiled C++ source in RSF as a side-effect of the compilation process [53]. Similarly, the C++ parser of the ISME tool uses VisualAge to generate information in the C++ Markup Language (CppML) [51].

Adobe GoLive is an integrated environment to produce Web sites. Since GoLive provides programmatic access to assets such as HTML and XML files via a JavaScript API, information for Web site reverse engineering can be extracted [28].

Compiler front-ends are another popular approach to realize fact extractors. The CPPX extractor modifies GCC's C++ front end to generate output in TA, VCG, and GXL [16]. To obtain information about the processed source, CPPX accesses GCC's internal representation (which is encoded as C structures). Instead of directly modifying `gcc`, Power and Malloy have extended the `bison` parser generator to emit a parse tree of the processed source in XML format [62]. Another approach, which does not require source code changes, is to apply GCC compiler switches that produce textual `tu` files of GCC's internal representation. XOgastan uses this approach to obtain a parse tree for C++ source code [2]. A Perl script translates the `tu` files into GXL. The g^{4}re tool uses the same approach [41]. Since `tu` files can get large, g^{4}re provides tools to prune them.

The C++ front end of the Edison Design Group is capable of a full syntax and semantic analysis of the source code. It is a mature product used by various compiler vendors; a free license can be obtained for non-commercial use. EDG can handle various C++ dialects, controlled by command-line options. EDG is written in C and has an API to hook up back ends, but can also generate cross-reference information. EDG is leveraged by Xrefactory, a source code understanding and refactoring tool [79], and the FORTRESS reverse engineering tool [25]. Given the features of EDG, it is surprising that relatively few academic tools take advantage of it.

SNiFF+ is a source code browsing and comprehension tool that supports various languages, among them C++, Java, and Tcl. To obtain source code information, SNiFF+ uses a fuzzy parsing technology. SNiFF+'s Symbol Table API provides programmatic access to the extracted information. CrocoCosmos [45] uses the SNiFF+ API to extract information from C++ and Java, which is then stored in a relational database. The Dali architecture reconstruction tool uses SNiFF+ to extract facts from C and C++ [10]. Tichelaar and Demeyer's `sniff2cdif` tool uses SNiFF+'s C API to produce UML diagrams of SNiFF+ projects [74].

Source Navigator (SN) is a source code analysis tool similar to SNiFF+. The parser populates a database that can be queried with a procedural API. Moise and Wong have written a fact extractor for C/C++ based on SN [56]. They retrieve information from SN's repository, taking advantage of its fuzzy parsing, but also use "local scans into the source files, especially in parts where Source Navigator is weak, such as local variables or templates" [56]. Pinzger et al. use SN to extract information from Visual Basic for program comprehension of COM+ components [60].

4.3. Observations Drawn from the Catalog

The presented catalog shows that CBTD is feasible and already successfully applied by researchers to build visualizers and fact extractors.

The collected catalog shows the diversity of components to implement visualization functionality. Microsoft Office is one of the more popular components. This may be explained with the fact that besides support for visualizations (graphs and charts), Office has support for data persistence and querying, spreadsheet computations, and text formatting. Furthermore, functionality of all Office components can be customized with a convenient scripting environment, and they can be made interoperable via OLE and COM. Whereas Office is most often used for editable visualizations, AT&T Graphviz is very popular to produce static graphs of software structures. Generally, Web-based tool interfaces are also popular. Web browsers offer a ubiquitous computing platform that is highly familiar to users. Technologies such as HTML, JavaScript, SVG, and applets allow the rapid development of sophisticated Web applications that are portable across browsers.

For a visualizer component, the capabilities of its user interface play an important role. In contrast, this is only a minor concern for extractor components, which are non-interactive. Instead, the extractor's parsing strategy is important, because it determines properties such as the extractor's accuracy, speed, and robustness. Thus, selecting a component requires an understanding of its employed extraction technique with its associated merits and drawbacks. GCC is a popular foundation for accurate C/C++ fact extractors that store the extracted facts in an exchange format. For Java, Eclipse is very popular for both Java fact extraction and analysis/visualization of the extracted facts. For lightweight fact extraction, both SNiFF and SN are popular. Since different extractors have different properties, researchers have combined the facts produced by multiple extractors to obtain complementary information (e.g., [60]).

The discussed tool building examples vary widely in the effort that has been expended when leveraging a component. Some component-based tools represent a significant programming effort by programmatically customizing a component. Other tools do not customize a component at all. In such cases the component is often used to render a document generated by the tool such as an HTML page rendered by a Web browser. For example, some researchers have modified GCC's source code, while others only use command-line switches to change its behavior.

Even though there are many examples of CBTD, few researchers actually bother to report their tool building experiences. It is often difficult or impossible to understand how researchers have built a tool. The fact that a component has been leveraged is often touched upon only in a few sentences. One reason for this lack of detail might be the perception that such experiences are not part of the publishable research results. I believe, however, that published experiences are valuable, greatly benefit other researchers, and constitute a first step towards a better and more formal understanding of tool building issues.

Some researchers briefly describe their experience with a component, but almost all of the published tool building examples lack a conscious exploration of their approach in the sense that there is no reflection or discussion of the impact that CBTD has on the development effort and the tool users. The authors of the Dali architecture reconstruction tool note that

"the use of commercial tools has provided a great deal of

leverage during the development of Dali: these tools are typically robust, well documented, and well tested" [10].

The authors of Xrefactory say that

"even if Xrefactory is using a professional C++ front-end provided by EDG it is still very difficult to take care of all special cases defined in the ANSI standard" [79].

O'Brien makes a recommendation based on his experiences with two fact extractors:

"Both Imagix-4D and SNiFF+ tools were applied to extract views from the source code. After analyzing the output from both tools, we concluded that the Imagix-4D was more useful for extraction purposes" [59].

Whereas such reported experiences provide valuable information and leads for other researchers to follow up on, they lack sufficient detail. Consequently, tool building continues to be approached in an ad hoc manner, lacking general solutions.

5. Reflections on CBTD

In this section I discuss a few selected examples of experiences and lessons learned in CBTD. These reflections are not meant to be exhaustive, but serve as an illustration of the kind of research that is necessary in order to understand CBTD and to advance the state-of-the-art better.

5.1. CBTD is CBD

CBD and CBTD have a lot in common. In a sense, CBTD is only the application of CBD for building research tools. Thus, CBTD can learn from the general experiences and approaches provided by CBD. However, CBTD must also come up with domain-specific experiences and approaches that are unique to CBTD, or that refine the more general experiences of CBD. Examples of issues that CBTD has to address are:

selection: Finding suitable components for implementing a tool is important because the characteristics of the component will determine the whole tool building effort. If the component is carefully selected, its existing baseline functionality can cover a significant part of the tool functionality. As a result, significantly less code needs to be written and subsequently maintained. Reiss reports that for Desert's editor, "FrameMaker provides many of the baseline features we needed" [65]. Similarly, the authors of SLEUTH say that "FrameMaker provides an effective starting point for our prototype. Many of the basic features necessary for document creation and editing are provided, allowing effort to be concentrated on more specialized features" [61]. The Rigi C++ extractor was developed rapidly with VisualAge C++. As a result, it was possible to "develop the parser much faster than if a parser had been written from scratch" [53]. Selecting of components would be easier if a catalog of candidate components were available.

customization: Customizations of components is limited by the functionality that the API and/or scripting language provides. As a consequence, customizations of components without the use of source code modifications always run the risk that certain desirable tool functionalities cannot be realized at all, or with less fidelity. Unfortunately, missing functionality is hard to identify upfront. Often, limitations materialize unexpectedly after a commitment for a certain component has been finalized and substantial development effort has been already invested. Nova's developers had to cope with undocumented restrictions of Microsoft Office components: "Due to [the] lack of package documentation, we discovered certain limitations of the packages only after working with them extensively. In some cases, the limitations were quite serious" [12]. They also report that "the maximum number of shapes Visio 4.0 could store on a single drawing page was approximately 5400. Interestingly, version 5.0 retained this limitation" [12]. Reiss reports the following limitation in the FrameMaker API: "While FrameMaker notified the API that a command was over, it did not provide any information about what the command did or what was changed" [64]. Gray and Welland note that Rational Rose cannot be customized to change the line width of inheritance links in UML diagrams [24]. Generally, problems with component customizations are much more pronounced in the implementation of interactive, graphical tools than batch-style fact extractors.

learning curve: Using components can drastically reduce the development effort when building new tools. However, to customize a component effectively, a significant learning effort is often required. For example, tool developers that wish to customize VisualAge C++ should be aware that "regardless of how good the programming interfaces are, [it] is a big system, and there is a steep learning curve" [34]. Similarly, Eclipse developers report that "the steep learning curve for plug-in developers in Eclipse is hard to master" [8]. Generally, tool developers need to be careful not to be overly optimistic about the required effort to become proficient with a component. Customization of components is easier if there are previous experiences that address specific tool building issues, or if there are customization examples in the component's source code itself (e.g., such as in Eclipse). For example, Reiss has described his approach to customizing FrameMaker to implement a software development tool [64]. The Adoption-Centric Software Engineering (ACSE) project at the University of Victoria has published experiences of customizing a number of OTS products, among them Visio [83] [11], Lotus Notes [50] [49], GoLive [28] [27], and Websphere Application Developer [38].

interoperability: The interoperability between components can be classified into data, control, and presentation integration. Generally, presentation integration is most desirable from the user's point-of-view because it can achieve seamless access of all of the tool's functionality with a single coherent GUI. However, this form of integration can be difficult or impractical to accomplish. Instead, data integration can be used, which often achieves an integration between components that is satisfactory to the tool's users. For example, (non-interactive) graphs can be displayed by calling an external viewer component. Control integration is needed if two components are collaborating more tightly and have to pass messages back and forth.

Components can have a wide range of interoperability mechanisms and make implicit assumptions about the way they interact with other components. Thus, it is unlikely that two independently developed components that do not adhere to the same component model will interoperate seamlessly out-of-the-box. This phenomenon is know as *architectural mismatch* [21]. Interoperability is less of an issue if the tool is based on a single component only. Also, architectural mismatch is less of a problem for data integration compared to control and presentation integration.

maintenance: It is necessary to take maintenance issues into account for tools that have a longer life-time than simple prototypes, which are quickly coded and then abandoned once research results have been reported. For instance, if a new version of a component

becomes available, researchers may be forced to upgrade to this component for their tool because the vendor may not support older versions of the component anymore. Balzer et al. recommend to implement an abstraction layer that interfaces with the component in order to contain changes due to component upgrades [4]. They also note that after a component upgrade the API calls may still be the same, but exhibit a different behavior; for example, "in PowerPoint we found that a newer version changed the number ordering on how connectors were attached to shapes" [4]. Interestingly, upgrading a component may enable new baseline functionality that users can immediately leverage in the tool. Researches that want to make several components interoperable should consider that the most significant variable that influences the cost of maintenance is the number of OTS components that need to be synchronized [63].

scalability: The constructed tool should be able to fulfill the tool users' expectations in terms of speed and scalability. If the tool's visualization functionality is realized with a component, it is important to explore scalability issues early on. Generally, components are able to handle the rendering of dozens of graphical objects. This is sufficient for small to medium software graphs. However, the rendering of larger graphs can cause problems in terms of screen updating and rendering speed. For example, the developers of the Nova tool state that their tool's "response time is largely dependent on package performance. ... We spent significant effort understanding Visio's drawing speed and exploring ways to use Visio that would give us better drawing performance" [12]. Also, different components and different versions of the same components can differ in their performance characteristics. For example, in Visio 4.1 the drawing speed of objects increases quadratically with the number of shapes already present on the page; in contrast, Visio 5.0's behavior is linear [12].

adoptability: Tool adoption depends on many factors. Besides technical considerations such as tool features, there are also many other issues that affect a tool user's adoption decision. Often tool developers pay scant attention to documentation even though it decreases complexity and helps trialability for the tool user. To simplify the generation of documentation, the capabilites of the component may be leveraged. For example, Microsoft Office's Assistant can be customized in Visual Basic to provide tool-specific help. The market share of components can also improve a tool's adoptability. The popularity of PowerPoint was an important factor for the developers of VDE: "Visio is a commercial product with many similarities to PowerPoint, provides extensive visibility into its state-change events. It might provide a better technical fit to our needs, but lacks PowerPoint's enormous user base" [23]. Familiarity of the target users with a certain product improves usability and helps adoptability. Reiss explains the decision to use FrameMaker as a component with the fact that "we wanted an editor that programmers would actually use. This meant that the base editor must be familiar to them, preferably one they were using already" [65].

5.2. Architecture Does Matter

When a tool is built following CBTD, the most important considerations are to select suitable components, to make these components interoperate, and to customize individual components. All of these considerations have an impact on the tool's architecture. On the one hand, deciding on a certain tool architecture may preclude selecting a component that may be suitable otherwise. On the other hand, characteristics of a component may constrain certain architectural choices.

To illustrate the impact of architecture on CBTD, I briefly outline two different kinds of (reference) architectures and their resulting trade-offs. The first architecture is based on a loose coupling of several components, the other one is an example of customizing a single component.

To realize a maintenance tool it is typically necessary to implement functionality for fact extraction, visualization, data storage (repository), and analyses. A typical architecture to implement such a tool decouples the fact extraction from the visualization with the help of a repository. The fact extractor writes to a repository (e.g., a file that adheres to an exchange format such as GXL), which is then read by the visualizer. (A similar approach is used by compilers to decouple the front-end from the back-end.) This architecture has several implications for component selection and integration. First, the extractor and visualizer component communicate via data integration, resulting in a weak coupling between the components. As a result, the extractor has to operate in batch-style because the whole repository needs to be populated before the visualizer can work on the data. Furthermore, synchronization between the component is typically driven by the user (i.e., the visualizer will not automatically update its view once new data becomes available in the repository). Second, if the repository is based on an exchange format, it may not be practical to use it for the storage of fine-grained information models (e.g., ASTs or PDGs) because of scalability issues. Third, the loose coupling makes it possible to transparently replace an extractor for another one; the same holds for visualizers. Forth, analyses that operate on the data can be placed in the extractor component, in the visualizer component, or as independent components. Imagine an analysis that computes the cyclomatic complexity of procedures. This computation can be performed for all procedures by the extractor (and then stored in the repository), or computed (on demand) by the visualizer (assuming that the data model is adequate). The visualizer or extractor could also communicate with a separate component that provides the computation as a service.

As an alternative, the same functionality can be realized with a quite different architecture that leverages a single component offering both extractor and visualizer functionality. A typical example of such a component is Eclipse, which offers APIs to its Java parser and to its GUI. In this case, the extractor and visualizer are part of the same component and as a result control integration can be easily realized. As a consequence of control integration, it may be possible to easily realize more fine-grained interactions that enable, for instance, an incremental extractor that parses source code on demand. Also, in this scenario an explicit repository component may be expendable because the extractor and visualizer keep their state as in-memory data structures. If the parser/compiler component scales, it can be expected that the extractor will scale as well. Because of the tight coupling in may be difficult or impossible to replace the extractor with a different one because certain interaction patterns may be not longer feasible. Analyses can be implemented very flexibly because they can operate on the data provided by the extractor, visualizer, or both.

There are several examples of work that have explored architectural issues for tool development. For example, Mendonça and Kramer identify and discuss several (reference) architectures for reverse engineering and program understanding tools [55]. The architectures identify tool functionalities (e.g., Parser, Analysis System, and Viewer), and artifacts that the tool produces and/or uses (e.g., Relational Source Model, Annotated AST, High-Level Models, and Program Abstraction Library). The tool functionalities of the architectures could be implemented in CBTD by selecting and customizing suitable components. Lethbridge and Anquetil propose an architecture for a program comprehension tool and discuss

how the architecture supports functional requirements and quality attributes that such a tool should have [44]. Again, CBTD could apply their findings to select suitable components and to map components to tool functionalities. Egyed and Balzer describe generic architectures for data and control integration of OTS components and provide two integration case studies based on Rose and Matlab [17]. Their case studies show that such an integration in feasible, but also that it is not trivial to accomplish.

6. Future Directions

As with any approach to software development, the assumptions and implications of CBTD should be continuously questioned and explored in order to advance the state-of-the-art. Possible future directions are as follows:

case studies: There is no such thing as too many case studies. Each well-described tool building experience can deepen our understanding of CBTD. However, since a case study draws valuable resources, it should be well planned and exercised. An interesting approach might be to conduct a "family of case studies" that strives to implement the same tool functionality on top of different components. For instance, Parnas' tabular representations could be implemented utilizing Excel, Word, SVG, and DHTML.

synthesis of lessons learned: It would be highly desirable to identify lessons learned that generalize over individual tool building experiences. Such lessons could address CBTD issues such as (reference) architectures, tool-building design patterns, and benefits and drawbacks of certain technologies and components. Currently there are few examples of researchers that have published such lessons learned (e.g., [4] [36]).

tool development process: It seems that few academic tool building efforts make use of an explicit process. This is hard to defend, because any software should be constructed based on a process. As a first step, I have proposed a dedicated process framework for tool building in academia [37]. Challenges in defining a suitable process must address the need to not stifle unnecessarily the creative elements in research, and the diversity of academic research projects (e.g., tool requirements, degree of technical uncertainty, complexity and size of the problem, and number and expertise of the development team).

In order to have an impact, the above issues have to be tackled by an emerging community of researchers that see the need to advance tool building in academia. Workshops such as IWICSS and WASDeTT can provide a platform for discussion. A yardstick for success is the generation of results that make tool building in academia more predictable and effective.

Besides further exploring CBTD, the research community should be open to new approaches to tool development. For example, CBTD may be replaced (or married) with model-based approaches to tool building [68] [9].

7. Conclusions

In this paper I have explored an emerging approach to tool building that I call component-based tool development, which is based on the idea of leveraging components as building blocks to realize tools via component integration and customization. I have identified suitable components for the construction of software maintenance tools, and have given examples of tools that have been built with these components.

The paper has described the state-of-the-art of component-based tool building and pointed out open issues that should be addressed to make tool building less ad hoc. This paper shows that reflection on tool building is happening, but not to the extent that it would be desirable given that it is necessary to advance applied research as a whole.

Drawing on Armor's levels of ignorance, there are three groups of researchers with respect to tool building: (1) researchers that know that tool building should be discussed and published, (2) researchers that are aware of this, but who choose not to do so, and (3) researches that are unaware that they should reflect upon the way that they are building tools. This paper hopes to address each of the three groups. Researchers in the first group will find a proposal of a tool building approach that they can critique and apply. Researchers in the second group may find compelling arguments to join the first group. Researches in the third group now will belong no longer to this group since they have read the paper.

Acknowledgments

I would like to thank Hausi Müller and Crina Vasiliu for comments and proofreading on a draft of this paper.

References

[1] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo. Program understanding and maintenance with the CANTO environment. *13th IEEE International Conference on Software Maintenance (ICSM'97)*, pages 72–81, Oct. 1997.

[2] G. Antoniol, M. D. Penta, G. Masone, and U. Villano. XOgastan: XML-oriented gcc AST analysis and transformations. *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 173–182, Sept. 2003.

[3] F. Arcelli, C. Tosi, M. Zanoni, and S. Maggioni. The MARPLE project – a tool for design pattern detection and software architecture reconstruction. *1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT-1)*, June 2008.

[4] R. Balzer, A. Egyed, N. Goldman, T. Hollebeek, M. Tallis, and D. Wile. Adapting COTS applications: An experience report. *2nd IEEE International Workshop on Incorporating COTS-Software into Software Systems: Tools and Techniques (IWICSS'07)*, May 2007.

[5] B. Berenbach. Towards a unified model for requirements engineering. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 26–29, May 2004.

[6] R. Biddle, A. Martin, and J. Noble. No name: Just notes on software reuse. *ACM SIGPLAN Notices*, 38(2):76–96, Feb. 2004.

[7] D. Birsan. On plug-ins and extensible architectures. *ACM Queue*, 3(2):40–46, Mar. 2005.

[8] P. Bouillon, M. Burger, and A. Zeller. Automated debugging in Eclipse (at the touch of not even a button). *2003 OOPSLA workshop on eclipse technology eXchange*, pages 1–5, Oct. 2003.

[9] R. I. Bull, M.-A. Storey, J.-M. Favre, and M. Litoiu. An architecture to support model driven software visualization. *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 100–106, June 2006.

[10] S. J. Carrière and R. Kazman. The perils of reconstructing architectures. *3rd International Software Architecture Workshop (ISAW-3)*, Nov. 1998.

[11] Y. Chen. Building a software metrics visualization tool using the Visio COTS product. Master's thesis, Department of Computer Science, University of Victoria, 2006.

[12] M. A. Copenhafer and K. J. Sullivan. Exploration harnesses: Tool-supported interactive discovery of commercial component properties. *14th IEEE Conference on Automated Software Engineering (ASE'99)*, pages 7–14, Oct. 1999.

[13] D. Coppit and K. J. Sullivan. Multiple mass-market applications as components. *22nd ACM/IEEE International Conference on Software Engineering (ICSE'00)*, pages 273–282, June 2000.

[14] D. Coppit and K. J. Sullivan. Sound methods and effective tools for engineering modeling and analysis. *25th ACM/IEEE International Conference on Software Engineering (ICSE'03)*, pages 198–207, May 2003.

[15] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[16] T. R. Dean, A. J. Malton, and R. Holt. Union schemas as a basis for a C++ extractor. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 59–67, Oct. 2001.

[17] A. Egyed and R. Balzer. Unfriendly COTS integration—instrumentation and interfaces for improved plugability. *16th International Conference of Automated Software Engineering (ASE'01)*, pages 223–231, Nov. 2001.

[18] A. Egyed and P. B. Kruchten. Rose/Architect: A tool to visualize architecture. *32rd IEEE Hawaii International Conference on System Sciences (HICSS'99)*, Jan. 1999.

[19] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, Oct. 1997.

[20] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopolous, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Apr. 1997.

[21] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *17th ACM/IEEE International Conference on Software Engineering (ICSE'95)*, pages 179–185, Apr. 1995.

[22] R. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information and Software Technology*, 44(8):491–506, June 2002.

[23] N. M. Goldman and R. M. Balzer. The ISI visual design editor generator. *IEEE Symposium on Visual Languages (VL'99)*, pages 20–27, Sept. 1999.

[24] P. Gray and R. Welland. Increasing the flexibility of modelling tools via constraint-based specification. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, Nov. 1999.

[25] M. Grechanik, D. E. Perry, and D. Batory. Reengineering large-scale polylingual systems. *International Workshop on Incorporating COTS-Software into Software Systems: Tools and Techniques (IWICSS'04)*, pages 22–32, Feb. 2004. `http://www.tuisr.utulsa.edu/iwicss/`.

[26] 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`.

[27] G. Gui. Extending a Web authoring tool for Web site reverse engineering. Master's thesis, Department of Computer Science, University of Victoria, 2005.

[28] G. Gui, H. M. Kienle, and H. A. Müller. REGoLive: Building a web site comprehension tool by extending GoLive. *7th IEEE International Symposium on Web Site Evolution (WSE'05)*, pages 46–53, Sept. 2005.

[29] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing UML class diagrams. *ACM Symposium on Software Visualization (SoftVis'03)*, pages 179–188, June 2003.

[30] J. Hartmann, S. Huang, and S. Tilley. Documenting software systems with views II: An integrated approach based on XML. *19th ACM International Conference on Computer Documentation (SIGDOC'01)*, pages 237–246, Oct. 2001.

[31] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[32] R. C. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171, Nov. 2000.

[33] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, Sept./Oct. 2006.

[34] M. Karasick. The architecture of Montana: An open and extensible programming environment with an incremental C++ compiler. *6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-6)*, pages 131–142, Nov. 1998.

[35] 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`.

[36] H. M. Kienle. Building reverse engineering tools with software components: Ten lessons learned. *14th IEEE Working Conference on Reverse Engineering (WCRE 2007)*, pages 289–292, Oct. 2007.

[37] H. M. Kienle. Building reverse engineering tools with software components: Towards a dedicated development process for academia. *24st IEEE International Conference on Software Maintenance (ICSM'08)*, Oct. 2008. To appear.

[38] H. M. Kienle and H. A. Müller. A WSAD-based fact extractor for J2EE web projects. *9th IEEE International Symposium on Web Site Evolution (WSE'07)*, Oct. 2007.

[39] B. A. Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. *26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pages 273–281, May 2004.

[40] R. Koschke. Software visualization in software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, Mar. 2003.

[41] 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.

[42] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[43] 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.

[44] 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.

[45] C. Lewerentz and F. Simon. Metrics-based 3D visualization of large object-oriented programs. *1st International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 70–77, June 2002.

[46] T. Y. Lin, F. Zou, H. M. Kienle, and H. A. Müller. A customizable SVG graph visualization engine. *SVG Open 2007*, Sept. 2007. `http://www.svgopen.org/2007/papers/CustomizableSVGGraphVisualizationEngine/`.

[47] R. Lintern, J. Michaud, M. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. *ACM Symposium on Software Visualization (SoftVis'03)*, pages 47–56, June 2003.

[48] M. Lungu and M. Lanza. The small project observatory. *1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT-1)*, June 2008.

[49] J. Ma. Building reverse engineering tools using Lotus Notes. Master's thesis, University of Victoria, Department of Computer Science, Oct. 2004.

[50] J. Ma, H. M. Kienle, P. Kaminski, A. Weber, and M. Litoiu. Customizing Lotus Notes to build software engineering tools. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 276–287, Oct. 2003.

[51] E. Mamas and K. Kontogiannis. Towards portable source code representations using XML. *7th IEEE Working Conference on Reverse Engineering (WCRE'00)*, pages 172–182, Nov. 2000.

[52] S. Manchoridis, T. S. Sounder, Y. Chen, D. R. Gansner, and J. L. Korn. REportal: A web-based portal site for reverse engineering. *8th IEEE Working Conference on Reverse Engineering (WCRE'01)*, pages 221–230, Oct. 2001.

[53] J. Martin. Leveraging IBM VisualAge for C++ for reverse engineering tasks. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 83–95, Nov. 1999.

[54] K. Mehner. Javis: A UML-based visualization and debugging environment for concurrent Java programs. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 163–175. Springer-Verlag, 2002.

[55] N. C. Mendonça and J. Kramer. Requirements for an effective architecture recovery framework. *2nd International Software Architecture Workshop (ISAW-2)*, pages 101–105, Oct. 1996.

[56] D. L. Moise and K. Wong. An industrial experience in reverse engineering. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 275–284, Nov. 2003.

[57] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 30(8):29–36, Aug. 1997.

[58] 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.

[59] L. O'Brien. Architecture reconstruction to support a product line effort: Case study. Technical Note CMU/SEI-2001-TN-015, Software Engineering Institute, Carnegie Mellon University, July 2001. `http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn015.pdf`.

[60] M. Pinzger, J. Oberleitner, and H. Gall. Analyzing and understanding architectural characteristics of COM+ components. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 54–63, May 2003.

[61] A. L. Powell, J. C. French, and J. C. Knight. A systematic approach to creating and maintaining software documentation. *11th ACM Symposium on Applied Computing (SAC'96)*, pages 201–208, 1996.

[62] J. F. Power and B. A. Malloy. Program annotation in xml: a parse-tree based approach. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 190–198, Nov. 2002.

[63] D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark. COTS-based systems—twelve lessons learned about maintenance. In R. Kazman and D. Ports, editors, *3rd International Conference on COTS-Based Software Systems (ICCBSS'04)*, volume 2959 of *Lecture Notes in Computer Science*, pages 137–145. Springer-Verlag, 2004.

[64] S. P. Reiss. Program editing in a software development environment (draft). `http://www.cs.brown.edu/~spr/research/desert/fredpaper.pdf`, 1995.

[65] S. P. Reiss. The Desert environment. *ACM Transactions on Software Engineering and Methodology*, 8(4):297–342, Oct. 1999.

[66] F. Ricca and P. Tonella. Understanding and restructuring Web sites with ReWeb. *IEEE MultiMedia*, 8(2):40–51, Apr.–June 2001.

[67] C. Riva and Y. Yang. Generation of architectural documentation using XML. *9th IEEE Working Conference on Reverse Engineering (WCRE'02)*, pages 161–169, Oct. 2002.

[68] S. Rugaber and K. Stirewalt. Model-driven reverse engineering. *IEEE Software*, 21(4):45–53, July/Aug. 2004.

[69] M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *Symposium on Software Reusability (SSR'95)*, pages 3–6, Apr. 1995.

[70] M. Shaw. The coming-of-age of software architecture research. *23rd ACM/IEEE International Conference on Software Engineering (ICSE'01)*, pages 657–664a, May 2001.

[71] S. E. Sim and R. Koschke. Wosef: Workshop on standard exchange format. *ACM SIGSOFT Software Engineering Notes*, 26(1):44–49, Jan. 2001.

[72] C. Stein, L. Etzkorn, and D. Utley. Computing software metrics from design documents. *42nd ACM Southeast Regional Conference (ACM-SE 42)*, pages 146–151, Apr. 2004.

[73] C. Szyperski. Emerging component software technologies—a strategic comparison. *Software – Concepts & Tools*, 19(1):2–10, June 1998.

[74] S. Tichelaar and S. Demeyer. SNiFF+ talks to Rational Rose: Interoperability using a common exchange model. *SNiFF+ User's Conference*, Jan. 1999. `http://www.iam.unibe.ch/~demeyer/Tich99m/`.

[75] W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, May 1998.

[76] M. van den Brand. Guest editor s introduction: Experimental software and toolkits (EST). *Science of Computer Programming*, 69(1–3):1–2, Dec. 2007.

[77] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[78] A. van Deursen and T. Kuipers. Building documentation generators. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 40–49, Aug. 1999.

[79] M. Vittek, P. Borovansky, and P. Moreau. A collection of C, C++, and Java code understanding and refactoring plugins. *ICSM 2005 Industrial & Tool Proceedings*, pages 61–64, Sept. 2005.

[80] R. Wettel and M. Lanza. Visualizing software systems as cities. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'07)*, pages 92–99, June 2007.

[81] X. Wu, A. Murray, M. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. *11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pages 90–99, Nov. 2004.

[82] F. Yang. Using Excel and PowerPoint to build a reverse engineering tool. Master's thesis, Department of Computer Science, University of Victoria, 2003.

[83] Q. Zhu, Y. Chen, P. Kaminski, A. Weber, H. Kienle, and H. A. Müller. Leveraging Visio for adoption-centric reverse engineering tools. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 270–274, Nov. 2003.