Domain-Specifically Tailorable Languages and Software Architectures

Uwe Zdun
Institute for Computer Science
University of Essen, Germany
zdun@acm.org

Abstract

In this position paper we will motivate open issues in the context of domain-specific tailorability of software systems, as we have observed them in numerous projects. These issues are largely unresolved in today's software engineering practices. However, we can identify a set of recurring forces in this context. Different interaction and programming techniques can be used to partially resolve these issues. As a vision of a solution, we argue for building task-specific and extensible languages that map well to the primitives of the domain. The languages should also be well integrated with the software architecture fragments wrapped by the language elements.

1 Introduction

Christopher Alexander describes his intentions for tailorability in the domain of built architectures as follows [1]:

> In our own time, the production of environment has gone out of the hands of people who use the environment. So, one of the efforts of the pattern language was not merely to try and identify structural features which would make the environment positive or nurturing, but also to do it in a fashion which could be in everybody's hands, so that the whole thing would effectively then generate itself.

In a similar fashion, software is largely produced by people who are experts in the domain of software engineering, but not in the target domain of the software system. However, to date it is widely accepted that many customization and extension requirements cannot be foreseen in upfront designs. Therefore, in principal, it would make sense to hand those customizations over to users of the software system.

Those users are usually not naive or novice users, but often they are highly trained domain experts. Among them, there is a high demand for tailorability of software to their particular tasks, but most often, once the software is written, the applications cannot easily be changed or extended. We think that this problem is mainly due to the lack of highly tailorable software systems that leverage the user's interests and demands for domain- and task-specific changes. Actually, besides the important intent of improving the work situation of the users, there are strong economical reasons for more tailorable software systems as well. Programmers who are qualified in the particular domain are often relatively expensive, rare, and, in many cases, programming all customizations by hand simply consumes too much times.

We argue that formal languages used in software engineering, such as architecture definition languages, design languages, and programming languages, are often impenetrable for people who are not trained to use them. In his essay in [4] Gabriel criticizes that computing theory is based on a particularly inexpressive set of mathematical constructs, which are still apparent in today's theories and languages. Most significant programming languages are expressively equivalent to Fortran and assembly language. Software development methodologies evolved under a mythical belief in master planning. There is a hard-to-reveal assumption that engineers, mathematicians, and computer scientists are the only ones who will write a program or contribute to a software system.

From this point of view, tailorable software systems should rather concentrate on the particular domain's tasks and the interests of the users than try to educate them as programmers in mainstream general-purpose programming languages. Modern cognitive psychology indicates that people perform better at problems that are appearing in form of familiar words or concepts [6]. The most effective strategy for recall and problem solution is to use memory structures that already have been created. Thus, in a task-specifically tailorable software system domain experts would be able and motivated to introduce their task-specific knowledge and interests. They would be able to understand the system's tasks more easily. Ideally the domain experts could express domain semantics themselves. This should ease application development and evolution, and especially leverage accurate and timely customizations.

The very idea of building languages that benefit from the domain expert's task-specific knowledge and interests goes back to Martin's vision of problem-oriented languages, back in 1967:

> We must develop languages that the scientist, the architect, the teacher, and the layman can use without being computer experts. The language for each user must be as natural as possible to her/him. The statistician must talk to his terminal in the language of statistics. The civil engineer must use the language of civil engineering. When a man learns his profession he must learn the *problem-oriented languages* to go with that profession.

There have been many interaction and programming techniques, such as visual programming, programming by example, and generative programming techniques, been proposed as *the* solution since. However, three decades later, Martin's vision is still far from being fully realized. We will argue that these interaction and programming techniques only partly solve the underlying problem. In [10, chapter 4] visual languages, forms-based programming, programming by example modification, programming by example, and automatic program generation are compared for their capabilities in the end-user programming domain. The analysis of these techniques shows that all have their limitations. No single technique can address the semantic issue of designing a task-specific language for the relevant domains. However, any of these techniques may prove useful *in combination* with such a task-specific language.

2 Typical Forces for Tailorable Software Systems

We have investigated domain-specific tailoring, as discussed in the previous section, in different development and maintenance projects, including a highly tailorable programming language (XOTcl, see [11]), a document archive system [8], a web object system [12], a conference management system [14], a product-line for the Multimedia Home Platform (MHP) [16], and many others. There is a set of typical forces recurring in most application scenarios in which domain experts that are non-programmers have an interest in tailoring software. Let us recapitulate the main forces in this area:

- Often "tailors" are not novice or naive users but domain experts. Tailorable software systems should primarily target these users.
- The industrial reality of components is that they are of a large-scale and have no enforced interfaces and boundaries [2]. That is, often it is hard to reengineer given industrial components to software units that are comprehensible for domain experts.
- In contrast to the scope of many domain-specific languages and end-user tools, tailorability also has to be available for small niches of users and niche systems.
- Domain experts usually are generally willing to transfer domain knowledge into software systems if they can directly see a benefit for their work, but usually they are burdened with a lot of work. Therefore, as an important motivation, the interfaces to tailorable software systems should reflect their domain and interests [10].
- Tailorability is often expensive to build.
- Often users are forced to switch between many languages and interfaces.
- It is difficult to know how specific a task-specific software system should be. Being too specific means to limit the design space of the users, while being too general means to get languages and interfaces that are incomprehensible to domain experts.

For specific problem-domains these issues are resolved. Examples are CAD systems, spreadsheets, and multimedia authoring tools. However, for less popular domains and niches it is hard to find a domain-specific and flexible solution that resolves the forces described above.

2.1 Interaction and Programming Technique for Tailoring

In this section, we will summarize a few interaction and programming techniques for resolving the forces, discussed in the previous section, to a certain degree:

- *Scripting Languages* include Tcl, Python, and Perl. Unlike most compiled languages, which make use of code libraries written in the same language, scripting languages are often used as glueing languages. That is, they are used to glue together components which may not, themselves, be written in the scripting language. These components can implement task-specific language extensions. Scripting languages are typically tools for more serious users, such as web page authors, engineers, and network administrators. However many of them are not primarily programmers.
- *Visual Programming* [9] refers to systems that allow the user to specify a program in two-(or more)-dimensionsional fashion. Conventional textual languages are not considered two dimensional since the compilers or interpreters process them as long, one-dimensional streams.

- *Domain-Specific Languages* are small, usually declarative languages. In general they are particularly expressible in a certain problem domain. Domain-specific languages are often created in the context of domain engineering projects focused on achieving reuse and reliability in particular problem domains. Domain-specific languages have been used in various domains, and aim at higher productivity, reliability, and flexibility. Although domain-specific languages are an attractive alternative, often mainstream languages are preferred for the wealth of libraries available. Often domain-specific languages have problems with portability and long-term support, since they tend to be research projects. Several domain-specific languages have problems regarding integration with other domain-specific languages.

- *GUI Builders* are tools for composing graphical user interfaces and automatically generating code for the GUI. Other tasks are specified using ordinary programming languages like C++, Java, Tcl, or Visual Basic.

- *Programming by Example* (or "programming by demonstration") is a technique for teaching the computer new behavior by demonstrating actions on concrete examples. The system records user actions and generalizes a program that can be used in new examples.

- *Software Tinkering* refers to software systems that offer a programming interface for changing the system's behavior. A typical example of a tinkerable software system is Emacs. It is required to have an intimate knowledge of the tinkering interface to master it, but novices can create even sophisticated customizations by trial-and-error. For instance, in Emacs one can create sophisticated initializations without mastering Emacs Lisp by cut & paste and guessing what certain expressions do. Of course, tinkering by trial-and-error is always limited in its capabilities, but it can motivate users to learn the full languages or interfaces of the tinkerable system.

- *Generative Programming* [3] refers to systems that generate customized components to fulfill specified requirements. It allows for modifications of the given code fragments and assembly apart from given patterns. Some generative programming approaches require an education as a programmer (as for instance aspect-oriented programming), others, like web fragment composition approaches, target at the domain expert user.

3 Success Factors for a Tailorable Language

The previous sections should have motivated the demand of domain-experts for a task-specifically tailorable languages. The question is: how should a task-specifically tailorable language be designed? From benefits and liabilities of the languages underlying existing interaction and programming techniques we can deduce a set of success factors.

The domain expert's interests lies in his domain, and only when people have particular interest they learn a formal language [6]. Therefore, the language should be equipped with *task-specific language elements* that can be recognized by the domain experts without knowing a programming language. Language primitives should map to the tasks in the domain expert's domain. Then domain experts can understand the language primitives, and they can directly express domain semantics and manipulate programs on their own.

To avoid the problem of domain-specific languages that they are only usable in a particular domain, and that their language elements do not integrate well with other domain-specific languages, a tailorable language should be *extensible with new language elements*. Otherwise the domain expert has to learn a new language for each particular task. Instead, one base language should be usable for integration, and new domains should be integrated and composed as reusable components extending the base language.

From the success of software tinkering approaches we can learn that a tailorable language should allow novice users to directly manipulate programs, say, with cut & paste without having to learn a larger part of the underlying language. That is for simple tasks the user should merely use the language extensions for *local changes* in a hot-spot or center that is interesting to her/him. Only for more sophisticated tasks, such as writing new language elements, more sophisticated parts of the tailorable language should have to be learned. Thus there should be a fully *incremental learning curve* and almost no prerequisites for starting to tailor a software system.

Nevertheless, to avoid limiting expert users the language should be a complete programming language. Since visual programming always limits the user's expressivity to what can be displayed on the screen there should be a *textual notation* for the language, but it should also be possible to derive *other (e.g. graphical) views* easily. That is, where appropriate, the interaction techniques from Section 2.1 can be used on top of the task-specific language.

To integrate with given legacy components *multi-language integration* (such as component glueing) should be supported. The tailored programs often have to used in context of given systems. Thus the language has to be *embeddable* in such systems as well.

Another important aspect is the *tight integration with the software architecture*. Each language element should map

to one architecture element. For instance in XOTcl [11] we use object IDs as language elements. Such objects can, for instance, be wrappers to legacy components [7]. Then each method of the object wraps one function of the legacy component. The language can dynamically be expended with new language elements, and task-specific primitives can be composed with these language elements. Only the required functionality of the component framework is exposed to the domain expert.

Another is example of integrating task-specific languages and architectures is the pattern Generic Content Format [13]. Here, a content format is defined in a content format language, say, by using a DTD in XML. The corresponding information architecture reflects this content format. For example, in XML we have a hierarchical content structure. Thus we can build a Composite [5] structure with one class for each XML tag supported by the DTD. The classes implement a set of callback to be called upon certain events in the lifetime of the information architecture, such as construction, destruction, or updates.

4 Conclusion

In this position paper we have summarized a set of forces in the context of tailoring software systems that are largely unresolved by today's mainstream software engineering practices. However, in practice there is a high demand for domain-specific tailoring of software systems. Typical reasons for this demand are ease-of-work for domain experts, work efficiency, rapid customization requirements, and economic reasons. Different interaction and programming techniques resolve different parts of this problem set. However, a task-specific language that is tightly integrated with the software architecture can be adjusted to the specifics of a concrete software project, and is thus even feasible for smaller projects. We have discussed our results in form of a set of success factors that should outline a general vision for the development of task-specific languages. In different projects different solutions exposing these success factors can be applied. For instance, our language implementation XOTcl provides high tailorability by explicit language support for interpretational language extension [15]. In other projects, such as [8, 16], a little customization language is built on top of a given implementation language. Sometimes different approaches are combined. For instance, our conference management system [14] uses XML as a customization language for conference chairs, while XOTcl language resources are used for flexibility in maintaining the system.

References

[1] C. Alexander. The origins of pattern theory, the future of the theory, and the generation of a living world. *IEEE Software*, September/October 1999.

[2] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 1999.

[4] R. P. Gabriel. The feyerabend project: An invitation to redefine computing. http://www.dreamsongs.com/FeyerabendInvite.html, 2001.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] H. Gardner. *The Mind's New Science : A History of the Cognitive Revolution*. Basic Books, Inc. NY, 1985.

[7] M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.

[8] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. Accepted for publication in Journal of Software Maintenance: Research and Practice, 2001.

[9] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[10] B. Nardi. *A small Matter of Programming: Perspecives on End User Computing*. 1993.

[11] G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[12] G. Neumann and U. Zdun. Distributed web application development with active web objects. In *Proceedings of The 2nd International Conference on Internet Computing (IC'2001)*, Las Vegas, Nevada, USA, June 2001.

[13] O. Vogel and U. Zdun. Content conversion and generation on the web: A pattern language. submitted, 2002.

[14] U. Zdun. Dynamically generating web application fragments from page templates. In *Proceedings of Symposium of Applied Computing (SAC 2002)*, Madrid, Spain, March 2002.

[15] U. Zdun. *Language Support for Dynamic and Evolving Software Architectures*. PhD thesis, University of Essen, Germany, January 2002.

[16] U. Zdun. Xml-based dynamic content generation and conversion for the multimedia home platform. In *Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT)*, Pasadena, USA, June 2002.