Incremental Development of a Domain-Specific Language That Supports Multiple Application Styles

Kevin Bierhoff
ISRI, Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
kevin.bierhoff @ cs.cmu.edu

Edy S. Liongosari
Accenture Technology Labs
161 North Clark Street
Chicago, IL 60601, USA

Kishore S. Swaminathan
Accenture Technology Labs
161 North Clark Street
Chicago, IL 60601, USA

{ edy.s.liongosari, k.s.swaminathan } @ accenture.com

ABSTRACT

Domain-Specific Languages (DSLs) are typically built top-down by domain experts for a class of applications (rather than a specific application) that are anticipated in a domain. This paper investigates taking the opposite route: Building a DSL incrementally based on a series of example applications in a domain. The investigated domain of CRUD applications (create, retrieve, update, delete) imposes challenges including independence from application styles and platforms. The major advantages of an incremental approach are that the language is available for use much sooner, there is less upfront cost and the language and domain boundaries are defined organically. Our initial experiments suggest that this could be a viable approach provided certain careful design decisions are made upfront.

Keywords
Domain-specific languages, incremental design, application style, CRUD applications.

1. INTRODUCTION

Domain-specific languages (DSLs) have received a lot of attention over the last few years in the research community and in industry. They promise software development using languages that provide higher-level abstractions that are particularly tailored to a domain. The ideal approach would involve a handful of highly skilled language developers and domain experts who would collaborate and define the boundaries of a domain and come up with optimal abstractions for expressing concepts in the domain. Once the language is finalized, a "compiler" (code generator) for this domain-specific language would be developed to generate code for a target platform or language. The compiler can then be used to generate applications. The major advantage of this approach is that developers will—from the outset—work with a robust language designed by experts. The disadvantage is that it requires significant upfront investment in developing language and compiler while the usefulness of the DSL for specific applications is unknown.

In this paper we investigate an alternative "incremental" approach [12]. Rather than defining the domain or its boundaries, we choose a "typical" application that we use as a seed to develop a DSL expressive enough to describe this application. Then we attempt to add additional features to the seed application and write other applications in the domain. This may require extensions to the language and corresponding extensions to the code generator. Thus the DSL is driven by a series of examples rather than a detailed understanding of the domain without specific applications in mind. The expectation is that the language would continue to evolve until it is no longer fruitful or cost effective to extend it—thereby

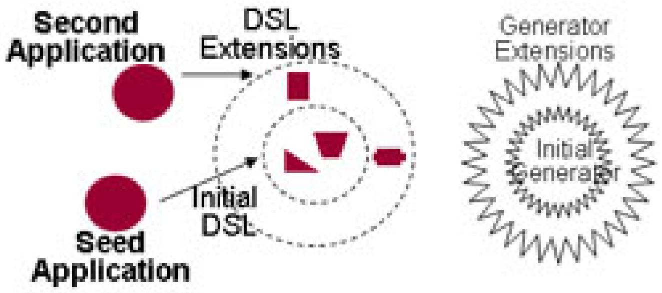


Figure 1: Incremental DSL Design Approach

defining the boundaries of the domain. The advantages of this approach are that the language is available for use much sooner, there is less upfront cost and the language and domain boundaries are defined organically. Moreover, the DSL is by definition useful for at least one application. The incremental approach is depicted in Figure 1.

This paper reports on incrementally building a DSL that targets the domain of CRUD applications (create, retrieve, update, delete). We design the language to cover multiple application styles and build two code generators to cover major style and platform choices. We evolve the language in three iterations. The incremental approach appears viable for this DSL; however, certain initial design decisions must be made carefully. Further investigation is necessary to understand some of the potential disadvantages of this approach such as language drift, proliferation of languages and their corresponding code generators and the dangers of language extensions made by novices.

In the following section we introduce the domain and application we chose for our investigation. The initial DSL we developed is discussed in section 3. Section 4 describes the code generators we implemented to target multiple platforms. Language evolution with additional features and examples is investigated in section 5. Section 6 summarizes related work and we conclude in section 7.

2. DOMAIN AND SEED APPLICATION

To explore the incremental evolution of a DSL, we chose the “domain” of CRUD (create, retrieve, update, delete) applications. We deliberately left the definition of the domain vague. We chose a simple application that involves creating and maintaining data.

The application was a “distributed to-do list” manager that allows a group of people to manage their to-do list as well as to assign tasks to each other. The to-do list can contain deadlines for the tasks and may be marked complete when done. The application did not have any security features other than login. Anybody can see everyone else’s to-do’s or assign tasks to anybody.

Below are more detailed functional requirements:

- Manage one to-do list that can be viewed and edited by the members of the group. All members view and edit the same to-do list.
- Each member of the group is represented by a separate user of the application (with initial login of some sort).
- The to-do list consists of to-do items.
- To-do items have a number of attributes:
    - title (required)
    - due date (required)
    - responsible group member (required)
    - completed flag (required)
    - free-text description (required)
- The to-do list can be viewed.
- The to-do list view displays each to-do item with title, due date, responsible group member, and completed flag.
- A to-do item from the list can be selected to view individually.
- The individual view of a to-do item shows all attributes of that item read-only.
- A to-do item can be selected for editing from the individual view. All attributes are made editable at the same time. Changes can be confirmed or cancelled.
- New items can be created at any time with a form similar to the one for changing an existing item.

- Saving a new or edited item occurs transactional. If two members edit an item concurrently there will be no warning and one member will override the changes of the other member.

3. A DSL FOR CRUD APPLICATIONS

This section discusses the domain-specific language that we developed based on the seed application presented in the last section. The requirements define our seed application based on its user interface and the managed data. Therefore we decided to split the DSL into two parts, one describing the manipulated data and one for the user interface. The following subsection deals with the definition of data. Afterwards we describe the definition of user interfaces. Tool support for testing and debugging is beyond the scope of this paper.

3.1 Data: Entities and Relationships

The vocabulary used for defining data ("to-do item", "responsible group member", etc.) is specific to the application. Unsurprisingly, the requirements distinguish different kinds of data ("to-do item", "group member") that are often called *entities* and represented as database tables. These entities have *attributes* ("completed flag" etc.) that sometimes refer to other entities ("responsible group member").

Since entity-relationship models are familiar to CRUD application designers we decided to borrow their principles for the definition of data in our DSL. We use a textual representation that is reminiscent of SQL statements for table creation. However, we simplify some aspects of the language. For example, we alleviate the developer from worrying about foreign key mappings. She can instead define relationships directly with keywords like "many-to-one" that cause automatic generation of appropriate foreign key constraints. Figure 2 shows the definition of entities for our seed application.

```
GroupMember = (
      username : String  key,
      password : String
)

TodoItem = (
      autokey,
      title : String,
      dueDate : Date,
      complete : Boolean,
      responsible : GroupMember  manytoone,
      description : String
)
```

Figure 2: To-do list example application entities

By borrowing from existing concepts (entity-relationship models in this case) we allow domain experts to use a framework they are already familiar with. We believe that this can reduce the problems of learning and adopting a domain-specific language. The textual representation is very useful for automatic tools (like the code generator) while a graphical representation is probably more convenient for the developer to use.

Our definition of entities conveniently introduces the notion of types into our system. We use some common base types to define primitive attributes. Entities become types themselves that can be used to type attributes (essentially defining relationships). Note that the type of each attribute is effectively given in the requirements of our seed application (strings, dates, and Booleans). Treating entities as types will allow us to perform static checks on the definition of the user interface. It also simplifies code generation. We believe that these are big benefits of a typed approach to designing a DSL that are often worth the effort.

3.2 User Interfaces

We now come to the definition of the user interfaces. The requirements of our seed application essentially describe two things: the different views (and edits) of the underlying data and the control and data flow among them. We call the former *interactions* and the latter *compositions*. Both are understood as *processes*.

A process is defined as a part of the program that has input values and ends with an *event*. This certainly applies to interactions (see below). By extending this idea to compositions we achieve a more scalable model that allows compositions to define the flow between processes (including other compositions) because compositions and interactions have a common interface (inputs and events).

The programmer has to explicitly write the signature (inputs and events) of each process. Notice that it would be quite simple to inject code on the level of a process. For example, this code could implement a decision and therefore end with one of two possible events. Code injection on this level is simple because the interface of the injected code is defined by the process signature.

Explicit signatures can also facilitate modular code generation. If the signature can be turned into constructs of the target platform then this representation is like a "calling convention" for processes in that platform. If one process invokes the other (when control flows from one to the other according to a composition) its generated code can use this calling convention and does not have to look beyond the signature of the invoked process.

3.2.1 Interactions

An interaction communicates data to the user, optionally allows changing that data, and ends with an event that the user initiates. Thus an interaction consists of a screen that is presented to the user and the logic to query and update the presented data.

Looking at the requirements of the to-do application it is interesting to notice that users can essentially only "view" or "edit" data. Thus the nature of user interactions is generic to the domain. Conversely, the exact information presented in an interaction is specific to an application. Thus we define an interaction in terms of *what* is presented to and edited by the user. Because the language should work both for Web and desktop applications we do *not* prescribe *how* data is presented.

For the definition of interactions we had to decide how developers can express what data they want to show (or edit). For example, viewing a to-do item should show the title of the item. We chose to follow the analogy of field selection in Java and C# and use a "dot" notation. Thus if a variable called "item" is passed into the interaction then "item.title" denotes its title. Field selections can be nested, just like in Java or C#. We will call such a construct a *term*.

Displaying a piece of data can be achieved by just writing the appropriate term, starting from a defined variable. Data can be made editable by surrounding a term with "edit". Moreover, editing has to happen inside a "update" or "insert" construct that makes certain variables subject to editing. Inside this construct, an event marked as "confirm" will commit changes into the database. Events that are not marked with "confirm" essentially discard any changes.

These concepts are again motivated by the domain. SQL distinguishes between inserting a new row and updating an existing row.[1] We combine this with the idea of "forms" in HTML that have the ability to submit or cancel any inputs made into the form by the user. However, in reflection on this work we think that the distinction between inserting and updating data could be hidden from the developer and silently addressed in the generated code.

Labels can be put next to displayed or edited terms to give the user some context about what she looks at. This is common practice in user interface design. The interactions in our seed

[1] While our language covers create and update operations, we did not need deletion for our example applications. Deletion may cause runtime errors when relational constraints are violated and could therefore motivate language extensions.

application either are concerned with an individual data element or iterate over a list of elements. We therefore define a “list” construct that implicitly iterates over the rows returned by a query. The “list” construct has a body that defines what should be displayed for each element in the list.

Similar to the definition of entities (above), the code generation for interactions is based on a textual representation. An equivalent hierarchically nested graphical representation of display and edit elements with their labels could be defined that would look very similar to graphical tools for designing desktop applications. Figure 3 shows some of the interactions for viewing and manipulating to-do items in our seed application.

Formally, an important concept in defining interactions is the scope of variables. Most variables are introduced with names for input values in the signature of an interaction. The “list” and “insert” constructs introduce additional variables (for the current and new data element, respectively) that are only available in their bodies. This means that the types of all variables are known statically. Based on the information given in the definition of entities, the types of terms can be inferred. This enables static checks to make sure that for example only primitive attributes are displayed. (It is not clear what displaying a “group member” would even mean, while displaying its username is straightforward.)

```
interaction TodoList() emits SelectItem(TodoItem),
NewItem(), NewUser(), Exit()
{
    list(select TodoItem i)
    {
        "Title"        i.title -> SelectItem(i),
        "Due"          i.dueDate,
        "Completed"    i.complete,
        "Responsible"  i.responsible.username
    },
    "New Item" -> NewItem(),
    "New User" -> NewUser(),
    "Exit" -> Exit()
}

interaction View(TodoItem item)
emits Edit(TodoItem), Back()
{
    "Title"            item.title,
    "Due"              item.dueDate,
    "Completed"        item.complete,
    "Responsible"      item.responsible.username,
    "Description"      item.description,
    "Edit" -> Edit(item),
    "Back" -> Back()
}

interaction Edit(TodoItem item) emits Continue()
{
    update(item)
    {
        "Title"        edit(item.title),
        "Due"          edit(item.dueDate),
        "Completed"    edit(item.complete),
        "Responsible"  edit(item.responsible),
        "Description"  edit(item.description),
        "OK"           confirm(Continue()),
        "Cancel" -> Continue()
    }
}
```

Figure 3: To-do list example application interactions

We point out that we can also rely on the types to determine automatically how terms are formatted for display and what controls are used for editing them. For example, a term that is a date should be formatted as such. A Boolean term should probably be edited through a checkbox while a term of entity type should be edited with a combo box that shows the available entries. However, additional configuration could change these defaults.

Thus we found that typing of terms not only enables static validity checks of programs written in our DSL; it also stratifies the DSL because the code generator can infer the omitted information (such as appropriate edit controls and formatting) from the types.

3.2.2 Compositions

Compositions define the flow of control between processes. They essentially “wire” each event of a process to another process that is invoked when the event occurs. Events can carry parameters that are transferred to the invoked process. Figure 4 shows how the interactions of our seed application are composed.

The overall flow can be defined with hierarchically aggregated compositions. This seems helpful from an engineering point of view. For example, we used this idea to define the flow between the various interactions for displaying and manipulating to-do items in a composition that is in turn composed with the login screen to form the complete to-do list application.

We point out that a complete separation of flow and interactions as realized in our DSL is not commonly achieved in desktop or Web application frameworks. Although they typically follow a Model-View-Controller pattern [5], flow information “leaks” into the view where buttons or links point directly to their following dialog or Web page. We strictly separate the two through our event abstraction that essentially represents clicks on buttons or links.

This has in our view a number of benefits. Firstly, this approach provides separation of concerns and potentially allows better re-use of interactions or even compositions. Unfortunately we could not achieve this in our seed application even though the interactions for creating a new to-do item and editing an existing one look identical. The problem was our distinction of insert and update that is made inside the interaction. Dropping this distinction would open the possibility of reusing the interaction as discussed here.

Secondly, we can conveniently define a graphical language that lets developers “wire” processes in a kind of box-and-line diagram. In fact this is reminiscent of component-and-connector diagrams in UML2 [6] where processes are components and compositions define the connectors. Processes have exactly one “input” port and each of their events defines a separate port. This seems to suggest that CRUD applications (at least in our DSL) follow a particular *architectural style* [1].

Thirdly, we can use the wiring to bridge gaps between the event coming from one process and the input values required for another interaction. For example, we can perform conversions, run a database query or add additional parameter in between. We used this feature to verify credentials between the login screen and the rest of the seed application.

```
composition Todo(GroupMember user)
{
    TodoList      itemList;
    View    viewItem;
    Edit    editItem;
    New           newItem;
    NewMember     newUser;

    begin with itemList()
    {
        itemList.SelectItem(item)->viewItem(item);
        itemList.NewItem() -> newItem();
        itemList.NewUser() -> newUser();
        itemList.Exit() -> Exit();
        viewItem.Edit(item) -> editItem(item);
        viewItem.Back() -> itemList();
        editItem.Continue() -> itemList();
        newItem.Continue() -> itemList();
        newUser.Continue() -> itemList();
    }
}
```

Figure 4: To-do list example application composition

4. CODE GENERATION FOR MULTIPLE PLATFORMS

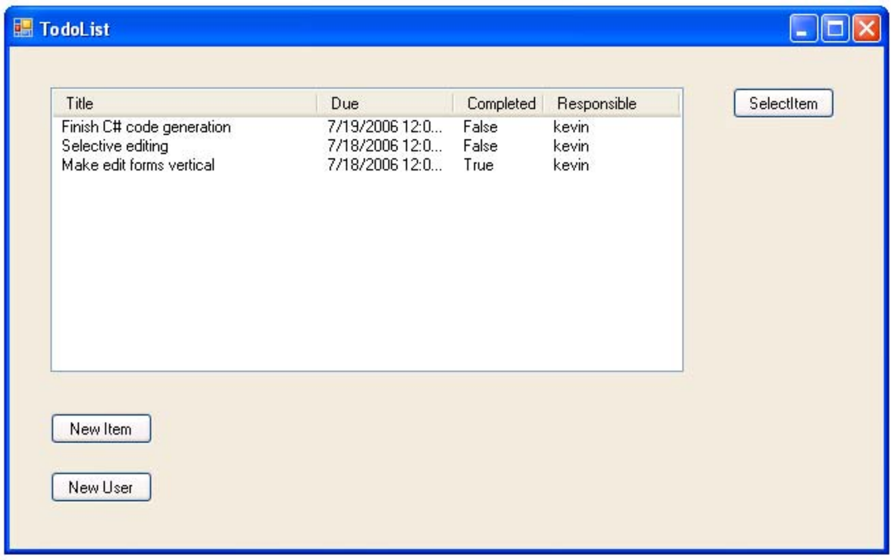
An interesting aspect of CRUD applications is that they are sometimes implemented as desktop applications and sometimes as Web applications. Desktop and Web applications have different architectures and user interfaces. We refer to these as *application styles*. Desktop applications are thick clients that interact with the user through dialog boxes and can potentially access the database directly. Web applications, on the other hand, use Web browsers as thin clients that display a typically HTML-based user interface. The actual application resides within an application server that in turn accesses the database. (Variations of these characterizations are possible, but these are in our experience typical implementation techniques.)

Our DSL supports both desktop and Web applications with their different application styles. Moreover, we are independent from a particular programming language and explicitly support both Java and C#. In particular, we implemented two code generators for our DSL. One generates C# desktop applications that communicate directly with an underlying database through SQL commands (using ADO.NET). The C# desktop applications use the .NET Windows forms library for their user interface (Figure 5). An implementation in Java would use corresponding standard Java libraries (JDBC and Swing).

The other code generator produces J2EE Web applications based on Enterprise JavaBeans (EJBs) and Java ServerPages (JSPs). EJBs provide a sophisticated way of representing and manipulating database rows through objects called Entity Beans. JSPs are essentially HTML files with interspersed Java code that are typically used for Java-based Web interfaces. The ASP.NET framework provides better support for handling state associated with Web pages than JSPs. On the other hand, .NET does not offer database abstractions that are as powerful as EJBs. We bridge these gaps by generating additional code. For example, our C# code generator creates SQL commands while our J2EE code generator relies on EJBs.

Independent from the problem of supporting multiple programming languages we had to address the discrepancies between desktop and Web applications in general. The differences in their architectures were relatively easy to handle. Conversely, their user interfaces are vastly different. We address this problem with two user interface representations specific to desktop and Web interfaces. In the case of desktop applications this representation is essentially a nesting of “panels” that contain atomic elements such as labels and buttons. In the case of Web applications the representation is HTML with special tags to represent concepts of our DSL. Thus these representations mostly capture the different natures of user interfaces (dialog boxes vs. Web pages).

The concepts of our DSL could often be turned directly into corresponding user interface concepts. For example, an event corresponds to a button in a dialog box and a link in a Web page. However, the treatment of lists posed a problem. In our DSL we allow to connect each element in the list with events that are pa-



Title	Due	Completed	Responsible
Finish C# code generation	7/19/2006 12:0...	False	kevin
Selective editing	7/18/2006 12:0...	False	kevin
Make edit forms vertical	7/18/2006 12:0...	True	kevin

Figure 5: Generated to-do list application dialog

rameterized by that element. In a Web page this can be represented as a table with links in each row. In a dialog box, this is not the typical user interface. Thus we instead let the user select an item in the list and provide one button for each event next to the list.

4.1 Code Generator Design

Both code generators use the same input language syntax and parser component. Code is typically generated in multiple steps. Each step is a transformation from one language into another. The most complicated transformations are required for generating code for interactions. Essentially we proceed in three steps.

1. We transform each interaction into a representation that reflects the desired type of user interface (desktop or Web-based). For both user interfaces this step introduces unique names for all elements in a screen.
2. The user interface representation is transformed into a language-dependent form that for example denotes the C# classes to be used for displaying certain elements.
3. This language-dependent form is then written into files.

The intermediate user interface representation output by the first step is intended to have a textual representation. It is designed to be checkable for consistency with the original interaction definition and could therefore be modified by the developer. Conversely, the subsequent steps should not be modified by the developer. Instead she can use the unique naming of elements introduced in step 1 to provide separate configuration files. The calling convention of processes (section 3.2) opens another way of injecting code.

We do not believe that all DSLs must provision these ways of influencing the code generator. We do believe that all three ways (direct editing of intermediate representation, configuration, and code injection) are viable options that have their individual tradeoffs. We think that explicit intermediate representations are particularly useful if they still carry domain concepts. Other ways of affecting a code generator become more useful once the code generation is tied to a particular programming language (because the developer can write code in that language directly).

The introduction of types into our DSL proved very helpful in making sure the code generator worked correctly. Essentially we could map types from the DSL into the target platform. The compiler of the generated code (Java or C#) could then use that information to typecheck our generated code. This helped detecting semantic errors in the code generator that are indicated by typing errors in the generated code. Of course this scheme still relies on test cases, but many code generator errors can be found before the generated code is executed. This idea of typed compilation [2] worked very well in our experience.

It is quite interesting to notice that there are alternatives to generating all necessary code. A powerful runtime could essentially perform the same tasks as a code generator but without the trouble of printing code lines. For example it is conceivable to write a C# program that takes an interaction definition and constructs a corresponding Windows forms object graph. Such a runtime would work like a virtual machine. In contrast, our code generation works like a compiler into binary code. A third alternative in the middle is to generate (hopefully less) code that plugs into an appropriate framework such as Apple's WebObjects or Ruby on Rails.

Understanding the benefits and drawbacks of these alternatives to make informed decisions about DSL implementation strategies is an important aspect of future work. We do point out that runtime and framework approaches are fundamentally limited by language and architecture boundaries. We believe that part of the value of DSLs lies in their ability to bridge these gaps and tie different kinds of artifacts together.

5. LANGUAGE EVOLUTION

In order to find out how incremental DSL design fares we employed the following methodology.

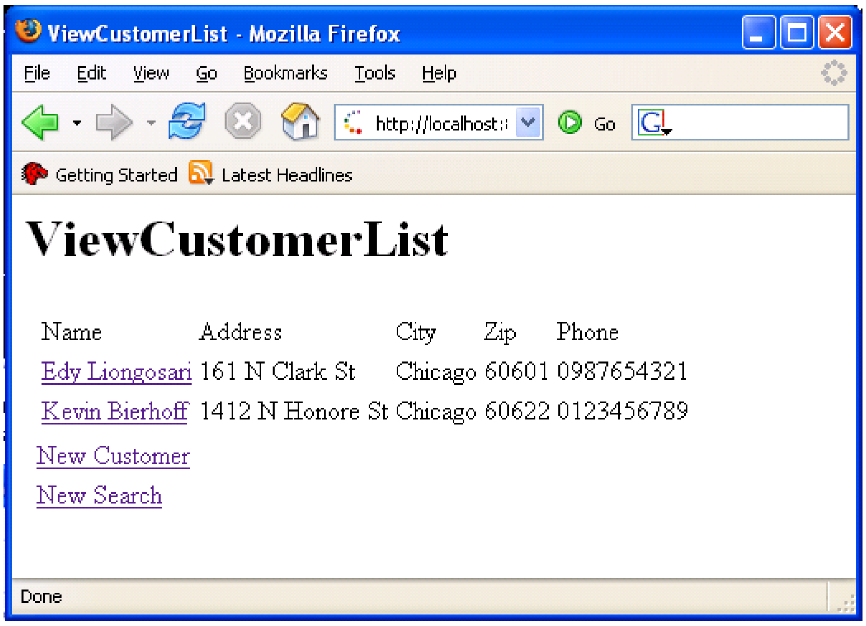
1. We chose a domain, in this case the domain of CRUD applications. The domain has the interesting feature that it spans multiple user interface paradigms and software architectures.
2. We defined informal requirements for a seed application in that domain. The application manages a simple to-do list for a group of people.
3. We implemented the application by hand with available technologies.
4. One of the authors designed a DSL based on the seed application and implemented code generators for the domain that cover the major user interface and architecture alternatives.
5. Another author then defined a new feature for the seed application that was implemented using the DSL.
6. Finally, a third author (successfully) attempted to write a completely new application in the DSL.

This section reports on how the language evolved with the addition of a new feature to the original seed application and the creation of a new application.

Additional Feature. The additional feature was to add a reporting facility to the to-do list application. It should be possible to look at the “unfinished business” of each group member, i.e. the incomplete to-do items the group member is responsible for.

This reporting facility was not immediately expressible in the DSL because Boolean constants were not part of the language and were also not allowed within “where” clauses of queries. (A “where” clause restricts the list of selected rows.) These shortcomings could be addressed easily.

New application. The new application was a “customer update” application that maintains customer records. It was a very simple application with only one entity (“customer”), five interactions, and one composition.



ViewCustomerList

Name	Address	City	Zip	Phone
Edy Liongosari	161 N Clark St	Chicago	60601	0987654321
Kevin Bierhoff	1412 N Honore St	Chicago	60622	0123456789

New Customer

New Search

Figure 6: Generated customer update Web page

The author who proposed this application wrote it himself. Notice that he had not been involved in the development of the original DSL. He wrote the application by adapting the seed application in just 30 minutes. His implementation was correct and had only minimal syntactic errors. We think that this is a somewhat surprising result.

The only construct that had to be added to the original DSL was support for “like” in “where” clauses of queries. (Originally, we had only supported equality tests.) This supports our hypothesis that references to existing domain concepts helps in adopting a language. The author that developed the application was familiar with SQL and therefore assumed the presence of a “like” keyword (with the typically associated semantics). Again, the additional keyword could be easily added.

5.1 Code Generator Evolution

Every change to the DSL has to be reflected in its implementing code generators. In the worst case, new code generators have to be built from scratch. However, it seems desirable to evolve the code generators with the language. With our code generators and the two evolution steps described above this was possible. In fact, the necessary changes to the code generators were minimal and could be implemented in about two hours. Language changes could be reflected easily in these cases for two reasons.

1. We used a parser generator that localized changes to the language grammar.
2. Our code generators implemented the necessary transformations using visitor patterns [6]. (We used Java to implement the code generators.)

Visitor patterns implement a recursive traversal of a data structure that calls a different method in the actual “visitor” depending on the type of node currently visited. Visitor patterns allow the compiler (of the language in which the code generator is implemented) to “drive” language extensions. Whenever a new kind of node is added to the data structure definition the compiler will require a traversal method for that node. The implementation of this traversal method will in turn require a new method in the abstract visitor interface. When this new method is added the compiler will point out all the places where concrete visitors need this additional method.

Thus we could quickly find all places in the code generator that were affected by the addition of new language constructs. Modifying existing constructs is somewhat more dangerous because the compiler cannot necessarily help with finding all places that have to be changed.

We believe that the ability to evolve code generators together with the DSL they implement is crucial for the success of an incremental design approach to DSLs. More research is needed to determine how code generators can be designed to achieve this.

5.2 Incremental Language Development

Our experiences with expressing additional requirements and a different application suggest that building DSLs incrementally is possible and useful in practice. We experienced two kinds of incremental language development. Firstly, the language itself can evolve (e.g., by defining new constructs) that in turn require changes to code generators. Secondly, code generators can be built incrementally to support more and more language features. For example, our code generators do not currently support nested lists even though they are allowed by the language grammar. Nested lists are simply not needed in any of our applications and also tedious to represent. Thus the code generators “trail” the language definition.

We believe that both kinds of incremental development mentioned above are useful. Changing the language itself is inevitable when facing requirements that cannot be expressed. Supporting only a subset of the language with code generators can potentially reduce the upfront investment in building a DSL. By first implementing more common constructs, a lot of the value of the DSL is available early. It even seems possible that some constructs will never be used. Of course, when constructs are not supported the code generator should detect these cases and notify the developer about the problem.

When designing the original DSL we constantly faced the tradeoff between a simpler, more general language and a language that strictly only accommodates the seed application. This tradeoff is probably inevitable in an incremental approach. We made choices in both ways along the way. However, whenever we chose a more expressive language we still had the option of not fully supporting it in the code generation.

6. RELATED WORK

Incremental approaches to application development are commonplace in software engineering methodologies [11]. Tolvanen investigated "incremental method engineering" for individual companies [12]. We focus specifically on DSLs and investigate design challenges, in particular independence from application styles. "Bottom-up" programming is a theme in the Lisp community to build higher-level abstractions out of lower-level abstractions [6]. Some of the challenges there are similar to ours although independence from a programming language is not a goal when using Lisp macros. Finally, possible tools for DSLs such as Microsoft Visio and Visual Studio as well as the Eclipse Modeling Framework (EMF) currently do not seem to provide a great deal of help in developing or even evolving the relatively complex transformations that we implemented using visitor patterns. A more in-depth analysis of these tools and in particular model transformation frameworks is future work.

In order to understand the impact of our incremental design approach better, we deliberately did not perform an extensive literature search on existing DSLs in the targeted domain. We were familiar with Strudel [4], a Web-site management system that can generate static Web pages based on an SQL database. We build on Strudel's syntax to define interactions with label-term pairs. In contrast to our work, Strudel does not support data updates and is only intended for Web applications. Other tools similar to Strudel ([2], [10]) have similar limits. We were also familiar with the domain itself and some of the technologies commonly used to implement CRUD applications.

Luoma et al. categorized DSLs by their "looks", i.e. the principle that was applied in defining their appearance [8]. Our language probably falls into the "expert's or developer's concepts" category. It is not apparent that any of the DSLs they surveyed was built incrementally.

There has been work in the research community on using continuation-passing style (CPS) in Web applications (e.g., [6]). Our events are essentially continuations. We provide the separate concept of composition that is not necessarily achieved with continuations.

7. CONCLUSIONS

DSLs are widely said to reduce development effort by providing high-level domain-specific abstractions. The reduced development effort, however, comes with high upfront investment into designing and implementing DSLs. In this paper we investigated an incremental approach to designing a DSL that is driven by a series of example applications. We found that such an approach is viable in that it produces a DSL general enough to extend to new examples. The DSL requires less careful initial design, possibly leading to reduced upfront design effort, and evolves incrementally. Even though developed for a seed application our DSL was able to span multiple platforms, user interface paradigms, and architectures. We believe a crucial condition for the success of such an approach is the ability to evolve code generators incrementally together with the DSL they implement.

Our results are very preliminary and motivate a number of research directions. We are curious to see how our language fares when applied to more applications. Moreover, it would be insightful to try an incremental approach on other domains. Arguably the domain we chose is a "horizontal" domain that provides computational and user interface abstractions. Applying an incremental approach to a vertical domain could be very beneficial because language designers need less initial domain knowledge. In addition, we make several observations about possible tradeoffs between language design and implementation alternatives that could be investigated. Finally, an analysis of how DSL tools support developing and evolving complex transformation engines would be insightful.

8. ACKNOWLEDGMENTS

The authors wish to thank Matthew Hellige, Scott Kurth, and the anonymous reviewers for their helpful feedback on this material.

9. REFERENCES

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4): 319-364, October 1995.

[2] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *International Conference on Very Large Databases* (VLDB), pp. 206-215, 1997.

[3] K. Crary. Toward a Foundational Typed Assembly Language. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[4] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *ACM SIGMOD International Conference on Management of Data*, pp. 414-425. Seattle (WA), USA, June 2-4, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[6] P. Graham. *On Lisp. Advanced Techniques for Common Lisp.* Prentice-Hall, 1993.

[7] P. T. Graunke, R. B. Findler, S. Krishnamurthi, M. Felleisen. Automatically Restructuring Programs for the Web. In *IEEE International Symposium on Automated Software Engineering*, 2001.

[8] J. Luoma, S. Kelly, J.-P. Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In *4th Workshop on Domain-Specific Modeling* (DSM), 2004.

[9] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual* (2nd ed). Addison-Wesley, 2005.

[10] J. Siméon and S. Cluet. Using YAT to Build a Web Server. In *International Workshop on the Web and Databases* (WebDB), Valencia, 1998.

[11] I. Sommerville. *Software Engineering* (7th ed). Addison-Wesley, 2004.

[12] J.-P. Tolvanen. *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence.* Ph.D. thesis, University of Jyväskylä, 1998.