The Journal of Systems and Software 101 (2014) 69–85

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A separation-based UI architecture with a DSL for role specialization

Ivan Gibbs*, Sergiu Dascalu, Frederick C. Harris, Jr.

University of Nevada, Reno, Reno, NV 89557, USA

ARTICLE INFO

Article history:
Received 6 May 2014
Revised 25 September 2014
Accepted 18 November 2014
Available online 28 November 2014

Keywords:
Domain specific language
Model driven engineering
User experience

ABSTRACT

This paper proposes an architecture and associated methodology to separate front end UI concerns from back end coding concerns to improve the platform flexibility, shorten the development time, and increase the productivity of developers. Typical UI development is heavily dependent upon the underlying platform, framework, or tool used to create it, which results in a number of problems. We took a separation-based UI architecture and modified it with a domain specific language to support the independence of UI creation thereby resolving some of the aforementioned problems. A methodology incorporating this architecture into the development process is proposed. A climate science application was created to verify the validity of the methodology using modern practices of UX, DSLs, code generation, and model-driven engineering. Analyzing related work provides an overview of other methods similar to our method. Subsequently we evaluate the climate science application, conclude, and detail future work.

© 2014 Published by Elsevier Inc.

1. Introduction

In software development there are many deadlines, dead ends, long hours, and other difficulties. We believe that a large amount of the accidental complexity (Brooks, 1995) contained in the development of a software project lies at the boundaries between programmers and higher level designers. Our focus was specifically on the gap between User Interface (UI) designers and programmers, which we believe is becoming more complex due to two trends: (i) the desire to attract and keep users is resulting in increasing complexities in the UI and (ii) the diversity of UI platforms is growing due to new devices such as tablets, smartphones, Google glass, and others that will be created in the future. The current status quo of UI development is to allow a UI designer to specify the UI while the programmer uses a UI builder and associated framework to create the UI. We believe that this status quo will become increasingly difficult to deal with due to the aforementioned trends. This exchange is hampered by a communication gap between those two groups, an accidental complexity that we have identified and attempted to rectify in our approach.

We address the UI–Code interface of the traditional separation based UI architecture (Fig. 1) in order to simplify that interface and thereby alleviate a number of difficulties pertaining to developing software. In contrast to other approaches, we attempt to determine the design of the UI–Code interface by basing it on *specialized roles* rather than solely on the code and some principle such as don't repeat yourself (DRY; Hunt and Thomas, 1999). Our architecture emphasizes specialized roles involving User Experi- 35 ence (UX) professional designers and programmers. The architecture and the resulting methodology based on it benefit from automated code generation and are generic and flexible enough to be applied in numerous software development projects. Our main premise is that specialization, combined with a mechanism for integration and bridging knowledge domains such as a Domain Specific Language (DSL) can be particularly effective in software applications with significant UIs.

We combined existing methods and technologies in order to fashion a software development approach to address the difficulties involved with changes of the UI that inevitable occur when UI design and programming are happening concurrently. The techniques used in our approach include Interaction Design (IxD), DSLs, code generation, Graphical User Interface (GUI) builders, and Integrated Development Environment (IDE)s. Due to time constraints, we focused on the tools as they are today and did not attempt to greatly modify them for our specific purposes.

The significance of this work stems first from addressing problems that arise from the current status quo in developing software with a UI. We see four problems that will become worse with trends (i) and (ii). The first two problems are related to the people developing the software: (1) UX professionals are resigned to an advisory role and, (2) communication gaps between UX professionals and programmers cause confusion and loss of productivity. Another set of problems relates to the technology, namely (3) UI dependence upon a framework, and (4) UI creation dependence on knowing programming. Take note that these problems are not orthogonal and that (1) is a result of (4). Our software development methodology addresses and alleviates each of these problems thereby providing a new status quo to deliver developers to a more productive future.

* Corresponding author. Tel.: +1 7753848968.
E-mail addresses: igibbs@cse.unr.edu, igibbs.cse@gmail.com (I. Gibbs), dascalus@cse.unr.edu (S. Dascalu), fredh@cse.unr.edu (F.C. Harris, Jr.).

http://dx.doi.org/10.1016/j.jss.2014.11.039
0164-1212/© 2014 Published by Elsevier Inc.

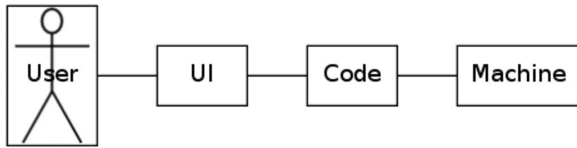

Fig. 1. Separation based UI diagram.

The paper is structured as follows. The derivation and description of the approach is presented in Section 2. The creation of an application using the methodology is described in Section 3. Section 4 provides an evaluation of the methodology as compared to similar approaches. The application we created is evaluated in Section 5 with a usability analysis, a code generation assessment, and a comparative analysis. Finally, Section 6 presents our conclusions and future work. This paper is based upon Gibbs (2013), which can be probed for further details of the proposed methodology.

2. The approach

2.1. Derivation of the approach

This section describes the essential aspects of our proposed approach. The approach is a combination of a separation-based UI architecture, UX considerations, and Model-Driven Engineering (MDE). We identify the premises we used to base our architectural decisions on and illustrate our deductive process to create the architecture. The approach is then described in detail with regards to how it should be implemented. The role of the developers is explained and the tools and techniques we used are described.

2.1.1. Premise 1: Role specialization increases productivity and success

Michael Jordan was a top athlete in professional basketball, who decided to play baseball. However, he was mediocre in baseball. Surprisingly he decided to go back to basketball and again became a top athlete. Why would Michael Jordan meet with less success in baseball than basketball? The problem here is specialization—people adapt to their environment and the more adapted they become to one environment the less adapted they will be to another environment. Though mental abilities are not as apparent as physical ones they are still there and without getting into a Darwinian discussion of heredity versus environment, we propose that mental abilities can limit the effectiveness of a person to a particular environment.

The Johnson O'Connor Research Center has measured the aptitudes of software engineers (Burke and Fitzgerald, 2003) and they have also done this for psychologists (Condon and Schroeder, 2005). Since UX designers often may have a background in psychology as they need detailed understanding of human users, we assume the aptitude profile of the UX designer to be close to that of a psychologist, in lieu of a better comparison. Fig. 2 provides evidence that the skills needed by UX designers and programmers are very different, thereby indicating poor performance of those individuals working in

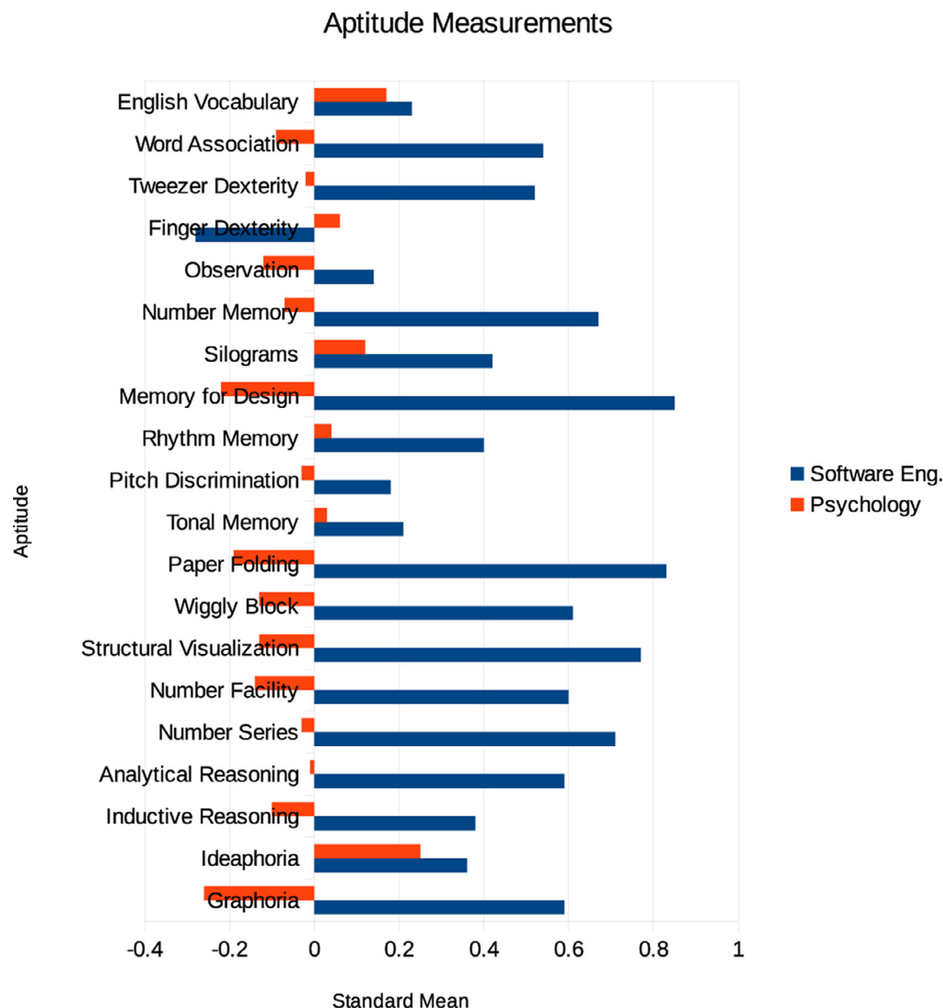

Fig. 2. Scores of software engineers and psychologists (Burke and Fitzgerald, 2003; Condon and Schroeder, 2005).

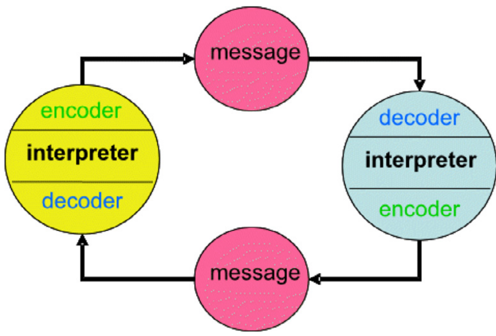

Fig. 3. A model of communication (Lynch, 2014).

the wrong area. To sum up our perspective, we think the invariant of role specialization in humans increases productivity and success.

2.1.2. Premise 2: Communication gaps cause confusion and inefficiency

A communication gap occurs when the sender and receiver have different conceptual meanings for words. A common phrase in advertising is "Nothing is too good for our customers." Gause and Weinberg (1990) Upon a close inspection, we can derive two different meanings for this statement: (1) "Our customers deserve so much that nothing in the world can actually meet this requirement", or (2) "Our customers are so undeserving that giving them nothing would be giving them too much." Another example Cooper et al. (2007) is when a user asks the computer to "Find restaurants in Virginia and Georgia," the user wants (restaurants in Virginia) AND (restaurants in Georgia). However, a computer programmer would set up the UI to interpret the phrase so as to finding restaurants in (Virginia AND Georgia)—an impossibility. So, the looseness of language contributes to possible confusions during communication.

The communication gap problem is complex and multiple models of communication have been proposed, such as the one shown in Fig. 3. A prime candidate for the illustration of communication gaps is provided by the creation of user interfaces by programmers. A popular UX professional writes

> "Our first four textual bloopers are about poor writing in the text displayed by software. They are the result of giving the job of writing text to the wrong people: Programmers." Johnson (2000)

Numerous examples of programmers failing to write software to communicate well with customers illustrates the fact that programmer skill sets are distinct and do not generally enable programmers to communicate clearly with the general population (Cooper, 1999). However, many programmers are not aware of the apparent fact that the sender and receiver of a message could have entirely different encoding/decoding mechanisms (Fig. 3).

2.1.3. Premise 3: The largest communication gap in a separation-based UI architecture is the UI–Code gap, between the user and machine knowledge domains

We found that in any piece of software there are many communication gaps that can be identified. In this paper, we attempt to classify the different communication gaps that we are aware of and to address only the largest of those gaps in order to reduce complexity in that way. Software constitutes whatever we insert in between the user and the machine. Therefore, we can consider this communication gap to be represented by CG. If we illustrate the software using a separation based UI architecture, then we notice that we actually have three separate communication gaps; cg_1, cg_2, and cg_3 (Fig. 4). Here, we can find the largest communication gap by specifying the actual knowledge domains that each component belongs to (see Fig. 5). It is this communication gap, cg_2, that we will address in order to alleviate some of the problems posed by communication gaps.

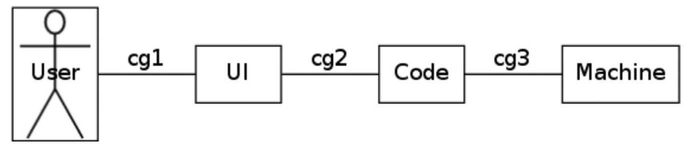

Fig. 4. Separation-based UI architecture labeled with communication gaps.

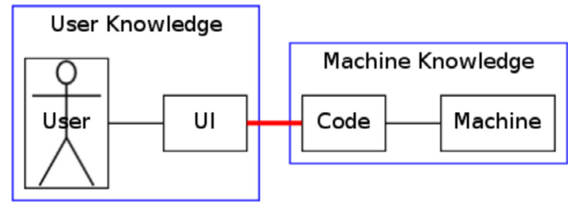

Fig. 5. Identification of knowledge domains.

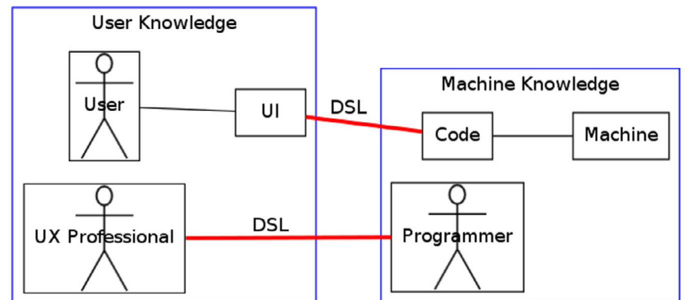

Fig. 6. Architecture of the approach.

2.1.4. Assertion: A commonly understood DSL can bridge the gap between the user and machine knowledge domains

There are ways to eliminate or reduce a communication gap, such as having one person learn the jargon of the other, having both people learn the others jargon, or providing an interpreter. From premise 1, we believe that attempting to educate the programmer regarding user knowledge is the wrong direction. The same applies to attempting to make the user learn more about machine knowledge. Instead, we subscribe to an interpreter option, and that interpreter is a UX professional who understands the realm of Human–Computer Interaction (HCI) theory (Rogers, 2012). The UX professional serves as the interpreter between the user and the code and is not required to understand machine knowledge.

By isolating our professionals in their respective knowledge domains, we also isolate their communication to that of a discussion of one communication gap between their knowledge domains—the largest communication gap of the architecture shown in Fig. 4 (Premise 3). This isolates much of the confusion and allows for a focused effort to be put on bridging the largest communication gap (Premise 2).

In an effort to tackle this gap, we will bridge it with a DSL in order to use a technology that already exists and has available literature describing it (see Fig. 6). This DSL will also eliminate the need for the UX professional to understand machine knowledge or how to program, thereby supporting Premise 1.

2.2. Description of the proposed approach

Our approach brings together three different ideas, that of UX , DSLs, and code generation. We believe that much of current industrial practice for software creation follows (Fig. 7). The designer is

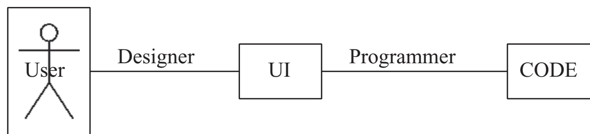

Fig. 7. Typical workflow.

responsible for defining the user's needs and specifying what needs to be done with the UI. The programmer then translates this specification into a program consisting of two parts—the UI and Code.

Our approach differs in that it specifies a UX designer instead of a general designer and the role specialization is isolated with a DSL. In order to further increase productivity for the machine knowledge domain, we use MDE code generation techniques to quicken software development, reduce errors, and increase flexibility (Fig. 8).

The UX designer will be an expert in user interfaces and is not required to understand how to program. The UI will be created by the designer through the use of a simple GUI builder or by directing a programmer. Though there exist numerous GUI builders, we have not found many that do not require some understanding of programming languages and these also have limitations in expressivity which detract from the creation of some UIs. The UI designer will interact with the DSL in order to communicate with the application.

The programmer will need to understand programming and DSL creation. The DSL will be defined by the stakeholders during the requirements and design stages of development. The programmer will create the DSL (another option would be to have this created by a language designer if resources exist). All technical issues arising during the UX design can also be solved by the programmer. The ultimate responsibility of the programmer is to create the software application code through code generation and manual edits.

The overall process is shown in Fig. 9. We begin with gathering requirements for the application and then create the software requirements specification (SRS; Sommerville, 2010) with a specific vocabulary of terms, precisely defined, in order to prepare for our eventual DSL creation. From there, the design will be created to describe how a technical solution will be reached to meet the SRS and here we also focus on specifying terms exactly for the next stage. The critical stage of creating the DSL then gives the design a formal description from the perspective of a UI interacting with an application, because the intention of creating the DSL is to make a well defined interface between the UI and the application. For example, in an ATM application, requirements such as "USER deposits MONEY," "USER withdraws MONEY," and "USER closes ACCOUNT" clearly indicate an interface between the USER and the ATM machine, which can be formalized with a DSL. Now the benefits begin to show up because the code creation and UI creation may now progress independently. The UI may be prototyped, tested with users, and finalized. Consequently, the DSL can be used to generate partial code in lieu of full generation, such that code can be added by the programmer after the generation phase. After the UI and code have been created, they will be integrated to create the final product. At this point, we note that the integration phase consists of merging a UI which interfaces with a human and produces a DSL script to communicate to the application, while the application is controlled via the DSL commands. The flexibility here is that two or more separate UIs can be integrated

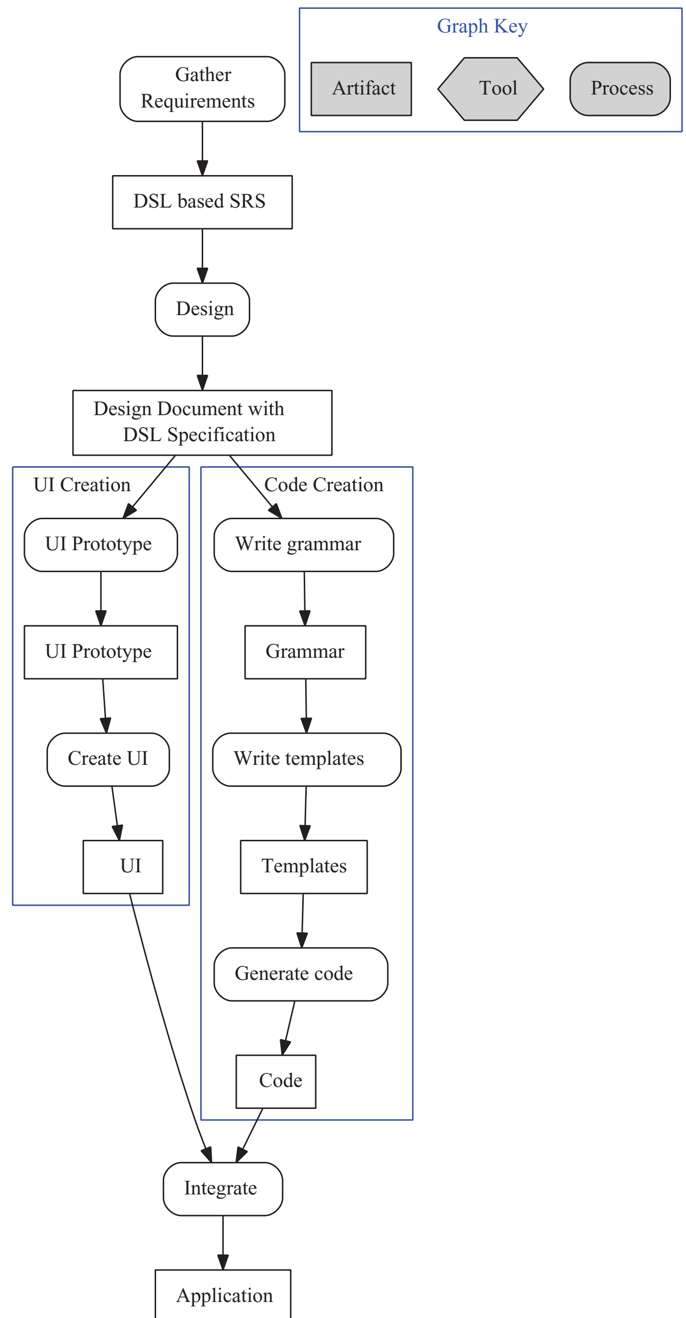

Fig. 9. Process diagram.

with the same application in order to accommodate different environments such as an instance running on an individual workstation or one running in a web browser. The connection between the UI and application can be managed via a connector component which handles the details of routing messages to and fro, thereby allowing both UI and the application to be ignorant of their distance from each other.

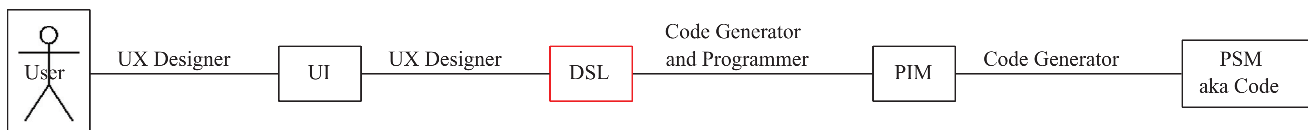

Fig. 8. Workflow with code generation.

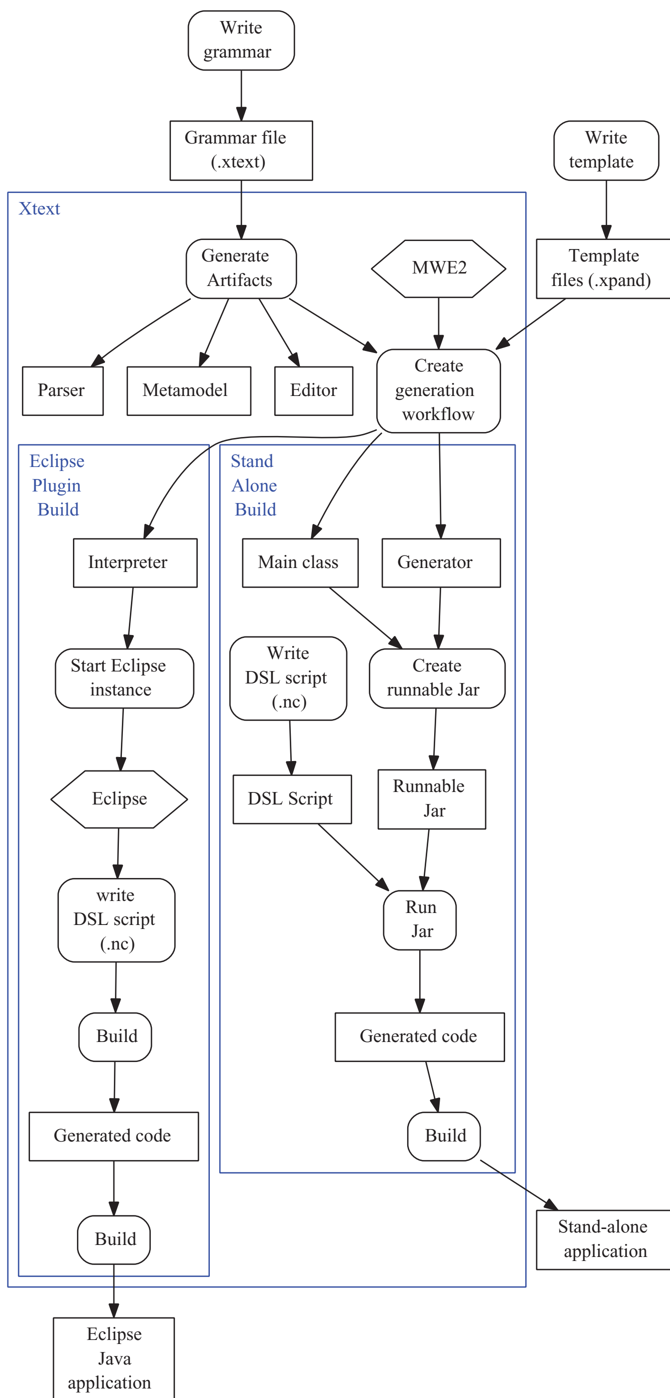

Fig. 10. Code creation details.

2.2.1. Code creation

The code creation process shown in Fig. 9 is a DSL-based code generation process that uses several development tools that we have selected. Fig. 10 presents a much more detailed code generation diagram highlighting our choice of using the Xtext tool. The first step is to take the DSL from the DSL specification and distill it into a grammar for Xtext. Once we have an understanding of the form of the grammar, we start a new Eclipse Xtext project and then enter the grammar. The creation of this grammar enables us to generate the parser, the metamodel, and an Eclipse editor tool via the 'Generate Xtext Artifacts' command. In the next stage, we create templates for the code generation; these templates are coded in Xtend. The Modeling Workflow

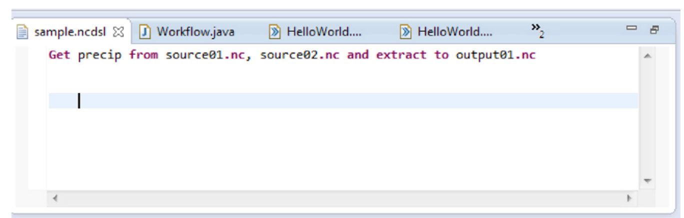

Fig. 11. A script for our DSL.

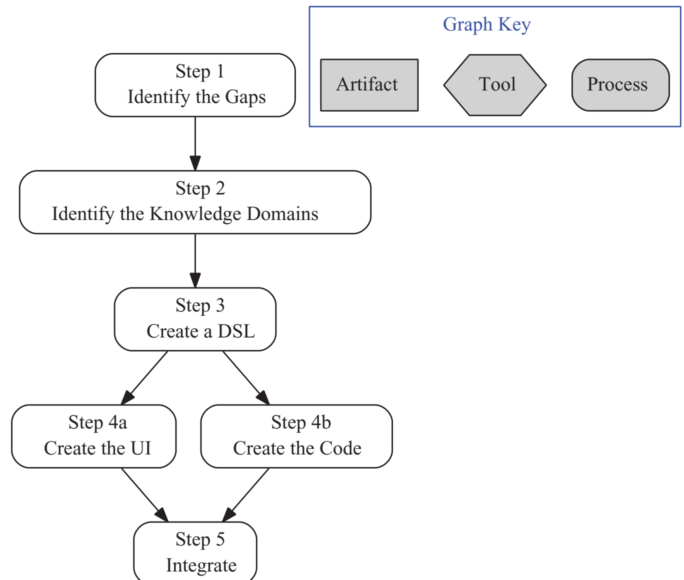

Fig. 12. A simplified workflow representation of our methodology.

Engine 2 (MWE2) tool then allows us to generate either a generator or an interpreter for our DSL. When we take the interpreter path, we start a new Eclipse instance that incorporates the editor tool created earlier and that tool provides syntax corrections while writing our DSL script (Fig. 11). After the DSL script has been created, we can then build that script which results in the creation of the generated code. Subsequently, the generated code is built to give us the final application. Here, we have only explained the Eclipse plugin branch of the code generation process, but if the readers desire to learn the details of the Stand Alone branch, they are referred to Bettini (2013).

2.2.2. UX creation

The creation of the UI will be largely performed by experienced UX designers. We closely followed the methods of IxD (Cooper et al., 2007) such as design ethnography and sketching the UI. Our ideal is for the UX professional to use tools such as GUI builders that will not require the understanding of programming; such as the MetaCase MetaEdit tool (MetaCase, 2013) or Meta-Gui-Builders (Luyten et al., 2008). The end result of this work is that the UI provided will be capable of generating scripts in the defined DSL and thereby communicate with the application code.

2.2.3. Steps of implementation

There are a number of well-defined steps which, if followed correctly, will allow the implementer to design software that conforms to our proposed approach (Fig. 12). Our approach does not replace traditional requirements analysis, and other design processes but instead enhances the design during use case construction, architecture design, and code production. Step 1 is to analyze the use cases and to determine what the actual gaps are that the software is being asked to address. Then, in Step 2, the implementer separates the tasks of the

use cases into a two columned table identifying if the particular task is in the User domain or in the Machine domain. With this table, the UX professional and the programmers can work out what information needs to be communicated between the machine and the user with the DSL (Step 3). Once the DSL has been specified, the UI and Code development can carry on independently with Steps 4a and 4b. Finally, in Step 5, the two artifacts from Step 4 are integrated with the formalized DSL.

3. Application

To illustrate the application of our methodology we chose to work on a problem which affected climate scientists—that of the subsetting of NETwork Common Data Form (NetCDF; Unidata, 2014c) files. Although the application was kept relatively short for simplicity, it is nevertheless intended to give the reader a comprehensive view of how the methodology works in practice. The current section is structured according to the flow of the methodology shown in Fig. 12.

3.1. Step 1: Identify the gaps

In order to learn how the user interacted with NetCDF files, the UX professional performed a number of ethnographic interviews (Rogers et al., 2011). With the information our UX professional collected, a Persona was created (Table 1) for reference and a use case to subset data from a NetCDF file was written (Table 2). In performing this use case, the scientists were accomplishing their goal of analyzing climate data for their research needs.

3.2. Step 2: Identify the knowledge domains

Our UX professional and programmer got together and discussed the steps of the subsetting task in order to categorize those steps as shown in Table 3 into their respective domains. This categorization was to illustrate our methodology with a simple example and there are many potential categorizations that could be chosen with this use case. With this identification of domains, we have separated the work of the team and can benefit from role specialization.

Table 1
Climate scientist persona.

Joe Greenfield	
Job:	Climate scientist
IQ:	High
Time spent:	Gather data
	Analyze data
	Write reports
NetCDF needs:	Explore data sets
	Grab data sets
	Put data into Matlab
	Analyze data

Table 2
Use cases for subsetting NetCDF files.

1.	Search for data
2.	Find data in NetCDF files
3.	Download NetCDF files
4.	Extract data from NetCDF
5.	Analyze data in Matlab

Table 3
Classifying steps into domains.

User knowledge	Machine knowledge
Search for data Find data in NetCDFs Download NetCDFs	
	Extract data from NetCDF
Analyze data in Matlab	

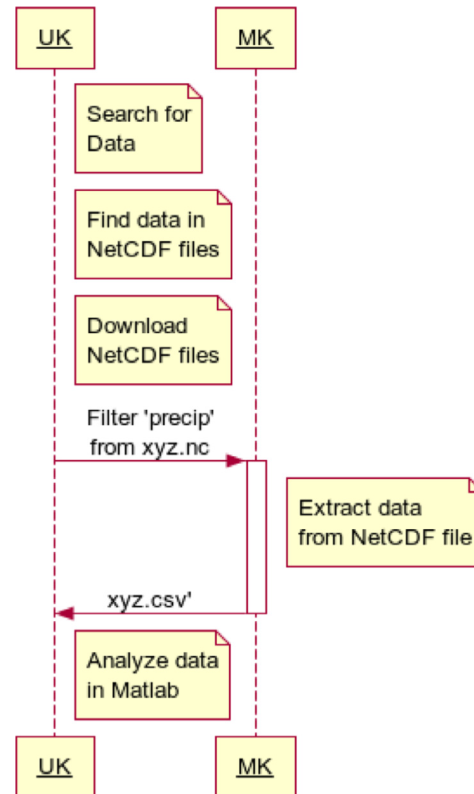

Fig. 13. Designing a DSL for the NetCDF file subsetting application: UK=user knowledge, MK=machine knowledge.

3.3. Step 3: Create a DSL for the domains to talk to one another

Our UX professional and programmer worked together to enhance Table 3 to that of Fig. 13 to show the messages to bridge the domains. This DSL will later be formalized, but for now it is only important to hash out the details of exactly what needs to be communicated.

3.4. Step 4a: Creating the UI

The UX professional created a prototype UI to get feedback from climate scientists. This first prototype was a drag and drop GUI that allows a climate scientist to create a workflow out of components that are familiar to them. The prototype was sketched, wireframed, and implemented. In order to be faithful to the methodology, we identified the MetaEdit+ application which could be used as a GUI builder without requiring the user to understand programming. The resulting MetaEdit+ prototype is shown in Fig. 14. Unfortunately, the UI had a number of problems: (1) it required user to have MetaEdit+

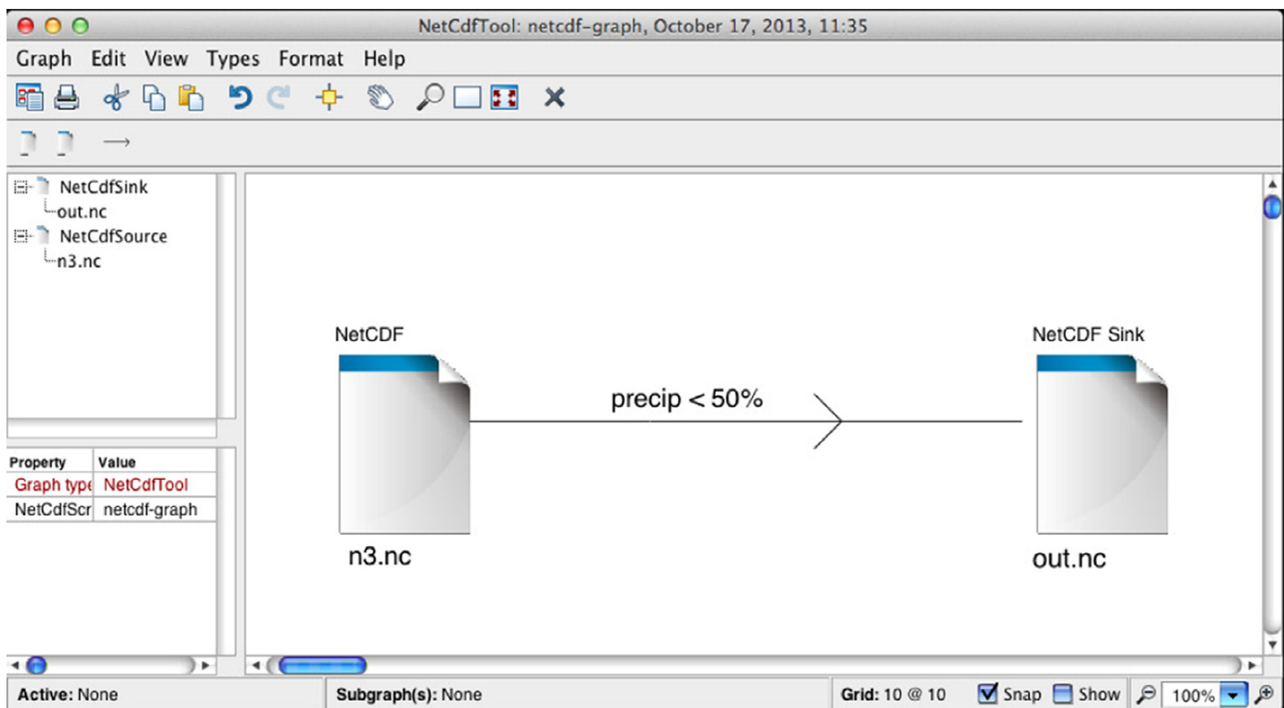

Fig. 14. A prototype in MetaEdit+.

installed, (2) it did not have a clear area for the tool icons for NetCDF files and the filters, and (3) it contained all MetaEdit+ controls rather than only the controls needed to create a NetCDF subsetting model. The limitations of this GUI builder motivated the UX professional to delegate the UI creation to a programmer.

A second prototype attempted to present the database contents as a node-link tree to the user; see Fig. 15. In searching a database, such as a library catalog, the user is presented with a textual hierarchy, but it is easy to lose yourself in the hierarchy. Our goal here was to be able to present a node-link tree that indicated how much data a particular node contained (Fig. 16) and allow the user to click on that node to show the sub-nodes. To specify a subset of the data, the user would choose a particular combination of nodes.

Though we feel that this interface had some promise, it had some real downsides regarding our methodology. First, the creation of this browser required some significant programming. We were using Sparx Enterprise Architect (Sparx Systems, 2014) to create the code and we used the D3 JavaScript Visualization package (Bostock et al., 2011). A second downside is that we ran into difficulties in how to display node contents and in getting information passed between the server and node-link tree in an efficient manner. Eventually after spending a significant time on this idea, we abandoned the effort.

A usability study informed us that a non-graphical approach is more effective than a graphical one for a climate science Search User Interfaces (SUI) in some cases. Three UI sketches were created: a graphical SUI, a text-oriented one, and a natural language output (Cooper et al., 2007) one. When tested with users and analyzed with GOMS (Card et al., 1983), the natural language output version was the most popular and efficient (Fig. 17).

At this point the UX professional began to test the UI with users to begin the iterative process of evolving the UI. Our testing for the application showed a marked improvement over earlier SUI designs. The eventual prototype consisted of a local desktop application which allowed the user to drag and drop NetCDF files to it. Once a file was dropped, the file would appear in a file list (Fig. 18). If the user chose the file in the file list, appropriate details and search terms would appear (Fig. 19).

3.5. Step 4b: Creating the code

The DSL specified in Table 3 was formalized for all messages that must be passed between the UI and Code components. Fig. 20 shows a couple of these formalized messages.

Our programmer used Xtext on the Eclipse platform with the Ecore architecture. The complex process of creating the textual DSL is shown in Fig. 10 and further details are provided in Bettini (2013). Using Xtext, we defined a grammar for our textual scripts (Fig. 21), wrote the program in Xpand and Java, and incorporated that code into our Xtext project as templates. We then generated the parser code (see Table 6) and tested scripts by feeding them into our parser and generating Java code.

3.6. Step 5: Integration

Our DSL served as the glue to connect the UI and Code components and this worked well. The actual climate scientist user would enter data into our SUI. Any user action that required information from the Code component would create a DSL script and query the Code component. For example, when the user presses the "Execute" button, the UI creates a script (Fig. 22) which is then fed into our parser (Fig. 23), which in turn places the parsed information into our code templates and then generates the Java source code for the specified workflow (Fig. 24). These files are then compiled into an executable program which is run to execute the workflow.

4. Related work

To place our work in the landscape of related software development methods, we surveyed prior work in the area of UI separation based architectures and UI creation techniques. Our approach was compared to related efforts in regards to user interaction and UI creation.

4.1. UI architectural patterns

Although there are many UI architectural patterns, here we address only those that isolate usability concerns to a UI component of

Fig. 15. A graph of the data in the database.

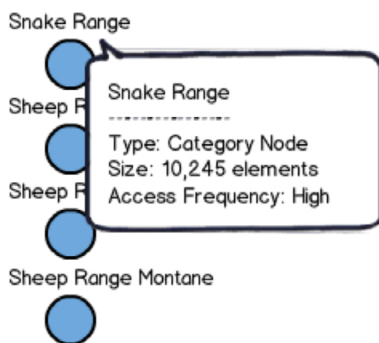

Fig. 16. Details of a particular node.

the software architecture. Though the reader may be familiar with patterns such as model-view-controller (MVC), our view is that many of these patterns primarily address non-usability concerns and therefore we do not include them here.

GUI wrappers have been used to improve the usability of text-based console UIs. Recently, Microsoft has registered a patent for using Windows Powershell commandlets for a UI (Pintos et al., 2009). An essential source of this pattern is that the UI code is conceptually different than the machine code and that inspires the developers to separate these parts. While there was a great debate regarding text versus graphical interfaces, we believe that the GUI reduces the cognitive load of a user and therefore improves usability for user interfaces.

Though GUI wrappers may actually have much in common with our architecture, it is still quite different. Our architecture requires the use of a DSL during the requirements and design stages of application development. However, a GUI wrapper is more of an ad hoc addition to a console based program. While one could introduce a GUI wrapper consideration in the requirements and design stages, the wrapper

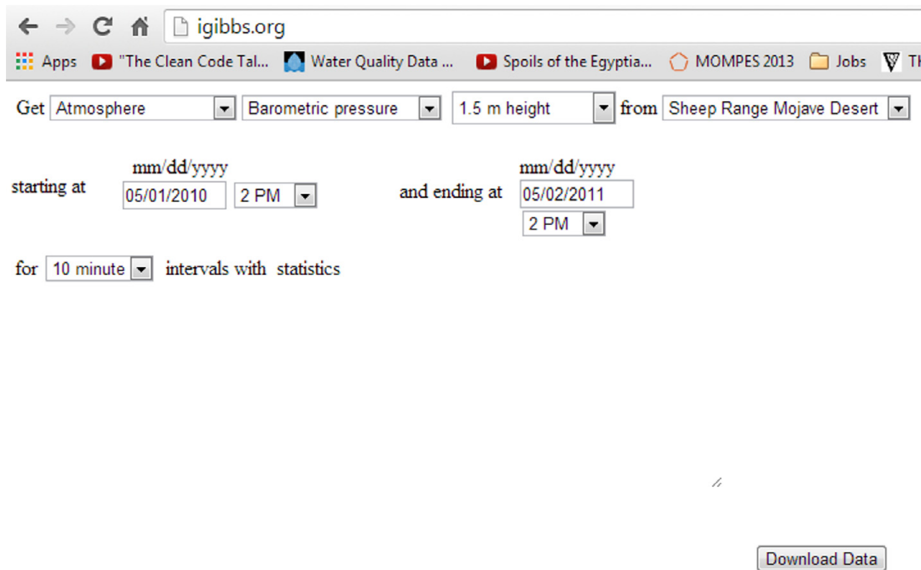

Fig. 17. Natural language search UI.

Fig. 18. Desktop NetCDF subsetting application after a NetCDF file was added.

pattern does not indicate any preference and therefore leaves implementers to their own devices, whereas our associated methodology provides a guide to developers as well as an architecture.

GUI builders offer a simple graphical building block interface for GUI construction to ease the difficulties of creating such GUIs. Though there are many of these builders in existence, there do not seem to be simple GUI builders where programming is not needed in order to create the final GUI. And though some GUI builders do pay attention to usability, such as Microsoft Expression Web's validator tool, usability is not the focus of any builder we are aware of.

Our focus is aligned with the overall concept of a GUI builder, but current GUI builders do not free the user from needing to know quite

Fig. 19. Desktop NetCDF subsetting application showing details of the NetCDF file that has been chosen.

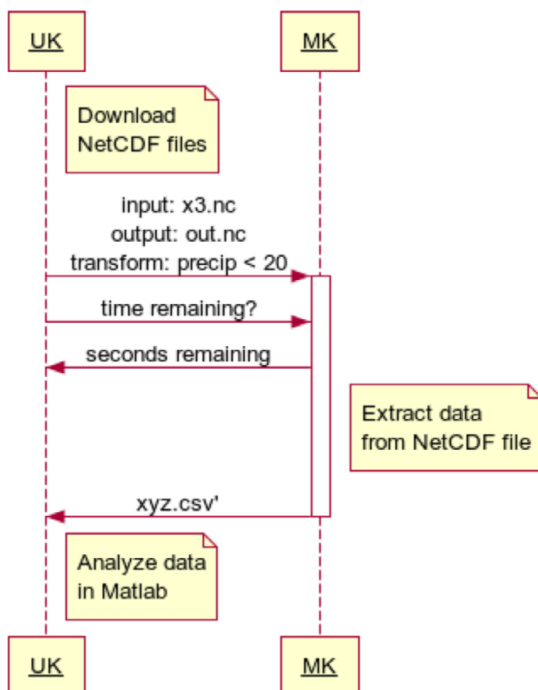

Fig. 20. The formalized DSL: UK=user knowledge, MK=machine knowledge.

a lot of programming. Our approach attempts to remove the need to understand programming from the construction of the entire GUI. And, although we have not found an optimal GUI builder to allow us to implement our method in this way, our GUI builder would be graphical and require only knowledge of UX and the DSL interface for that particular application.

Usability-supporting architecture patterns (USAPs) arose as some researchers have took an interest in the limitations imposed on the UI from the supporting architecture. Some usability concerns such as a Cancel feature generally need a considerable amount of support external to the UI module and therefore are difficult to add later in a project lifecycle. Ways to identify these architecturally sensitive UI features of a software product during the requirements phase have been developed (Juristo et al., 2007; Rafla et al., 2007). Deriving architecture patterns from the requirements has also been described in the literature (Bass and John, 2003; John et al., 2009). USAPs have been created to provide insight for designers in order to deal with incorporating usability into the software architecture (Bass et al., 2004); a pattern-language has also been created with these USAPs (John et al., 2009). Research regarding the effectiveness of USAPs has found them to be effective (Golden et al., 2005) and addressing usability concerns during the architectural and design stages of product lines has also verified the practice (Stoll et al., 2009).

The USAPs allow designers to recognize the dependencies between the UI and the code. However, they do point to a methodology as our approach does. Nevertheless, our approach is subject to these

```
NcDsl.xtext    NcDslGenerator.xtend
grammar org.xtext.demo.mydsl.NcDsl with org.eclipse.xtext.common.

generate ncDsl "http://www.xtext.org/demo/mydsl/NcDsl"

NcDsl:
    workflow+=Workflow*;

Workflow: 'Workflow' name=ID
    'input: ' (source+=Source';')
    'output: ' (sink+=Sink';')
    'transform: ' (transition+=Transition';')
;

Transition returns Transition:
    (Extraction)// | Transformation)
;
```

Fig. 21. Xtext grammar.

```
Workflow w01
        input: /cygdrive/c/final−
            draft/x3.nc;
        output: /cygdrive/c/final−
            draft/out.nc;
        transform: precip > 20;
```

Fig. 22. The original.ncdsl script for our DSL.

USAPs as well and the USAPs may be used in conjunction with our methodology.

4.2. User Interface Markup Languages

The declarative creation of a GUI has gotten a fair amount of attention, but the focus seems to be on separation of concerns rather than usability (Goderis and Deridder, 2004; Goderis and Lab, 2007). Flexibility for multiplatform UIs (Bendsen, 2004; Falb et al., 2009; Fatolahi et al., 2011; Helms and Abrams, 2008; Nebeling et al., 2012; Nichols and Myers, 2009), consolidation of multiple UI markup languages, and customizability (Jones et al., 2007) are common directions in this area. UI markup language work that considers multiplatform flexibility conflicts with the expressivity of the UI that can be gained by using the platform's native framework. Since most User Interface Markup Languages (UIMLs) are cryptic, they are generally accompanied by a GUI builder to make them usable. In this sense, they form a meta-UI language which can be mapped to different UI platforms.

In contrast to a UI markup language, we are focusing on a DSL to create an interface based on role specialization. Another important difference is that while many UIMLs are motivated by generalizations, our DSL is specific to each application and does not attempt to be a generalized solution. Our method can support the use of a UI markup language, but is by no means required to use them. As UI markup languages have certain limitations, the decision to use them is not prescribed or prohibited by our methodology and they are expected to be isolated in the UI component of our architecture (and therefore will not affect our Code component).

One UIML which is comparable to our DSL is the Game Maker Language (GML), which has been created to enable people to make computer games more easily. The GameMaker:Studio™ environment (YoYo Games, 2014) integrates with the language to facilitate the creation of the scripts. Users are allowed to use menus, dialogs, and drag and drop commands to set up a large portion of their games and this can be enhanced with textual scripts written by the user. Though the term scripts is used, these scripts are as complicated as a programming language and serve to allow GameMaker:Studio™ to export the game to iOS, Android, OS X, PlayStation®, or Windows. The purveyor of GML claims that one can create games 80% faster with their tool and scripting language.

While GML supports creating games, it appears that the user still needs to know quite a lot about programming, such as objects, events, sprites, drawing depth, and basic code. While the supporting GameMaker:Studio™ does help, it does not transform the writing of GML into a simple activity. Additionally, the user is not expected to

```
/cygdrive/c/final-draft $ ls
ncdsl-compiler.jar  original.ncdsl
/cygdrive/c/final-draft $ java -jar ncdsl-compiler.jar original.ncdsl
Code generation finished.
/cygdrive/c/final-draft $ ls
ncdsl-compiler.jar  original.ncdsl  src-gen
/cygdrive/c/final-draft $ |
```

Fig. 23. Command to create generated java files from original.ncdsl.

```
/cygdrive/c/final-draft/src-gen $ ls
NetCdf.java  NetCdf_Sink.java  Predicate.java  Workflow.java
/cygdrive/c/final-draft/src-gen $ |
```

Fig. 24. Generated java files.

have any UX knowledge and therefore introduces potential usability problems into the game created. Our approach instead attempts to remove the need to know programming from the UX professional—and this goes beyond just attempting to increase the productivity of programmers by offering them a higher level of abstraction to program in.

Another comparable UIML is the Linden Scripting Language (LSL), which allows users of the virtual 3D world game Second Life® to write scripts to control their games. This scripting language resembles the C programming language syntax and uses event-based programming. Objects in the virtual world can be created and imbued with behavior with this language. The game interface is used to program and therefore offers some similar capabilities as an IDE in working with LSL. The Second Life® game was built by the users through the use of LSL.

Although LSL does offer a programming interface to users, that interface still requires knowledge of programming as evidenced by the C programming language syntax. It seems to us that the market for LSL could be greatly expanded by simplifying this interface. In contrast, we are not offering users a way to program, but are seeking instead to enable a UX professional to work independently of code creation in order to improve the productivity of the software development process. Although UX professionals may want to create a UX for the user to be able to program, they generally are not concerned with it, and therefore our methodology does not focus itself on end-user programming.

For contrast to the previous UIMLs of GML and LSL, we can also look at a multi-platform UIML. The Interaction Flow Modeling Language (IFML; Rossi, 2013) is an Object Management Group (OMG) standard to express a GUI and to interact with a supporting back-end application. This standard was based on the long-standing Web Modeling Language (WebML; Ceri et al., 2009). The primary benefits offered by IFML are the ability to define a GUI with a graphical DSL and thereby allowing for a Unified Modeling Language (UML)-Profile type of solution to the GUI. The standard also may permit different GUI builders to interoperate. While humans could write IFML in textual form, it is expected that a UML IDE, such as WebRatio or Eclipse, will provide the user with a way to create the interface.

Compared to our proposed method, IFML is limited to GUIs, while our method is applicable to more general UI such as speech or touch. Additionally, our DSL method does not require the user to understand and work with the graphical IFML tool, but instead is able to use the more general available DSL tools. IFML requires the user to understand the language and the background object oriented design (OOD) paradigm. If a typical GUI builder is used to create the UI and IFML is generated in the background, then this solution is similar to our approach with the exclusion of easy understanding by laypersons of what the interface between the UI and Code components is doing.

Plastic user interfaces attempt to address the lack of collaboration between the human–computer interface and software engineering (SE) by using MDE to attain an ability to easily modify the UI. UI plasticity is defined as 'the capacity of user interfaces to adapt to the context of use while preserving usability' (Thevenin and Coutaz, 1999). Three context models are specified as the user model, the environmental model, and the platform model. The usability of the UI is defined differently for different contexts. If a UI is intended to run in a set of different contexts (C) then we can define a set of values (V) to represent our usability. The set V can then be mapped to the set C of contexts in unique ways to preserve usability across the contexts. With this formal model of the UI with usability values V and contexts C, further conclusions may be reached (Sears and Jacko, 2007)

Now, with a solid base, MDE is brought in and a meta-modeling language is created (Calvary et al., 2001, 2002). This method of combining MDE to generate UIs while preserving usability defines plastic user interfaces (Coutaz, 2010).

In contrast to our method, plastic user interfaces require a fair amount of specialized and sophisticated knowledge of software modeling. Additionally, they do not directly address usability and one wonders how easy it would be to later modify the model to address usability concerns. Instead, our method focuses on more popular and commonly known methods of programming and only the DSL requires more advanced skills. Also, by focusing on the UX professional creating multiple UIs for different platforms, we can meet multiplatform requirements but also allow our UIs to be easily modified based on feedback from users.

Another MDE methodology uses the Unified Communication Platform (UCP; Popp and Raneburger, 2011; Raneburger et al., 2014). Domain experts create a graphical discourse model (Falb et al., 2006) based on the communication between the user and the computer. The discourse model is defined by domain experts, the GUI prototype can then be automatically generated, which is followed by the incremental and iterative development of that prototype. Once the prototype is finalized, development on the back end code and refining of the GUI by hand can be done.

This UCP methodology has much in common with our method. The graphical discourse model serves a similar purpose as describing a conversation in our DSL, but the DSL conversation is between the UI and the back-end whereas in the discourse model it is between a computer and a user. UCP also uses code generation to create a GUI during a prototype refinement stage. Our method only employs code generation for the back-end code and is more general in that it applies to all UIs and not just a GUI. Although both methodologies allow for parallel development of the UI and back end code after an initial stage, the UCP focuses on this initial stage while our method focuses on both the initial stage and the later parallel development. All in all, the UCP is an effort to speed up development and to raise the quality of products with MDE while our method focuses on separating the roles of programmer and UX professional to enhance these specializations and thereby resulting in better products.

The idea of Intentional Programming began in the 1990s and has constantly progressed since. This method departs from traditional Object Oriented Programming (OOP) by modifying the editor and compiler, and by introducing the concept of intentions. With Intentional Programming, a class is created, but in addition to the class itself, a viewer for the class must be created along with a parser, and a version control component. By including with the class a bunch of functionality that is now concentrated in tools, we can create a higher level of programming IDE. The advantages are that different representations can easily be mixed and matched to create programs with better comprehensibility. The Intentional Software company (Intentional Software, 2014) promotes the concept of intentional programming. Their method will allow one to have a domain expert program the software on a domain level while the programmer will write and generate the underlying code for the domain.

The overall approach of intentional programming is very much like our methodology. However, the focus here is on providing the domain experts with a means to code an application themselves. The domain representation is now the interface between the code and the domain language. In contrast, our approach focuses on the usability of the method by involving a UX professional to create the UI instead of a having a domain expert to code the application.

A number of applications have attempted to use a DSL to increase the flexibility of a traditional application. Some researchers used DSLs to create an elevator application and allow people to manipulate that application with a DSL (Wienands and Golm, 2009). As enterprise applications are typically composed of multiple independent units, other researchers have addressed this by creating a DSL for these independent units to communicate (Shtelma et al., 2009). Yet others have used DSLs to integrate multiple applications (Berger et al., 2010).

Though DSLs have been used in numerous applications, we did not find any indication of using them between the UI and Code sections of the architecture. And, although client server systems are somewhat

Table 4
Approach comparison: L=layman, P=programmer, DE=domain expert, GD=game designer, UXP=UX professional.

Test	Who is the user?	Can the user program?	Who creates the UI?
Architecture patterns			
GUI wrapper	L	No	P
GUI builder	L	No	P
USAP	L	No	P
UIMLs	P	Yes	P
GML	GD	Yes	GD
LSL	L	Yes	N/A
IFML	P	No	P
Plastic UIs	L	No	P
UCP	DE	No	P,UXP
Intentional Programming	DE	No	DE
Our method	L	No	UXP

Table 5
How the UI is created.

Test	Code	Drag-n-drop	Script	GUI builder	MDE
Architecture patterns					
GUI wrapper	X				
GUI builder	X	X			
USAP	X				
UIMLs					
GML	X			X	
LSL	X		X		
IFML				X	
Plastic UIs					X
UCP	X				X
Intentional Programming				X	
Our method				X	

similar, they are really concerned about the distance between the client and server rather than the usability or the creation of the UI. For examples, HyperText Markup Language (HTML) could be considered a DSL, but it is much more of a general DSL in comparison to our method which aims at creating an application-specific DSL. Our method differs from all related work we have surveyed because it focuses on making the UI creation feasible without needing to understand programming.

4.3. Comparison

Here we compare and contrast the various approaches identified in the previous section to provide an overall picture of how our methodology is positioned in relation to the listed methods. We decided to compare the roles of people involved with the related software development and use this comparison in order to explain the differences between the various approaches. However, we are not experts in any of the other approaches; hence our comparisons should be viewed from this perspective. Tables 4 and 5 show that our method is the only one to specify a UX Professional as the one to create the UI and this is the key distinguishing characteristic of our approach. The closest to our method in respect to who creates the GUI is the Game Maker Language which contains a simplified builder for video games. In fact, we see that the need for a new type of GUI builder which excludes the user from a need to understand code is somewhat represented by the Game Marker Language GUI and other game creation platforms of this ilk. So the UI creator and the GUI builder are closely related and the game creation platforms may provide a fertile source of ideas for how to create this new type of GUI builder. As it can be seen, our proposed method is more generic, and flexible, as it can involve using various types of GUI builders.

4.4. Discussion

The approach presented here provides the following benefits. Communication is improved between the UX designers and the programmers by focusing on the DSL as an intermediate language. Increased role specialization helps to increase the productivity of UX designers and programmers, which may improve the product. The method is (i) compatible with web applications by using the DSL as a protocol between UI client and the application server, (ii) accommodates the increasingly diverse UIs such as mobile, voice, tactile, etc. by allowing separate UIs to be created without affecting the application code, (iii) supports testing at the DSL level, and (iv) provides for simple tracing of requirements.

Some drawbacks exist in our proposed approach. First, it requires a UX professional and a programmer. In approaching the problem of platform diversity, the UI will typically not be generalizable such as USer Interface eXtended Markup Language (UsiXML), but would instead be a GUI builder that is simple enough for the UX professional to understand. It may be difficult to use this method if one is dependent upon some software framework such as dotNet, because the use of this method may affect some advantages of any particular software framework. Using the DSL with code generation may affect the ability to finely tune the code (i.e. for special speed or other considerations) or the UI for any specialized purpose beyond its general realization. The DSL now becomes an important artifact in the software application and therefore the DSL needs to be designed well and may require a language designer if things become complex. Certain cross cutting USAPs will still be a problem if they have not been considered during the design phase.

If a team decides to use this methodology they can expect the following impacts on the applications. The long term impact will be improved usability through the UX approach and increased programmer productivity through role specialization and clear communication of application requirements via the DSL. Overall, the team will accomplish UI flexibility through independence from a specific platform, and improved communication between designers and programmers. In the short term, the team may experience confusion in adapting to this methodology, especially if they are not familiar with UX, large-scale design up front methods, DSL creation, or code generation. Breaking from the current framework paradigm, supported by a majority of software producers today, could cause a certain discomfort, as it usually happens when departing from the well known (in our case, departing from the existing frameworks).

5. Evaluation

In this section, we evaluate our proposed methodology by assessing the results of applying the methodology, that is, by evaluating the "NetCDF application" created following our approach. The components of this evaluation include a usability study that informed our approach, a usability analysis, specific results of the code generation component of our approach, and a comparative analysis of tools that address the same case study as our NetCDF application.

5.1. Usability analysis

Two usability tests were performed in an iterative fashion. The first usability test, which we called an informative test, was an A/B comparison of a climate data search page to allow scientists to find and download data from a website. One page was a checkbox listing of parameters, while the other one included a slightly different parameter choice method along with a graphical display for location selections. Ten users were tested and the results were that the version A of the site did slightly better than B, but the statistical significance was inconclusive (Gibbs, 2013).

Table 6
Parser code generation.

Artifact	LOC
Written code	
Grammar definition	54
Templates	476
Total lines written	530
Generated code	
xtext.mydsl.ncDsl	282
xtext.mydsl.ncDsl.impl	2005
xtext.mydsl.ncDsl.util	363
xtext.mydsl.parser.antlr	35
xtext.mydsl.parser.antlr.internal	2238
xtext.mydsl.serializer	241
xtext.mydsl.services	467
xtext.mydsl.validation	12
total lines generated	5643

Table 7
Script code generation.

Artifact	Details	LOC
Written code		
dsl script	Fig. 21	10
Total lines written		10
Generated code		
NetCdf.java		122
NetCdf_Sink.java		99
Predicate.java		20
Workflow.java		11
Total lines generated		252

After searching the UX literature, we attempted to create a natural language output interface (Cooper et al., 2007). This interface was developed following our methodology and we tested it against version A data from the previous informative test. We discovered that the natural language output version was more satisfying to climate scientists than version A and that they executed tasks up to twice as fast on the natural language output SUI (Gibbs, 2013). The result of this test is that we chose a natural language SUI in our final application artifact. Overall, the test results provide positive indications of the benefits of applying our approach. These results also provide direction for others working with climate scientists to create SUIs and thereby have a broader impact than just for this application.

5.2. Code generation

We have two stages of code generation to address here. The first stage is in the Eclipse IDE with Xtext, which generates our parser generator for the second stage of code generation. This first stage requires us to define the grammar and write the associated code templates and after building we get a Java Jar file which accepts DSL scripts as input and generates Java code as output. Table 6 summarizes the lines of code (LOC) measurement which results in a code generated:written ratio of 10:1. The second stage of code generation involves using the Java parser generator Jar file to read scripts and create Java source code files (Table 7). The code generated:written ratio at this second stage was 25:1.

5.3. Comparative analysis

To give the reader an understanding of how our NetCDF application relates to other such tools, we present several tools that are currently available to help climate scientists work with NetCDF files and compare them with ours. In fact, there are a great number of tools which manipulate NetCDF files and not all of them are listed here as the time it would take to compare all of them would be quite significant, but we do provide a spectrum of the contemporary free tools. We have chosen tools which are close to ours in size and scope or they are tools that were mentioned by the climate scientists we have worked with.

Since we are not climate scientists, our evaluation of these tools is based on user concerns rather than the detailed abilities offered to climate scientists. Thus, this tool does not aim to compete with these various capabilities but instead it aims to provide a better interface for the user and to illustrate the effects of using our prescribed methodology. The limited comparison presented here serves that purpose and gives the reader a view of how our tools UX features compare with currently available tools.

A command line program, ncdump (Unidata, 2014a), allows users to extract data from a NetCDF file and export it to a network Common Data form Language (CDL) or NetCDF Markup Language (NcML) format. A second program, ncgen (Unidata, 2014b), can be used to create a NetCDF file from CDL. In using ncdump, we first want to know what variables our sample.nc file contains. Next, knowing what variables are there, we can filter for only the ‘precip’ variable and put it into a CDL file. In the final step, we create our sample-precip.nc from the CDL file.

MATLAB offers a function based extension to their language for NetCDF files (Mathworks, 2014). The user must be familiar with the MATLAB language. Users need to be able to compose a list of commands to get the information they want from the NetCDF file such as getting the variables available or getting specific information about a variable.

EverVIEW is an attempt by Joint Ecosystem Modeling Group (JEM) and the United States Geological Survey (USGS) to create a tool to work with NetCDF files (Roszell et al., 2014). The project has three goals: subset NetCDF files, view tabular data, and convert to Comma-Separated Values (csv) files. The project is currently in the Beta stage of development (Roszell et al., 2009; Visualizing NetCDF Files by Using the EverVIEW Data Viewer, 2013). The focus of the tool is to subset a NetCDF file with regards to time or geography (NetCDF Slice and Dice Tool, 2014).

The Ncview application (Pierce, 2014) runs on Unix systems and provides a graphical picture of the contents of a NetCDF file. Upon starting the program, the user is confronted with a command screen and asked to choose a variable. If the user chooses one of the variables, such as ‘precip’ then a plot of that variable is shown. Subsequently clicking on any point of the plot brings up a detailed 2D graph of the ‘precip’ variable.

Data Basin (2014) enables users to analyze and map data from NetCDF files. This is an online tool requiring an account to use the tool and special permission to be able to upload one’s own NetCDF files rather than work with the data that Data Basin provides (NetCDF Data in Data Basin, 2014). The tool is free and it does not require sophisticated computer knowledge of programming languages to use.

The process of using Data Basin requires the user to first upload a file, then they have to select from a number of settings. Results are then presented to the user.

The Interactive Data Language (IDL) was created to enable users to analyze data. It supports the use of many types of files used in the scientific fields such as Hierarchical Data Format 5 (HDF5), NetCDF-3, NetCDF-4, Common Data Format (CDF), Hierarchical Data Format-Earth Observing System (HDF-EOS), Hierarchical Data Format 4 (HDF4), GRidded Binary (GRIB), and others. The language is ready to access data in large files and can even access data from remote locations via the HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP). Programming knowledge and skill are required as the language looks much like a modern programming language.

5.4. Comparison

When comparing tools, we do so from a user perspective because our knowledge of how these tools were designed or coded is very

Table 8
Approach comparison: CS = climate scientist, KD = knowledge domain.

Test	User?	Command line KD?	Programming KD?	NetCDF KD?
ncdump and ncgen	CS	Yes	No	Yes
MATLAB	CS	Yes	Yes	Yes
EverView Slice and Dice	CS	No	No	Yes
Ncview	CS	No	No	Yes
Data Basin	CS	No	No	Yes
Interactive Data Language	CS	Yes	Yes	Yes
Our method	CS	No	No	Yes

limited. By comparing from a user's point of view we also hope to achieve a picture of the actual utility of these tools to a user rather than a plain listing of various features. Table 8 shows all of the tools and the presence or absence of specific user related features. We broke down the required information by the user into different mental loads that are pushed onto the user when using a tool. These mental loads are categorized according to knowledge domains of command line, programming, and NetCDF knowledge domains.

5.5. Discussion

We performed a benefit/drawback analysis of the application to help assess its current state of development. The benefits offered by this application are a search UI based on real usability studies, code generation aiding in productivity gains, a favorable comparative analysis, and a simple UI which does not require knowledge outside of the realm of climate science. The drawbacks are a complex build process regarding generation and the need to understand how to write grammars, dependency on Xtext and Eclipse, and the awareness of climate scientists that the tool exists.

Other options are available to users besides using our tool described here. There are many tools that can be used to subset NetCDF files and those in this comparison are a small sample. Some climate scientists may feel great independence and freedom in being able to program and therefore resist changing to a simpler application. If a particular organization is large enough, they may assign a NetCDF expert to do the work of subsetting NetCDF files and consequently there would not be as much of a productivity gain by the use of our application.

There are a few expected impacts of creating a final version of our tool to release to climate scientists. First, we expect that some climate scientists will find the tool helpful. The use of the tool may at first be limited due to awareness of its existence in the world of NetCDF tools. In order for the tool to slowly increase a user base, it will need to be updated with advances in NetCDF files, new operating system platforms, error fixes, and change requests.

6. Conclusions and future work

We have presented a new methodology, described its specific concepts and steps, and provided a detailed example of applying the methodology. We have also evaluated the methodology through comparison with related work, usability studies, and analysis of its results. Our assertion (Section 2.1.4) has been that a separation-based UI architecture can enhance the process of software development. A key to our proposed approach is the use of a DSL to bridge the gap between UI design and writing code. Much of the research, development, and testing conducted has been promising regarding the validity of our assertion, but inherently more work remains to be done to fully prove our methodology.

We have shown that there is a clear communication gap between a UX professional and programmer working on a software development project. Our architecture addresses this communication gap and the associated methodology illustrates how to use this architecture in a development process. With regards to problem (4) and its resulting effect of problem (1), we did not find an adequate GUI builder that does not require programming knowledge and therefore were not able to address these problems in this work. The use of a DSL to separate the UI and Code components provides a significant tool to reduce the communication gap and thereby addresses problem (2) of the current status quo. Secondly, the DSL also provides separation of concerns between the UI and Code components and therefore addresses problem (3) of the current status quo.

We expect that our approach contributes to reducing the gulfs of execution and evaluation as described by Norman (2013). Specifically, they can be reduced because the UX designer knows how the software is expected to behave (by the user), and the programmer knows how to implement the software to make that happen.

In considering future work, we must discuss both work on the architecture and the developer's experience. In the case of the UI component, we did not find a suitably flexible GUI builder that would not require the need to understand code. Having a suitable UI builder could really make our approach easier to implement. Though we have shown how to create the DSL, we still want to find a generic solution for passing the DSL messages back and forth from the UI and Code components. And, with regards to the Code component, we could work with different code generation techniques to find a best of breed. In addition, more testing may further specialize the architecture, with three major roles rather than just two: UX professional, DSL designer, and code programmer. And although our methodology looks promising in the climate science environment, we still need to test to determine if it is effective in other areas.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under grants no. EPS-0814372 and IIA-1301726. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Bass, L., John, B.E., 2003. Linking usability to software architecture patterns through general scenarios. J. Syst. Softw. 66, 187–197. doi:10.1016/S0164-1212(02)00076-6.
Bass, L., John, B.E., Juristo, N., Sanchez-Segura, M.I., 2004. Usability-supporting architectural patterns. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 716–717.
Bendsen, P., 2004. Model-driven business UI based on maps. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of data. ACM, New York, NY, USA, pp. 887–891. doi:10.1145/1007568.1007678.
Berger, S., Grossmann, G., Stumptner, M., Schrefl, M., 2010. Metamodel-based information integration at industrial scale. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II. Springer-Verlag, Berlin/Heidelberg, pp. 153–167.
Bettini, L., 2013. Implementing Domain-Specific Languages with Xtext and Xtend, first ed. Packt Publishing, Birmingham, UK.
Bostock, M., Ogievetsky, V., Heer, J., 2011. D3 data-driven documents. IEEE Trans. Visual. Comput. Graph. 17, 2301–2309. doi:10.1109/TVCG.2011.185.

Brooks Jr., F.P., 1995. The Mythical Man-month (Anniversary Ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Burke, R.E., Fitzgerald, T., 2003. The aptitudes of software engineers: Technical Report 1. Johnson O'Connor Research Foundation, Inc.
Calvary, G., Coutaz, J., Thevenin, D., 2001. A unifying reference framework for the development of plastic user interfaces. In: Little, M., Nigay, L. (Eds.), Engineering for Human-Computer Interaction. Lecture Notes in Computer Science, vol. 2254, Springer Berlin/Heidelberg, pp. 173–192. doi:10.1007/3-540-45348-2_17.
Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., Vanderdonckt, J., 2002. Plasticity of user iInterfaces: A revised reference framework. In: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design. INFOREC Publishing House Bucharest, pp. 127–134.
Card, S.K., Newell, A., Moran, T.P., 1983. The psychology of human-computer interaction. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.
Ceri, S., Brambilla, M., Fraternali, P., 2009. The history of WebML lessons learned from 10 years of model-driven development of web applications. Conceptual Modeling: Foundations and Application. Springer-Verlag, Berlin/Heidelberg, pp. 273–292. doi:10.1007/978-3-642-02463-4_15.
Condon, C., Schroeder, D., 2005. Statistical bulletin 2005-14, occupational plots: Displayed by occupation. Technical Report. Johnson O'Connor Research Foundation, Inc.
Cooper, A., 1999. The Inmates are Running the Asylum. Macmillan Publishing Co., Inc., Indianapolis, IN, USA.
Cooper, A., Reimann, R., Cronin, D., 2007. About Face 3: The Essentials of Interaction Design. John Wiley & Sons, Inc., New York, NY, USA.
Coutaz, J., 2010. User interface plasticity: model driven engineering to the limit! In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems. ACM, New York, NY, USA, pp. 1–8. doi:10.1145/1822018.1822019.
Data Basin, 2014. http://databasin.org/ (accessed April 2014).
Falb, J., Kaindl, H., Horacek, H., Bogdan, C., Popp, R., Arnautovic, E., 2006. A discourse model for interaction design based on theories of human communication. In: CHI'06 Extended Abstracts on Human Factors in Computing Systems. ACM, New York, NY, USA, pp. 754–759. doi:10.1145/1125451.1125602.
Falb, J., Popp, R., Rock, T., Jelinek, H., Arnautovic, E., Kaindl, H., 2009. Fully automatic generation of web user interfaces for multiple devices from a high-level model based on communicative acts. Int. J. Web Eng. Technol. 5, 135–161. doi:10.1504/IJWET.2009.028618.
Fatolahi, A., Somé, S., Lethbridge, T.C., 2011. Model-driven web development for multiple platforms. J. Web Eng. 10, 109–152.
Gause, D.C., Weinberg, G.M., 1990. Are Your Lights On? How to Figure Out What the Problem REALLY Is. Dorset House Publishing, New York, NY.
Gibbs, I., 2013. A Separation-Based UI Architecture with a DSL for Role Specialization (Ph.D. thesis). University of Nevada, Reno.
Goderis, S., Deridder, D., 2004. A Declarative DSL Approach to UI Specification—Making UI's Programming Language Independent. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.7041 (accessed January 2014).
Goderis, S., Deridder, D., Van Paesschen, E., D'Hondt, T., 2007. DEUCE: A Declarative Framework for Extricating User Interface Concerns. J. Obj. Technol. 6, 87–104. doi:10.5381/jot.2007.6.9.a5.
Golden, E., John, B.E., Bass, L., 2005. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. In: Proceedings of the 27th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 460–469. doi:10.1145/1062455.1062538.
Helms, J., Abrams, M., 2008. Retrospective on UI description languages, based on eight years'; experience with the user interface markup language (UIML). Int. J. Web Eng. Technol. 4, 138–162. doi:10.1504/IJWET.2008.018095.
Hunt, A., Thomas, D., 1999. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Intentional Software, 2014. http://www.intentsoft.com/ (accessed April 2014).
John, B.E., Bass, L., Golden, E., Stoll, P., 2009. A responsibility-based pattern language for usability-supporting architectural patterns. In: Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems. ACM, New York, NY, USA, pp. 3–12. doi:10.1145/1570433.1570437.
Johnson, J. (Ed.), 2000. GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
Jones, C., Blanchette, C., Brooke, M., Harris, J., Jones, M., Schildhauer, M., 2007. A Metadata-Driven Framework for Generating Field Data Entry Interfaces in Ecology. Ecological Informatics 2, 270–278. Meta-information systems and ontologies. A Special Feature from the 5th International Conference on Ecological Informatics ISEI5, Santa Barbara, CA, December 4–7, 2006 Novel Concepts of Ecological Data Management S.I. http://dx.doi.org/10.1016/j.ecoinf.2007.06.005.
Juristo, N., Moreno, A., Sanchez-Segura, M.I., 2007. Guidelines for eliciting usability functionalities. IEEE Trans. Softw. Eng. 33, 744–758. doi:10.1109/TSE.2007.70741.
Luyten, K., Meskens, J., Vermeulen, J., Coninx, K., 2008. Meta-Gui-builders: generating domain-specific interface builders for multi-device user interface creation. In: CHI'08 Extended Abstracts on Human Factors in Computing Systems. ACM, New York, NY, USA, pp. 3189–3194. doi:10.1145/1358628.1358829.
Lynch, A., 2014. Communication model explained and basic concepts related to the model. http://www.comprofessor.com/2009/10/i.html (accessed January 2014).
Mathworks, 2014. MATLAB NetCDF capabilities. http://www.mathworks.com/help/matlab/ref/netcdf.html (accessed April 2014).
MetaCase, 2013. MetaEdit+Modeler. http://www.metacase.com/mep/ (accessed March 2013).
Nebeling, M., Grossniklaus, M., Leone, S., Norrie, M.C., 2012. XCML: providing context-aware language extensions for the specification of multi-device web applications. World Wide Web 15, 447–481. doi:10.1007/s11280-011-0152-2.
NetCDF Data in Data Basin [Video file], 2014. http://www.youtube.com/watch?v=L3ydfi5BqEk (accessed April 2014).
NetCDF Slice and Dice Tool, 2014. http://www.jem.gov/Modeling/SliceAndDice (accessed April 2014).
Nichols, J., Myers, B.A., 2009. Creating a lightweight user interface description language: an overview and analysis of the personal universal controller project. ACM Trans. Comput.-Hum. Interact. 16, 17:1–17:37. doi:10.1145/1614390.1614392.
Norman, D.A., 2013. The Design of Everyday Things, revised and expanded edition. Basic Books, New York, NY, USA.
Pierce, D.W., 2014. Ncview: A NetCDF visual browser. http://meteora.ucsd.edu/~pierce/ncview_home_page.html (accessed April 2014).
Pintos, A.F., Sharma, V., Jalobeanu, M.R., Feliberti, V., Clark, B., 2009. Constructing User Interfaces on Top of Cmdlets. U.S. Patent 7,581,190, filed 2006.
Popp, R., Raneburger, D., 2011. A high-level agent interaction protocol based on a communication ontology. In: Huemer, C., Setzer, T. (Eds.), E-Commerce and Web Technologies, Lecture Notes in Business Information Processing, vol. 85. Springer, pp. 233–245. doi:10.1007/978-3-642-23014-1_20.
Rafla, T., Robillard, P.N., Desmarais, M., 2007. A method to elicit architecturally sensitive usability requirements: Its integration into a software development process. Softw. Qual. Control 15, 117–133. doi:10.1007/s11219-006-9009-9.
Raneburger, D., Kaindl, H., Popp, R., Šajatović, V., Armbruster, A., 2014. A process for facilitating interaction design through automated GUI generation. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1324–1330. doi:10.1145/2554850.2555053
Rogers, Y., 2012. HCI Theory: Classical, Modern, and Contemporary. Morgan & Claypool Publishers.
Rogers, Y., Sharp, H., Preece, J., 2011. Interaction Design: Beyond Human–Computer Interaction, third ed. Wiley Publishing.
Rossi, G., 2013. Web modeling languages strike back. IEEE Internet Comput. 17, 4–6. doi:10.1109/MIC.2013.78
Roszell, D., Conzelmann, C., Chimmula, S., Chandrasekaran, A., Hunnicut, C., 2009. Users' Manual and Installation Guide for the EverVIEW Slice and Dice Tool, open-file report 2009-1177 edition. U.S. Geological Survey. Denver Federal Center, Denver, CO.
Roszell, D., Conzelmann, C., Chimmula, S., Chandrasekaran, A., Hunnicut, C., 2014. Users' Manual and Istallation Guide for the EverVIEW Slice and Dice Tool (Version 1.0 Beta). U.S. Geological Survey, Denver Federal Center, Denver, CO. http://pubs.usgs.gov/of/2009/1177/pdf/OF09-1177.pdf (accessed April 2014).
Sears, A., Jacko, J.A., 2007. The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications (Human Factors and Ergonomics Series). L. Erlbaum Associates Inc., Hillsdale, NJ, USA.
Shtelma, M., Cartsburg, M., Milanovic, N., 2009. Executable domain specific language for message-based system integration. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin/Heidelberg, pp. 622–626. doi:10.1007/978-3-642-04425-0_48
Sommerville, I., 2010. Software Engineering, 9th ed. Addison-Wesley Publishing Company, USA.
Sparx Systems, 2014. Sparx Enterprise Architect. http://www.sparxsystems.com/products/ea/index.html (accessed September 2014).
Stoll, P., Bass, L., Golden, E., John, B.E., 2009. Supporting usability in product line architectures. In: Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, Pittsburgh, PA, USA, pp. 241–248.
Thevenin, D., Coutaz, J., 1999. Plasticity of user interfaces: Framework and research agenda. In: Sasse, M.A., Johnson, C. (Eds.), Proceedings of INTERACT 99—IFIP TC13 Seventh International Conference on Human-Computer Interaction. IOS Press, Lansdale, PA. doi:10.1007/978-3-540-85992-5_14
Unidata, 2014a. ncdump. UCAR Community Programs and the University for Atmospheric Research. https://www.unidata.ucar.edu/software/netcdf/docs/netcdf/ncdump.html (accessed April 2014).
Unidata, 2014b. ncgen. UCAR Community Programs and the University for Atmospheric Research. http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/ncgen.html (April 2014).
Unidata, 2014c. Network Common Data Form (NetCDF). http://www.unidata.ucar.edu/software/netcdf/ (accessed January 2014).
Visualizing NetCDF Files by Using the EverVIEW Data Viewer, 2013. http://pubs.usgs.gov/fs/2010/3046/pdf/FS10-3046.pdf (accessed August 2013).
Wienands, C., Golm, M., 2009. Anatomy of a visual domain-specific language project in an industrial context. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin/Heidelberg, pp. 453–467. doi:10.1007/978-3-642-04425-0_35
YoYo Games, 2014. GameMaker: Studio. https://www.yoyogames.com/studio (accessed April 2014).

Ivan Gibbs is an independent researcher and consultant. The University of Nevada, Reno awarded him a BS in Electrical Engineering and a BS in Engineering Physics. After a career in industry he returned to academia and was awarded an MS and recently a Ph.D. in Computer Science and Engineering from the University of Nevada, Reno. He has published several papers on software engineering, model driven engineering, and usability.

Sergiu Dascalu is a Professor in the Department of Computer Science and Engineering at the University of Nevada, Reno, USA, which he joined in 2002. In 1982 he received a Master's degree in Automatic Control and Computers from the Polytechnic University of Bucharest, Romania and in 2001 a Ph.D. in Computer Science from Dalhousie University, Halifax, NS, Canada. His main research interests are in the areas of software engineering and human–computer interaction. He has published over 140 peer-reviewed papers and has been involved in numerous projects funded by industrial companies as well as federal agencies such as NSF, NASA, and ONR.

Frederick C. Harris, Jr. is a Professor in the Department of Computer Science and Engineering and the Director of the High Performance Computation and Visualization Lab and the Brain Computation Lab at the University of Nevada, Reno, USA. He received his B.S. and M.S. degrees in Mathematics and Educational Administration from Bob Jones University in 1986 and 1988 respectively, and his M.S. and Ph.D. degrees in Computer Science from Clemson University in 1991 and 1994 respectively. He is a Senior Member of ACM and ISCA, and a member of IEEE. His research interests are in parallel computation, computational neuroscience, computer graphics and virtual reality.