University of Technology, Sydney
Faculty of Engineering and Information Technology

Derivation of a General Purpose Architecture for Automatic User Interface Generation

Submitted by:

Richard Kennard

B. Sc (Computer Science) Hons. (1st)

2011

Supervisor: John Leaney

Submitted for the degree of

DOCTORATE OF PHILOSOPHY

Certificate of Authorship/Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Richard Kennard

Acknowledgements

Completion of this thesis would not have been possible without a number of people whom I would like to acknowledge and thank for their support, advice and encouragement.

My supervisor, John Leaney, was instrumental in opening me up to the world of academic rigour. I would like to thank him for challenging me to think about epistemologies and research methodologies, interview techniques, reflection, VVT and GQM, and a host of other disciplines. His support and enthusiasm for my work never faltered. Most of all I would like to thank him for being able to readily understand my goal, and bridge the divide between the industrial world I knew and the research community I aspired to join.

I would like to thank those people who posted forum messages, blogged, tweeted, wrote magazine articles, and published papers regarding my work. These people are not named in this thesis, but I include excerpts from their feedback and their words are publicly searchable online. I would like to thank my interviewees and adoption study participants, most of whom must remain anonymous but who know who they are, for their time and feedback. Given the Action Research methodology of this thesis, their observations were critical to my progress. I would like to thank Dan Haywood of the Naked Objects team, who was always friendly and supportive, and very professional considering the sometimes conflicting nature of our projects. I would also like to thank Dan Allen, for his help shepherding my research within Red Hat.

Finally I would like to thank and apologise to my family who lived with my long absences throughout – both physical absences during the weeks, and mental absences during the weekends when I couldn't drag my head out of the work.

Table of Contents

Table of Figures

Glossary of Terms

API.....Application Programming Interface

AST.....Abstract Syntax Tree

BPM.....Business Process Modelling

CLR.....Common Language Runtime

CRUD.....Create, Retrieve, Update and Delete

DRY.....Don't Repeat Yourself

DSL.....Domain Specific Language

ERP.....Enterprise Resource Planning

GP.....General Practitioner

GQM.....Goals, Questions and Metrics

GUI.....Graphical User Interface

HFD.....Human Factors Designers

JAXB.....Java API for XML Binding

JPA.....Java Persistence Architecture

JSF.....Java Server Faces

JSP.....Java Server Pages

JVM.....Java Virtual Machine

MVC.....Model View Controller

NHS.....National Health System

OID.....Object Identifier

OOUI.....Object Oriented User Interface

ORM.....Object Relational Database Mapper

PDG.....Program Dependency Graph

SSOT.....Single Source of Truth

UI.....User Interface

VVT.....Validity, Verification and Testing

WYSIWYG.....What You See Is What You Get

Abstract

Many software projects spend a significant proportion of their time developing the User Interface (UI), therefore any degree of automation in this area has clear benefits. Research projects to date generally take one of three approaches: interactive graphical specification tools, model-based generation tools, or language-based tools. The first two have proven popular in industry but are labour intensive and error-prone. The third is more automated but has practical problems which have led to a lack of industry adoption.

This thesis set out to understand and address these limitations. It studied the issues of UI generation in practice using Action Research cycles guided by interviews, adoption studies, case studies and close collaboration with industry practitioners. It further applied the emerging field of software mining to address some of these issues. Software mining is used to collate multiple inspections of an application's artefacts into a detailed model, which can then be used to drive UI generation. Finally, this thesis explicitly defined bounds to the generation, such that it can usefully automate some parts of the UI development process without restricting the practitioner's freedom in other parts. It proposed UI generation as a way to augment manual UI construction rather than replace it.

To verify the research, this thesis built an Open Source project using successive generations of Iterative Development, and released and promoted it to organisations and practitioners. It tracked and validated the project's reception and adoption within the community, with an ultimate goal of mainstream industry acceptance. This goal was achieved on a number of levels, including when the project was recognised by Red Hat, an industry leader in enterprise middleware. Red Hat acknowledged the applicability and potential of the research within industry and integrated it into their next generation products.

1. Introduction

1.1. Objective

The objective of this thesis is to derive a general purpose architecture for automatic User Interface (UI) generation.

Let us unpack that sentence. By 'general purpose architecture' I mean: targeting a broad range of applications, from research projects to industry applications. I am placing particular focus on the latter because, as we shall see, the goal of automatic UI generation has been attempted multiple times by the research community (section 2.1.1). However such attempts have generally seen little industry adoption (section 2.1.3). Achieving a general purpose solution will require the novel application of emerging technologies (section 2.2).

By 'derive' I mean: to observe and reflect upon existing UI development practices, and codify and distil them into themes. This will require modern research techniques (section 3.2.1) and will again be focussed particularly on industry practices. Industry applications will be considered a key source of observations and a primary validation of reflections. Industry adoption will be regarded a key measure of success (sections 4.3.1.5, 5.3.1.1, 7.2 and 8.1.2).

By 'automatic UI generation' I mean: the machine-based generation of the same UI the practitioner would previously have constructed manually. This will remove the repetitive code that must currently be written and maintained by the practitioner. Repetitive code is that which can be inferred using existing sources within an application's architecture. For example the maximum length of a UI text box can be inferred from a database schema; the correct format for an e-mail address can be inferred from a validation subsystem; the available navigation buttons can be inferred from a business process flow engine. Such repeated values must be declared identically, and must be kept identical throughout an application's lifetime. If they diverge, for example if the UI allows text to be input that is longer that the database can store, the application is prone to error.

Finally, as a point of clarification, I should emphasise that although this thesis will talk much about UIs and the difficulties of building them, it is concerned purely with the mechanics of their construction. Issues such as the aesthetics of UI design or their usability are beyond the scope of this work. My research will not touch upon them other than to honour the flexibility of

existing UI toolkits by not restricting them for the sake of automation. I will make an explicit point of defining 'useful bounds' to the automation (section 4.1.1.4).

1.2. Significance

There are three significant features of this thesis that distinguish it from existing research in the field.

The first is the application of an emerging technology, namely software mining, to the problem of automatic UI generation. I believe the combination of these two, previously unrelated, fields has significant potential to address long-standing problems. My particular focus is on addressing the problem of repetition between the UI layer and the rest of the application by mining multiple, heterogeneous artefacts. But the impact of bringing together two unrelated fields may itself expose new possibilities.

The second novel feature of this thesis is its focus on industry applicability. I will employ a modern research technique, namely Action Research, to frame industrial practices within an academically rigorous context. Whilst other researchers have previously investigated automatic UI generation solutions, the results have generally seen little industry adoption. Their research has tended to ignore the existing technologies and practices present in industry, which are both mature and diverse. I believe it is unrealistic to expect a UI tool that does not integrate with existing front-end and back-end technologies, and with established practices, to be widely applicable to industry.

The third novel feature is to regard UI generation not as an alternative to manual UI construction but as a way to augment it. There is clear friction between the aesthetic and functional considerations of UI generation, and I believe the right way to resolve this is to delimit it: to define bounds around what can usefully be automated, and what should be left to the UI designer. In this way, the bounded portion can be designed to integrate with high fidelity into the UI designer's existing toolkit. The aim will be to generate the same UI the practitioner would previously have constructed manually.

Figure 1 depicts these significant features visually. It shows a UI generation process driven by existing front-end technologies (labelled 1) mining multiple, heterogeneous back-end artefacts (labelled 2) and generating a well-bounded output (labelled 3) that integrates back into the front-end. This combination of applying emerging technologies, focussing on integrating with

existing industry practices, and defining useful bounds around the generation process, will result in a unique and compelling piece of research.

Figure 1: The significant features combine to a greater whole

1.3. Structure of this Thesis

Chapter 2 will begin with a review of the existing literature. It will define the gap my thesis intends to fill. In particular section 2.2 will discuss the difficulties present in UI development, and section 2.3 will discuss the application of software mining to address them. Chapter 3 will outline the research method I propose to use to fill the gap, including my epistemology and methodology. In particular section 3.2.1 will introduce the use of Action Research. This methodology will dictate the structure of chapters 4, 5 and 6 as I progress through successive Action Research cycles. I will explore different qualities of the gap, reflecting upon and refining my research based on observations from industry practitioners and the research community. These cycles will ultimately culminate with chapter 7. This will be a verification of the applicability of my solution from both a research community and an industry perspective, and a validation of whether the gap has been filled. Chapter 8 will conclude with future work and closing remarks.

2. Literature Review

Many software projects spend a significant proportion of their time developing the UI. Research in the early 90s found that some 48% of application code and 50% of application time was devoted to implementing UIs (Myers 1992). These figures are still considered relevant today, more so with the increased demands of richly graphical and Web-based UIs (Jha 2005; Daniel et al. 2007). Basin et al. (2010) confirm "creating user interfaces is a common task in application development and one that is often time consuming and therefore expensive". Automating this task is difficult because UIs bring together many qualities of a system that are not formally specified in any one place, or are not specified in a machine-readable form. For example a dropdown widget on a UI screen may have a data type specified in a database schema but a range of choices drawn from within application code. Bringing these diverse characteristics together in one place to enable automatic UI generation is a significant challenge, and research in this field dates back over three decades. The work was given increased urgency with the emergence of ubiquitous computing (Weiser 1993) and its proliferation of different UI devices with varying capabilities. Approaches to date can be broadly grouped into three categories: interactive graphical specification tools, model-based generation tools, or language-based tools. Each has significant disadvantages which have limited its success in industry (Myers, Hudson & Pausch 2000).

The main disadvantage of the first two approaches – interactive graphical specification tools and model-based generation tools – is that they inherently require software practitioners to restate information that is already encoded elsewhere in the application. This duplication is both laborious and a source of errors, as the practitioner must take care that the application code and the UI model stay synchronised (Jelinek & Slavik 2004). The main disadvantage of the third approach – language-based tools – is that, generally, programming languages "are not sufficient enough for complex UI modelling. To completely and formally depict the UI composition and behaviour, new attributes and properties are needed to describe the object data members" (Xudong & Jiancheng 2007). Without these new attributes and properties, language-based tools must resort to generalised heuristics when generating their UIs. These "generalised heuristics result in a UI that appears quite differently from, and functions less effectively than, one that has been designed with consideration to its specific purpose" (Falb et al. 2007).

A promising technology to address these disadvantages comes from the emerging field of

software mining. Software mining is a branch of data mining focused on mining software artefacts such as code bases, program states and structural entities for useful information related to the characteristics of a system (Xie, Pei & Hassan 2007). Such a technique would be readily applicable to automatic UI generation. Rather than requiring practitioners to restate information in an interactive graphical specification tool or a model-based generation tool, a UI generator utilising software mining could potentially determine such information for itself.

Section 2.1 of this literature review surveys the field of UIs, with particular focus on automatic UI generation but also considers secondary services relevant to a software mining approach. Section 2.2 summarises the state of the art in software mining, with particular reference to those aspects that are potentially significant to automatic UI generation. Together, these two sections establish a foundation from which the thesis can explore the development of a general purpose architecture for automatic UI generation.

2.1. User Interfaces

The User Interface (UI) is that part of a software system that allows a person (the user) to interact (to interface) with the application. UIs come in many forms, though the predominant form for the past thirty years, and the one most typically associated with UIs (X Business Group 1994), is the *Graphical* User Interface (GUI).

GUIs developed from research at the Stanford Research Institute, Xerox Palo Alto Research Centre (PARC) and MIT in the 1970s, as an alternative to the then dominant 'command line' UIs. Whereas command line UIs used one-dimensional lines of text for interacting with the user, GUIs used the full two dimensions (in many cases pseudo-three dimensions) of the screen. As well as text, they could display graphics, animation and even video (Myers, Hudson & Pausch 2000), as shown in Figure 2.

GUIs represented a significant advance in the usability of software systems for non-technical users. Unfortunately, they were also significantly more complicated to build (Myers 1994). As GUIs became more mainstream and more in demand, research focused on making them easier for software practitioners to create. A logical attempt was the field of automatic GUI generation, covered in the following section. A number of less ambitious techniques were also developed – aimed at easing the burden on practitioners by reducing the complexity of key areas of GUI creation. A representative selection of these is covered in section 2.1.2.

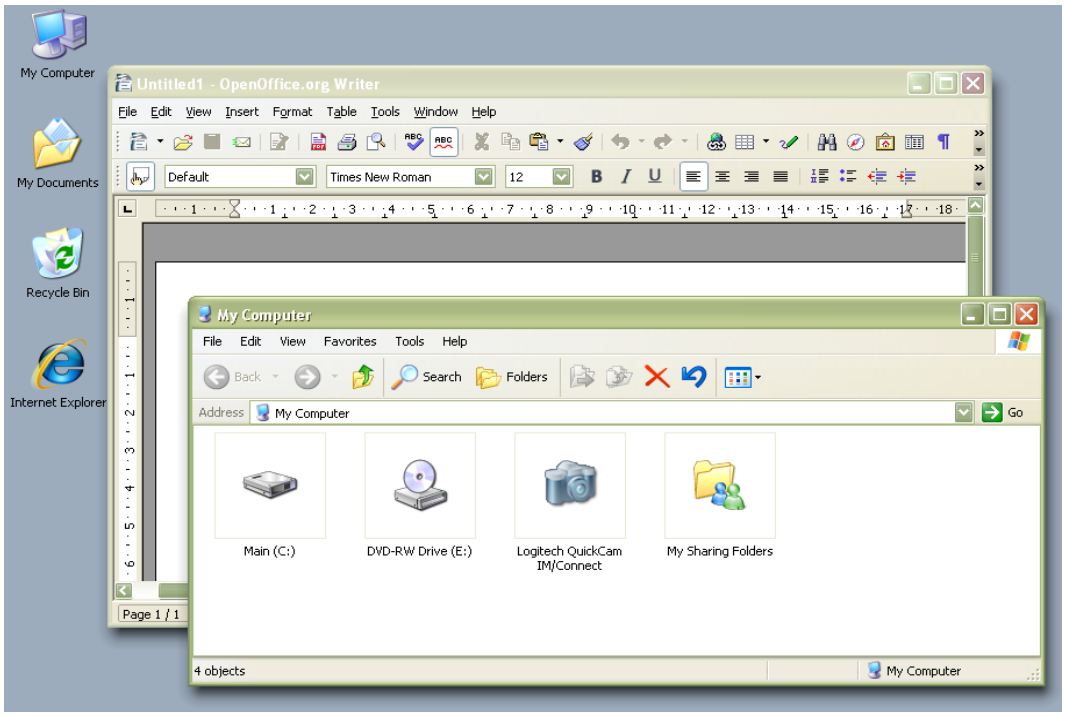


Figure 2: Graphical User Interface

It should be noted the association of GUIs with UIs is now so commonplace that, for the rest of this thesis whenever I, or any of the practitioner's interviewed, refer to a UI what I am strictly meaning is a GUI (as opposed to, say, a command-line UI).

2.1.1. Automatic UI Generation

A logical solution to the difficulty of creating UIs was to have the computer automate the process. Towards the end of the 1970s, research into automatic UI generation began at laboratories at BBN, the University of Toronto, Xerox PARC and others (Myers, Hudson & Pausch 2000). Over the next three decades, dozens of research projects attempted to address the problem. Projects including, though by no means limited to, COUSIN (Hayes, Szekely & Lerner 1985), TRIDENT (Bodart et al. 1995), Cameleon (Calvary et al. 2003), Naked Objects (Pawson 2004), UsiXML (Vanderdonckt et al. 2004), AUI (Xudong & Jiancheng 2007) and DB-USE (Tran et al. 2010) all explored a variety of techniques. The work was given increased urgency with the emergence of ubiquitous computing (Weiser 1993) and its proliferation of different UI devices with varying capabilities.

To avoid confusion, I should clarify there was similar research conducted over a similar period related to the *architecture* of an application and its connection with its User Interface. A notable

example being the Model View Controller (MVC) pattern (Krasner & Pope 1988). Such research focused on issues such as separation of concerns (Dijkstra 1982), loose coupling (Stevens 1974) and the process of building applications. This is distinct from automating parts of the process itself. For example MVC makes no prescription as to *how* its Views are constructed, either manually by humans or automatically by machines.

From research in the field of UI generation to date, it can broadly be observed there are three approaches to automatic UI generation: interactive graphical specification tools, model-based generation tools, and language-based tools (Myers, Hudson & Pausch 2000). Each of these will be considered in the following sections.

2.1.1.1. Interactive Graphical Specification Tools

Interactive graphical specification tools allow practitioners to build UIs in a similar way to how they might be sketched on paper (Vanderdonckt et al. 2004). As shown in Figure 3, these tools generally display a graphical representation of a UI form alongside a 'palette' of UI widgets (such as text boxes and check boxes) based on what the underlying framework provides.

The practitioner drags and drops widgets into position on the form, and can further customise them through sets of widget properties such as colour and font. In What You See Is What You Get (WYSIWYG) fashion, the representation of the UI in the interactive graphical specification tool closely matches the appearance of the UI when executing in the final application. Indeed many tools allow the practitioner to toggle a lightweight preview mode whilst still in the tool which simply takes the existing dragged and dropped widgets and makes them 'live': they behave as if running in the final application and accept input, rather than allowing themselves to be repositioned (Cardelli 1988). This technique is often referred to as 'visual' programming because the practitioner effectively builds code not by typing words but by dragging and dropping and making visual gestures with the mouse.

Once the UI widgets have been positioned, interactive graphical specification tools generally use static code generation to output code using the native programming language and Application Programming Interface (API) of the underlying framework. In most cases this is the same API that is available separately were the practitioner to build the UI programmatically. A subset of interactive graphical specification tools allow the statically generated code to be edited manually, and a further subset perform two-way synchronisation between the edited code and the tool. Finally, recent projects have also explored interactive graphical specification tools that

output a modelling language that is then rendered to multiple frameworks (Vanderdonckt et al. 2004).

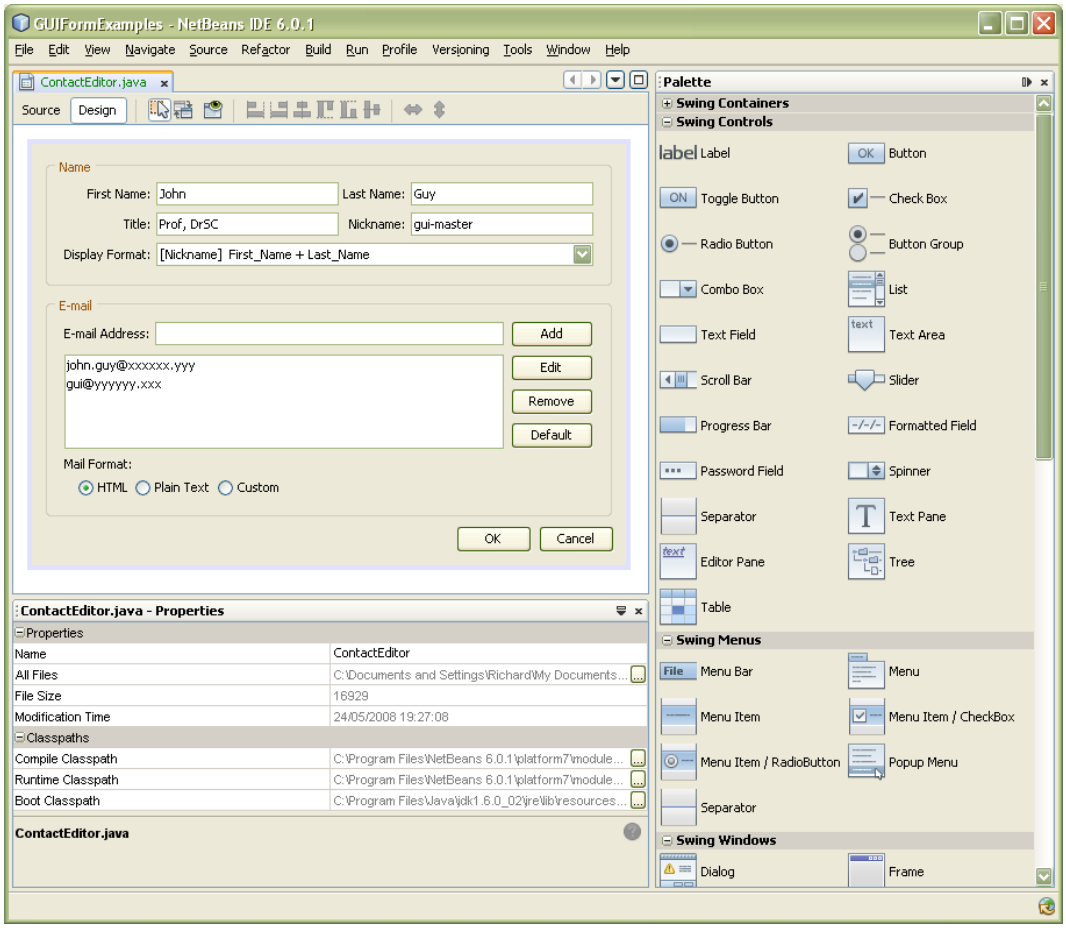


Figure 3: Interactive graphical specification tool

Overall, interactive graphical specification tools have proven very popular in industry. They have an intuitive appeal, and most established UI frameworks provide such tools. Notable examples include Microsoft Visual Studio (Visual Studio 2011) and the NetBeans Matisse Editor (NetBeans 2011).

2.1.1.2. Model-Based Generation Tools

Rather than requiring practitioners to specify precisely where each widget should be positioned (and more onerously, how they should resize), model-based generation tools encourage a declarative approach: practitioners specify *what* widgets are required, but their exact appearance and layout are left to whichever implementation ultimately renders the language (Bodart et al. 1995; Hayes, Szekely & Lerner 1985; Xudong & Jiancheng 2007). There are two significant

advantages to this approach.

Firstly, whilst interactive graphical specification tools theoretically allow fine-grained customisation of every property of every widget, in practice practitioners generally want every widget to appear the same. This is because uniformity is a desirable trait in UIs and inconsistent use of, for example, colours or fonts detracts from usability (Myers, Hudson & Pausch 2000). Achieving consistency in a model-based generation tool is easier than with an interactive graphical specification tool, because model-based generation tools defer the responsibility of choosing fonts, colours and so forth to the renderer. The second advantage is that, because exact choice of widgets is deferred, the same model can target multiple UI frameworks. A notable example is HyperText Markup Language (HTML 1999). A Web browser reads a declarative HTML model, such as Figure 4, and renders it using a platform-specific UI framework, such as Win32 controls or X-Windows widgets (Raggett, Hors & Jacobs 1999).

```
<input type="text" name="firstname">
```

Figure 4: Declarative HTML model

In a subset of these model-based generation tools, the same model is further used to target multiple devices, for example Web and mobile-based devices. These attempts tend to be less successful, however, because generally the model does not encapsulate sufficient information to automatically regenerate the application to suit the varying device constraints. For example a UI screen that fits comfortably on a single desktop monitor may need to be rendered across several screens for mobile devices with a restricted screen size. The model typically does not define suitable points at which to split the screen, nor which navigation buttons to display after doing so, though some projects have investigated adding such demarcation (Gajos & Weld 2004; Menkhaus & Pree 2002).

Despite this shortcoming, model-based generation tools have proven very popular in industry. They are arguably less intuitive than interactive graphical specification tools, but offer an easier way to rapidly develop consistent UIs. Notable examples include HTML (HTML 1999) and XML User Interface Language (XUL 2001).

2.1.1.3. Language-Based Tools

Other UI generation approaches eschew interactive graphical specification tools and models in

favour of deriving the UI from an underlying programming language. Pawson's (2004) Naked Objects thesis works by imposing a 'behaviourally complete' methodology on a system's architecture, such that “all the functionality associated with a given entity [must be] encapsulated in that entity, rather than being provided in the form of external functional procedures that act upon the entities”. The UI can then be rendered as a direct representation of those objects, with UI actions explicitly creating and retrieving domain objects and invoking an object's methods.

The advantage of the Naked Objects approach is that the UI can be built and reworked very rapidly from the domain. This advantage has been lauded by both the research community and industry. Trygve Reenskaug, inventor of the Model-View-Controller (MVC) pattern extolled “In the quarter century since the inception of MVC, there has been little progress in empowering the users. This is where Pawson’s work comes as a fresh contribution in an otherwise drab market… Naked Objects represent a new beginning pointing towards a novel generation of human-centred information systems” (Pawson 2004). It has been similarly praised by Dave Thomas, co-author of The Agile Manifesto: “Naked Objects is the embodiment of the Agile movement: lean, elegant, user-focused, and with testing built right in. Reduce a problem to its bare essentials, code it up with no extra fluff, then ship it out. Naked Objects brings programming back to its real purpose: expressing and solving business problems” (Pawson & Matthews 2002).

However most industry systems are not behaviourally complete. Pawson's (2004) treatise is more concerned with espousing an approach to object-oriented architecture, with UI generation as a useful by-product, rather than being about a UI generator applicable to industry systems. Pawson realises that “most object-oriented designs, and especially object-oriented designs for business systems, do not match [my] ideal of behavioural-completeness”. Rather they adopt an “anaemic domain model” (Fowler 2002) characterised by “dumb entity objects controlled by a number of controller objects” (Firesmith 1996). The term 'controller objects' would include, but not be limited to, persistence contexts, validation subsystems, rule engines and Business Process Modelling (BPM) languages. It is important to appreciate this is not, as Pawson suggests, because most business systems are poorly designed. Rather, they are seeking to leverage functionality provided by a rich ecosystem of mature subsystems available in industry, in order to increase productivity and reduce development cost (see 2.1.2).

But even behaviourally complete domain objects are poor vehicles with which to express all the abstractions and characteristics of a UI. As Xudong and Jiancheng (2007) observe “simple

naked objects are not sufficient enough for complex UI modelling. To completely and formally depict the UI composition and behaviour, new attributes and properties are needed to describe the object data members". In order to work around such limited domain model information, language-based tools necessarily resort to generic and stylised sets of screens and actions, such as shown in figure 5. These are sometimes referred to as Object Oriented User Interfaces (OOUI) because they closely mirror the underlying object oriented domain. The set of actions known as Create, Update, Retrieve and Delete (CRUD) is another popular generalisation (Tran et al. 2010).

Figure 5: Language-based tool

Unfortunately the use of such generalised heuristics results in UIs that appear quite differently from, and function less effectively than, those designed with consideration to their specific purpose (Falb et al. 2007). There is no attempt to generate the same UI the practitioner would previously have constructed manually. Constantine (2002) criticises: "The [Naked Object] solution for usability? Eliminate user interface design altogether… The greatest usability problem with Naked Objects is the one-size-fits-all premise on which the approach rests. Instead of tailoring the presentation of information and the operation of the user interface to fit the unique aspects of the context, the application, and the user needs, one solution is presumed to fit all problems, provided all the relevant domain objects are properly identified with all their important behaviour fully modelled". Raneburger (2010) agrees "without the explicit

specification of details concerning layout issues or design for example, the resulting UIs are hardly ever satisfying. The problem is that high level models... focus on high level aspects of the interaction. They clearly specify the interaction with the system but they are not able to capture requirements on design or usability issues". Overall this disadvantage has meant that language-based tools have had limited popularity in industry. Notable examples include JMatter (JMatter 2011) and OpenXava (OpenXava 2011).

Another significant disadvantage of language-based tools is that by trying to automatically generate the entire UI they neglect many of the services UI frameworks offer. These services are often very mature and feature rich, and practitioners have grown accustomed to having them available. A representative selection of these are discussed in the next section.

2.1.2. Other UI Framework Services

Whilst the ideal solution to the difficulty of creating UIs is to have the computer automate the entire process, this likely requires a degree of machine intelligence not possible with current technologies. A number of less ambitious, but more immediately practical and arguably more successful, approaches have "focused on a particular part of the user interface that was a significant problem, and which could be addressed thoroughly and effectively" (Myers, Hudson & Pausch 2000).

Most modern UI frameworks provide these approaches as orthogonal services to their interactive graphical specification tools or model-based generation tools, as discussed in sections 2.1.1.1 and 2.1.1.2. The services significantly reduce the difficulty of UI development, and their implementations are usually very robust and feature rich. Automatic UI generators must either accommodate or replicate these services, and failure to do so significantly degrades the practicality of the generator for software practitioners. As Myers, Hudson & Pausch (2000) critique "for every user interface, it is important to control the low-level pragmatics of how the interactions look and feel, which these [automatic generation tools] tried to isolate from the designer".

A representative selection of these services is discussed in detail in the following sections.

2.1.2.1. Reusable Widgets

One of the primary functions of a UI is to allow users to input data. Many UIs provide similar

features for doing this, such as text boxes, list boxes and check boxes. UI consistency across applications is a desirable trait for usability, and it is inefficient for practitioners to continually code the same, or very similar, features for each UI. A more practical approach is to delimit each feature into a widget that can be reused from project to project (Palay et al. 1988).

An advantage of this approach is it allows the widgets to be developed in isolation of any particular application's UI. Indeed they can be developed by a separate company from the one developing the application itself. This has led to a burgeoning ecosystem of highly mature third-party widgets, covering everything from advanced input controls such as expandable trees to sophisticated output such as graphing and visualisation components. A UI generation solution that excludes this widget ecosystem limits the quality of UIs a practitioner can create.

2.1.2.2. Validation

Having input the data using the widgets described in section 2.1.2.1, an immediate secondary concern is to validate whether what was input is allowed within the context of the application. Sometimes the allowable values are enforced by the widget itself, for example a calendar widget will generally not allow users to input invalid dates. In many other cases, however, the input is open ended and unconstrained, and validation must be performed after the data has been entered (Hayes 1998).

Validation is such a common concern that many UI frameworks provide validation services. They define a validation layer and allow pluggable third-party validators that specialise in validating particular forms of input. For example an e-mail address validator can validate the contents of a text box to ensure the entered data conforms to the RFC 822 standard for valid e-mail addresses (Apache Commons Validator 2006). A UI generation solution that excludes these pluggable validators limits the robustness of UIs a practitioner can create.

2.1.2.3. Data Binding

Having input and validated the data as described in section 2.1.2.2, it must generally be transferred from the UI to the underlying application. To do this, its representation often needs to be converted. For example a text box widget may accept text input which must be converted to a number before it can be passed to the underlying domain object. In some cases, the code required to perform the conversion can be very verbose and error-prone. For example the data from an expandable tree widget may need to be converted to a hierarchical list structure before

use on the back-end.

Many UI frameworks provide data binding adapters to help address this complexity (Woolf 1994). A UI generation solution that excludes such adapters limits the richness of both the UI and the domain model a practitioner can create.

2.1.2.4. Navigation

Having input, validated, and sent the data to the underlying application as described in section 2.1.2.3, the user will typically need to be moved to a new screen in the UI. In some cases the next screen will be pre-determined, but often it will depend on the particular values that were input.

The mechanism behind deciding which screen to display next may be highly complex, involving externalised navigation files, rule engines or BPM languages. UI frameworks often provide native support for navigation (Leung 2000). A UI generation solution that excludes this navigation support limits the sophistication of UIs a practitioner can create.

Despite the rich array of UI services available, and the success of interactive graphical specification tools and model-based generation tools, there are still significant difficulties facing UI developers. These difficulties are discussed in the next section.

2.1.3. Difficulties of UI Development

As introduced in section 2.1.1 a logical solution to the difficulty of creating UIs is to have the computer automate the process. The predominant approaches to this are interactive graphical specification tools, model-based generation tools, and language-based tools. Unfortunately, each has significant disadvantages (Myers, Hudson & Pausch 2000).

The main disadvantage of language-based tools is they resort to generalised heuristics, as discussed in section 2.1.1.3. As Schramm (2010) criticises “fully automated UI generation based on a domain model produces significantly limited interfaces in terms of clarity, understandability, and usability, especially for complex models”. These tools also tend to neglect UI services such as reusable widgets, validation, data binding and navigation as discussed in section 2.1.2.

The main disadvantage of the other two approaches – interactive graphical specification tools and model-based generation tools – is that they inherently require practitioners to restate information that is already encoded elsewhere in the application. In doing so, they introduce a margin for inconsistencies and maintenance errors. As Cruz and Faria (2010) recognise "a large part of the UI structure and functionality is closely related with the structure and functionality of the domain entities described in the domain model".

For example a domain model may encapsulate the concept of a `Person`. A `Person` has a first name, and this is represented in a database schema as a text field. When the practitioner turns to building the UI, they must explicitly redefine the `firstname` field. Either they have to drag and drop a `firstname` text box in an interactive graphical specification tool, or they have to describe a `firstname` input field in a model-based generation tool. Schramm (2010) concludes "a drawback of [interactive graphical specification tools and model-based generation tools] with respect to the effort of UI creation is that every UI widget still has to be modelled manually. In particular, the mapping of data fields of the domain model to UI widgets is a recurring task that can be automated to a great extent".

Constraints such as the maximum length of the field also have to be respecified. As the domain model changes over time these changes have to be made in both the database schema and in the UI. For example if the maximum allowed length of `firstname` is increased or a `surname` field is added, the screen code must be updated. If a `Person` appears on multiple screens, the changes usually have to be re-made separately for every screen. This problem has been recognised before, and attempts have been made to minimise it by enabling forms to 'visually inherit' widgets from one another (Miller & Ragsdale 2003). This succeeds in reducing the need to update domain model changes across multiple screens, but there is still duplication between the domain model and at least one screen in the UI. This duplication is both laborious and a source of errors, as the practitioner must be careful that the application code and the UI model stay synchronised (Jelinek & Slavik 2004). A promising technology to address this disadvantage comes from the emerging field of software mining. This could potentially allow the automatic UI generator to determine the encoded information for itself, in a general purpose fashion, rather than requiring the practitioner to restate it. Software mining is discussed in the next section.

2.2. Software Mining

As outlined at the beginning of chapter 2, the main disadvantage of interactive graphical

specification tools and model-based generation tools is that they inherently require software practitioners to restate information that is already encoded elsewhere in the application (Jelinek & Slavik 2004). The main disadvantage of language-based tools is that they must resort to generalised heuristics when generating their UI, resulting in less effective UIs (Falb et al. 2007). A promising technology to address these disadvantages comes from the emerging field of software mining.

Software mining is a sub-discipline of reverse engineering concerned with the retrieval, collation and analysis of properties pertaining to a software system. As its name implies the field is also related to data mining. Rather than mining data sets, however, software mining focuses on mining software artefacts such as code bases, program states and structural entities for useful information related to the characteristics of a system (Xie, Pei & Hassan 2007). Software mining has a broad definition that can be applied in many ways. To conceptualise the field it is instructive to review a representative, though by no means exhaustive, list of the ways it has been applied to different areas of software development.

First, with respect to programming. Grcar, Grobelnik & Mladenci (2007) describe mining class names, field names and types, along with inheritance and interface information, in order to construct API documentation and domain ontologies. Their approach further mines source code comments by relying on one of the principles of 'literate programming' – keeping documentation close to the code it refers to (Knuth 1984). Ma, Amor & Tempero (2008) do something similar, mining the names of classes and fields to discern their semantics. Sahavechaphan & Claypool (2006) analyse and demarcate sections of code as being relevant snippets for programmers to use when 'developing by example' which they describe as a “largely unwritten, yet standard, practice”.

Next, with respect to debugging. Kagdi, Collar & Maletic (2007) describe mining usages of groups of methods that are generally invoked together. A simple example might be a call to `file.open` followed sometime later by a call to `file.close`. By identifying such typical call usage patterns it is possible to expose atypical ones, which may be useful indicators of programming defects. Tan et al. (2007) use Natural Language processing to compare the consistency of source code comments with the code itself, and therefore identify either misleading comments or bugs in the code. Xie & Notkin (2005) use software mining to inspect classes and generate likely unit tests, which practitioners can then review for inclusion in their own test suites.

Finally, with respect to maintenance. Kim et al. (2007) report on analysing software version control repositories to recognise clusters of files which are generally updated together. They are therefore able to predict which files should be updated, or at least thoroughly reviewed, following future changes. Nagappan, Ball & Zeller (2006) compare different code complexity metrics along with history from defect tracking systems to predict areas of code that are likely to be defect-prone going forward. Breu, Zimmermann & Lindig (2006) mine classes and methods for 'cross cutting concerns' – areas of duplicated functionality in the codebase that emerge unplanned over time and should potentially be refactored into a common subsystem.

From this diverse range of research projects, we can distil a general understanding of what is considered 'software mining'. By way of clarification it is worth noting there is a strong emphasis in the literature on mining software repositories. Indeed one of software mining's premier annual conferences is titled Mining Software Repositories (MSR 2004-present). This is arguably a misleading title because in industry parlance the phrase 'software repository' typically refers to software systems such as CVS (CVS 2011) or SVN (SVN 2011). The primary feature of such systems is their versioning capability – the 'V' in their names – which tracks changes over time. While some of the examples of software mining covered in this section do indeed mine such version histories (Kim et al. 2007), most use software repositories simply as a convenient place to locate source code, defect reports and other documents. There are, of course, other places to locate such artefacts in which case the repository has little bearing on the software mining itself.

Note also that although statistical methods from the field of data mining are often employed, this is not always the case. For our goal of UI generation, analysing software mining data statistically is less desirable because “UI tools which use automatic techniques that are sometimes unpredictable have been poorly received by programmers” and “predictability seems to be very important to [UI] programmers and should not be sacrificed to make the tools 'smarter'” (Myers, Hudson & Pausch 2000).

Having discerned a general understanding of the field of software mining, we turn now to a more in-depth discussion. Cerulo (2006) divides software mining broadly into three categories: static analysis, dynamic analysis and historic analysis. Each of these is discussed in the following sections.

2.2.1. Static Analysis

Static analysis “happens when software is analysed on its descriptive dimension. It is performed on software artefacts without actually executing them” (Cerulo 2006). The most commonly thought of static artefacts are source code files, which are discussed in the following section. It is also important to consider files that contain externalised behaviour, which are considered in section 2.2.1.2.

2.2.1.1. Source code

The term 'source code' is generally used to refer to the human-readable programming language code that is later compiled or interpreted into machine-readable code. The majority of an application's functionality is crystallised into its source code, making it a rich source for mining interesting software artefacts.

The techniques used to extract software mining information from source code are well established, being the same techniques traditionally used by compilers or IDEs – such as Abstract Syntax Trees (ASTs) and program dependency graphs (PDGs). Neamtiu, Foster & Hicks (2005) describe using ASTs in conjunction with software version histories to mine the evolution of an application's code over time. ASTs enable practitioners to model the semantics in a language – such as its global variables, types and functions. This allows more meaningful reporting on an application's evolution than, say, comparing number of lines of code or number of files. Liu et al. (2006) discuss using PDGs to compare source code for plagiarism between codebases of different applications. PDGs model the program structure of an application, allowing plagiarism comparisons on a deeper level that cannot be deceived by function renaming or statement reordering.

As rich a source of information as source code analysis is, however, not all of an application's behaviour can be determined from source code alone. Some behaviour is externalised, as discussed in the next section.

2.2.1.2. Externalised behaviour

As mentioned in the previous section, the majority of an application's functionality is crystallised into its source code. Increasingly, however, significant amounts of behaviour are being externalised (Rouvellou et al. 1999). There are several motivating factors for this.

Firstly, the externalised behaviour can be expressed in a form more natural to its content. For example system configuration settings may be expressed in a hierarchical XML (XML 2008) file with XML Schema validation. This file can be reviewed and modified by system administrators rather than requiring software practitioners. Similarly, business rules may be expressed in a BPM language that is closer to natural language and easier for non-technical, business users to read and verify (Rouvellou et al. 1999).

Secondly, the externalised behaviour can be updated at a different frequency to the application code. This is desirable because behaviour such as business rules are often more volatile than the rest of an application's code. Modifying the application itself requires specialised skills and carries with it the risk of introducing defects, whereas modifying configuration files or business rules is a more defined process with less margin for error and can therefore be performed by system administrators or business users (Rouvellou et al. 1999).

Configuration files, BPML files and other such externalised representations are not generally considered 'source code', but are still valuable repositories of information for software mining. Another valuable set of information can be discerned by dynamic analysis, discussed in the next section.

2.2.2. Dynamic Analysis

Unlike static analysis, which centres on mining source code and externalised behaviour files, dynamic analysis “happens when software is analysed on its executive dimension” (Cerulo 2006). The rationale for dynamic analysis is similar to that of mining files containing externalised behaviour: whilst source code captures many properties of an application, a significant amount of its behaviour can only be determined from other places.

For dynamic analysis, a notable example is user input. What the user chooses to input can only be determined at execution time, but may have a large impact on the application's behaviour. In the most complicated case the user may be allowed to input source code itself, such as a spreadsheet formula or a scripting language macro, which can add new functionality and screens to an application.

The most common approach to dynamic analysis is reflection, which is discussed in the following section. Some programming environments further support dynamic analysis of embedded metadata, which is considered in section 2.2.2.2.

2.2.2.1. Reflection

Reflection allows an executing program to dynamically analyse itself, by inspecting not just the values but the structure of its own data. Unlike other programming paradigms such as procedural and object-oriented programming, which specify pre-determined sequences of operations, reflection allows the sequences of operations to be determined at execution time based on the data being operated on (Maes 1987).

Reflection can be used to mine the characteristics of an executing program more accurately than static source code analysis. For example a method `foo` may be declared in source code to accept an object of type `bar`. At runtime, the actual type passed to `foo` may be `subBar`, a subtype of `bar` with additional properties. Reflection can correctly detect the subtype, whereas static source code analysis could never make this prediction. As Cerulo (2006) puts it "static analysis is affected by the undecidability problem". Furthermore, reflection can not only detect the subtype, but can also discern its properties – even if it had no prior knowledge of the subtype existing. This makes reflection a powerful tool to handle cases where user input can dynamically add scripts and screens to an application, as discussed in the previous section.

Despite its power, reflection is limited in that it can only inspect characteristics that are pre-defined by the platform. For example a platform that supports concepts such as classes and methods may allow reflection of class names and method names but may not allow, say, reflection of the cardinality of the relationship between two classes. For example, one-to-one or many-to-one. To determine such application-specific characteristics, the platform can support an extensible mechanism such as embedding arbitrary metadata.

2.2.2.2. Embedded Metadata

Beyond reflection, some software environments provide explicit support for dynamic analysis of embedded metadata. For example the 'attributes' feature in the .NET Common Language Runtime (CLR) (Miller & Ragsdale 2003) and the 'annotations' feature in the Java Virtual Machine (JVM) (Gosling 2005). Most languages that run atop these environments expose this capability, allowing practitioners to embed metadata into their source code. For example C# (Hejlsberg 2006) and VB.NET (VB.NET 2005) on the CLR, and Java (Gosling 2005) and Groovy (Groovy 2011) on the JVM, all offer this capability. The metadata can then be dynamically extracted from instantiated objects at runtime.

Embedded metadata allows practitioners to tag domain objects with arbitrary information, beyond the normal capabilities of the platform. This can later be inspected by application frameworks to allow specialised processing. For example the Java platform does not natively support specifying a maximum length for a String (Gosling 2005). This is a problem for Object Relational Mapping (ORM) frameworks which require such information when generating database schemas. Using embedded metadata, this information can be added to the source code alongside the Java field it refers to. The metadata is largely ignored by the JVM, but is significant to the ORM framework that is watching for it (DeMichiel & Keith 2006).

Whilst such metadata is valuable, its disadvantage is that it must be specified explicitly by the practitioner, adding complexity to the code. There is another form of metadata that is more implicit, and can be extracted from existing resources. We turn now to discuss this in section 2.2.3.

2.2.3. Historic Analysis

Historic analysis "happens when software is analysed on its evolutive dimension… on software process trails left by developers during their activities and stored in software [version] repositories" (Cerulo 2006). A great deal of insight into the characteristics of a piece of code can be gained by studying how it has changed over time: code that sees frequent changes may be considered less mature; code that sees changes by many different practitioners may be more prone to error; code that is often involved in defect reports may be a candidate for rework. Such metadata accrues transparently. It can be extracted months or years after the fact, and does not require practitioners to have deliberately embedded it.

Historic analysis mines the 'human' element of a project more than static or dynamic analysis can. Version control is a primary resource but there are others, such as defect tracking databases and project mailing lists. Rigby and Hassan (2007) describe using psychometric text analysis across mailing lists to understand the practitioner personalities behind an application and their impact on the relative success of the project.

To summarise the three forms of software mining, it can be seen they offer different perspectives on retrieving information related to the characteristics of a system (table 1). The promise of software mining is to be able to *combine* these perspectives. The potent combination of mining source code, embedded metadata, version histories and other artefacts brings significant potential benefits. These are discussed in the next section.

Categories of Software Mining				
Static		**Dynamic**		**Historic**
Source Code	Externalised Behaviour	Reflection	Embedded Metadata	Process Trails

Table 1: Categories of Software Mining

2.2.4. Potential Benefits of Software Mining

To conclude our discussion of software mining, we consider its application to our stated problem of automatic UI generation. There are two potential benefits.

First is software mining's ability to eliminate re-specification of a UI by examining the existing system architecture. Such an ability addresses many of the problems discussed in section 2.1.3 (having to redefine field names, types, constraints). By itself, however, it is no better than the stylised, language-based approaches discussed in section 2.1.1.3, which invariably resort to generalised heuristics because no single source of examination is comprehensive. As Schofield et al. (2006) observe, "our understanding, as a community, has shown that multiple types of analyses may be relevant to understanding some aspect of [a] system… different types of evidence might be complementary, in which case their cross-referenced analysis and interpretation should increase the quality of the inferred knowledge, assuming that the computational resources for their extraction are available". Similarly, German, Cubranic & Storey (2005) stress "extracting information from most information sources is relatively straightforward. But many questions can only be answered by correlating information from multiple sources".

The second benefit, therefore, is software mining's ability to collate information from multiple, heterogeneous sources. This is a unique proposition of software mining. Indeed without its collation and analysis dimensions, software mining would be little more than a modern, umbrella term for long established techniques such as parsing and reflection. Collation is made possible because each of the heterogeneous techniques can be applied with the understanding that it is part of a larger software mining process. It can be written to normalise and homogenise the results of its analysis with results from complementary analyses. The promise of software mining is that, by completing such a diverse and thorough analysis, we can obtain sufficient information to avoid resorting to generalised UI generation heuristics.

To demonstrate this concretely, the next section will explore a sample of industry subsystems, applicable to software mining for the purposes of UI generation, and show how they may be collated to construct a UI.

2.2.5. Demonstration of Software Mining

This section explores a sample of mainstream subsystems in use by industry applications, and demonstrates how software mining can use them to construct a UI. The example subsystems are taken from the Java platform, being one of the dominant industry platforms in use today (TIOBE 2011). Note the sample is not limited to those subsystems that typically exist on the same application tier. A promise of software mining is that it can source information from wherever it can be found, including subsystems that may be located remotely from one another, even if special considerations must be made in order to access them securely.

2.2.5.1. Properties Subsystem

The JavaBean (JavaBean 2008) specification was introduced in version 1.1 of the Java platform to enable the declaration of publicly accessible properties. It is more a convention than a part of the language, as it relies on methods with a particular signature. For example to declare a JavaBean `Hotel` with two properties `name` and `stars`, a practitioner would write:

```
public class Hotel {
   private String mName;
   private int    mStars;
   public String getName() {
      return mName;
   }
   public void setName(String name) {
      mName = name;
   }
   public int getStars() {
      return mStars;
   }
   public void setStars(int stars) {
      mStars = stars;
   }
}
```

A common criticism of the JavaBean convention, aside from its verbosity, is that it contains

error-prone duplication: the methods `getName` and `setName` must both contain the word `Name`, and must both use a type of `String`. In most cases, this name and type will further be mirrored by the private member variable `mName`. More modern languages provide better support for properties. Groovy (a language which runs on the JVM and has a similar syntax but different features) supports properties at the language-level (Groovy 2008). In Groovy, a practitioner would write:

```
class Hotel {
   String name;
   int stars;
}
```

Whether properties are specified using JavaBeans, Groovy, or some other mechanism, the important point is both the name and type are concretely specified by the properties subsystem.

2.2.5.2. Persistence Subsystem

Most business applications persist their data to long-term storage, such as a database. To continue the `Hotel` example from the previous section, the practitioner may define the following SQL (1987) schema to store a `Hotel`:

```
TABLE hotel (
   name varchar(30) NOT NULL,
   stars int
);
```

The persistence subsystem contains new information compared to the properties subsystem. Strings in Java do not have any concept of maximum length (Gosling 2005). They are also implicitly nullable. Conversely, from the SQL schema it can be seen that name is actually limited to 30 characters and is not-nullable (i.e. is a required field). Clearly the properties subsystem alone is not sufficient to fully describe the domain model.

Length and nullability information can also be embedded in Object Relational Mapping (ORM) subsystems. Hibernate (2008) allows the practitioner to specify mapping files to map properties to database schemas:

```
<hibernate-mapping>
   <class name="Hotel">
      <property name="name" length="30" not-null="true"/>
```

```
            <property name="stars"/>
        </class>
    </hibernate-mapping>
```

Another ORM, the Java Persistence API (JPA) (DeMichiel & Keith 2006), encapsulates the same information using metadata annotations (Gosling 2005) atop the properties subsystem:

```
    public class Hotel {
        …
        @Column(length=30,nullable=false)
        public String getName() {
            return mName;
        }
```

It can be seen there are multiple ways to specify persistence-specific metadata. This metadata is both external to other subsystems (i.e. the properties subsystem from the previous section, the validation subsystem in the next section) but essential to capturing a complete understanding of the domain model.

2.2.5.3. Validation Subsystem

Persistence subsystems generally fail poorly when presented with invalid data, either returning error messages unsuitable for end-user consumption or lacking the expressiveness with which to define business constraints (such as minimum and maximum values). Therefore it is desirable to pre-validate the data and, if necessary, constrain it or return more meaningful messages. Validation subsystems such as Commons Validator (2008) use XML files to specify validation rules:

```
    <form name="hotel">
        <field property="stars" depends="intRange">
            <var>
                <var-name>min</var-name><var-value>1</var-value>
                <var-name>max</var-name><var-value>5</var-value>
            </var>
        </field>
    </form>
```

Another validation subsystem, OVal (OVal 2011) uses metadata annotations on the properties:

```
    public class Hotel {
```

```
…
@Range( min=1, max=5 )
public int getStars() {
   return mStars;
}
```

Again, it can be seen there are multiple ways to specify validation-specific metadata. This is both orthogonal to property, persistence and other subsystems, and an important part of the domain model as a whole.

2.2.5.4. Business Process Modelling Subsystems

BPM subsystems externalise and formalise the business rules of an application. For example using JBoss jBPM (jBPM 2008) a practitioner can specify the valid actions available when editing a `Hotel`.

```
<page name="editHotel">
   <transition name="save" to="hotelSaved"/>
   <transition name="delete" to="hotelDeleted"/>
</page>
```

Generally it is these actions, and only these actions, that should be presented to the user in the UI.

2.2.5.5. Collating Software Mining Results

Let us propose a simple intermediary XML format for our software mining results. We could mine the properties subsystem (see 2.2.5.1) using a mechanism such as the JavaBean API. Defined in pseudo-code:

```
function mine( toMine )
   var xml = new XmlDocument
   xml.startElement "inspection-result"
   xml.startElement "entity"
   xml.writeAttribute "type" toMine.type
   foreach property in toMine
      xml.startElement "property"
      xml.writeAttribute "name" property.name
      xml.writeAttribute "type" property.type
```

```
            xml.endElement "property"
         end foreach
         xml.endElement "entity"
         xml.endElement "inspection-result"
         return xml
      end function
```

Applying this to the definition of the `Hotel` class we could return the following:

```
      <inspection-result>
         <entity type="Hotel">
            <property name="name" type="string"/>
            <property name="stars" type="integer"/>
         </entity>
      </inspection-result>
```

Separately, we could mine the validation subsystem's XML files (see 2.2.5.3) using a mechanism such as the DOM API, for the `Hotel` constraints:

```
      <inspection-result>
         <entity type="Hotel">
           <property name="stars" minimum-value="1" maximum-value="5"/>
         </entity>
      </inspection-result>
```

These results could further be collated with those from other subsystems, such as the persistence and BPM subsystems. There are several algorithms we could posit for such collation, which we shall explore further in section 4.2.1.3. For now we will simply collate entity nodes based on their `type` attribute (which is unique across all entities) and property nodes based on their `name` attribute (which is unique within an entity). Defined recursively in pseudo-code:

```
      function collate( master, toAdd )
         foreach attribute in toAdd
            master.setAttribute attribute
         end foreach
         foreach childNode in toAdd
            var matched = false
            foreach masterNode in master
               if childNode.name == masterNode.name
                  matched = true
                  collate( masterNode, childNode ); break
```

```
            endif
            end foreach
            if !matched
               master.appendChild childNode
            endif
         end foreach
   end function
```

This would result in:

```
   <inspection-result>
      <entity type="Hotel">
         <property name="name" type="string" maximum-length="30" required="true"/>
         <property name="type" type="hotelType"
                   lookup="Bed and Breakfast,Apartment"/>
         <property name="stars" type="integer" minimum-value="1" maximum-value="5"/>
         <property name="rating" type="string" read-only="true"/>
         <property name="notes" type="string" large="true"/>
         <action name="save"/>
         <action name="delete"/>
      </entity>
   </inspection-result>
```

Whereby we would have mined metadata from the sources shown in table 2.

Field	**Subsystem**			
	Property	**Validation**	**Persistence**	**BPM**
Name	label, type		max length, required	
Type	label, type, lookup enum values			
Stars	label, type	min/max value		
Rating	label, type, read-only			
Notes	label, type		large field (LOB)	
Save				label, action
Delete				label, action

Table 2: Sources of software mining metadata

Together, this would be sufficient information to construct the UI shown in Figure 6. There is

still a level of ambiguity around layout and presentation, which we hope will be addressed by our approach to useful bounds of generation (see 4.1.1.4). But this example has demonstrated there are a multiplicity of sources of UI metadata. Furthermore, it is not difficult to posit other useful sources, such as Web Services Description Language files (WSDL 2001).

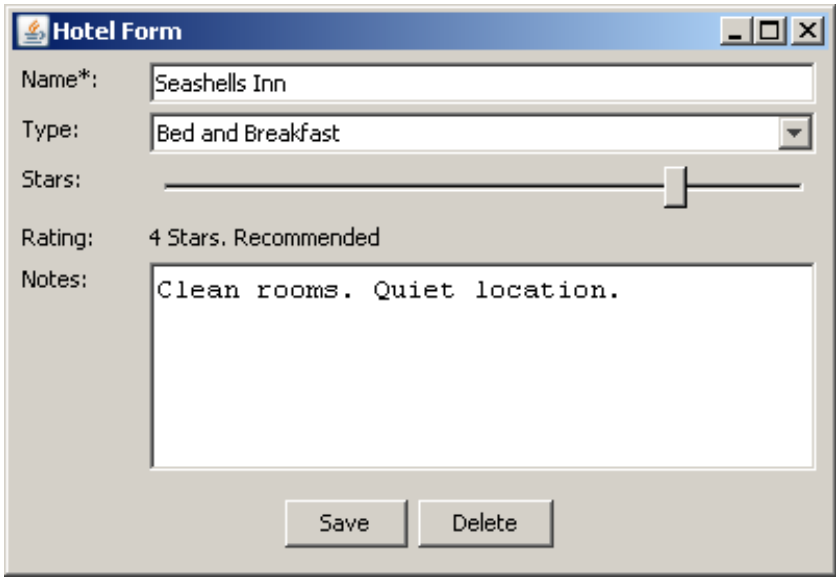


Figure 6: UI constructed from metadata

By way of closing we should delineate our discussion, and indeed this thesis, by briefly outlining the limits of software mining both today and in the future.

2.2.5.6. Limitations of Software Mining

A first limitation of software mining is that it can only mine artefacts that are actually part of the software. In a journal article (see 6.2.1.2) we wrote “automation is difficult because UIs bring together many qualities of a system that are not formally specified in any one place, if at all”. One of our reviewers criticised: “The authors are stating that the design of the UI brings together functionalities of a system that are not formally specified. I would question this assertion as an increasing number of [corporations] are hiring Human-Factors designers (HFD), with solid understanding of software development best practices, to work with potential users of the system to identify the UI needs. These HFD are part of the development team and work closely with the systems analysts and programmers to incorporate the design of the UI into the software process”. Here, the reviewer is pointing out that where we said 'formally specified' what we really meant was 'specified in machine-readable form'. There may well be detailed written documentation on, say, the correct font size for a text box – but unless this is codified in

a form that a machine can interpret it is inaccessible to software mining. Tan et al. (2007) have made some progress incorporating natural language processing into software mining, but this research is still in its infancy.

A second limitation is that, even if the software mining can interpret the data, it will be unable to collate it unless it can be mapped to some normalised key. Software mining can be quite flexible in this regard, with different approaches to mapping for different sources, but the mapping must be deterministic. If a database schema cannot be mapped to a corresponding domain object type, or a domain object type cannot be normalised with some fragment of an XML configuration file, then no meaningful collation between them can occur. In practice many of these mappings are already defined in an application and can themselves be mined. For example the persistence subsystem must have a well-defined mapping between object types and database tables, and the validation subsystem must unambiguously understand which types it applies to. The issue is less clear, however, with subsystems such as business rule engines (Rouvellou et al. 1999) whose input may be a collection of domain objects and whose output may be some rule execution. The inner workings and mappings of such subsystems may be opaque to the rest of the application and non-deterministic to mine.

A third limitation is that it may be difficult to determine when the software mining has mined 'enough'. In section 4.2.1.2. I discuss using guided software mining to prevent mining too much metadata (i.e. that will not actually be needed during UI construction). But it may be impossible for the software to know whether there was additional metadata available that could have benefited UI generation. Such metadata may be missed, resulting in a less functional UI, yet this may only be detectable using traditional techniques such as unit testing and Quality Assurance.

Despite these limitations, software mining has the potential to resolve many of the problems of extracting sufficient metadata to produce non-generic UIs, whilst at the same time not requiring the practitioner to restate that metadata in repetitive and error-prone ways. It is a promising approach and forms the core of my proposed research.

2.3. Proposed Research

Given the mainstream success of UIs described in section 2.1 and the difficulties of UI development described in section 2.1.3 I suggest there is a gap to fill. There is scope to improve the approaches of interactive graphical specification tools, model-based generation tools and language-based tools, described in section 2.1.1. Specifically, I propose the problems of

insufficient information (see 2.1.1.3) and of restating information (see 2.1.3) can be effectively addressed by the emerging field of software mining (see 2.2) and its ability to pull together multiple sources of information into a coherent whole (see 2.2.5.5).

In addition, I propose to combine this emerging technology with an industry-focussed research methodology. This will be aimed at understanding the difficulties of industrial UI development and the shortcomings of existing solutions. Such a research methodology will be critical to deriving a general purpose UI generator that is applicable to industry. We turn now to discuss this methodology in the next chapter.

3. Research Method

This chapter details the research methods this thesis chose, and the reasons for choosing them. These decisions significantly impact and shape the chapters to come.

3.1. Epistemology

Crotty (1998) urges us to consider early on the assumptions underlying our research. However this is not so we can eradicate them. The term 'assumption' often carries a negative connotation, and certainly at a high level to assume without rigour of proof is folly. At a much lower level, however, all knowledge rests on certain base assumptions. It is important to lay these bare so as to recognise the boundaries and limits of our research. In Crotty's words “at every point in our research – in our observing, our interpreting, our reporting – we inject a host of assumptions… such assumptions shape for us the meaning of research questions, the purposiveness of research methodologies, and the interpretability of research findings. Without unpacking these assumptions and clarifying them, no one (including ourselves!) can really divine what our research has been”. He puts it even more forcibly: “without [clarifying our assumptions], research is not research” (Crotty 1998).

Crotty's (1998) intent is that by examining our hitherto implicit assumptions, we may expose weaknesses in them and harden them for the defence of our thesis – to “ensure the soundness of our research and make its outcomes convincing”. He identifies a primary assumption as being which epistemology, which theory of knowledge, our research conforms to – objectivism, constructionism or subjectivism. However he stresses such labels should be “educative, not prescriptive”: we use them only to describe our viewpoint in recognised terms, not to dictate our approach. In this sense, they are rather like Software Patterns (Gamma et al. 1995).

Stated briefly, objectivism is the epistemology that every important property of some thing (or 'object') resides purely within the object itself. By 'important' we could perhaps say 'researchable': an objectivist recognises that other properties exist, for example that an object is 'nice', but sees no valid, empirical way to integrate such properties into their epistemology. Constructionism, by contrast, recognises that some important properties are 'constructed' as a result of the interaction between object and observer (or 'subject'). Crucially, constructionism weights both objective properties and constructed properties as equally 'true'. For example if the

title of my thesis was To Build A Better Lounge Chair, I might observe that my existing lounge chair was blue and uncomfortable. I might set out to make it more comfortable and, if I didn't take time to consider my epistemology, I might not realise that the truth of it being 'blue' and the truth of it being 'uncomfortable' were not, in fact, the same. The chair is objectively blue, regardless of my involvement, but its comfort is a constructed truth borne out of the interaction between myself and the chair: the concept of comfort does not exist independently of myself nor the chair. This has bearing on my research because, whilst everybody will agree the chair is blue, not everybody will have the same opinion of whether it is uncomfortable. The problem of discomfort that we are trying to solve is, at its outset, a 'soft target'. Indeed some people might opine the chair is *already* comfortable and there is no problem to solve! Finally, subjectivism is the logical other end of the spectrum whereby every important property of an object resides purely in the subject's point of view.

Situated as we are in the field of Computer Science, it is tempting to say we are of an objectivist epistemology: software programs exist (at least in the sense their existence is not a matter of opinion), lines of code are 'real', and algorithms either work or they don't (at least in the sense they do what they were intended to do). Upon deeper inspection, however, I must acknowledge the problems I am trying to solve are not objective. On the problem of duplication (see 2.1.3), whilst it may be an objective fact that multiple lines of code exist, and that their intent is the same, whether this constitutes something undesirable is the viewpoint of the individual practitioner. Other practitioners could conceivably argue having multiple copies provides flexibility, or acts as some form of 'checks and balances' safeguard. If the practitioner is being paid by the line of code, they might even see an advantage in having to write multiple copies! Clearly I disagree with such viewpoints, and believe they are in the minority, but they demonstrate the essence of the problem I am trying to solve lies partly in the viewpoint of the practitioner.

The problem of duplication (see 2.1.3), therefore, requires a constructivist epistemology: the object (the code) and the subject (the practitioner) are both required in order that the problem exist. Having acknowledged this assumption, what practical implications does it have for my research? For one, it throws into sharp relief that there are a range of practitioner experiences, and my thesis only targets a subset of them. Although I believe myself to be in the majority, I should verify that claim lest it be questioned. We shall see more on this in section 4.2.1.4.

Another consideration is that when I talk of duplication in UIs I must be precise in my language – what I really mean is duplication in g*raphical* UIs. I must be aware there are a large class of

UIs outside that. For example there are applications such as Unix's `grep` which have a command-line UI. Such applications have a UI that is far removed from their internal domain model. One could even say they have a *declarative* UI – not in the sense the UI is defined declaratively, but in that the user interfaces with it declaratively: the user says 'find me all lines that match this regular expression', with little regard to the domain model (characters, input streams and so forth). I believe the ideas in this thesis could still be of value here. Unlike other approaches this thesis' use of software mining frees it from requiring a well-defined domain model on which to operate. It would seem possible to generate a command line interface from an application based on mining its defined actions. Still, this is not a direction I will be exploring further and I acknowledge it as a boundary of my thesis.

Crotty's intent is not to undermine research by exposing its underlying assumptions. Rather, it is to strengthen research by being honest and transparent about what we know, what we don't know, and what we assume. He asserts it is important to be clear about our epistemology, because epistemology is the underpinning for all research methodologies – such as Action Research (Dick 2000) and Grounded Theory (Dick 2005) – and for all research methods – such as experiments, adoption studies and interviews. It is these we turn to in the next section.

3.2. Methodology

The previous section considered the epistemology upon which my research is built, being constructionism. Atop that, I must choose a suitable methodology. Crotty (1998) stresses there are multiple methodologies appropriate to any given epistemology, but for my purposes I require one that lends itself to regular industry interaction. Action Research (Dick 2000) seems ideally suited.

3.2.1. Action Research

As my thesis has a strong focus on industry practicality, it is important to engage my target audience early and often. One industry-based approach effective in doing this is Iterative Development. There are various definitions of Iterative Development, but all are common in trying “to avoid a single-pass sequential, document-driven, gated-step approach” (Larman & Basil 2003). The Iterative Development methodology has interesting parallels to the research methodology of Action Research (Dick 2000). Indeed Action Research's definition of a cycle of “plan, act, observe, reflect, then plan again” phases (Kemmis & McTaggart 1988) would be a

fitting description for Iterative Development. The outcomes from each cycle drive the planning for the next cycle, both in terms of expanding those areas that worked well and revisiting those that were less successful. This is particularly effective when either the problem or the solution are not well-defined, as they afford the work the agility to change as its goals become clearer. For my research methodology, I will employ Action Research as a framework within which to formalise Iterative Development. This will allow the work to be accessible to, and guided by, feedback from both industry and the research community.

3.2.1.1. Plan

I will plan each Action Research cycle by taking the lessons from the previous cycle, researching the literature, evaluating current trends within industry, then deciding on a set of features and technologies to target for the next cycle. To foster a strong, combined research community and industry approach I will document all planning using a blog. Blogging is a very familiar medium to industry, and affords a high level of immediacy and transparency. This encourages early feedback.

Planning is a very fluid process, containing a significant amount of experimentation, intuition and “conversing with the problem” (Schön 1983). It proceeds in small steps, small experimental changes, with each change “talking back” to the planner to help them better understand the problem, adjust their plans, and proceed to the next small step. Such fluid processes, whilst common (though not necessarily explicit) in industry, have traditionally seemed uncomfortably opaque to the research community as they lack the rigour of well-defined goals and evaluation criteria. One methodology that attempts to provide rigour is Reflection In Action (Schön 1983). This methodology stresses the importance of regular and explicit reflection after each small step. Whilst the goals and criteria for the reflection itself are still relatively opaque, relying on the skill and intuition of the practitioner, at least the experimental process becomes much more rigorous.

3.2.1.2. Act

To continue my theme of a strong, combined research community and industry approach I propose to establish an Open Source project that is developed using Iterative Development. The concrete result of each Action Research 'act' phase will be a new release of the Open Source project to industry.

As with the planning phase, the 'act' phase will be dependent on the skill and intuition of me as a practitioner. There will be many hundreds of small design decisions, trade-offs and dead-ends involved in moving from plan to release. In order to give some rigour to these decisions, I will again make use of Reflection In Action (Schön 1983). These reflections will be documented throughout chapters 4, 5 and 6.

3.2.1.3. Observe

Following each 'act' phase, and its associated new release of the Open Source project, I will employ a variety of techniques to promote the project both within industry and the research community. I will publish research papers and journal articles, speak at conferences, and contact authors and users of related software. I will construct an attractive and easily accessible Web site from which industry practitioners can download the Open Source project, as well as maintaining a blog, message forum, User Guide, reference manual, and interviews on industry portals to publicise it. All this activity should result in feedback for me to collate.

In addition, I can conduct experiments at this phase and give them well-defined evaluation criteria. One concrete measure of my thesis' objective (a general purpose architecture) would be how successfully I can retrofit existing applications to reduce duplication in their UI layer without significantly altering their code. This would include being able to *partially* migrate applications one screen, or even one piece of a screen, at a time. I will select existing applications and verify how effectively I can apply my Open Source project to them. Assuming the project manages to gain a following in industry, later Action Research cycles may be able to gather further feedback data from adoption studies of others using my software. These may be adoption studies of retrofitting existing applications, or of developing new applications.

Further data can be collected from interviews. I will conduct interviews using Simplified Grounded Theory (Dick 2005), discussed in the next section.

3.2.1.3.1. Grounded Theory

Grounded Theory is a qualitative methodology for deriving theories from data. It proceeds by first gathering the data, in my case through interviews. It then codifies the data into categories and themes, and finally constructs hypothesises. Grounded Theory stresses that outcomes should be explicitly emergent, rather than simply verification of existing hypothesises.

This is well suited to my need because, whilst I can draw on my own experiences as an industry practitioner, it is important I am open minded to the experiences of others. I cannot simply assume the problem and dictate it to my interviewees. Rather, I should compose standardised lists of directed but unbiased, open-ended questions (Valenzuela & Shrivastava 2002). The lists should be short and each question framed broadly so as to allow the candidates room to talk openly. At the same time, they should guide the interviewee into the gap defined in my literature review (see 2.1.3). I then document, compare and categorise what emerges. Ideally the resultant categories will include my own understanding of the problem, but should also expose perspectives I have not considered.

3.2.1.4. Reflect

Reflection is arguably the most important phase of the Action Research cycle. Certainly, it is the part that adds rigour to the proceedings, solidifying their validity from a research standpoint. Whilst Action Research delineates an explicit 'reflect' phase at the end of each cycle (Kemmis & McTaggart 1988) reflection actually happens on a smaller, more implicit, scale throughout every phase.

During the 'plan', 'act' and 'observe' phases, reflection happens 'in action': the practitioner approaches each seemingly unique problem and sub-problem by first attempting to reframe them in familiar terms or using familiar analogies. He then proceeds under this re-framed understanding, which may or may not prove valid as the problem "talks back" (Schön 1983) to him. This "reflective conversation with the situation" either validates or challenges his original attempt to re-frame it. If the latter, he "surfaces the issues", based on his deeper understanding, and attempts a new re-framing which may allow him to progress. Once the re-framing seems valid, he relies on his "knowing in practice" to evaluate it using criteria based on intuitive judgement. Whilst these criteria may be difficult to articulate, they bring validity and are objective in the sense they are independent of mere opinion – they are based on invariants of the discipline, such as overarching theories (e.g. Object Oriented design, separation of concerns), vocabulary (e.g. design patterns) and established metrics (e.g. elegance, coherence, performance) (Schön 1983). It is very much a constructionist epistemology: when "at work and excited by his project. [The practitioner's] first practical step is retrospective… to consider or reconsider… to engage in a sort of dialogue with [the problem]" (Lévi-Strauss 1966)

These sorts of reflections take place implicitly during the first three Action Research phases,

when the “action present” (Schön 1983) is measured in minutes or hours. During planning, the practitioner may change his mind while fleshing out a task. Whilst implementing, he may feel a subcomponent doesn't fit elegantly with other components and refactor it. Finally, during observing, he may get a surprising answer to an interview question and posit a probe question (Dick 2005). During the final Action Research phase, the action present is much more deliberate, being measured in days or weeks, which leads to more formality in the Validity, Verification and Testing (VVT) of the Action Research cycle. The practitioner may develop automated test scripts and use code coverage tools. They may conduct internal experiments. They may reflect on feedback from users, whose evaluation criteria will be external to his own and therefore less biased. They may redredge interview recordings looking for new insights. Finally, they may seek outside adoption studies from companies that have adopted the project and study its successes and shortcomings within their organisation.

Combined, the overarching research methodology of Action Research, and within its cycles the methodologies of Iterative Development, reflection in action, Grounded Theory, interviews, experiments and adoption studies should allow me to approach the problem from sufficient angles to both concretely define it and address it.

3.3. Design

To return to our epistemological example of an uncomfortable lounge chair (see 3.1), some researchers stress the point further. Bucciarelli (1994) argues that even focussing on purely objective properties such as 'blue', and without considering subjective properties such as 'comfort', the number and variety of objective properties alone leads to many different interpretations.

For example an upholsterer regarding our blue lounge chair may define it principally in terms of the colour and texture of the fabric, with less emphasis on the shape and maybe only passing reference to the timber used or its internal construction. Conversely, a woodworker may see the lounge chair primarily in terms of structure, type of timber, joints and so on, with little consideration for the fabric. Each practitioner assembles for themselves their own “object world” (Bucciarelli 1994) that defines an object in terms most familiar to them.

There is sufficient complexity in even everyday objects that it quickly becomes impractical to regard them completely from all dimensions. Every practitioner must manage this complexity by minimising some properties, emphasising others. This inevitably leads to difficulties when

'object worlds collide' – when woodworkers need to talk to upholsterers – as practitioners must translate each other's object worlds into their own. This “negotiation of meaning” becomes a leading cause of misunderstanding and conflict. Indeed such is the magnitude of the problem it leads Bucciarelli to conclude design is a social process as much as a scientific one. This has significant implications for those seeking to refine it: “attempts to improve the engineering design process by critics and assessors of that process have been, for the most part, couched wholly in instrumental terms… these instrumental approaches are deficient when applied to design process considered as a social process awash in uncertainty and ambiguity. They miss many of the trees in the forest.” (Bucciarelli 2002)

All of this is not to say Bucciarelli (1994) is a subjectivist. He recognises there are real constraints, real scientific laws, and contends only that design is *more* subjective than generally considered. The problem, the solution, even the structure of the organisation within which the solution is devised, are fluid and subject to interpretation as object worlds. “Object worlds are personal worlds. They derive from an individual's schooling in a discipline, are tempered and shaped further by an individual's work experience” (Bucciarelli 1994). They represent “the 'feel' that designers bring to the work… that enables them to bridge the gap between the formal, abstract knowledge of underlying form and the practical concreteness of the immediate object” (Bucciarelli 1994). This echoes Schön's (1983) attempt to give a practitioner's instinct sufficient rigour that it may be incorporated in academic discourse. An intangible instinct must be valued, its “distinction must be allowed if we are to explain how design outcomes are successful in some contexts, not in others, given roughly the same design task” (Bucciarelli 1994).

What practical implications does this have for my research? Bucciarelli's (1994) is an approach to design, to engineering, and as such has lessons for the implementation of my ideas. An object world is a model, constructed by an engineer either in a computer, or using a prototype, or simply in their head. It should satisfy all available observations and evidence, but even within those constraints several possible object worlds may be valid depending on the perspective of the individual engineer. An object world is not reality, it is only ever an approximation of reality, skewed by personal bias. As Schön (1983) would put it, there is creativity in framing the problem. Engineers should be confident in their object worlds – for if they themselves do not believe in them why should they expect others to – but must always recognise there is the potential for their object world to be invalidated: an object world can never be absolutely definitive, or 'correct'.

It will benefit my research to anticipate, and therefore be ready to recognise, what some of these

differing object worlds may be. For example there may be practitioners for whom the ability to vary the layout of the User Interface (collapsible panels, tabbed sections, and so forth) is of paramount importance. Such practitioners will want this ability exposed in a well-defined, pluggable fashion rather than being left to them to extend and override pieces of my code in an ad hoc way. Alternatively there may be practitioners who focus on having a rich mixture of different User Interface components (date pickers, map controls, graphing widgets and so forth) and will expect this capability to be a first class, pluggable abstraction. Finally, there may be engineers who identify most closely with the back-end, wanting a clear approach to combining different technologies in their architecture.

Perhaps more than anything Bucciarelli's is a call to humility – a reminder that, when designing, multiple viewpoints are valid. An engineer should always remain open to interpreting the object worlds of others, no matter how alien they may at first seem, in pursuit of the lessons they provide. Not only can the object worlds of others afford us valuable insights, they are also an important way to validate our own object worlds. As Bucciarelli (2002) puts it “the act of producing an artefact helps bring the diverse object world perspectives into coherence”.

3.4. Ethical Issues

There were no ethical issues related to the theoretical portion of my research. Those portions requiring interviews, adoption studies and case studies were de-identified, non-personal and non-intrusive. They were classified as Low Risk by the UTS Criteria (UTS 2008). This required completion of the corresponding UTS Human Research Ethics Committee Approval Form and approval from the ethics committee, which was granted. No other ethical issues arose during my research.

4. Action Research: Alpha Cycle

This chapter covers my first Action Research cycle, which ran from Q1-Q4 2008.

A critical aspect of Action Research is that each successive cycle builds on the previous, with reflections from one being used to drive planning for the next. This is, as Brooks (1995) observes, “an acceptance of the fact that as one learns, one changes the design”. Naturally however, the cycles must start somewhere. For input to this first cycle I was fortunate to be able to draw on my own experiences as an industry practitioner. I reflected on my existing understanding of the gap in the literature, using lessons learned and scenarios from current and previous projects. In particular, my observations on the amount of repetitive code and its impact on time to build, time to maintain, and bugs encountered. After all, it was those observations that led me to undertake my thesis in the first place.

4.1. Planning

As noted in section 3.2.1.4, reflection is the part of the Action Research cycle that adds rigour to the research. To reflect is to focus on the 'why', rather than simply the 'what' or the 'how' of the project. In the sections that follow, we will explore each Action Research phase with respect to the reflections undertaken during that phase.

4.1.1. Reflections During Planning

As Schön (1983) identifies, reflection during planning happens quickly. It takes place “in action” on short “action present” time scales and therefore often lacks third-party input. It relies instead on the judgement of the practitioner to “know in practice” based on evaluation against intuitive criteria. Nevertheless, it is important to make conscious this “reflective conversation with the situation”, so as to surface issues and better frame the problem. This section looks at some of the more significant reflections that occurred during the planning phase.

4.1.1.1. Naming the Project

The naming of a software project is at once inconsequential and highly important. The project will function equally well regardless of its name, but a good name helps accurately convey the

purpose of the project, avoiding initial misunderstandings from its users and attracting interest from practitioners jaded to an endless stream of new products. As Bloch (2006) simply puts it: “names matter”.

Foremost I considered how the user, being a software practitioner, would approach and interact with my project. My own perspective was on applying the emerging field of software mining to allow integration with existing architectures. But as Bucciarelli (1994) cautions us my object world can be markedly different to others. Typical use cases would see the user adding the project to an existing UI form, either by dragging it visually into place or declaring it in their existing modelling language. To them the use of software mining is opaque and inconsequential. It therefore seemed most natural to begin framing the project in terms of a UI 'component' or 'control'. The standard term for such an abstraction is a *widget*.

But the widget my potential user would be dragging into their UI would be an unusual kind of widget: it would not be accepting any input or producing any output in and of itself. Rather, it would be a compound widget composed of many simpler widgets – such as labels, text boxes and list boxes – representing the fields of a domain object. It is popular to use the Greek term *meta* for an abstraction that is a composite of itself. This term has other desirable connotations for the project: the compound widget would be sourcing the fields of domain objects from software-mined *meta*data.

Therefore, after much trial and error exploring other forms of naming (for example, choosing names that are non-descriptive of the project but short and memorable, such as 'knol' or 'cucumber'), I settled on the name Metawidget for the project. A Metawidget would be a UI widget, composed of other widgets, driven by metadata about domain objects.

4.1.1.2. Technology Neutral Interfaces

As outlined in section 1.2, and delineated in sections 2.1 and 2.2, a key contribution of this thesis is to bring together two existing, but hitherto uncombined, fields. This meant a key design decision would be the nature of the interface between the software mining and the UI generation. For any UI, performance (or at least perceived performance) is important but this had to be weighed against the diversity of front-end and back-end architectures Metawidget intended to support. As I researched more and more existing architectures, the interface between software mining and UI generation was iteratively refactored to become increasingly technology neutral. Ultimately it reached the point of being a Unicode string containing XML text. This is

not likely to be performant compared to a compact binary representation, but was considered unavoidable in order to reliably interpolate technologies as diverse as, say, ECMAScript (ECMAScript 2011) on the front-end and Groovy (Groovy 2011) on the back-end. This is because whilst disparate technologies generally support XML parsing, they can seldom read each other's binary serialization formats reliably.

4.1.1.3. Runtime Code Generation

Software mining is most commonly performed statically, traversing large amounts of source code and repository files to gather information. Source code generation is also commonly performed statically in industry, for example generating SQL statements from domain model definitions (CodeSmith 2009). However during planning I reflected on two shortcomings of the static approach.

First, the usefulness of static code generation diminishes over time. Consider the tasks typically required to produce a piece of code: to write it, debug it, test, update and document it. Static code generation helps with the writing but generally makes the other four worse. This is because the quality of the code is usually poor (having been generated by generic algorithms) and nobody, not even the practitioner running the tool, initially understands the volumes of code it produces. Static approaches simply automate the initial creation of the UI, leaving the practitioner to manually tweak the result. This is a short term gain – once the repetition has been introduced, it must be maintained throughout the rest of the application's lifetime. Simply automating its introduction is not of great benefit. Rather, as Hunt and Thomas (1999) point out "the trick is to make the process active: this cannot be a one-time conversion, or we're back in a position of duplicating data".

Second, there are many important properties of a system and the relationships between its parts that are only discernible at runtime. For example a UI screen may need to be adapted based on the permissions of the logged-in user, or it may need to display a polymorphic domain model entity, or its data may need to be bound to a runtime instance of a class.

These shortcomings led me to focus instead on *runtime* software mining and code generation. There were compelling precedents for this shift: the Java Enterprise Edition (Java EE) community had recently standardised on a runtime persistence architecture rather than statically generated SQL statements (DeMichiel & Keith 2006).

4.1.1.4. Useful Bounds of Generation

Many characteristics of a UI are tightly bound to its underlying domain model. For example the name, type and allowable value of any widget is constrained by the domain object which will ultimately be used to store its data. As automatic UI generation moves away from such rigidly defined characteristics, however, it rapidly becomes speculative. Determining how to display a domain object is much more subjective than determining what fields to display. Determining how to represent relationships between multiple domain objects is more subjective still. The practical usefulness of UI generation diminishes when in these areas, because the generated UI bears less and less resemblance to how it would have appeared and functioned had it been designed manually, with consideration to its specific purpose (Falb et al. 2007). Constantine (2002) strongly criticises: "the usability problems with such [speculatively generated] interfaces under most conditions of use are too numerous… will be fairly obvious to any competent usability professional… this simplistic approach to user interfaces could be dismissed as laughable".

It is, therefore, possible to delimit useful bounds to UI generation. Broadly stated: that which can be unambiguously determined from what is already rigidly defined in the application's architecture can and should be automated. That which requires a degree of interpretation should be left to the practitioner and their existing gamut of UI development techniques. Automatic UI generation can be seen as a way to augment the UI development process, rather than replace it.

An approach that automatically generates the entire UI distances the practitioner from the underlying framework. Most UI frameworks provide a rich set of services such as validation (see 2.1.2.2), data binding (see 2.1.2.3) and navigation flow control (see 2.1.2.4). They further support third-party widgets (see 2.1.2.1), and there is usually a rich marketplace of components such as charting and data visualisation plug-ins. When a UI generator restricts access to native APIs it precludes practitioners from using these capabilities. By overlapping a UI framework, the generator effectively puts itself in competition with it: the generator needs to both generate UIs and match, feature for feature, other frameworks. Those generators that broaden their focus, for example to task-based modelling, become increasingly vulnerable to this problem. Their scope overlaps the domain of rule engines, BPM systems and even traditional programming languages, all of which they must match in features, stability and documentation.

Having identified the useful bounds of generation, ways to automate the generation of domain model widgets whilst providing high fidelity integration with existing UI and other frameworks

can be explored. This significantly increases the practical usefulness of the UI generator because, whilst much of a UI can be determined from the underlying application architecture, there will always be attributes that are not embodied elsewhere (for example the font to use within a text field). It is important the UI generator does not restrict the practitioner's ability to control such attributes.

A technology that successfully limited itself to useful bounds would present less as a conventional UI generator, more 'just another tool' in a practitioner's arsenal. Its aim would be to generate the same UI the practitioner would previously have constructed manually. This would be more analogous to a tool such as an ORM than to a pattern such as Naked Objects (Pawson 2004). We might term it an Object Interface Mapping tool: an OIM. The symmetry with the acronym ORM would be intentional, as an ORM and an OIM could be used in tandem to automate the mapping of an Object (the 'O') from Relational Database (the 'R') to User Interface (the 'I').

4.1.1.5. Layouts

After the machinations of software mining the metadata and constructing appropriate widgets, the final step for any UI generator is to arrange the components on the screen. This is perhaps the most intractable issue in UI generation. Certainly, it is one of the most visible if not executed well. It significantly detracts from the practicality of automated generation if it in any way compromises the final product in usability, or even in aesthetics (Myers, Hudson & Pausch 2000). This realisation exposes a myriad of small details around UI appearance, navigation, menu placement and so on. It leads Raneburger (2010) to conclude “it is hard to automate all aspects of UI engineering and we even think that it is unrealistic in the near future”.

As discussed in the previous section, my approach largely sidesteps the issue by sharply restricting the bounds of its generation. Specifically, it does not attempt to generate the entire UI. Rather, it focuses on generating only a small piece of it – the 'inside' of each page, the area around the fields themselves. Ultimately, this is the only piece that is actually constrained by the back-end architecture. The UI appearance, navigation, menu placement and overall usability are far more device-specific, not to mention specific to the aesthetic taste of the UI designer. I explicitly keep these out of scope. Multiple Metawidgets can be positioned on the screen in arbitrary formations to suit the designer's needs.

As testament to how impractical generation of an entire UI is, even after restricting UI

generation to just the area around fields we find there is still a formidable degree of variability. Fields may typically be arranged in a column, with the widget on the right and its label on the left. But other times the practitioner may want two or three such columns side by side. If so, they may need some widgets – such as large text areas – to span multiple columns. Or they may abandon columns altogether and want the fields arranged in a single, horizontal row. Furthermore, it is not difficult to posit other arrangements, such as right-to-left arrangements for the Arabic world. It is important to accommodate this variety if the generator is to achieve the exact look the practitioner desires. If it cannot achieve that exact look, the practitioner is compromising usability – the most determining factor of a UI – for the sake of automatic generation.

I proposed to address this characteristic of supporting multiple ways to arrange widgets by defining pluggable layouts. Metawidget would have a `Layout` interface and ship with a number of implementations of this interface. But it would also be straightforward for a practitioner to add their own.

Together the approaches of technology neutral interfaces, runtime code generation, useful bounds of generation and pluggable layouts conclude the significant reflections that occurred during my first planning phase. We turn now to the acting phase.

4.2. Acting

This section records the 'act' phase of my first Action Research cycle. As with the previous section, I will primarily document the work with respect to the reflections undertaken during the phase.

4.2.1. Reflections In Action

As with reflections during planning, reflections in action happen on short time scales. They are highly dependant on the intuitive “knowing in practice” of the practitioner (Schön 1983) – though they are not entirely subjective, being grounded in the accumulated knowledge of the field. Knowledge such as theories of Object Oriented design and separation of concerns; a vocabulary of design patterns; an appreciation of elegance, coherence and performance. This section recaps the most significant reflections that occurred when putting my planning into action.

4.2.1.1. Widget Builders

When starting a new project, my practitioner's instinct is to embody its essence in some central touchstone: a class or group of classes. For Metawidget, this started out as an actual class called Metawidget. The Metawidget class delegated to various software mining classes to inspect the back-end, and to various widget builder classes to choose and configure widgets for the front-end.

This seemed fine in principle, but as discussed in section 4.1.1.1 there was also the expectation of having a well-defined native UI component that could be added into existing UIs. For graphical interactive specification tools, this meant a widget that appeared in the component palette. For model-based tools, this meant an XML tag. Almost all UI frameworks require native components to extend their base class (as opposed to implementing an interface). Extending a base class is problematic in languages such as C# (Hejlsberg 2006) and Java (Gosling 2005), because the languages deliberately only support single inheritance. If a component must extend a UI framework base class, it cannot also extend a Metawidget base class.

This led the design to having a Metawidget class, software mining classes, widget builder classes *and* native framework classes. The latter two were closely coupled, being tied to the same UI framework, and during early prototyping I collapsed them. But it still felt problematic that there was a separate Metawidget class. It felt like this was stipulating a Metawidget-first perspective, which was certainly my own perspective, but I realised my users would more naturally be coming from a UI component-first perspective (Bucciarelli 1994). I reflected this problem existed only because of the lack of multiple inheritance – what I wanted was to extend both a Metawidget base class and a UI framework base class at the same time. Having re-framed the problem as one of multiple inheritance, new possibilities presented themselves. Specifically, a common compromise to multiple inheritance in single inheritance languages is to use mixins (Bracha & Cook 1990). The native framework class could now become the practitioner-facing 'face' of Metawidget, with the original Metawidget class relegated to a mixin behind-the-scenes.

Re-framing the problem as one of multiple inheritance further presented the reverse notion that perhaps Metawidget, not the UI framework, could be interface-based. This notion would not help with re-using code (the mixin would still be needed for that), but it *would* satisfy the instinct to have a touchstone for the central essence of Metawidget – that of inspecting domain objects, choosing UI widgets, binding front-ends to back-ends and so on. In practice, however,

this essence turned out to be elusive. Implicit to Metawidget's goal of integrating with existing front-ends is not to overlap their behaviour. If a UI framework already provides, say, a data binding solution Metawidget should leverage it rather than impose some parallel API. Unfortunately my investigations revealed very little consistency in the feature set of different UI frameworks: some had standardised hooks to domain objects, some had native binding, some had native validation, some had none of these. What little commonality there was (none provided automatically choosing simple widgets for primitive data types, for example) did not capture any meaningful essence. I decided a base Metawidget interface was not justified.

4.2.1.2. Guided Software Mining

As discussed in section 4.1.1.3 software mining is generally performed statically, with generous performance constraints. Report generation generally emphasises accuracy over timeliness, so it is reasonable for static software mining to exhaustively traverse all domain objects and configuration files looking for metadata. Implementing such an unconstrained approach for runtime UI generation, however, would result in unacceptable performance. It would be extremely wasteful, as most of the metadata gathered would not be needed for any given UI screen. I reflected that Metawidget's goal of runtime UI generation meant the software mining could not proceed unguided.

This surfaced the issue of whose responsibility it was to do the guiding. The best placed abstraction seemed the native-framework extending class, discussed in section 4.1.1.1. This was the component that was placed on a concrete UI page, so it knew much about the type of UI and the domain objects it needed to render. It could not only give the software mining a starting point within the graph of all possible domain objects, it could stop the software mining when it had gone 'far enough'.

For example figure 7 shows a simple UI screen for editing a `Person` object. This screen could know to guide the software mining to only look for `Person`-related metadata. Such guidance would prune large portions of the domain model graph, ignoring such entities as, say, `Company` or `Invoice`. The screen does need, however, a nested `Address` entity so it could instruct the software mining to drill down into `Address`-related metadata but no further.

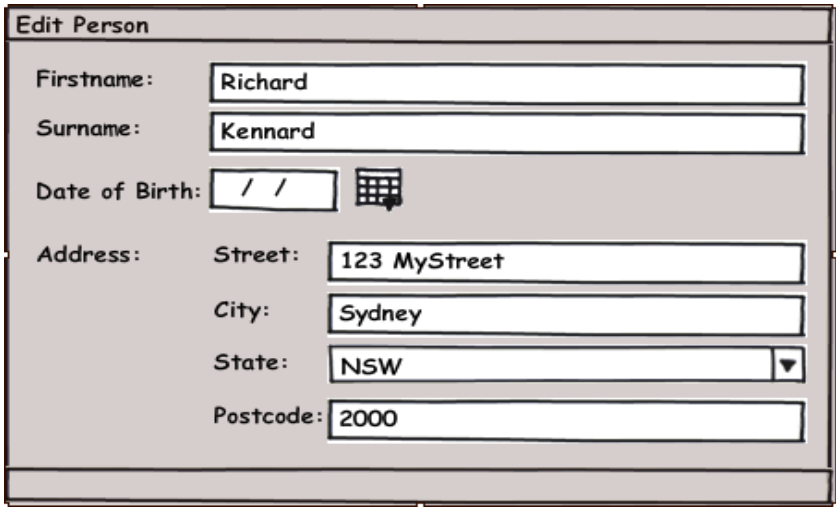


Figure 7: The UI drives the software mining

4.2.1.3. CompositeInspector

Implicit to Metawidget's goal of using software mining to inspect existing architectures was supporting a variety of back-end technologies. This in turn implied supporting a pluggable software mining layer for inspecting different architectures. Because application architectures invariably combine several technologies, this further implied plugging together multiple inspectors to mine different facets of the back-end.

My initial practitioner's instinct was to have the mixin described in section 4.2.1.1 maintain and iterate over a configurable list of inspectors. As each inspector returned the result of mining its particular facet of the back-end architecture, the mixin would collate them into an overall result. This overall result was a Unicode string of XML, as discussed in section 4.1.1.2. The collation algorithm itself was a little ambiguous. Many behaviours were self-evident: that inspection results be collated so as not to 'surprise the developer' (Bloch 2006), that ordering of business entity fields was preserved, and so on. But there were some behaviours that could reasonably be implemented either way: should later inspectors in the list override earlier ones or vice versa, should new fields discovered by later inspectors be added before or after existing fields, and so on.

My initial approach was simply to use my best judgement as a practitioner. To recap the collation algorithm described in section 2.2.5.5, suppose one inspector returns the following result:

```
<inspection-result>
   <entity type="Person">
      <property name="name" type="string"/>
      <property name="age" type="integer"/>
   </entity>
</inspection-result>
```

Then another inspector, say one inspecting validation metadata, returns this result:

```
<inspection-result>
   <entity type="Person">
      <property name="age" minimum-value="0" maximum-value="150"/>
   </entity>
</inspection-result>
```

Collated, the practitioner would probably expect these two inspection results to appear as:

```
<inspection-result>
   <entity type="Person">
      <property name="name" type="string"/>
      <property name="age" type="integer" minimum-value="0" maximum-value="150"/>
   </entity>
</inspection-result>
```

The rules of the collation algorithm would simply be 1) combine entity nodes based on their type attribute (which is unique across all entities), 2) combine property nodes based on their name attribute (which is unique within an entity) and 3) copy all other attributes verbatim.

Now suppose a third inspector returns this result:

```
<inspection-result>
   <entity type="Person">
      <property name="name" length="50"/>
      <property name="age" minimum-value="1"/>
      <property name="address" type="Address"/>
   </entity>
</inspection-result>
```

One could posit a rule 4) if an inspector later in the list returns an attribute with the same name but a different value, it should override the earlier result. One could also posit 5) any property nodes found by later inspectors that do not match ones found by earlier inspectors should be added at the end of the entity. This would produce:

```
<inspection-result>
   <entity type="Person">
      <property name="name" type="string" length="50"/>
      <property name="age" type="integer" minimum-value="1" maximum-value="150"/>
      <property name="address" type="Address"/>
   </entity>
</inspection-result>
```

This seems a reasonable result. But rules 4 and 5 are quite arbitrary. One could equally well posit 4) different values should be ignored, preserving the earlier result and 5) new property nodes should be inserted at the start of the entity. So while the collation algorithm worked, enforcing such arbitrary choices felt uneasy.

On the other end of the spectrum from worrying about internals of the algorithm was the pathological case where no collation algorithm was required at all. For example some domain models may be completely codified in an XML file that only requires a single XML inspector. In this scenario, the mixin's list of inspectors would be a list of 1 and its collation code would never need executing. Such inelegance also felt uneasy.

Stopping and surfacing the issues around this unease, there seemed two points: that the collation algorithm itself should be pluggable, to allow practitioners to choose variations on the ambiguous behaviours (i.e. rules 4 and 5); and that the mixin should more elegantly support the single inspector scenario. This reframing allowed a new possibility: what if the list and the collation algorithm were themselves another inspector? This would be an 'inspector of inspectors': externally it would appear as a single inspector, internally it would delegate to a list of inspectors and collate their results, as shown in figure 8.

This would mean the mixin need only deal with a single inspector, the simplest case. It would also mean practitioners could write their own 'inspector of inspectors' with their own collation algorithm, if required. Belying my object world, I initially named this inspector a 'meta inspector'. But consulting Design Patterns (Gamma et al. 1995) suggested the more conventional term would be 'composite inspector': “Composite lets clients treat individual objects and compositions of objects uniformly”.

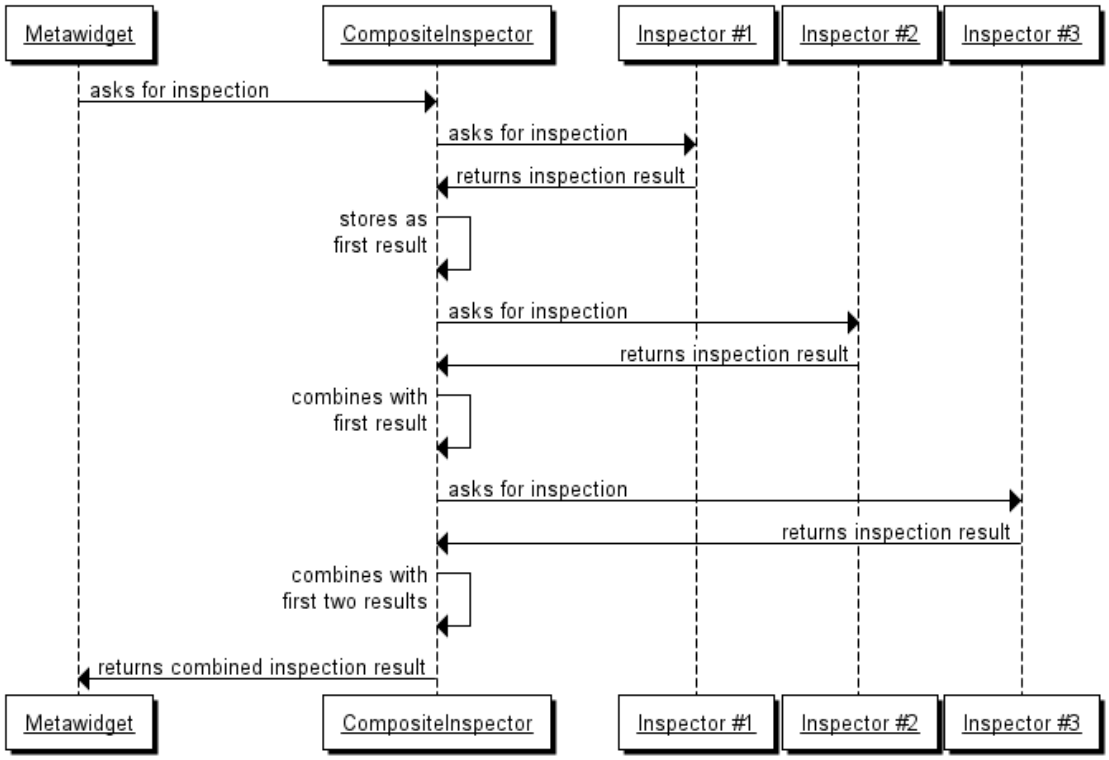


Figure 8: CompositeInspector collates results and appears as a single Inspector externally

This approach of refactoring all inspection and collation code into a pluggable architecture, away from the mixin, was later validated when I considered remote inspection. In many architectures metadata is only available in silos. For example persistence and XML serialization metadata may only be accessible on the back-end, layout metadata may only be accessible on the front-end. This would require separate rounds of remote inspection, one for each silo, and then collation. Such distributed arrangements of software mining would require intricate co-ordination between inspectors, possibly with several sub-collations before the final collation. It seemed appropriate this complexity should be hidden from the UI generation mixin.

4.2.1.4. Papers

An important part of the 'act' phase of an Action Research cycle is to develop something which inspires feedback to drive the 'observe' phase. For industry, this generally means implementing and documenting software for practitioners to try. For the research community, this generally means writing and publishing papers for researchers to review.

For this alpha cycle I produced an introductory paper that framed Metawidget in the context of

the fields of UI generation and software mining (echoing chapter 2 of this thesis), enumerated my design objectives and touched briefly on implementation progress to date. This paper was accepted and published at the 7th International Conference on Software Methodologies, Tools and Techniques (Kennard & Steele 2008). Producing the paper was itself a revealing exercise in reflection, as reframing intuitive decisions and preliminary ideas into a structured, formal context requires a level of rigour and candour not generally practised in the brief action present that characterises reflection in action.

Whilst the paper was accepted, one reviewer had reservations: “the paper is based on the premise that the disadvantage of current UI generation approaches is that they require 'restating information'”. This premise is discussed in section 2.1.3. But the reviewer felt “the reason for re-specification is that the display fields typically have different length and other attributes than the database schema specification. This is a useful feature, not a 'disadvantage'”. The reviewer's phrase “different length” is ambiguous. Possibly they were referring to the visual width of the field, rather than the length of data it could hold, in which case it may of course differ from the maximum length in the database schema. If this is what the reviewer meant, then I had failed to accurately convey my meaning in the paper. However the reviewer may also have been referring to the length of data. This is indeed what my paper meant, and in this case I did not believe it was a useful feature. Rather, I believed a mismatch between UI lengths and database lengths was a common source of application defects. This was certainly my own experience as an industry practitioner. However on reflection I had to concede I lacked evidence with which to bolster my claims: they called for further observations, as we shall come to in section 4.3.1.1.

4.2.1.5. Experiments

Given the inspiration for this thesis was my own experience as an industry practitioner, I had resources to draw on to verify Metawidget's effectiveness. In particular, Metawidget could be integrated into products I developed for my clients. Part of my strategy for the alpha 'act' phase was to be a user of Metawidget myself – verifying that the project at least satisfied my own needs before releasing it to a wider audience. To “eat one's own dog food” (Microsoft 2000) in this way gives perhaps the most honest form of reflection, though it means being mindful not to align the work too much to one's own requirements.

The aim of the experiments was to assess, on a limited scale, how effectively Metawidget could be integrated into industry applications. Both experiments involved approximately 50 domain

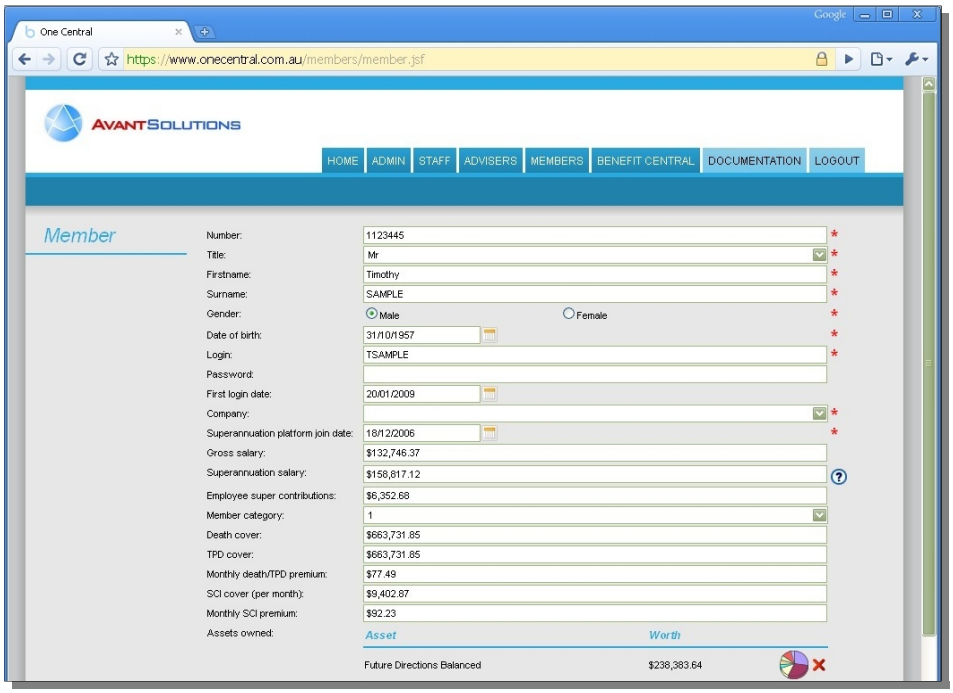
model entities and 150 UI screens. Whilst ideally Metawidget would scale linearly, it remains an open question how these experiments apply to larger applications. This will be addressed in later adoption studies.

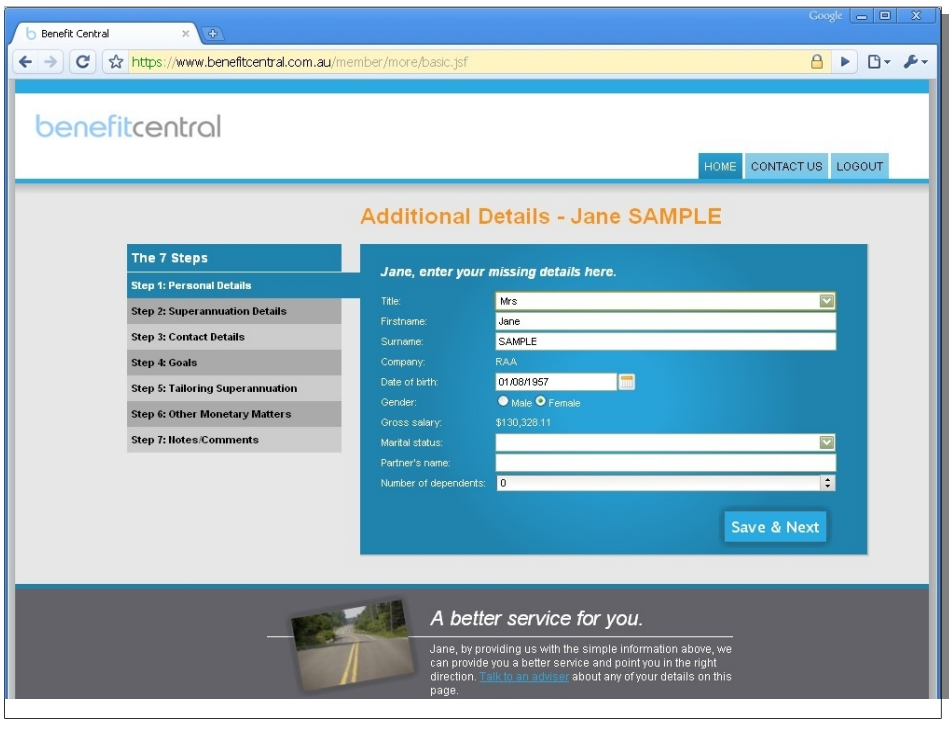
4.2.1.5.1. Experiment 1

Company: Financial company providing tools to financial advisers.

Application: Web-based application

Technologies: Java, JSF, JPA





4.2.1.5.1.1. Synopsis

The application was concerned with the collection and reporting of financial information for members of Australian superannuation funds. There was a particular focus on *relevancy* of reporting, so the application collected very detailed information which required approximately 50 domain model entities, including superannuation fund members, their employers, their superannuation plan and other details.

The typical infrastructure required to support 50 domain model entities would be considerable. Code to support persisting the entities to a database, validating them, presenting them to the user, reporting on them and importing/exporting them to external systems would run into tens of thousands of lines – all requiring testing, debugging and documenting. As an independent consultant with limited resources, I therefore had a strong motivation to minimise the amount of infrastructure I needed to manually build. Traditionally, this involved the use of ORMs, XML serializers, validation frameworks and Web frameworks. The intent of this experiment was to reduce that amount of manual infrastructure still further.

The application called for approximately 150 UI screens, and the client had specific graphic design requirements for many of them, including different look and feels for different user roles. This presented an excellent proving ground for Metawidget: a medium-sized application, with both a varied set of fields and varied presentation styles. Trying to replace my existing UI code (which was inevitably tightly woven into the application) with Metawidget code that was more external and more generic, whilst still retaining support for all existing field types and presentation styles and being mindful not to harm application performance, taught me much about Metawidget's strengths and weaknesses.

A particular insight was how pluggable Metawidget needed to be. Industry applications often bring together many different technologies, but each generally performs its work in isolation. For example persistence frameworks do not concern themselves with UI frameworks. But Metawidget's software mining requires it to be across as many technologies as possible on the back-end, and its goal of being a general purpose UI generator requires it to integrate with many front-ends. Dimensions such as different layouts, different third-party component suites and different validation engines all needed to be pluggable, requiring high-levels of abstraction and generalised interfaces between many areas of functionality.

Integration was ultimately successful, and I was able to replace significant portions of my

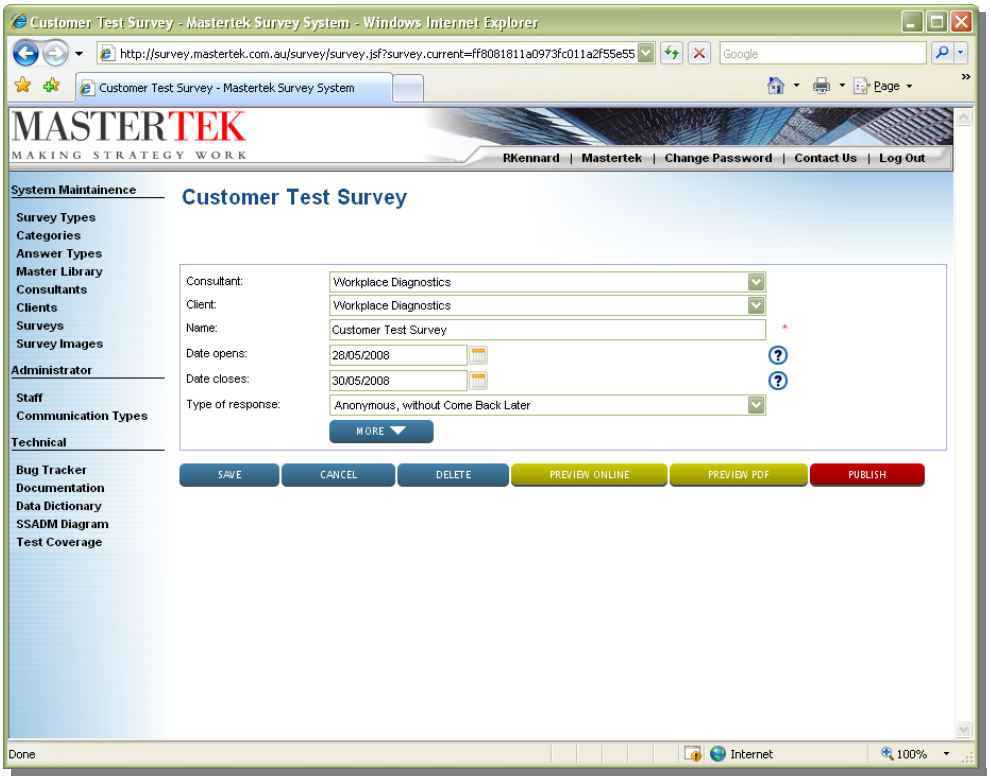
existing UI code with Metawidget. This had a very positive effect on my productivity going forward. Adding or changing fields in any of the 50 domain model entities only required changing the domain model entity itself – the UI was updated automatically and reliably, with less scope for error and less requirement for debugging.

4.2.1.5.2. Experiment 2

Company: Strategic company providing tools to help businesses gather performance data in relation to key stakeholders.

Application: Web-based application

Technologies: Java, JSF, JPA



4.2.1.5.2.1. Synopsis

The application was concerned with the creation, execution and reporting of employee and customer opinion surveys. The structure of the surveys needed to be particularly flexible, requiring approximately 45 domain model entities to capture the necessary functionality, including text-based and multimedia-based questions, open-ended and Likert-scaled (1932) answers and matrices. There were also associated entities such as consultants, employers and survey respondents. As with Experiment 1, typical infrastructure for this application would be considerable, so integrating Metawidget was an attractive option. Again like Experiment 1, doing so taught me much about using Metawidget in industry scenarios.

Particularly challenging was this project's focus on polymorphic behaviour. The system was

required to be extensible, so it included base classes and subclasses for different question types (text questions, matrix questions, etc.) and answer types (text answers, multiple choice answers, etc.). This impacted the UI, with screens themselves required to morph on demand. As shown in Figures 9 and 10, when the user made different selections from different underlying data types, different fields were required to appear and disappear.

Integrating Metawidget therefore required a fine-grained balance between automatic generation and manual specification. Some areas of each screen needed to be manually-specified to achieve the necessary experience, but it was desirable the rest be automatically generated so as to reduce code and ongoing maintenance. This led to a key requirement of being able to selectively override portions of the automatic generation.

An instinctive first approach to overriding Metawidget-based generation was to introduce a Metawidget-specific mechanism. For example, for those UI frameworks that were model-based, such as HTML...

```
<html>
   <metawidget name="question"/>
</html>
```

...one might posit a Metawidget-specific tag attribute:

```
<html>
   <metawidget name="question" exclude="answerType"/>
</html>
```

This worked for simply excluding automatically generated fields, but it did not scale well to *overriding* them with different content. The tag attribute became cumbersome:

```
<html>
   <metawidget name="question" override="answerType=&lt;select&gt;"/>
</html>
```

While reflecting in action, I realised what I was trying to do was specify fragments of content. Reframed in this way, I realised the UI framework already had a native mechanism for specifying fragments of content: child tags. Child tags are fragments of arbitrarily deep content, scoped to the parent, that are typically rendered verbatim. But by not rendering them verbatim, by instead using them as cues, it was possible to selectively instruct automatic generation. By supplying special 'stub tags'...

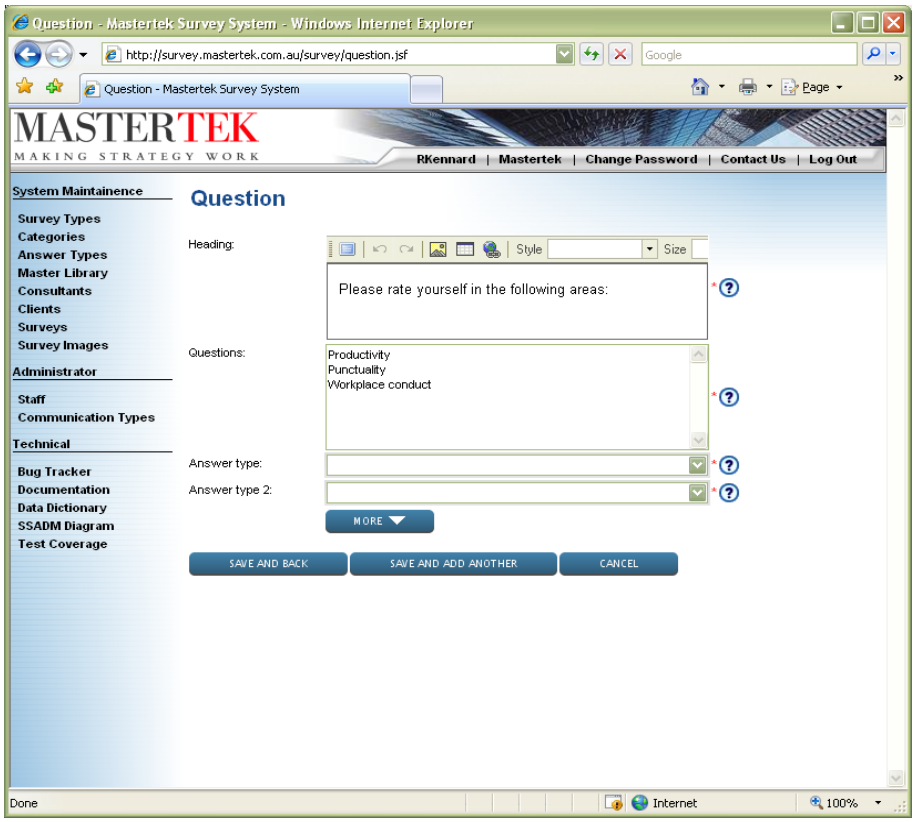


Figure 9: Question screen before answer type selected

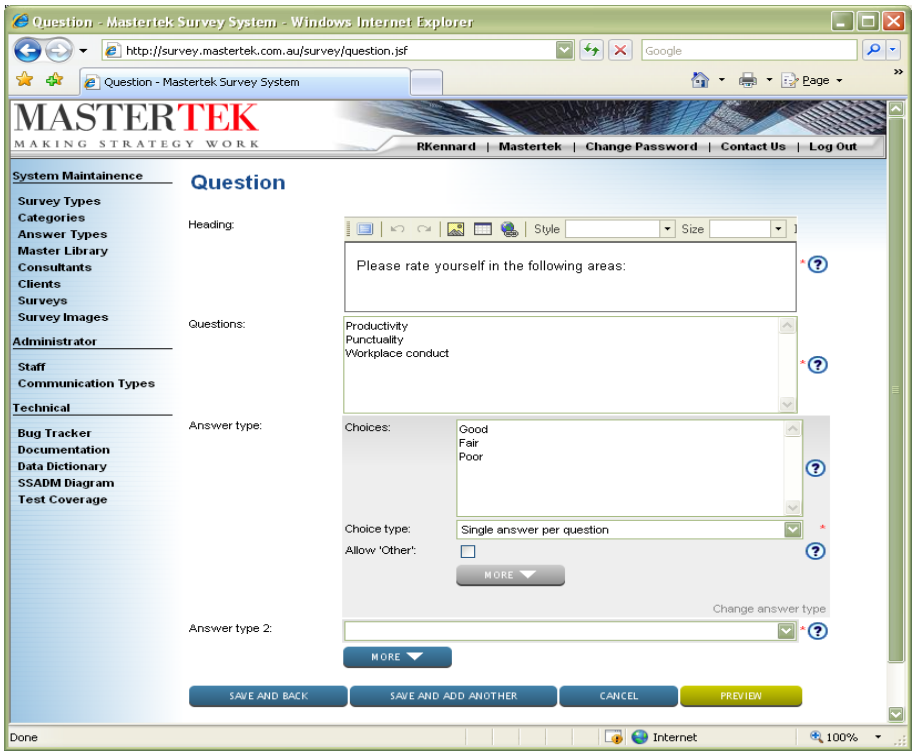


Figure 10: Question screen after answer type selected

```
<metawidget name="question">
   <stub name="answerType"/>
</metawidget>
```

...it was possible to 'stub out' (exclude) fields from the generation process. Furthermore, the stubs could contain arbitrary content...

```
<metawidget name="question">
   <stub name="answerType">
      <select name="answerType">
         …
      </select>
   </stub>
</metawidget>
```

...allowing manually-specified content to override automatic content on a fine-grained basis. A similar mechanism applied to interactive graphical specification tools, which allowed visually dropping widgets inside parent widgets. Both approaches reused existing framework features, which had the advantage of being familiar to the user as well as integrating with existing UI framework tools.

4.2.2. Action Outcomes

There is a tendency when writing a thesis that much of the work *behind* the thesis becomes subsumed. It may serve as a vehicle for generating observations and validating ideas, but may otherwise go unmentioned. In my case, however, many of the sections to come will concern themselves with feedback and reflections on specific pieces of the implementation. Therefore, whilst documenting the software at a detailed, technical level is inappropriate, it will be helpful to briefly outline the architecture of the solution as it stands at the end of the alpha 'act' phase. This will provide context for the observations and reflections that are to follow.

4.2.2.1. Screenshots

We begin with a screenshot from one of the sample applications distributed with Metawidget. This screenshot, figure 11, visually embodies the idea that Metawidget does not try to 'own' the entire UI (see 2.1.1.3): the look and feel of the controls, the white and faded blue background, the icons of the man and woman, and the choice of UI platform (Java Swing in this case) are all implemented using the native platform APIs and are not part of Metawidget. The choice of

mechanism to navigate from summary screen to detail screen (in this case a pop-up modal dialog) is also not part of Metawidget.

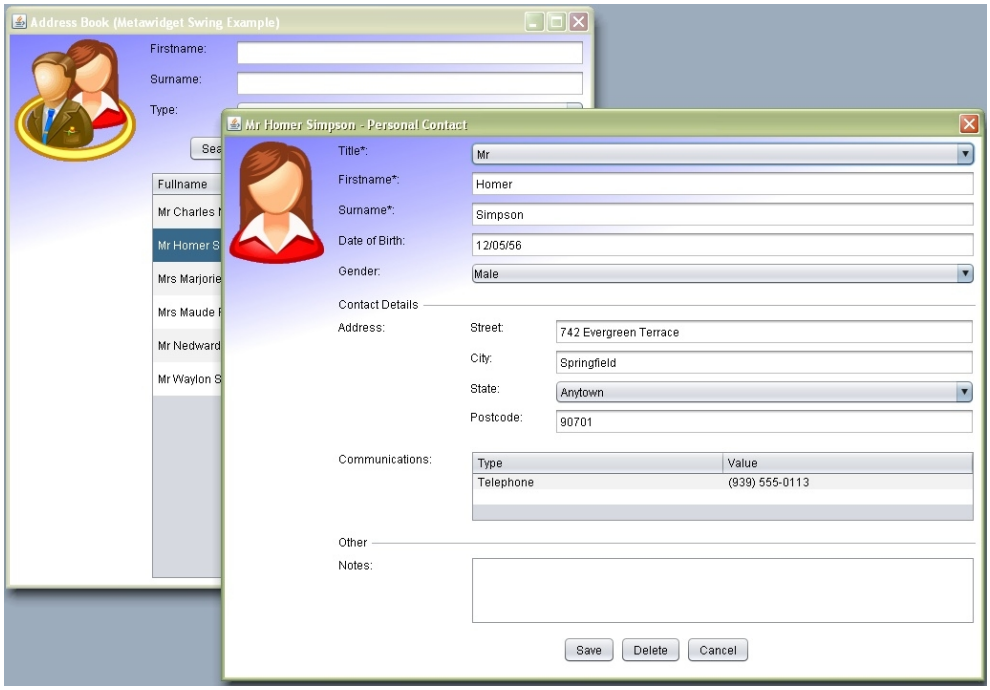


Figure 11: Metawidget does not try to 'own' the UI

The Metawidgets themselves are highlighted in figure 12. There are five Metawidgets: two are used on the summary screen (shown behind the pop-up dialog) – one to generate the three search fields at the top and one to generate the search buttons (partially obscured in the screenshot); the other three Metawidgets are used on the detail screen – one to generate the majority of the fields, one to generate the action buttons at the bottom, and an embedded Metawidget to generate the address fields. This latter Metawidget is not added by the practitioner. Rather, it is generated dynamically as the default behaviour whenever a parent Metawidget (the one rendering the fields `Title`, `Firstname`, `Surname` and so forth) encounters a data type (in this case, `Address`) it does not know how to represent with a single UI widget.

The use of five Metawidgets for such a simple UI screenshot underscores one of the novelties of my approach: Metawidget treats UI generation not as a heavyweight, all-encompassing process but as small, lightweight fragments of UI. Metawidget *expects* practitioners to liberally scatter small amounts of UI generation throughout their application, applying automatic generation to those parts of the screen where it is appropriate, using traditional techniques where it is not.

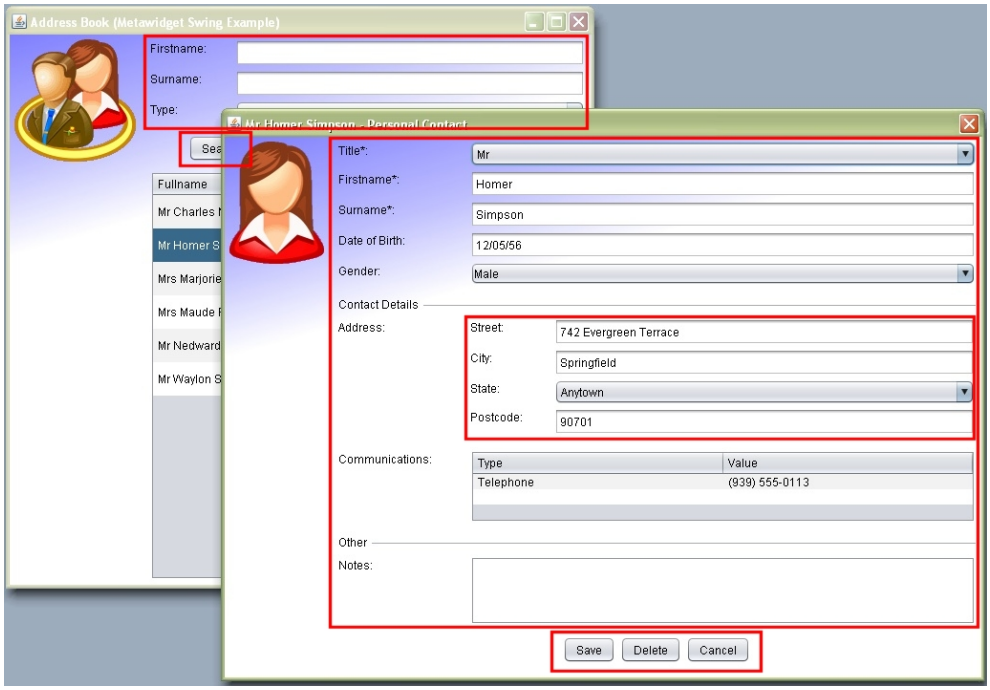


Figure 12: Five Metawidgets are used in the UI

Having established where Metawidget is used, we turn now to how it works. This requires a brief exposition into the code itself.

4.2.2.2. UML

Figure 13 shows the basic Metawidget architecture in UML form. Note this figure is much simplified: the actual alpha release of Metawidget, as of Q4 2008, had over 150 classes supporting seven front-end UI frameworks (Android, GWT, JSF, JSP, Spring, Struts and Swing) and nine back-end frameworks (Bean Validation, Apache Commons JEXL, Apache Commons Validator, Groovy, Hibernate, Hibernate Validator, Javassist, JPA, Swing AppFramework) and encompassed over 9,000 lines of code. This was a significant engineering effort. It was important because, given my thesis is research on practice, the nature of Action Research is such that authentic practice is the best vehicle. However there was concern such a significant effort could in other ways detract from my research. In particular, the time required for bug fixing and general maintenance could reduce the time available for thorough reflection. To reduce this risk, I drew on my experience as an industry practitioner: I supplemented the alpha release with an additional 4,000 lines of unit tests and 2,000 lines of example applications. These combined to provide over 85% test code coverage as a safeguard against serious defects and regressions. That said, such engineering complexity is a prerequisite to, rather than a part of, this thesis. The

UML figure shows only the relevant points for the purpose of the sections to follow.

We begin by considering the `SwingMetawidget` class. This is a natural entry point for a practitioner studying Metawidget because `SwingMetawidget` is the native component (in this case, native to the Java Swing platform) that the practitioner drags and drops on to their UI in their IDE. The UML figure 13 includes a subset of the public methods exposed by `SwingMetawidget`, other methods being omitted for clarity.

The first design point to observe is that the majority of `SwingMetawidget`'s methods are concerned with setting pluggable components. For example `setLayoutClass`, `setPropertyBindingClass`, `setActionBindingClass` and `setValidatorClass` all defer to interfaces behind which are multiple implementations. The UML depicts a subset of these implementations. For example `PropertyBinding` can be implemented via either `BeanUtilsBinding` or `BeansBinding`, depending on the practitioner's chosen architecture. Similarly, validation can be provided by either `JGoodiesValidator` or `InputVerifierValidator`. This high-level of pluggability reflects the emphasis on integrating with existing architectures, as discussed in section 2.2.4.

Similarly, the software mining aspect of the project can be seen through the `setInspector` method. The UML depicts a handful of the inspectors available in the alpha release, each targeting different subsystems as discussed in section 2.2.5. `SwingMetawidget` uses the configured inspector to inspect the domain object specified in `setToInspect`. This may include running multiple inspectors and collating their results using a `CompositeInspector`, as discussed in section 4.2.1.3. Importantly, it is the Metawidget that drives the inspector, not the other way around, as discussed in section 4.2.1.2. The inspector then returns a platform-neutral string, as discussed in section 4.1.1.2.

The internal processing of the software mining result is deferred to a `MetawidgetMixin` class. As discussed in section 4.2.1.1, in order for Metawidgets to be native UI components they typically need to inherit a base class from their native framework. This is unfortunate because there is potentially a lot of inheritable functionality across Metawidgets. In order to reuse this functionality, and because Java does not support multiple inheritance, I employ a common workaround for multiple inheritance in single inheritance environments – the mixin. This has proven an effective abstraction. The mixin's implementation refactors a significant amount of code, yet the same mixin is successfully reused between desktop, Web and mobile Metawidgets

(not shown in the UML diagram).

The mixin determines if the result of the software mining can be represented by a single UI widget (`buildSingleWidget`) or requires arranging multiple UI widgets (`buildCompoundWidget`). For example, for the `Address` data type in section 4.2.2.1 there may be a native widget available on the target platform suited to handling address input. If so, Metawidget will use that. If not, the `Address` object will be decomposed into, say, `street`, `city` and `postcode` – which can be represented using native text box widgets. For each widget in the compound case, the mixin first checks whether the practitioner has added ad hoc child widgets to the Metawidget to override default generation (`getOverriddenWidget`), as described in section 4.2.1.5.2. If so, these stub components are used in preference.

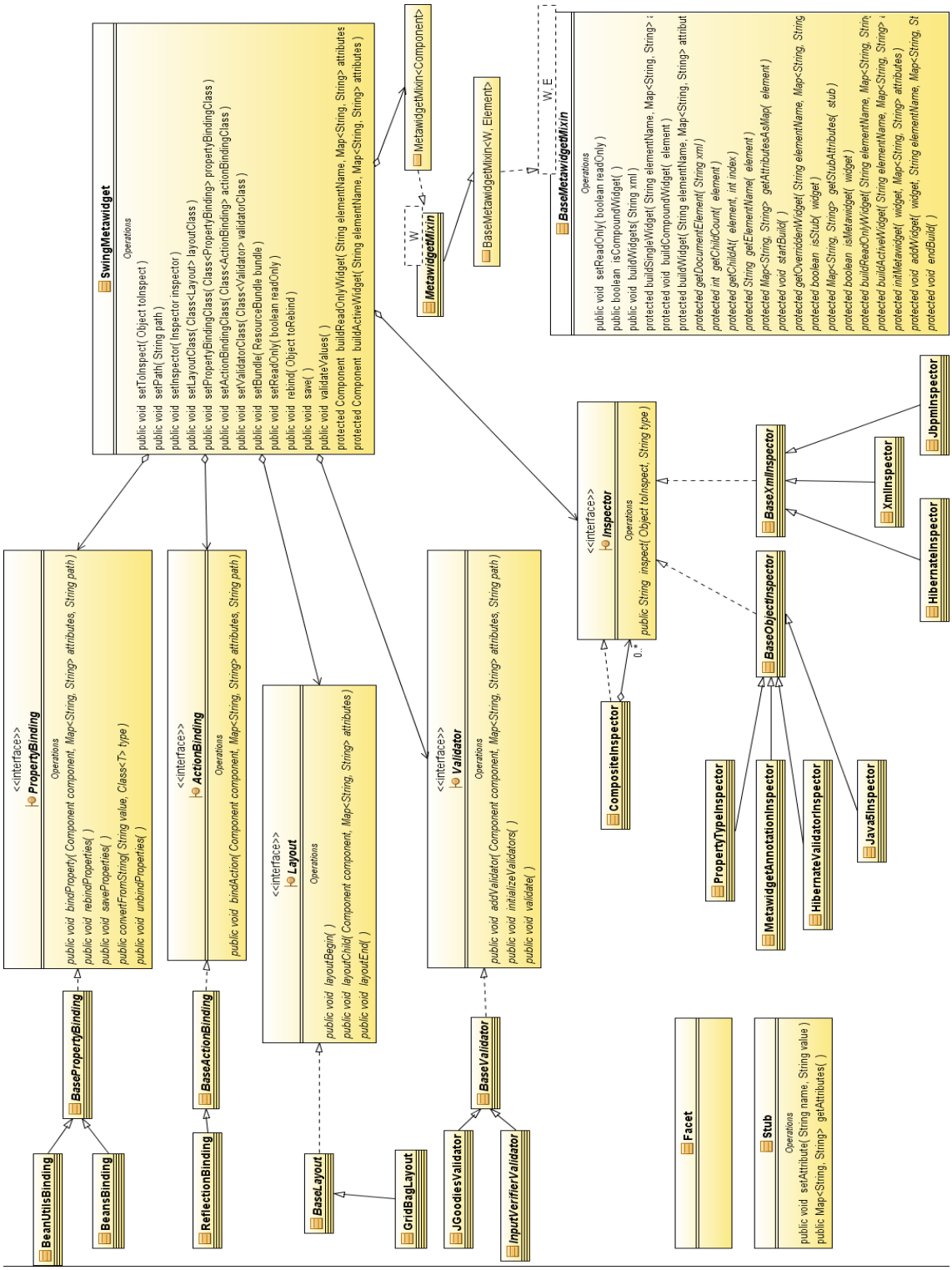


Figure 13: UML class diagram of Alpha Action Research Cycle

The chosen UI widgets are then handed back to the `SwingMetawidget`, which passes them to its platform-specific data binding, validation and other plug-ins. These attach event handlers and wrapper classes as needed. Finally, the widgets are passed to the configured layout which arranges them according to the practitioner's preferences (i.e. single column or two columns, section headings or tabbed panels etc.), possibly adorning them with other widgets (e.g. labels for field names, stars for required fields, etc.). The layout may also use `Facet` components, which can be added by the practitioner as child widgets of the Metawidget to encapsulate such features as button bars.

This, then, summarises the basic process followed by all Metawidgets on all platforms: a series of well-defined steps with numerous extensible plug-in points. This requirement for extensibility brings with it some challenges for performance, as discussed in the next section.

4.2.2.2.1. Immutability

A key goal for Metawidget's architecture was that it must accommodate, rather than dictate, different technologies. This required it be modular in many different dimensions. In turn, this required the architecture be composed not of one monolithic object (i.e. a single Metawidget class) but dozens of smaller, pluggable objects (i.e. inspectors, layouts, validators etc.).

Yet ultimately Metawidget itself is but a small piece of a larger puzzle: it is a fragment of a UI, and equally that UI is just a fragment of the overall application. There is risk in making Metawidget so fine-grained that the overhead of executing its architecture makes it perform poorly relative to acceptable metrics within the wider application. For example section 4.2.2.1 described half a dozen Metawidgets on a single screen. A practitioner may reasonably expect them to perform – in terms of speed, memory consumption and other metrics – roughly equivalent to other UI widgets such as a rich text editor or a scrolling map component.

How then to resolve this dilemma of needing dozens of pluggable objects on the one hand, yet needing a 'right sized' architecture on the other? Generally speaking, the reason dozens of objects are considered non-performant is the overhead associated with instantiating them, making copies of them, synchronizing thread access to them and deallocating them – the actual behaviour of the object is less of a concern. After all, few would argue that breaking an application into dozens of *functions* is non-performant. Indeed this is generally considered a good thing architecturally, with compilers left to inline those functions where appropriate.

Therefore what is needed is a way to make objects perform more like functions – to eliminate as much as possible the need to instantiate them, make copies of them, synchronise them and deallocate them. This need arises because objects have *state*, or more strictly mutable state. Two objects of the same class whose state is immutable are, to all extents and purposes, the same object (provided they were initialised with the same parameters). An application only ever needs a single instance of an immutable object: it only needs allocating once, it never needs copying, it never needs synchronizing (immutability implies thread-safety) and it never needs deallocating (until the end of the application). Of course, mandating immutability in anticipation of unacceptable performance could be considered premature optimisation – a topic whose pitfalls are well documented (Knuth 1974). But this is less about implementing any one optimisation early on, more about providing scope for multiple optimisations in the future.

Bloch (2001) stresses the advantages of immutable objects. He encourages us to "favour immutability… immutable objects are inherently thread-safe; they require no synchronisation. They cannot be corrupted by multiple threads accessing them concurrently… therefore immutable objects can be shared freely… you never have to make defensive copies. In fact, you never have to make any copies at all". Bloch points out the value of immutability in architectures composed of many smaller parts: "immutable objects make great building blocks for other objects, where mutable or immutable. It's much easier to maintain the invariants of a complex object if you know that its component objects will not change underneath it". He even goes so far as to turn the issue on its head, saying "classes should be immutable unless there's a very good reason to make them mutable" (Bloch 2001).

However immutable classes are not without their problems. First, they can be cumbersome to configure, because all configuration must be done at instantiation time. This can lead to their constructors requiring many parameters. For example:

```
new ImmutableRectangle(width, height, colour, edgeColour, shadowColour);
```

Such constructors are unwieldy and not particularly type-safe. For example both `width` and `height` are integers, so their values could be accidentally transposed. One alternative is to introduce separate configuration classes which are first configured and then passed to the immutable class' constructor:

```
ImmutableRectangleConf config = new ImmutableRectangleConf();
config.setWidth(10);
config.setHeight(10);
```

```
new ImmutableRectangle(config);
```

This is more type-safe, but no less cumbersome to work with. Fowler and Evans (2005) propose a compromise they term a 'fluent interface':

```
new ImmutableRectangle(new ImmutableRectangleConf().width(10).height(10));
```

The fluent interface approach has significant advantages: the configuration parameters are type-safe, they can be specified in any order, and it is not necessary to specify parameters that do not differ from the defaults (i.e. `edgeColour` and `shadowColour`).

Another problem with immutable classes is that much of their benefit is lost if clients neglect to reuse them. There is little to prevent a practitioner simply instantiating multiple, unnecessary copies of an immutable class:

```
new ImmutableRectangle(new ImmutableRectangleConf().width(10).height(10));
new ImmutableRectangle(new ImmutableRectangleConf().width(10).height(10));
```

This problem is particularly acute because mutable objects tend to be the norm. Instantiating new objects is the 'natural thing' for practitioners to do in languages such as C# (Hejlsberg 2006) and Java (Gosling 2005). It can also be cumbersome for practitioners to maintain a reference to a single, immutable object in their code such that the reference can be accessed from anywhere that needs it – sometimes it is easier just to call `new`.

Bloch's (2001) advice is simply to use encouragement: “immutable classes should take advantage of [the fact they can be shared] by encouraging clients to reuse existing instances wherever possible”. Metawidget attempts to do better, by 'rewarding' clients. Both interactive graphical specification tools (see 2.1.1.1) and model-based tools (see 2.1.1.2) often make it difficult for practitioners to access the Metawidget API programmatically. This is because interactive graphical specification tools employ a 'visual' metaphor, and model-based tools employ Domain Specific Languages (DSL) such as XML (XML 2008) or JSP (JSP 2006). Therefore Metawidget supports external configuration through a `ConfigReader` class and a `metawidget.xml` file. For example:

```
<metawidget>
   <htmlMetawidget xmlns=”org.metawidget.faces.component.html”>
      <inspector>
         <propertyTypeInspector />
```

```
            </inspector>
          </htmlMetawidget>
        </metawidget>
```

Primarily, this external configuration means the practitioner can instantiate and configure a Metawidget without needing to access the API programmatically. But because such instantiation is now out of the practitioner's hands we can, as a 'reward' for using `metawidget.xml`, intelligently instantiate, cache and reuse the immutable objects automatically.

This combination of immutable objects, a fluent API, and intelligent external configuration, forms an effective compromise between requiring a highly modular architecture composed of dozens of objects while at the same time requiring a simple, performant solution that does not impose undue overhead for its part in the wider application.

4.2.2.3. Promotion

A final important activity of the alpha 'act' phase was to spend time promoting the project to both industry and the research community. This was a critical step in order to generate sufficient feedback and observations for the next phase of the Action Research cycle. Central to this initiative was a Web site[1] to draw practitioners to the project. As Bacon (2009) stresses: “A Web site is *essential* to achieve [community interest]… the ubiquity of the Internet and the low cost of equipment have made an online presence the storefront for your community. If someone hears about your community, the first step he will take is to search for it [online]”.

The Web site was carefully constructed to have a variety of 'attractive paths' to suit different practitioner's experiences, tastes and time commitment. Web Analytics tools were used to evaluate the effectiveness of the paths. For example the opening page, shown in figure 14, was designed around a single large diagram giving an immediate overview of the project. I commissioned this artwork from a graphic design studio, and worked with them over a series of iterations to capture and convey a difficult technical message with clarity and in an attractive style. Bacon (2009) frames this as: “If I visit your Web site, I want to be able to get an overview of the community, its goals, and how to get involved, all within the space of a few minutes. This information should be up-front, easy to access, and easy to read, and should have a simple Web address that you can point people to”.

1 See http://metawidget.org

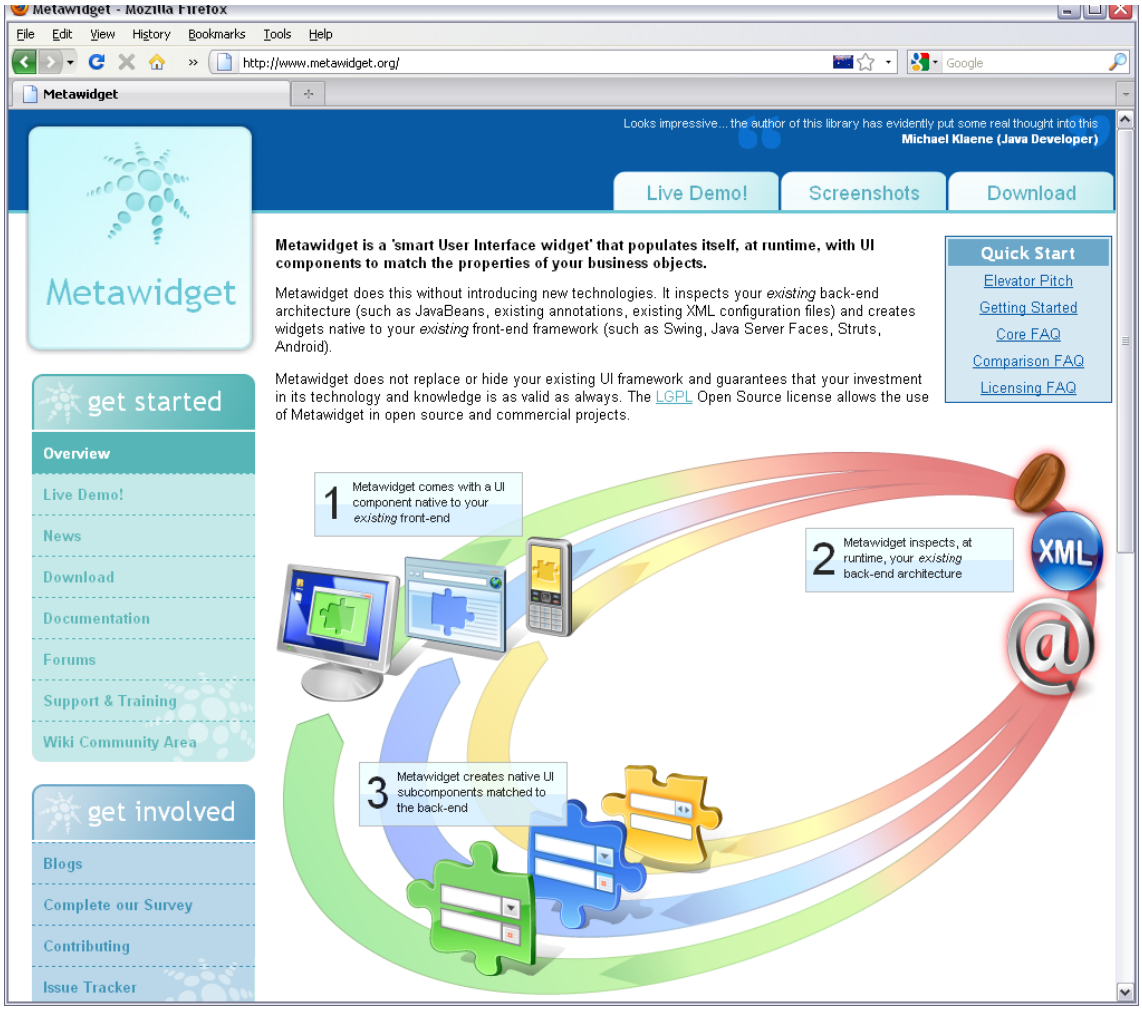


Figure 14: metawidget.org Web site

Following this opening page I split the site into a number of streams targeting different audiences: “Your Web site should provide documentation and guidance for all aspects of your community” (Bacon 2009). For project managers, or less technical practitioners, looking for a quick overview I storyboarded an 'elevator pitch' cartoon shown in figure 15. The cartoon format was chosen after experience taught people tended to skim-read traditional text. This is a problem when the text itself is already condensed into summary form, as key messages can be missed. For more technical practitioners I added a screenshot gallery showcasing Metawidget deployed in various production systems. For more technical practitioners still, I developed a 'live demo' which ran inside the Web browser and delivered a limited, pre-configured development environment containing pre-written example code as shown in figure 16. This allowed what Sahavechaphan & Claypool (2006) describe as “developing by example… a largely unwritten, yet standard, practice”: practitioners could immediately run and experiment with Metawidget without needing to download the project.

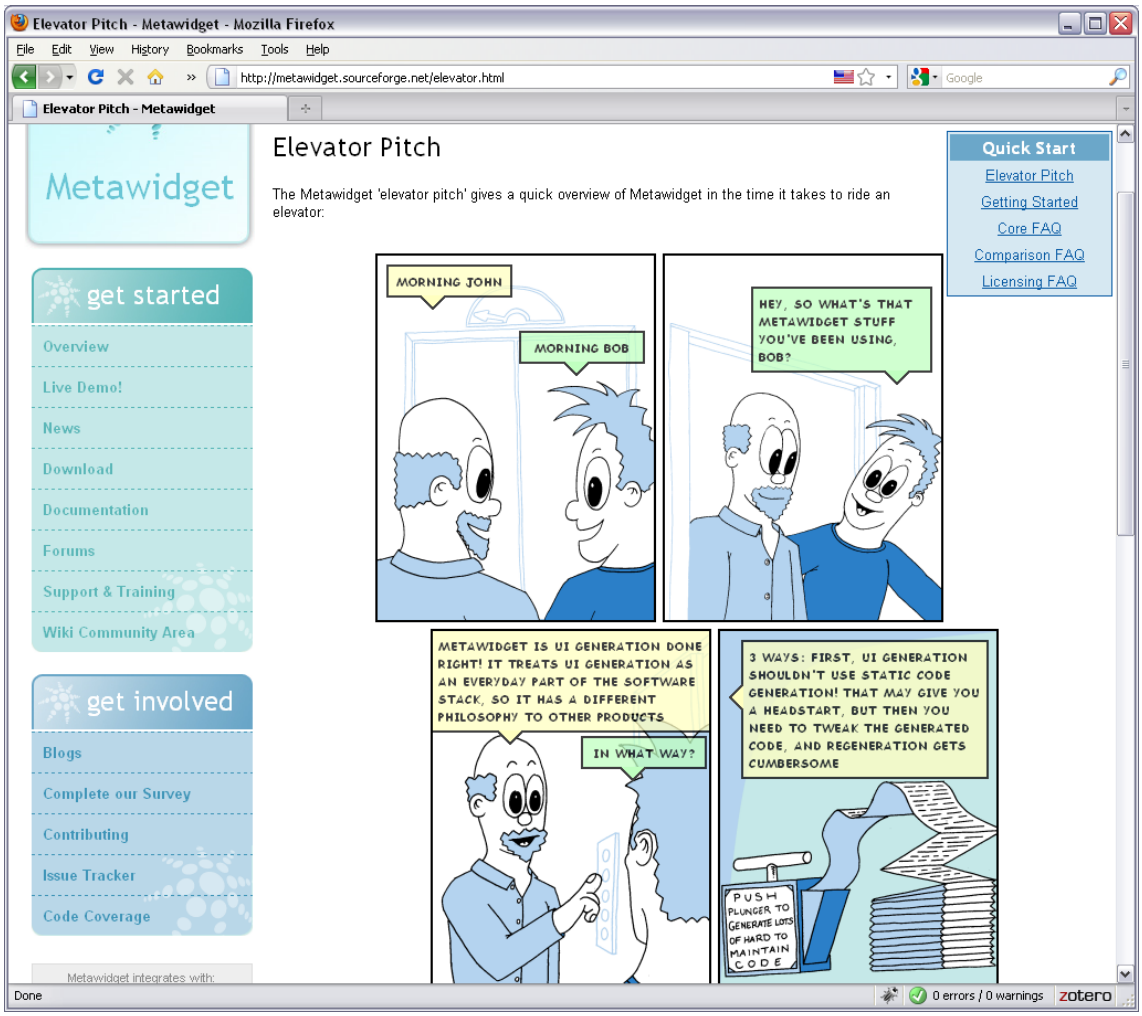


Figure 15: Elevator pitch cartoon

Paths such as overview diagrams, cartoons, screenshot galleries and live demos are designed to be effective under very short time frames: where the window of opportunity to attract practitioners is only a few seconds. But the paths must not stop there. Once a practitioner has 'bought in' and decided to download the project and dedicate say, 10 minutes to evaluating it, it is equally critical they have a positive experience. I developed an illustrated User Guide that ran to over 100 pages containing step-by-step tutorials and detailed reference information. I additionally implemented over a dozen example applications showcasing Metawidget running on various architectures. By now I hoped the practitioner would be well engaged, but it was still important not to lose them as they adopted the project. I set up message forums and replied to hundreds of queries and support requests. I wrote dozens of blog entries covering a variety of technical aspects of using the software.

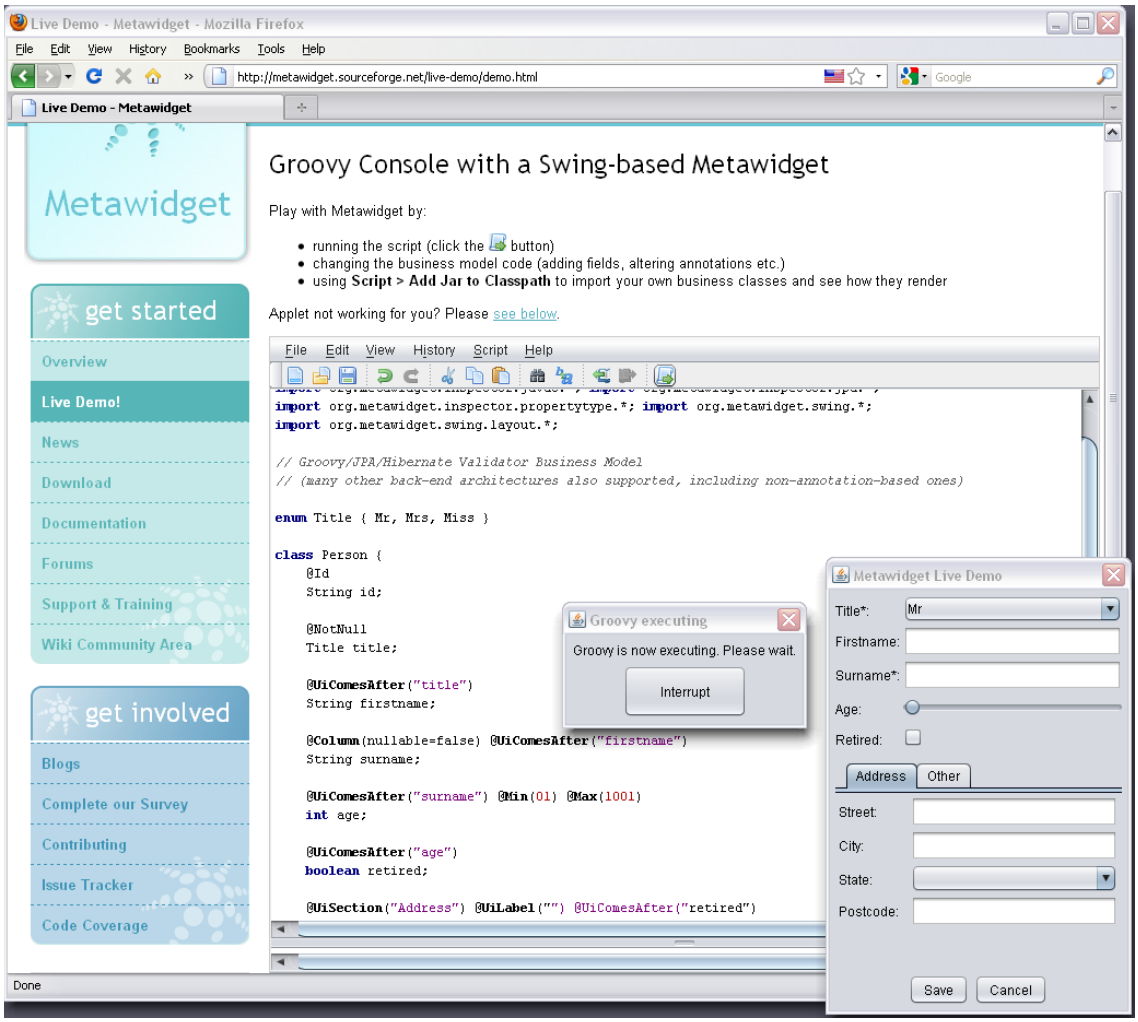


Figure 16: Live demo running inside the Web browser

Of course, the best promotion is no promotion: to attract by impartial recommendation or 'word of mouth'. To encourage this, I targeted key industry figures and technologies. For example I would develop a plug-in between Metawidget and a technology whose endorsement I was interested in, then send it to that project's team and work with them to promote it. Or I would research academic papers related to, say, loose coupling and code duplication, and write to the authors offering Metawidget as a solution to the issues they raised. As a final example, there are a number of industry news sites and software catalogues specialising in different market segments, such as Web application technologies. Because Metawidget cuts across multiple technologies, it was appropriate to promote it on several such sites. Such 'grass roots' efforts are unpredictable, and most specific instances were unfruitful (i.e. the industry team was too busy to look at my work, the research group not interested in my direction), but fundamentally they are the best way to attract practitioners.

Promotion often seems a fruitless task because there is an unpredictable latency between production of the promotional material and its impact. This is particularly true in online environments, where material is not so much distributed by the vendor as requested by the consumer: the Web Analytics tools showed people conducting Web searches, finding and commenting on blog entries weeks after they had been written; the adoption studies found teams who encountered Metawidget but were not able to utilise it until months later when they began work on their next project. This lag can be frustrating for the vendor, but is appropriate as far as the consumer is concerned: it takes time to establish a reputation. Also, it only requires a small number of successes in order to start generating the momentum necessary for useful observations and feedback. It is these observations we turn to in the next section.

4.3. Observing

This section records reflections from the 'observe' phase of my first Action Research cycle.

4.3.1. Reflections Following Observations

Once initial versions of the software have been released it becomes possible to gather third-party, and hence more impartial, observations on its effectiveness. These are a variety of ways to obtain such observations, and they are a rich source of reflections. This section details the methods used and the insights gained.

4.3.1.1. Interviews

Section 4.2.1.4 discussed reflections from the first conference paper. One reviewer had questioned whether restating of information between the UI and the business layer was really a problem. At the time I had to concede I lacked evidence other than my own experiences as an industry practitioner. Given this was a fundamental premise of my thesis, as discussed in section 2.1.3, it was important to validate it. And because of my constructivist epistemology (see 3.1) this meant canvassing the experiences of others. To do so, I conducted six[2] interviews with senior software developers chosen from different segments of industry – including finance, medical and middleware – across the UK, the US and Australia. Candidates were selected that had a broad understanding of the software development process, but were unaware of my work

2 Interviews continued until the convergence of themes and strong corroborative evidence made it clear further candidates were not necessary

on Metawidget.

I chose a standardised, open-ended format for the interview (Valenzuela & Shrivastava 2002). This approach involves asking the same standardised set of questions to each interviewee, but the set is necessarily short because each question is framed broadly so as to allow the candidates room to talk openly about their experiences. Standardised, open-ended interviews allow accurate comparison and analysis of results, whilst avoiding leading the interviewee and therefore minimising bias. To analyse the results, I employed a simplified version of Grounded Theory (Dick 2005). This theory involves coding, comparing and sorting categories that emerge from the interview sessions. Of principal interest to this interview was the category of duplication. Sub-categories included defects caused by duplication and the prevalence of duplication. The following sections discuss each of these categories in turn.

4.3.1.1.1. Duplication

I began each interview by informing the practitioner I wanted to talk about the mechanics, not the aesthetics, of developing a UI and its relation to the rest of an application. I asked each practitioner to describe the process they would go through to add a Date of Birth field to an existing `Person` object in their current software system, including both the back-end and front-end. This initial question was deliberately phrased to be as open-ended as possible. Specifically, it avoided the bias of mentioning duplication. However because I didn't explicitly prompt duplication, it was important to have each practitioner talk not just about the UI but all steps of the process, from back-end to front-end. In this way, the duplication would become apparent of its own accord. I phrased the question around updating an *existing* domain object, rather than a new one. I did this in order to expose the weaknesses of static code generation tools (see 4.1.1.3), but again because I didn't explicitly refer to static code generation I didn't feel this biased the responses. Finally, I chose a date field in order to expose issues around data conversion errors between layers, which is symptomatic of duplication.

All the practitioners gave similar answers for the process. One enumerated “first off we would add [the Date of Birth field] to the database, in the table. We'd then add it to the stored procedures going up. Add it into the Data Access Layer for the purposes of getting it out of the recordset. And then you'd add the property into the business level, the business layer. And then, on the UI, on the front-end, we'd have to add the field in the HTML”. Another practitioner said “I would go to the persistence level, I'd work out how that field should be modelled in the problem domain. For date of birth, you'd have a date column. I'd look at the `Person` class, work

out its relationship with the `Person` schema. Work out its name, what its type would be, `date` or `datetime` depending on the database. Then I'd work out how I should change the `Person` class – there'd probably just be a getter and setter – and then I'd tie it back to the persistence layer, map it back to the table. For validation constraints, yeah, this is always a problem, you need to validate it both in the UI and at the persistence layer if that's a business rule, so it's always a problem. In terms of the UI, I'd go and find the bit of UI code and work out the position where this field should be added".

It was noted those practitioners using newer technologies had considerably fewer steps. One said "we would obviously add that field to the actual business object that [JPA] maps to the database, that's already there. And then any validation constraints that are around that – we use Hibernate Validator (Hibernate Validator 2008) so we'd put the validation constraints on the entity, we don't have to do anything more for validation other than that, and all that's left now is dropping the field on to the UI, and that should be it really. Using the IDE we have we'd drag and drop UI components, then we'd have to apply some kind of formatting as well, some formatting to the underlying XHTML". However I observed this sub-category (Dick 2005) of fewer steps was generally from the domain objects 'down' the stack through to the persistence layer. It removed the manual coding of schemas, stored procedures and recordsets. But it did not reduce steps 'up' to the UI layer.

I then summarised the steps back to the practitioner and asked whether they thought any steps were deficient. Not all the practitioners were immediately aware of any problem. This is understandable for such an entrenched issue: some interviewees simply don't know any different. One said "what we have now is pretty good, certainly compared to Java Server Pages (JSP) or something like that. Two steps to add a field [one for back-end, one for front-end] is pretty good. The framework handles quite a lot and we can develop much faster than we normally do". For those practitioners I used a further probe question (Valenzuela & Shrivastava 2002), which specifically raised awareness of restating information: I asked whether any steps seemed redundant, or contained duplicated information from previous steps. Such a question has inherent bias, so it was not asked unless the practitioner failed to identify duplication naturally.

Following the probe question, all interviewees converged on recognising duplication amongst the steps. "The problem definitely exists. It's more from the business layer forward to the screen is the biggest problem because there are things out there like Hibernate (Hibernate 2009) which do from, sort of, business layer down". Another echoed this sentiment "the drudgery at the

moment is adding the UI code, and adding the validation and giving that feedback. That's really quite unpleasant. It's the most complex of all the steps, actually, depending on the magnitude of the change. Given a very simple change, just adding a single field, the bulk of the work, the bulk of the drudgery, in the coding is at the UI level. Being able to more concisely express the relationship between the UI and the model and the change I want to make in one place, or at most two places, in a very concise fashion would help". Another warned "it's a fairly established software engineering principle that the more you have to repeat something the higher the error is, the higher the chances there's going to be an error in the code".

4.3.1.1.2. Defects

Following on from this, I asked each practitioner whether they had ever encountered defects that were a result of this recognised deficiency in their process. All of the interviewees responded that such defects were common. "Definitely. There's always a chance that someone's going to get a bug somewhere along the line, especially with Date of Birth – as it goes down the date gets mixed up because someone's used the incorrect data type. With some of our junior developers we have here that's quite a common thing where they get a bit muddled up… it's definitely an issue that should be far simpler". Another agreed "All the time. That would be me overlooking various aspects of the user feedback loop, in the validation, me forgetting to persist various fields that I've added, so the validation happens but then it never gets persisted, so having to tie the new field to the model, with validation, in multiple places, gives a number of points where I could fail to do that". Another said, of reviewing other developer's code, "a large percentage of mistakes were always they'd copy and pasted [another field] and they'd changed that [declaration], and that one, and that one – but not that one. So it creates a higher chance of there being a minor error".

Several practitioners echoed this difficulty of identifying duplication related defects, because they generally evade static checking and projects must rely on runtime testing to detect them. One financial software practitioner explained "we've got a `BigDecimal` (Gosling 2005), and [the back-end has] set the scale to 8 but the UI puts through 10, it [gets silently rounded and] passes all the way through. That becomes a real issue because it's really hard to find. That's caused us huge problems before". Another agreed "it's the biggest problem I personally face. These sorts of errors. You're updating, say, you change the type of a field and you try updating it with, say, a `datetime` object but you've actually now changed it to an `integer` field, you don't realise until you actually start testing the application, or if you miss it in testing and send it out

to customers, you don't realise that there's a problem until you get the bug reports – not ideal".

One practitioner described how, because duplication is generally not understood by refactoring tools, it works against his preferred methodology of aggressive refactoring: "if you change a field name, and I do like to change field names – I don't know why – so I'll decide after a year of using the program 'what's that field name doing there?' I did it the other day: I've got a stock control module in the program and there's [a field] called `stock_reorder_level_reminder` and I thought 'what a stupid name for a field', so I just changed it to `reorder_level` because that's much easier. Now, generally changing that could have massive implications couldn't it? You could change that and it could break the application in several parts".

4.3.1.1.3. Prevalence

Finally, I asked each practitioner whether the themes explored in the interview were commonplace across all software systems they had developed. One said "I've built a number of UIs over the course of my career, some of them have been desktop applications, some of them have been Web applications, and I think this is a general problem. For desktop applications it's hard but it's relatively easy. For Web I think it becomes a lot more difficult because the technologies involved are a lot more fiddly, there are a lot more moving parts in Web application UIs. But yes I think it's a general problem.". Another said "quite honestly laying out UI forms is time consuming, it's fairly standard how a UI is – it shouldn't be a problem to say, okay, you have these things you probably want to interface in a particular way, here's what we suggest – we being the computer – you've got a `datetime` here, here's the calendar control we suggest. Oh you don't want a calendar, you want to use a text box, go for it. Something along those lines would definitely detract from the tedium of putting together the UI, which is an important step and everything but is a really repetitive process. If it's a `varchar` in the database, it's going to manifest as some form of a text box on the form. If I've got a foreign key in my database, it's going to manifest as some form of list box, dropdown, radio button, check box. It's not a huge leap". One practitioner summarised it as "every developer who writes anything more than a Hello World application will have this problem. Most developers who strive to make their work better, who aren't lazy, do sense this problem, do encounter this problem on a daily basis as a constant friction in their daily work".

I observed a sub-category (Dick 2005) that this friction had driven several practitioners to fashion their own ad hoc solutions by combining existing tools. "For a brand new screen we're currently using CodeSmith (CodeSmith 2009), so if you design the database table you can hit

generate and it'll go through and generate everything right up to the screen". However because of subsequent editing of the generated code, they found CodeSmith to be of limited use outside of new screens: "if you could do the same thing where you could add a new field to the database and it generated and added it into the [existing] code for you as it goes up that'd be excellent". Other solutions had similar shortcomings. Microsoft LinQ (LinQ 2009) helped with the persistence layer, but "if I go in and create a field, LinQ creates a nullable version of that field, where the [UI] control I'm binding it to is expecting a non-nullable version. That's caused a number of problems. That's come up a number of times and you've really got to kind of juggle to make it work right. Keep in mind when that could happen and keep track of the potential for it to happen". Asked why they had invested the considerable resources to fashion their own solutions: "I do genuinely believe that kind of thing makes the development cycle better in the long run. It makes things much cleaner, there's less coding to go on. If I were to have to write, well, in my application the basic objects I have, I have patients, contacts, appointments, items, invoice, payments, refunds, credits and then a load of secondary objects like appointment status', patient categories, all of these are objects. If I had to code a separate form for each one it's just tedious. Interface work is not that much fun. It's quite tedious, dropping controls on a form, lining them up with the other controls and fiddling around for ages". Another practitioner echoed this sentiment saying, if such tedium could be reduced, "you'd have more time for the actual problem solving: defining, clarifying, implementing the problem rather than the mechanics of the 'auto pilot' of gotta code up this method, gotta code this, gotta code that. Give you more time to concentrate on the more energy-requiring things rather than the monotonous reproducing of stuff. Because, I mean, despite the fact they tell everyone not to, normally you end up copying and pasting things".

4.3.1.1.4. Conclusion

The results of the interviews suggested UI duplication was indeed a prevalent and serious problem in software development. I observed practitioners across industry segments and across software platforms, and saw a common theme of duplication. I also observed common themes of defects caused by duplication, how newer technologies only addressed duplication 'down' the stack, a tendency of practitioners to fashion ad hoc solutions, and a common desire for duplication to be addressed. These results were compiled into a second research paper (see section 5.2.1.2) and this time its reviewers seemed more convinced. One wrote "In this [second] paper, the authors provide extensive evidence for the problem of manual code duplication among subsystems in the process of software… the work is exceptionally well motivated".

Another: “the work is important for the [research] community to hear… nobody in a senior position in a software company today is going to not be aware of this problem and the various hacks, workarounds, and band-aids used to cover up the pain that it causes (at least not in a successful software company). Again, academics may be surprised by some of the comments and the gravity of the concerns, but I wasn't; this is a serious problem and there are partial solutions out there that various people employ with varying degrees of success”.

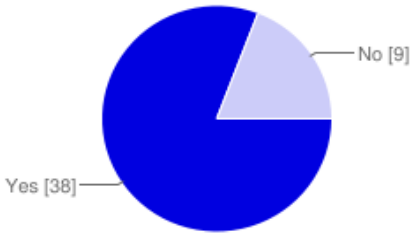
4.3.1.2. Self-Administered Survey

As further validation of my interview results, I conducted a self-administered survey. Self-administered surveys can quickly reach more candidates than interviews, but carry risks of ambiguity and ultimately invalidity because of their lack of an administrator. I used the constructs obtained from my interviews to operationalise a questionnaire. For example, whereas it was sufficient in my structured, open-ended interviews to ask “describe the process you would go through to add a Date of Birth field”, this question would be too ambiguous in a self-administered survey to return valid data. Instead, the principal category of 'duplication' needed to be decomposed into a number of unambiguous attributes, such as “when you add a new field to your back-end, do you also have to drag/drop a label in your UI builder?” and “do you also have to drag/drop a widget?”.

I conducted the self-administered survey through the project Web site (see 4.2.2.3) and collected 47 responses. The results are summarised on the following two pages. I should caveat that the respondents were inevitably 'pre-selected' by the fact they came looking for the project Web site in the first place.

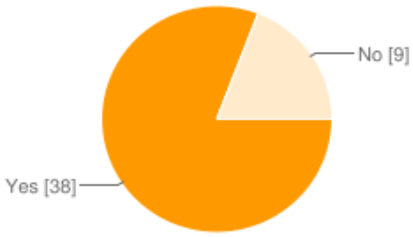
In your app, if you were to add a new field to the existing back-end (e.g. by specifying its name/type, maximum length, whether it is nullable etc.) you would also need to:

1. Manually add a label to your UI



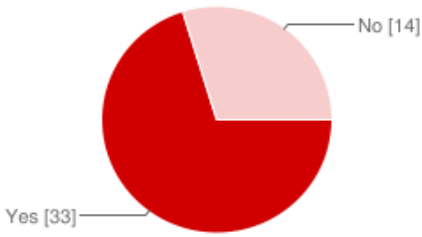
Yes	**38**	81%
No	**9**	19%

2. Manually add a widget to your UI



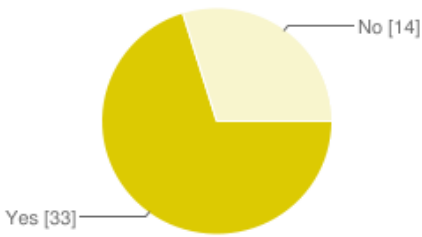
Yes	**38**	81%
No	**9**	19%

3. Manually apply constraints to the widget



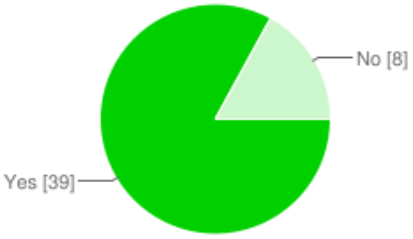
Yes	**33**	70%
No	**14**	30%

4. Manually indicate whether it's a required field



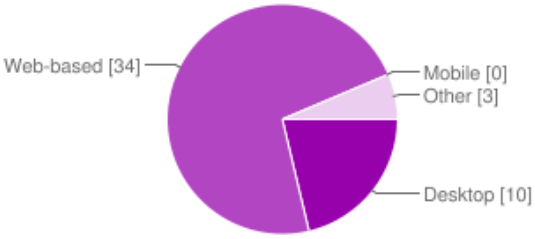
Yes	**33**	70%
No	**14**	30%

5. Manually repeat the above for every UI screen the field appears on



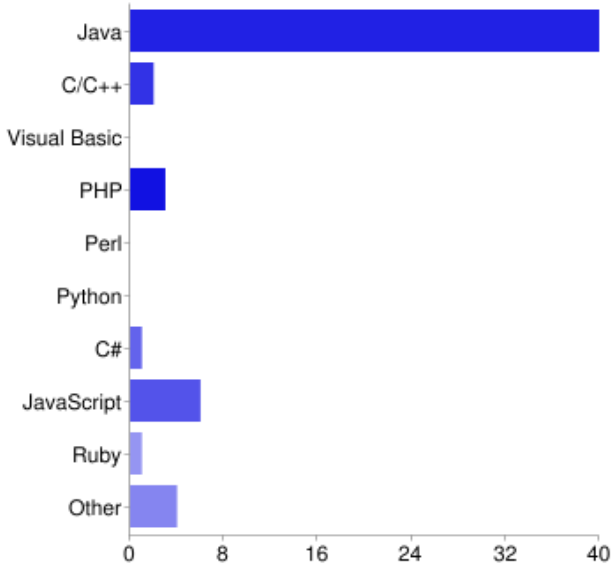
Yes	39	83%
No	8	17%

6. What type of app is it?



Desktop	10	21%
Web-based	34	72%
Mobile	0	0%
Other	3	6%

7. What platform is the app built on?



Java	40	85%
C/C++	2	4%
Visual Basic	0	0%
PHP	3	6%
Perl	0	0%
Python	0	0%
C#	1	2%
JavaScript	6	13%
Ruby	1	2%
Other	4	9%

People may select more than one checkbox, so percentages may add up to more than 100%.

The results from the self-administered survey correlated well with the results from my interviews. The majority of respondents were experiencing symptoms of UI duplication. This provided further evidence of the prevalence and severity of UI duplication in software development.

4.3.1.3. Forums

Given my research has such a strong industry-focus, message forums are a powerful tool because they are a familiar medium to industry. They offer a high level of immediacy and transparency, encouraging feedback and observations. They also carry a risk of bias and misinterpretation, unless the issues and ideas that come out of them are substantiated beyond a single point of view. When conducting the interviews (see 4.3.1.1) I posed open questions and received open answers, albeit citing specific incidents. But forum postings are often wholly focussed on specific incidents. These must be filtered and generalised in order to be valid. Simplified Grounded Theory (Dick 2005) is again a useful methodology here.

This section reviews the most notable forum postings.

4.3.1.3.1. Rebinding

Traditionally, UI frameworks tend to concern themselves purely with creating/arranging widgets, leaving it to the practitioner to pass values contained in those widgets to and from the domain model. For example practitioners have to manually write code to take the string `John` from a `Person` object and set it into a text box, then afterwards take it back from the text box and set it on the `Person` object. For practitioners using older UI frameworks such as JSP (2006) and Swing (1998), it would be enough for Metawidget to automate the creation/arrangement of widgets: the practitioners are used to being responsible for shuttling values back and forth.

Newer UI frameworks, and also third-party plug-ins for the older frameworks, alleviate this burden. They automate binding the UI to the domain objects, converting and synchronizing values between them (see section 2.1.2.3). In these cases, practitioners requested that Metawidget automate not just the creation/arrangement but also configure the binding.

This additional requirement introduced a performance problem: creation/arrangement of widgets is slow compared to updating their values. This is not a problem so long as the practitioner is manually updating the values, as he will generally reuse the created/arranged widgets after initial generation. But if widget creation/arrangement and binding are automated into a single step, it becomes necessary to re-create/re-arrange and re-bind every time the domain model changes. As one forum poster commented “I've hooked up the binding and when I start paging through my entities… I expected Metawidget to do a one-time scan of my entity and then simply swap the values [but] performance is poor to say the least… does Metawidget

do a complete reanalysis?". Yes, it did.

To resolve this, it was necessary to separate creation/arrangement from binding. But this seemed onerous, because some UI frameworks that *do* support binding do *not* separate creation/arrangement. For most Web frameworks, for example, creation/arrangement and binding are all performed during Web page rendering. Ignoring some advanced techniques (such as AJAX) it is not possible to re-bind Web components without also re-creating/re-arranging them (i.e. re-rendering the page). In order to support separated binding without burdening such use cases, a 'rebind' method was introduced: initial creation/arrangement/binding was still a single step, re-binding an additional step for those use cases and frameworks that supported it. This was well received. The forum poster commented "paging through the records now works! And at a normal speed, so things are starting to look good".

However rebinding did not fit well for those frameworks that did not support separate creation/arrangement, such as Web frameworks. Practitioners had to be instructed to use the original approach and to ignore the re-binding method. This was confusing, and instinctively 'felt' wrong (Schön 1983). It would be resolved later with the introduction of widget processors (see 6.1.1.1).

4.3.1.3.2. Conditional Expressions

Metawidget looks to integrate and automate existing UI frameworks. Many of these frameworks were designed for manual UI construction. So whilst they have low-level programmatic APIs they do not have any kind of high-level, declarative facility for coordinating interactions between widgets. For example if a practitioner wants a text box to be enabled/disabled depending on the state of a check box, they must programmatically attach event listeners to the check box which then programmatically enable the text box.

Interestingly, Metawidget's automation of the UI framework's low-level API made the lack of a declarative facility more apparent. One forum poster wrote "I have an object `Figure` with… `haveLegend` is a boolean. What I want if `haveLegend` is true: Metawidget show the legend part, and when it's false to disable it or it remove it". The user's chosen UI framework (Swing) had no facility to embed conditional expressions for showing or hiding widgets. But other UI frameworks do have such an expression language – typically those that are more declarative in nature, such as XML-based Web frameworks.

To satisfy this use case required Metawidget support both the framework's native expression language, where available, and also pluggable third-party expression languages for those frameworks that lack their own. This emergent problem could be summarised as: by automating imperative frameworks, Metawidget makes them declarative. But in making them declarative, it becomes Metawidget's responsibility to complete the missing pieces that declarative frameworks generally provide.

4.3.1.3.3. 1-to-M relationships

Metawidget is focused on generating UI widgets for one domain model entity at a time. If the entity has sub-entities with a 1-to-1 relationship, such as a `Person` entity with an `Address`, the software mining will 'drill down' to show fields within the sub-entity (as demonstrated in section 4.2.1.2). But if the relationship is 1-to-M it will not attempt to show a list of sub-entities. An emergent category (Dick 2005) from several forum posters was to ask why: "how does Metawidget work with 1-N, N-1 and N-M relationships", another "it seems not to be possible to generate a list with all properties of an entity… are any plans to do so?" and another "there is no list-widget, right? I mean a widget where certain properties are listed in columns (and one can click add/edit to invoke the other editor-widget)".

The reason for not supporting lists of entities is I considered it a slippery slope: first one has to render the list. Quickly afterwards come requirements for it to be sortable, to resize columns, scroll and paginate, highlight rows and so forth. It did not seem appropriate to the philosophy of integrating with existing UI frameworks for Metawidget to take ownership of such large amounts of functionality. Worse, such lists are generally not read-only: there is often a requirement to click on rows and navigate to a 'detail screen'. Again, this seems a slippery slope: should the detail screen be a dialog box? A separate page? Should the list instead support in-place editing? It did not seem appropriate for Metawidget to take ownership of screen navigation within an application, though this is the approach taken by others (Pawson 2004).

However 1-to-M support became such a common theme that, on reflection, I realised it should be supported as much as possible. We shall return to this in section 5.1.1.2.

4.3.1.3.4. SWT support

One of the earliest Metawidget adopters, after completing my Swing-based tutorial, commented: "The risk with a Swing example is two-fold – few people write Swing any more (SWT is more

popular for most real apps) and the resultant UIs are generally unimpressive… you need to add SWT support as I can well see SWT developers wanting to use your technology – and there are a good deal of them out there". Clearly the practitioner is expressing their opinion here, as it is difficult to gauge the true popularity of different UI frameworks such as Swing versus the Standard Widget Toolkit (SWT 2008). Nevertheless SWT is unquestionably a significant UI framework and it is important Metawidget support it.

SWT was developed by IBM as a successor to AWT (AWT 1998) and an alternative to Swing. Like AWT, it reuses the UI widgets of the native platform upon which it runs. For example an SWT application running on Windows will invoke Win32 APIs to instantiate actual Windows buttons and text boxes. This is in contrast to Swing, which will attempt to manually render, pixel-for-pixel, its own version of the buttons and text boxes. From a user's perspective the output of both frameworks should be visually identical, but the difference in their underlying philosophy has implications for both their performance metrics and their API design.

Most significant for my research, the design of the SWT API couples the instantiation of a widget with its addition to a parent container. It is impossible to instantiate a widget outside of a container, or to move a widget from one container to another once instantiated. Even removing a widget from a container is a relatively heavyweight operation compared to other frameworks. This had implications for Metawidget. In its current form Metawidget's API reused a single method, `buildWidget`, for both single-widget and compound-widget scenarios (see 4.2.2.2). This method was designed to return either null to indicate no widget was required; an appropriate widget for the given domain object property if available; or a nested Metawidget for a compound-widget scenario. In the latter case, the nested Metawidget wasn't actually added to the parent container, it just triggered `buildCompoundWidget`. This was problematic for SWT, because creating a widget as just a lightweight trigger to be subsequently thrown away did not suit SWT's heavyweight approach to widget handling.

The current design *did* work well for the seven other UI frameworks already implemented, including ones for desktop, Web and mobile (section 4.2.2.2). And there was a certain intuitiveness, from my practitioner's perspective, to returning null, a widget or a nested Metawidget. And finally it was possible to work around this limitation in SWT, albeit in a sub-optimal way (add the nested Metawidget to the container, then take it back out again). Nevertheless, it was clear the current API needed some reflection to fully accommodate the diverse needs of different UI frameworks with divergent philosophies. We shall return to this in

section 5.1.1.2.

4.3.1.4. Blogs

Like message forums, blogs are a familiar medium to industry – encouraging observations and feedback. This section explores the most significant blog exchange during the alpha Action Research cycle.

4.3.1.4.1. Explicit field ordering

There is an inconsistency amongst domain modelling technologies around whether to maintain the ordering of fields. For example elements in XML are explicitly ordered (XML 2008) whereas property definitions in CLR and JVM bytecode have no defined ordering (Miller & Ragsdale 2003; Gosling 2005). Metawidget's goal to integrate with existing back-end architectures meant it must accommodate both approaches. The latter, however, requires an additional mechanism to maintain field ordering. There are several possibilities. C# (Hejlsberg 2006) and Java (Gosling 2005) both support augmenting classes with metadata to describe the ordering. It is also possible to leverage debug information to reconstruct the ordering based on source code line number (Javassist 2008). By default, I had settled on requiring the practitioner to annotate each field with a `@UiComesAfter` annotation to indicate which field 'comes after' which other field.

One blogger suggested an alternative: have each Metawidget explicitly state the field ordering, rather than mine it from the back-end. I had explored this approach myself as part of Experiment 2 (see 4.2.1.5.2). As the blogger wrote, it has advantages both in clarity: “adding [field ordering information] to basically every field of your business model strongly reduces clarity… one of the key principles of Metawidget is the possibility to directly use your unchanged domain objects [but this] doesn't really follow that principle” and flexibility: “one wants to automatically create many different views based on a single business object with components of different sequence and visibility”. However it also introduces duplication (re-stating the fields used in a business object) and compromises polymorphism (Metawidget needs to statically know the fields of the business object). On reflection, I felt since the root of this problem was a shortcoming of particular back-end architectures (i.e. CLR and Java bytecode), it was inappropriate to introduce such an explicit mechanism. It would be of no use to those practitioners using XML back-ends, for example. It would also be of no use should the CLR or

JVM specifications ever be updated to maintain property ordering. The blogger agreed, but with suggestions for the next planning phase: “I agree, the default behaviour of Metawidget should be as it is now… still, I see [explicit field ordering] as an advantageous option for cases with specific concerns such as flexibility”. We shall return to this in section 6.1.1.2.

4.3.1.5. Adoption Studies

In addition to internal experiments on products for my own clients, having Metawidget released also made it possible to conduct adoption studies of third-parties. Adoption studies are interviews focused on adoption of a technology within an interviewee's organisation. The goal is to draw out all experiences and to understand usage and the environment in which the adoption occurred. While the experiments of section 4.2.1.5 were an important source of verification, adoption studies are a complementary form of validation – albeit the third-party products were necessarily of smaller scale, because Metawidget was still in an alpha state.

As with the interviews in section 4.3.1.1 the adoption studies used a list of standardised, open-ended questions (Valenzuela & Shrivastava 2002). Care was taken not to lead the subject. Candidates were gathered retrospectively from companies who had independently discovered and decided to apply Metawidget of their own volition, based on my promotion (see 4.2.2.3). These were discovered through message forums then contacted to solicit a study of their experiences. The interviewee was asked to discuss, and demonstrate, aspects of their adoption experience. Unlike section 4.3.1.1, however, the adoption studies were not looking to generalise, to codify, or to identify themes. Rather every comment, even in isolation, was valuable feedback to help improve the research.

I posed the following list of standard questions to each interviewee:

- What led you to find Metawidget? Did you perceive a need and then find it? Or find Metawidget and then understand its applicability? If the latter, were you aware of this need beforehand?

- Was the application already built and then you retrofitted Metawidget to it? Or did you start the application with Metawidget in mind?

- Have you used similar products to Metawidget in the past? If so, which ones and how do they compare? Have you ever built something similar to Metawidget, for your own

purposes?

- When working with Metawidget, what were some of the most important features to you? Where did you find Metawidget lacking?

- Did the Metawidget concept immediately make sense to you, or did it seem unnecessarily complicated (i.e. multiple inspectors, pluggable layouts, etc.)?

- Was being able to incorporate Metawidget within an existing UI important?

- Was supporting different back-end architectures, and being able to plug-in your own back-end inspectors, important to you?

- Did you have any feelings about putting UI-specific annotations on domain objects?

- Would you say that the problem Metawidget is trying to solve is a current and prevalent problem in software development? If so, why do you think it has not already been solved?

In practice, each adoption study diverged somewhat from this list. This was to be expected because I was, in effect, asking each practitioner to become a reflector of their own work. They had to be given time to explore their own reflections, to 'just let them talk', with no attempt on my part to immediately respond to their criticisms. Rather the goal was for them to surface as many positives and negatives as possible, which I could then review and reflect upon at a more considered pace.

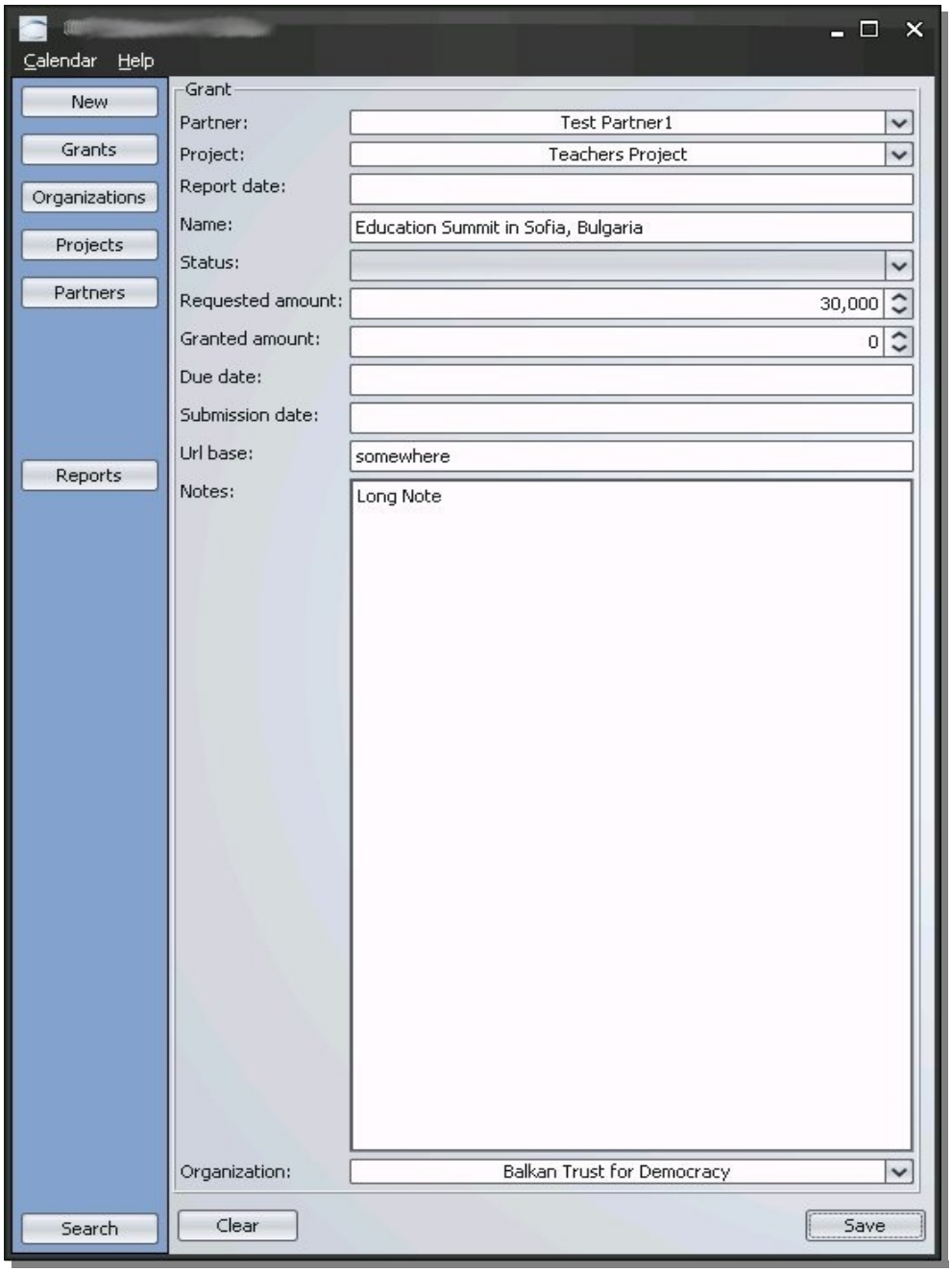
This first Action Research cycle attracted three adoption studies, which we will explore in the next three sections.

4.3.1.5.1. Adoption Study 1

Industry: Energy efficiency company based in Bulgaria, involved with receiving funds from the European Union Commission and assigning it to projects within their country.

Application: Internal, desktop application

Technologies: Java, Swing, JPA



4.3.1.5.1.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

The developer first described how, after being assigned a new project, he started thinking along the same lines as Metawidget. “I perceived the need, then a few days later as I was thinking of how to create a smaller scale version of what Metawidget is (due to client time constraints) I found Metawidget on [an industry Web site] and couldn't believe my luck – it fit the bill perfectly”. Metawidget is promoted on industry Web sites after each new release, as a means to gain exposure (see 4.2.2.3). A typical promotion would be a short write-up of the features included in the new version, with links for finding out more information, such as to user manuals or blog entries. Regarding the need the lead developer had perceived, had he used products that addressed this need before? “Nothing that has to do with the user end. But of course, [the ORM] Hibernate comes so close, and it goes the other way, toward the DB layer”.

The interview then turned to discussion of the objective of Metawidget. Was integration with existing UI frameworks significant? “Yes, very much so. I had already decided that I would go with Swing”. Did the UI generation seem limiting? “I didn't feel there were any special thing I could not include by hacking around in the `SwingMetawidget` code itself [because it's Open Source]”. What about integration with existing back-end architectures? “Many frameworks or tools enforce the designer's vision on how solutions should be architected. What I liked about Metawidget is that I could drop it in whatever architecture I was using”. For places where the back-end architecture had to be augmented with additional UI information (for example, the order of fields), did the developer prefer separate configuration files or augmenting the domain objects themselves? “While I appreciate the power within the XML inspectors, I used annotations to configure Metawidget”.

Finally, returning to the overall theme of the problem Metawidget is trying to solve. Is it a common problem? “I would say the problem is prevalent, yes”. Why has it not been addressed before? “I assume… because it is not a 'hot' topic in the Java community. Maybe people think it is too easy to solve. Maybe people want fine control over their UIs, and since they have not tried Metawidget they think it will invade their code (as is common in other frameworks)”.

4.3.1.5.1.2. Reflection

This adoption study considered a project that was being newly architected, but the developer

had already chosen his front-end (Swing) and back-end (Hibernate) before considering a UI generator. Therefore, it would not have suited him had the UI generator attempted to dictate the architecture (see 2.1.1.3).

During implementation, the developer preferred direct augmentation of the domain objects to separate configuration. This has the advantage of ease of development and ease of maintenance because of reduced duplication, but at the price of increased coupling between the UI and the domain objects. The weighing of such pros and cons is at the developer's discretion, so again it would not have suited him had the UI generator attempted to dictate one way or the other.

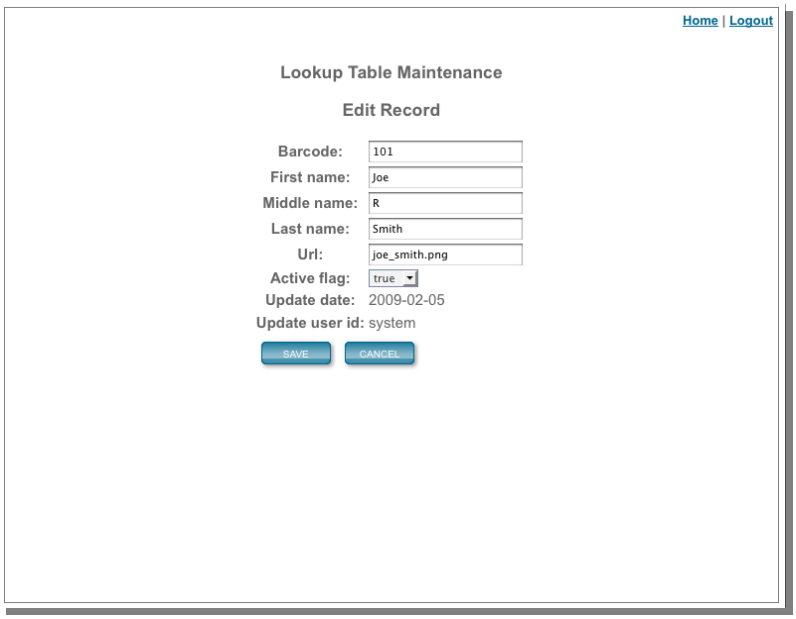
Finally, whilst the developer understood the problem and liked the solution, he felt that in order to address it in the wider community it needed to become a “hot topic” – it was important to get people to try Metawidget. This emphasises the necessity of well-designed, accessible resources: screenshots, demos, product comparisons, testimonials and a variety of other promotional materials (see 4.2.2.3). Directing energies into such marketing can seem tangential to my goal, but it is critical in order to attract users, observations and feedback. I have said Metawidget has a strong industry focus – it cannot succeed in a vacuum. But equally, I cannot expect that vacuum to be filled for me. I must actively seek my reviewers.

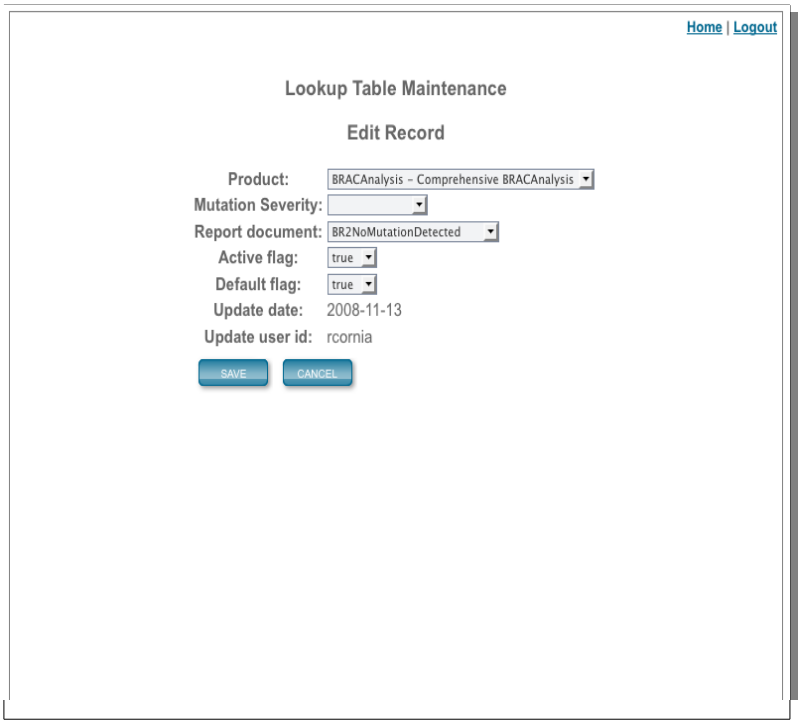
4.3.1.5.2. Adoption Study 2

Industry: U.S.-based biopharmaceutical company specialising in molecular diagnostics

Application: Web application

Technologies: Java, Spring, JPA





4.3.1.5.2.1. Synopsis

This adoption study was conducted by interviewing the team lead.

I began by discussing the team's need: “this was initially used for a new application. We wanted a way to add (easily) lookup table maintenance in our application tables so users could manage those changes themselves without having to enlist a developer or DBA to make the changes”. How did they attempt to address this? “Initially we wrote our own lookup table maintenance widget in Swing. This worked well, but could not be applied to Web applications as we began moving in that direction. The discussion revolved around how to reuse as much code as possible from our Swing implementation in a Web version. Being a common business problem, I searched for pre-existing tools and frameworks to fit this need, rather than write our own”. The team considered it a common business problem because “with previous companies, we wrote our own, simple frameworks for editing look up tables”. So the developer had repeatedly built such frameworks? “Yes, in several different positions. They all worked, but lacked the flexibility and applicability to a large range of problems. None were cross UI”.

This time the team decided on a different approach. Building their own framework “did not add any business specific value if we could find a third-party solution that solved the same problem”. What type of third-party solution were they looking for? “I would say we were looking for ease of use, yet flexible; something that required minimal code to get the job done; cross UI was important but ultimately would not have been the single driver”. Whilst being cross UI is an emergent property of Metawidget's objective, it is not a primary one. Rather, being 'UI agnostic' is. Was this important? “We needed to integrate it with a Spring MVC app, and in the future we may want to integrate with some existing Swing applications… also possibly Java Server Faces (JSF)”.

What impact did Metawidget have on development? “it made sense very quickly… setting up our initial prototype screen was very fast. I believe we had a working prototype in a few hours, certainly well under a day. That is a much smaller investment than had we written something from scratch”. What did the team decide regarding places where the back-end architecture had to be augmented with additional UI information? “There was some spirited debate, since [augmenting the business objects can] degrade gracefully if not in use. It still, to us, seemed cleaner to put UI-specific code outside of our business objects [in XML files]”.

Returning to the overall theme, is this a prevalent problem? “Absolutely. In 10 years of software

development, I can't count the number of times I've needed a simple form for users to enter or update data. I think it is a problem that has likely been 'solved' by many in their own specific companies, but no one has extended that in a general way to apply to a broad audience. Certainly there have been 'form code generators', but creating the form at runtime from metadata is a far more elegant approach in my opinion".

4.3.1.5.2.2. Reflection

This adoption study underscored that application architectures change from project to project. The team had moved from building Swing (1998) applications to Spring MVC (2011) ones, and were considering JSF (2011) in the future. Whilst being cross UI was not critical to them for any one project, being UI agnostic across multiple projects was. A UI generator that tied itself to any one architecture would have limited appeal over the long-term (see 2.1.1.3).

A key driver for the adoption was the recognition that UI duplication was a "common business problem". The team lead reiterated its prevalence (see 4.3.1.1.3). He expressed a desire for it to be solved by a third-party because solving it internally "did not add any business value". He recounted how he had built similar solutions to Metawidget for internal projects at a number of companies. He opined that other developers had probably done the same. This suggests there may be a latent body of knowledge around building UI generators that exists behind company walls. It may be a powerful approach to bring this knowledge to the fore, encouraging public debate and exposing lessons learned. This again suggests a need to promote and raise awareness of the issues surrounding UI generation (see 4.2.2.3).

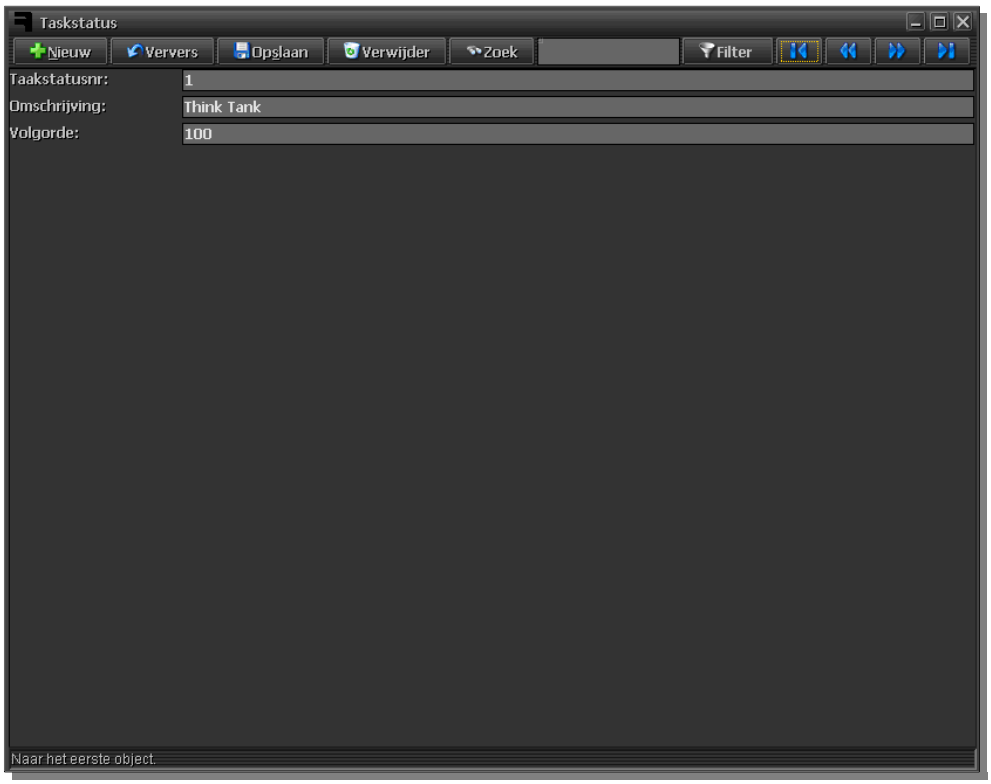
In general, however, this was a positive adoption study. Not all adoption studies would be so successful, as we shall see in the next section.

4.3.1.5.3. Adoption Study 3

Industry: Enterprise Resource Planning

Application: Desktop application

Technologies: Java, Swing



4.3.1.5.3.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

Was this a new project, or already built? “Already built. I was creating the '2.0' version, switching from a direct SQL-based to an object-persistence approach”. Why switch to Metawidget? “I already used dynamic screens… so I understood Metawidget's applicability immediately. I was aware of the need, since – because of the 2.0 version – I was rebuilding a lot of very simple screens”. Taking advantage of the opportunity afforded by the 2.0 rewrite, the developer was trying to keep everything as generic as possible: “classes handling [generic] instance navigation, classes to keep track of what instances were changed, classes for rendering 1-N relations, etc, etc. About when I had most generic code componentised, I noticed that I was copy-pasting the actual panels from which a screen is built; copy panel, replace entity, change labels and fields. I figured that should be componentised as well. This is when I introduced

Metawidget."

How did the introduction go? "It did seem complicated. I understand the need to support multiple platforms, but maybe it is wise to provide some sort of pre-set access points, if you understand what I mean".

Was the switch ultimately successful? "[No.] I found Metawidget to be somewhat too generic… very often I decided after a while to still build a specific panel instead. It is not that much work and the screen is just that little bit more tuned". Could this be improved by augmenting the business model with additional UI information? "Naturally I can add a lot of UI information to the business model, but I'm a strong believer in layers (for as long a possible) and UI information simply does not belong in a business model". As an alternative, Metawidget allows the developer to keep UI information in separate metadata files, but doing so does not solve the issue of duplication – fields must be declared and maintained in both the business model and the metadata files.

Perhaps Metawidget could at least validate the duplication, to ensure consistency? Would that be useful? "[Yes.] Validation is paramount if you have a layered architecture". Many object-persistence frameworks perform such validation between the business model and the persistence layer, does the developer use that? "[No.] For persistence I [augment the business model], this matches the view I have of persistence; it is an integral part of the business model". So persistence information belongs in the business model but UI information doesn't? Isn't that inconsistent? "I know this differs from other approaches, but every developer has his stubbornness (laughs)".

There were additional reasons Metawidget did not succeed: "if you have a more exotic UI component used for certain properties (JCalendar?), there is more work needed to get that to render, as opposed to simply creating the component in your UI layer". Also: "the additional information required to get the layout in Metawidget right, competes with the amount of code needed for a custom panel… I use JGoodies binding, MigLayout and some utility classes, so adding a field to a screen consists of 3 lines: create the component, bind it to the property, place it on the panel. Simple, minimal lines of code, understandable". Although simpler, is it less maintainable managing duplication between the domain objects and the UI layer? "That is initially the biggest advantage of Metawidget; it automatically updates the UI. But if you just let it do its thing, after adding the field, then it is dumped somewhere on the screen… [to reposition it] you need additional UI information… I may not even want [the field] to display. Changes to

the business model layer may automatically have unwanted consequences for the layers above". As a suggestion: "can't we have a `SortingInspector` which I provide with an array of property names, and those names come first in their array order, while the rest is appended alphabetically?". Overall: "I find the level of abstraction… not sufficiently rewarding above simply coding it out. For fat clients I believe the generic layout is not the quality of screens that people expect. The finer details get in the way. For [thin clients] this is less of an issue".

4.3.1.5.3.2. Reflection

This adoption study was revealing precisely because it was negative. The developer understood the problem, and tried Metawidget as a solution, yet concluded solving it automatically offered no compelling advantage over solving it manually. Six points stand out:

First, Metawidget's flexibility can make it seem complicated at first. A more considered approach with some sensible defaults may smooth the initial experience.

Second, there is a level of personal choice over the 'purity' of separation between layers. Experiments 1 and 2, and adoption study 1 were with developers happy to augment the business model with whatever metadata was required: UI, persistence, XML serialization and more. Adoption study 2 found developers who wanted to keep the business model free of anything unrelated. This adoption study found a developer who tolerated some metadata (persistence) but not others (UI). It is not clear which approach is better, if any. What *is* clear is the debate over purity of architecture is beyond the scope of a UI generator: any UI generator that attempts to dictate the approach alienates a segment of its audience.

Third, developers who choose to keep UI metadata in separate metadata files, as opposed to augmenting the business model, do not see as much benefit from Metawidget because they still have to maintain duplication between the metadata files and the business model. This situation could be improved. For example Metawidget could validate the metadata files in the same way many persistence solutions do.

Fourth, whilst Metawidget does support third-party UI components, currently this requires more work on the developer's part than necessary. Metawidget should improve its third-party component support – especially, on reflection, mixing third-party component libraries in the same project. If the developer wishes to use, say, the third-party JCalendar (as a date picker) that should not preclude using the third-party JFreeChart (for charting). This should also extend to any custom components the developer may have created.

Fifth, there was a desire for a different mechanism to sorting fields than the default annotation-based one (the developer annotates each field with a `@UiComesAfter` annotation, see 4.3.1.4.1). Specifically, a request to be able to "provide an array of property names" to an inspector. This requirement is similar to that discussed in section 4.3.1.4.1. It sits uncomfortably within Metawidget's architecture, because presumably the array of names must be specified per UI screen yet currently inspectors do not have any direct connection to the screen. This is an important design decision because some inspectors are designed to run remotely on different application tiers, where there is no screen available (see 4.2.1.3). Indeed some inspectors need to run on heterogeneous platforms to their UI widget. For example a Java-based back-end inspector may return information to an ECMAScript-based front-end widget.

Finally, it is apparent there is a tipping point to the usefulness of Metawidget, based on the initial overhead of introducing it into a project. For an application with a small number of unique-looking screens it is more cumbersome than working by hand. As the number and similarity of screens increase, Metawidget becomes more compelling. The challenge is to reduce the initial overhead so as to move the tipping point as close as possible to being useful for applications with small numbers of screens.

We will revisit each of these points (sensible defaults; not dictating the architecture; validating metadata; supporting third-party components; sorting fields; tipping point) in section 5.1.1.3, during our beta Action Research cycle.

5. Action Research: Beta Cycle

This chapter covers the second Action Research cycle, which ran from Q1-Q4 2009.

5.1. Planning

Unlike the first Action Research cycle, this time I had solid, third-party observations and reflections to draw upon to drive the cycle. My planning consisted of reviewing the observations and feedback from the alpha cycle and casting them in light of the year ahead.

5.1.1. Reflections During Planning

Planning can be a chaotic, unpredictable affair: periods of slow consideration followed by flashes of insight, sudden ideas and unfurling time lines. This section summarises the significant outcomes of the beta planning phase.

5.1.1.1. Reflection on Reflection

The methodology of Action Research instructs that the reflections from the previous phase should drive the planning for the next phase. Before detailing the planning for the beta phase, though, it is perhaps worthwhile to reinforce the value of this reflection. After all, reflection is expensive. Conducting experiments, interviews and adoption studies consumes valuable time and resources. It is legitimate to question whether its benefits outweigh its cost. To that end, let us pause to explicitly demonstrate the value of reflection with a practical example.

One of the most important factors in software development is scope: deciding what to include and what to leave out. 'Scope creep' and 'feature bloat' are recognised risks, impacting development costs and release schedules. Good practitioners carefully apply rules of thumb: every design decision should “pull its weight” (Bloch 2001), and strive to 'kill several birds with one stone'. But an implicit difficulty in evaluating this is knowing what the birds are. Once out of its initial planning phases, software development has a tendency to lurch from immediate issue to immediate issue, dealing with each new requirement as it arises. Considering requirements in isolation invariably means the burden of large-scale redesign to satisfy any one requirement will seem onerous: a smaller-scale, more imperfect but less impactful alternative

will always seem the better option. Over time, many such small, imperfect design decisions inevitably degrade the quality of the software. Reflection, on the other hand, allows the practitioner to consider many weeks worth of problems in a holistic light: they can see all the birds at once, and an approach that once seemed over-engineered now appears justified. Surfacing all the issues at the same time clears a path forward that otherwise would have seemed prohibitive.

This phenomena is analogous to neural networks. While progressing to solve a given problem, a neural network may get trapped, still short of the best solution, in a local minima (Figure 17). The local minima itself does not represent the best answer, but none of the immediate ways out of the minima are enough of an improvement to overcome the walls of the valley. It takes a combined push, a sort of disruptive excitation, to escape the trough so that a better solution can be found. In Action Research, reflection provides this combined push.

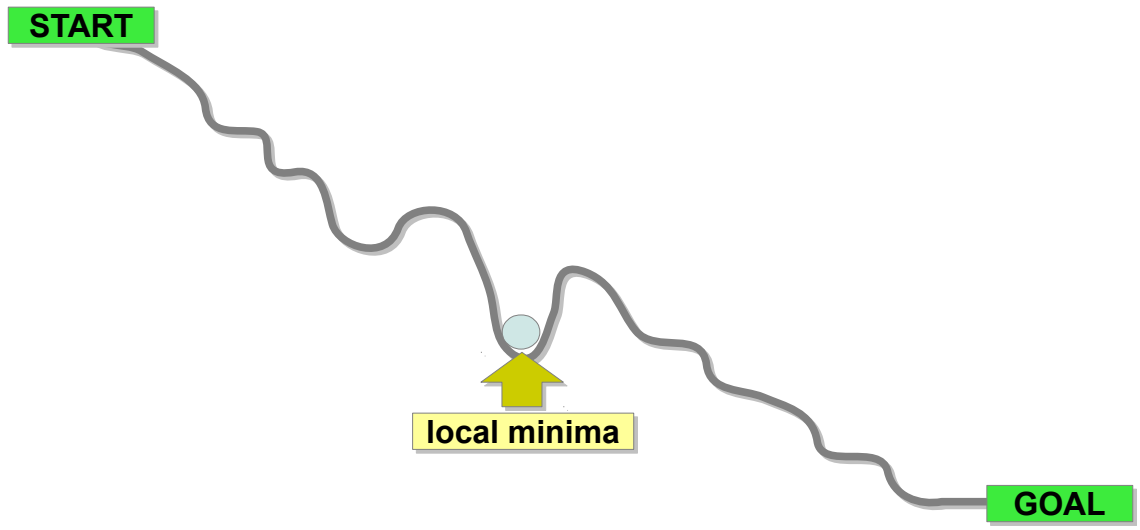


Figure 17: Neural network trapped in a local minima

The next section uses a concrete example to demonstrate this theory in practice. Other key examples include widget processors (see 6.1.1.1), inspection result processors (see 6.1.1.2) and ultimately a complete generation pipeline (see 6.1.1.4).

5.1.1.2. Widget Builders Revisited

As discussed in section 5.1.1.1, one of the primary values of explicit reflection is it enables surfacing multiple issues at the same time, whereby a holistic solution may become apparent. A

new approach to widget builders (see 4.2.1.1) was envisioned through precisely this process. A number of themes had emerged from the alpha cycle Action Research reflections.

First, support for 1-to-M relations (see 4.3.1.3.3). Until now, Metawidget had primarily concerned itself with rendering single entities (albeit including single entities with 1-to-1 nested entities) such as would typically be required to render a UI form. Compared to this, rendering 1-to-M relations is far less rigorously defined. For example, say we decide to render the 1-to-M relation in tabular form (by no means the only way a UI designer may choose to represent such a relation, but a reasonable one). Immediately there are concerns about whether the table should be sortable (if so, which columns); should it be paginated (if so, how many rows per page); should it be editable in-place, or should there be a separate click-through to view detail (if so, should this be a pop-up dialog or a separate page)? To complicate the problem, many of these concerns are not purely aesthetic but affect the back-end mechanics of the system: if we decide to enable pagination, we would need to optimise the amount of data retrieved from the database; if we decide to click-through, we must engage the page navigation subsystem. As discussed in section 4.1.1.4, one of Metawidget's goals is not to try and generate too much, becoming less useful in the process. Considered in isolation 1-to-M relations seemed better left to traditional UI mechanisms.

The second theme was supporting third-party, and also custom, UI components. The most challenging adoption study (see 4.3.1.5.3) indicated this would have improved adoption. At the time, supporting third-party widget libraries required the practitioner to inherit and override the Metawidget class itself. This was not only cumbersome, but because inheritance in Java is single it precluded the ability to mix multiple third-party libraries within the same UI.

The final theme was supporting the SWT library. The current design of “return either null to indicate no widget was required... or a nested Metawidget for a compound-widget scenario” (see 4.3.1.3.4) was backwards for SWT's purposes. The SWT API would be better suited to 'return a stub widget to indicate no widget was required, return null for a compound-widget scenario'. For now, this was being worked around in a sub-optimal way (add the nested Metawidget to the form, then remove it again), because considered in isolation it seemed like a corner case.

Individually, none of these requirements seemed enough to justify a significant reworking of the widget creation subsystem. Indeed the theme of 1-to-M relations gnawed at me for months with no obvious solution within the existing architecture. It was only reframing it within the context

of the additional requirements of 'supporting third-party components' and 'reversing widget creation' that a new path presented itself. Looking back, I realise I was probably *especially* resistant to seeing this path because it was in an area I had already considered, albeit in slightly different form, and decided against (see 4.2.1.1).

Widget builders abstract the `Metawidget.buildWidgets` method into a pluggable end-point. This makes it straightforward to plug-in support for third-party widget builders. But because the end-point is orthogonal to the Metawidget, it also makes it straightforward to plug-in a `CompositeWidgetBuilder`. By favouring composition over inheritance, this can mix together an arbitrary number of third-party widget libraries and decide which ones get precedence. This is shown in figure 18.

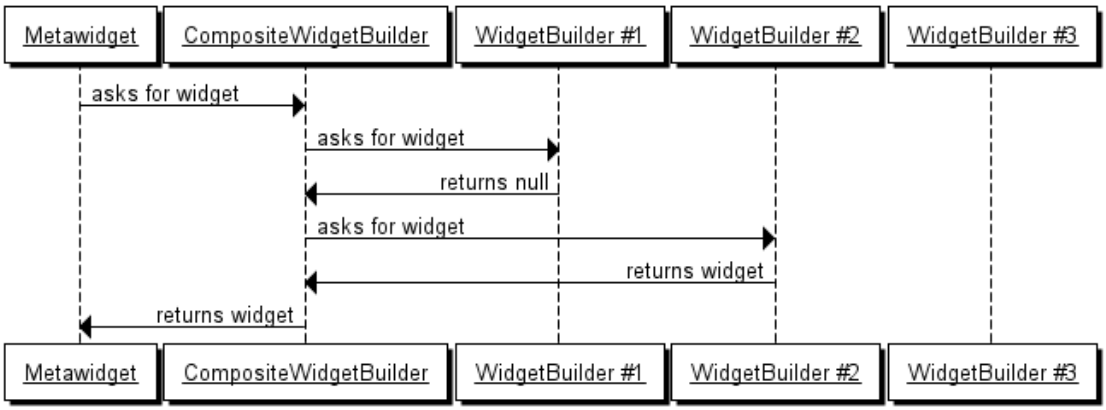


Figure 18: CompositeWidgetBuilder can compose multiple Widget Builders together

Secondly, because widget building is now a pluggable concern, we can support arbitrarily complex renderings of 1-to-M relations. Metawidget can supply a simple default (say, a tabular 1-to-M relation with no sorting and no pagination), but practitioners can plug in their own. Realistically, there is no 'one size fits all' solution to 1-to-M relations: every project will want to handle them differently. But *within* a given project, most 1-to-M relations will likely be handled the same because it is desirable a UI be consistent throughout an application (Myers, Hudson & Pausch 2000). By providing a path for practitioners to automate their own 1-to-M relations to suit their needs, they can gain the advantages of automation without the disadvantages of generalisation. Finally, the refactoring gave me the opportunity to invert the existing design of 'return null to render nothing' into 'return null to trigger nesting', which provided for cleaner SWT support.

This example demonstrates how holistic reflection over a group of problems can lead to deep insights. It can present opportunities for design which would otherwise go unnoticed or be considered over-engineering. Ultimately it can be used to address all of the, seemingly unrelated, problems with the one solution.

5.1.1.3. Effectiveness

To maximise the potential of my Action Research cycles, it was important to re-engage with individual practitioners following their observations from the previous cycle. I could then work with them as I began planning for the next cycle. Adoption study 3 (see 4.3.1.5.3) had been particularly challenging, documenting an adoption experience where Metawidget had proven ineffective versus manual coding. The practitioner had summarised: “I find the level of abstraction… not sufficiently rewarding above simply coding it out”. The bulk of the practitioner's feedback was negative, but also specific, making it a valuable starting point for improvement. Six points had stood out.

The first point concerned the learning curve of Metawidget. The practitioner had said: "It did seem complicated. I understand the need to support multiple platforms, but maybe it is wise to provide some sort of preset access points”. Here, the practitioner was making an appeal for sensible defaults: for an API that worked with minimal initial configuration, albeit without compromising configurability for more advanced use cases. I revisited my existing initial configuration:

```
SwingMetawidget metawidget = new SwingMetawidget();
metawidget.setInspector( new PropertyTypeInspector() );
metawidget.setWidgetBuilder( new SwingWidgetBuilder() );
metawidget.setToInspect( person );
```

This was only four lines of code, but as Bloch (2006) stresses: “minimizing conceptual weight is more important than class- or method-count”. I realised there was quite a significant conceptual weight in the four lines. In particular, I was requiring the practitioner be introduced to both inspectors and widget builders before running their first example. To remedy this, I looked into preconfiguring each Metawidget. I had already implemented an *external* configuration mechanism in `ConfigReader` (see 4.2.2.2.1) and I now looked to reuse this internally: supplying each Metawidget with a default XML configuration file that it read on instantiation. This succeeded in reducing the code to two lines, but more importantly in reducing

the conceptual weight:

```
SwingMetawidget metawidget = new SwingMetawidget();
metawidget.setToInspect( person );
```

The second point of feedback was a validation of my approach. The third point was an enhancement we shall return to in section 8.3.3. The fourth was addressed by the introduction of widget builders (see 5.1.1.2). The fifth concerned a different mechanism for sorting fields: “can't we have a `SortingInspector` which I provide with an array of property names, and those names come first in their array order, while the rest is appended alphabetically?”. This would require further reflection before a solution could be found (see 6.1.1.2).

The final point of feedback concerned the amount of code required to use Metawidget compared to manual coding. The practitioner had said: “I use JGoodies binding, MigLayout and some utility classes, so adding a field to a screen consists of 3 lines: create the component, bind it to the property, place it on the panel. Simple, minimal lines of code, understandable”. I went back to the practitioner and together we built a small project to demonstrate his technique. Upon closer inspection, he conceded he had underestimated the lines of code: “I would count 2 lines per form, and 4 per field: create component; create binding; create and layout label; layout component”. With these new metrics, and in light of the API refinements, widget builders and some other improvements, the practitioner agreed Metawidget was worth another try: “it should not be that hard to include Metawidget again and I still have a number of maintenance screens I'd love to improve. Let me see what Metawidget has evolved to”.

5.2. Acting

This section records reflections from the 'act' phase of the second Action Research cycle.

5.2.1. Reflections In Action

5.2.1.1. Exposure

As with promotion during the alpha cycle (see 4.2.2.3) an important part of the 'act' phase of the beta cycle was to stimulate feedback by increasing my exposure to, and adoption among, industry practitioners. As discussed in 4.2.2.3, there are a variety of ways to increase such exposure. Some of these are only available after the alpha cycle, once a version of the software

has been released. One such approach is to search online discussion forums and bug databases, looking for practitioners and project teams that have encountered similar problems to the ones Metawidget addresses, then suggest Metawidget to them. Care must be taken to target practitioners accurately, not to appear conceited or engaged in advertising, and often there will be little response. But in this section I record one of my successes.

I encountered a feature request for seam-gen, a static UI generation tool for the JBoss Seam Framework (Seam 2009). The feature request suggested it would "be convenient to be able to specify model or UI [re]generation for one or more specified entities after the initial generation". I added a comment to the feature request: "Would you guys find Metawidget useful for this purpose? Instead of statically generating the JSF components upon initial generation, you would put a Metawidget component on the page. Then as the model changes in the future the Metawidget will update automatically, because Metawidget uses runtime UI generation". The Seam developers replied: "I can tell you that we are not going to put Metawidget into seam-gen, at least not until we have some sort of plug-in system for it (no fixed plans yet). We are, however, completely open to hosting a Metawidget example under the examples directory, if you wish to demonstrate your tool for use with Seam to those who have merely downloaded the Seam distribution. Please take discussion of that topic offline from this issue, however".

I followed up this invitation and the Seam development team established the requirements for hosting a Metawidget example: "[the team leader] asked that we use the Hotel Booking [a sample already included in the Seam distribution] domain model… Feel free to enhance it if you need to demonstrate some specific feature. It doesn't have to be a blind copy. However try to adhere to the look and feel. The idea is to give the developer a basis for comparison". Here, the team leader was expressing the importance of having two versions of the same application, with Metawidget being the only variable. This was an excellent chance for Metawidget to demonstrate its ability to adapt to existing architectures – to *retrofit* an application.

The work required building Metawidget plug-ins for a number of JBoss' other technologies, including Bean Validation, Hibernate and JBoss jBPM. I completed the Hotel Booking and two other 'before and after' applications using three of the existing Seam samples. All were hosted as part of the Seam 2.1.2.GA release. This was an important milestone for Metawidget, as at that time Metawidget itself was averaging approximately 400 downloads a month whereas Seam, with which it was now included, was averaging some 10,000. As an added bonus, the work inspired the Seam team to recommended future directions: "I'd like to introduce you to the ICEfaces team to consider a possible integration with the ICEfaces component set… ICEfaces

would expose Metawidget to an even wider audience". The Seam team also tried Metawidget themselves: "BTW, I finally had a chance to try out Metawidget on an example of my own and it is quite amazing". These conversations, in turn, opened a dialogue with JBoss' parent company Red Hat, which would prove serendipitous (see 8.1.2).

5.2.1.2. Papers

Like the alpha cycle, it was important to generate feedback by writing and publishing papers for researchers to review. For the beta cycle I produced an in-depth paper that demonstrated and examined the prevalence of duplication in UI development (echoing section 4.3.1.1 of this thesis). This paper was accepted and published at the 2nd International Conference on Human System Interaction (Kennard, Edwards & Leaney 2009).

As with the first paper, the negative feedback brought the most insights. One reviewer wrote "This problem, in general, is both widely known and driving a great deal of activity… I am afraid with[out] concrete evidence… it's hard to see how this work is significant enough. The author's failed to convince me that some 19 yr old Finn isn't diligently working on the solution right now by combining open-source tool A and tool B and it's just a matter of waiting a few months".

The reviewer is concerned the work is not sufficiently difficult from a theoretical level: that the problem of UI generation is already widely recognised and numerous solutions have already been proposed. But the difficulty of UI generation is in the execution. For example Naked Objects (Pawson 2004) makes a significant attempt, but its decision to enforce a stylised behaviourally complete domain model and OOUI limits its effectiveness. It is critical to understand the nature of the problem in industry if we are to derive a practical result. This reinforces the importance of Action Research to my thesis: by completing successive cycles in close collaboration with both industry and the research community, I hope to produce a solution that is highly applicable to both.

We shall return to this reviewer's comments in section 6.2.1.2.

5.2.1.3. Experiments

5.2.1.3.1. Experiment 3

As part of the work to have Metawidget hosted with Seam (see 5.2.1.1), I developed a number of sample applications. These provided interesting 'before and after' experiments in retrofitting Metawidget to an existing architecture.

Retrofitting is a laudable but largely unpursued goal. It is laudable because the number of existing applications, both mature and currently under development, far outweighs the number of 'green field' applications that could reasonably be expected to adopt a new UI generator. The ability to remove duplication from *existing* applications, in addition to preventing duplication in new ones, has potential savings of many orders of magnitude. Retrofitting is largely unpursued presumably because it requires a level of UI generator flexibility, particularly flexibility towards back-end architectures, that is difficult to target. I discovered the retrofitting activity encompassed three main areas.

First was to explore what existing metadata could be leveraged from the application's back-end architecture. For example the Seam Hotel Booking sample contained some UI metadata embedded within its persistence subsystem, some within its validation subsystem, and some within a scripting language. Conversely, the Seam DVD Store sample contained UI metadata embedded within its BPM subsystem. I was able to plug in inspectors for each of these. The second activity was to introduce UI metadata that did not exist in the application but was required for generation. For example business field ordering information had to be incorporated. The final activity was to replicate the application's original UI appearance. I was able to plug in widget builders for this. In particular, I was able to plug in a mixture of widget builders to replicate the application's original choice of two widget libraries.

Overall there was a significant amount of existing code that could be removed, though notably some new code also had to be introduced – such as field ordering information and configuration files. Nevertheless, when comparing aggregate sizes of the files in the sample projects before and after retrofitting, I realised between a 30% to 40% reduction in UI code through the introduction of my implementation. For some individual UI screens this metric was as high as 70%, as shown in figure 19. On the left is the original XHTML source code for a single UI screen, on the right the retrofitted version (the source code is not meant to be legible in the

figure, it suffices just to be able to discern its structure[3]). The red boxes and lines convey which portions of the original source were able to be replaced, and their equivalent size in the retrofitted version.

Figure 19: Portions of code saved by retrofitting

3 Larger version available at http://blog.kennardconsulting.com/2009/05/metawidget-and-seam-saying-goodbye-to.html

5.2.2. Action Outcomes

In order to provide context for the observations and reflections that are to follow, this section briefly outlines the architecture of the solution as it stands at the end of the beta 'act' phase.

5.2.2.1. UML

Figure 20 shows the updated Metawidget architecture in UML form. As with Figure 13, it is much simplified from the actual architecture of the beta release, showing only relevant areas.

Specifically, Figure 20 illustrates the impact that the introduction of widget builders (see 5.1.1.2) had on the architecture. It can be seen that methods that were formerly part of `BaseMetawidgetMixin` are now split out into a separate `WidgetBuilder` interface with multiple, pluggable implementations. A `CompositeWidgetBuilder` implementation allows combining arbitrary numbers of widget builders in a configurable order. This provides simplified support for third-party and custom widget libraries, as raised in 4.3.1.5.3. It also shares a symmetry with `CompositeInspector` (see 4.2.1.3).

5.3. Observing

This section records reflections from the 'observe' phase of my second Action Research cycle.

5.3.1. Reflections Following Observations

5.3.1.1. Adoption Studies

As with the alpha cycle, adoption studies provided important validation from neutral third parties about their experiences with Metawidget. With the project now being in a beta cycle, the adoption studies were expected to be larger-scale. However the list of standard questions (see 4.3.1.5) remained the same.

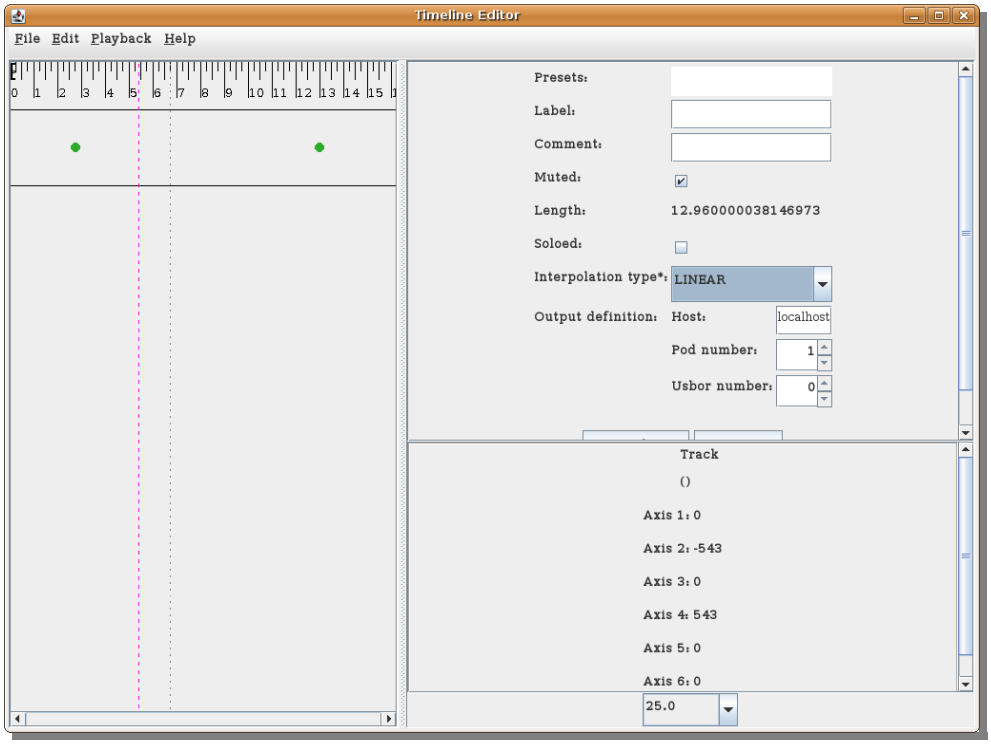
Figure 20: UML class diagram of Beta Action Research Cycle

5.3.1.1.1. Adoption Study 4

Industry: Light and sound engineering at the New York City College of Technology (part of CUNY)

Application: Desktop application

Technologies: Java, Swing



5.3.1.1.1.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

I began by discussing the project and Metawidget's role in it. “The project I am working on now is a time-line editor… [for] controlling robots, theatre lighting and sound playback. I tend to design UIs that correspond very closely to the back-end data model. Each object on the screen can usually be selected and edited in a side pane. That side pane is an obvious application of Metawidget”. With reference to the screenshot: “The project uses Metawidget in two places: one, as an object editor (top right pane) and two, as a read-only output status display (bottom right pane). One of the really important reasons I used Metawidget for this was that I have plug-ins (three of them as of now) that provide data objects for, and I/O support for, different types of [theatre] systems. I wanted to be able to just add data objects and have them be immediately editable without needing to implement any UI code at all.”

What led the developer to Metawidget? “I was looking for a tool to auto-generate UIs”. Had the developer encountered similar tools in the past? “Not [as] products per se, just homebrew… I've implemented my own tools that do the same thing in both Java and Python… so I knew it was possible and useful”. Clearly the homebrew Python tool would not be suitable this time because the new project needed to be Java-based. But the homebrew Java tool was not suitable either? “[No, it] extracts information from an OWL ontology instead of [Java]beans”. So being able to integrate with existing front-end and back-end architectures was important? “Sort of. [This was a new project so] the API was written with Metawidget in mind, but if I'd had to do it differently because of Metawidget I would have been unhappy. As things are I was just able to treat it as a normal Swing widget with was nice”.

There were some shortcomings, however. The developer found Metawidget lacked “a way to attach event handlers to widget value changes. This would allow you to respond to change… not just do a bi-directional [data] binding (for example you could enable a save button that starts disabled)”. Also “I found that there is no good way to define the order of the widgets on the UI across different levels of the class hierarchy”. The practitioner explained he had many derived classes, some of which were externally defined by his plug-ins, and therefore needed fine-grained control over widget ordering. “I want, say, `name` to always appear at the top of the UI. I also want all properties in one class [to] appear in a specific order without other properties between”. Metawidget does have such facilities – were they insufficient? The developer had “tried both the `@UiComesAfter` annotation and XML files. `@UiComesAfter` doesn't define a strict order (only that one property should be after another). XML works but a section in the XML file would be needed for every derived class because the ordering does not automatically apply to derived classes”. What would have been better? “[`@UiComesAfter` requires] naming lots of properties (that might be in superclasses) in the annotations [which] breaks modularity a bit. The superclass might change after all and that could break everything if you name specific properties. I would rather give the properties *priorities* so that I can say 'this one comes first' instead of 'this one comes after that other one'. It's just more natural to me”. With respect to the XML approach “I found that I could get it to work, but I had to specify that [the] subclass should inherit from [the] superclass in the XML file. I guess the real issue is that I think this inheritance should be implied by the fact that [the] subclass extends [the] superclass [in the Java code]”.

Finally, the discussion turned to the overall usefulness of Metawidget. “I don't do enough front-end coding to speak on the overall usefulness of this type of tool in general. However I'm not very good at UI development (and I don't enjoy it), so I find it very useful to be able to say 'I

want a UI for this bean' and have one generated automatically without any work on my part". So Metawidget saved you writing code? "I think Metawidget's binding implementations are critical. Without them you still have to write [UI helper code] for every class that touches the widget for every property (to fill in the values). My main use case was allowing plug-ins to add data classes that the user can interact with without [the developer] needing to do any UI coding".

On the whole a success? "I was really happy with Metawidget and I will probably use it again… I have thought about using it to build UIs for OWL individuals (the properties would be extracted from the ontology and that sort of thing)".

5.3.1.1.1.2. Reflection

This adoption study contained a further reinforcement of the 'do not dictate the architecture' tenet. The developer had written a similar tool for a Python-based front-end, but could not re-use it here. The developer had also written a similar tool for an OWL-based back-end, but again could not re-use it here. If a tool is written specifically for a front-end or back-end technology, changing either will likely exclude the tool as a candidate for future projects. Such short-sighted design decisions are understandable in homebrew projects – they are less forgiveable in dedicated frameworks.

It is particularly worth emphasising that integration with existing architectures was important even though "the API was written with Metawidget in mind". The developer expressed that even for new projects – where one is free to choose tools and frameworks and, having chosen them, able to make concessions as to how they will integrate – it is impractical for any one tool to make strong demands about its place in the whole: "if I'd had to [design the API] differently [just] because of Metawidget I would have been unhappy".

This adoption study also highlighted areas in need of improvement. The practitioner found Metawidget's existing methods for ordering widgets too awkward in scenarios involving inheritance across a class hierarchy. The shortcomings were twofold: the XML-based metadata was implicitly ordered but required explicit declaration of the class hierarchy (we will return to this in 8.3.3); the annotation-based metadata implicitly followed the class hierarchy, but was not based on priorities (we will return to this in 6.1.1.2). This last point was another reminder that different developers prefer different techniques: "it's just more natural to me".

Another negative comment was that the developer had difficulty attaching event handlers to

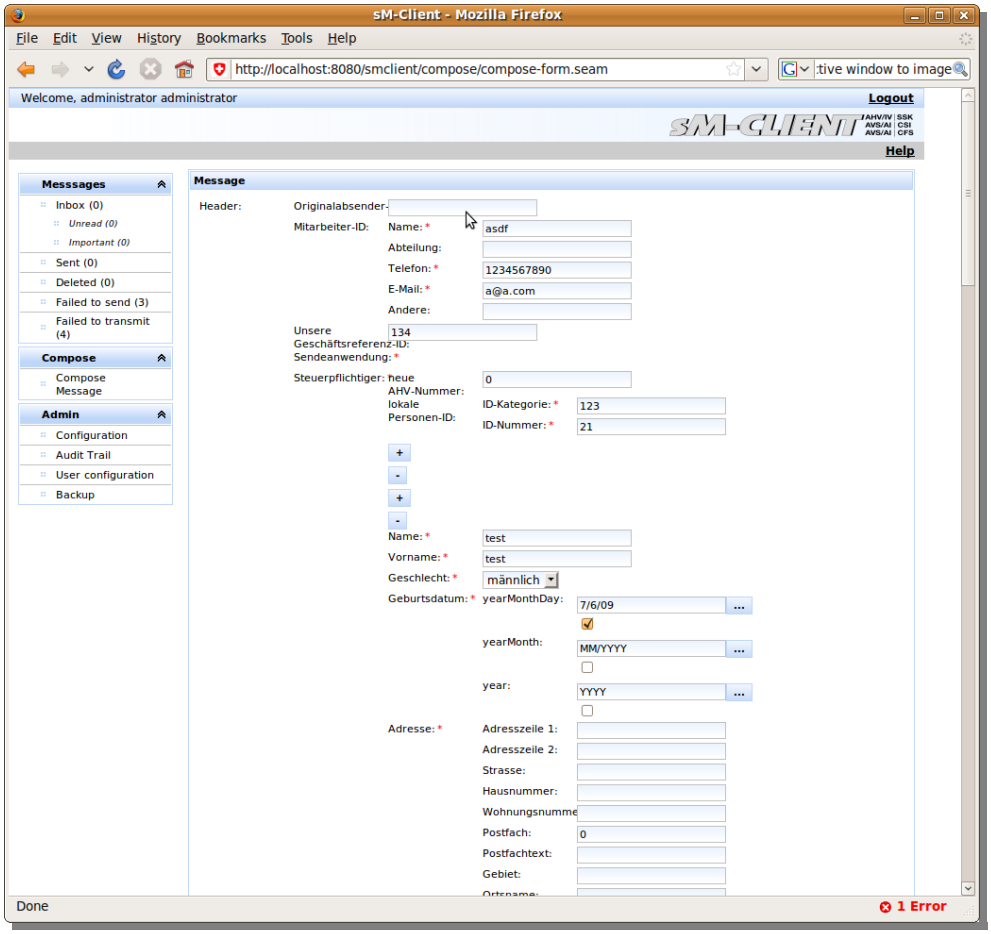
generated widgets. This stems from a general observation that whatever is automatically generated becomes much more opaque to, and less controllable by, the developer. There is an impedance mismatch between the API the generator exposes and the native API of the target platform. Metawidget has this to a lesser degree, because it 'owns' the UI to a lesser extent, but more work should be done to reduce this mismatch where possible. We will return to this in section 6.1.1.1.

5.3.1.1.2. Adoption Study 5

Industry: Swiss Government

Application: Web application

Technologies: JSF, Facelets, JAXB



5.3.1.1.2.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

The project was a tool for use by the Swiss government. They had already built a platform for the creation and transmission of confidential, encrypted XML messages but to date these messages were only able to be created by machines. The government was now looking to build a Web interface so that humans could easily create the messages.

I began by discussing what led the developer to find Metawidget: “I didn't find it, a colleague of mine did. He knew our project and I think he actively went and looked for something we could

use. We wanted something that we could use to change our XML [messages] into a [UI] form". The developer explained there were many types of XML message, and the prospect of developing (and maintaining) UI representations of each was onerous. The team considered several choices: "things I've also looked at include XML-Forms and writing custom XSL stylesheets [to convert the XML into HTML]. Metawidget is a way better solution than the other options".

What were some of Metawidget's strengths? From a front-end perspective "[being] able to integrate our own validation and custom rendering of components". And from a back-end perspective "[being able to write our own] inspector that knows our XML schema and can find all restrictions of the currently inspected field and add that to the attributes returned". Finally Metawidget was architecturally "very intuitive, [the] names are well chosen". Where was Metawidget less successful? "For some specific things, for instance I wanted the fields to be in a specific order you have to extend some of the Metawidget classes, would be better if Metawidget was even more pluggable".

Finally, what was the developer's position on mixing UI metadata and business model code? "I don't mind about that, an annotation is just metadata". And is Metawidget solving a current and prevalent problem? "Yes, it's solving a problem, but to say that its one of *the* prevalent problems in software development is a bit much I think".

5.3.1.1.2.2. Reflection

This adoption study was notable for the amount of customisation the practitioner required. He needed to be able to plug-in both his own project-specific front-end ("own validation and custom rendering") and his own project-specific back-end ("inspector that knows our XML schema"). Such adaptability to different architectures was critical, in fact he would have liked it to go further and be "even more pluggable". We will revisit this in section 6.1.1.2.

Also notable was this project's use of the Java API for XML Binding (JAXB). JAXB is a technology for creating and parsing XML-based representations of object data. In many ways, XML could be thought of as the machine interface to a system just as a UI is the user interface. Indeed JAXB takes a similar approach to Metawidget – it inspects the existing object data and class structure in order to determine an XML representation. Where JAXB differs from Metawidget is when additional, XML-specific metadata is required. Here, JAXB requires the developer to supply the metadata through its own, JAXB-specific annotations. For example to

declare that a property in a business class is a required field, the developer must annotate their class with `@XmlElement`:

```
class Person {
   private String mName;
   @XmlElement( required = true )
   public String getName() {
      return mName;
   }
}
```

Unlike Metawidget, JAXB doesn't support mining this metadata from arbitrary sources. It is interesting to consider, though outside the scope of this thesis, the contribution software mining could make to this Object to XML mapping. For example it would seem likely the name property would also have a corresponding database schema:

```
TABLE person (
   name varchar(30) NOT NULL
);
```

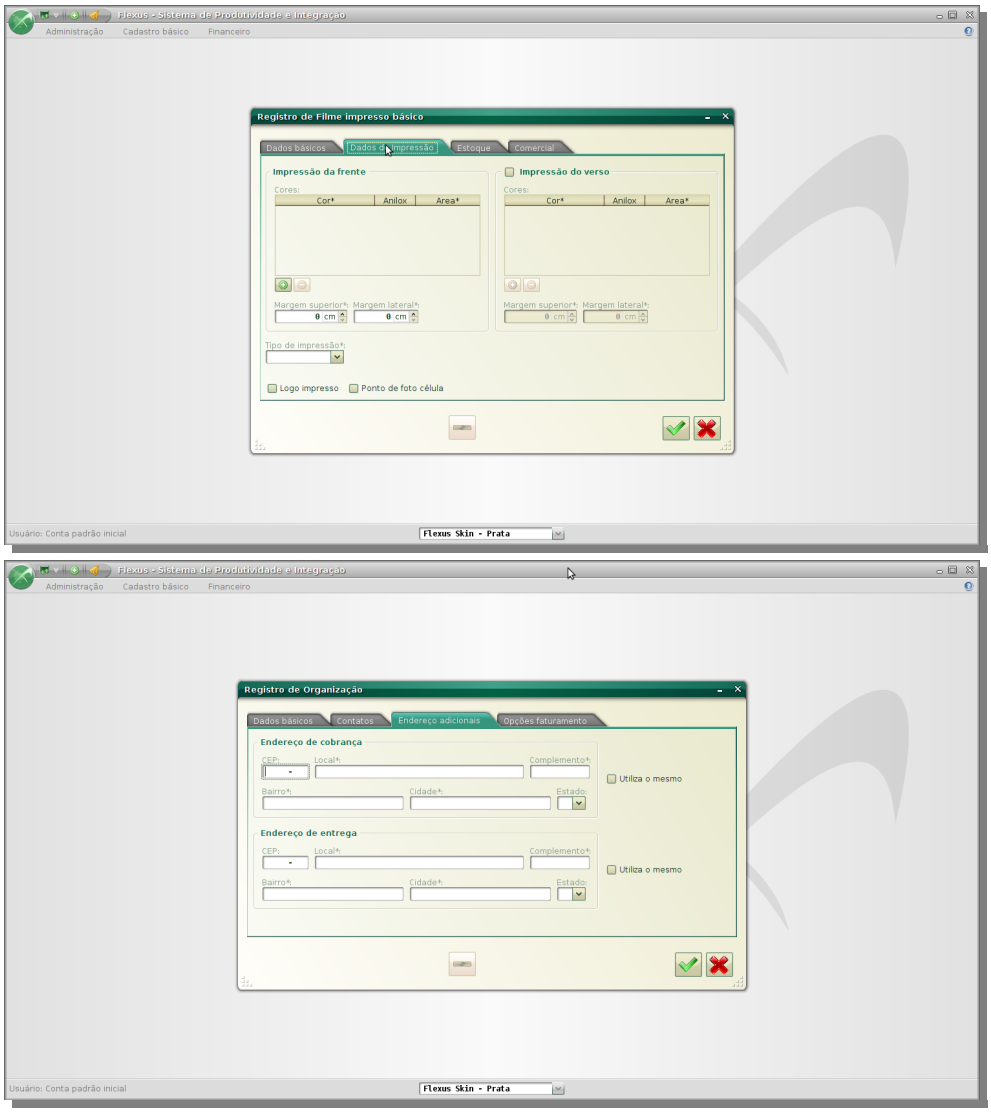
Theoretically it would be possible to remove the need for the `@XmlElement` annotation, and hence any margin for error in specifying it, by using software mining to extract the metadata directly from the database schema. Metawidget's architecture already promotes a strong separation between its software mining inspectors and any UI-related code (see 4.1.1.2). It may be possible to reuse its existing inspectors for other purposes such as this. However this is not a direction I will be exploring further and I acknowledge it as a boundary of my thesis.

5.3.1.1.3. Adoption Study 6

Industry: Brazilian packing industry

Application: Specialised Enterprise Resource Planning (ERP)

Technologies: Swing, OVal



5.3.1.1.3.1. Synopsis

This adoption study interviewed a senior developer on the team.

As with all adoption studies, I began by discussing how the team encountered Metawidget. “We were working on another project where we started doing some dynamic UI generation… much more simple stuff, like printing the required fields and formatting the numbers based on the

model… and then we read an announcement on [an industry Web site] and so I bookmarked it to take a look for the next project". When their next project began, the team "started from scratch with Metawidget in mind".

Using a third-party solution instead of developing one in-house has clear advantages. But why Metawidget in particular? "We've taken a look at other frameworks, but most of them were inactive, poorly documented or not as flexible as Metawidget". With Metawidget "the ability to extend it (property styles, inspectors and widget builders) to fit to your needs [was attractive]". Was the design of Metawidget clear to them? "Totally, very well designed and documented". And its focus on integrating with existing architectures important? "Yes, we work with JDO, OVal, and some custom annotations, so being able to extend [Metawidget] was a must for us. [Being able to integrate a data binding library] is fundamental for us, because a lot of dirty work comes from creating the bindings".

Where did the team find Metawidget lacking? For one, when doing "complex layouts… we had to extend the [built-in] layout to support more complex layouts. [This is] not necessarily a lack, since Metawidget provides means of doing it, but it would be interesting to have more powerful layouts ready [out of the box]". What else? "When you want to customise [the components Metawidget generates], like replacing or adding more info to [them] you have to refer to them by property names. We have this problem not only for Metawidget, but when you have a lot of dynamic stuff". By this the developer meant that many dynamic frameworks, such as ORMs, have the same problem. "It would be nice to either solve this or offer a solution for that (maybe tooling?)".

The team also found Metawidget lacking because of their position on mixing UI metadata and domain model code: "for small projects it might not be a concern, so we find [the fact that Metawidget supports] it valid, but for larger projects where the architecture is more important, usually we want to keep a clear separation between layers, and it is not desirable to 'pollute' the model… we did not want to place view stuff into the model, like `@UiComesAfter`, and we did not want to place it in an XML file either, because we would have to replicate the property name. [Instead] we built a property style based on our properties files". Metawidget supports pluggable property styles, allowing the developer to redefine what is considered a property. The team leveraged this with an innovative approach that read the property information from their localisation resource bundles. This allowed them to avoid adding UI information into their domain model, while at the same time avoiding duplicate property definitions.

Overall, the team found they were able to apply Metawidget widely: “all [our] UIs [screens] have Metawidget behind [them], even the complex and most important. We wrote our own layout and widget builder. An interesting thing to notice is that in [second screenshot at start of adoption study], both tables are created by Metawidget. They are [custom] components we developed and Metawidget instantiates them. The whole screen is generated dynamically by Metawidget”. The practitioner then reflected on the technology as a whole: “I think that there are two main problems that Metawidget helps to solve: Independence of View Technology and Dynamic UI Generation, both [these problems are] current and prevalent. Regarding the first one, I think that although it is theoretically possible to solve this, in practice it is generally not feasible to re-write the view into different technologies [automatically]. Even in scenarios where you have to design, for instance, the same screen with different versions for desktop and mobile, the screen cannot fit/support the same functionality”. The practitioner approved of the way Metawidget does not try to 'own' the UI (see 2.1.1.3). Rather it positions itself as just a piece of the larger UI landscape, recognising the uniqueness of different platforms and devices.

Regarding the second problem the developer identified, Dynamic UI Generation: “Metawidget is really useful in the way it is the foundation to build your solution. We had an experience in our last project, that a lot of view related bugs would come from missing required fields, wrong formatting and changing the model and not changing the view. Also, keeping those in synch, required a lot of effort, not complex, but we had most of our junior programmers dedicated to fixing those silly problems. That is when we thought that generating it based on the model would solve this, and [when] this really happened, it simplified a lot and this category of bug has simply disappeared. Another great advantage [of UI Generation] is UI standards. It is really hard to keep consistency, visual or functional standards when building UI in a large team. However when it is generated dynamically, the rules are centred and even the customisation is somehow controlled”.

“Besides that, for simple input interfaces, or prototyping, it is simply amazing. You have to do nothing and you have a fully functional UI. [In production] we have really complex UIs, since we have a strong usability concern. Another approach that tries to solve this problem is using [statically generated] MDA tools. I personally dislike this solution, mainly because of the idea that I find really important: 'everything that you create (generate) you have to maintain'. I did not find solution with a decent support for maintaining the generated code, and a lot of garbage code is generated”.

If the problem is current and prevalent, why has it not been addressed? “I don't have a clear idea

why it has not been solved yet. It is not a simple question, but I think that dynamic interface generation is a strong tool to address this problem, and the evolution of the hardware, frameworks, languages and runtimes are making this feasible now. Take the [object] reflection performance improvements for instance, hardware capacity, the new view technologies APIs. We have considerably complex screens being generated by Metawidget running on modest old desktops with acceptable performance".

5.3.1.1.3.2. Reflection

This adoption study found a team who had deeply integrated Metawidget into their architecture: they had eschewed both of Metawidget's built-in approaches to providing UI metadata (either annotations on the domain model code, or external XML files) in favour of plugging in their own implementation based on localisation resource bundles; they had developed custom components and plugged in custom widget builders to instantiate them; they had plugged in custom layouts to achieve the exact look they required. Furthermore, the team instinctively understood the problem and were in agreeance with Metawidget's approach to solving it, such as the feeling that statically generated code was impractical. Finally, they experienced tangible benefits to the integration, including: freeing up junior programmers rather than having them "dedicated to fixing those silly problems"; "visual and functional consistency"; even that a "category of bug has simply disappeared".

Despite this positive outcome, there were still further areas to be addressed. Some were straightforward enhancements ("more powerful layouts ready"), but others were more intractable. In particular, being able to refer to generated widgets in cases where the developer doesn't know their name in advance was a "problem not only for Metawidget [but] it would be nice to either solve this or offer a solution for that". This echoes a theme from adoption study 4 that "whatever is automatically generated becomes much more opaque to, and less controllable by, the developer".

We will revisit this point in section 6.1.1.1, during our release candidate Action Research cycle.

6. Action Research: Release Candidate Cycle

This chapter covers the third and final Action Research cycle, which ran from Q1-Q4 2010.

6.1. Planning

This section records reflections from the 'plan' phase of my final Action Research cycle.

6.1.1. Reflections During Planning

6.1.1.1. Widget Processors

As discussed in section 5.1.1.1, one of the primary values of explicit reflection is it enables surfacing multiple issues at the same time, whereby a holistic solution may become apparent. The concept of widget processors was envisioned through precisely this process. Four themes converged:

Firstly, I was encouraged by the positive feedback widget builders (see 5.1.1.2) had been receiving. I reflected that part of their appeal was that I had taken a key Metawidget method `Metawidget.buildWidgets` and abstracted it into a pluggable, orthogonal end-point. Whereas previously (and commonly) practitioners were needing to use inheritance to extend and override `Metawidget.buildWidgets` to alter its behaviour, now I had provided a lightweight, composition-based approach. In turn, this allowed a multiplicity of behaviours that were not possible before, because inheritance in Java is single whereas composition supports arbitrary dimensions. It became possible not only to easily support third-party widget libraries, but to *mix* multiple libraries in the same UI. This reflection started me thinking about other commonly overridden Metawidget methods that might be usefully abstracted into end-points.

Secondly, one of the adoption studies (see 5.3.1.1.1) had lamented that the automatic generation made some common UI development tasks, specifically attaching event handlers, more opaque to the practitioner. It seemed what was needed was a mechanism to 'get inside' the generation process and be able to attach event handlers during widget construction.

Thirdly, another of the adoption studies (see 5.3.1.1.3) had exposed a weakness around

referencing generated widgets. Currently, the only way to reference a generated widget – say, to change its background colour because it was an important field – was by its property name. This could be problematic in highly dynamic systems where names were not known in advance, so the practitioner had wanted a richer mechanism for identifying a widget. It seemed an ideal solution would be to give the identification mechanism access to the same detailed set of inspection results used by the widget builders. Then it could identify widgets based on name, or type, or length of field, or any other piece of metadata.

Finally, I had been uncomfortable with the introduction of a re-binding method (see 4.3.1.3.1). Although my observations had shown it was critical for some use cases and technologies, it was redundant and confusing for others.

These four themes converged into a pluggable mechanism to post process a widget. Processing could occur following the widget's creation by the widget builder but before its inclusion in the layout. This would enable attaching runtime entities such as event handlers. It would also enable identifying widgets in a way that provided full access to the detailed metadata from the inspection results. Finally it would allow mechanisms such as rebinding to only be included on those platforms where it made sense. I named this mechanism a widget processor, as shown in figure 21.

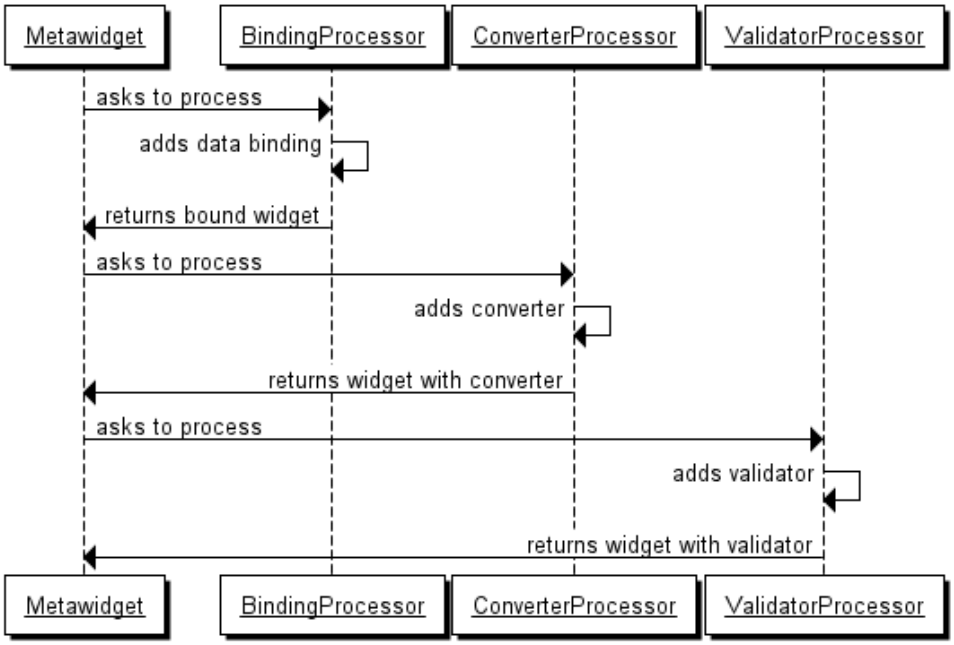


Figure 21: Widget Processors

As I started to implement widget processors, I was encouraged by how much of the existing code could be rolled into them. I was able to refactor the existing binding and verification mechanisms (see 4.2.2.2) into one unified, over-arching concept. In turn, this enabled me to reduce the amount of code, needing fewer interfaces and fewer base classes. Widget processors proved themselves well placed and versatile, and I imagined them being used in a variety of novel ways.

6.1.1.2. Inspection Result Processors

An emergent theme from my Experiment 2 (see 4.2.1.5.2), community feedback (see 4.3.1.4.1) and adoption study 3 (see 4.3.1.5.3) was for different preferences around ordering and excluding fields on each screen. Further discussions with employees at Red Hat (see 5.2.1.1) and researchers within the Naked Objects team (see 6.3.1.1) demonstrated that field ordering was a surprisingly deep topic, with a number of different approaches. There was no clear right answer among these approaches. Many were constructed more from personal preference than any objective measure, but that didn't make them any less important from the perspective of the UI generator. As we have discussed, if the UI generator requires the practitioner adapt to it, rather than the other way around, it will limit its effectiveness.

I sampled a number of field ordering preferences and grouped them into three categories: per screen; per domain; and universal. Per screen ordering required each Metawidget specify the fields to be displayed. For example:

```
<m:metawidget value="#{person}>
   <f:param name="fields" value="name,age,retired,notes"/>
</m:metawidget>
```

This has advantages in that it allows fine-grained control over appearance. Its disadvantage is that it involves hard-coding domain property field names into the UI, which will neither refactor well nor adapt as the domain model evolves. A slightly less brittle suggestion was to only *exclude* fields:

```
<m:metawidget value="#{person}>
   <f:param name="exclude" value="company"/>
</m:metawidget>
```

This is less fine-grained, but adapts better to the introduction of new fields and removal of existing ones. Another suggested improvement was to introduce 'view groups' whereby each

screen specified the group but the domain model retained control over the contents of that group:

```
<m:metawidget value="#{person}>
   <f:param name="group" value="summary"/>
</m:metawidget>
…
public class Person {
   @UiViewGroup( { "summary", "detail" } )
   public String name;
   @UiViewGroup( { "summary", "detail" } )
   public int age;
   @UiViewGroup( "detail" )
   public String company;
}
```

A further improved suggestion, with better type-safety:

```
<m:metawidget value="#{person}>
   <f:param name="group" value="summary"/>
</m:metawidget>
…
public class Person {
   @Summary @Detail
   public String name;
   @Summary @Detail
   public int age;
   @Detail
   public String company;
}
```

Such approaches are more robust but also more stylised, so would not fit every practitioner's architecture or taste.

Per domain ordering suggestions included simple priorities (see 5.3.1.1.1):

```
public class Person {
   @UiOrder( 1 )
   public String name;
   @UiOrder( 2 )
   public int age;
   @UiOrder( 3 )
```

```
        public String company;
    }
```

The more complex Dewey Decimal (Dewey 1965):

```
    public class Person {
        @UiOrder( "1.1" )
        public String name;
        @UiOrder( "1.2" )
        public int age;
        @UiOrder( "1.2.1" )
        public String company;
    }
```

Or class-level ordering:

```
    @UiFieldOrder( "name", "age", "company" )
    public class Person {
        public String name;
        public int age;
        public String company;
    }
```

Universal ordering suggestions included alphabetical and 'order declared in source code'.

From such diversity of approaches, and with the benefit of my holistic viewpoint (see 5.1.1.1), it became apparent I needed to introduce a new concept into the system. This would essentially be an inspector that also had access to each per-screen Metawidget. I was uncomfortable modifying the existing `Inspector` interface, because tying inspectors to screens would compromise their ability to run on remote tiers (see 4.2.1.3) and their potential to be reused for non-UI applications (see 5.3.1.1.2.2). Equally, whilst widget builders (see 5.1.1.2) had per-screen access they currently operated at a per-field level. This would be sufficient for simply excluding individual fields, but not for ordering fields relative to each other.

Instead I introduced a new concept: pluggable inspection result processors. These were named as a symmetry to widget processors (see 6.1.1.1) and designed to operate over the final inspection result returned from the inspectors, as shown in figure 22.

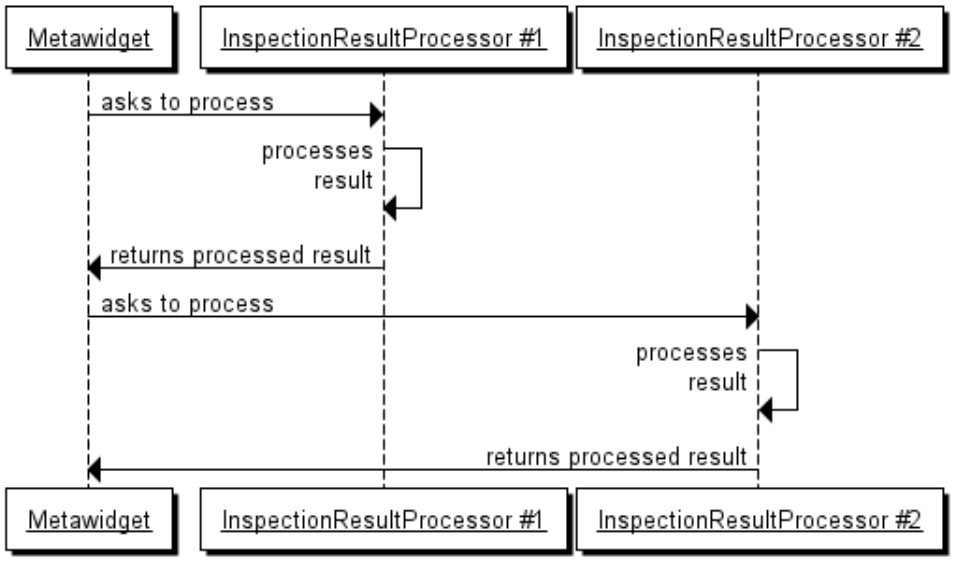


Figure 22: Inspection Result Processors parallel Widget Processors

Inspection result processors had access to the per-screen Metawidget as well as all fields in the inspection result, so could order and exclude them on either a per-screen, per-domain or universal basis. Having introduced inspection result processors, I verified their interface by first extracting the existing `@UiComesAfter` code into a `ComesAfterInspectionResultProcessor`, then providing proof-of-concept and example implementations for each of the ordering preferences I had enumerated[4]. However I did not try and integrate each of these preferences into the Metawidget core. This would present a confusing array of alternatives to practitioners, particularly because some preferences were conflicting in their philosophy. There was never going to be an exhaustive list of approaches. With inspection result processors, as with all parts of Metawidget, pluggability was a more practical goal than completeness.

6.1.1.3. Decoratable Layouts

A theme from two adoption studies (sections 4.3.1.5.3 and 5.3.1.1.3) suggested that my original approach of pluggable layouts (see 4.1.1.5) could be improved. Reflecting holistically, I noticed recurring patterns emerging in Metawidget's architecture for composing multiple plugins together, such as `CompositeInspector` (see 4.2.1.3) and `CompositeWidgetBuilder` (see 5.1.1.2). It seemed an ideal approach to increasing the power of pluggable layouts would be if they too were composable. That way some of the existing layouts could be broken apart and

4 See http://blog.kennardconsulting.com/2010/08/customizing-which-form-fields-are_04.html

decomposed, leaving it to the practitioner to recombine them to suit their needs.

A barrier to this idea was that layouts are not quite the same as other composites. `CompositeInspector` and `CompositeWidgetBuilder` chain their parts together serially, with inspection or widget building passing down their chains sequentially. But it makes little sense to talk of chaining layouts serially: if one layout in a composite arranges a component, and then a successive layout rearranges it, we may just as well have used the second layout to begin with. There is no apparent merit to serially chaining layouts. However there *is* merit in arranging layouts hierarchically, with one layout 'decorating' and delegating to another. One layout could be responsible for, say, arranging its widgets in a two column format. Its decorator could be responsible for breaking different sections into tabs in a tabbed panel. An alternate decorator could be responsible for breaking sections using a horizontal rule.

Formerly, the logic for choosing to render section headings as either horizontal rules or tabbed panes was part of a single layout. If this logic were decomposed it would be possible to split and recombine layouts to suit the practitioner's needs: perhaps they needed horizontal rules inside tabbed panes, perhaps tabbed panes inside horizontal rules. Both these variations are shown in figures 23 and 24.

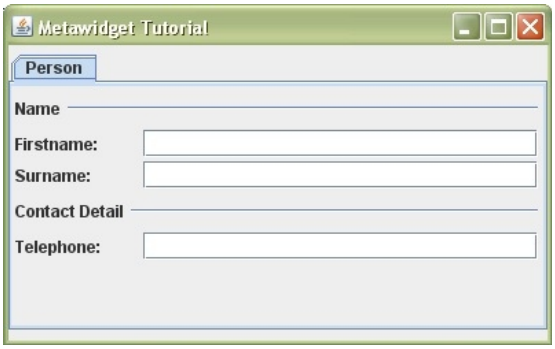


Figure 23: Layout decorated with horizontal rules inside tabs

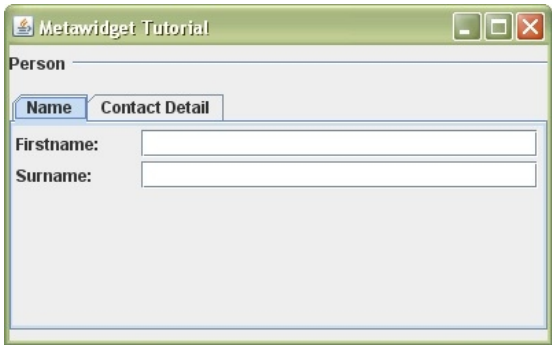


Figure 24: Layout decorated with tabs inside horizontal rules

This approach has further advantages. Specifically, it becomes possible to mix layouts from third-party widget libraries. Third-party libraries often supply layout widgets in addition to their data entry widgets. For example they may provide collapsible panels. If a practitioner is accustomed to using these layout widgets, the UI generator needs to support them or it will not be able to achieve parity with the original UI design.

6.1.1.4. Generation Pipeline

UI design is inherently a mixture of science (data binding, validation etc.) and art (layout, colour schemes etc.). Because Metawidget exists at the boundary of these concerns, it must accommodate a wide variety of preferences both hard and soft. Its approach to managing this diversity is not to expose lots of flags to tweak lots of small variables, but to establish well-defined points for plugging in alternate implementations. These points began as a mixin (see 4.2.1.1) but matured into a 'generation pipeline'. This pipeline, as shown in figure 25, is divided into five stages (shown to the right) and co-ordinated by a Metawidget object (shown to the left). The Metawidget object drives the UI generation process from left to right, starting with a runtime inspection of the object (the software mining) then with a series of stages to build, process and layout suitable UI widgets.

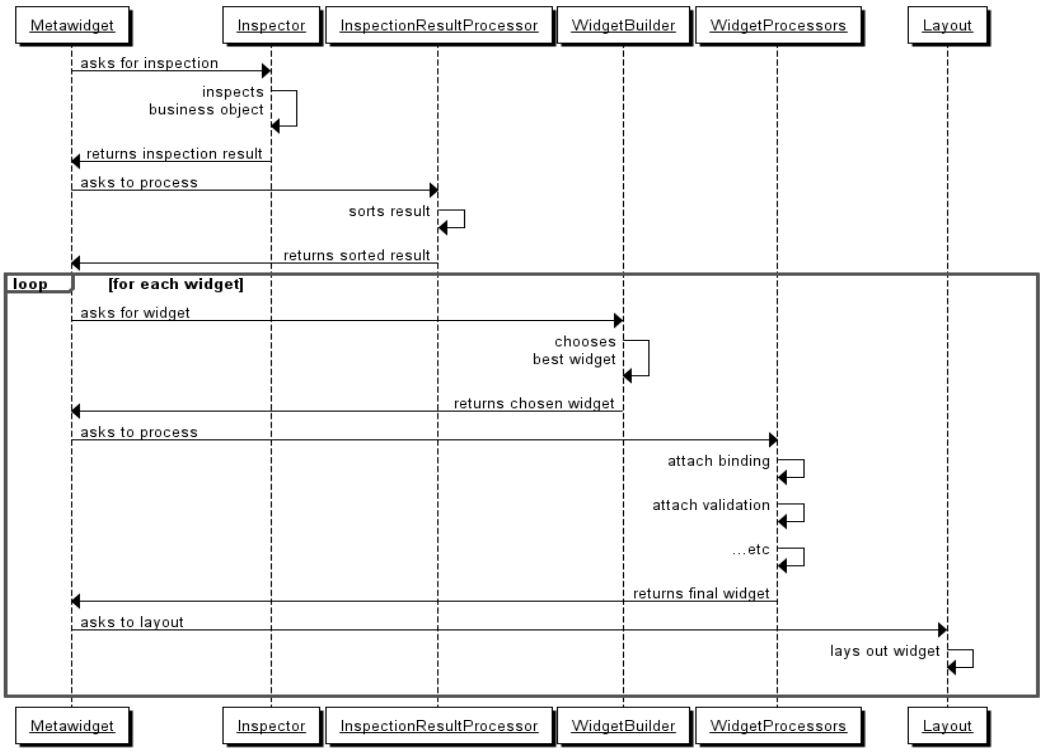


Figure 25: Metawidget pipeline

The most compelling feature of the pipeline, from the perspective of my thesis, is that over half of it arose as a direct result of my Action Research reflections: the inspectors were there from the start, an embodiment of my software mining approach (see 4.1.1.3); the layouts were an obvious initial requirement, albeit in slightly different form (see 4.1.1.5). But inspection result processors (see 6.1.1.2), widget builders (see 5.1.1.2) and widget processors (see 6.1.1.1) were

introduced as a direct result of my adoption studies, observations and reflections. Specifically:

- Inspection result processors provide support for a number of exotic use cases – ones I don't necessarily agree with but were a recurring theme from my observations – such as explicit field ordering (see 4.3.1.4.1).

- Widget builders provide easy support for adding third-party widget libraries, but they also support *mixing* multiple widget libraries in the same project. This is a non-obvious requirement borne out of practitioner feedback (see 5.1.1.2). The high profile third-party widget libraries, the obvious ones to support, generally provide a full suite of widgets (e.g. RichFaces, ICEfaces, ExtGWT etc.). But there exists a vibrant ecosystem of smaller widget libraries that specialise in one area, such as charting components or mapping components. Industry UIs need to be able to selectively mix such libraries.

- Widget processors have proven themselves adaptable to multiple concerns, unifying many of the mechanisms around data binding, validators, component styling and more. These mechanisms were separate and somewhat ad hoc in earlier versions of Metawidget. The introduction of widget processors not only reduced a significant amount of code, it also enabled a new set of possibilities around event handling (see 5.3.1.1.1), widget refinement and widget identification (see 5.3.1.1.3).

The pipeline approach has proven itself flexible enough to accommodate diverse application architectures, including those where the UI layer is: remote from the business layer (e.g. WAR files and EJB JARs); written in a different language to the business layer (e.g. ECMAScript versus Java); uses a mixture of different languages (e.g. Groovy and JSF). Having a pipeline for a UI widget could perhaps seem over-engineered and non-performant, but this concern is mitigated by each element of the pipeline being immutable (see 4.2.2.2.1). Indeed there are cases where the pluggability of the pipeline actually makes Metawidget *more* performant. For example applications that only use a single inspector or a single widget builder can forgo the overhead of `CompositeInspector` and `CompositeWidgetBuilder`. If the composition algorithms were part of the Metawidget, not the pipeline, this would not be an option.

6.2. Acting

This section records reflections from the 'act' phase of my final Action Research cycle.

6.2.1. Reflections In Action

6.2.1.1. Exposure

The final Action Research cycle called for increased exposure within industry in order to solicit feedback. It was one of my VVT measures that Metawidget should see strong adoption and acceptance amongst industry practitioners. To this end I strengthened my previous ties (see 5.2.1.1) with JBoss and Red Hat by organising a number of internal meetings with their senior engineers and management. This ultimately led to Red Hat agreeing to start promoting Metawidget at industry events, with a view to increasing its adoption and user base. Such industry events included JBoss World 2010, Red Hat Summit 2011 and JavaOne 2011.

This increased exposure also had the effect of inspiring other projects to integrate with Metawidget. FakeReplace (FakeReplace 2010), a library for deploying changes to classes without requiring a server restart, produced a Metawidget plugin to allow domain objects to be modified and changes reflected in a UI with no redeploy cycle. Gracelets (Gracelets 2010), a DSL for Web application development, added a Metawidget plugin. Griffon (Griffon 2010), a framework for desktop application development, also added a Metawidget plugin.

6.2.1.2. Journal Article

As with previous cycles, it was important to engage the research community during the 'act' phase of the release candidate cycle. As is to be expected, by this third cycle the research was much more established and required a longer, more rigorous treatment. I produced a journal article exploring my emergent concept of a UI generation pipeline (see 6.1.1.4) and its five stages – being inspectors (see 4.1.1.3), inspection result processors (see 6.1.1.2), widget builders (see 5.1.1.2), widget processors (see 6.1.1.1) and layouts (see 4.1.1.5).

The article generalised these stages into five fundamental characteristics. It asserted that a UI generator must embody these characteristics before it should expect wide industry adoption or standardisation. It supported this assertion with evidence from industry interviews (see 4.3.1.1) and adoption studies (see 4.3.1.5.1 to 4.3.1.5.3, 5.3.1.1.1 to 5.3.1.1.3). The five characteristics it generalised were:

1. *Inspecting existing, heterogeneous back-end architectures*: many business systems are

modelled using what Fowler (2002) calls “anaemic entities”. These are surrounded in an arrangement that Firesmith (1996) describes as “dumb entity objects controlled by a number of controller objects”. Such controller objects include persistence contexts, validation subsystems and Business Process Modelling (BPM) languages. As far as UI generation is concerned, there is a single source of truth (SSOT) but it is decentralised amongst these multiple subsystems. As Shan et al. (2006) enumerate, there are often competing implementations of the same subsystem. Furthermore as Rouvellou et al. (1999) show, different types of subsystems become popular over time, such as rule engines. Any UI generator that seeks to dictate, rather than adapt to, a system's architecture therefore has limited practical value.

2. *Appreciating different practices in applying inspection results*: adoption studies showed the raw inspection result invariably needs post processing before it is suitable for consumption by a UI generator (see 6.1.1.2). For example fields generally need to be arranged in a business defined order, or excluded based on business defined criteria. In some cases this processing can be performed independent of any particular UI screen. For example globally excluding fields that represent database synthetic keys. In other cases it requires knowledge of which UI screen the user has navigated to. For example a summary screen versus a detail screen. Furthermore, different practitioners had different preferences on how to perform such post processing. Some preferred a 'comes after' approach, whereby each business field can specify the field that immediately precedes it. But some adoption studies reported “I would rather give the properties *priorities* so that I can say 'this one comes first' instead of 'this one comes after that other one'. It's just more natural to me”.

3. *Recognising multiple, and mixtures of, UI widget libraries*: practitioners expressed the need to support a variety of front-end frameworks, including third-party and in-house widget libraries. In particular, they talked about the need to mix multiple third-party and in-house widget libraries within the same UI, in order to achieve a high quality user experience. Any UI generator that limits this choice compromises usability – the most determining factor of a UI – for the sake of automatic generation.

4. *Supporting multiple, and mixtures of, UI adornments*: in raw form, a widget is unlikely to be suitable for inclusion in a UI. For example end users interacting with a raw text field are able to enter arbitrary text. However the business requirement may be for, say, a credit card number. Widgets therefore need to be further adorned with data validators,

data binding frameworks and event handlers. Of particular note is that some of these mechanisms, such as a credit card validator, may come from a different third-party library than the raw widget.

5. *Applying multiple, and mixtures of, UI layouts*: a final characteristic was supporting multiple ways to arrange widgets on the screen. It significantly detracts from the practicality of automated generation if it in any way compromises the final product in usability, or even in aesthetics (Myers, Hudson & Pausch 2000). Yet there is a formidable degree of variability. Fields may typically be arranged in a column, with the widget on the right and its label on the left. But other times the practitioner may want two or three such columns side by side. If so, they may need some widgets – such as large text areas – to span multiple columns. Or they may abandon columns altogether and want the fields arranged in a single, horizontal row. Furthermore, it is not difficult to posit other arrangements, such as right-to-left arrangements for the Arabic world. It is important to accommodate this variety if the generator is to achieve the exact look the practitioner desires.

The article was accepted and published in the Journal of Systems and Software (Kennard & Leaney 2010). During review, there was no concern this time regarding the theoretical depth of the work (see 5.2.1.2). Reviewers wrote: "I really like this article. It identifies, as it claims, a number of inhibitors to the automatic generation of UIs and, essentially, proposes flexible solutions in the form of plug ins for these inhibitors… the authors are identifying well the need of their research and are presenting a structured approach to tackle the different challenges that the research on UI generation is facing".

6.2.1.3. Performance Measurements

A common theme from presentations and demonstrations of Metawidget was the question of performance. Given that Metawidget performs software mining and UI generation at runtime, it is expected there will be a performance impact compared to defining the UI statically. Rendering speed and memory consumption will likely get worse. Conversely, deployment size will likely get better because screens are generated on-demand, rather than pre-compiled and distributed with the application.

Unfortunately meaningful performance metrics are notoriously difficult to achieve. Bull et al. (2000) discuss the principal issues affecting benchmark design. They highlight problems such as

choosing benchmarks that are representative (not too small that they do not represent realistic scenarios, not too large that the item being benchmarked is lost in the noise), robust (controlling for such effects as caching and warm-up time of Just-In-Time compilers) and transparent (available to others to reproduce). One promising approach was suggested by a presentation attendee: “Perhaps flush out an existing app, with Metawidget being the only variable. Beat on the two apps for a while and see what's different”.

I had already experimented with retrofitting a number of existing applications (see 5.2.1.3.1) and these were a good size to be representative without being overly complex. A logical next step was to develop benchmarks around them for both 'with' and 'without' Metawidget variants. I designed the benchmarks to be long-running, so as to control for effects of caching and warm-up time, and distributed them with the Metawidget source code, to allow others to verify them. A sample of the results, after running multiple times on different machines, is shown in table 3.

Test Script	Without Metawidget	With Metawidget	Impact
`1.seam-booking-performance-test.xml`	7 minutes, 44 seconds 7 minutes, 55 seconds 7 minutes, 50 seconds	6 minutes, 56 seconds 6 minutes, 42 seconds 6 minutes, 48 seconds	111% 118% 115%
`2.seam-dvdstore-performance-test.xml`	5 minutes, 17 seconds 5 minutes, 17 seconds 5 minutes, 21 seconds	5 minutes, 40 seconds 5 minutes, 47 seconds 5 minutes, 38 seconds	93% 91% 94%
`3.seam-groovybooking-performance-test.xml`	48 minutes, 1 second 47 minutes, 50 seconds 47 minutes, 50 seconds	45 minutes, 1 second 44 minutes, 6 seconds 44 minutes, 33 seconds	106% 108% 107%

Table 3: Performance impact of introducing Metawidget

The results were a surprise. Whilst application 2 ran predictably slower (though encouragingly not significantly slower) at around 93% of its original speed, applications 1 and 3 actually ran *faster*, at around 114% and 107% respectively. To understand this counter-intuitive outcome, it is necessary to dissect the architecture of the applications being benchmarked. All three applications use JSF (2011). Atop that all three use Facelets, an intermediate XHTML-based language for declaring UIs. For those versions of the application not using Metawidget, every UI widget must be fully described in Facelets XML, which the Facelets runtime then parses and converts into JSF API calls. For versions of the application that *are* using Metawidget, far fewer UI widgets need be described in Facelets XML. The Facelets runtime essentially just executes a single Metawidget API call, which Metawidget then maps to the JSF API. Whilst the Facelets-to-JSF and Metawidget-to-JSF code paths are equivalent, they are parallel and do not necessarily share the same performance characteristics: if Metawidget can invoke the JSF API faster than Facelets, the overall speed of the application will improve.

It must be stressed this does not indicate that an application will run faster when using Metawidget. What it *does* indicate, however, is that incorporating Metawidget is likely to have only a minimal performance impact. This is an important outcome, as it suggests performance concerns need not be a factor in choosing whether to adopt Metawidget. My original presentation attendee was satisfied: “Wow those results are surprising I agree, however I bet there are some good reasons. Perhaps your caching is better than [Facelets], or something like that. Either way thanks for the info. I agree with you that tests like this are difficult to make meaningful, but at the very least it is interesting”.

6.2.1.4. DomInspector

Conducting the performance measurements of section 6.2.1.3 piqued my interest to construct some micro-benchmarks. Micro-benchmarks are synthetic, highly-targeted benchmarks that are seldom representative of real-world performance (Bull et al. 2000) but can be useful in gauging where code is spending its time. A profiler employed around the micro-benchmark indicated that a key bottleneck in Metawidget's pipeline was the serialization and deserialization of XML strings.

As often with Action Research reflections, the seed of this idea was reinforced by a previous observation I had recorded but not acted upon. One practitioner had commented, in relation to inspection result processors (see 6.1.1.2) “I find it a bit weird that you have to work with XML to move fields around. I'm assuming that this is the payload you have traversing the processor

life cycle? Why didn't you choose to use a Java type instead? That would seem to me the most flexible option… and would make element ordering and removal a whole lot simpler". At the time I explained to the practitioner that my choice of XML was important to maximise interoperability between different platforms and technologies (see 4.1.1.2). They accepted this, and it was still a valid reason, but in light of my new micro-benchmark it was perhaps not reason enough.

Closer analysis of the Metawidget pipeline revealed that much of the bottleneck of serialization and deserialization could be avoided. For example the inspectors were constructing the XML internally as a Java type (a DOM document) then serializing to a String and returning it – but it was then being immediately *deserialized* by the receiving Metawidget back into a DOM document, as shown in figure 26. If the inspector could simply return the DOM directly as a Java type most of this overhead could be avoided, as shown in figure 27. This would, inevitably, introduce a platform-specific dependency but provided this was optional that was acceptable.

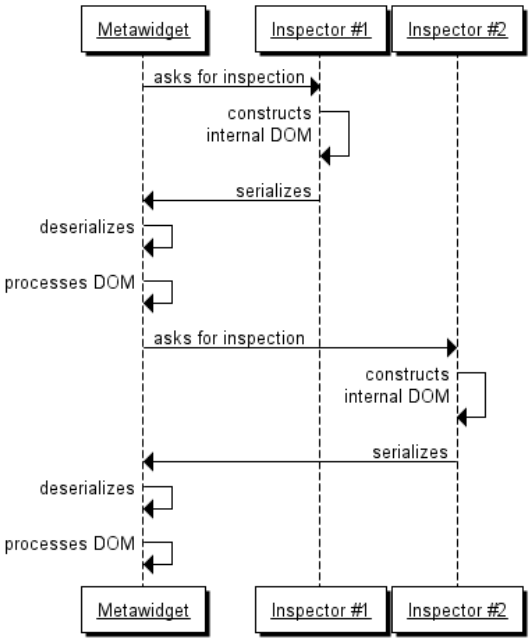


Figure 26: Unnecessary serialization and deserialization

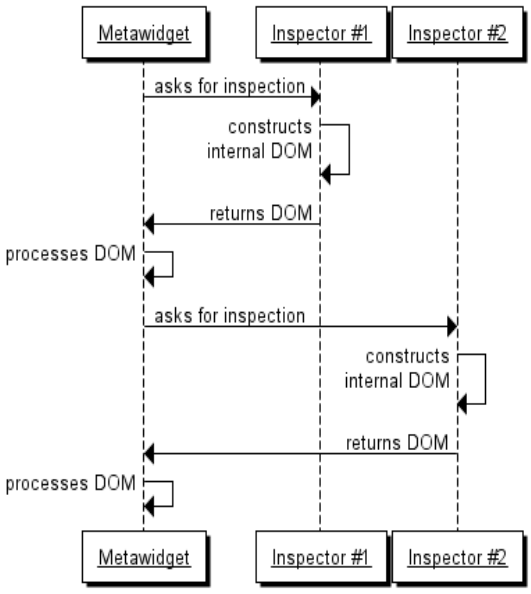


Figure 27: Optimised DOM passing

I reworked the code to introduce an optional `DomInspector` interface. As expected for something discovered during a micro-benchmark, this optimisation had relatively little impact when tested against real applications. I re-ran the performance metrics from section 6.2.1.3 and did not observe any significant improvement. This was understandable given that so little of a

real application's time is spent rendering its UI versus, say, waiting for network latency, invoking business logic or executing database persistence. Nevertheless the introduction of a `DomInspector` seemed a reasonable optimisation, being based on simplifying and short-circuiting a code-path rather than increasing its complexity through, say, additional caching.

6.2.2. Action Outcomes

This section briefly outlines the architecture as it stands at the end of the release cycle 'act' phase. This provides context for the observations and reflections that are to follow.

6.2.2.1. UML

Figure 28 shows the updated Metawidget architecture in UML form[5]. This is a more extensive diagram than figure 20, showing all supported front-end and back-end frameworks. It is not exhaustive – internal classes have been omitted for clarity – but it does convey the level of reuse achieved between technologies. As of the release candidate Action Research cycle, Metawidget supported some 8 front-end platforms and 15 back-end platforms. Clearly, developing and maintaining code for 8 UI platforms requires more engineering effort than developing for 1. Importantly, however, it is not 8 times more work. There is significant overlap. With careful planning, supporting multiple technologies can simplify and strengthen, rather than complicate, an architecture.

Figure 28 illustrates the impact that the introduction of widget processors (see 5.1.1.2), inspection result processors (see 6.1.1.2), decoratable layouts (see 6.1.1.3), `DomInspector` (see 6.2.1.4) and a Generation Pipeline (see 6.1.1.4) had on the architecture. It can be seen these abstractions (depicted in white boxes across the top of figure 28) are defined at a high level, neutral of any particular technology, and reused across all implementations.

5 Larger version available at http://metawidget.org/wallchart.php

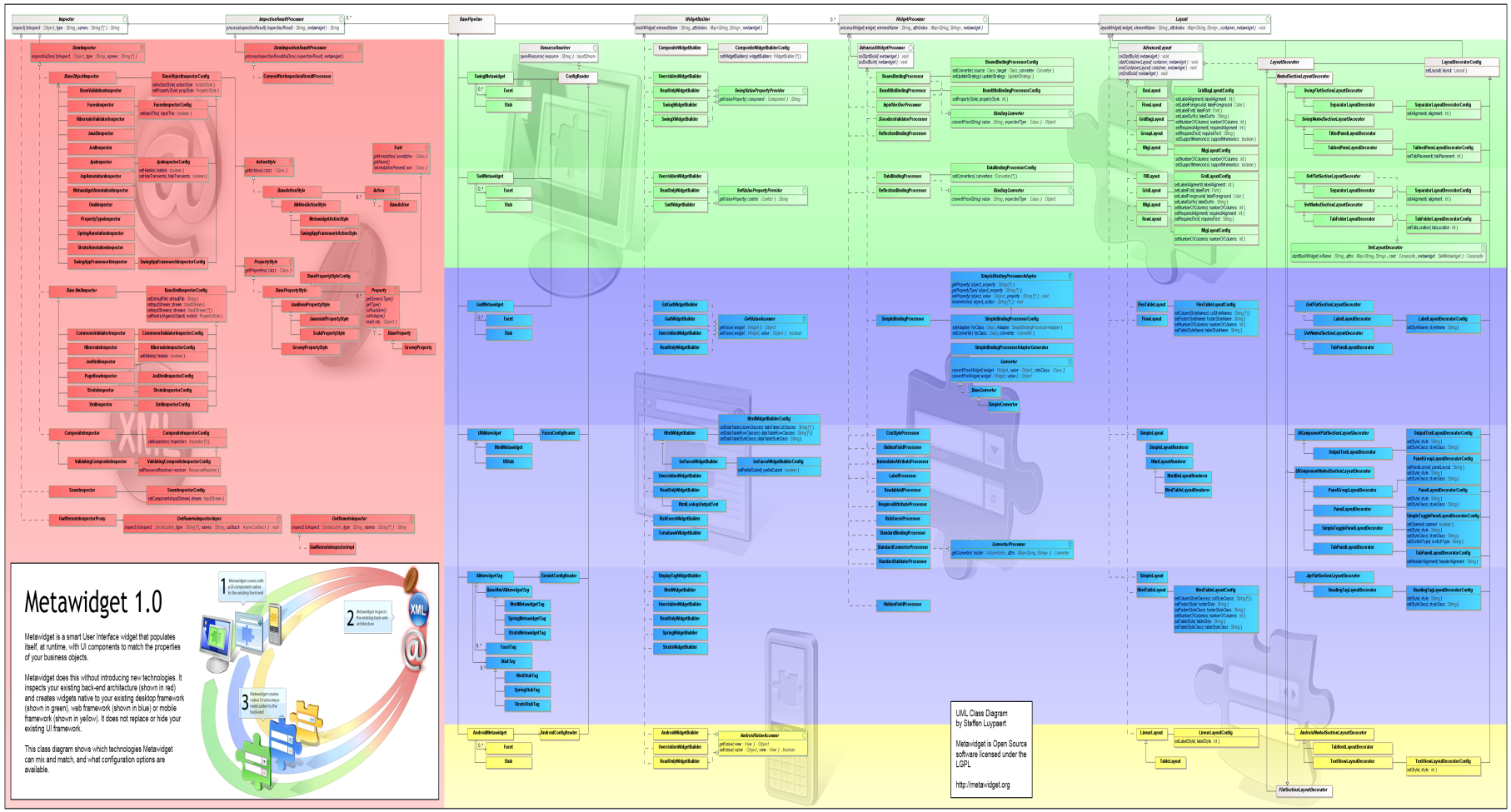


Figure 28: UML class diagram of Release Candidate Action Research Cycle

6.3. Observing

This section records reflections from the 'observe' phase of my final Action Research cycle.

6.3.1. Reflections Following Observations

6.3.1.1. Blogs

This thesis has been critical of the Naked Objects (Pawson 2004) approach. In particular, that by attempting to generate an entire UI it compromises usability for genericity (see 4.1.1.4). Furthermore, that its behaviourally complete methodology limits its applicability to industry because, by Pawson's own admission, most business systems are not behaviourally complete (see 2.1.1.3). My blogs on these subjects did not go unnoticed.

Dan Haywood, one of the lead developers on the Naked Objects project, commented on one of my blog articles. He wrote “Just picking up on [your blog article's reference to] the Naked Objects thing… you are right that a Naked Objects GUI will always be generic, though… in Naked Objects 2.0 [the appearance] is improved somewhat and is now rolled out to over 800 users for the Irish Government. But, in practice we also agree that there will be many cases that a naked domain object needs to be skinned [so that it appears less generically]… Another thing that Metawidget and Naked Objects [now] share is the idea that the metamodel can be built up using more than one source. I think with Metawidget you are in some ways ahead of us, though in the latest version [of Naked Objects] we've completely rewritten the metamodel builder… our new [metamodel] is also pluggable [to accommodate scenarios where the domain model is not behaviourally complete]”.

Through these observations, I discovered that the Naked Objects team had made a number of changes to their original approach, and that our two teams were converging on my five characteristics (see 6.2.1.2). This prompted an interview and ultimately the publication of a journal article in collaboration with the Naked Objects team (see 7.1).

6.3.1.2. Validation, Verification and Testing

Much as the final phase of each Action Research cycle has a slower, more deliberate 'action present' that allows for more thorough reflection, so the final cycle of all Action Research cycles

calls for a deeper VVT. As such we will devote a separate chapter, chapter 7, to the last round of observations. Our focus this time will be more on validation than on further planning, because at this stage active development on Metawidget was drawing to a close. A final, version 1.0 release was imminent. We will follow up with reflections on future work and concluding remarks in chapter 8.

7. Validation

This chapter covers the observations from the third and final Action Research Cycle. The increased exposure and adoption of Metawidget afforded the final cycle a deeper observation phase. Coming at the end of my research, this phase was more focused on validation rather than insights into further planning. Any future work that emerged will be discussed in Chapter 8.

7.1. Research Community Validation

As discussed in section 6.1.1.4, the majority of my planning and acting in this thesis were directly informed by my observations, reflections and adoption studies. Some of the core features (inspectors, layouts) were part of my initial design, but many of them (inspection result processors, widget builders, widget processors, generation pipeline) arose through close consultation, verification and validation with industry practitioners. This was driven by a deliberate focus on industry practicality.

In a 2010 journal article (see section 6.2.1.2) I detailed five fundamental characteristics a UI generator must embody before it should expect wide industry adoption or standardisation. I supported these five characteristics with evidence from industry interviews and industry adoption studies. A further source of validation would be to see if other research teams, who were also conducting industry field trials, were independently converging on this same set of characteristics. If they were as fundamental as I believed, other teams should have been identifying similar constructs. In a 2011 journal article (Kennard & Leaney 2011) I looked for such validation by interviewing the team behind one of the research community's most significant UI generators: Naked Objects. Clearly this was a qualitative measure, not quantitative, but had validity in conjunction with my previous work as a triangulation between industry and the research community.

In section 2.1.1.3 I had been critical of Naked Objects for its behaviourally complete methodology. Specifically that this limited their applicability to industry because, by their own admission, most business systems were not behaviourally complete. However this criticism was levelled at the version of Naked Objects last described by the literature (Pawson 2004). Like any good software project, Naked Objects had evolved over the intervening years. I contacted the Naked Objects team for an update on their work.

7.1.1. Methodology

I interviewed Dan Haywood over a series of e-mails in late 2010. Dan is a UK-based freelance consultant specialising in enterprise application development using domain driven design approaches and agile development. He is the project lead on the Apache Isis project (the effort to standardise Naked Objects within the Apache Software Foundation), the author of "Domain Driven Design using Naked Objects" and a long-time advocate of the Naked Objects pattern. I had come into contact with Dan through my earlier blog entires (see 6.3.1.1).

I chose a standardised, open-ended format for the interview (Valenzuela & Shrivastava 2002). This approach involves asking broadly framed questions to allow the candidate room to talk openly, avoiding leading the interviewee and therefore minimising bias. The overarching theme of the interview was the current feature set of the Naked Objects architecture. The interviewee was asked to discuss, and contextualise, aspects of their approach. The goal was to draw out decisions underlying the original design and motivations that led to subsequent changes. I observed patterns of convergence, then posited probe questions (Dick 2005) to drill down and confirm observations. Unlike previous interviews (see 4.3.1.1) I was not looking to generalise or codify. Rather every comment, even in isolation, was valuable to help understand the team's research.

The Naked Objects team and I had worked independently up to this point, but we learnt much about each other's work during the course of the exchange. Most notably, we discovered a significant amount of convergence around the key characteristics previously identified (see 6.2.1.2). The findings are recorded in the next section, framed in the context of the five characteristics.

Interview

Dan started the discussion with an overview of the current design. In his original thesis Pawson (2004) had summarised: "Using the Naked Objects approach to designing a business system, the domain objects are exposed explicitly, and automatically, to the user, such that all user actions consist of viewing objects, and invoking behaviours that are encapsulated in those objects". This results in an OOUI as shown in figure 29. The UI is a direct representation of the domain objects, with UI actions explicitly creating and retrieving domain objects and invoking an object's methods. The advantage of this approach is that the UI can be built and reworked very

rapidly from the domain.

Dan then elaborated on the architecture. “First off, in terms of what Naked Objects/Apache Isis actually *is*, these days I think of it in terms of the hexagonal architecture (figure 30). The hexagon core has got two main bits to it – the metamodel (a class) and the runtime (an object). Plugging into the hexagon are the back-end object stores [labelled 'persistence' in figure 30], and the front-end viewers [labelled 'main' and 'webapp' in figure 30]”.

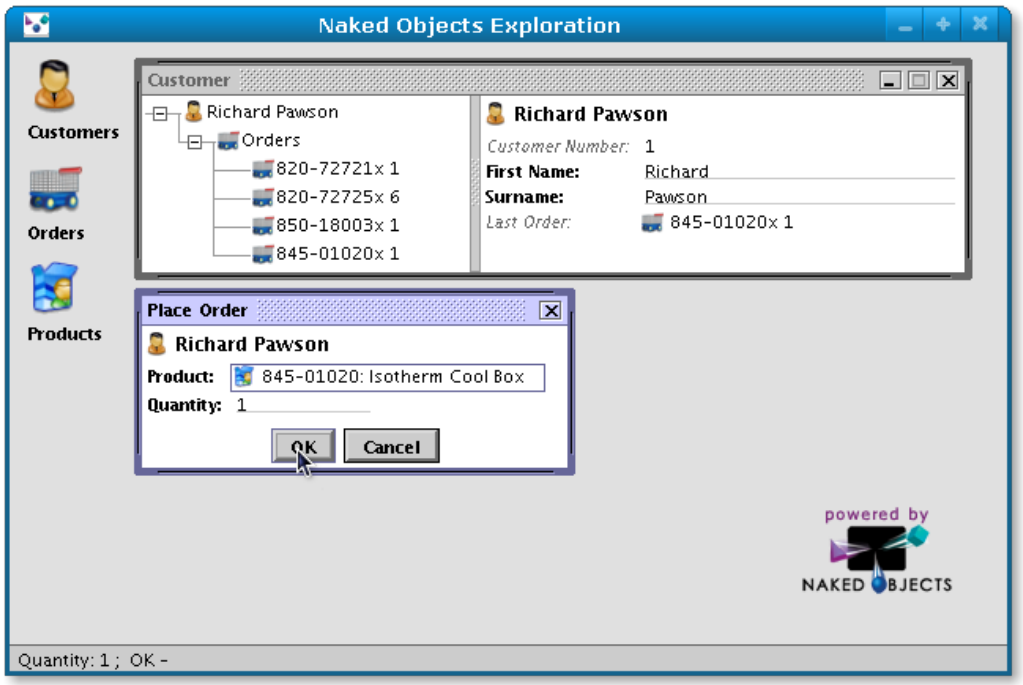


Figure 29: Naked Objects' Object Oriented User Interface

“The metamodel [the hexagon in figure 30] defines the `ObjectSpecification` which describes the class, its inheritance hierarchy, its class members. The runtime defines the `ObjectAdapter`, which wraps each [domain object]. This references the `ObjectSpecification` and also references an opaque Object Identifier (OID), basically an abstraction over primary keys (since it is assigned by the object store) though non-persisted objects also have an OID. The runtime also manages the identity of the (entity) object, each of which is identified in a multiway identity map of [domain object] <-> `ObjectAdapter` <-> OID. The back-end object store implementations deal mostly with the runtime; some use the metamodel (e.g. XML object store), some don't (e.g. JPA object store, because [JPA] builds its own metamodel). The front-end viewer implementations deal mostly with the metamodel in that they use it to render the [domain objects]”.

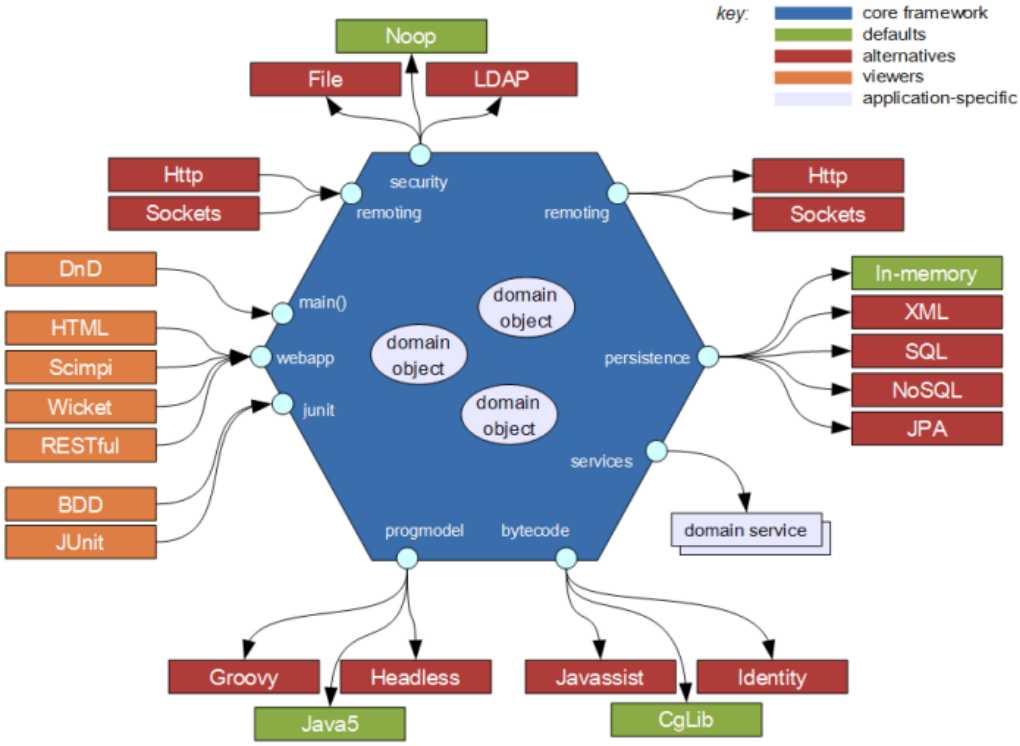


Figure 30: Naked Objects hexagonal architecture

The interview then moved to discuss back-end object stores, the metamodel and front-end viewers in detail.

7.1.1.1. Inspecting existing, heterogeneous back-end architectures

I had previously identified (see 6.2.1.2) that supporting a mixture of heterogeneous sources of UI metadata was an important characteristic for a practical UI generator. Interestingly, the latest release of the Naked Objects framework included a concept called 'facets'. I wanted to clarify if these were in the original design? Dan Haywood responded: “No, they weren't”. What was their background? “Up until 3.0 (late 2007) I had actually been working on my own [clean room implementation of a Naked Objects framework] based on Eclipse RCP. For various reasons, I wrote it off. But all was not lost: a lot of my thoughts on what the programming model should look like went into Naked Objects 4.0 (2009). I also had become enamoured with the extension object pattern, something used a lot in the Eclipse APIs. It was this that eventually evolved into facets”.

Dan explained that facets were a form of an extension object pattern, allowing capture of metadata from heterogeneous sources. How did this fit in to a Naked Objects architecture? Dan

explained “Rob [Matthews] and I started refactoring the Naked Objects metamodel to bring in this idea [of the extension object pattern]. My original idea didn't go much beyond representing the [existing Naked Objects] annotations as facets… like all good collaborations, one of us (and I'm pretty sure it was Rob) realised that the imperative helper methods could also be captured as facets too”. Is it one facet per technology? “No. It's one facet per piece of information to be captured. A collection of facets define the Naked Objects `ProgrammingModel`. Suppose there's a JPA annotation (or bit of XML, it could be) to indicate that a field is nullable… that would correspond to a `JpaMandatoryFacet`. And if we wanted to capture which property was the `Id` (which I do, to manufacture the framework's internal identifier) then there's also a `JpaIdFacet`. This is what I meant about a programming model: the `JpaProgrammingModel` is the collection of the `FacetFactories` for detecting these features/semantics/pieces of information and adding them to the code”. Can a facet be targeted outside of the entity (i.e. XML files, database schemas, rule engines)? “Yes… a `FacetFactory` can pick up information from anywhere. We have a little example showing how names could be picked up from a flat file”.

In short, are facets close to Metawidget's inspectors and its `CompositeInspector`? “Pretty similar, but more fine-grained. I think `CompositeInspector` = a Naked Objects `ProgrammingModel` = collection of `Facets`. But it'd be good if we went closer to [Metawidget's] design, with an inspector = a group of related facets that shouldn't be split apart. Our facets are too fine-grained and I think we should instead be dealing in an aggregation of facets, which I'm calling a `ProgrammingModel`, basically equivalent to your `Inspector`. Then, another idea I intend to borrow is that of `CompositeInspector` which for us would be a `CompositeProgrammingModel`”.

I discovered that with their introduction of facets, particularly XML and flat file facets, the Naked Objects team had effectively extended their philosophy of behavioural-completeness (Pawson 2004) to go beyond just the semantics intrinsic within the code. They had converged on a need to support a mixture of heterogeneous sources of UI metadata. Their implementation differed a little from my own in that it was “more fine-grained”. But the fundamental notion of opening up the Naked Objects framework to metadata from other subsystems, such as persistence contexts, rule engines and XML files, had close parity with my software mining approach.

7.1.1.2. Appreciating different practices in applying inspection results

A second key characteristic (see 6.2.1.2) was to support a variety of ways to post process the UI metadata. For example different practitioners had different preferences regarding how to sort or exclude business properties from the UI.

I discovered the Naked Objects team had also perceived this need. Dan described: "our `FacetFactorys` can optionally implement various additional interfaces. To identify the properties and collections (i.e. identify the main scaffolding of the classes) we look for `FacetFactorys` (typically just one) that implements the `PropertyOrCollectionIdentifyingFacetFactory` interface. These are run through first. I think it might be better to pull this out as a distinct phase of the metamodel building process. Similarly, after we've processed all the `FacetFactories` and added the facets then we go looking for `MemberOrderFacets` to sort the members; it's just a call to a method. Again, it might make sense to factor this out into a separate API".

The Naked Objects team were clearly thinking about introducing post processing into their facets. They were already using multiple passes implicitly – "these are run through first… similarly, after we've processed all the [others]" – and were now seeing that separating this out into an explicit post processing phase may be advantageous. Metawidget had followed a similar evolution.

7.1.1.3. Recognising multiple, and mixtures of, UI widget libraries

Another key characteristic (see 6.2.1.2) was supporting mixtures of widget libraries. This included mixing multiple third-party widget libraries, and the practitioner's own custom widget libraries, in addition to the UI platform's standard widget libraries.

I wanted to gain an understanding of how this characteristic had been handled in the original Naked Objects. Dan recounted: "the different viewers implement this differently. The original viewer had something similar, but restricted to just using AWT. That's fine, but it does mean that a developer wanting to extend the viewer has to learn all this new API".

Had newer viewers tried to incorporate better support for third-party, or custom, widget libraries? "Talking about the Apache Wicket viewer, the API is actually called

`ComponentFactory`. Part of the reason for using that terminology is to use a term that's already known by Wicket developers who might want to extend the UI generated by the Wicket viewer. I have a registry of `ComponentFactories`, which are asked in a chain-of-responsibility pattern to render model objects. They may use third-party libraries if necessary". Can you compose multiple `ComponentFactories` in one project? "Yes. Basically the Wicket viewer is just a registered collection of `ComponentFactories` that can render any entity or collection of entities. But the list is pluggable so that custom widgets can be provided if required". Can you specify precedence? "Yes, it's a first-come-first-served. So, any `ComponentFactories` picked up on the classpath are placed before the defaults. But our programmatic approach provides full control".

In short, are they close to Metawidget's widget builders and its `CompositeWidgetBuilder`? "So, yes, kind of similar. But, as I say, [`ComponentFactories` are] an implementation detail of each viewer – the nature of the API is not standardised across viewers. In theory that sounds like a good objective, and I think it's something you've managed to achieve with Metawidget. I'm hoping that within Apache Isis we'll be able to move some of this stuff into core, so that it can be reused more widely. It might also make sense to move the `ComponentFactoryRegistry` stuff there too, though I'd need to figure out how to remove any Wicket-specific stuff".

The Naked Objects team had progressed from originally using proprietary, low-level APIs to fully supporting pluggable third-party libraries via `ComponentFactories`. In the future they hoped to back port this approach from its current per-viewer implementation into the Naked Objects core proper. There was clear convergence in this characteristic.

7.1.1.4. Supporting multiple, and mixtures of, UI adornments

I had further identified (see 6.2.1.2) that supporting a variety of ways to post process UI widgets was an important characteristic for a practical UI generator. Once created, widgets may need to be adorned with such mechanisms as data validators, data binding frameworks and event handlers. Some of these mechanisms, such as validators, may come from a different third-party library to the widget itself.

The latest release of Naked Objects included a concept called 'advisors'. Were these in the original design? "No; we introduced them in Naked Objects 4.0 (2009)". Dan explained

advisors handled such operations as hiding, disabling, and validating components. “To clarify: some facets are also advisors, some facets aren't. There are no advisors that aren't also facets. Consider the [Naked Objects] `@Disabled` annotation. That is going to get picked up by the `DisabledViaAnnotationFacetFactory`, which installs a `DisabledFacet` on the property. When the viewer creates the text box [for the property] it doesn't go looking directly for a `DisabledFacet`. What it does instead is call `property.isDisabled` which iterates through all installed facets looking for those that implement `DisableInteractionAdvisor` (of which the `DisabledFacet` will be one). If one of those advisors or facets vetoes, then it configures the text box accordingly”.

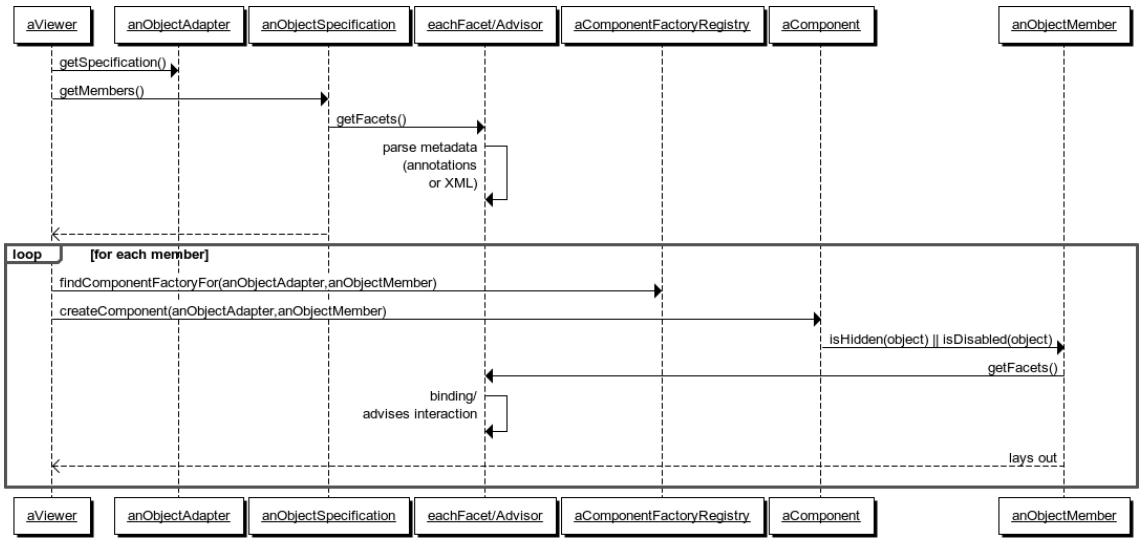


Figure 31: Naked objects sequence diagram as implemented by Isis Wicket viewer

This results in the sequence diagram depicted in figure 31. The sequence has similarities with Metawidget's own pipeline (Kennard & Leaney 2010) depicted in figure 25. In particular, the similarities between facets and inspectors, component factories and widget builders, and advisors and widget processors. Considering neither facets, component factories nor advisors were part of the original Naked Objects (Pawson 2004) this showed strong evidence of convergence.

However Dan's description was of a single object implementing both the `Facet` interface and the `Advisor` interface. These interfaces are roughly analogous to Metawidget's `Inspector` and `WidgetProcessor` interfaces, but Metawidget implements these as separate objects. This is because the former is tied to the back-end architecture whereas the latter is tied to the front-end UI. Is this something the Naked Objects team have considered? “[Yes] all the above said… I'm

not entirely sure that there's any real need for a facet to be an advisor. Where at the moment an `ObjectMember` implements `FacetHolder`, it could perhaps be also an `AdvisorHolder`. This would separate out these concerns, converging our two designs further".

7.1.1.5. Applying multiple, and mixtures of, UI layouts

A final characteristic (see 6.2.1.2) was supporting multiple ways to arrange widgets on the screen. It significantly detracts from the practicality of automated generation if it in any way compromises the final product in usability, or even in aesthetics (Myers, Hudson & Pausch 2000). This realisation exposes a myriad of small details around UI appearance, navigation, menu placement and so on. The problem is so difficult, in fact, I believe it insoluble. Metawidget sidesteps the issue by not attempting to generate the entire UI. Rather, it defines useful bounds to UI generation (see 4.1.1.4).

In contrast, the original Naked Objects viewers did not admit this level of subjectivity. Dan explains: "there's lots of scepticism that a fully generic UI is sufficient [but] I don't think we recognise that… at least not for the enterprise applications that we have built thus far. In the Irish [Department of Social Protection] system there are about 5 or 6 transient entities [intermediate, subjective representations of domain objects] out of over 300 sovereign entities [direct representations of domain objects]. So, the point is… most entities don't need them". Dan summarised "[Metawidget] just provides, well, a widget (a rather large and clever one, but a widget nonetheless)". Naked Objects, on the other hand provides the full UI – "the scope of Naked Objects is larger than Metawidget".

But Dan also noted that the Naked Objects framework provides a blunt level of pluggability so that entire viewers can be plugged in to provide different UIs: "the technology used by any given viewer is generally fixed (AWT, Wicket, JSF etc.). But some viewers do provide pluggable layouts in a manner similar to that allowed by Metawidget. Rob's Scimpi Web viewer [a Naked Objects viewer that produces similar results to figure 29 but using a Web-based platform], for example, provides a whole slew of tags that can be assembled onto a page to provide a rendering of an object, collection or action dialog [Scimpi 2010]. The only real assumption that Scimpi makes is what is being rendered is going to be one of these things (an object, collection or action dialog). Even then, Scimpi's tags allow other information to be 'mixed-into' the page (e.g. the name of the currently logged-on user)". Dan then went on to describe his Wicket viewer: "the Wicket viewer likewise looks for a page to render an object,

collection or whatever. Where it differs from Scimpi is really just that its rendering is done not using tags but using Wicket components".

The Scimpi and Wicket viewers therefore provide evidence of convergence. There is still a gap between approaches in that Metawidget places no expectations on the 'outside' of each page, whereas the Naked Objects viewers expect the entire page to map to an instance of something within their metamodel. Also, plugging in new viewer implementations is not something the Naked Objects team expects practitioners to undertake. Dan agreed: “it's a lot of work. Maybe in time we'll mature this somewhat and make it easier by pulling some common building blocks into the core (i.e. closer to how Metawidget works, I imagine), but for now that isn't the case".

7.1.2. Conclusion

Through interviewing the Naked Objects team, I discovered broad agreement on four out of five of the key characteristics. None of these were considered an explicit feature of the original Naked Objects and all had evolved independently within our two projects, so this represented good validation. The interview also established there were areas where our project philosophies differed, and were likely to remain in disagreement.

In a previous article (Kennard & Leaney 2010) I had reasoned that implementation of all five characteristics would lead to an emergent advantage: being able to retrofit an existing application that was not built with UI generation in mind (see 5.2.1.3.1). For example one could retrofit a word processor: the main word processing area would be left untouched, but the numerous dialog boxes for application and formatting preferences could be retrofitted to use UI generation. Metawidget demonstrates this advantage but Naked Objects does not, because there is still a gap around pluggable layouts which limits the categories of applications Naked Objects can be applied to. Dan agreed: “realistically, we aren't ever going to see a word processor written in Naked Objects… but I'm interested in figuring out how many UI screens can be thought of as a rendering of an object, a collection or action dialog. The customisable UI then amounts to allowing the developer to specify which properties/collections/actions of that single object to appear where, and which to be omitted… the short answer is yes, we want Naked Objects to be more applicable. It's about removing objections from folks trying out the framework”. Clearly there is room for future research and healthy competition between the two projects.

In conclusion I considered it significant that a good many core, non-obvious characteristics had been established. These had been agreed upon both by our own industry adoption studies, and independently by the industry field trials of the Naked Objects team.

7.2. Industrial Validation

Over the course of my three Action Research cycles, my adoption studies had encountered increasingly sophisticated applications. This was to be expected because, as Metawidget matured, so did the complexity of applications that could be built using it. By the end of my Release Candidate cycle I was encountering large-scale industry applications deployed to thousands of users. This section presents a case study of one of those applications in detail.

7.2.1. Methodology

The opportunity to conduct a case study on a large-scale industry application presented clear advantages to my thesis. Industry applicability is a key objective (see 1.1). At the same time, case studies can be academically dangerous. In comparison to experiments or even adoption studies, the researcher must relinquish more control over the environment, the chain of events, even the manner of feedback. In light of this, several methodologies have been constructed to help maintain academic rigour and preserve validity of outcomes. Yin (1984) identifies four validities that must hold in order for a case study as a whole to be considered academically valid. To summarise them in reverse order:

- **Reliability** – the likelihood that another researcher conducting a similar case study on a different organisation would yield the same conclusions. For my case study, this was ensured by recording a thorough, open and honest account of how the case study was conducted.

- **External Validity** – the generalising of a case study's findings beyond the immediate study. For my case study, the results can be generalised to similar products and organisations based on the detailed descriptions in sections 7.2.2 and 7.2.3.

- **Internal Validity** – the assurance of the causal relationship between dependent and independent variables, through techniques such as pattern matching, explanation building and time-series analysis. For my case study, this was dependent on the quality

of my observations and reflections in being able to accurately discern cause and effect.

- **Construct Validity –** the conformance of the measurements to the goal of the study. To achieve construct validity it is important that the situations selected for observation are relevant and that the selected measurements indeed measure what the researcher intends.

Of the four, Construct Validity is perhaps the most challenging to formulate. In particular, identifying the goals of the study and selecting their measures requires careful reflection. To assist with this, I utilised a second methodology known as Goals, Questions and Metrics (Basili 1992).

7.2.1.1. Goals, Questions and Metrics (GQM)

Basili (1992) introduced the methodology Goals, Questions, Metrics (GQM) to assist researchers by defining a funnel through which they can progressively refine their ideas. I used this to help select my measures. Basili first advocates identifying the high-level goals (G) of the project, then using these to extract themes and questions (Q), and finally honing those questions into validatable metrics (M).

To apply this to my work, the goal of this thesis is to derive a general purpose architecture for automatic UI generation. A consistent theme has been to define 'general purpose' as being 'acceptance by industry practitioners'. To move to the next step of Basili's technique, I considered the questions and themes arising from this goal. Acceptance is a multi-faceted concern. First, the solution must have an obviousness to it: it must be approachable and straightforward to conceptualise, with a learning curve no steeper than necessary. Second, the solution should be convenient to use: it's API must be powerful but not cumbersome, and be more productive than developing the same application without it. Third, the solution must be adaptable: it must work well within a broad range of architectures, both front-end UI frameworks and back-end technologies. Finally, the solution must be performant: imposing reasonable processor time, bandwidth and memory constraints that do not outweigh its benefits.

To complete Basili's approach, the metrics derived from these themes (obviousness, convenience, adaptability, performance) can be tested either quantitatively or qualitatively. There is appeal in the former, as metrics such as 'number of lines of UI code saved', 'hours

required to update the UI following changes to the domain model' or 'number of API methods necessary to implement a UI' have an impersonal, impartial character to them that conveys a sense of neutrality. However such thinking misses a critical point of my work: its success *is* tied to the personal, to the partial. If Metawidget saves practitioners 25% of their UI code but they find it awkward and laborious to use, it will not achieve practitioner acceptance in significant numbers. If Metawidget can do more with fewer API calls but those calls are obtuse and inflexible, its long-term adoption in a project will be unlikely to survive a handover from one practitioner to the next. If Metawidget can automatically update a UI in seconds, but that UI does not appear the way the designer intended, it will not pass client usability tests.

Rather, a more reliable measure arose from qualitative metrics. Themes such as practitioner thoughts, preferences, and satisfaction. It is possible to give these an impersonal, quantitative flavour using techniques such as Likert scales (1932), but again in doing so I would risk losing a critical essence. Given the fragile, elusive nature of a quality such as 'acceptance', it seemed prudent the case study remain qualitative. It would be similar to the interviews and adoption studies already undertaken, albeit with an important difference. Whilst the feedback from the adoption studies was an integral part of the reflection and planning that matured each phase of the Action Research, at this stage my implementation was largely finalised so the case study concerned itself more with validation. Any suggestions and enhancements that did arise formed part of future work (see 8.3).

Akin to the interviews and adoption studies my metrics consisted of standardised, open ended questions. Following GQM (Basili 1992), and in order to achieve Construct Validity, I derived these metrics directly from my questions. They were:

- **Obviousness:** prior to encountering Metawidget did you have any preconceptions regarding UI generation? If so, how did Metawidget fit with those preconceptions? If not, could you identify with the gap Metawidget defines? As you were getting started with Metawidget, did you find its parts arranged roughly where you expected to find them? Were there any areas that stood out as being designed differently to what you expected? If so, what were they and what were you expecting?

- **Convenience:** having determined what you wanted Metawidget to do, how difficult did you find getting Metawidget to do it? Were there scenarios where Metawidget demonstrated clear benefits over your usual techniques? Were there scenarios where Metawidget was demonstrably worse than your usual techniques, or did not represent a

compelling advantage? If so, what would have helped tip the balance?

- **Adaptability:** how did you find Metawidget initially fit with your existing architecture? Were there parts that 'just worked'? Were there areas where you had to write your own plugins, and if so how did you find writing them? Were there areas where Metawidget couldn't be made to fit?

- **Performance:** how did your application compare, both in terms of speed and memory, before and after the introduction of Metawidget? Did you find the before and after reasonable in terms of the costs and benefits of UI generation?

With these metrics, explicitly derived from my questions and goals, I could secure the final of the four validities (Yin 1984) necessary for an academically sound case study.

7.2.2. Organisation and Product Overview

This case study was with Telefónica.

Telefónica is one of the largest fixed-line and mobile telecommunications companies in the world. It operates globally across Europe and Latin America with headquarters in Madrid, Spain. Telefónica was founded in 1924, and was originally government owned until being privatised in 1997. Since then it has grown to over 260,000 employees with an annual revenue in excess of 60 billion Euros.

The company was looking to develop a product for the Spanish National Health System (NHS). The Spanish NHS is similar to that found in many European countries. It consists of a network of health clinics and hospitals across different states and territories. Each centre employs multiple healthcare workers with an array of specialities including General Practitioners (GP), paediatricians and physiotherapists. They are funded through both public, government healthcare and private healthcare insurers.

The Telefónica Health Portal is designed as an online platform providing a range of services to health clinics. The Health Portal's functionality includes administering a clinic (see figure 32) and scheduling physicians (see figure 33). Most relevant to this case study, the Health Portal serves as an intermediary between clinics and healthcare insurers. Such an intermediary provides three key benefits compared to manual processes. First, it provides interactivity: if

additional documentation or authorisation codes are required during submission of an insurance claim, the insurer can request them at the time the claim is being lodged. Second, it provides immediacy: after the claim is lodged, the Health Portal can report back a status such as approved, rejected or pending validation. Finally, it improves processing time: claims can be lodged and payments made more quickly, and clinics can see real-time reports of payments settled against their account as they approach month end.

This description of the Health Portal, simplified for the purposes of this case study, is depicted visually in figure 34.

Figure 32: Health Portal administration

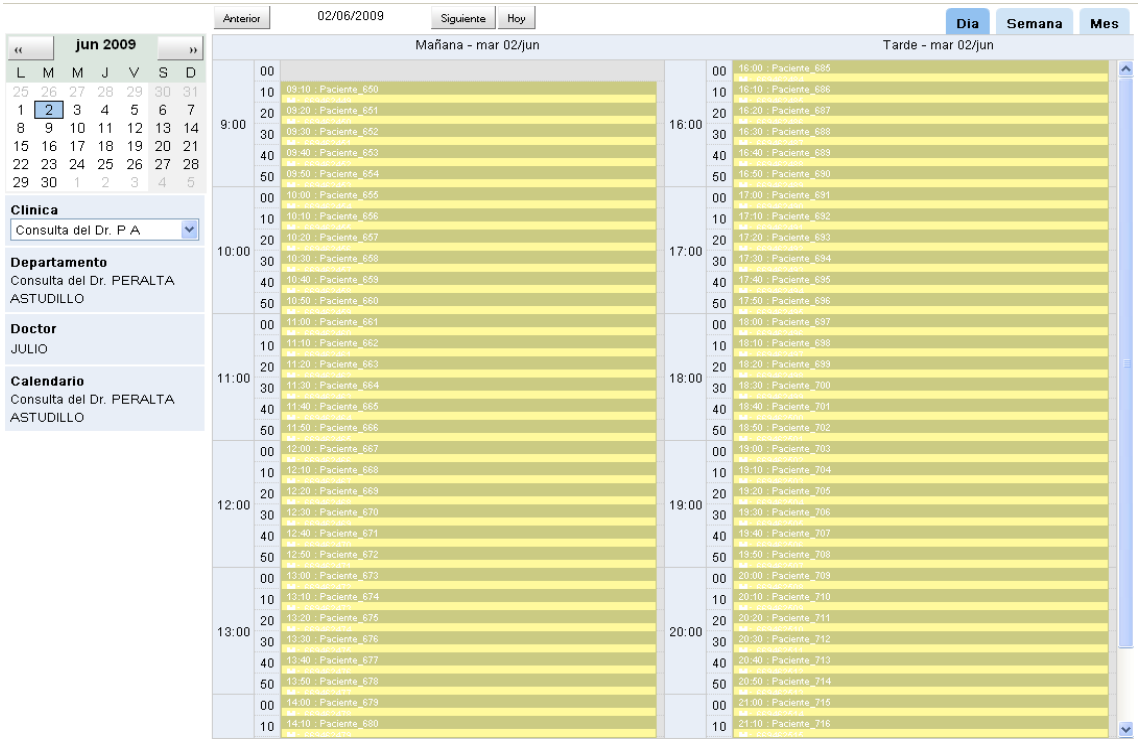


Figure 33: Health Portal scheduler

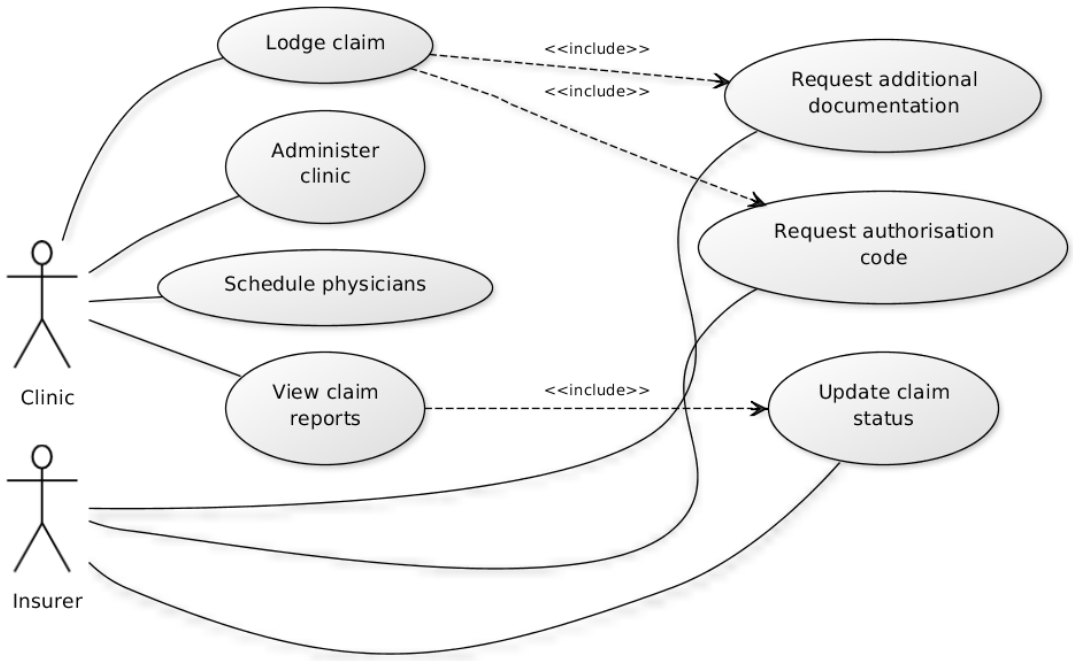


Figure 34: Simplified UML diagram of Health Portal

However, business analysis showed that the data needed in order to lodge a claim varied for each private healthcare insurer. Within each insurer, it further varied by speciality (GP, paediatrician, etc.). And within each speciality, it further varied by type of activity (initial consultation, follow-up visit etc.). The Health Portal would need one insurance claim form per insurer, per speciality and per activity. Worse, as new insurance companies signed on to the service, new forms would need to be developed. This ongoing development cost would threaten the economic viability of the Health Portal. Instead, Telefónica decided they needed a way to dynamically define portions of each insurance screen. Indeed, they wanted the insurer to be able to dynamically define their forms themselves. This was where Metawidget came in.

7.2.3. Integration of Metawidget

This case study interviewed members of the project team, including the project manager.

The discussion opened around the Health Portal's requirement to dynamically define portions of each insurance form. The project manager explained: “We had a need to dynamically create input data screens, we searched the different alternatives available in the market, and the one that fitted best was Metawidget”. He explained they considered several alternatives but “after an exhaustive analysis of available tools we decided that the tool that best fitted our needs was Metawidget”.

The Health Portal needed to provide a range of functionality. This required a rich UI with several different types of screens and aesthetics. I observed there was no requirement to automatically generate the entire UI. Indeed for many screens doing so would have been impractical. For example figures 32 and 33 show screens that are manually tuned for usability. It would not have suited the project to use a language-based UI generator (see 2.1.1.3) or impose a generic, stylised OOUI across every screen. The team only wanted to use automatic generation for selected portions of their application. In addition, they had already chosen their preferred UI framework and tools (GWT 2011) and developed several screens using traditional techniques. It would not have suited them if the UI generator had tried to dictate their technology choices. Together, these observations validated my approach to defining useful bounds (see 4.1.1.4).

The team wanted the dynamic portions of their insurance claim forms to be definable by the insurer. They built a UI to allow the insurer to specify their particular fields, including the name, data type and other metadata (such as whether they were optional fields). The team then needed these fields to be reflected on the clinic's screens. The application was built around a rich, Web-

based UI making extensive use of JavaScript and client-side AJAX calls to Web services. The design was that, upon initiating an insurance claim, the UI would first invoke a Web service and supply the id of the insurer. The Web service would respond with an XML definition of the insurer's form requirements, including portions that described the dynamic fields. A typical response would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<mensaje co_op="R00210">
   <R00210>
      <cif-aseguradora>00000000X</cif-aseguradora>
      <co-facturador>00000000X</co-facturador>
      <respuesta></respuesta>
      <timestamp>0000000000000000</timestamp>
      <agrupaciones>
         <agrupacion codigo="0001">
            <nombre>AGRUPACION</nombre>
            <especialidades>
               <especialidad codigo="01">
                  <nombre>MEDICINA GENERAL</nombre>
                  <actos>
                     <acto codigo="0001">
                        <nombre>CONSULTA</nombre>
                        <campos-variables>
                           ...GP initial consult dynamic fields...
                        </campos-variables>
                     </acto>
                     <acto codigo="0002">
                        <nombre>REVISION</nombre>
                        <campos-variables>
                           ...GP follow-up visit dynamic fields...
                        </campos-variables>
                     </acto>
                  </actos>
               </especialidad>
               <especialidad codigo="02">
                  <nombre>PEDIATRIA</nombre>
                  <actos>
                     <acto codigo="0001">
                        <nombre>CONSULTA</nombre>
                        <campos-variables>
                           ...pediatrician initial consult dynamic fields...
                        </campos-variables>
```

```
                        </acto>
                        <acto codigo="0002">
                           <nombre>REVISION</nombre>
                           <campos-variables>
                              ...pediatrician follow-up visit dynamic fields...
                           </campos-variables>
                        </acto>
                     </actos>
                  </especialidad>
                  ...more especialidad...
               </especialidades>
            </agrupacion>
            ...more agrupacion...
         </agrupaciones>
      </R00210>
   </mensaje>
```

The UI generator would extract those portions of the XML response related to dynamic fields and use them to generate its UI. This requirement validated my earlier decision to perform software mining at runtime (see 4.1.1.3). It is an example of the scenario covered in section 2.2.2, whereby a system's input is itself source code – adding new functionality and screens to an application. Runtime analysis is needed to accommodate such a scenario.

Because the fields were to be defined declaratively, interactive graphical specification tools (see 2.1.1.1) were not applicable. And because the screens must be generated dynamically at runtime, rather than statically at development time, model-based tools (see 2.1.1.2) were not suitable either. What was needed was a runtime generator that could source its metadata from arbitrary sources, in this case embedded in an XML response from a Web service. As the project manager commented: “The main feature [of Metawidget] for us was the possibility to dynamically, based on rules stored in our database [and exposed via a Web service], create input screens based on user selections”. The team were able to plug in a custom inspector to suit their needs. But they did not require *multiple* inspectors, as the Web service provided a single source of metadata, so they did not require collation. This validated my decision to make collation pluggable (see 4.2.1.3).

Once the UI had been generated and the data captured, it was to be written back into the *same* XML structure and returned via a second Web service. This was an interesting design decision. Its rationale was that there would then be a single piece of XML containing both field names,

data types and values. This XML could be stored directly in the database. Screens using it could then be recreated and redisplayed at a later time, even if the insurer's original XML definition changed. For example if, having used the Health Portal for a few months, the insurer decided they needed to alter the fields on their form, the previous several months worth of claims and associated invoices could still be rendered in their original format. This was an unusual requirement because it meant the UI data was not to be stored back to a domain object. Indeed, there was no domain object to store back to. Rather, data values had to be read and written into a fragment of XML. The team were able to plug in a custom widget processor to achieve this, validating my decision to make binding pluggable (see 6.1.1.1).

Finally, the presentation of the dynamic portions was required to be different for different screens, so as to blend with the non-dynamic portions. For the 'lodge individual claim' and 'lodge multiple claims' screens a three column layout was required, as shown in figures 35 and 36 (the area generated by Metawidget is highlighted in red). For the invoice screen a single column layout was preferred, as shown in figure 37 (again, the area generated by Metawidget is highlighted in red). These differences in layout validated my decision to make layouts pluggable (see 4.1.1.5).

Having observed the organisation and product, and understood Metawidget's integration within it, the next step was to validate Metawidget's effectiveness.

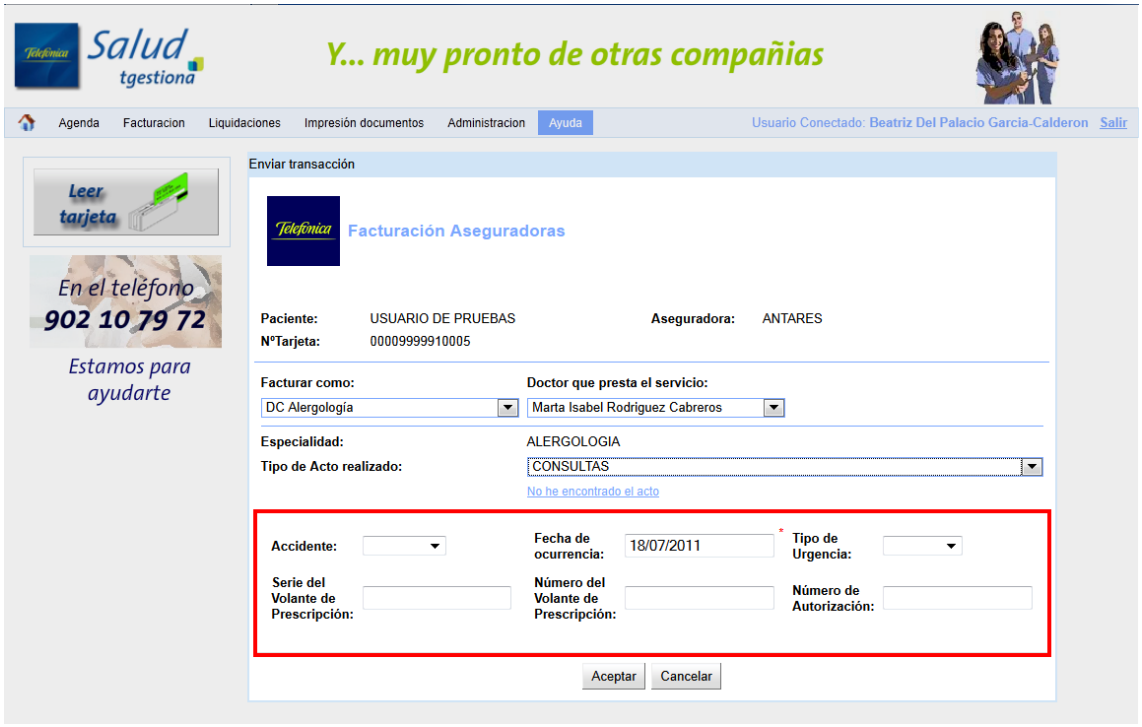


Figure 35: Metawidget is used while lodging individual claims

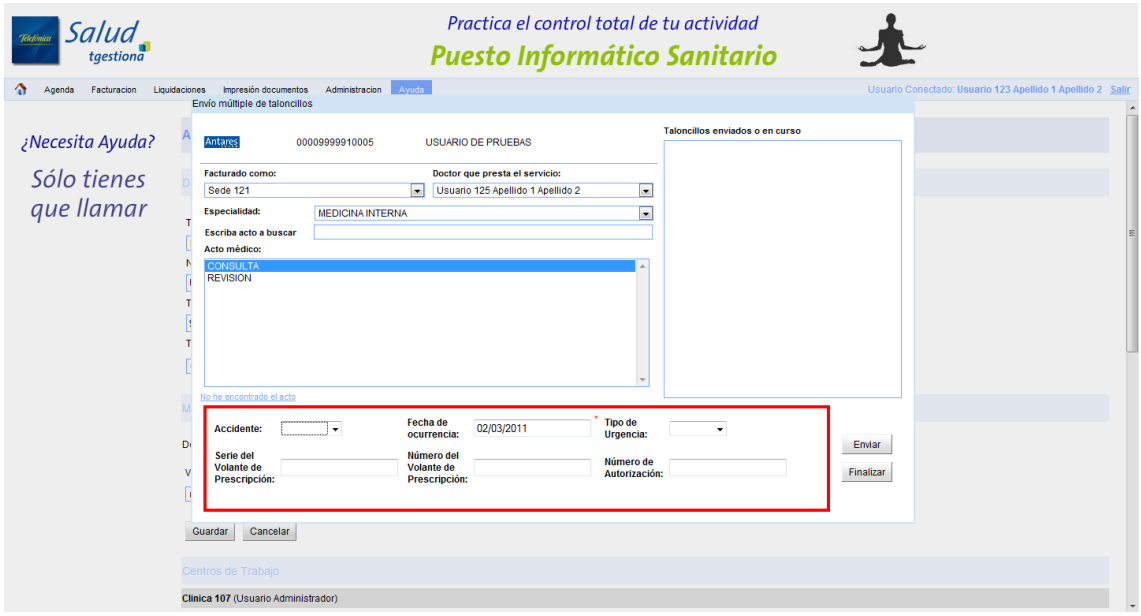


Figure 36: Metawidget is used while lodging multiple claims

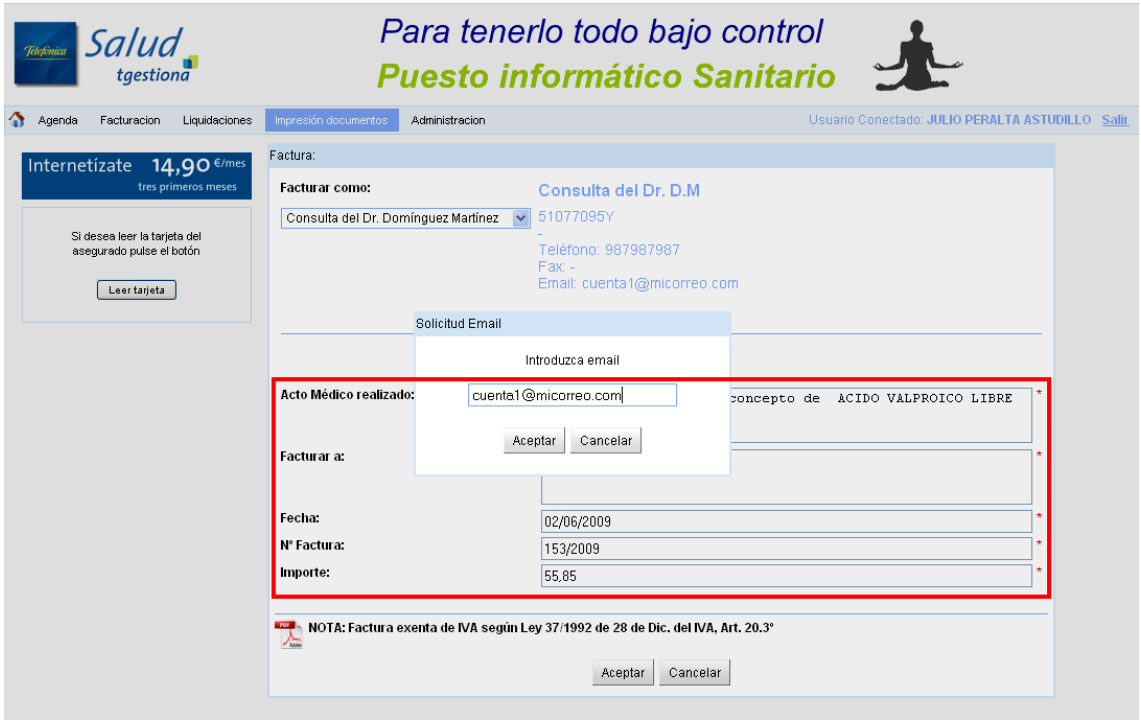


Figure 37: Metawidget is used while printing invoices

7.2.4. Validation of Metawidget

We turn now to discuss Metawidget in the context of my four GQM metrics (see 7.2.1.1).

7.2.4.1. Obviousness

Prior to encountering Metawidget did you have any preconceptions regarding UI generation? If so, how did Metawidget fit with those preconceptions? If not, could you identify with the gap Metawidget defines? One team member recalled: "In our case [it was] more than preconceptions. We had actual requirements. Requirements in concrete cases for generating UI, i.e. we needed a technology compatible with GWT, it had to work with XML, and it also had to be able to work dynamically". The team already had a product specification whose requirements included UI generation, so they were very clear about the gap they needed to fill.

The team member elaborated: "The Health Portal acts like a broker between insurance companies and clinics/hospitals. When data flows between those two parts (e.g. a clinic sends a bill to the insurance company), certain parts are common to all the insurance companies (such as the structure of worker's 'profiles' and of the medical 'acts') and others are not (bill numbering can be different, some include authorisation number etc.). We wanted the insurance companies

to be able to define and provide (through XML) themselves these variable data for the benefit of both parties". The team understood this was not a requirement they could fulfil using their existing technologies. It was an explicit requirement for UI generation. This is unusual. It is distinct from a team who, say, were already using manual techniques to construct their UIs and were looking for a way to automate their processes (see adoption studies at 4.3.1.5 and 5.3.1.1).

As you were getting started with Metawidget, did you find its parts arranged roughly where you expected to find them? Were there any areas that stood out as being designed differently to you expected? If so, what were they and what were you expecting? The team member reflected: "we really didn't have so much expectation about that". Nevertheless, they were comfortable with what they found. The project manager confirmed: "The [Metawidget] concept makes sense, and it gives opportunities to create very flexible applications, where the input screens are easy to adapt to the user needs". Such input screens can be seen in figures 35, 36 and 37, described previously.

7.2.4.2. Convenience

Having determined what you wanted Metawidget to do, how difficult did you find getting Metawidget to do it? One team member responded: "Let's say the difficulty was medium. There were some features we wanted but which Metawidget did not have at that time, and that did require some customisation of the code". For these, the team were able to plug in their own inspectors and widget processors.

Were there scenarios where Metawidget demonstrated clear benefits over your usual techniques? The team member validated: "More than clear benefits. With our requirements Metawidget was basically the only option. Our usual techniques would not have done the job. The only other solution that came close to meet our requirements was TICBO [a Customer Relationship Management tool] but in end it did not meet all of them". Metawidget met all requirements because "it was compatible with GWT, could work with XML, and could work dynamically". This was a validation of Metawidget's mixture of useful bounds (see 4.1.1.4) and runtime software mining (see 4.1.1.3). They created a solution unlike any other available.

Were there scenarios where Metawidget was demonstrably worse than your usual techniques, or did not represent a compelling advantage? If so, what would have helped tip the balance? The project manager replied there were no demonstrably worse scenarios, but that "we think it

would be nice to have conditional fields, so that [a field's] behaviour would depend on the user selections from other fields". This sentiment echoed section 4.3.1.3.2 from the alpha cycle. Following that reflection, I had actually done much work on implementing conditional fields for different environments. For those environments that already had an expression language, I had leveraged it to bind fields together in conditional ways. For those environments that lacked their own expression language, I had added pluggable support for third-party expression languages. However I had not catered for the particular combination of front-end and back-end this project chose. Specifically, Metawidget had no solution for client-side, browser-based (i.e. ECMAScript) conditional fields using GWT. More work was needed there, though there was good precedent for incorporating this kind of technology based on the other platforms.

7.2.4.3. Adaptability

How did you find Metawidget initially fit with your existing architecture? Were there parts that 'just worked'? The team member responded: "As [I said] before, the use of Metawidget was somewhat concrete and, where we used it, it did meet our requirements and worked. [On top of that] the code was customised to include features not yet present at the time". There were two examples of such customisation. The first was a custom inspector, `CamposVariablesInspector`. This was used to inspect fragments of the XML response returned by the insurer Web service, as shown in figure 38. This was different to Metawidget's standard inspectors, which generally inspected objects or whole XML configuration files.

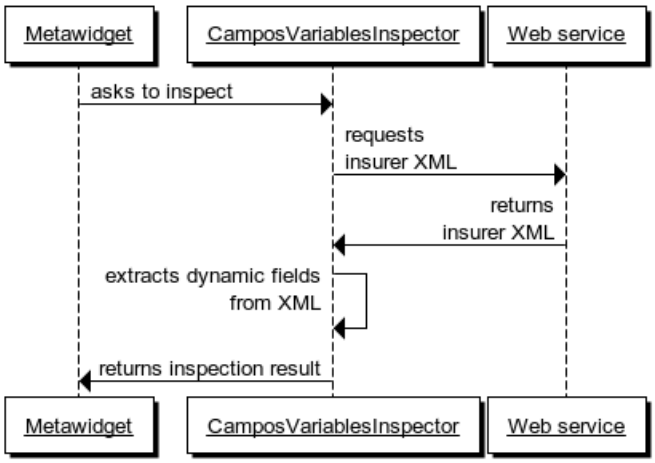


Figure 38: Health Portal uses a custom inspector

The second piece of customisation was a custom widget processor, `CamposVariablesBinding`. This both extracted data values from the XML fragment and

wrote them into the generated widgets, and also read them back from the generated widget and inserted them into the XML fragment, as shown in figure 39. Again this was different to Metawidget's standard binding, which bound data values to domain objects.

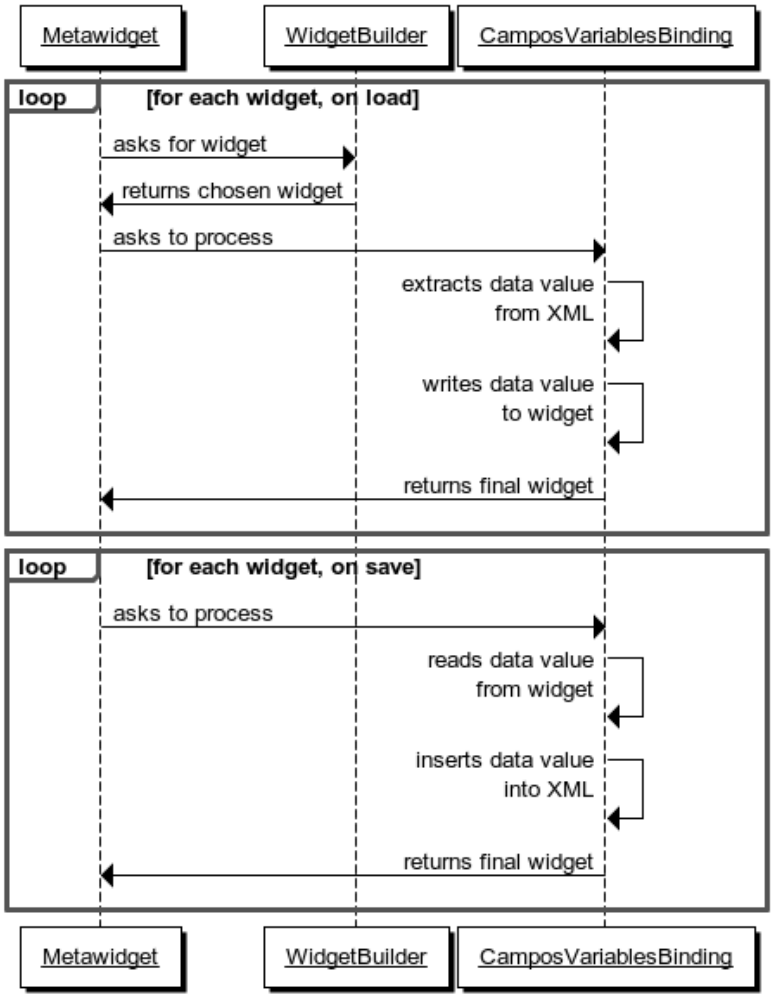


Figure 39: Health Portal uses a custom widget processor

Were there areas where you had to write your own plugins, and if so how did you find writing them? The project manager explained: “Being able to incorporate Metawidget within an existing UI was important. It's fundamental for our project”. Similarly to integrate with their existing back-end: “It was important it supported our back-end. Being able to plug-in our back-end inspectors gave us the flexibility needed, it is impossible for Metawidget to support everyone's requirements”. I observed the inverse of this statement is that it is unrealistic to expect everyone to change their application to suit Metawidget's requirements. This ability to integrate was so important, in fact, that the project manager summarised “It was critical Metawidget supported both our front-end and back-end, otherwise we probably would not have

even tried it [for the Health Portal]". This provided further validation of my 'do not dictate the architecture' tenet.

Were there areas where Metawidget couldn't be made to fit? The team member couldn't recall any: "No, where we used Metawidget it did fit".

7.2.4.4. Performance

How did your application compare, both in terms of speed and memory, before and after the introduction of Metawidget? The team member replied: "In this case there was no before and after. No alternatives to Metawidget were ever developed, it was included from the beginning". However the team had encountered no performance problems, having deployed the Health Portal to thousands of clinics across Spain.

Did you find the before and after reasonable in terms of the costs and benefits of UI generation? The team member opined: "As [I said] before, there were no alternatives developed. However, we consider the choice reasonable in terms of cost and benefits; it was really the only option that met our requirements. If not, these requirements would have had to be changed. That would have meant less flexibility to all the parties of the project. Of course, another option would have been to develop some in-house solution similar to what Metawidget does, but that was never really an option considering the costs and benefits". This validates a sentiment from one of my adoption studies (see 4.3.1.5.2). Namely, that UI generation is conceptually a common problem that calls for a general purpose solution rather than an in-house one. "[It] did not add any business specific value if we could find a third-party solution that solved the same problem"

7.2.5. Conclusion

Through this case study I gathered responses to my GQM metrics (Basili 1992) of obviousness, convenience, adaptability and performance. For obviousness, the team reflected they had "more than preconceptions. We had actual requirements.... the [Metawidget] concept makes sense". For convenience, they confirmed Metawidget had "more than clear benefits. With our requirements Metawidget was basically the only option". However there was a learning curve: "let's say the difficulty was medium". For adaptability they recounted "being able to incorporate Metawidget within an existing UI was important... being able to plug-in our back-end inspectors

gave us the flexibility needed". Finally for performance, the team had encountered no performance problems.

These metrics in turn provided an understanding of my GQM questions and validated my GQM goal of 'practitioner acceptance'. The case study found industry practitioners who had accepted, and successfully adopted, Metawidget for use in their application. At the time of writing, the Telefónica Health Portal has been in production for several months and deployed to some 3,000 health clinics across Spain. This presents strong validation of industry applicability and practitioner acceptance, though not the final word. Further validation will be presented in section 8.1.2.

In closing, I asked the project manager how he would sum up the team's experiences with Metawidget? "Since we use [Metawidget] as a dynamic information capture tool, it gives us great flexibility towards fulfilling customer requirements in record time. Even more of the information captured is almost as a black box where our application does not apply any business rules, [letting] our customers [the insurers] be the ones that define the business rules. Our application is a bridge between the user and our customer, and from that point of view Metawidget fits our needs perfectly, since it allows us to offer the customer [insurer] with a tool for him to decide and customise, without our help, the information that needs to be captured from the user [clinic]".

This chapter has validated my thesis from the perspectives of both a research team and an industry team. It found them broadly in favour of Metawidget's approach, both in terms of architectural philosophy and industry applicability. Together these findings present a fitting conclusion to my Action Research cycles. For a conclusion to my thesis as a whole, we turn now to the final chapter.

8. Conclusion

The objective of this thesis was to derive a general purpose architecture for automatic UI generation. It did this using a combination of modern research techniques and emerging technologies. By using Action Research and observing and reflecting upon industry practitioners, it distilled five fundamental characteristics. The implementation of the first of these characteristics (inspectors) involved the emerging technology of software mining. This chapter presents a final conclusion to my thesis, including strengths and challenges, and outlines future work.

8.1. Strengths

8.1.1. Contributions to Field

This thesis has made a number of novel contributions to the field.

First, it has derived five characteristics that a general purpose UI generator should embody in order to be applicable to industry. It has identified inspectors, inspection result processors, widget builders, widget processors and layouts as required functionality (see 6.1.1.4). It has supported this conclusion with evidence from industry experiments (see 4.2.1.5 and 5.2.1.3), adoption studies (see 4.3.1.5 and 5.3.1.1) and a case study (see 7.2). Also by feedback from forums, blogs and industry white papers. It has produced a well-received journal article defining these characteristics. As one reviewer expressed: “I really like this article… the authors are identifying well the need of their research and are presenting a structured approach to tackle the different challenges that the research on UI generation is facing.”

This thesis has further sought to validate its five characteristics by looking for convergence with another research team (see 7.1). It has successfully reached consensus on four out of five characteristics. Such convergence is a significant step towards industry standardisation. As Ed Burns, industry project lead for the Java EE UI specification, put it: “Thank you for sharing this [paper on convergence]. I have long held that the reason for the incessant churn in how we best complete the UI architecture task is due to the continued quest for economic value. This paper shows there are benefits to separating the elements of that task from that churn".

Second, this thesis has applied the emerging technology of software mining to the established field of UI generation. In doing so, it has shown it possible to resolve long-standing problems in the discipline. Specifically, software mining can remove the repetition of interactive graphical specification and model-based tools. This repetition is both laborious and a source of errors (Jelinek & Slavik 2004) and a barrier to industry adoption (see 4.3.1.1.1).

Third, this thesis has viewed UI generation as a way to augment, rather than replace, traditional UI development practices. It has pursued retrofitting as a novel goal (see 5.2.1.3.1), explored mechanisms to integrate existing industry practices into automatic generation (see 5.1.1.2, 6.1.1.1 and 6.1.1.2), and resolved long-standing problems with generalised heuristics in language-based tools (see 2.1.1.3). To embody its viewpoint, this thesis has coined a new term: the OIM (see 4.1.1.4).

Finally, this thesis is a worked example of the effectiveness of Action Research as a methodology to address practice-based problems. Specifically, it has demonstrated how successive cycles of Action Research can be used to observe and derive solutions in consultation with practitioners. Over half of the Metawidget pipeline (see 6.1.1.4) arose as a direct result of Action Research reflections. By staying in close contact with industry throughout, this thesis ensured its final result was well received (see 7.2).

8.1.2. Industry Adoption

The objective throughout this thesis has been a general purpose architecture, applicable to and adopted by industry (see section 1.1). It has successfully achieved this goal. This is evidenced by my industry experiments (see sections 4.2.1.5 and 5.2.1.3.1), adoption studies (see sections 4.3.1.5, 5.3.1.1) and case study (7.2). Also by feedback from forums, blogs and industry white papers.

Further validation is evidenced by Red Hat's decision to integrate Metawidget into their next generation products. Red Hat is a leading industry provider of enterprise middleware. Their products include Red Hat Enterprise Linux and JBoss Application Server. They are consistently ranked one of the leading industry vendors for providing software to the enterprise. I asked Dan Allen, Principal Software Engineer at Red Hat, to summarise their decision to adopt Metawidget. Dan explained:

“At Red Hat, we look for practical solutions to distil complex enterprise behaviour into expressive programming models. In other words, we look for the boilerplate, low-level tasks where bugs are easily introduced and provide higher level controls in their place that allow programmers to describe the behaviour they need without concerning themselves with how the work is performed. The objective is for the developer to work quickly and safely, and to remain true to the golden rule of programming: Don't Repeat Yourself (DRY).

But we do more than create abstraction layers. We believe that success in the enterprise depends on defining these industry standards transparently and developing Open Source implementations for them. This approach results in higher quality technologies that ensure compatibility between implementations and interoperability with other tools and systems, requirements which our customers continually stress. This process also helps foster an ecosystem of technologies that can build on these programming models in ways that were not originally envisioned or explored, often serving as a source of future standards.

One of the areas within the standard Java enterprise development stack where the golden rule (DRY) is often broken is the UI. Not a day goes by that a developer, somewhere, isn't putting in overtime hand editing form elements page after page. The job is to make these form elements consistent with changes to the model, or the requirements. Software teams spend a lot of time developing the UI, partly because the UI is such a critical part of the application. Any time saved in this area can have a major impact on the efficiency of the development process, either to reduce costs or make time to polish the user experience. Either way, it's better for the business.

In searching for a solution [for our next generation product], we aimed to bridge the gap between server-side and client-side technologies in a way that would leverage the intelligence resident in the standards-based programming model. In particular, we wanted a project that could integrate well with the Java EE technologies and, in addition, Red Hat's companion extensions. Combinations included JSF and RichFaces [UI component libraries], JPA and Hibernate [persistence frameworks] and Bean Validation and Hibernate Validator [validation frameworks]. The UI, particularly the forms, should align with the metadata the server-side technologies define, without intervention from the programmer.

> Furthermore, the alignment should not be limited to only these standard sources.
>
> We found Metawidget uniquely qualified. It gathers all the information it can from the existing architecture, whether it be standards-based or otherwise, and uses that information to generate consistent interface controls. With each technology introduced into the application, Metawidget's intelligence only grows stronger, thanks to its plugin architecture. Metawidget allows those developers putting in extra hours to go home early because changes to the model, or any metadata source, is immediately reflected in the UI. The result is less tedious labour and less opportunity for human error.
>
> Personally, I was intrigued by the idea of plugging in less orthodox technologies to generate UIs from novel sources, such as generating a UI from a rule engine. We have a lot of customers using Drools [Red Hat's rule engine], and it would be great to open new possibilities for them. This is where Red Hat's focus on interoperability, as well as choice, plays such an important role. Metawidget both benefits from this competitive advantage and helps to propagate it.
>
> In summary, we found Metawidget to be an excellent solution to keep UIs DRY. It's clear that a lot of work has gone into understanding different enterprise architectures, both standard and non-standard, to ensure Metawidget integrates well with them. The result is a project we think will be very relevant and applicable to our enterprise customers in the future."

Dan's comments, together with evidence from adoption studies, case studies, forums and blogs, are testament to the level of industry adoption my research has achieved. Combined with its contributions to the field of UI generation (see 8.1.1), this demonstrates my thesis has been broadly successful. However, some challenges remain. We turn to these in the next section.

8.2. Challenges

8.2.1. Lack of Standardisation

Industry consensus takes time. At the time of writing, there is insufficient consensus to justify proposing a standard in the OIM space. Standardisation requires broad agreement and, ideally,

multiple mature and competing implementations. The teams behind such implementations can then come together, identify their commonalities, and work to generify them. This can be evidenced by such projects as Hibernate (2008) and TopLink (2008) standardising the ORM space under JPA (DeMichiel & Keith 2006).

It is premature to pursue standardisation before such broad consensus has been achieved. The OIM space is unfortunately not there yet. However with corporations such as Red Hat promoting Metawidget, and the Apache Software Foundation promoting Naked Objects (see 7.1), I believe it is only a matter of time.

8.2.2. Unbalanced User Documentation

A recent piece of feedback drove home Bucciarelli's (1994) treatise on object worlds. Specifically because I, as a practitioner, have been developing Metawidget from a cross platform, technology neutral perspective much of its user documentation is similarly cross platform and technology neutral.

For example the User Guide contains paragraphs such as "the [buttons in this tutorial] are either manually specified in the JSP page (for GWT, Spring and Struts) or created by `UIMetawidget` based on annotated methods in the `ContactBean` (for JSF)". The text regularly skips between, and contrasts, industry technologies such as GWT (2011), Spring MVC (2011), Struts (2011) and JSF (2011). This is natural from a Metawidget perspective but quite jarring for a practitioner whose object world is typically focussed on a single technology stack. It is clearly unintuitive. However it is also unclear how one could write separate documentation for each potential combination of technology stacks. It will be a challenge to rework the User Guide to be more practitioner focussed without becoming too voluminous.

8.3. Future Work

This thesis leaves open a number of areas for future work, both at an industry and a research level. These are outlined in the sections to follow.

8.3.1. Tooling

A key driver for industry adoption is tooling support. At the time of writing, no industry IDEs

have explicit support for Metawidget, though Metawidget does integrate with them by adhering to existing standards. In the future, however, richer tooling will likely be very important. Many industry practitioners expect high quality tool implementations for the technologies they use, and an OIM should be no different.

Metawidget has been designed, from the outset, to be a well-behaved member of the development stack. To be 'just another tool' in a practitioner's arsenal. The type of tooling that would benefit Metawidget would be: support for editing `metawidget.xml` configuration files (see 4.2.2.2.1) including namespace schemas for each pipeline component; support for Metawidget-specific annotations (see 6.1.1.2) and tags in autocomplete; support for Metawidget-specific widgets in component palettes.

Future work will involve liaising with IDE vendors to encourage and assist them in adding Metawidget support. Given the Red Hat adoption, the most likely first candidate will be JBoss Tools.

8.3.2. Packaging

Authentic practice is the best vehicle for Action Research. To this end, over the course of this thesis I have added Metawidget integrations for over two dozen industry technologies and frameworks. Implementing these proved instructive, exposing either new constraints or requiring more flexible architectures. One emergent requirement that is still outstanding concerns more flexible packaging for Metawidget.

An early design decision, centred on ease-of-adoption, was to package Metawidget in a single, easy-to-deploy library. This library contained all integrations for all technologies. In early releases it ran to approximately 200KB. But as further integrations were added it more than doubled in size. When considering deployments for mobile environments, 450KB for a UI widget seems onerous. Worse, the trend can only continue as Metawidget matures and further integrations are added.

Pausing to reflect, it is apparent that little of this 450KB is actually required for a mobile deployment. Many of its integrations are for desktop or Web technologies and could be stripped away. What is needed is a more flexible packaging architecture so that practitioners can include only those technologies required for their environment. This is complicated by the fact that

many technologies are interdependent or overlap. For example both the SwingX (a Swing widget library) and the BeansBinding (a Swing data binding library) integrations require the Swing Metawidget (see section 4.2.2.2), which in turn requires the core pipeline implementation (see section 6.1.1.4). Ideally, specifying a dependency on SwingX would automatically 'drag in' the Swing Metawidget and core pipeline implementations. But specifying a further dependency on BeansBinding should *reuse* the dragged in Swing Metawidget and core, without any conflict. Amongst this complexity, however, I wouldn't want to lose sight of my single, easy-to-deploy library design as I believe that is central to adoption.

There are a number of mature packaging frameworks available within industry. Technologies such as the Apache Maven dependency model (Maven 2011) and OSGi (2011) seem promising approaches. It remains to evaluate these and select which, if any, are suitable. It would then be necessary to refactor the existing codebase. With many integrations already implemented, such a refactoring would not be trivial. However it would be justified if it could halt the inevitable growth of the file size before it becomes untenable for industry deployments.

8.3.3. Metadata Validation

Feedback from adoption studies (see 4.3.1.5.3 and 5.3.1.1.1) suggested Metawidget could better assist those practitioners who prefer to keep UI metadata in a separate file. This metadata necessarily involves repeated values. Metawidget could validate those values against an application's object model to ensure they are consistent. Many ORM solutions such as Hibernate (2008) do this for their database metadata. In addition, any inheritance hierarchies in the metadata could be inferred based on the object model.

The issue is complicated because Metawidget's inspectors are designed to operate in isolation. For example its `XmlInspector` parses XML files but does not concern itself with any classes or objects. That is the responsibility of the `ObjectInspector`. In order for `XmlInspector` to validate against the domain objects this design principle may need to be relaxed so that a single inspector can assume multiple responsibilities.

It will require further reflection to determine the best way to incorporate this.

8.3.4. Release Train

Metawidget brings together many heterogeneous technologies to drive UI generation. Independently, each of these underlying technologies progresses at its own rate with its own release cycle, adding features and resolving defects. This presents a challenge, because for any given version of Metawidget (say, Metawidget 1.0) there will be a host of versions of the underlying technologies (say, GWT 2.2, BeansBinding 1.2.1). This makes it difficult for a practitioner looking for an upgrade to a recent version of a technology (say, GWT 2.3) because they must wait for a complete new cycle of Metawidget (say, Metawidget 1.1). That cycle could be several months away.

It is difficult to know how best to resolve this. One approach would be to maintain separate life cycles for each of Metawidget's plugins. That way, they could progress in step with the underlying technology they represent. But this may become confusing because Metawidget has dozens of plugins. The current approach of having a single, monolithic release every few months is much easier to understand. But it can mean practitioners are left waiting.

An alternative approach would be for the owners of each underlying technology to maintain their Metawidget plugin themselves. This is not unprecedented, for example most database vendors maintain their own Java Database Connectivity (Andersen 2006) drivers. But it requires broad scale adoption of a technology before it makes economic sense for the vendor to invest their resources. Metawidget is not there yet. I hope one day it will be.

8.3.5. Future Research

This thesis has recognised a number of boundaries to its investigations. These provide promising starting points for future research.

First, this thesis has combined two hitherto unrelated fields: software mining and UI generation. This union opens new possibilities for generating different interfaces than those explored in this thesis, which were exclusively *Graphical* UIs (GUI). For example, it would seem possible to generate a command line interface for an application based on mining its defined actions (see section 3.1). Equally it would seem possible to generate a machine interface for an application, suitable for XML web services (see section 5.3.1.1.2.2).

Second, there are further depths to the union of software mining and UI generation. None of the five characteristics I derived through my Action Research (inspectors, inspection result processors, widget builders, widget processors and layouts – see section 6.1.1.4) depend explicitly on either runtime or static code generation. As discussed in section 4.1.1.3, this thesis adopted a runtime approach. But it would be interesting to explore applying the five characteristics statically instead.

Finally, this thesis serves as a successful example of the effectiveness of Action Research as a methodology to address practice-based problems. It could be valuable to combine this with other successful examples of Action Research in order to extract themes and lessons for the future.

8.4. Closing Remarks

This thesis has been about scratching an itch.

As an industry practitioner myself, I had struggled for many years with the issue of duplication in software architectures. I was inspired by evolutions in areas such as Object to Relational Database Mapping (ORM) and Object to XML Mapping. I was frustrated similar research did not appear to be progressing in the UI space. In the industry applications I worked on, the amount of time and code devoted to UI mapping was at least as great as that devoted to database and XML mapping. Yet the UI space was not receiving the same level of attention as the others. The ORM space, in particular, had been a rich source of debate between those exploring lightweight versus heavyweight solutions, those exploring performance trade-offs, those exploring ease of use versus flexibility. From this melting pot standards such as JPA (DeMichiel & Keith 2006) had emerged, to the benefit of all practitioners. I longed for something similar for UIs.

My hope for this thesis was that by involving industry early and often, by harnessing emerging technologies, and by injecting novel ideas, I could eliminate barriers to adoption and elevate the field to trigger such a debate. I believe that, at the close of this thesis, we are at the beginning of this process. With growing consensus from research teams such as the Naked Objects project, and industry support from companies like Red Hat, I believe OIMs will become more mature and plentiful. I welcome competing implementations and differing viewpoints, because my ultimate goal is not that my particular research project becomes mainstream. Rather, it is that the

field becomes mainstream: that OIMs become an accepted, everyday part of software development. This would benefit all practitioners. In turn, it would also benefit myself. It would resolve, in my day to day work, the issue of duplication that plagued me all those years ago and inspired me to embark on this journey of research and learning.

Research Publications

Publications by the author

Kennard, R. & Leaney, J. 2011, 'Is There Convergence in the Field of UI Generation?', *Journal of Systems and Software.*

Kennard, R. & Leaney, J. 2010, 'Towards a General Purpose Architecture for UI Generation', *Journal of Systems and Software.*

Kennard, R., Edmonds, E. & Leaney, J. 2009, 'Separation Anxiety: stresses of developing a modern day Separable User Interface', *2nd International Conference on Human System Interaction.*

Kennard, R. & Steele, R. 2008, 'Application of Software Mining to Automatic User Interface Generation', *7th International Conference on Software Methodologies, Tools and Techniques*.

Publications citing Metawidget

Lienert, C., Jenny, B., Schnabel, O. & Hurni, L. 2012, 'Current Trends in Vector-Based Internet Mapping: A Technical Review', *Online Maps with APIs and WebServices*, pp. 23-36.

Cerny, T. & Song, E. 2011, 'UML-based enhanced rich form generation', *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, pp. 192-199.

Chalupa, V. 2011, 'Nová generace nástroje FormBuilder', Ceské vysoké ucení technické v Praze.

Oh, J. M., Lee, Y. S. & Moon, N. 2011, 'Towards Cultural User Interface Generator Principles', *5th International Conference on Multimedia and Ubiquitous Engineering.*

Stehno, B. 2011, 'Generování aplikací pro OS Android', Ceské vysoké ucení technické v Praze.

Bertoncelj, R. 2010, 'Parsek CMS 2.0', University of Ljubljana.

Schill, A. 2010, 'Generierung von interaktiven OSGi-Komponenten für Android', Technische Universität Dresden.

Aispuro, E.E., Licea, G., Suárez, J., Sandoval, A., Carreño, M.A., Estrada, I., Juárez-Ramírez, R., Aguilar, L. & Martínez, L.G. 2009, 'Supporting the development of interactive applications in introductory programming courses'.

Feldmann, M., Nestler, T., Muthmann, K., Jugel, U., Hübsch, G. & Schill, A. 2009, 'Overview of an end-user enabled model-driven development approach for interactive applications based on annotated services', *4th Workshop on Emerging Web Services Technology.*

O'Hear, T. & Boudjenane, Y. 2009, 'Using Activity Descriptions to Generate User Interfaces for ERP Software', *13th International Conference on Human-Computer Interaction*, p. 586.

Industry Publications

Publications by the author

Allen, D. & Kennard R. 2011, 'DRY UIs: Let the metadata do the work', *JavaOne 2011,* retrieved from http://parleys.com

Kennard, R. 2011, 'Case Study: Telefónica Health Portal', retrieved from http://metawidget.org/media/whitepaper/MetawidgetWhitePaper-TelefonicaHealthPortal.pdf

Kennard, R. 2011, 'Adoption Studies', retrieved from http://metawidget.org/media/whitepaper/MetawidgetWhitePaper-AdoptionStudies.pdf

Kennard, R. 2011, 'What Good is an OIM?', retrieved from http://metawidget.org/media/whitepaper/MetawidgetWhitePaper-WhatGoodIsAnOIM.pdf

Publications citing Metawidget

Baxter, L. 2011, 'JBoss Forge', retrieved from http://jboss.org/forge

Gruber, W. 2009, 'Metawidget: King of the Hill', retrieved from http://it-republik.de/jaxenter/artikel/Metawidget-King-of-the-Hill-2681.html

Wielenga, G. 2008, 'Interview: Creator of Metawidget, the Automatic UI Generator', retrieved from http://java.dzone.com/news/interview-creator-metawidget-a

References

Andersen, L. 2006, http://www.jcp.org/en/jsr/detail?id=221

Apache Commons Validator 2006, http://commons.apache.org/validator

AWT 1998, http://java.sun.com/products/jdk/awt

Bacon, J. 2009, 'The Art of Community: Building the new age of participation', O'Reilly Media, Inc.

Basili, V. 1992, 'Software modeling and measurement: the Goal/Question/Metric paradigm'.

Basin, D., Clavel, M., Egea, M. & Schläpfer, M. 2010, 'Automatic generation of smart, security-aware GUI models', *Engineering Secure Software and Systems*, pp. 201-217.

Bloch, J. 2001, 'Effective Java programming language guide', Sun Microsystems, Inc. Mountain View, CA, USA.

Bloch, J. 2006, 'How to design a good API and why it matters', Association for Computing Machinery, p. 507.

Bodart, F., Hennebert, A.M., Leheureux, J.M., Provot, I., Sacre, B. & Vanderdonckt, J. 1995, 'Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide', *Design, Specification and Verification of Interactive Systems*, pp. 262-278.

Bracha, G. & Cook, W. 1990, 'Mixin-based inheritance', vol. 25, *ACM*, pp. 303-311.

Breu, S., Zimmermann, T. & Lindig, C. 2006, 'Mining eclipse for cross-cutting concerns', Association for Computing Machinery, pp. 94-97.

Brooks, F.P. 1995, 'The mythical man-month', Addison-Wesley Reading.

Bucciarelli, L.L. 1994, 'Designing engineers', MIT press.

Bull, J., Smith, L., Westhead, M., Henty, D. & Davey, R. 2000, 'A benchmark suite for high performance Java', *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 375-388.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L. & Vanderdonckt, J. 2003, 'A unifying reference framework for multi-target user interfaces', *Interacting with Computers*, vol. 15, no. 3, pp. 289-308.

Cardelli, L.1988, 'Building user interfaces by direct manipulation', *Proceedings of the 1st Annual symposium on User Interface Software*, pp. 152-166.

Cerulo, L. 2006, 'On the Use of Process Trails to Understand Software Development', Dipartimento di Ingegneria dottorato di Ricerca in ingegneria dell 'informazione, retrieved from http://rcost.unisannio.it/cerulo/phdthesis.pdf

CodeSmith. 2009, http://codesmithtools.com

Commons Validator 2008, http://commons.apache.org

Constantine, L. 2002, 'The emperor has no clothes: Naked Objects meet the interface', retrieved from http://foruse.com/articles

Crotty, M. 1998, 'The foundations of social research: Meaning and perspective in the research process', Sage.

Cruz, A. & Faria, J. 2010, 'A metamodel-based approach for automatic user interface generation', *Model Driven Engineering Languages and Systems*, pp. 256-270.

CVS 2011, http://nongnu.org/cvs

Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R. & Casati, F. 2007, 'Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities', IEEE Internet Computing, pp. 59-66.

DeMichiel, L. & Keith, M. 2006. 'Java Persistence API', retrieved from http://jcp.org/en/jsr/detail?id=220

Dewey, M. 1965, 'Dewey; Decimal classification and relative index'.

Dick, B. 2000, 'A beginner's guide to action research', retrieved from http://scu.edu.au/schools/gcm/ar/arp/guide.html

Dick, B. 2005, 'Grounded theory: a thumbnail sketch', retrieved from http://scu.edu.au/schools/gcm/ar/arp/grounded.html

Dijkstra, E. W. 1982, 'On the role of scientific thought', *Selected writings on Computing: A Personal Perspective*, pp. 60–66.

ECMAScript 2011, http://ecma-international.org/publications/files/ECMA-ST/ecma-262.pdf

FakeReplace 2010, http://code.google.com/p/fakereplace

Falb, J., Popp, R., Rock, T., Jelinek, H., Arnautovic, E. & Kaindl, H. 2007, 'Fully-automatic generation of user interfaces for multiple devices from a high-level model based on communicative acts', *40th Annual International Conference on System Sciences*, p. 26.

Firesmith, D.G. 1996, 'Use Cases: The Pros and Cons', *Wisdom of the Gurus: A Vision for Object Technology.*

Fowler, M. 2002, 'Patterns of Enterprise Application Architecture', Addison-Wesley.

Fowler, M. & Evans, E. 2005, 'Fluent interfaces', retrieved from http://martinfowler.com/bliki/fluentinterface.html

Gajos, K. & Weld, D.S. 2004, 'SUPPLE: automatically generating user interfaces', *Proceedings of the 9th International Conference on Intelligent User Interfaces*, pp. 93-100.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995, 'Design patterns: elements of reusable object-oriented software', Addison-Wesley.

German, D.M., Cubranic, D. & Storey, M.A.D. 2005, 'A framework for describing and understanding mining tools in software development', *Proceedings of the 2005 international workshop on Mining Software Repositories*, pp. 1-5.

Gosling, J. 2005, 'The Java Language Specification', Addison-Wesley.

Gracelets 2010, http://gracelets.sourceforge.net

Grcar, M., Grobelnik, M. & Mladeni, D. 2007, 'Using text mining and link analysis for

software'.

Griffon 2010, http://griffon.codehaus.org

Groovy 2011, http://groovy.codehaus.org

GWT 2011, http://code.google.com/webtoolkit

Hayes, P.J., Szekely, P.A. & Lerner, R.A. 1985, 'Design alternatives for user interface management sytems based on experience with COUSIN', *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 169-175.

Hayes, J.H. 1998, 'Input Validation Testing: A System Level, Early Lifecycle Technique', George Mason University.

Hejlsberg 2006, http://ecma-international.org/publications/files/ECMA-ST/ecma-334.pdf

Hibernate 2008, http://hibernate.org

Hibernate Validator 2008, http://validator.hibernate.org

HTML 1999, http://w3.org/TR/html401

Hunt, A. & Thomas, D. 1999, 'The Evils of Duplication', *The Pragmatic Programmer.* Addison-Wesley, pp. 26-33.

Javassist. 2009, http://www.jboss.org/javassist

JavaBeans 2008, http://java.sun.com/javase/technologies/desktop/javabeans

JBoss jBPM 2008, http://jboss.com/products/jbpm.

Jelinek, J. & Slavik, P. 2004, 'GUI generation from annotated source code', *Proceedings of the 3rd Annual Conference on Task models and diagrams*, pp. 129-136.

Jha, N.K. 2005, 'Low-power system scheduling, synthesis and displays', *Computers and Digital Techniques,* vol. 152, no. 3, pp. 344-352.

JMatter 2011, http://jmatter.org

JSF 2011, http://www.jcp.org/en/jsr/detail?id=314

JSP 2006, http://jcp.org/en/jsr/summary?id=245

Kagdi, H., Collard, M.L. & Maletic, J.I. 2007, 'Comparing Approaches to Mining Source Code for Call-Usage Patterns', *Proceedings of the 4th International Workshop on Mining Software Repositories*.

Kemmis, S. & McTaggart, R. 1988, 'The action research planner'. Deakin University.

Kim, S., Zimmermann, T., Whitehead Jr, E.J. & Zeller, A. 2007, 'Predicting faults from cached history', IEEE Computer Society, pp. 489-498.

Knuth, D.E. 1974, 'Structured Programming with go to Statements', *Computing Surveys*, vol. 6, no. 4, p. 268.

Knuth, D.E. 1984, 'Literate programming', *The Computer Journal*, vol. 27, no. 2, pp. 97-111.

Krasner, G.E. & Pope, S.T. 1988, 'A description of the model-view-controller user interface paradigm in the smalltalk-80 system', *Journal of Object Oriented Programming,* vol. 1, no. 3, pp. 26-49.

Larman, C. & Basili, V. R. 2003, 'Iterative and Incremental Development: A Brief History', *Computer*, pp. 47-56.

Leung, K., Hui, L.C.K., Yiu, S.M. & Tang, R.W.M. 2000, 'Modeling Web navigation by statechart', *24th Annual International Conference on Computer Software and Applications,* pp. 41-47.

Lévi-Strauss, C. 1966, 'The savage mind', University of Chicago Press.

Likert, R. A. 1932, 'A technique for the measurement of attitudes', *Archives of Psychology*, No. 140.

LinQ 2009, http://msdn.microsoft.com/en-us/vcsharp/aa904594.aspx

Liu, C., Chen, C., Han, J. & Yu, P.S. 2006, 'GPLAG: detection of software plagiarism by program dependence graph analysis', Association of Computing Machinery, pp. 872-881.

Ma, H., Amor, R. & Tempero, E. 2008, 'Indexing the Java API Using Source Code', *19th Australian Conference on Software Engineering,* pp. 451-460.

Maes, P. 1987, 'Concepts and experiments in computational reflection', *Conference on Object Oriented Programming Systems Languages and Applications*, pp. 147-155.

Maven 2011. http://maven.apache.org

Menkhaus, G. & Pree, W. 2002, 'A hybrid approach to adaptive user interface generation', *Proceedings of the 24th International Conference on Information Technology Interfaces*, pp. 185-190.

Microsoft, 2000, 'Inside Out: Microsoft – In Our Own Words', ISBN 0446527394.

Miller, J.S. & Ragsdale, S. 2003, 'The Common Language Infrastructure Annotated Standard', Addison-Wesley Professional.

MSR 2004, http://msrconf.org

Myers, B. 1994, 'Challenges of HCI design and implementation', *Interactions*, vol. 1, no. 1, pp. 73-83.

Myers, B.A. & Rosson, M.B. 1992, 'Survey on user interface programming', Association of Computing Machinery.

Myers, B., Hudson, S.E. & Pausch, R. 2000, 'Past, present, and future of user interface software tools', *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, no. 1, pp. 3-28.

Nagappan, N., Ball, T. & Zeller, A. 2006, 'Mining metrics to predict component failures', Association of Computing Machinery, pp. 452-461.

NetBeans 2011, http://netbeans.org

Neamtiu, I., Foster, J.S. & Hicks, M. 2005, 'Understanding source code evolution using abstract syntax tree matching', Association of Computing Machinery, pp. 1-5.

OSGi 2011, http://www.osgi.org

OpenXava 2011, http://openxava.org

OVal 2011, http://oval.sourceforge.net

Palay, A.J., Hansen, W.J., Kazar, M.L., Sherman, M., Wadlow, M.G., Neuendorffer, T.P., Stern, Z., Bader, M. & Peters, T. 1988, 'The Andrew Toolkit-An Overview', *USENIX Winter Conference*, pp. 9-21.

Pawson, R. 2004, 'Naked Objects', Trinity College, Dublin.

Pawson, R. & Matthews, R. 2002 'Naked Objects', Wiley.

Raggett, D., Le Hors, A. & Jacobs, I. 1999, 'HTML 4.01 Specification', World Wide Web Consortium (W3C)

Raneburger, D. 2010, 'Interactive model driven graphical user interface generation', Association of Computing Machinery, pp. 321-324.

Rigby, P.C. & Hassan, A.E. 2007, 'What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list', IEEE Computer Society.

Rouvellou, I., Degenaro, L., Rasmus, K., Ehnebuske, D. & Mc Kee, B. 1999, 'Externalizing Business Rules from Enterprise Applications: An Experience Report', *Practitioner Reports in the OOPSLA*, vol. 99.

Sahavechaphan, N. & Claypool, K. 2006, 'XSnippet: mining for sample code', Association of Computing Machinery, pp. 413-430.

Schofield, C., Tansey, B., Xing, Z. & Stroulia, E. 2006, 'Digging the Development Dust for Refactorings', *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 23-34.

Schön, D.A. 1983, 'The Reflective Practitioner: How Professionals Think in Action', Basic Books.

Schramm, A., Preußner, A., Heinrich, M. & Vogel, L. 2010, 'Rapid UI Development for Enterprise Applications: Combining Manual and Model-Driven Techniques', *Model Driven Engineering Languages and Systems*, pp. 271-285.

Scimpi 2010, http://nakedobjects.org/plugins/scimpi

Seam 2009, http://seamframework.org.

Shan, T.C., Hua, W.W., Bank, W. & Wilmington, N.C., 2006. 'Taxonomy of Java Web application frameworks', pp. 378-385.

Spring MVC 2011, http://springsource.org

Stevens W., Myers G. & Constantine L. 1974, 'Structured Design'. IBM Systems Journal 13, pp. 115-139.

Struts 2011, http://struts.apache.org

Swing 1998, http://java.sun.com/javase/technologies/desktop

SQL 1987. ISO 9075, Information Processing Systems. Database Language SQL.

SVN 2011, http://subversion.tigris.org

SWT 2008, http://eclipse.org/swt.

Tan, L., Yuan, D., Krishna, G. & Zhou, Y. 2007, 'iComment: Bugs or Bad Comments?', *Operating Systems Review*, vol. 41, no. 6, p. 145.

TIOBE 2011, 'Programming Community Index', retrieved from http://tiobe.com/index.php/content/paperinfo/tpci

TopLink 2008, http://www.oracle.com/technetwork/middleware/toplink

Tran, V., Vanderdonckt, J., Kolp, M. & Faure, D. 2010, 'Generating User Interface for Information Applications from Task, Domain and User models with DB-USE'.

UTS 2008, 'Criteria for Level of Risk in Ethics Applications', UTS Human Research Committee.

Valenzuela, D. & Shrivastava, P. 2002, 'Interview as a Method for Qualitative Research', retrieved from http://www.public.asu.edu/~kroel/www500/Interview%20Fri.pdf

Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D. & Florins, M. 2004, 'USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces', *W3C Workshop on Multimodal Interaction*, pp. 19-20.

VB.NET 2005, http://microsoft.com/download/en/details.aspx?id=22306

Visual Studio 2011, http://microsoft.com/visualstudio

Weiser, M. 1993, 'Hot topics-ubiquitous computing', *Computer*, vol. 26, no. 10, pp. 71-72.

Woolf, B. 1994, 'Understanding and Using ValueModels', retrieved from http://www.ksccary.com/valujrnl.htm.

WSDL 2001. http://www.w3.org/TR/wsdl

X Business Group. 1994. 'Interface Development Technology'. X Business Group, Inc., Fremont, California.

XML. 2008. http://www.w3.org/TR/REC-xml

Xie, T. & Notkin, D. 2005, 'Automatically identifying special and common unit tests for object-oriented programs', pp. 277–287.

Xie, T., Pei, J. & Hassan, A.E. 2007, 'Mining Software Engineering Data', *International Conference on Software Engineering*, pp. 172-173.

XML 2008, http://w3.org/TR/xml

Xudong, L. & Jiancheng, W. 2007, 'User Interface Design Model', *8th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, vol. 3.

XUL 2001, http://mozilla.org/projects/xul/xul.html

Yang, J., Evans, D., Bhardwaj, D., Bhat, T. & Das, M. 2006, 'Perracotta: Mining temporal API rules from imperfect traces', Association of Computing Machinery, pp. 282-291.

Yin, R.K. 1984, 'Case Study Research: Design and Methods', Sage Publications.