Is There Convergence in the Field of UI Generation?

Richard Kennard
Faculty of Engineering and Information Technology
University of Technology, Sydney
rkennard@it.uts.edu.au

John Leaney
Faculty of Engineering and Information Technology
University of Technology, Sydney
john.leaney@uts.edu.au

Abstract

For many software projects, the construction of the User Interface (UI) consumes a significant proportion of their development time. Any degree of automation in this area therefore has clear benefits. But it is difficult to achieve such automation in a way that will be widely adopted by industry because of the diversity of UIs, software architectures, platforms and development environments. In a previous article, the authors identified five key characteristics any UI generator would need in order to address this diversity. We asserted that, without these characteristics, a UI generator should not expect wide industry adoption or standardisation. We supported this assertion with evidence from 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. Conversely, it would be instructive if they were found to be converging on a different set of characteristics. In this article, the authors look for such evidence of convergence by interviewing the team behind one of the research community's most significant UI generators: Naked Objects. We observe strong signs of convergence, which we believe signal the beginning of a general purpose architecture for UI generation, one that both industry and the research community could standardise upon.

Keywords

user interface generation; convergence; Naked Objects; Metawidget; interview

1. Introduction

For many software projects, the construction of the User Interface (UI) consumes a significant proportion of their development time. Research in the early 1990s 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), therefore any degree of automation in this area has clear benefits. But it is difficult to achieve such automation in a way that is useful and widely adopted by industry. In previous papers (Kennard & Steele 2008; Kennard, Edmonds & Leaney 2009) the authors explored the diversity of UIs, architectures, platforms and development environments that contribute to this difficulty. And in a recent article (Kennard & Leaney 2010) the authors identified five key characteristics any UI generator would need in order to address this diversity. The authors asserted that, without these five characteristics, a UI generator should not expect wide industry adoption or standardisation. We summarised these five characteristics as:

1. Inspecting existing, heterogeneous back-end architectures: the authors found 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) shows, 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 (Kennard & Leaney 2010) showed the raw inspection result invariably needs post processing before it is suitable for consumption by a UI generator. 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 discussed how industry UI libraries were diverging from any notion of a single, ubiquitous UI framework. They 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, p. 13). 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 real world 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.

Having defined these five characteristics, the authors supported them with evidence from industry adoption studies (Kennard & Leaney 2010). 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 we believed, other teams should have been identifying similar constructs. Conversely, if they were found to be converging on a *different* set of characteristics, that would also be very instructive. In this article, the authors look for such evidence of convergence by interviewing the team behind one of the research community's most significant UI generators: Naked Objects. Clearly this is a qualitative measure, not quantitative, but has validity in conjunction with our previous work (Kennard & Leaney 2010) as a triangulation between industry and the research community.

2. Related Work

Research in the field of UI generation dates back over two decades. Projects including, though by no means limited to, COUSIN (Hayes, Szekely & Lerner 1985), TRIDENT (Bodart et al. 1995), UsiXML (Vanderdonckt et al. 2004), OliviaNOVA and OOWS (Valverd et al. 2006) and AUI (Xudong & Jiancheng 2007) have 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 widely varying capabilities.

However most approaches have not taken industry acceptance and adoption as a key driver for their work. This means they have tended to exhibit significant disadvantages from an industry perspective (Myers, Hudson & Pausch 2000). Many require developers laboriously restate information that is already encoded elsewhere in an application. “A common disadvantage of both [interactive graphical editing tools and UI modelling languages] is the fact that the user interface is defined explicitly and separately” (Jelinek & Slavik 2004). This makes them a source of errors should the application code and the UI model not stay synchronised. Other UI generation approaches have imposed generalised interfaces which appear quite differently from, and function less effectively than, those designed with consideration to their specific purpose (Falb et al. 2007). Such disadvantages have led to limited adoption of these tools within industry.

There have been some notable industry successes, however. The Cameleon reference framework (Calvary et al. 2003) defines a Unifying Reference Framework for diverse, ubiquitous devices. The team have worked closely with the World Wide Web Consortium (W3C) with a view to industry standardization. Cameleon introduces “the notions of multi-targeting and plasticity” and “serves as a reference for classifying user interfaces supporting multiple targets, or multiple contexts of use in the field of context-aware computing”. It further incorporates UsiXML (Vanderdonckt et al. 2004), a common User Interface Description Language (UIDL), which has also been submitted to the W3C for standardization. In doing so, it hopes to address a shortcoming of “AUIML, UIML, XAML, XIML, XUL” in that they “result in many XML-compliant dialects that are not (yet) largely used and that do not allow interoperability between tools that have been developed around the[ir own] UIDL”.

As an approach gains acceptance and adoption within industry it generally inspires multiple implementations. These implementations first compete, then converge on a set of core characteristics with some additional features as a differentiator. This convergence ultimately leads industry to standardize upon the core characteristics, reflecting a pivotal moment in the maturity of the approach. This industry convergence can be evidenced in such projects as Hibernate (Hibernate 2009) and TopLink (TopLink 2009) standardizing under JPA (JPA 2009), and Spring (Spring 2009), Guice (Guice 2009) and Seam (Seam 2009) standardizing under JSR-330 (JSR-330 2009).

Convergence, therefore, requires the experience and validation that come from large scale adoption. One approach that has seen significant adoption, being used by large organisations such as the Irish Department of Social Protection and deployed to thousands of users, is the Naked Objects pattern (Pawson 2004). Given the authors' focus on industry applicability, Naked Objects presents as a forerunner in bridging the divide between the theoretical and the practical.

The naked objects pattern considers "an application solely in terms of the [domain] objects. These objects are then rendered directly visible to the user by means of a generic presentation layer. The user undertakes all tasks by directly invoking methods on those [domain] objects. This approach has been dubbed 'naked objects', because as far as the user is concerned he or she is viewing and manipulating the 'naked' business domain objects". If there is any doubt as to the significance of Pawson's approach, one need look no further than the polarising effect it has had on the research community. It has been lauded by renowned pioneers such as Trygve Reenskaug, inventor of the Model-View-Controller (MVC) pattern: "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, p. 3). 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). But the Naked Objects pattern has also been derided by equally renowned pioneers such as Larry Constantine, expert in usage centred design: "The usability problems with [Naked Objects] interfaces under most conditions are too numerous to go into... Pawson and [framework implementer Robert] Matthews, who by their own admission are neither usability experts nor well-versed in user interface design, seem to be blissfully ignorant of the problems... Their book cites sources arguing for 'object oriented user interfaces' but ignores the critical literature that long ago discredited the concept" (Constantine 2002).

The authors, too, have been critical of Naked Objects. When developing our own UI generator, called Metawidget, we have expressed doubts over the practicality of the Naked Objects 'behaviourally-complete' methodology. This methodology dictates "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" (Pawson 2004). Yet Pawson himself recognises "most object-oriented designs, and especially object-oriented designs for business systems, do not match this ideal of behavioural-completeness". The authors believe this is not, as Pawson suggests, because they are poorly designed. Rather, they are seeking to leverage functionality provided by the large number of mature subsystems available in industry, in order to increase productivity and reduce development cost (Kennard & Leaney 2010). For Naked Objects to be applicable to such business systems, it needs to accommodate their approach rather than expect them to be re-architected as behaviourally-complete.

However, all such criticisms are levelled at the version of Naked Objects last described by the academic literature (Pawson 2004). Like any good software project, Naked Objects has evolved over the intervening years. The team now have both Java and .NET implementations (Naked Objects MVC 2010), have published several books (Pawson & Matthews 2002; Haywood 2009), and have continued to refine their approach. They have recently moved to standardise and increase adoption by moving Naked Objects for Java to the Apache Software Foundation under the name Apache Isis. The frameworks implementing the naked objects pattern in 2010 have many differences to the original Naked Objects Framework of 2004.

The authors contacted the Naked Objects team for an update on their work. What we discovered was a surprising amount of independent convergence with our own ideas.

3. Methodology

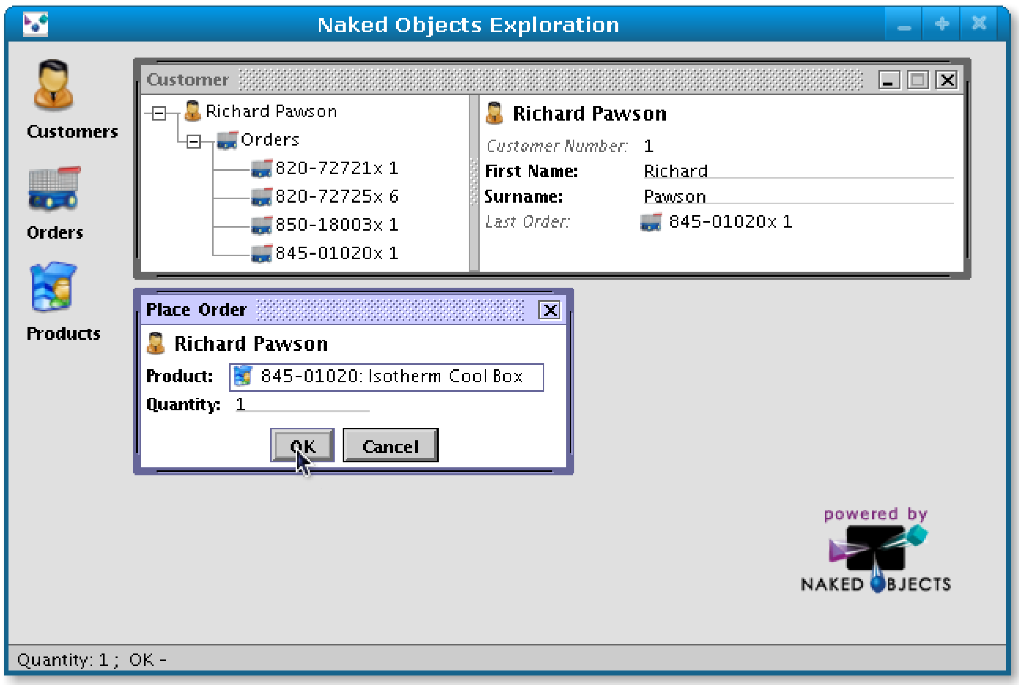
The authors interviewed Dan Haywood over a series of e-mails in late 2010. Dan is a UK-based freelance consultant specialising in enterprise application development using domain driven design approaches and agile development. He is the project lead on the Apache Isis project (the effort to standard Naked 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.

We chose a standardized, 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 minimizing bias. The overarching theme of the interview was the current feature set of the Naked Objects architecture. The interviewee was asked to discuss, and contextualize, aspects of their approach. The goal was to draw out decisions underlying the original design and motivations that led to subsequent changes. We observed patterns of convergence, then posited probe questions (Dick 2005) to drill down and confirm our observations. Unlike our previous interviews (Kennard, Edmonds & Leaney 2009) we were not looking to generalise or codify. Rather every comment, even in isolation, was valuable to help understand the team's research.

The authors and the Naked Objects team had worked independently up to this point, but learnt much about each other's work during the course of the exchange. Most notably, we discovered a surprising amount of convergence around the key characteristics we had previously identified (Kennard & Leaney 2010). In the next section, we record our findings, framed in the context of the five characteristics defined in section 1.

4. Interview

Dan started the discussion with an overview of the current design. Recap that Pawson (2004) had summarized: “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 Object Oriented User Interface (OOUI) as shown in Figure 1. 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.



Richard Pawson
Orders
820-72721x 1
820-72725x 6
850-18003x 1
845-01020x 1

Richard Pawson
Customer Number: 1
First Name: Richard
Surname: Pawson
Last Order: 845-01020x 1

Richard Pawson
Product: 845-01020: Isotherm Cool Box
Quantity: 1

Quantity: 1 ; OK -

Figure 1: Naked Objects' Object Oriented User Interface

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 2). The hexagon core has got two main bits to it – the metamodel (cf. a class) and the runtime (cf. an object). Plugging into the hexagon are the back-end object stores [labelled 'persistence' in Figure 2], and the front-end viewers [labelled 'main' and 'webapp' in Figure 2]”.

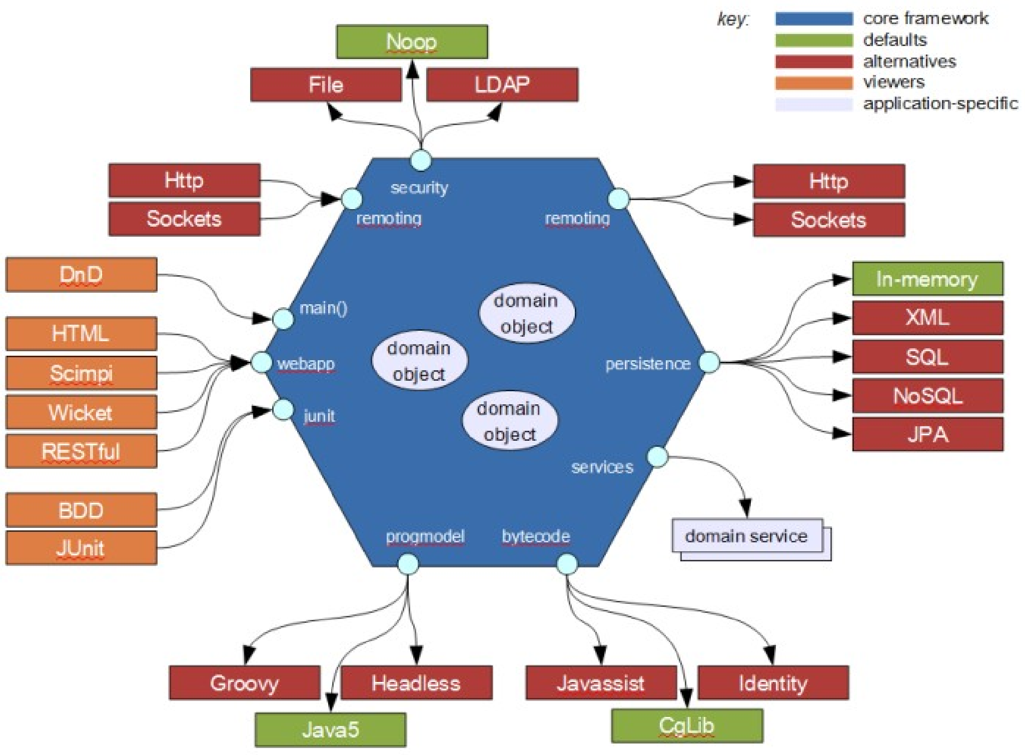


Figure 2: Naked objects hexagonal architecture, as implemented in Apache Isis

“The metamodel [the hexagon in Figure 2] 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]”.

The interview then moved to discuss back-end object stores, the metamodel and front-end viewers in detail. The authors discovered a significant amount of convergence with our own key characteristics, identified previously (Kennard & Leaney 2010).

4.1 Inspecting existing, heterogeneous back-end architectures

The authors had previously identified (Kennard & Leaney 2010) that supporting a mixture of heterogeneous sources of UI metadata was an important characteristic for a practical UI generator. Many business systems are modelled as 'anaemic' (Fowler 2002) or 'dumb' (Firesmith 1996) entities surrounded by controller objects such as persistence contexts, validation subsystems and rule engines. This has implications for frameworks implementing the naked objects pattern, with its original tenet of 'behavioural-completeness' dictating 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” (Pawson 2004). Our adoption studies showed practitioners resisted “many frameworks or tools [that] enforce the [framework] designer's vision on how solutions should be architected” (Kennard & Leaney 2010).

Interestingly, however, the latest release of the naked objects frameworks include a concept called 'facets'. The authors 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 Java Persistence API (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`".

The authors 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 the authors' in that it was "more fine-grained". But the fundamental notion of opening up the naked objects frameworks to metadata from other subsystems, such as persistence contexts, rule engines and XML files, had close parity with our approach.

4.2 Appreciating different practices in applying inspection results

A second key characteristic the authors identified (Kennard & Leaney 2010) 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.

The authors 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. The authors had originally had the `Inspectors` performing the sorting themselves, but factored this out into a separate `InspectionResultProcessor` API.

4.3 Recognising multiple, and mixtures of, UI widget libraries

Another key characteristic the authors identified (Kennard & Leaney 2010) 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.

The authors wanted to gain an understanding of how this characteristic had been handled in the original naked objects pattern. 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 `WidgetBuilders` 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 a 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.

4.4 Supporting multiple, and mixtures of, UI adornments

The authors further identified (Kennard & Leaney 2010) 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”.

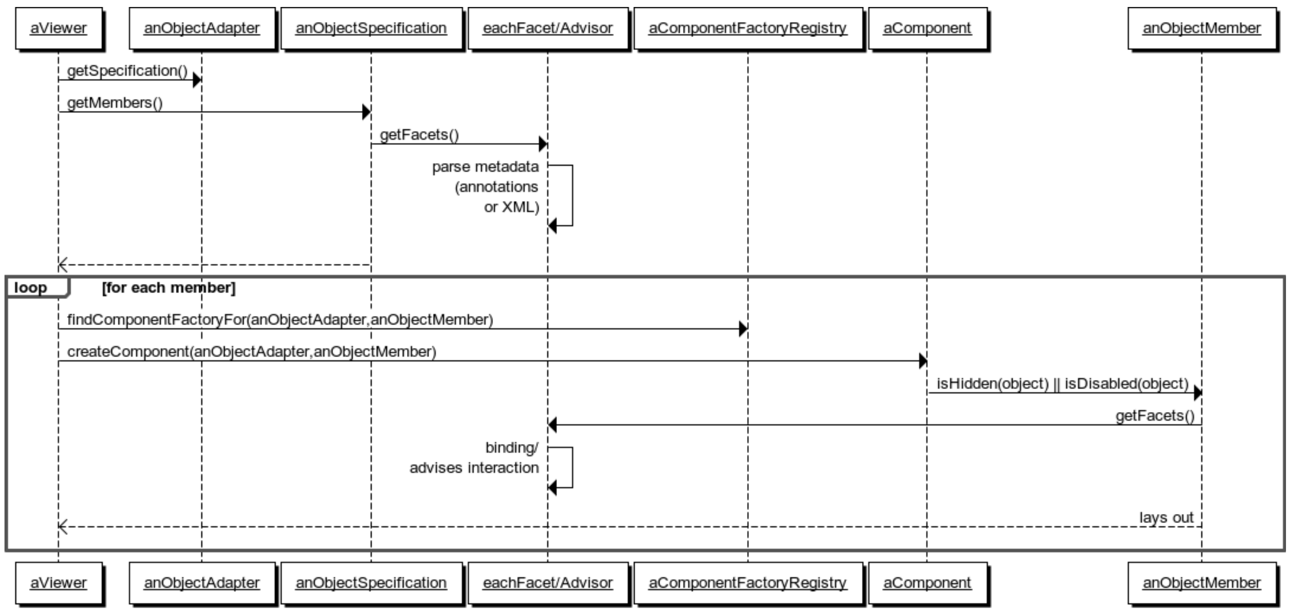

aViewer
anObjectAdapter
anObjectSpecification
eachFacet/Advisor
aComponentFactoryRegistry
aComponent
anObjectMember
getSpecification()
getMembers()
getFacets()
parse metadata (annotations or XML)
loop
[for each member]
findComponentFactoryFor(anObjectAdapter,anObjectMember)
createComponent(anObjectAdapter,anObjectMember)
isHidden(object) || isDisabled(object)
getFacets()
binding/ advises interaction
lays out

Figure 3: Naked objects sequence diagram, as implemented by the Apache Isis wicket viewer

This results in the sequence diagram depicted in Figure 3. The sequence has similarities with Metawidget's own pipeline (Kennard & Leaney 2010) depicted in Figure 4. 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 pattern (Pawson 2004) this showed strong evidence of convergence.

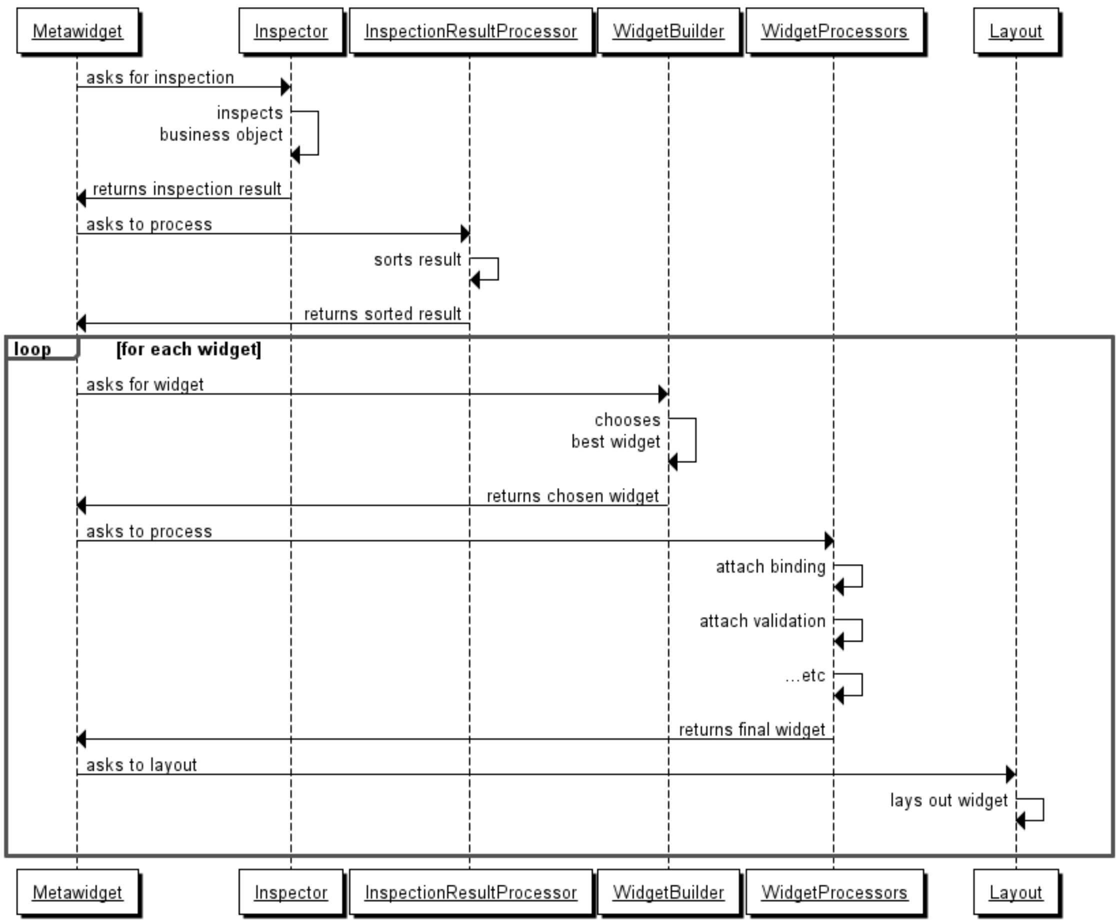


Figure 4: Metawidget sequence diagram

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 a `ObjectMember` implements `FacetHolder`, it could perhaps be also an `AdvisorHolder`. This would separate out these concerns, converging our two designs further".

4.5 Applying multiple, and mixtures of, UI layouts

A final characteristic the authors identified (Kennard & Leaney 2010) was supporting multiple ways to arrange widgets on the screen. We pointed out that 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, p. 13). This realisation exposes a myriad of small details around UI appearance, navigation, menu placement and so on. The problem is so difficult, in fact, we believe it insoluble.

Metawidget sidesteps the issue by not attempting to generate the entire UI. Rather, it believes there are 'useful bounds' to UI generation: “as automatic UI generation moves away from [simply modelling fields as forms] it rapidly becomes highly 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 once 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” (Kennard & Steele 2008). As Constantine (2002) puts it “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 UI to fit the unique aspects of the context, the application, and the user needs, one [generic presentation layer] is presumed to fit all problems”. Instead, Metawidget focuses on generating only a small piece of the UI – the 'inside' of each page, the area around the fields themselves. The UI appearance, navigation, menu placement and overall usability are far more subjective and we explicitly keep these out of scope.

Even after Metawidget has restricted its 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 there are many other real world arrangements, such as all fields arranged in a single, horizontal row; or right-to-left arrangements for the Arabic world. It is important to accommodate this variability if the generator is to achieve the exact look the practitioner desires. Metawidget addresses this characteristic of supporting multiple ways to arrange widgets by defining pluggable layouts.

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 frameworks provide 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 1 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".

5. Convergence in Other Projects

This article has explored convergence between our own work and that of the Naked Objects team. Given the intricacies and subtleties of the design decisions involved, it was necessary to cover their project in depth and in detail. Therefore this study focussed on only a single project.

Clearly there are other UI generator projects that also demonstrate a significant industry presence. For example, UsiXML (Vanderdonckt et al. 2004) is a cross device, platform and modality UIDL that has been submitted to the W3C industry body for standardization. OlivaNOVA (Valverde et al. 2007) is a commercially available product that can generate UIs for both Java and Microsoft .NET environments. Such projects have differentiating characteristics beyond, though not necessarily mutually exclusive to, our own five. We have asserted that our five characteristics are fundamental for a UI generator to be useful to industry. The characteristics have been derived over successive iterations of Action Research cycles with industry practitioners, and validated by industry adoption studies and open-ended interviews (Kennard & Leaney 2010). Demonstrably, they are important enough that the Naked Objects team found the need to add them atop their original design.

However ours are not the only characteristics by which UI generators may be assessed, nor the only ones by which they may converge. Examples of other generators and their differentiating characteristics are shown in Table 1. This table shows a selection of features drawn both from our own five and from the feature sets of the generators as described in the literature. It can be seen there are differing approaches to implementing any given characteristic, and some can be left to the practitioner to implement manually without the UI generator's explicit support. This may or may not be enough for the generator to prove sufficiently useful to industry.

Feature/Generator	Metawidget	Naked Objects	UsiXML	OlivaNOVA
Inspecting existing, heterogeneous back-end architectures	Yes	Yes (see 4.1)	No, has its own UIDL	No, has its own UIDL
Appreciating different practices in applying inspection results	Yes	Yes (see 4.2)	No, but can edit generated code	No, but can edit generated code
Recognising multiple, and mixtures of, UI widget libraries	Yes	Yes (see 4.3)	No, but can edit generated code	No, but can edit generated code
Supporting multiple, and mixtures of, UI adornments	Yes	Yes (see 4.4)	No, but can edit generated code	No, but can edit generated code
Applying multiple, and mixtures of, UI layouts	Yes	Partial (see 4.5)	No, but can edit generated code	No, but can edit generated code
Mode of operation	Runtime	Runtime	Static code generation	Static code generation
Translation to different devices, platforms, modalities	Manual (see 4.5)	Automatic	Automatic	Automatic
Object Oriented User Interface	No	Yes (see 4)	No	Yes

Table 1: Examples of UI generators with significant industry presence

Research projects such as UsiXML and OlivaNOVA will require further exploration to look for signs of convergence. If our characteristics are as fundamental as we believe, these research teams should be identifying similar constructs through their own industry field trials. Conversely, if we find these teams to be converging on a *different* set of characteristics, that would also be very instructive.

6. Conclusions and Future Work

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

In our previous article (Kennard & Leaney 2010) the authors reasoned that implementation of all five of our characteristics would lead to a notable emergent advantage: being able to retrofit an existing application that was not built with UI generation in mind. 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 the naked objects frameworks do not, because there is still a gap around pluggable layouts which limits the categories of applications the naked objects pattern 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 the authors consider it significant that a good many core, non-obvious characteristics have been established. These have been agreed upon both by our own industry adoption studies, and independently by the industry field trials of the Naked Objects team. We believe these signal the beginning of a general purpose architecture for UI generation, one that both industry and the research community could standardise upon. This standardisation could then be used to drive adoption and ultimately to realise the full potential of UI generation technology.

References

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, SpecCeification and Verification of Interactive Systems*. Wien: Springer, pp. 262-278.

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.

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

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.

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

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', *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, ed. P. Roman, 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.

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.

Haywood, D. 2009, 'Domain Driven Design using Naked Objects', Pragmatic Bookshelf.

Hibernate 2009. http://www.hibernate.org

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, IEE Proceedings*, vol. 152, no. 3, pp. 344-352.

JSR-330 2009. http://www.jcp.org/en/jsr/detail?id=330

JPA 2009. http://jcp.org/en/jsr/detail?id=220

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

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

Myers, B.A. & Rosson, M.B. 1992, *Survey on user interface programming*, ACM Press New York, NY, USA.

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.

Naked Objects MVC 2010. http://nakedobjects.net

Pawson, R. 2004, 'Naked Objects' thesis, Trinity College, Dublin.

Pawson, R. & Matthews, R. 2002 'Naked Objects', Wiley.

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.

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 2009. http://springframework.org

TopLink 2009. http://www.oracle.com/technetwork/middleware/toplink

Valenzuela, D. & Shrivastava, P. 2002, 'Interview as a Method for Qualitative Research', http://www.public.asu.edu/~kroel/www500/Interview%20Fri.pdf.

Valverde, F., Valderas, P., Fons, J. & Pastor, O. 2007, 'A MDA-Based Environment for Web Applications Development: From Conceptual Models to Code', Citeseer.

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. Sophia Antipolis*, pp. 19-20.

Weiser, M. 1993, 'Hot topics-ubiquitous computing', *Computer*, vol. 26, no. 10, pp. 71-72.

Xudong, L. & Jiancheng, W. 2007, 'User Interface Design Model', *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, vol. 3.

Biography

Richard Kennard is a PhD student in the Faculty of Engineering and Information Technology at the University of Technology, Sydney. He is an independent industry consultant with fifteen years experience and an active member of the Open Source community.

John Leaney is an Adjunct Professor in the Faculty of Engineering and Information Technology at the University of Technology, Sydney.