Towards a General Purpose Architecture for 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

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. Such automation is difficult due principally to the diversity of architectures, platforms and development environments. Attempts to automate UI generation to date have contained restrictions which did not accommodate this diversity, leading to a lack of wide industry adoption or standardisation. The authors set out to understand and address these restrictions. We studied the issues of UI generation (especially duplication) in practice, using action research cycles guided by interviews, adoption studies and close collaboration with industry practitioners. In addressing the issues raised in our investigation, we identified five key characteristics any UI generation technique would need before it should expect wide adoption or standardisation. These can be summarised as: inspecting existing, heterogeneous back-end architectures; appreciating different practices in applying inspection results; recognising multiple, and mixtures of, UI widget libraries; supporting multiple, and mixtures of, UI adornments; applying multiple, and mixtures of, UI layouts. Many of these characteristics seem ignored by current approaches. In addition, we discovered an emergent feature of these characteristics that opens the possibility of a previously unattempted goal - namely, retrofitting UI generation to an existing application.

Keywords

user interface generation; software mining; retrofitting; action research; interviews; adoption studies

1. Introduction

Many software projects spend a significant proportion of their time developing the User Interface (UI). 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. This automation 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, a range of choices drawn from within application code, and a look and feel specified by a human-readable design document. Bringing these diverse characteristics together to enable automatic UI generation is a significant challenge and research in this field dates back over two decades. 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. Approaches to date can be broadly grouped into three categories: interactive graphical specification tools, model-based generation tools, or language-based tools (Myers 1995). 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 developers to restate information that is already encoded elsewhere in the application. This duplication is both laborious and a source of errors, as the developer 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, p. 540). Without these new attributes and properties, language-based tools must resort to generalised heuristics when generating their UIs. These generalisations invariably mean 'the generated User Interfaces [are] not as good as those created with conventional programming techniques' (Myers, Hudson & Pausch 2000, p. 13).

UI generation has not experienced the same wide industry adoption and standardisation as, say, Object Relational Mapping (ORM) (JPA 2008). We claim that this is because existing approaches are impractical, not because the issue itself lacks urgency. A series of interviews conducted with practitioners across industry segments and software platforms (Kennard, Edmonds & Leaney 2009) found common experiences of duplication (Jelinek & Slavik 2004), common experiences of bugs caused by it, and a common desire for it to be addressed. One interviewee confirmed "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 [ORM] which do from, sort of, business layer down". Another lamented that, when making a typical change to an enterprise system "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". Some practitioners gave concrete examples "we've got a BigDecimal (Gosling 2005), and [the back-end has] set the scale to 8 but the GUI 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". One developer reflected "it's a fairly established software engineering principle that the more you have to repeat something [defining fields in both the business layer and the UI layer] the higher the chances there's going to be an error in the code". Finally one engineer summarised "every developer who writes anything more than a Hello World application will have this problem. Most developers encounter this problem on a daily basis as a constant friction in their daily work". During publication of our 2009 paper, reviewers reframed our interviews in an academic context “the work is exceptionally well motivated... it *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”.

Having established the urgency of the problem, the authors set out to address the impracticalities with existing UI generation approaches. We sought a well-motivated, well-justified approach guided by practice based research and human centred design approaches. These included open ended interviews, adoption studies and close collaboration with industry practitioners, as described in section 2. In doing so we identified five key characteristics we believe are necessary for any UI generation technique to achieve wide adoption or standardisation. Many of these characteristics seem ignored by current approaches. The full adoption studies will be published at a later date, but we discuss each characteristic in detail in section 3. In addition, we discovered an emergent feature of these characteristics is that they open the possibility of retrofitting - a previously unattempted goal - which we explore in section 4. Finally, we discuss future work in section 5.

2. Methodology

Given our strong focus on industry practice, and ensuring any UI generation is practical, it was important to engage our 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, 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' (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.

Our research methodology employed action research as a framework within which to formalise iterative development, allowing the work to be both accessible to, and guided by, observations from both industry and the research community. Observations were gathered through forum messages, open-ended interviews using simplified grounded theory (Dick 2005) and adoption studies[1]. Adoption studies were gathered retroactively from companies who had independently discovered and decided to apply our work, of their own volition, based on publicity from previous action research cycles. Companies were discovered through message forums then contacted to solicit a study of their experiences. We conducted seven such studies in all, ranging from private biotechnology and telecommunications companies, to government departments and academic institutions.

The last phase of each action research cycle reflected upon the observations from interviews and adoption studies as drivers for the next cycle. Action research delineates an explicit 'reflect' phase (Kemmis & McTaggart 1988) so as to bring academic rigour to the proceedings. During the 'plan', 'act' and 'observe' phases, reflection happens 'in action' (Schön 1983) with the 'action present' measured in minutes or hours. During the 'reflect' phase, however, the 'action present' is much more deliberate, being measured in days or weeks, which lends more diligence and formality to the Validity, Verification and Testing (VVT) of the research. It facilities the reconsideration of all design decisions, implementation details and observations from the preceding action research cycle in a holistic light, leading to key new insights.

This practice based methodology, conducted across three action research cycles over a period of

1 Adoption studies are interviews focussed 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. The interviewee is asked to discuss, and demonstrate, aspects of their adoption experience. The guiding questions include: what led them to adopt the technology; have they used or built similar technologies in the past; what were some of the most important features to them; where did they find the technology lacking.

three years, afforded us a breakthrough in our understanding of the problem domain. Our most significant finding is the identification of five key characteristics any UI generation technique needs before it can be practical in an industry setting. Notably, many of these characteristics seem ignored by current UI generators, which we believe is hindering their widespread adoption. The next section presents our action research results in detail.

3. Characteristics of a Practical UI Generator

This section explains the five characteristics we believe, based on our action research cycles, are key to a practical UI generator. We have derived these characteristics through interviews, adoption studies and close collaboration with industry practitioners. They can be summarised as: inspecting existing, heterogeneous back-end architectures; appreciating different practices in applying inspection results; recognising multiple, and mixtures of, UI widget libraries; supporting multiple, and mixtures of, UI adornments; applying multiple, and mixtures of, UI layouts. We will justify each of these characteristics in turn in the following sections. We will begin each section by first framing the characteristic within the literature and within observations from our discovery research. We will then move to justify each characteristic and discuss its technical details. Because of our strong focus on a practical UI generator, we will discuss these technical details in the context of our own implementation, which we have called Metawidget (http://metawidget.org). This implementation is the outcome of our action research, and the proof of our work.

Before we begin, however, it is important to first frame how the five characteristics fit together. There are a number of ways they could be designed into a UI generator, though we would caution against any approach based on presenting practitioners with multiple options or 'flags' - such approaches likely underestimate the sheer volume of variations and combinations the characteristics need to accommodate. In our particular implementation we have implemented the five characteristics as a pluggable pipeline (Vermeulen, Beged-Dov & Thompson 1995) as shown in figure 1. Practitioners can plug in alternate implementations, or custom implementations, of each characteristic. This allows our implementation to accommodate the volume of variability while avoiding the performance overhead and maintenance cost of an exponential combination of 'flags'.

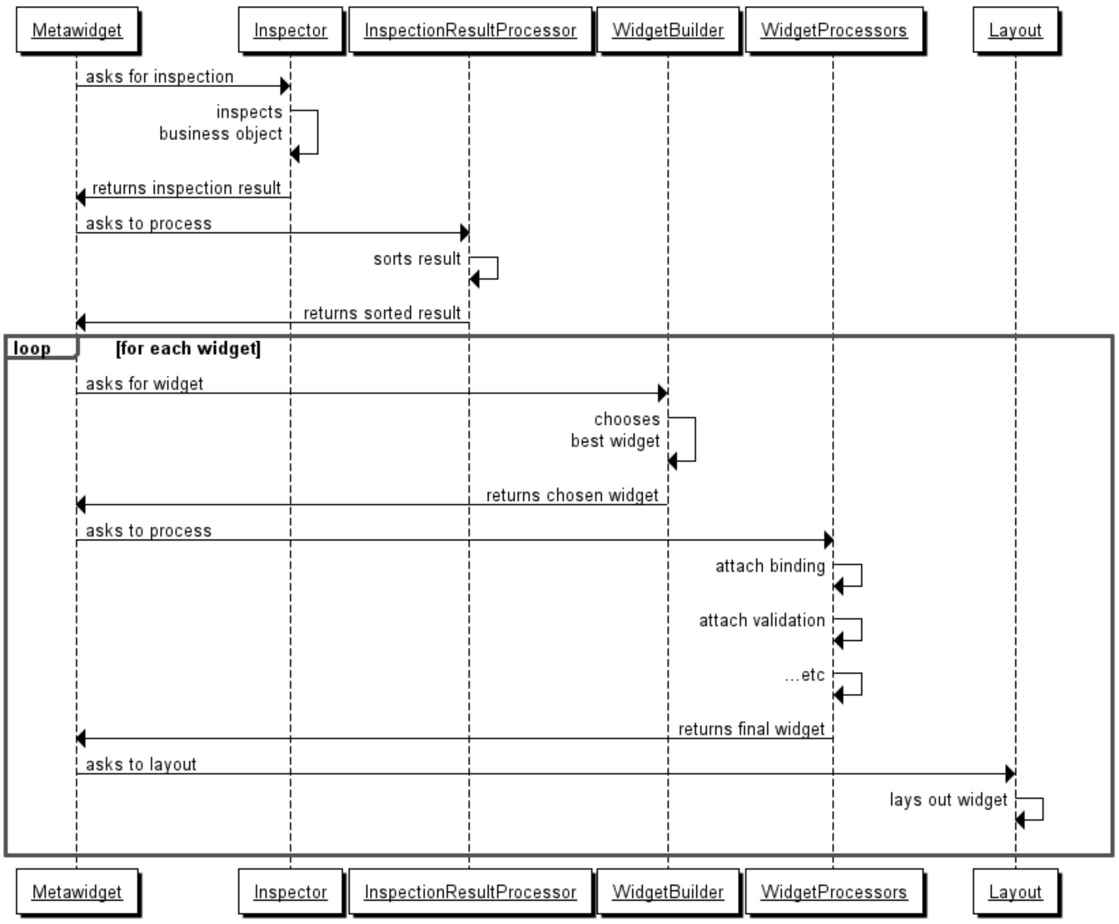


Figure 1: Metawidget pipeline, showing the five characteristics

The leftmost 'lifeline' in the UML sequence diagram (the Metawidget) serves to coordinate the others and is not considered part of the pipeline. The remaining five UML lifelines correspond to the five key characteristics we have identified from our action research. The UI generation proceeds in a well-defined fashion along each stage of the pipeline, from left to right. Each stage is pluggable. As an implementation detail, each stage is also immutable. This helps with memory consumption and performance when multiple Metawidgets are used in the same UI.

Each of the UML lifelines and their corresponding characteristic is discussed in turn in the sections to follow. We begin with the first UML lifeline, the 'inspector'.

3.1. Pluggable Inspectors

Most UI generation solutions dictate that practitioners provide the generator with a centralised, single source of truth (SSOT). The solutions do this by defining their own UI modelling language or graphical builder tool, then require the practitioner to use their tool to define the UI. For example, UsiXML (Vanderdonckt et al. 2004) consists of “a User Interface Description Language (UIDL) that is a declarative language capturing the essence of what a UI is or should be independently of physical characteristics. UsiXML describes at a high level of abstraction the constituting elements of the UI of an application: widgets, controls, containers”. GUMMY (Meskens 2008) is designed “in a similar way to traditional GUI builders in order to allow designers to reuse their knowledge of single-platform user interface design tools... there is a toolbox showing the available user interface elements, a canvas to build the actual user interface and a properties panel to change the properties of the user interface elements on the canvas”. Internally, “GUMMY builds a platform-independent representation of the user interface and updates it as the designer makes changes. This allows for an abstract specification of the user interface” which it then uses to drive UI generation for multiple physical devices. However as Jelinek and Slavik (2004) observe, “a common disadvantage [of modelling languages and graphical builder tools] is the fact that the user interface is defined explicitly and separately” and therefore “the application and the corresponding [UI] model need to be kept consistent”. Prat et. al (1990) agree, finding that in practical application design “the [back-end] requires a considerable amount of knowledge [much of which is] similar to that required by the [UI] modules”.

Some UI generation solutions acknowledge this shortcoming and seek to reuse existing information already embedded in an application's architecture. Unfortunately they still require this to be rigidly centralised. For example, Naked Objects 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). Most industry systems do not adhere to such a 'behaviourally-complete' methodology. Rather, they use what Firesmith (1996) calls “dumb entity objects controlled by a number of controller objects”. They do this in order to benefit from a rich ecosystem of technologies. What Firesmith terms 'controller objects' industry practitioners would term, but not be limited to, validation subsystems, workflow subsystems, rule engines and Business Process Modelling (BPM) languages. Indeed, as Pawson himself later concedes “most object-oriented (OO) designs, and especially object-oriented designs for business systems, do not match this ideal of behavioural-completeness”. It is important to appreciate this is not because most business systems 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. It is still possible to have an SSOT whilst being decentralised amongst multiple subsystems, provided there is no overlap in the decentralisation (i.e. the workflow subsystem is concerned with different aspects of the application to the validation subsystem).

Our adoption studies underscored this point with industry practitioners. One commented “many frameworks or tools enforce the designer's vision on how solutions should be architected. What I liked about [your implementation] is that I could drop it in whatever architecture I was using”. The phrase 'whatever architecture I am using' turned out to be critical, because real world architectures were found to vary widely - dependent on both hard, business requirements and softer, aesthetic judgements.

As an example of a business requirement, many architectures abstract their UI data into OO classes. It is certainly possible to generate UIs based on instances of those classes - indeed, this is the approach Naked Objects mandates. But business requirements often prevent this approach and it was found that being able to plug in a company's own back-end inspector was fundamental to the usefulness of a UI generator. One adoption study had defined the UI in their own, proprietary XML. “[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 [was a key strength]”. Another study had used rules in a database: “The main feature for us was the possibility to dynamically, based on rules stored in our database, create input screens based on user selections... 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 to support everyone's requirements [out of the box]... otherwise we probably would not have even tried it”. Another adoption study used a mixture of third-party libraries: “we work with JDO, OVal, and some custom annotations, so being able to extend was a must for us”. Another adoption study “we did not want to place view stuff into the model, and we did not want to place it in an [external] XML file either, because we would have to replicate the property name. [Instead] we built [an inspector] based on our properties files”. By 'properties files' the team is referring to reading UI information from localization (internationalization) resource bundles. These examples demonstrate 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).

A common example of an aesthetic judgement revolved around the mixing of presentation information with business logic. Some adoption studies reported “there was some spirited debate [in the development team], since [annotating 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]”. Another said “business objects should be neutral regarding presentation - I am a supporter of separated tiers”. But other adoption studies disagreed “while I appreciate the power within the XML inspectors, I used annotations to configure [my business objects]”. Another “I don't mind [annotating], an annotation is just metadata”. Interestingly, one study considered it a matter of scale: “for small projects it might not be a concern, so we find [the fact that it is supported] 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”.

It is important to note practitioners are not simply talking about *supporting* many different back-end technologies. Rather, they are talking about being able to *mix* technologies, including custom subsystems and alternate implementations of the same subsystem. The latter is less an issue in 'full stack' proprietary software environments such as the Microsoft application stack, but in Open Source enterprise ecosystems such as Java EE there are often dozens of competing implementations of the same subsystem (Shan et al. 2006). There are even competing implementations of the core language (e.g. Java versus Groovy versus Scala). Software architecture therefore involves a myriad of choices, many of which have no 'right' or 'wrong', and opinions on which evolve over time. For example, business rule engines are becoming increasingly popular (Rouvellou et al. 1999). Any UI generator that seeks to dictate, rather than adapt to, a system's architecture therefore has limited practical value.

Our implementation addresses this characteristic of supporting mixtures of back-end technologies by defining pluggable 'inspectors'. It defines a minimal 'Inspector' interface and ships with a number of implementations of this interface to support extracting metadata from different, heterogeneous subsystems. The parent Metawidget (leftmost lifeline in figure 1) requests information regarding a particular business object and it is the responsibility of each individual inspector to gather as much information as possible about that business object from its particular subsystem. A current limitation and open question of this approach is how to validate the completeness of the inspection. The issue of completeness is ongoing work.

Returning to our theme of an SSOT, our implementation also supplies a CompositeInspector, named after the composite design pattern (Gamma et al. 1995), to support *combining* the inspection results from multiple inspectors into a more detailed whole result, as shown in figure 2. This more detailed whole result forms a *temporary* centralised SSOT from the subsystems it is split across, so that it can be used to drive UI generation. This extraction and collation of metadata from multiple, heterogeneous subsystems is commonly referred to as software mining[2].

2 Software mining relates to the discovery of artefacts in code (Xie, Pei & Hassan 2007) for purposes such as constructing domain ontologies (Grcar, Grobelnik & Mladenci 2007) or identifying call usage patterns (Kagdi, Collar & Maletic 2007). This is distinct from the statistical collection of information to understand relationships (in for example, customer relationship management) which is typically referred to as data mining (Cao et al. 2010; Li, Zhang & Wang 2006).

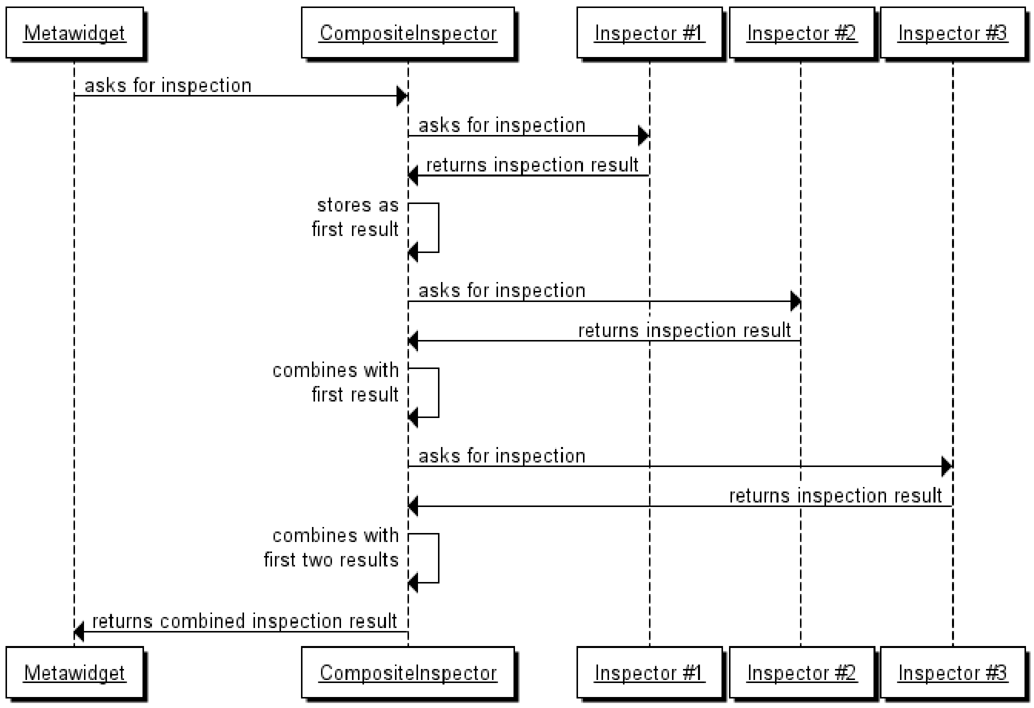


Figure 2: CompositeInspector implements Software Mining

It is often said in software development that each design decision should “pull its weight” (Bloch 2001). That is to say, it is a validation of good design if each decision provides multiple advantages that demonstrably outweigh the inevitable disadvantage of its added complexity. Encouragingly, several of our five characteristics demonstrate such emergent advantages. For inspectors, an emergent advantage of CompositeInspector is that it becomes possible to run *remote* inspections. Multiple groups of inspections can be run remotely on different application tiers, passing the inspection results back to the UI in a well-defined, secure manner, where they can be recombined. This is important in real world architectures where often the UI layer is prohibited from, say, directly accessing the database schema.

To reiterate, throughout section 3 we are referencing our own particular implementation of each characteristic in order to provide a deeper analysis. Clearly, however, ours is not the only approach. For example, the Naked Objects team report (Haywood 2008.1) they are introducing pluggable 'facets' as a method of supporting mixtures of back-end technologies, including XML and database sources. This somewhat relaxes their 'behaviourally-complete' methodology and suggests some convergence (Haywood 2009).

This section has demonstrated that supporting a mixture of heterogeneous sources of UI metadata is an important characteristic for a practical UI generator. Having retrieved and collated all available metadata from the back-end subsystems, it is generally necessary to post process it. This characteristic is discussed in the next section.

3.2. Pluggable Inspection Result Processors

The raw inspection result returned from the inspector invariably needs post processing before it is suitable for consumption. For example, the fields generally need to be arranged in a business-defined order. There are various ways to achieve this, dependent on the metadata source. For example, if the data is sourced from an XML document its nodes are inherently ordered (XML 2008). But if the data comes from JavaBean properties in Java class files then it will not retain any ordering information (Gosling 2005) so one must be imposed. In the latter case, our adoption studies showed that the method used to impose ordering is open to practitioner taste. By default our implementation uses 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". Equally, the Naked Objects team reports “by way of comparison (and as an alternative idea), the Naked Objects programming model uses an annotation called @MemberOrder, which takes an ordering in Dewey decimal notation. So, we have @MemberOrder("1.1")” (Haywood 2008.2).

Such explicit field ordering at the business model-level has disadvantages, however, as one practitioner noted "adding [field ordering information] to basically every field of your business model strongly reduces clarity... one of the key principles of [pluggable inspectors] is the possibility to directly use your unchanged domain objects [but this] doesn't really follow that principle”. It is also less flexible in cases where “one wants to automatically create many different views based on a single business object with components of different sequence and visibility”. The alternative, ordering the fields at the UI-level, introduces duplication - re-stating the fields used in a business object (Jelinek & Slavik 2004) - and compromises polymorphism - the UI needs to statically know the fields in advance - but practitioners advocated both sides: “I agree, the default behaviour [model-level ordering] should be as it is now... still, I see [UI-level ordering] as an advantageous option for cases with specific concerns such as flexibility".

Field ordering, then, whether specified by the back-end or the front-end, must be pluggable. Another example of an inspection result post processing operation, open to similar interpretations of practitioner taste, is excluding fields: should the model-level dictate which fields are to be excluded from the UI, or should this be decided on a per-screen basis? Most likely, a combination of both would be required – some clearly inappropriate fields, such as database primary keys, should be excluded from the UI at the model-level whereas other fields may need excluding on a per-screen, or per-user-role basis. Such post processing requirements are commonplace and a UI generator that does not accommodate different practitioners' preferred approaches would limit its appeal.

Our implementation addresses this characteristic of supporting multiple ways to post process the UI data by defining pluggable 'inspection result processors'. It defines a minimal 'InspectionResultProcessor' interface and ships with a number of implementations of this interface to support post processing metadata. Distinct from inspectors, which are designed to be detached from the UI and executable on application tiers inaccessible to the UI, InspectionResultProcessors maintain a reference to the current UI page. This is required for the aforementioned ability to implement UI-level ordering and exclusion.

This section has demonstrated that supporting a variety of ways to post process UI metadata is an important characteristic for a practical UI generator. Once post processing is complete, the inspection result is ready to drive the UI generation, as discussed in the remaining three sections.

3.3. Pluggable Widget Builders

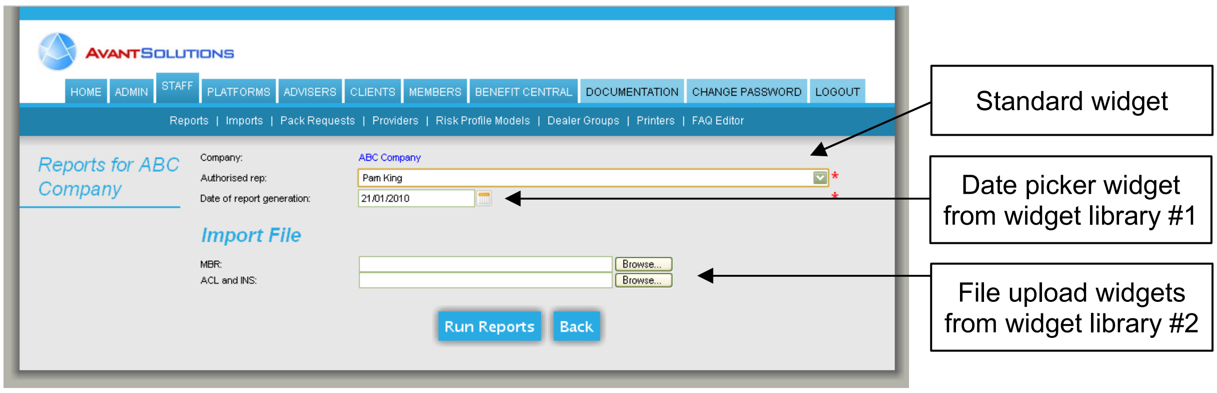
An SSOT acts as a valuable starting point for UI generation. We have discussed how, in industry, this metadata is seldom centralised but rather must be brought together from disparate sources. There does not appear to be any movement among industry systems to converge this situation. On the contrary, the movement is generally towards *additional* types of subsystems, such as business rule engines (Rouvellou et al. 1999). Similarly, and despite research community ideals to the contrary, industry UI frameworks continue to diverge. Most notably, the Web browser does not appear to be the ubiquitous UI for which many were hoping.

For example, when Apple debuted their iPhone in 2007, they originally advocated their Safari Web browser as the recommended way to develop applications for the mobile device. A year later, however, the pressure for native UIs was recognised and a traditional SDK was released. This allowed better performance and access to device-specific hardware such as accelerometers and third-party peripherals (Square 2010). The Google Android mobile platform similarly supports native UI applications in addition to browser-based ones. Even if the browser *were* to become the ubiquitous UI, the plethora of Web frameworks and approaches to Web development (Shan et al. 2006) suggest there will not be a convergence of UI platforms in the near future.

This need to support a variety of front-end frameworks was apparent in several adoption studies. One reported: “We needed to integrate 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)”. Another: "if I'd had to [design my architecture] differently because of [your implementation] I would have been unhappy. As things are I was just able to treat it as a normal Swing widget which was nice".

Another important requirement for a UI generator is that it support third-party, or custom, widgets. This is important not only for flexibility but also robustness. For example, a frequent problem when developing Web applications is ensuring compatibility across browsers. By leveraging existing widget libraries, rather than attempting to generate HTML directly, a UI generator can delegate all issues of browser incompatibility to the widget library – which has typically already been vetted in a variety of production environments. An adoption study from an earlier action research cycle criticised "if you have a more exotic GUI component used for certain properties, [your implementation requires] more work needed to get that to render, as opposed to simply creating the component in your GUI layer". After we improved this, adoption studies from later action research cycles reported: "We wrote our own widget builder.... [to use custom] components we developed and Metawidget instantiates them".

Implicit to this requirement to support third-party widget libraries is the ability to mix *multiple* third-party widget libraries in the same UI, as shown in figure 3. Most third-party libraries only specialise on a certain set of widgets, rather than trying to replace every widget the platform provides.


AVANTSOLUTIONS
HOME
ADMIN
STAFF
PLATFORMS
ADVISERS
CLIENTS
MEMBERS
BENEFIT CENTRAL
DOCUMENTATION
CHANGE PASSWORD
LOGOUT
Reports | Imports | Pack Requests | Providers | Risk Profile Models | Dealer Groups | Printers | FAQ Editor
Browse...
Browse...
Run Reports
Back
Standard widget
Date picker widget from widget library #1
File upload widgets from widget library #2

Reports for ABC Company

Company: ABC Company
Authorised rep: Pam King *
Date of report generation: 21/01/2010 *

Import File

MBR:
ACL and INS:

Figure 3: Mixing multiple widget libraries in a UI

Our implementation addresses this characteristic of supporting mixtures of widget libraries by defining pluggable 'widget builders'. It defines a minimal 'WidgetBuilder' interface and ships with a number of implementations of this interface to support popular UI frameworks, such as the aforementioned Spring MVC, Swing and Java Server Faces. Notably, our implementation also supplies a CompositeWidgetBuilder, named after the composite design pattern (Gamma et al. 1995), to support *combining* the widget libraries from multiple WidgetBuilders. The ordering of the WidgetBuilders is significant, so that widget choice can prefer one third-party library's widgets over another, or fall back to the default platform widgets, as shown in figure 4.

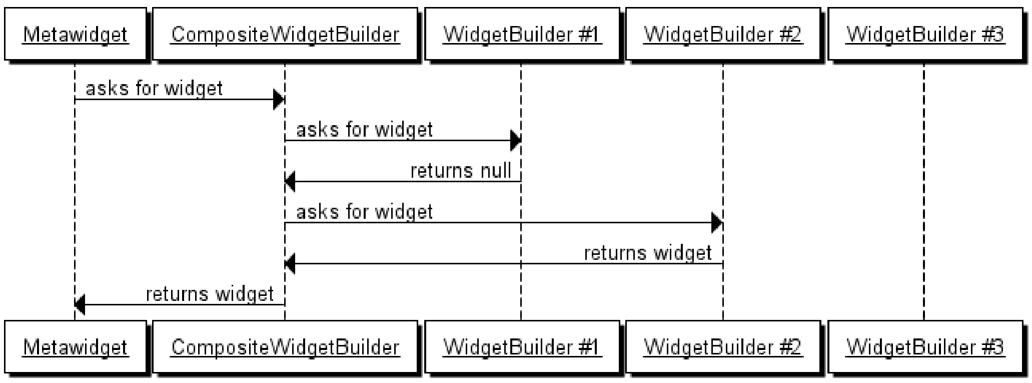


Figure 4: CompositeWidgetBuilder combines multiple widget libraries

As with inspectors, widget builders demonstrate emergent advantages. Specifically, they present a compelling use case for practitioners to adopt UI generation. Once the developer has delegated their widget creation to a widget builder, it becomes possible to plug in new widget libraries as they become available – either new versions of existing libraries, or competing libraries offering more desirable widgets. For example, a third-party widget library may be released that offers a 'colour picker' widget with a user-friendly colour wheel. Imagine a practitioner has an application whose existing approach is to present a text field constrained to only accept hexadecimal input in RGB format (e.g. #ff00ff). The text field works, but the practitioner decides the colour wheel is more usable (it doesn't require the user understand hexadecimal, for one). Using widget builders, the practitioner would simply need to insert a WidgetBuilder for this new widget library as the first in a CompositeWidgetBuilder's list of priorities. Every screen in the application that previously used the hexadecimal text field would be immediately upgraded to use the colour picker, and this would happen automatically across the entire application – a significant saving in practitioner time over upgrading each screen manually. Notably, the WidgetBuilder itself could be provided by the widget library author, their incentive being to increase ease-of-adoption of their library. This reasoning extends to other areas of our pipeline too, such as inspectors, widget processors and layouts (covered in subsequent sections): an author of a new product, such as a business rules engine, a validation subsystem, or a layout manager, could increase its install base by providing UI generator plugins for it – alleviating the need for a practitioner to learn much of their product's API and significantly easing its adoption. Such plugins are not unprecedented, and grow the ecosystem of a product segment by levelling the playing field for new entrants. For example, database vendors typically provide plugins to popular ORMs so as to ease adoption amongst practitioners using those ORMs.

This section has demonstrated that supporting mixtures of widget libraries is an important characteristic for a practical UI generator. Widget builders simplify the process of choosing the most appropriate widget for a business field. Simply *choosing* the widget is rarely sufficient, however. There are generally a host of supporting technologies that also need to be attached, such as data validators, data binding frameworks, and event handlers. This characteristic is discussed in the next section.

3.4. Pluggable Widget Processors

In raw form, a widget is not likely 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 validators, may come from a different third-party library than the raw widget.

Processing such adornments is made difficult because of a general problem with automatic generation of any kind, not just UI generation: generated code is opaque to the practitioner. It is difficult to reference and attach mechanisms to objects that are not well-known in advance. One early adoption study identified this as “when you want to customise [the generated widgets], like replacing or adding more info to [them] you have to refer to them by property names. We have this problem not only for [UI generation], but when you have a lot of dynamic stuff [such as ORM frameworks]. It would be nice to either solve this or offer a solution for that”. Another gave a concrete example, needing “a way to attach event handlers to widget value changes. This would allow you to respond to change... not just do a bi-directional binding (for example you could enable a save button that starts disabled)”. A third adoption study had custom mechanisms they wanted to attach: “[we'd like to be] able to integrate our own validation and custom rendering of components”.

Practitioners require a pluggable mechanism that allows post processing of a built widget. Critically, this mechanism must expose the same richness of metadata as the original widget building did, so that widgets may be identified not just by their name but also by their type, their constraints, their labels or any amount of other metadata.

Our implementation addresses this characteristic by defining pluggable 'widget processors'. It defines a minimal 'WidgetProcessor' interface and ships with a number of implementations of this interface to support popular adornments, such as the aforementioned data validators and data bindings, as shown in figure 5. Many of these implementations are defined by third-party libraries, not by the same base UI library as the raw widget. Notably the list of widget processors is maintained not by an immutable CompositeWidgetProcessor, as with CompositeWidgetBuilder, but by a mutable list. This is important so that individual UI screens can dynamically add widget processors that, say, attach event handlers. The methods these event handlers call are, by definition, tied to a particular UI screen rather than being immutable across the entire application. If we were to implement an immutable CompositeWidgetProcessor we would be unable to add such event handler processors without invalidating the immutability of the overall composite. Equally however many other types of widget processor, such as data validators and data bindings, are not tied to a particular screen and *are* immutable. We do not want to unnecessarily instantiate multiple instances of such processors. Designing the widget processors as a mutable list, therefore, allows us the best of both approaches.

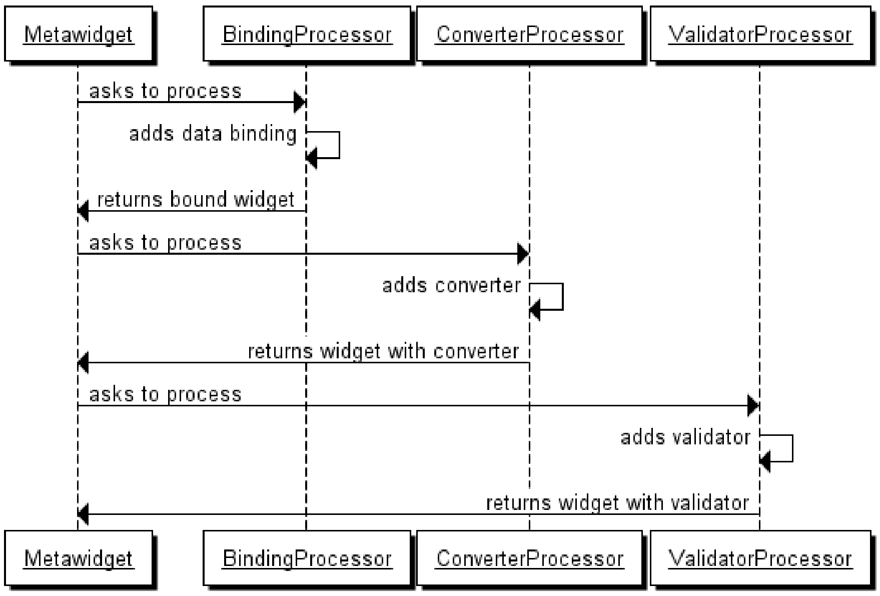


Figure 5: WidgetProcessors allow post processing of a widget

This section has demonstrated that supporting a variety of ways to post process UI widgets is an important characteristic for a practical UI generator. Following post processing, the widget is robust enough to be placed in front of the user. However, it still remains to lay it out appropriately among its siblings. This final characteristic is discussed in the next section.

3.5. Pluggable Layouts

Having inspected, built and processed the final widget its layout on the screen is perhaps the most intractable issue in UI generation. In particular, 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.

Our implementation sidesteps the issue by sharply restricting the bounds of its generation (Kennard

& Steele 2008). 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. We explicitly keep these out of scope. Our adoption studies confirmed: "I think that although it is theoretically possible to solve this [automatically], 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". Human-based, aesthetic judgements must be made as to what can fit, what can be supported, and what is most usable.

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 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. 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 (Myers, Hudson & Pausch 2000, p. 13).

Our implementation addresses this characteristic of supporting multiple ways to arrange widgets by defining pluggable 'layouts'. It defines a minimal 'Layout' interface and ships with a number of implementations of this interface to support different layouts. Notably, our implementation also supplies a LayoutDecorator, named after the decorator design pattern (Gamma et al. 1995), to support *decorating* one layout with another. For example, fields may need to be divided into sections separated by headings, or sections divided into tabs in a tab panel. Such concerns should be orthogonal to the field arrangement themselves (i.e. one column or two columns, labels to the left of the widgets or the right) to curb a combinatorial explosion of UI options, as shown in figures 6 and 7.

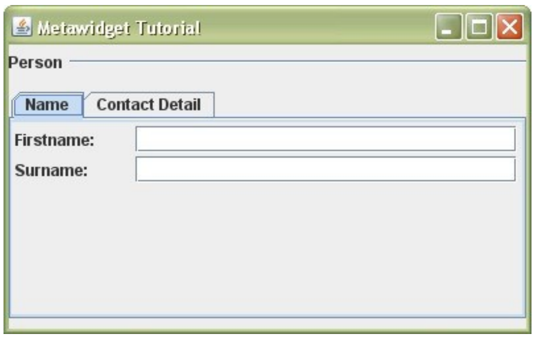


Figure 6: Layout decorated with tabs inside section headings

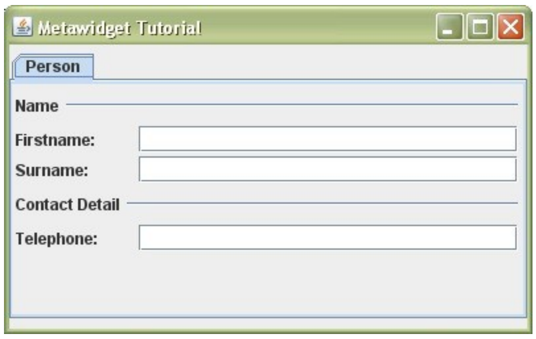


Figure 7: Layout decorated with section headings inside tabs

Adoption studies informed and agree the benefits of achieving parity between the UI designer's intent and the automatically generated UI are invaluable: “It is really hard to keep consistency, visual or functional standards when building [a] GUI in a large team. However, when it is generated dynamically, the rules are centred and even the customisation is somehow controlled”.

As with inspectors and widget builders, layouts further demonstrate emergent 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 the practitioner is accustomed to using these layout widgets, the UI generator must support them or it will not be able to achieve parity with the original UI design. In addition, leveraging layout widgets affords the inherent robustness of delegating Web browser incompatibility issues, as discussed in section 3.3.

This section has demonstrated that supporting multiple ways to arrange widgets is an important characteristic for a practical UI generator. This, combined with the other four characteristics demonstrated in previous sections, results in a richly flexible UI generator. As shown, many of these five characteristics individually exhibit emergent advantages. However, when all five are combined there is a further emergent advantage – one that appears unique in the literature. We turn to this advantage in the next section.

4. Retrofitting

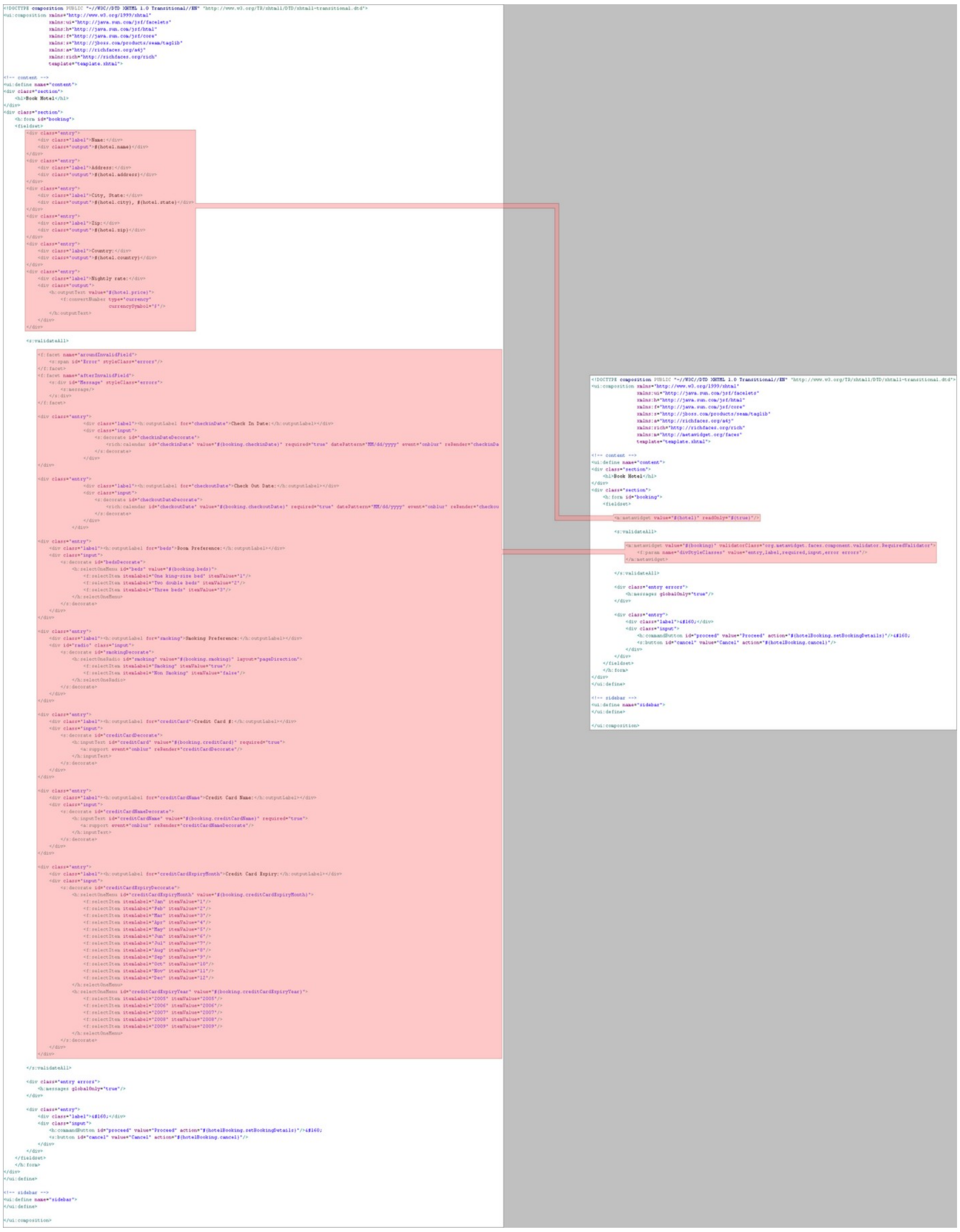
The previous section demonstrated five characteristics that are necessary for a practical UI generator. In addition, it showed how these characteristics give rise to a number of emergent advantages, which in themselves are an encouraging validation of their value. However there is a further notable advantage that emerges when all five characteristics are present in a UI generator - an advantage that is seldom even discussed, much less targeted, in the literature. Specifically, these five characteristics combined make it possible to *retrofit* UI generation into existing applications.

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 boilerplate code from *existing* applications, in addition to preventing boilerplate code 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. However several adoption studies successfully applied our implementation to existing applications. This included them being able to *partially* migrate applications one screen, or even one piece of a screen, at a time. This prompted us to conduct internal experiments retrofitting our implementation to other projects. For example, we chose three applications from the samples included with the Open Source JBoss Seam distribution (JBoss Seam 2010). We 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. We were 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. We were able to plug in widget builders for this. In particular, we were able to plug in a mixture of widget builders to replicate the application's original choice of two third-party 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, we realised between a 30% to 40% reduction in UI code through the introduction of our implementation. For some individual files this metric was as high as 70%, as shown in figure 8. On the left is the original XHTML source code for a single UI screen, and 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). 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.

This section has introduced retrofitting as an emergent advantage of the five characteristics demonstrated in the previous section. The ability to retrofit existing applications is an encouraging indicator of the overall value of the five characteristics. We now turn to this 'overall value' for our final section.



```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:s="http://jboss.com/products/seam/taglib"
                xmlns:a="http://richfaces.org/a4j"
                xmlns:rich="http://richfaces.org/rich"
                template="template.xhtml">

<!-- content -->
<ui:define name="content">
<div class="section">
    <h1>Book Hotel</h1>
</div>
<div class="section">
    <h:form id="booking">
    <fieldset>
        <div class="entry">
            <div class="label">Name:</div>
            <div class="output">#{hotel.name}</div>
        </div>
        <div class="entry">
            <div class="label">Address:</div>
            <div class="output">#{hotel.address}</div>
        </div>
        <div class="entry">
            <div class="label">City, State:</div>
            <div class="output">#{hotel.city}, #{hotel.state}</div>
        </div>
        <div class="entry">
            <div class="label">Zip:</div>
            <div class="output">#{hotel.zip}</div>
        </div>
        <div class="entry">
            <div class="label">Country:</div>
            <div class="output">#{hotel.country}</div>
        </div>
        <div class="entry">
            <div class="label">Nightly rate:</div>
            <div class="output">
                <h:outputText value="#{hotel.price}">
                    <f:convertNumber type="currency"
                                     currencySymbol="$"/>
                </h:outputText>
            </div>
        </div>

        <s:validateAll>

            <f:facet name="aroundInvalidField">
                <s:span id="Error" styleClass="errors"/>
            </f:facet>
            <f:facet name="afterInvalidField">
                <s:div id="Message" styleClass="errors">
                    <s:message/>
                </s:div>
            </f:facet>

            <div class="entry">
                        <div class="label"><h:outputLabel for="checkinDate">Check In Date:</h:outputLabel></div>
                        <div class="input">
                            <s:decorate id="checkinDateDecorate">
                                <rich:calendar id="checkinDate" value="#{booking.checkinDate}" required="true" datePattern="MM/dd/yyyy" event="onblur" reRender="checkinDa
                            </s:decorate>
                        </div>
            </div>

            <div class="entry">
                        <div class="label"><h:outputLabel for="checkoutDate">Check Out Date:</h:outputLabel></div>
                        <div class="input">
                            <s:decorate id="checkoutDateDecorate">
                                <rich:calendar id="checkoutDate" value="#{booking.checkoutDate}" required="true" datePattern="MM/dd/yyyy" event="onblur" reRender="checkou
                            </s:decorate>
                        </div>
                    </div>

            <div class="entry">
                <div class="label"><h:outputLabel for="beds">Room Preference:</h:outputLabel></div>
                <div class="input">
                    <s:decorate id="bedsDecorate">
                        <h:selectOneMenu id="beds" value="#{booking.beds}">
                            <f:selectItem itemLabel="One king-size bed" itemValue="1"/>
                            <f:selectItem itemLabel="Two double beds" itemValue="2"/>
                            <f:selectItem itemLabel="Three beds" itemValue="3"/>
                        </h:selectOneMenu>
                    </s:decorate>
                </div>
            </div>

            <div class="entry">
                <div class="label"><h:outputLabel for="smoking">Smoking Preference:</h:outputLabel></div>
                <div id="radio" class="input">
                    <s:decorate id="smokingDecorate">
                        <h:selectOneRadio id="smoking" value="#{booking.smoking}" layout="pageDirection">
                            <f:selectItem itemLabel="Smoking" itemValue="true"/>
                            <f:selectItem itemLabel="Non Smoking" itemValue="false"/>
                        </h:selectOneRadio>
                    </s:decorate>
                </div>
            </div>

            <div class="entry">
                <div class="label"><h:outputLabel for="creditCard">Credit Card #:</h:outputLabel></div>
                <div class="input">
                    <s:decorate id="creditCardDecorate">
                        <h:inputText id="creditCard" value="#{booking.creditCard}" required="true">
                           <a:support event="onblur" reRender="creditCardDecorate"/>
                        </h:inputText>
                    </s:decorate>
                </div>
            </div>

            <div class="entry">
                <div class="label"><h:outputLabel for="creditCardName">Credit Card Name:</h:outputLabel></div>
                <div class="input">
                    <s:decorate id="creditCardNameDecorate">
                        <h:inputText id="creditCardName" value="#{booking.creditCardName}" required="true">
                           <a:support event="onblur" reRender="creditCardNameDecorate"/>
                        </h:inputText>
                    </s:decorate>
                </div>
            </div>

            <div class="entry">
                <div class="label"><h:outputLabel for="creditCardExpiryMonth">Credit Card Expiry:</h:outputLabel></div>
                <div class="input">
                    <s:decorate id="creditCardExpiryDecorate">
                        <h:selectOneMenu id="creditCardExpiryMonth" value="#{booking.creditCardExpiryMonth}">
                            <f:selectItem itemLabel="Jan" itemValue="1"/>
                            <f:selectItem itemLabel="Feb" itemValue="2"/>
                            <f:selectItem itemLabel="Mar" itemValue="3"/>
                            <f:selectItem itemLabel="Apr" itemValue="4"/>
                            <f:selectItem itemLabel="May" itemValue="5"/>
                            <f:selectItem itemLabel="Jun" itemValue="6"/>
                            <f:selectItem itemLabel="Jul" itemValue="7"/>
                            <f:selectItem itemLabel="Aug" itemValue="8"/>
                            <f:selectItem itemLabel="Sep" itemValue="9"/>
                            <f:selectItem itemLabel="Oct" itemValue="10"/>
                            <f:selectItem itemLabel="Nov" itemValue="11"/>
                            <f:selectItem itemLabel="Dec" itemValue="12"/>
                        </h:selectOneMenu>
                        <h:selectOneMenu id="creditCardExpiryYear" value="#{booking.creditCardExpiryYear}">
                            <f:selectItem itemLabel="2005" itemValue="2005"/>
                            <f:selectItem itemLabel="2006" itemValue="2006"/>
                            <f:selectItem itemLabel="2007" itemValue="2007"/>
                            <f:selectItem itemLabel="2008" itemValue="2008"/>
                            <f:selectItem itemLabel="2009" itemValue="2009"/>
                        </h:selectOneMenu>
                    </s:decorate>
                </div>
            </div>

        </s:validateAll>

        <div class="entry errors">
            <h:messages globalOnly="true"/>
        </div>

        <div class="entry">
            <div class="label">&#160;</div>
            <div class="input">
                <h:commandButton id="proceed" value="Proceed" action="#{hotelBooking.setBookingDetails}"/>&#160;
                <s:button id="cancel" value="Cancel" action="#{hotelBooking.cancel}"/>
            </div>
        </div>
    </fieldset>
    </h:form>
</div>
</ui:define>

<!-- sidebar -->
<ui:define name="sidebar">
</ui:define>

</ui:composition>
```

```
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core"
                xmlns:s="http://jboss.com/products/seam/taglib"
                xmlns:a="http://richfaces.org/a4j"
                xmlns:rich="http://richfaces.org/rich"
                xmlns:m="http://metawidget.org/faces"
                template="template.xhtml">

<!-- content -->
<ui:define name="content">
<div class="section">
    <h1>Book Hotel</h1>
</div>
<div class="section">
    <h:form id="booking">
    <fieldset>

        <m:metawidget value="#{hotel}" readOnly="#{true}"/>

        <s:validateAll>

            <m:metawidget value="#{booking}" validatorClass="org.metawidget.faces.component.validator.RequiredValidator">
                <f:param name="divStyleClasses" value="entry,label,required,input,error errors"/>
            </m:metawidget>

        </s:validateAll>

        <div class="entry errors">
            <h:messages globalOnly="true"/>
        </div>

        <div class="entry">
            <div class="label">&#160;</div>
            <div class="input">
                <h:commandButton id="proceed" value="Proceed" action="#{hotelBooking.setBookingDetails}"/>&#160;
                <s:button id="cancel" value="Cancel" action="#{hotelBooking.cancel}"/>
            </div>
        </div>
    </fieldset>
    </h:form>
</div>
</ui:define>

<!-- sidebar -->
<ui:define name="sidebar">
</ui:define>

</ui:composition>
```

Figure 8: Portions of code saved by retrofitting

5. Summary, Conclusions and Future Work

This article has emphasised the need for UI generation within mainstream software development, and underscored impracticalities with existing approaches. It has employed practice based research and human centred design approaches to identify five characteristics that are necessary for a practical UI generator. Interviews, adoption studies and industry collaboration suggest these characteristics are key to the real world adoption of a generator. These characteristics can be summarised as: inspecting existing, heterogeneous back-end architectures; appreciating different practices in applying inspection results; recognising multiple, and mixtures of, UI widget libraries; supporting multiple, and mixtures of, UI adornments; applying multiple, and mixtures of, UI layouts. Many of these characteristics seem ignored by current UI generators and we believe this is hindering their widespread adoption. This article has further identified the impact of implementing all five characteristics: they allow practitioners to retrofit UI generation into existing applications. This significantly broadens the range of applications amenable to UI generation and increases its value to practitioners.

This article has drawn several far reaching conclusions. First, that any UI generator that seeks to dictate, rather than adapt to, a system's architecture has limited practical value. Second, it is important to appreciate that the need for diversity in industry architectures is not because most business systems are poorly designed. Rather, it is because business systems seek to leverage functionality provided by a large number of mature subsystems available in industry, in order to increase productivity and reduce development cost. Finally this article has emphasised that action research, as a methodology for research and development, is fundamental to the usefulness of a resulting product in supporting diversity and richness of practice.

Immediate future work will concentrate on further developing our UI generator implementation. There are many design decisions for which the five characteristics identified in this article could be implemented either way. For example, there are opposing schools of thought whether generation should be runtime based or statically generated. We also plan to continue evaluating our implementation against real world business systems, to assess the degree to which those systems can be retrofitted to reduce UI code. A longer term goal is to standardise the five UI generation characteristics, such that they can be adopted as part of mainstream UI development.

References

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

Cao, L., Yu, P.S., Zhang, C. & Zhao, Y., 2010. Domain Driven Data Mining, Springer-Verlag New York Inc.

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., 2000. 'A beginner's guide to action research', *http://www.scu.edu.au/schools/gcm/ar/arp/guide.html*

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

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

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

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

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

Haywood D., 2008.1. Comments section under http://kennardconsulting.blogspot.com/2008/04/useful-bounds-of-automatic-ui.html

Haywood D., 2008.2. Comments section under http://kennardconsulting.blogspot.com/2008/07/metawidget-javassist-versus.html

Haywood D., 2009. Comments section under http://www.theserverside.com/news/thread.tss?thread_id=58283#328193

JBoss Seam, 2010. http://www.seamframework.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.

JPA 2008. http://jcp.org/en/jsr/detail?id=220.

Kagdi, H., Collard, M.L. & Maletic, J.I., 2007. 'Comparing Approaches to Mining Source Code for Call-Usage Patterns', *Proceedings of the Fourth International Workshop on Mining Software Repositories*.

Kemmis, S. & McTaggart, R., 1988. 'The action research planner'. Deakin University.

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', *New Trends in Software Methodologies, Tools and Techniques*, p. 244.

Larman, C. & Basili, V. R., 2003. 'Iterative and Incremental Development: A Brief History', *Computer*, pp. 47-56.

Li, X., Zhang, S. & Wang, S. 2006, 'IJDWM Special Issue: Advances in Data Mining Applications', International Journal of Data Warehousing and Mining, vol. 2, no. 3.

Manheimer, J.M., Burnett, R.C. & Wallers, J.A., 1989. A case study of user interface management system development and application. Proceedings of the SIGCHI conference on Human factors in computing systems: Wings for the mind , 127-132.

Meskens, J., Vermeulen, J., Luyten, K. & Coninx, K., 2008. 'Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me', ACM, pp. 233-240.

Myers, B.A., 1995. 'User Interface Software Tools', *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 1, pp. 64-103.

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.

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

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

Prat, A., Lores, J., Fletcher, P. & Catot, J.M., 1990. Back-End Manager: An Interface between a Knowledge-based Front End and its Application Subsystems. Knowledge-Based Systems, vol. 3, no. 4.

Rouvellou, I., Degenaro, L., Rasmus, K., Ehnebuske, D. and Mc Kee, B., 1999. Externalizing Business Rules from Enterprise Applications: An Experience Report. Practitioner Reports in OOPSLA, vol. 99.

Schön, D.A., 1983. The Reflective Practitioner: How Professionals Think in Action, Basic Books.

Shan, T.C., Hua, W.W., Bank, W. & Wilmington, N.C., 2006. 'Taxonomy of java web application frameworks', pp. 378-385.

Square 2010. http://www.squareup.com

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, 19-20.

Vermeulen, A., Beged-Dov, G. & Thompson, P. 1995, 'The pipeline design pattern', OOPSLA ’95 Workshop on Design

WSDL 2001. http://www.w3.org/TR/wsdl

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

Xie, T., Pei, J. & Hassan, A.E., 2007. 'Mining Software Engineering Data', *International Conference on Software Engineering*, pp. 172-173.

XML 2008. http://www.w3.org/TR/REC-xml

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.