Metawidget White Paper

Adoption Studies

Richard Kennard

July 2011

http://metawidget.org

1. Introduction

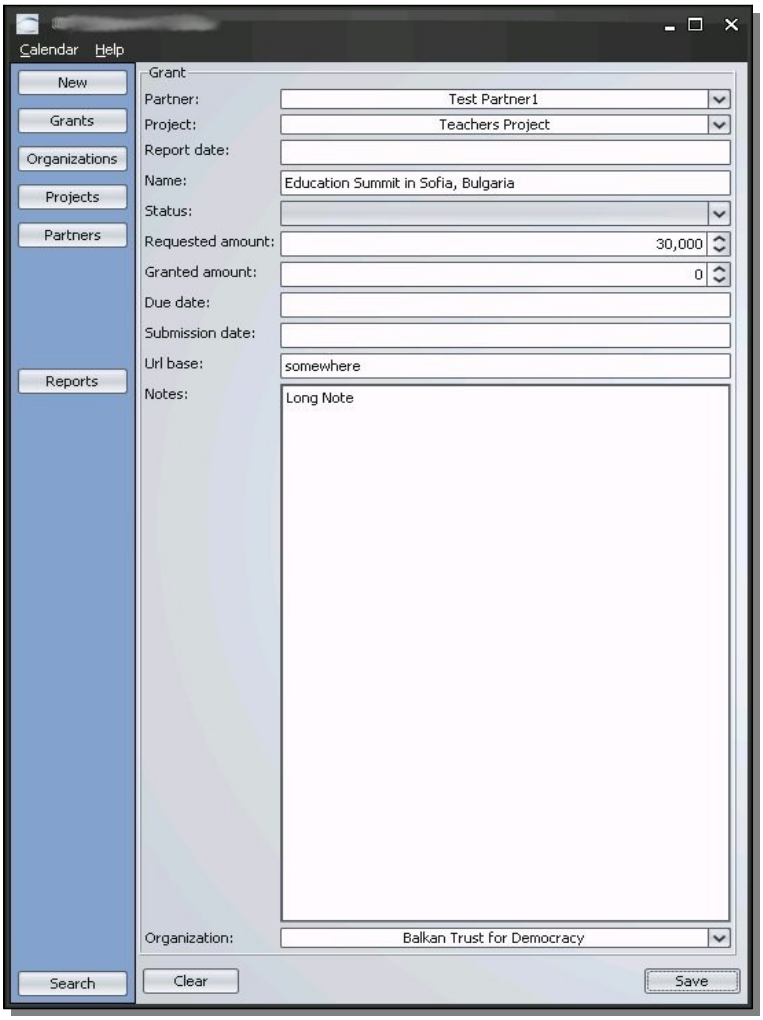
This white paper presents a collection of adoption studies from companies using Metawidget. Adoption studies are interviews focused on adoption of a technology within an interviewee's organisation. The goal is to draw out all experiences and to understand usage and the environment in which the adoption occurred. These learnings are then followed by a period of reflection to assess strengths and weaknesses. The findings are used to identify ways to improve Metawidget.

2. Adoption Study 1

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

Application: Internal, desktop application

Technologies: Java, Swing, JPA



2.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

The lead developer first described how, after being assigned a new project, he started thinking along the same lines as Metawidget. “I perceived the need, then a few days later as I was thinking of how to create a smaller scale version of what Metawidget is (due to client time constraints) I found Metawidget on [an industry Web site] and couldn't believe my luck – it fit the bill perfectly”. Metawidget is promoted on industry Web sites after each new release, as a means to gain exposure. A typical promotion would be a short write-up of the features included in the new version, with links for finding out more information, such as to

user manuals or blog entries. Regarding the need the lead developer had perceived, had he used products that addressed this need before? “Nothing that has to do with the user end. But of course, [the ORM] Hibernate comes so close, and it goes the other way, toward the DB layer”.

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

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

2.2. Reflection

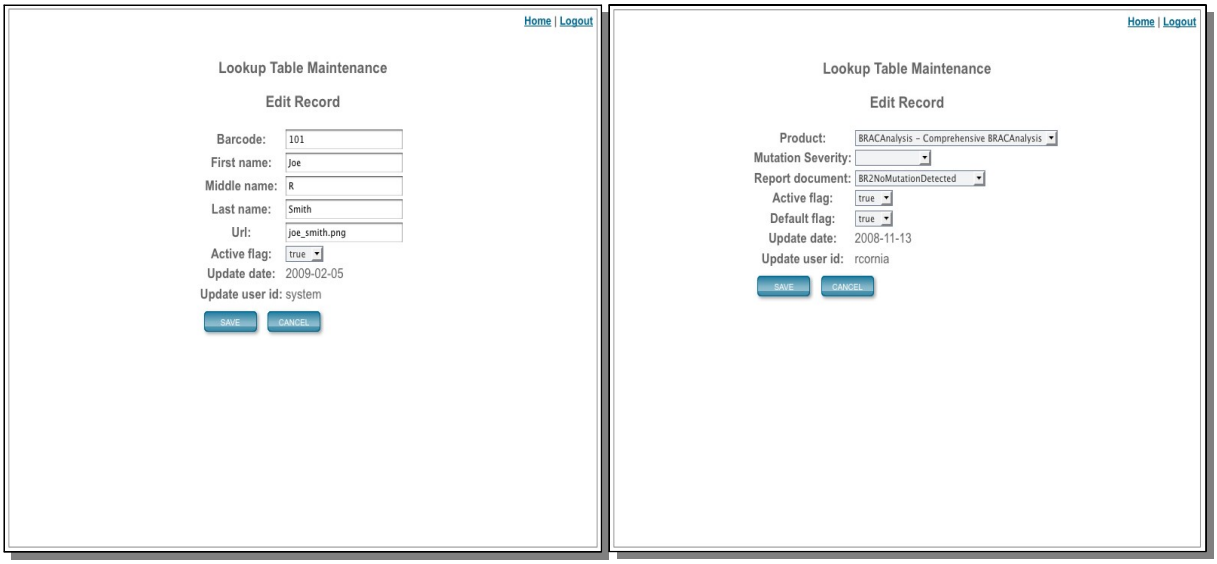
This adoption study considered a project that was being newly architected, but the developer had already chosen his front-end (Swing) and back-end (Hibernate) before considering a UI generator. Therefore, it would not have suited him had the UI generator attempted to dictate the architecture.

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

Finally, whilst the developer understood the problem and liked the solution, he felt that in order to address it in the wider community it needed to become a “hot topic” – it was important to get people to try Metawidget. This emphasises the necessity of well-designed, accessible resources: screenshots, demos, product comparisons, testimonials and a variety of other promotional materials.

3. Adoption Study 2

Industry: U.S.-based biopharmaceutical company specialising in molecular diagnostics
Application: Web application
Technologies: Java, Spring, JPA



3.1. Synopsis

This adoption study was conducted by interviewing the team lead.

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

This time the team decided on a different approach. Building their own framework “did not add any business specific value if we could find a third-party solution that solved the same problem”. What type of third-party solution were they looking for? “I would say we were looking for ease of use, yet flexible; something that required minimal code to get the job done; cross UI was important but ultimately would not have been the

single driver”. Whilst being cross UI is an emergent property of Metawidget's objective, it is not a primary one. Rather, being 'UI agnostic' is. Was this important? “We needed to integrate it with a Spring MVC app, and in the future we may want to integrate with some existing Swing applications… also possibly Java Server Faces (JSF)”.

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

Returning to the overall theme, is this a prevalent problem? “Absolutely. In 10 years of software development, I can't count the number of times I've needed a simple form for users to enter or update data. I think it is a problem that has likely been 'solved' by many in their own specific companies, but no one has extended that in a general way to apply to a broad audience. Certainly there have been 'form code generators', but creating the form at runtime from metadata is a far more elegant approach in my opinion”.

3.2. Reflection

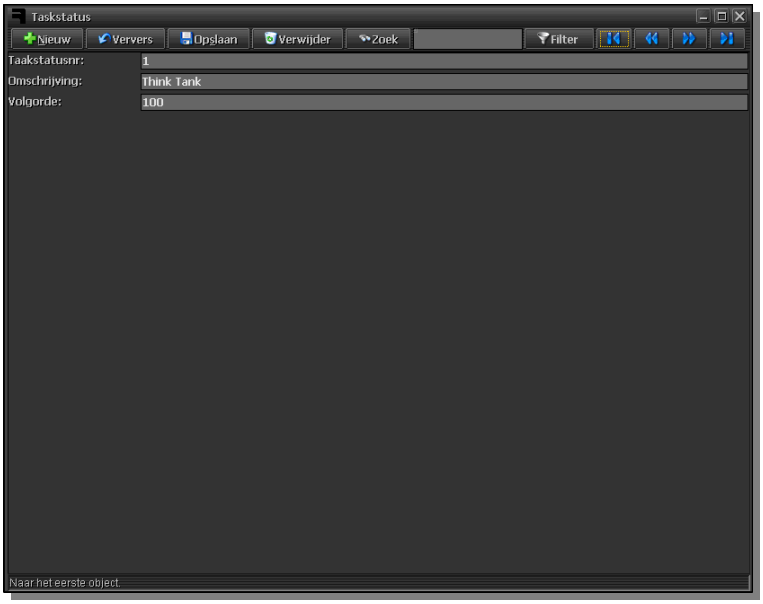
This adoption study underscored that application architectures change from project to project. The team had moved from building Swing applications to Spring MVC ones, and were considering JSF in the future. Whilst being cross UI was not critical to them for any one project, being UI agnostic across multiple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

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

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

4. Adoption Study 3

Industry: Enterprise Resource Planning
Application: Desktop application
Technologies: Java, Swing



4.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

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

How did the introduction go? “It did seem complicated. I understand the need to support multiple platforms, but maybe it is wise to provide some sort of pre-set access points, if you understand what I mean”. Was the switch ultimately successful? “[No.] I found Metawidget to be somewhat too generic… very often I decided after a while to still build a specific panel instead. It is not that much work and the screen is just that little bit more tuned”. Could this be improved by augmenting the business model with additional UI information? “Naturally I can add a lot of UI information to the business model, but I'm a strong believer in layers (for as

long a possible) and UI information simply does not belong in a business model". As an alternative, Metawidget allows the developer to keep UI information in separate metadata files, but doing so does not solve the issue of duplication – fields must be declared and maintained in both the business model and the metadata files.

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

There were additional reasons Metawidget did not succeed: "if you have a more exotic UI component used for certain properties (JCalendar?), there is more work needed to get that to render, as opposed to simply creating the component in your UI layer". Also: "the additional information required to get the layout in Metawidget right, competes with the amount of code needed for a custom panel… I use JGoodies binding, MigLayout and some utility classes, so creating a form consists of 2 lines and adding a field is 4 lines: create the component, bind it to the property, place it on the panel. Simple, minimal lines of code, understandable". Although simpler, is it less maintainable managing duplication between the domain objects and the UI layer? "That is initially the biggest advantage of Metawidget; it automatically updates the UI. But if you just let it do its thing, after adding the field, then it is dumped somewhere on the screen… [to reposition it] you need additional UI information… I may not even want [the field] to display. Changes to the business model layer may automatically have unwanted consequences for the layers above". As a suggestion: "can't we have a `SortingInspector` which I provide with an array of property names, and those names come first in their array order, while the rest is appended alphabetically?". Overall: "I find the level of abstraction… not sufficiently rewarding above simply coding it out. For fat clients I believe the generic layout is not the quality of screens that people expect. The finer details get in the way. For [thin clients] this is less of an issue".

4.2. Reflection

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

First, Metawidget's flexibility can make it seem complicated at first. A more considered approach with some

sensible defaults may smooth the initial experience[1].

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

Third, developers who choose to keep UI metadata in separate metadata files, as opposed to augmenting the business model, do not see as much benefit from Metawidget because they still have to maintain duplication between the metadata files and the business model. This situation could be improved. For example Metawidget could validate the metadata files in the same way many persistence solutions do[2].

Fourth, whilst Metawidget does support third-party UI components, currently this requires more work on the developer's part than necessary. Metawidget should improve its third-party component support – especially, on reflection, mixing third-party component libraries in the same project. If the developer wishes to use, say, the third-party JCalendar (as a date picker) that should not preclude using the third-party JFreeChart (for charting). This should also extend to any custom components the developer may have created[3].

Fifth, there was a desire for a different mechanism to sorting fields than the default annotation-based one (the developer annotates each field with a `@UiComesAfter` annotation). Specifically, a request to be able to "provide an array of property names" to an inspector. This requirement sits uncomfortably within Metawidget's architecture, because presumably the array of names must be specified per UI screen yet currently inspectors do not have any direct connection to the screen. This is an important design decision because some inspectors are designed to run remotely on different application tiers, where there is no screen available. Indeed some inspectors need to run on heterogeneous platforms to their UI widget. For example a Java-based back-end inspector may return information to an ECMAScript-based front-end widget[4].

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

---

1 *This reflection, and others like it, led to the introduction of pre-configured Metawidgets.*

2 *This reflection, and others like it, led to the introduction of* `XmlInspectorConfig.setRestrictAgainstObject`

3 *This reflection, and others like it, led to the introduction of widget builders.*

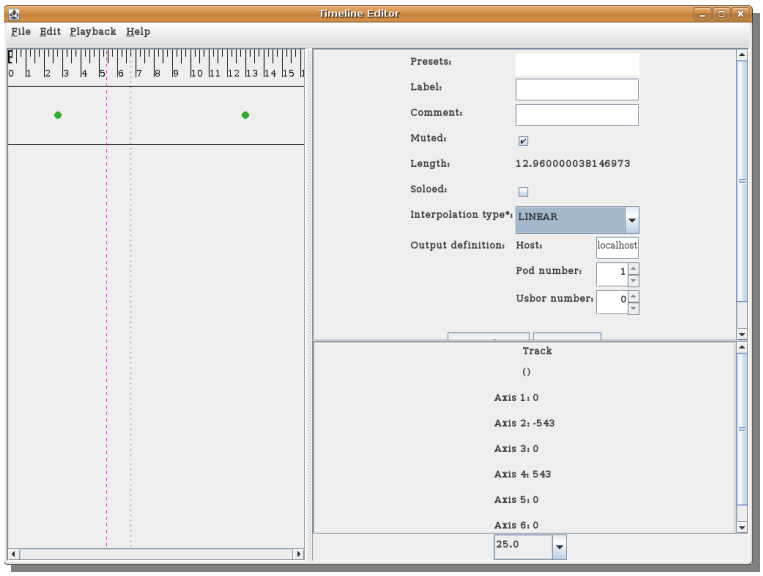
4 *This reflection, and others like it, led to the introduction of inspection result processors.*

5. Adoption Study 4

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

Application: Desktop application

Technologies: Java, Swing



5.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

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

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

important? “Sort of. [This was a new project so] the API was written with Metawidget in mind, but if I'd had to do it differently because of Metawidget I would have been unhappy. As things are I was just able to treat it as a normal Swing widget with was nice”.

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

Finally, the discussion turned to the overall usefulness of Metawidget. “I don't do enough front-end coding to speak on the overall usefulness of this type of tool in general. However I'm not very good at UI development (and I don't enjoy it), so I find it very useful to be able to say 'I want a UI for this bean' and have one generated automatically without any work on my part”. So Metawidget saved you writing code? “I think Metawidget's binding implementations are critical. Without them you still have to write [UI helper code] for every class that touches the widget for every property (to fill in the values). My main use case was allowing plug-ins to add data classes that the user can interact with without [the developer] needing to do any UI coding”.

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

5.2. Reflection

This adoption study contained a further reinforcement of the 'do not dictate the architecture' tenet. The

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

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

This adoption study also highlighted areas in need of improvement. The developer found Metawidget's existing methods for ordering widgets too awkward in scenarios involving inheritance across a class hierarchy. The shortcomings were twofold: the XML-based metadata was implicitly ordered but required explicit declaration of the class hierarchy[5]; the annotation-based metadata implicitly followed the class hierarchy, but was not based on priorities[6]. This last point was another reminder that different developers prefer different techniques: "it's just more natural to me".

Another negative comment was that the developer had difficulty attaching event handlers to generated widgets. This stems from a general observation that whatever is automatically generated becomes much more opaque to, and less controllable by, the developer. There is an impedance mismatch between the API the generator exposes and the native API of the target platform. Metawidget has this to a lesser degree, because it 'owns' the UI to a lesser extent, but more work should be done to reduce this mismatch where possible[7].

---

5 *This reflection, and others like it, led to the introduction of* `XmlInspectorConfig.setRestrictAgainstObject`

6 *This reflection, and others like it, led to the introduction of inspection result processors.*

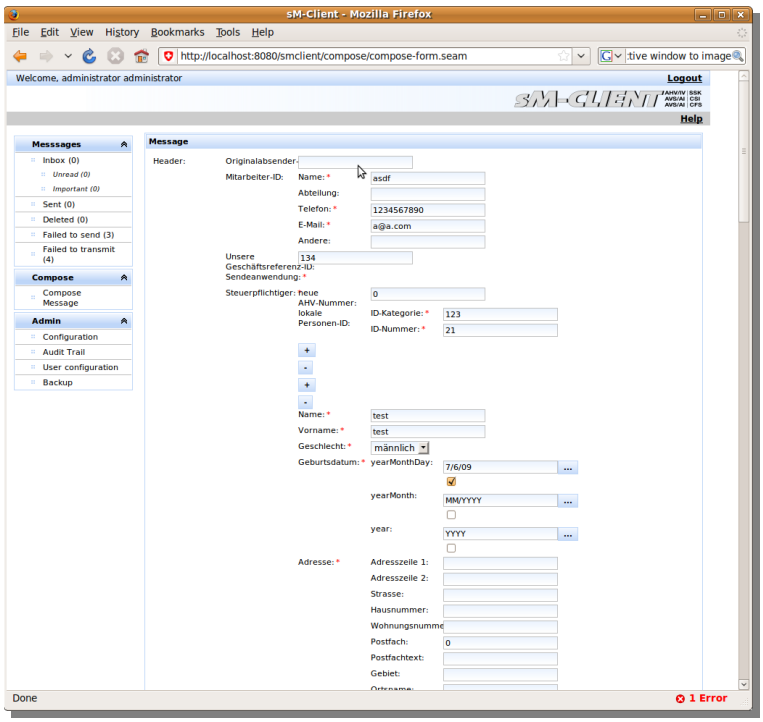
7 *This reflection, and others like it, led to the introduction of widget processors.*

6. Adoption Study 5

Industry: Swiss Government

Application: Web application

Technologies: JSF, Facelets, JAXB



6.1. Synopsis

This adoption study was conducted by interviewing the lead developer.

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

I began by discussing what led the developer to find Metawidget: “I didn't find it, a collage of mine did. He knew our project and I think he actively went and looked for something we could use. We wanted something that we could use to change our XML [messages] into a [UI] form”. The developer explained there were many types of XML message, and the prospect of developing (and maintaining) UI representations of each was onerous. The team considered several choices: “things I've also looked at include XML-Forms and writing custom XSL stylesheets [to convert the XML into HTML]. Metawidget is a way better solution than the other options”.

What were some of Metawidget's strengths? From a front-end perspective “[being] able to integrate our own

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

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

6.2. Reflection

This adoption study was notable for the amount of customisation the developer required. He needed to be able to plug-in both his own project-specific front-end ("own validation and custom rendering") and his own project-specific back-end ("inspector that knows our XML schema"). Such adaptability to different architectures was critical, in fact he would have liked it to go further and be "even more pluggable"[8].

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

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

Unlike Metawidget, JAXB doesn't support extracting this metadata from arbitrary sources. It is interesting to consider, though outside our current scope, the contribution Metawidget could make to the Object to XML mapping domain. Metawidget's architecture already promotes a strong separation between its inspectors and any UI-related code. It may be possible to reuse its existing inspectors for other purposes such as this.

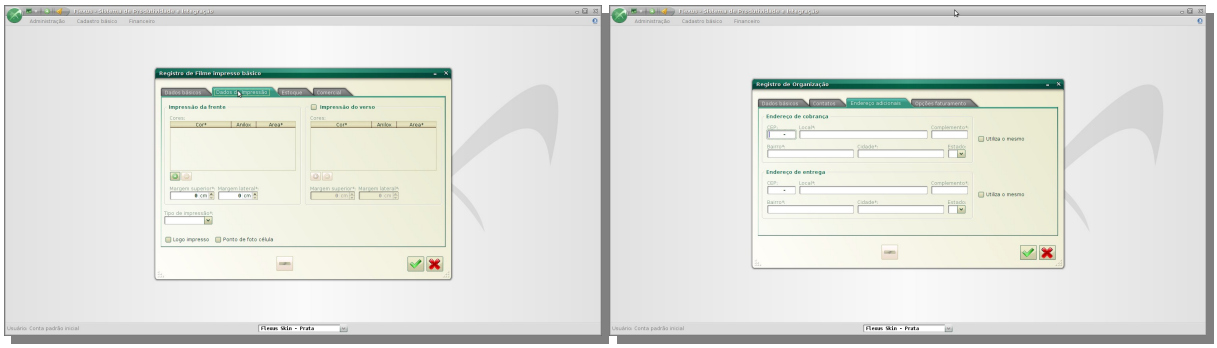
8 *This reflection, and others like it, led to the introduction of inspection result processors.*

7. Adoption Study 6

Industry: Brazilian packing industry

Application: Specialised Enterprise Resource Planning (ERP)

Technologies: Swing, OVal



7.1. Synopsis

This adoption study interviewed a senior developer on the team.

As with all adoption studies, I began by discussing how the team encountered Metawidget. “We were working on another project where we started doing some dynamic UI generation… much more simple stuff, like printing the required fields and formatting the numbers based on the model… and then we read an announcement on [an industry Web site] and so I bookmarked it to take a look for the next project”. When their next project began, the team “started from scratch with Metawidget in mind”.

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

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

such as ORMs, have the same problem. “It would be nice to either solve this or offer a solution for that (maybe tooling?)”.

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

Overall, the team found they were able to apply Metawidget widely: “all [our] UIs [screens] have Metawidget behind [them], even the complex and most important. We wrote our own layout and widget builder. An interesting thing to notice is that in [second screenshot at start of adoption study], both tables are created by Metawidget. They are [custom] components we developed and Metawidget instantiates them. The whole screen is generated dynamically by Metawidget”. The developer then reflected on the technology as a whole: “I think that there are two main problems that Metawidget helps to solve: Independence of View Technology and Dynamic UI Generation, both [these problems are] current and prevalent. Regarding the first one, I think that although it is theoretically possible to solve this [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”. The developer approved of the way Metawidget does not try to 'own' the UI. Rather it positions itself as just a piece of the larger UI landscape, recognising the uniqueness of different platforms and devices.

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

“Besides that, for simple input interfaces, or prototyping, it is simply amazing. You have to do nothing and

you have a fully functional UI. [In production] we have really complex UIs, since we have a strong usability concern. Another approach that tries to solve this problem is using [statically generated] MDA tools. I personally dislike this solution, mainly because of the idea that I find really important: 'everything that you create (generate) you have to maintain'. I did not find solution with a decent support for maintaining the generated code, and a lot of garbage code is generated".

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

7.2. Reflection

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

Despite this positive outcome, there were still further areas to be addressed. Some were straightforward enhancements "more powerful layouts ready"[9] but others were more intractable. In particular, being able to refer to generated widgets in cases where the developer doesn't know their name in advance was a "problem not only for Metawidget... [but] it would be nice to either solve this or offer a solution for that"[10].

---

9 *This reflection, and others like it, led to the introduction of layout decorators.*

10 *This reflection, and others like it, led to the introduction of widget processors.*

8. Conclusion

This white paper has considered a variety of adoption studies across different domains and organisations. Each study has been followed by a period of reflection, which attempted to objectively assess strengths and weaknesses of Metawidget and identify areas for improvement. It is hoped that by maintaining close collaboration with developers through studies such as this, Metawidget can continue to improve and fulfil its goal of becoming a general purpose automatic UI generator.