Metawidget White Paper

Duplication in User Interfaces

Richard Kennard

November 2010

http://metawidget.org

1. Introduction

This white paper examines the issue of duplication in User Interfaces (UIs). It investigates whether 'UI duplication' (that is, restating information between the UI and the business layer) is considered a problem by software developers: do they see duplication as a prevalent and detrimental source of errors, or does it provide an advantageous level of flexibility? To answer this, we conducted 6 interviews with senior software developers chosen from different segments of industry - including finance, medical and middleware - across the UK, the US and Australia.

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

2. Duplication

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

All the practitioners gave similar answers for the process. One enumerated “first off we would add [the Date of Birth field] to the database, in the table. We'd then add it to the stored procedures going up. Add it into the Data Access Layer for the purposes of getting it out of the recordset. And then you'd add the property into the business level, the business layer. And then, on the UI, on the front-end, we'd have to add the field in the HTML”. Another practitioner said “I would go to the persistence level, I'd work out how that field should be modelled in the problem domain. For date of birth, you'd have a date column. I'd look at the `Person` class, work out its relationship with the `Person` schema. Work out its name, what its type would be, `date` or `datetime` depending on the database. Then I'd work out how I should change the `Person` class – there'd probably just be a getter and setter – and then I'd tie it back to the persistence layer, map it back to the table. For validation constraints, yeah, this is always a problem, you need to validate it both in the UI and at the persistence layer if that's a business rule, so it's always a problem. In terms of the UI, I'd go and find the bit of UI code and work out the position where this field should be added”.

It was noted those practitioners using newer technologies had considerably fewer steps. One said “we would

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

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

Following the probe question, all interviewees converged on recognising duplication amongst the steps. "The problem definitely exists. It's more from the business layer forward to the screen is the biggest problem because there are things out there like Hibernate (Hibernate 2009) which do from, sort of, business layer down". Another echoed this sentiment "the drudgery at the moment is adding the UI code, and adding the validation and giving that feedback. That's really quite unpleasant. It's the most complex of all the steps, actually, depending on the magnitude of the change. Given a very simple change, just adding a single field, the bulk of the work, the bulk of the drudgery, in the coding is at the UI level. Being able to more concisely express the relationship between the UI and the model and the change I want to make in one place, or at most two places, in a very concise fashion would help". Another warned "it's a fairly established software engineering principle that the more you have to repeat something the higher the error is, the higher the chances there's going to be an error in the code".

3. Defects

Following on from this, we asked each practitioner whether they had ever encountered defects that were a result of this recognised deficiency in their process. All of the interviewees responded that such defects were common. "Definitely. There's always a chance that someone's going to get a bug somewhere along the line, especially with Date of Birth – as it goes down the date gets mixed up because someone's used the incorrect data type. With some of our junior developers we have here that's quite a common thing where they get a bit muddled up… it's definitely an issue that should be far simpler". Another agreed "All the time. That would

be me overlooking various aspects of the user feedback loop, in the validation, me forgetting to persist various fields that I've added, so the validation happens but then it never gets persisted, so having to tie the new field to the model, with validation, in multiple places, gives a number of points where I could fail to do that". Another said, of reviewing other developer's code, "a large percentage of mistakes were always they'd copy and pasted [another field] and they'd changed that [declaration], and that one, and that one – but not that one. So it creates a higher chance of there being a minor error".

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

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

4. Prevalence

Finally, we asked each practitioner whether the themes explored in the interview were commonplace across all software systems they had developed. One said "I've built a number of UIs over the course of my career, some of them have been desktop applications, some of them have been Web applications, and I think this is a general problem. For desktop applications it's hard but it's relatively easy. For Web I think it becomes a lot more difficult because the technologies involved are a lot more fiddly, there are a lot more moving parts in Web application UIs. But yes I think it's a general problem.". Another said "quite honestly laying out UI forms is time consuming, it's fairly standard how a UI is – it shouldn't be a problem to say, okay, you have these things you probably want to interface in a particular way, here's what we suggest – we being the computer – you've got a `datetime` here, here's the calendar control we suggest. Oh you don't want a calendar, you want to use a text box, go for it. Something along those lines would definitely detract from the tedium of putting together the UI, which is an important step and everything but is a really repetitive process.

If it's a `varchar` in the database, it's going to manifest as some form of a text box on the form. If I've got a foreign key in my database, it's going to manifest as some form of list box, dropdown, radio button, check box. It's not a huge leap". One practitioner summarised it as "every developer who writes anything more than a Hello World application will have this problem. Most developers who strive to make their work better, who aren't lazy, do sense this problem, do encounter this problem on a daily basis as a constant friction in their daily work".

We observed a sub-category (Dick 2005) that this friction had driven several practitioners to fashion their own ad hoc solutions by combining existing tools. "For a brand new screen we're currently using CodeSmith (CodeSmith 2009), so if you design the database table you can hit generate and it'll go through and generate everything right up to the screen". However because of subsequent editing of the generated code, they found CodeSmith to be of limited use outside of new screens: "if you could do the same thing where you could add a new field to the database and it generated and added it into the [existing] code for you as it goes up that'd be excellent". Other solutions had similar shortcomings. Microsoft LinQ (LinQ 2009) helped with the persistence layer, but "if I go in and create a field, LinQ creates a nullable version of that field, where the [UI] control I'm binding it to is expecting a non-nullable version. That's caused a number of problems. That's come up a number of times and you've really got to kind of juggle to make it work right. Keep in mind when that could happen and keep track of the potential for it to happen". Asked why they had invested the considerable resources to fashion their own solutions: "I do genuinely believe that kind of thing makes the development cycle better in the long run. It makes things much cleaner, there's less coding to go on. If I were to have to write, well, in my application the basic objects I have, I have patients, contacts, appointments, items, invoice, payments, refunds, credits and then a load of secondary objects like appointment status', patient categories, all of these are objects. If I had to code a separate form for each one it's just tedious. Interface work is not that much fun. It's quite tedious, dropping controls on a form, lining them up with the other controls and fiddling around for ages". Another practitioner echoed this sentiment saying, if such tedium could be reduced, "you'd have more time for the actual problem solving: defining, clarifying, implementing the problem rather than the mechanics of the 'auto pilot' of gotta code up this method, gotta code this, gotta code that. Give you more time to concentrate on the more energy-requiring things rather than the monotonous reproducing of stuff. Because, I mean, despite the fact they tell everyone not to, normally you end up copying and pasting things".

5. Conclusion

The results of our interviews suggested UI duplication was indeed a prevalent and serious problem in software development. We observed practitioners across industry segments and across software platforms, and saw a common theme of duplication. We also observed common themes of defects caused by duplication, the prevalence of duplication, how newer technologies only addressed duplication 'down' the stack, a tendency of practitioners to fashion ad hoc solutions, and a common desire for duplication to be

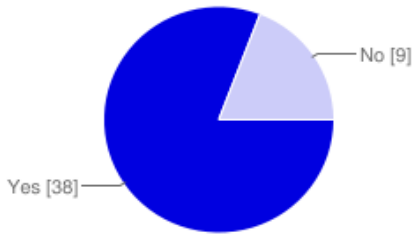
addressed.

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

The results from the self-administered survey (summarised on the following two pages) correlated well with the results from our interviews. The majority of respondents were experiencing symptoms of UI duplication. This provided further evidence of the prevalence and severity of UI duplication in software development.

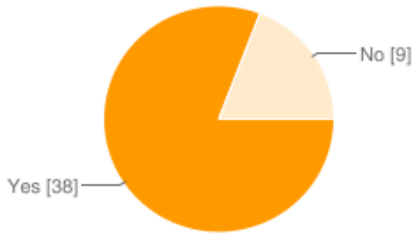
In your app, if you were to add a new field to the existing back-end (e.g. by specifying its name/type, maximum length, whether it is nullable etc.) you would also need to:

1. Manually add a label to your UI



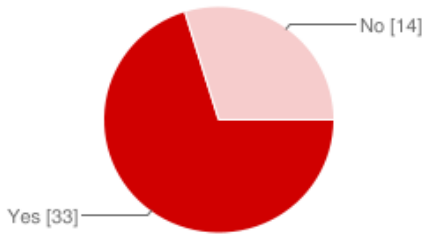
Yes	**38**	81%
No	**9**	19%

2. Manually add a widget to your UI



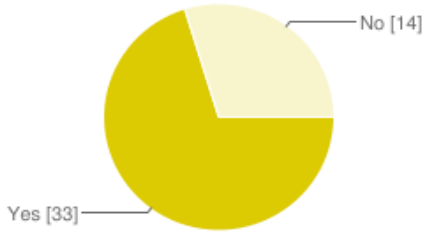
Yes	**38**	81%
No	**9**	19%

3. Manually apply constraints to the widget



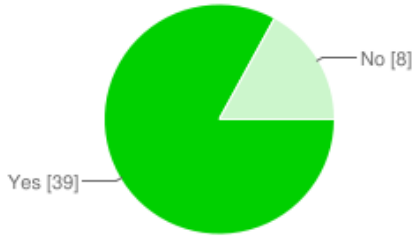
Yes	**33**	70%
No	**14**	30%

4. Manually indicate whether it's a required field



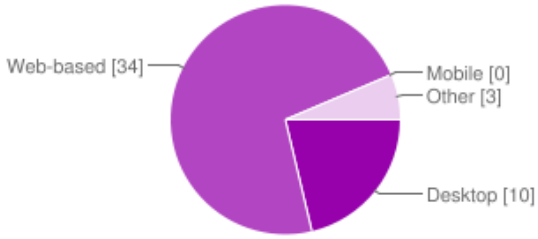
Yes	**33**	70%
No	**14**	30%

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



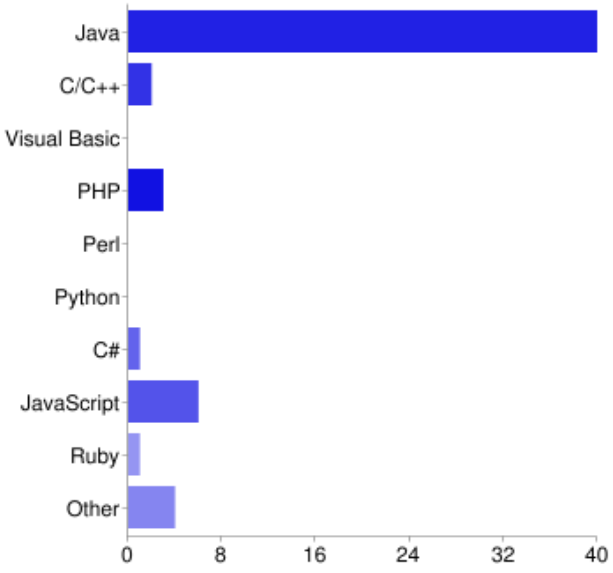
Yes	**39**	83%
No	**8**	17%

6. What type of app is it?



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

7. What platform is the app built on?



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

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

6. References

CodeSmith. 2009. http://codesmithtools.com

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

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

Hibernate. 2008. http://www.hibernate.org

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

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

LinQ 2009. http://msdn.microsoft.com/en-us/vcsharp/aa904594.aspx

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