Separation Anxiety: stresses of developing a modern day Separable User Interface

Richard Kennard, Ernest Edmonds and John Leaney, *University of Technology, Sydney*

Abstract - **The evolution of User Interface (UI) tools has generally regarded the UI as separable from the underlying application it represents. This viewpoint leaves the UI having to restate invariants already specified in other subsystems of an application, and any discrepancy between the versions in the UI and those in the subsystems leads to errors.**

This paper explores a sample of real world subsystems in use by enterprise applications today, and underscores the problem of duplication between them and the UI. It then surveys the prevalence of this issue within mainstream software development.

Keywords - **separable user interface, duplication.**

I. INTRODUCTION

The evolution of software tools for constructing User Interfaces (UI) has often regarded the UI as separable from the underlying application it represents [1]. This viewpoint is sometimes implicit, but as Edmonds observes “whilst work on the specification of UIs has not in general addressed the issue of separability directly, it is quite clear that the whole enterprise is founded upon the notion of separability” [2].

In general, 'separation of concerns' [3] is a powerful abstraction for managing complexity. By encapsulating the implementation details of other subsystems, tools can develop independently to a high degree of sophistication, and “the management of design and development teams can benefit significantly” [4].

However, a consequence of such separability is that information such as names and types of fields, validation constraints and actions become restated between the UI and the other subsystems. The “separable user interface [cannot be] ignorant of the functions of the system” [2], but the desire to 'couple loosely' with minimal connecting Application Programming Interfaces (APIs) [5] makes it seem purer to resort to duplicated information between the UI and the subsystems than to introduce numerous additional APIs to exchange many small pieces of information.

From a theoretical point of view the UI is seen to hold a description of the total system at a high level of abstraction: the objects and actions of the application described at a level that is close to the users’ understanding [6]. However, in operation it is important to notice that this description must refer to dynamic data values that are set or controlled by the underlying functional code. Hence, the separable UI is left restating invariants [7] that are in reality beyond its responsibility. Any appearance that these values are arbitrary and in some way useful in providing the UI greater flexibility is an illusion. As Szekely [8] makes clear, “the functionality defines what the program can do, and the user interface defines how users tell the program what to do”. The invariants are governed by the functionality, not by the UI. As we will demonstrate, any discrepancy between the versions in the UI and those in the subsystems will result in a non-functional application.

This paper explores a sample of such underlying subsystems in use by real world enterprise applications today. It demonstrates these subsystems often have duplication within themselves, but shows how many have evolved to eliminate such duplication. It compares their approach to that of the UI layer, which remains comparatively ignorant of the issue, and gives an example of the scale of the duplication. The paper then surveys the prevalence of the issue within mainstream software development.

II. RELATED WORK

Whilst the separable UI is well represented in the existing literature, the issue of duplication is largely unaddressed. Many research projects have concerned themselves with UI builders and model-based techniques [9,10,11,12] but, as Jelinek and Slavik observe, “a common disadvantage [of UI builders and model-based techniques] 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” [13].

Whilst the general principle of minimizing inconsistency and duplication across a codebase is well established [14], its specific relevance to UIs is largely overlooked. In a comprehensive review of the state of UI tools Myers, Hudson and Pausch highlight a number of problems, such as the need to learn a new language for the modelling, but duplication is not amongst them [15].

Several researchers have at least flagged the problem. When discussing the proportion of an application's code dedicated to the UI, Edmonds points out “size is not the problem, although duplicating application code within the interface would be, of course" [2]. Researchers building end-to-end applications, as opposed to just UI tools, also have experiences that "do not support [the principle of] a high degree of separation between the UI and the application" [16] and realise that "the [back-end] requires a considerable amount of knowledge [much of which is] similar to that required by the [UI] modules" [17].

One approach that does successfully target duplication is presented by Pawson in his Naked Objects thesis [18]. However, Naked Objects removes duplication not by leveraging that in the existing subsystems, but by imposing a stylized 'behaviourally-complete' methodology on the architecture. This methodology dictates "all the functionality associated with a given entity [must be] encapsulated in that entity, rather than being provided in the form of external functional procedures that act upon the entities" [18].

As Pawson himself concedes "most object-oriented designs, and especially object-oriented designs for business systems, do not match this ideal of behavioural-completeness" [18]. Rather, most systems adopt what Firesmith calls "dumb entity objects controlled by a number of controller objects" [19], where the term 'controller objects' would include, but not be limited to, existing validation subsystems, workflow subsystems, rule engines and Business Process Modelling (BPM) languages.

As we shall sample in the next section, there are a large number of possible application architectures and a diverse range of subsystems from which to construct them. Furthermore, new subsystems become popular over time. For example, whilst business applications have been developed for many years, recently business rules engines have been introduced to formalize and externalize business rules [20]. It is not that business applications cannot be developed without rule engines - it is that they can be developed better with one. It is important not to stifle this ecosystem of subsystems by dictating the methodology or software stack.

Despite the strengths of the Naked Objects approach, its mismatch with real world business systems imposes a significant barrier of adoption, making it unsuitable for most business systems even though they would benefit from its feature of removing duplication. In order to appreciate the scale of this duplication, it is instructive to explore what those subsystems are.

III. TYPES OF EXISTING SUBSYSTEMS

This section explores a sample of mainstream subsystems in use by real world applications. The example subsystems are all taken from the Java platform, being one of the dominant enterprise platforms in use today [21].

There are a large number of possible application architectures, and not all will use all subsystems. Some will use different implementations from different vendors, some will use new types of subsystems, some will use no equivalent subsystem. The important point is that wherever a subsystem *is* used the UI must be consistent with it or, as will be discussed, the application cannot be expected to function correctly.

A. Properties Subsystem

The JavaBean [22] specification was introduced in version 1.1 of the Java platform to enable the declaration of publicly accessible properties. It is more a convention than a part of the language, as it relies on methods with a particular signature. For example, to declare a JavaBean 'Person' with two properties 'name' and 'age', a developer would write:

```
public class Person {
   private String  mName;
   private int     mAge;
   public String getName() {
         return mName;
   }
   public void setName(String name) {
         mName = name;
   }
   public int getAge() {
         return mAge;
   }
   public void setAge(int age) {
         mAge = age;
   }
}
```

Even within the boundaries of its own convention, the JavaBean syntax contains duplication. The methods getName and setName must both contain the word 'Name', and must both use a type of String. If they do not, an application error will result. In most cases, this name and type will further be mirrored by the private member variable 'mName'.

This verbosity and unnecessary duplication is a frequent criticism of the JavaBean convention. In recognition of the problem, language-level support for properties is a proposed feature for the next iteration of the Java language [23]. In the meantime, other languages already provide such support. Groovy (a language which runs on the Java Virtual Machine and has a similar syntax but different features) supports properties [24]. In Groovy, a developer would write:

```
class Person {
   String name;
   int age;
}
```

There is now no duplication. Both the name and the type only appear once for each property. Note this is quite different from simply declaring two public member variables, as properties have implicit methods that guard the setting and retrieval of their values. These implicit methods can be explicitly overridden to introduce finer-grained controls (such as a check for 'negative age') at any stage in the application's development, even after other code has already been written against the implicit methods. This 'implicit by default' approach is a significant improvement over the explicitness of JavaBeans because in general finer-grained controls will be the exception, not the rule.

Whether properties are specified using JavaBeans, Groovy, or some other mechanism, the important point is both the name and type are concretely specified by the properties subsystem. It is duplication to restate them anywhere else.

B. Persistence Subsystem

Most business systems persist their data to long-term storage, such as a database. To continue the Person example from the previous section, the developer may define the following SQL [25] schema to store a Person:

```
TABLE person (
   name varchar(30) NOT NULL,
   age int NOT NULL
);
```

The persistence subsystem contains new information compared to the properties subsystem. Strings in Java are immutable, so do not have any concept of 'maximum length'. They are also implicitly nullable [26]. Conversely, from the SQL schema it can be seen that 'name' is actually limited to 30 characters and is not-nullable (i.e. is a required field). Clearly, the properties subsystem alone is not sufficient to fully describe the business model.

However, there is also duplication. The names and types of each property have already been defined by the properties subsystem. It would not lead to a functioning system if the persistence subsystem was inconsistent. An ideal solution would be to eliminate the duplicated information whilst retaining the new information. Such a solution is provided by Object Relational Mappers (ORM) – a notable one being Hibernate [27]. Hibernate allows the developer to specify mapping files to map properties to database schemas. These mapping files include the new information:

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

There is still duplication in that 'Person', 'name' and 'age' are restated, but the duplication is at least able to be validated: if there is inconsistency between the properties and the mapping file, Hibernate will raise an error during application startup. This is an important step in reducing the margin for error, even if it doesn't reduce the duplication itself.

A next generation ORM is the Java Persistence Architecture (JPA) standard [28]. JPA achieves the goal of removing duplication entirely, whilst at the same time preserving the new information, by using metadata annotations [26] on the properties:

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

The important point is that persistence subsystems have evolved from SQL, through iterative generations of ORMs, to standardization - with a specific goal of removing duplication. A similar evolution and standardization for UIs would be highly beneficial. It might be thought of as Object Interface Mapping (OIM).

C. Validation Subsystem

Persistence subsystems generally fail poorly when presented with invalid data, returning error messages that are not suitable for end-user consumption. Therefore it is desirable to pre-validate the data and, if necessary, return more meaningful messages. Early validation subsystems, such as the Apache Commons Validator [30], use XML files to specify validation rules:

```
<form name="person">
   <field property="age" depends="intRange">
      <var>
         <var-name>min</var-name><var-value>0</var-value>
         <var-name>max</var-name><var-value>150</var-value>
      </var>
   </field>
</form>
```

As with the Hibernate mapping file in the previous section, it is evident these validation files contain both duplication ('age') and new information (minimum and maximum values). Again, it is desirable to remove the duplication whilst retaining the new information.

Next generation validation subsystems such as Hibernate Validator [30] achieve this, again using metadata annotations on the properties:

```
public class Person {
   ...
   @Min(0) @Max(150)
   public int getAge() {
         return mAge;
}
```

Standardization efforts around future validation subsystems are ongoing. They allow the developer to define sophisticated scenarios including partial validation and interrelated validation between properties [31]. For example, two properties could be mutually exclusive. If such properties were represented in a UI, filling in one may disable the other.

D. XML Serialization Subsystem

If the UI is the *user* interface to an application, XML messaging could be thought of as the machine interface. From this perspective, it shares the same problem of duplication. For example, a Web service request to load a Person may return the following XML:

```
<person age="35">
   <name>John Doe</name>
</person>
```

The 'age' attribute and the 'name' element must be consistent with those defined in the property, persistence and validation subsystems, else those systems will fail.

Modern solutions eliminate this duplication whilst retaining the extra information necessary to format the XML. For example, the Java Architecture for XML Binding [32] uses metadata annotations on the properties:

```
@XmlRootElement
public class Person {
   public String getName() {
         return mName;
   }
   @XmlAttribute
   public int getAge() {
         return mAge;
   }
}
```

The 'Person' class has metadata that declares it as an XML root element. The 'age' property has metadata that declares it as an XML attribute. The 'name' property will be implicitly treated as an XML element by default.

E. Internationalization Subsystem

In order to internationalize and localize an application, all human-readable text is generally factored into an internationalization subsystem. For example, the Java platform defines ResourceBundles [33] of key/value pairs:

Resource-en-AU.properties	Resources-it-IT.properties
name=Name	name=Nome
age=Age	age=Eta

Internationalization is seldom used during a prototyping phase, but is an important subsystem once in production. It is mentioned here as it is one of the subsystems referred to in the next section.

F. Business Process Modelling Subsystem

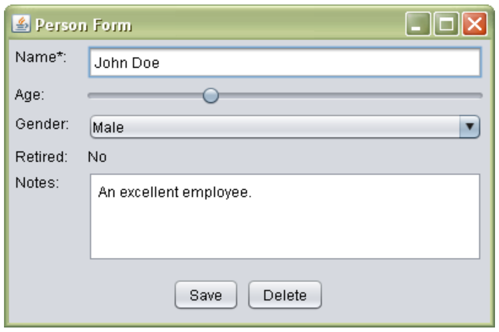
In a similar vein to validation subsystems, BPM subsystems externalize and formalize the business rules of an application. For example, using JBoss jBPM [34] a developer can specify the valid actions available when editing a Person. Generally it is these actions, and only these actions, that should be presented to the user in the UI:

```
<page name="editPerson">
   <transition name="save" to="personSaved"/>
   <transition name="delete" to="personDeleted"/>
</page>
```

The cumulative effect of the sample subsystems explored in this section is a high level of duplication with the UI. We demonstrate this in the next section.

IV. IMPACT OF DUPLICATION

To appreciate the cumulative effect of the sample subsystems identified in the previous section, this section explores constructing a hypothetical UI using a conventional UI builder or modelling language and demonstrates how much of that work is, in fact, duplication from other subsystems.


Person Form

Name*: John Doe

Age:

Gender: Male

Retired: No

Notes: An excellent employee.

Figure 1: Example UI with 5 fields

To construct the simple UI show in Figure 1, the developer must first drag (in a UI builder) or declare (in a modelling language) the labels for each of the 5 fields. The text on the labels must be semantically consistent with those defined in the properties subsystem. It would not lead to a functional system if, for example, the UI labelled a field 'Notes' which the property subsystem considered to be 'Name'. There may be slight differences – such as using a different language or more explanatory wording in the UI – but these would generally be handled by the internationalization subsystem as described in section 3E.

Second, the developer would choose appropriate UI widgets for each field. There is some flexibility here, but only a little. It would not lead to a functional system if, for example, a date picker widget was used for the 'Age' field. Similarly, the widget for the read-only 'Retired' field (which displays 'Yes' or 'No' based on age and gender) can never be an input widget. Whilst it is important to preserve the flexibility of introducing higher level abstractions (for example, a UI may choose to represent the 'Name' field as two fields 'Firstname' and 'Surname') UIs are generally 'implicit by default' - rather like the examples in sections 3A and 3D. Higher level abstractions are the exception, not the norm, as our interviews in the next section will demonstrate.

Third, the developer would apply constraints to each widget. These constraints must match those imposed in the other subsystems. The 'Name' textbox must be limited in the maximum amount of text it accepts to the same length declared in the persistence subsystem (this is different to its visual length, which may be shorter than the maximum and scroll as the user types). The 'Age' slider must have the same minimum and maximum values as declared in the validation subsystem. The 'Gender' dropdown must only contain valid values as defined by, say, an enum [26].

Fourth, the developer would designate certain fields as required fields, and label them appropriately. For example, the 'Name' field is labelled with a star. These must correspond with the persistence subsystem. It would not lead to a functional system if the UI allowed a field to be optional that the database considered not nullable.

Finally, the developer would choose appropriate command buttons. These must correspond to the subsystem that handles the action, and must be named consistently. It would not lead to a functional system if, for example, the Save button executed the Delete action. In addition, a subsystem such as a BPM would already define whether a button is applicable in a given context. For example, the Delete button may not be considered valid when entering a new Person.

In total it can be seen there are over twenty 'points of duplication' with other subsystems for only a simple UI screen with 5 fields:

- Name: *label, type, maximum length, required*
- Age: *label, type, minimum/maximum value*
- Gender: *label, type, enum values*
- Retired: *label, type, read-only*
- Notes: *label, type, large field (LOB)*
- Save: *label, action*
- Delete: *label, action*

Scaled up to real world applications with hundreds of screens and thousands of fields, such duplication goes from being unnecessary to being a significant potential for application errors. Worse is that these errors can rarely be identified statically, such as at compile-time or during application startup. The developer must rely on runtime testing to expose them.

By exploring ways to remove duplication it is possible to not only reduce such errors, but to create more robust UIs. This is because developers may choose to simply omit the duplication rather than risk it becoming inconsistent over time. For example, a developer may not specify the maximum text length on the 'Name' field in the UI at all, in the hope the validation subsystem will catch any overflows.

Not all applications will use property, validation, persistence and BPM subsystems. Some will use no equivalent subsystem, some will use new types of subsystem. Wherever a subsystem is used, however, the UI must be consistent with it or, as this hypothetical example demonstrates, only defects can result: there is no usefulness to the duplication.

V. PREVALENCE OF DUPLICATION

Having both explained and demonstrated the issue of duplication in the UI layer, it remains to understand the prevalence of this issue within mainstream software development. The authors conducted 6 interviews with senior software development practitioners from different segments of industry – including finance, medical and middleware, across the UK, the US and Australia.

The authors chose a standardized, open-ended format for the interview [35]. This approach involves asking the same standardized set of questions to each interviewee, but the set is necessarily short because each question is framed broadly so as to allow the candidates to talk openly about their experiences. Standardized, open-ended interviews allow accurate comparison and analysis of results, whilst avoiding leading the interview and therefore minimizing bias. To analyse the results, the authors employed a simplified version of grounded theory [36]. This approach involves coding, comparing and sorting categories that emerge from the interview sessions. Of principal interest to this paper was the category of duplication. Other categories that emerged will be used in future work.

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, say, a Date of Birth field to an existing Person business 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 avoids 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.

All the practitioners gave answers similar to the example in section 3. 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 - they'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 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 GUI, and that should be it really. Using the IDE we have we'd drag and drop GUI 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 [36] of reducing steps was generally from the business objects 'down' through to the persistence layer, removing the manual coding of schemas, stored procedures and recordsets, not 'up' to the UI layer.

The authors 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 to be expected 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 a Java Server Pages (JSP) or something like that. 2 steps to add a field is pretty good. The framework handles quite a lot and we can develop much faster than we normally do”. For those practitioners the authors used a further probe question [35], which specifically raised awareness of restating information: the authors 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 [27] 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”.

Following on from this, the authors 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 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 and they'd changed that one, 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 developers must rely on runtime testing to detect them. One financial software practitioner explained "we've got a BigDecimal [26], 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". 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".

Finally, the authors 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 textbox, 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 textbox on the form. If I've got a foreign key in my database, it's going to manifest as some form of listbox, dropdown, radio button, checkbox. 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".

We observed a sub-category [36] 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 [37], 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 [37] 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".

The results of our interviews suggest UI duplication is a prevalent and serious problem in software development. We observed developers across industry segments and across software platforms, and saw they had common experiences of duplication, common experiences of bugs caused by it, and a common desire for it to be addressed.

VI. CONCLUSION

We have identified a pattern of duplication in the UI layer of software applications. We have explored a sample of underlying subsystems in use by enterprise applications today and demonstrated duplication between those subsystems. In a number of cases we have shown how these subsystems have evolved to eliminate their duplication, but that the UI layer has not attempted such elimination. Finally, we have surveyed the prevalence of this issue within mainstream software development.

The authors are building an Open Source prototype to explore possible solutions to this duplication [39]. The prototype leverages the emerging field of software mining to allow the UI to inspect existing, heterogeneous subsystems rather than impose its own architectural methodology [18]. Preliminary results have been encouraging, and the authors have begun empirically evaluating the prototype by taking existing applications, enumerating the points of duplication in their UI layer (in the same way as in section 4), refactoring them to incorporate the prototype and then re-evaluating how many points of duplication have been removed. These evaluations, as well as the prototype itself, can be downloaded from http://metawidget.org.

Immediate future work will concentrate on evaluating the prototype against real world business systems, to assess the degree to which it can reduce UI duplication. A longer term goal will be to standardize such mechanisms, such that they can be adopted as part of mainstream UI development.

REFERENCES

[1] Krasner, G.E. & Pope, S.T. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of Object Oriented Programming (1988), vol. 1, no. 3, pp. 26-49.

[2] Edmonds, E. The Emergence of the Separable User Interface. The Separable User Interface. Academic Press (1992), 5-18.

[3] Dijkstra, E.W. On the role of scientific thought. Selected writings on Computing: A Personal Perspective (1982), 60–66.

[4] Newman W.A. & Edmonds, E. The Separable User Interface. Academic Press (1992), 349-355.

[5] Stevens W., Myers G. & Constantine L. Structured Design. IBM Systems Journal 13, 2 (1974), 115-139.

[6] Edmonds, E. The man–computer interface: a note on concepts and design. International Journal of Man-Machine Studies (1982), vol. 16, 231-236.

[7] Green, M. A methodology for the specification of graphics user interface. Computer Graphics 15 (1981), 99-109.

[8] Szekely, P.A. Separating the user interface from the functionality of application programs. PhD Thesis (1988).

[9] Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D. & Florins, M. USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces, W3C Workshop on Multimodal Interaction (2004), 19-20.

[10] Xudong, L. & Jiancheng, W. User Interface Design Model. Software Engineering, Artificial Intelligence, Networking, and Parallel Computing (2007).

[11] Gajos, K. & Weld, D.S. SUPPLE: automatically generating user interfaces, Proceedings of the 9th international conference on Intelligent user interface (2004), 93-100.

[12] Menkhaus, G. & Pree, W. A hybrid approach to adaptive user interface generation, Proceedings of the 24th International Conference on Information Technology Interfaces (2002), 185-190.

[13] Jelinek, J. & Slavik, P. GUI generation from annotated source code. Proceedings of the 3rd annual conference on Task Models and diagrams (2004), 129-136.

[14] Hunt, A. & Thomas, D. The Evils of Duplication. The Pragmatic Programmer. Addison-Wesley (1999), 26-33.

[15] Myers, B., Hudson, S.E. & Pausch, R. Past, present, and future of user interface software tools. ACM Transactions on Computer-Human Interaction, vol. 7, no. 1 (2000), 3-28.

[16] Manheimer, J.M., Burnett, R.C. & Wallers, J.A. 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 (1989), 127-132.

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

[18] Pawson, R. Naked objects. PhD thesis, University of Dublin, Trinity College (2004), 9.

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

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

[21] TIOBE Programming Community Index. http://tiobe.com

[22] JavaBeans. http://java.sun.com/javase/technologies/desktop/javabeans

[23] Coward, D. What's coming in Java SE 7 (2006). http://java.cz/dwn/1003/2664_Java7Overview_Prague_JUG.pdf

[24] Groovy Beans. http://groovy.codehaus.org.

[25] ISO 9075, Information Processing Systems. SQL (1987).

[26] Gosling, J. The Java Language Specification. Addison-Wesley (2005).

[27] Hibernate. http://hibernate.org.

[28] JPA. http://jcp.org/en/jsr/detail?id=220.

[29] Commons Validator. http://commons.apache.org

[30] Hibernate Validator. http://validator.hibernate.org.

[31] Bean Validation. http://jcp.org/en/jsr/detail?id=303.

[32] Java Architecture for XML Binding. https://jaxb.dev.java.net

[33] Java ResourceBundles. http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles.

[34] JBoss jBPM. http://jboss.com/products/jbpm.

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

[36] Dick, B. Grounded theory: a thumbnail sketch (2005). http://scu.edu.au/schools/gcm/ar/arp/grounded.html.

[37] CodeSmith. http://codesmithtools.com.

[38] Microsoft LINQ. http://programminglinq.com.

[39] Kennard, R. & Steele, R. Application of Software Mining to Automatic User Interface Generation. 7th International Conference on Software Methodolgies, Tools and Techniques (2008)