Application of Software Mining to Automatic User Interface Generation

Richard KENNARD and Robert STEELE
Faculty of Information Technology, University of Technology, Sydney
rkennard@it.uts.edu.au rsteele@it.uts.edu.au

Abstract. Many software projects spend a significant proportion of their time developing the User Interface, so any degree of automation in this area has clear benefits. Research projects to date generally take one of three approaches: interactive graphical specification tools, model-based generation tools, or language-based tools. The first two have proven popular in industry but are labour intensive and error-prone. The third is more automated but has practical problems which limit its usefulness.

This paper proposes applying the emerging field of software mining to perform runtime inspection of an application's architecture and reduce the labour intensive nature of interactive graphical specification tools and model-based generation tools. It also proposes UI generation can be made more practical by delimiting useful bounds to the generation process. The paper concludes with a description of a prototype project that implements these ideas.

Keywords. Automatic user interface generation, software mining

1. Introduction

Many software projects spend a significant proportion of their time developing the User Interface (UI), so any degree of automation in this area has clear benefits. Indeed, automatic UI generation has been a widely discussed topic in the research community for many years.

Research to date has shown that some 48% of application code and 50% of application time is devoted to implementing UIs [1]. Furthermore, current solutions to automatic UI generation generally take one of three approaches: interactive graphical specification tools, model-based generation tools, or language-based tools [2]. Each of these approaches has significant disadvantages which have limited their success in industry [3].

The disadvantage of the first two approaches – interactive graphical specification tools and model-based generation tools – is that they inherently require software developers to restate information that is already encoded elsewhere in the application. This duplication is both laborious and a source of errors, as the developer must be careful the application code and the UI model stay synchronized [4]. The disadvantage of the third approach – language-based tools – is that generally programming languages do not embody sufficient information to drive automatic UI generation [5], meaning language-based tools must resort to using generalised heuristics. These generalisations result in UIs that are less effective than when designed by UI experts, with due

consideration to their problem domain [6].

The main contributions of this paper are 1) to apply software mining to UI generation to automatically derive the information without the developer having to restate it, 2) to emphasise the importance of runtime software mining as opposed to static code generation for automatic UI generation, 3) to delimit the useful bounds of UI generation, such that it need not resort to generalised heuristics, and 4) to emphasise the importance of achieving high fidelity integration with target UI frameworks, allowing deep access to each one's unique capabilities.

The contents of the following sections are: section 2 discusses the Related Works; section 3 explores problems with the Related Works; section 4 proposes solutions to these problems; and section 5 describes a prototype project that implements those solutions.

2. Related Works

UIs bring together many qualities of a system that are not formally specified in any one place, if at all. For example, a dropdown widget on a UI screen may have a data type specified in a database schema but a range of choices drawn from within application code. Bringing these diverse characteristics together in one place to enable automatic UI generation is a significant challenge, and research in this field dates back over two decades. Projects including, though by no means limited to, COUSIN [7], TRIDENT [8], Naked Objects [9], UsiXML [10] and AUI [5] have all explored a variety of techniques. From these, it can broadly be observed there are three approaches: interactive graphical specification tools, model-based generation tools, and language-based tools.

2.1 Interactive Graphical Specification Tools

Interactive graphical specification tools [10] allow developers to draw UIs on the screen in a 'visual' WYSIWYG fashion, similar to how they might be sketched on paper. The tools are usually specific to a particular programming language and UI framework, and have a 'palette' of widgets based on what the underlying framework provides. The developer drags and drops widgets into position on screen, and can usually further customize them through sets of 'widget properties' such as colour and font. The tools then use static code generation to output code using the native programming language and API of the framework. In most cases this is the same API that is available separately were the developer to build the UI programmatically. A subset of interactive graphical specification tools allow the generated code to be edited manually, and a further subset perform two-way synchronization between the edited code and the tool. This latter case is fraught with complexities, however, and few implementations support it.

Overall, interactive graphical specification tools have proven very popular in industry. They have an intuitive appeal, and most established UI frameworks provide such tools. Notable examples include Microsoft Visual Studio and the NetBeans Matisse Editor.

2.2 Model-Based Generation Tools

Rather than requiring developers to specify precisely where each widget should be positioned (and more onerously, how they should resize), model-based generation tools [5][7][8] encourage a declarative approach: developers specify *what* widgets are required, but their exact appearance and layout are left to whichever implementation ultimately renders the language. There are two significant advantages to this approach.

First, whilst interactive graphical specification tools theoretically allow fine-grained customization of every property of every widget, in practice developers generally want every widget to appear the same. This is because uniformity is a desirable trait in UIs and inconsistent use of, for example, colours and fonts detracts from usability. Achieving consistency in a model-based generation tool is easier than with an interactive graphical specification tool, because model-based generation tools defer the responsibility of choosing fonts, colours and so forth to the renderer. The second advantage is that, because exact choice of widgets is deferred, the same model can target multiple UI frameworks. A notable example is HTML. A Web browser reads a declarative HTML model such as...

```
<input type="text" name="firstname">
```

...and renders it using a platform-specific UI framework, such as Win32 controls or X-Windows widgets.

In a subset of these model-based generation tools, the same model is further used to target multiple devices, for example Web and mobile-based devices. These attempts tend to be less successful, however, because generally the model does not encapsulate sufficient information to automatically regenerate the application to suit the widely varying device constraints. For example, a UI screen that fits comfortably on a single Web browser page may need to be rendered across several screens for mobile devices with a restricted screen size. The model typically does not define suitable points at which to split the screen, nor what navigation buttons to display after doing so, though some projects have investigated adding such demarcation [11][12].

Overall, model-based generation tools have proven very popular in industry. They are arguably less intuitive than interactive graphical specification tools, but offer an easier way to rapidly develop consistent UIs. Notable examples include HTML and XUIL.

2.3 Language-Based Tools

Other UI generation approaches eschew interactive graphical specification tools and models in favour of deriving the UI directly from the language of the underlying domain objects [9]. These approaches assert that all business logic should be encapsulated by the domain objects, and that the UI should be a direct representation of those objects. User actions should consist explicitly of creating and retrieving domain objects and invoking an object's methods. The advantage of this approach is that the UI can be built and reworked very rapidly from the domain.

Overall, language-based tools have had limited popularity in industry, for reasons explored in section 3. Notable examples include Naked Objects and JMatter.

3. Existing Problems

We suggest there is scope to both improve and even combine the approaches of interactive graphical specification tools, model-based generation tools, and language-based tools. To begin, it is important to understand the inherent disadvantages of each.

3.1. Disadvantage of Interactive Graphical Specification and Model-Based Tools (D1)

Whilst interactive graphical specification tools and model-based generation tools have undoubtedly proven useful in industry, they have an inherent disadvantage: they require developers to restate information that is already encoded elsewhere in the system. In doing so, they introduce room for inconsistencies and maintenance errors.

For example, a domain model may encapsulate the concept of a Person. A Person has a first name, and this is represented in a database schema as a text field. When the developer turns to building the UI, they must explicitly redefine the 'firstname' field. Either they have to drag and drop a 'firstname' text box in an interactive graphical specification tool, or they have to describe a 'firstname' input field in a model-based generation tool. Constraints such as the maximum length of the field also have to be respecified, and must be the same in both the database schema and the UI (if the UI permits a longer length than the schema, data will overflow. If the UI only accepts a shorter length, the field can never be fully used).

As the domain model changes over time these changes have to be made in both the database schema and in the UI. For example if the maximum allowed length of 'firstname' is increased or a 'surname' field is added, the screen code must be updated. If a Person appears on multiple screens, the changes usually have to be re-made separately for every screen.

UI fields often have more attributes than those embodied in a database schema. For example, a field may constrain numerical input between minimum and maximum values. Such boundaries are generally not defined by the database, but often they *are* defined by some other part of the system's architecture (in the case of minimum and maximum values, by a validation engine). An important realization is that UIs are generally descriptive, not prescriptive, of the underlying application and therefore much UI behaviour is already specified elsewhere in the system's architecture. Restating it is duplicate work.

This problem of duplication is not unique to UIs. Typically the domain model will be mirrored both in UIs and in multiple places in application code including classes, API signatures and database schemas. However, UIs tend to compound the problem because of their aforementioned characteristic of bringing together many qualities of a system. Furthermore, whilst much work has been done in addressing this problem for other areas (for example, the Java Persistence Architecture to reduce class/schema duplication), UIs have received comparatively little attention.

Those approaches that advocate interactive graphical specification tools and model-based generation tools exhibit a form of 'bottom-up thinking': they consider the problem of UI generation in isolation to the rest of the system. In reality, a real world application must satisfy both the demands of its UI and many other concerns and in doing so will already (and inherently) embody the additional information required, albeit it may be difficult to extract.

3.2. Disadvantage of Language-Based Tools (D2)

Language-based tools inherently enforce an Object Oriented User Interface (OOUI). To their proponents this is an advantage. However, they limit developers from choosing more traditional function-oriented interfaces, even if that would prove more intuitive and useful to the end-user for the given domain model.

Furthermore, language-based tools must work around the fact that, as Xudong and Jiancheng observe, 'simple naked objects are not sufficient enough for complex UI modelling. To completely and formally depict the UI composition and behaviour, new attributes and properties are needed to describe the object data members' [5]. Domain objects are generally poor vehicles with which to express all the abstractions and characteristics of a UI. Properties such as data types and allowable data values may be scattered across database schemas and application code, and characteristics such as what icons to use and how to navigate between screens may exist only in paper-based specifications.

Language-based tools must attempt to fill these gaps automatically, and necessarily resort to highly generic and stylized sets of screens and actions to do so. For example, the set of actions known as Create, Update, Retrieve and Delete (CRUD) is a popular generalisation. The use of such generalised heuristics results in a UI that appears quite differently from, and functions less effectively than, one that has been designed with consideration to its specific purpose [6].

Some approaches have explored adding additional metadata to the language in order to avoid resorting to generalised heuristics [4][13]. These are promising, but in their current form have the same disadvantage of restating information described in D1.

4. Proposed Solutions

Section 3 has identified two problems to solve: having to restate information (D1) and the use of generalised heuristics (D2). This section will propose solutions to both of these, as well as identifying two new, emergent advantages.

4.1. Application of Software Mining to D1

Software mining is a branch of data mining focused on mining software artefacts such as source files and database schemas for useful information related to the characteristics of a system. It has seen immediate application to project management, where it has been used to understand a project's status, progress and evolution [14].

However, a thorough understanding of a domain model, gathered and collated from multiple heterogeneous sources, is also readily applicable to automatic UI generation. Rather than requiring developers to restate information in an interactive graphical specification tool or a model-based generation tool, a UI generator utilising software mining could determine such information for itself.

4.2. Additional Advantage of Software Mining (A1)

The information that interactive graphical specification tools and model-based generation tools restate is necessarily limited to 'static' information, such as might be

found in source files and database schemas. Software mining, on the other hand, can also inspect runtime characteristics. This is important for three reasons.

First and foremost, many useful properties of a system are only discernible dynamically. For example, polymorphic data types and valid data values may change and be constrained depending on the use-case in progress or the access rights of the user. Second, the artefacts mined by runtime mining are inherently 'live'. They are instantiated domain objects, and can be inspected not only for their abstract characteristics but also to read and modify their current state. Finally, the cost of writing, testing, updating and documenting code should not be underestimated. Static code generation only helps with the *writing*. It does not alleviate testing and can even exacerbate updating and documenting because it quickly generates volumes of code that is generally low quality (as a consequence of generic generation algorithms) and that nobody, not even the developer who ran the generation tool, initially understands.

All combined, runtime mining significantly increases the usefulness of the UI generator. It can both generate the UI automatically and data-bind to the domain objects in both directions, whilst at the same time reducing the cost of updating and documenting.

4.3. Application of Useful Bounds of Generation to D2

Many characteristics of a UI are tightly bound to its underlying domain model. For example the name, type and allowable value of any widget is constrained by the domain object which will ultimately be used to store its data. As automatic UI generation moves away from such rigidly defined characteristics, however, it rapidly becomes highly speculative. Determining *how* to display a domain object is much more subjective than determining what fields to display. Determining how to represent relationships between multiple domain objects is more subjective still. The practical usefulness of UI generation diminishes once in these areas, because the generated UI bears less and less resemblance to how it would have appeared and functioned had it been designed manually, with consideration to its specific purpose [15].

It is, therefore, possible to delimit the *useful bounds of UI generation*. Broadly stated: that which can be unambiguously determined from what is already rigidly defined in the application's architecture can and should be automated. That which requires a degree of interpretation involving the use of generalised heuristics (such as CRUD) should be left to the developer and their existing gamut of UI development technologies.

Automatic UI generation can be seen as a way to *augment* the UI development process, rather than own it.

4.4. Additional Advantage of Useful Bounds of Generation (A2)

An approach that automatically generates the entire UI distances the developer from the underlying framework. Most UI frameworks provide a rich set of services such as data validation, navigation flow control and localisation. They further support third-party widgets, and there is usually a rich marketplace of components such as charting and data visualisation plug-ins. When a UI generator restricts access to native APIs it precludes developers from using these capabilities. By overlapping a UI framework, the generator effectively puts itself in *competition* with it: the generator needs to both

generate UIs and match, feature for feature, other frameworks. Those generators that broaden their focus, for example to task-based modelling, become increasingly vulnerable to this problem. Their scope overlaps the domain of Rule Engines, Business Process Modelling Engines and even traditional programming languages, all of which they must match in features, stability and documentation.

Having identified the useful bounds of generation, ways to automate the generation of domain model widgets whilst providing high fidelity integration with existing UI and other frameworks can be explored. This significantly increases the practical usefulness of the UI generator because, whilst much of a UI can be determined from the underlying application architecture, there will always some attributes that are not embodied elsewhere (for example, the font to use within a text field). It is important the UI generator does not restrict the developers ability to control such attributes.

5. Prototype Implementation

To explore these ideas in practice, we have built a prototype automatic UI generator. The conceptual approach of the generator is illustrated in Figure 1, showing the software mining (labelled 2, and explained in the following section) and adhering to useful bounds of generation (labelled 1 and 3, and explained in section 5.2). The prototype can be downloaded from http://www.metawidget.org.

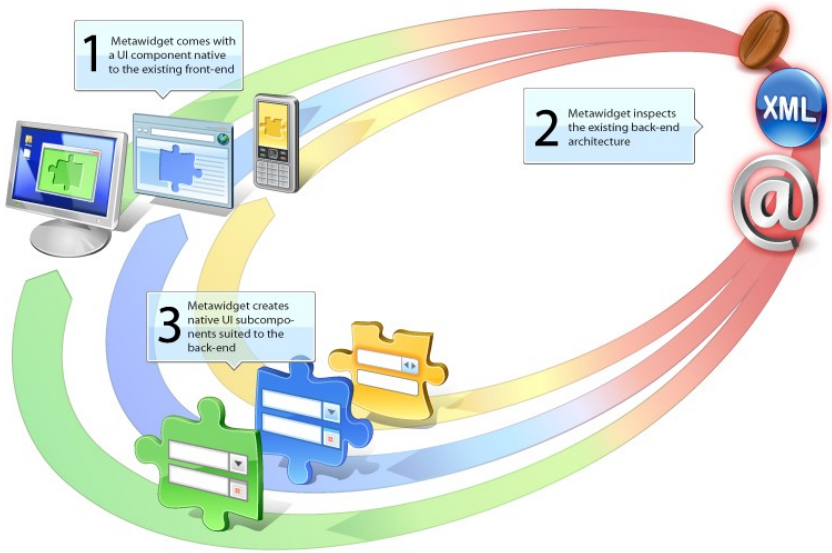

Figure 1 – Software Mining and Useful Bounds in the prototype

5.1. Software Mining in the Prototype

The prototype defines an 'inspection layer'. This layer contains a set of pluggable inspectors, each mining a different characteristic of the underlying system using whichever technology is most appropriate. This enables mining multiple, heterogeneous sources of information including, but not limited to:

- inspecting runtime objects using reflection
- parsing XML configuration files using DOM APIs
- examining language-based metadata using language-specific APIs
- parsing SQL schemas using database-specific libraries
- traversing source trees for line numbering (and hence method ordering) information

The choice of which inspectors to deploy in the inspection layer for a given application is left to the developer, and depends on the diversity of technologies in the existing architecture. The inspectors chosen should complement and leverage libraries already in use by the application, rather than introduce additional dependencies.

There is a danger with some inspection technologies that the inspectors could inspect 'too much'. Whilst inspecting, say, an XML configuration file is a reasonably well-bounded process, inspecting runtime objects is a potentially long-running task, traversing the graph of every domain object in the application. To constrain this it is important the inspection process is guided. The inspection layer in Figure 1 is shown labelled 2 because the inspectors are guided in what (and what not) to inspect by the front-end widget (labelled 1).

Each inspector returns an XML document of inspection results, conforming to a defined XSD schema. The choice of XML, as opposed to a binary representation, allows easy serialization of inspection results. This is important in multi-tier environments as it allows the inspectors to inspect, say, the database architecture on the back-end (where both the SQL schema and the APIs to access it are readily available) before returning the inspection results to the front-end (where the schema is generally inaccessible) to create the widgets.

A CompositeInspector (named after the composite design pattern [16]) is used to merge the results from several inspectors together by combining the nodes of their XML documents. For example, for two inspection results:

```
<entity type="Person">
   <property name="firstname"/>
   <property name="surname"
      required="true"/>
</entity>

<entity type="Company">
   <property name="name"/>
</entity>
```

Inspection Result 1

```
<entity type="Company">
   <property name="address"/>
</entity>

<entity type="Person">
   <property name="surname"
      length="10"/>
</entity>
```

Inspection Result 2

Top level elements sharing the same 'type' attribute are treated as representing different inspection results for the same application artefact and their child nodes are interleaved. The value of 'type' is arbitrary, but must be unique across the application architecture as this is what the merging algorithm matches on. An obvious candidate is the fully-

qualified (namespaced) class name.

Within top level elements, child nodes sharing the same 'name' attribute are treated as representing different inspection results for the same property, and their attributes are combined. This gives the final merged XML document:

```
<entity type=”Person”>
   <property name=”firstname”/>
   <property name=”surname”
      required=”true” length=”10”/>
</entity>

<entity type=”Company”>
   <property name=”name”/>
   <property name=”address”/>
</entity>
```

Merged Inspection Result

The final document is similar to those defined by the model-based generation tools discussed in section 2.2. The significant difference is that it is derived automatically: it does not have to be written or maintained by the developer (D1).

The inspection process is fed using runtime objects. This increases the accuracy of the inspection, as it can correctly discern runtime properties such as polymorphic types. It also anchors the XML document such that it can be used to data bind back to 'live' domain objects (A1).

5.2. Useful Bounds of Generation in the Prototype

Following inspection, the XML document is passed to a 'metawidget'. This is a term we have used to describe a UI widget composed of other UI widgets. The metawidget uses the XML document to populate itself with the most appropriate sub-widgets available. The metawidget is *part* of the UI, rather than trying to control the whole of it. It does not dictate such areas as what operations are available or how the user should navigate between screens. Such concerns are left to the native UI framework, which typically has a rich selection of facilities, including menu bars, tool bars and wizards (D2).

There is one metawidget per target UI framework. Both the metawidget and its generated sub-widgets are native to the framework, so can be incorporated into UIs the same way as any other widget: they can be instantiated using the framework's standard API, or dragged and dropped using the framework's interactive graphical specification tool. This means developers can utilise the metawidget whilst fully leveraging the strengths of the framework and/or device. For example, the text and input boxes in Figure 2 are automatically generated but the left-hand image and the Save/Delete/Cancel buttons are added manually using the desktop framework's native interactive graphical specification tool. Mobile phone interfaces have a very different set of constraints. The input boxes in Figure 3 are generated, but the image is omitted to save space and the Save/Delete/Cancel buttons are embedded in the phone's menu system using the phone framework's native API. The useful bounds approach provides high fidelity integration with the target framework, and avoids any hiding or restricting access to native APIs (A2).

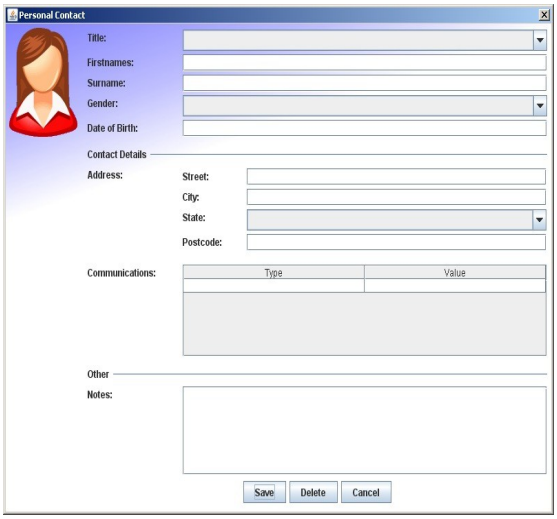

Figure 2 – A metawidget on the desktop

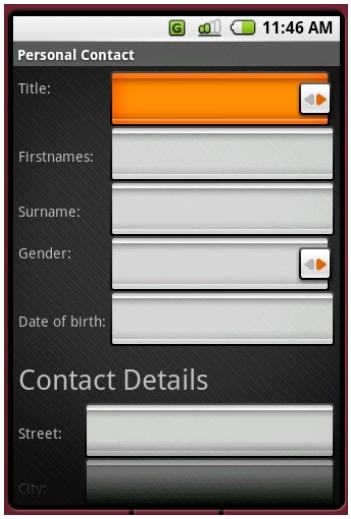

Figure 3 – A metawidget on a mobile phone

6. Conclusions and Future Work

This paper has explored current automatic UI generation solutions. It has recognised three main approaches: interactive graphical specification tools, model-based generation tools, and language-based tools. It has further identified inherent disadvantages to both the first two approaches - restating information - and the third approach - generalised heuristics. The paper has then proposed solutions to address the disadvantages: software mining and useful bounds. It has also described two emergent advantages: data binding and high fidelity integration. Finally, the paper has presented a prototype that embodies these ideas.

Immediate future work will centre on incorporating the prototype into a diverse number of applications to test its effectiveness in practice. In particular the success, adaptability and performance implications of software mining as it applies to automatic UI generation, and the effectiveness of placing useful bounds on UI generation, will need to be evaluated. The intended longer term impact of this research is to increase the flexibility and usefulness of automatic UI generation such that it becomes an accepted part of mainstream software development.

References

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

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

[3] Myers, B., Hudson, S.E. & Pausch, R. 2000, 'Past, present, and future of user interface software tools', *ACM Transactions on Computer-Human Interaction*

(TOCHI), vol. 7, no. 1, pp. 3-28.
[4] Jelinek, J. & Slavik, P. 2004, 'GUI generation from annotated source code', *Proceedings of the 3rd annual conference on Task models and diagrams*, pp. 129-136.
[5] Xudong, L. & Jiancheng, W. 2007, 'User Interface Design Model', *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, vol. 3.
[6] Falb, J., Popp, R., Rock, T., Jelinek, H., Arnautovic, E. & Kaindl, H. 2007, 'Fully-automatic generation of user interfaces for multiple devices from a high-level model based on communicative acts', *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, ed. P. Roman, pp. 26-26.
[7] Hayes, P.J., Szekely, P.A. & Lerner, R.A. 1985, 'Design alternatives for user interface management sytems based on experience with COUSIN', *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 169-175.
[8] Bodart, F., Hennebert, A.M., Leheureux, J.M., Provot, I., Sacre, B. & Vanderdonckt, J. 1995, 'Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide', *Design, Specification and Verification of Interactive Systems*. Wien: Springer, pp. 262-278.
[9] Pawson, R. 2002, 'Naked Objects', *Software, IEEE*, vol. 19, no. 4, pp. 81-83.
[10] Vanderdonckt, J., Limbourg, Q., Michotte, B., Bouillon, L., Trevisan, D. & Florins, M. 2004, 'USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces', *W3C Workshop on Multimodal Interaction. Sophia Antipolis*, pp. 19-20.
[11] Gajos, K. & Weld, D.S. 2004, 'SUPPLE: automatically generating user interfaces', *Proceedings of the 9th international conference on Intelligent user interface*, pp. 93-100.
[12] Menkhaus, G. & Pree, W. 2002, 'A hybrid approach to adaptive user interface generation', *Information Technology Interfaces, 2002. ITI 2002. Proceedings of the 24th International Conference on*, pp. 185-190.
[13] Rahman, J.M., Seaton, S.P. & Cuddy, S.M. 2004, 'Making frameworks more useable: using model introspection and metadata to develop model processing tools', *Environmental Modelling & Software*, vol. 19, no. 3, p. 275.
[14] German, D.M., Cubranic, D. & Storey, M.A.D. 2005, 'A framework for describing and understanding mining tools in software development', *Proceedings of the 2005 international workshop on Mining software repositories*, pp. 1-5.
[15] Constantine, L., 'The Emperor Has No Clothes: Naked Objects Meet the Interface', *Constantine Lockwood, Ltd–http://www. foruse. com/articles*.
[16] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.