Pattern Tool Support to Guide Interface Design

Russell Beale, Behzad Bordbar
School of Computer Science
University of Birmingham
Edgbaston, Birmingham, B15 2TT, UK

R.Beale@cs.bham.ac.uk, B.Bordbar@cs.bham.ac.uk

Abstract. Design patterns have proved very helpful in encapsulating the knowledge required for solving design related problems, and have found their way into the CHI domain. Many interface patterns can be formalised and expressed via UML models, which provides the opportunity to incorporate such patterns into CASE tools in order to assist user interface designers. This paper presents an implemented tool-based approach for the discovery of an appropriate set of design patterns applicable to a high-level model of the system. The tool accepts a UML model of the system and presents a set of interface design patterns that can be used to create an effective implementation. The tool is aimed at providing designers with guidance as to which successful design approaches are potentially appropriate for a new interactive system, acting as a supportive aid to the design process. The use of high-level modelling approaches allows designers to focus on the interactions and nature of their systems, rather than on the technologically-driven details.

Keywords: UML, Design Patterns, Modelling, Tools.

1 Introduction

Designing effective interactive systems is recognised as a difficult task. Not only is the initial design itself a non-trivial problem, but this original solution also has to be modified and reworked in the light of changing tasks or commercial requirements. The technologies people use tend to be replaced on a regular basis, adding further variables to the mix. The software engineering community has adapted to these changes by developing a component-based approach to systems design [1], in which systems are composed of a set of smaller, simpler mechanisms that solve certain issues reliably and effectively. During the process of building a large system, components are assembled together to create a more complex system. This works relatively well from an implementation perspective, in that each relatively small module can be swapped out for an improved approach iteratively improving the system, or one more suited to a new technology, keeping it up to date. However, as technologies change, the pressure to provide revised solutions means we end up hacking the once-clean design – it is not able to evolve as cleanly as each of the separate modules can.

From an HCI perspective, the constraints on interaction design imposed by the technologies are rapidly changing (different screen sizes and resolutions, ever-changing input devices, new functionalities such as digital cameras, and so on) and

providing a consistent, coherent design solution in such a rapidly moving environment is a major challenge. These difficulties are worsened over the course of rapid cycles of software production - the user can be forgotten as the technology advances and all too often new features appear in originally well-designed systems that are unnecessary, unwanted, or simply inaccessible [2]. Even when well designed initially, systems can evolve away from users' needs. Ideally, what is needed is a high level approach to designing systems that captures the requirements of the user but which is not directly linked to the underlying technologies. We also want to be able to do our interaction and user experience design and capture the resulting system in a way that enables us to instantiate it in alternative technologies, and which provides us with a framework to refer back to when revisiting the design, as we will inevitably do when new technologies come along. Essentially, we need to abstract many aspects of the design away from the low-level details, in order to allow us to reuse successful fundamentals in changing implementations.

This paper, building on previous work in the formal modelling of HCI patterns [3], reports on the design of a tool that accepts a UML model of the system and then identifies appropriate HCI design patterns suitable for the implementation/refinement of different parts of the model. This aims to assist the designer of such system by enhancing the functionality of existing UML tools. The paper also reports and evaluates a prototype tool developed on the basis of the presented method.

2 Patterns in HCI

Design patterns build upon Alexander's pioneering work in architecture [4-6], in which he introduced patterns as an approach to framing and discussing architectural problems and possible solutions. Later the "Gang of Four" [7] popularised this approach for software development. They have been embraced by parts of the HCI community as an approach to design [8-10] A *pattern* describes a recurring problem that occurs in a given context, and based on a set of guiding principles, suggests a solution. The solution is usually a simple mechanism: a certain style of layout, a particular presentation of information; techniques that work together to resolve the problem identified in the pattern. Patterns are useful because they document simple mechanisms that work; provide a common vocabulary and taxonomy for designers, developers and architects; enable solutions to be described concisely as collections of patterns; enable reuse of architecture, design and implementation decisions. Patterns are useful as they allow us to capture the salient features of a design, and the accompanying issues associated with that choice. They give us a way of sharing concepts, an approach to discussing different options, and a repository of design practices.

As well as in interface design in software, HCI design patterns have been extensively used in website development, and the consistency now observed in navigation bars, side menus and so on are down to an adoption of common approaches to solving the navigation and maneuvering issues encountered on the web, and these have been encapsulated into a set of design patterns e.g. Duyne, Landay and Hong [10]. Van Welie and van der Veer [11] provide a detailed discussion of HCI

design patterns and their formalization into pattern languages. Tidwell [12] and the accompanying site (www.designinginterfaces.com) provide a fairly comprehensive collection of design patterns describing: "what the pattern consists of", "when the pattern should be used" and "why the pattern is useful". There is also an illustration of "how the pattern can be implemented" and examples of user interfaces from real applications that implement each design pattern.

At a high level, patterns are therefore highly useful constructs. Design patterns are not perfect, however. There is no commonly accepted pattern language, and those that exist provide a framework for textual descriptions. Design patterns are usually expressed as semi-structured free-form text: they have a regularised layout of name, uses, problems and so on, with the details of the patterns described in natural language[13]. Efforts are ongoing to devise a standard XML expression (e.g. CHI 2003 workshop, 2004 workshop) [9], which will provide a framework for effective sharing and exchange of HCI patterns. Being essentially textual, design patterns rely on large quantities of real-world knowledge to interpret and understand them, are not machine-understandable, and so are hard to apply without a great deal of craft knowledge. In itself, this does not limit the scope of patterns for describing solutions to problems, but it does make accessing relevant patterns particularly difficult. Pattern libraries have essentially to be browsed manually, with the rapid identification of a suitable pattern usually coming about only by the designer having extensive familiarity with the complete set. As different authors often create patterns, it becomes hard for any substantial pattern library to identify patterns that are in fact identical, or at least very similar. The textual nature also makes the different descriptions used by different designers more critical, in that this can confuse the search of a relevant solution to a problem. Some designers would argue that this is an inherent advantage of patterns: they provide the core of a solution to a problem but in a way that allows you to use it many times without ever repeating the exact same result, enabling them to express their creativity and reflect their specific understanding of the problems of the user. But for others less experienced, not being able to access individual solutions to problems without understanding the whole pattern set is too much of a hurdle to overcome. A pattern language offers a relatively familiar structure and so suggested solutions can be understood more quickly. But being able to identify the set of candidate patterns, to find related ones, to understand the constraints imposed by one choice over another, is an unsupported, difficult task. We are not advocating an approach that allows uncritical application of patterns to problems, but do believe that designers are in increasing need of support in their tasks. One of the successes of the gang of four's book [14] was to offer standardised approaches to common software engineering problems in a way that programmers could easily select from, comprehend, use and adapt, without them having to have detailed craft knowledge of all the other patterns as well. Clearly, expert designers who understand the details will produce consistently better solutions than those who do not, but such is the pace of technological development that many of our systems are being designed by non-experts, who may well benefit from any support we can give them.

2.1 Sketch of the solution

The aim of this paper is describe a method for the discovery of design patterns, which are applicable to UML models of the system. Identifying patterns is the first step in automating the implementation of user interface design patterns. Figure 1 describes such a tool. The diagram indicates that the tool receives a system design model as its input and a set of templates modelling user interface design patterns in the UML. The tool compares a number of HCI patterns with the model and returns a report highlighting where the particular design pattern matches the input model. In effect the tool proposes a set of suitable patterns for a portion of the model. The designer can decide on a suitable pattern from the presented list. Existing UML tools can be adopted to apply such patterns to a design and create an implementation. In this paper we solely focus on identifying patterns and do not deal with the automated implementation of chosen patterns, say in a programming language such as Java, an activity which has its own challenges [15].

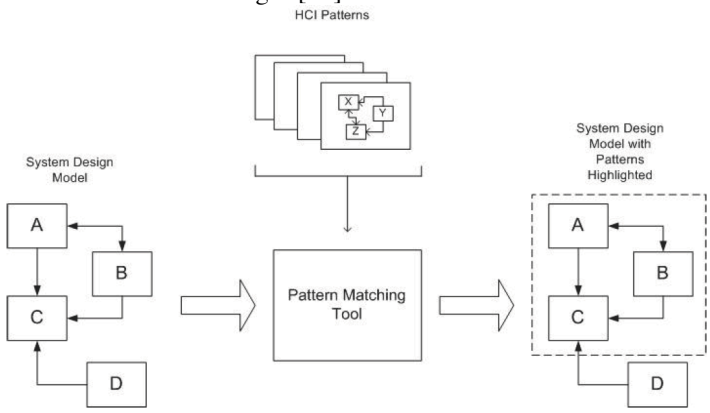

Figure 1: High-level design for a CASE tool to match HCI patterns to a system design

To create such a pattern-matching tool, two steps are involved. First, HCI patterns, which are described in an informal manner, say in Tidwell [12], must be modelled as machine-readable form. Secondly, the pattern-matching tool must implement an algorithm, which probes the model of the system to discover part of the model corresponding to each pattern. These two steps are described in the next two sections.

3 Formalizing HCI patterns in UML

Formal modelling of design patterns [7] via UML has received considerable attention. Sunye et al [16] adopt a meta programming approach, applying design patterns by means of successive transformation steps, though they do not address the issue of interaction and focus on static aspects. [17] and [18] address both static and interaction aspects of the specification of the design patterns. [19] and [20] both make use of UML class diagrams and OCL statements and suggest extensions of the

UML via a profile for the modelling of the patterns, and [21] studies the composition of design patterns.

Our recent work deals exclusively with modelling of HCI patterns via the UML and describes models for a number of design patterns. To describe the process of modelling, we shall present two related examples of a design pattern and their formal modelling (taken from [3]).

3.1 Overview Plus Detail

Tidwell [12] describes the Overview Plus Details (OPD) as follows**:** "*Use when: You need to present a large amount of content - messages in a mailbox, sections of a website, frames of a video - that is too big, complex, or dynamic to show in a simple linear form. You want the user to see the overall structure of the content; you also want the user to traverse the content at their own pace, in an order of their choosing.*
Why: It's an age-old way of dealing with complexity: present a high-level view of what's going on, and let the user "drill down" from that view into the details ... the overview can serve as a "You are here" sign. A user can tell at a glance where they are in the larger context….

How: The overview panel serves as a selectable index or map. Put it on one side of the page. When the user selects an element in it, details about that element - text, images, data, controls, etc. - appear on the other side. …"

Examples of this can be seen in the Windows Explorer and typical email clients, shown in Figure 2.

The Overview is shown in the pane on the left in Figure 2: Folders in Explorer, Mail folders in Mozilla, with details about the selected item on the right – files and folders in Explorer's case, an email message header in Mozilla's. In the Mozilla example, there is also a pane below the detail pane; the overview detail pattern is applied again to the contents of the mailbox, with a set of message headers in the top as the overview and the detail given in the pane below them.

Tidwell's description of the design pattern is typical of many – a flowing, clear, description in natural language that conveys the essence of both the circumstances under which it is appropriate and what it actually means for the interface. The full pattern also identifies other patterns that are related. The problem is that using these patterns requires very good craft knowledge, as there is no tool support or effective way of browsing or searching them. This makes sharing knowledge with up and coming designers more difficult.

Common examples of the application of Overview Plus Details are the file browsers discussed earlier, but here we shall take an email client as our example. It keeps a selectable list of email in an *overview* pane and by clicking on each email the contents are shown in another pane.

Figure 3 formalizes the *overview plus details* as a class diagram; each *Window* includes two panes; one pane is for the *overview*, which presents a high-level view of the data and the second pane is for the *detail,* which depicts the details related to the high-level view. The *overview* is in correspondence with the only one detail: this is depicted via a unary association connecting the two.

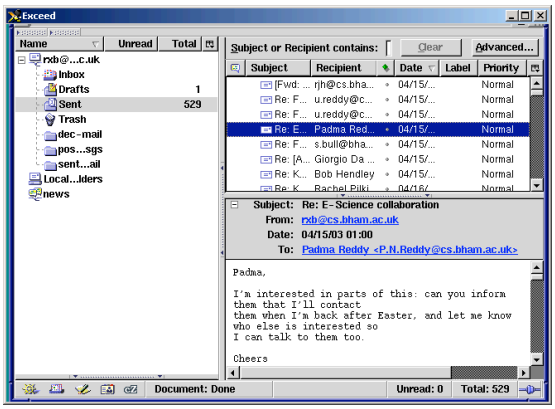

Figure 2: Overview Detail design pattern example - Mozilla

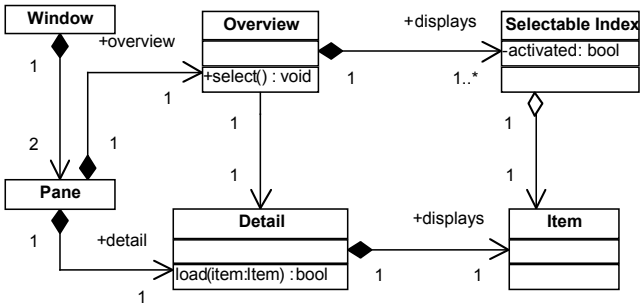

Figure 3: High level representation of Overview Plus Details

Figure 3 depicts only a static view of the Overview Plus Detail. To complete the specification of the pattern, we have to specify the dynamic aspect of the pattern by specifying the interaction between the elements. To explain this, consider the mailer example. If the user `select()`s a Selectable Index, e.g. a mail header, its state is changed on the GUI: for example, the email within the Overview window gets highlighted. This results in the change `activated = true`. As a result, the corresponding Item is downloaded to the Detail (invoking `load()`). In case of success in displaying the item `true` is returned, otherwise `false` is returned. As a result, as specified in `load(item:Item):bool`, `load` accepts an object `item` of type `Item` and return Boolean (`bool`).

Such interaction aspects of the system can be represented via a sequence diagram[22] or an OCL statement. The sequence diagram, which represents a possible interaction of the metamodel elements, is shown in Figure 4.

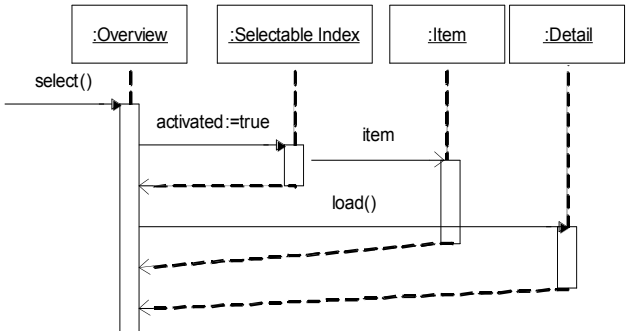

Figure 4: Sequence Diagram representing an interaction in the Overview-Detail pattern

For those unfamiliar with sequence diagrams, the diagram is read as follows. Semantically, the colon and underline around some text (:___) identifies an object. The vertical dotted line represents that object over time. The thin vertical bars represent the object within the system, with the vertical bar representing the lifeline of the object. The 'invokes' action is denoted by the horizontal arrow, and the label is the message or invocation of message or creation – i.e. the method call. The dotted line is the return of the message (the passing back of control). Progress of time moves from top to bottom. This diagram expresses that the Overview has a Selectable Index which is shown if selected (activated), and which causes the Detail to be loaded, in that order.

We can also use OCL to represent the interaction between the metamodel elements of Figure 3. The OCL representation consists of three main parts, representing the expected behaviour of each method in the context of its related model element. OCL gives us a more precise explanation, which is a logical formalism that can be automatically transformed into code and incorporated into a software tool. The OCL is presented here for completeness, as shown in Figure 5, though for general explanation and usage the sequence diagram captures the main elements perfectly adequately. Comments in the OCL are prefixed by --

```
context Overview :: select()
-- There are SelectableIndex items to select
pre selectConstraint : self.displays -> size() > 0
post selectConstraint_1 :
-- There is one item selected from the collection
-- of "displays"

self.displays -> select(s:SelectableIndex |
s.activated = true) ->
size() = 1 and
-- The selectedItem gets loaded in the Detail window.
-- The select operation returns a Set,
-- so we have to convert it to a sequence
-- and retreive the first item
-- (which is the only item of the set and the
-- selected one) so as to load it to the
-- Detail window of the overview.
-- That way we also specify that the Item related to
-- the SelectableIndex is the same as the item shown
--by the Detail window.
(
let selectedItem: Item = ((self.displays -> select(s:SelectableIndex |
s.activated = true))->asSequence->first).item in
self.detail.load(selectedItem)
)
context Overview inv itemsSelected :
-- There is at most one item selected at a time
self . displays -> select ( s : SelectableIndex | s . activated = true )
```

```
-> size() = 0 or self . displays -> select ( s : SelectableIndex | s .
activated = true ) -> size() = 1

context Detail::load (item: Item): boolean
post: if self.displays = self.displays@pre->including(item) then
    return = true
else
    return = false
```

Figure 5: OCL statement capturing Overview Plus Detail pattern interactions

3.2 Modelling One Window Drilldown

Having modelled one pattern, and shown that the concept works, we can extend this to look at a related pattern. One Window Drilldown (OWD) is an alternative to OPD. It is often used for the user interface of a device with tight space restrictions, such as a handheld device such as a mobile phone. OWD can also be used in building interfaces for applications running on desktops or laptop screens, if complexity is to be avoided. In particular, if the user is not used to computers, they might have little patience with (or understanding of) having many application windows open at once. Users of information kiosks fall into this category; as do novice PC users.

Figure 6 depicts the metamodel for the OWD. There is a single Pane to which a Current and Next data are loaded. On the selection of an item from the Selectable Index, the corresponding Item is loaded as the Next pane.

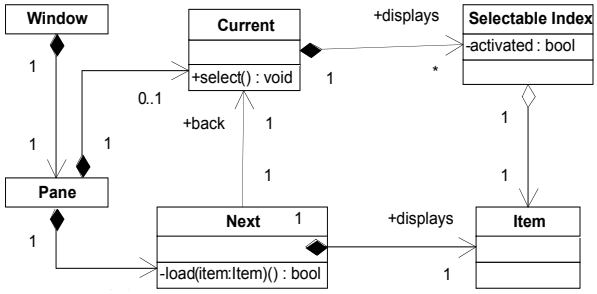

Figure 6: High level model for specification of One Window Drilldown

To ensure in the above model there is only one in Current or Next the following OCL constraint is added.

```
   context Pane
invariant:
     -- There is either a current or a next item (or both)
     -- The if statement takes out the "both" possibility
     self.current -> size() = 1 or self.next->size() = 1 and
      (if self.current -> size() = 1 then
         self.next -> size() = 0
           else
             self.next->size() = 1
       endif
      )
```

This essentially says that, in the diagram, there could be 0 or 1 Current screen and 0 or 1 Next screen – and we can only display one at a time, hence the need for the constraint.

The behavioural model of the OWD is exactly the same as in Figure 4, which we would expect since the interaction is very much the same. The OCL statement is essentially the same as well (with minor variations, not presented here).

3.3 Extensions to other patterns

Clearly, there are many more HCI design patterns than Overview Plus Detail and One-Window Drilldown. However, these two patterns show that we are able to capture both structural *and* behavioural aspects of the pattern, allowing us to represent the essence of the interaction formally. This ensures that this approach is general, able to capture salient aspects of other HCI patterns, for both existing patterns, and for new ones. This is critically important, for if the approach only captured one or other aspect of the system, it would not allow us to apply it more widely. Three further patterns are given, very briefly, to show the general applicability of the approach (taken from [12]).

3.3.1 Card Stack

The card stack interface is typically used when there are multiple pages of information to display that can be segmented into a relatively small number of meaningful categories. The meta-representation of the pattern is more or less the same as Overview Plus Details (OPD), shown in

Figure 7.

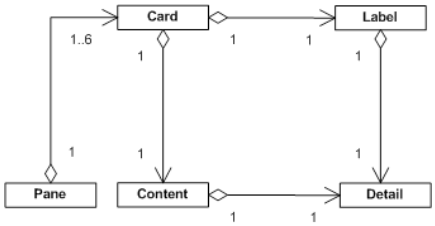

Figure 7: Meta-representation of card stack pattern

The data model of this is very similar to the data model of Overview Plus Detail, except for the following two points: the overview part in Card Stack must be small, consisting of one or two words (or small icons), and secondly that it is better to have under six cards. The data model is shown in Figure 8: similar to OPD, information represents the data, label is the same as overview and detail is not changed.

Figure 8: Card stack data model

3.3.2 Cascading list

The meta representation, shown in

Figure 9, consists of a number of panes, each at the left hand side of 0 or 1 panes. Each pane contains 1 or more Items. Each Item is related to a list of Items in lower hierarchy. By clicking on each Item, the method `LoadAtRightPane()` is invoked.

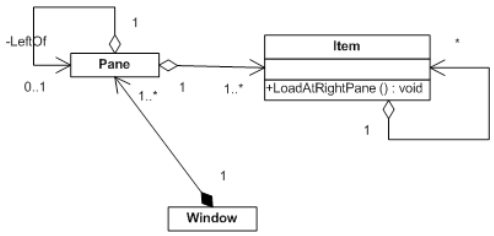

Figure 9: Cascading list meta-representation

The data model represents a hierarchy of information, in which each item has potentially many more subitems -

Figure 10.

Figure 10: Cascading list data model

3.3.3 Top-level Navigation

Commonly seen in websites and other internet-based applications, the top-level navigation model puts tabs or links across the header of the page to provide main access to the key areas of the site or application. The conceptual model of related data is captured in Figure 11. The emphasis is on: "Application had a number of main divisions" [12].

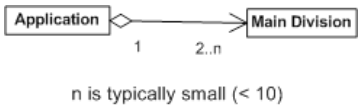

Figure 11: Data model for top-level navigation

Top-level navigation implements the above information as follows (Figure 12). The UI has a single *Top level navigation bar* and a single Content Area. The top-level navigation bar has a clickable affordance. The method `Click()`, if invoked, uploads the Division into the Content Area.

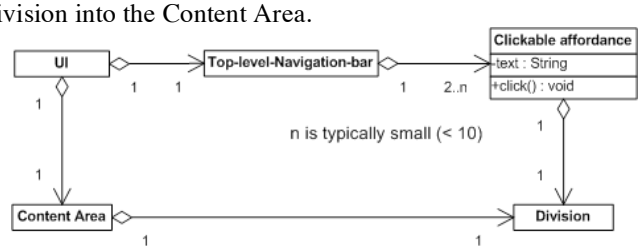

Figure 12: Top-level navigation meta-representation

3.4 Issues and limitations with the representation

The UML representation captures the behavioral and structural characteristics of an interaction artifact that provides a solution to an interface design problem. One criticism of the approach is that it only captures this aspect, and does not include issues such as the problems of the user – represented as the 'Use When' approach in Tidwell's formalism – or the set of other possible actions that they could undertake. Our design pattern is constructed on the basis that, if the data has a specific structure, and you wish to display it to the user, then you could use this particular pattern to guide your solution. It does not identify whether you actually do want to display this part of the system to the user. This represents a subtle change in thinking about design: rather than patterns representing solutions to questions of the sort "I want to show this information to the user; how can I best achieve this" we have instead "I have this sort of information; if I wanted to show it to the user, this is how I could do it". The designer is then faced with a series of choices of what to show from a set of possibilities, derived from applying the patterns to the system model.

Notice too that aesthetic aspects are not captured in the abstract representation – nor are things such as the 'clickable affordance' mentioned earlier. There are two possible resolutions to this issue: the first is to recognise that the approach is focused on assisting designers identify a set of potential solutions to a particular problem, but is not dictating specifics, allowing them to focus their efforts on developing appropriate, effective implementations for the identified areas. The second, more software engineering oriented approach, is to capture some of these specific aspects in a Device Profile Model which provides details of the instantiations of UI elements for specific platforms and device. Our approach is aimed at formalising, structuring and supporting the use of design knowledge and past effective solutions: it is not aimed as

a machine-based replacement for the activity of designers, but tries to support them in suggesting areas to focus their efforts, and outline solutions for them to consider and modify.

We do not claim that our system automates interface design, or removes the need for subjective, aesthetic expertise – but it does guide the designer and narrow down the multiplicity of patterns to only a relevant subset. How we achieve this is discussed in the next section.

4 A tool for recognizing HCI patterns

If we examine the UML meta-representation for, for example, Overview plus Details, we can see that it comprises both graphical aspects – the window, containing 2 panes – and interactional aspects: when an item is selected in the overview pane, the detail corresponding to that is shown in the detail pane. If we concentrate on the from of data that could be represented in this way, it can be seen that the data structure suitable for the above must have the following general shape: a data type *A* (*overview*) has a list of items *B* (*Selectable Index*) and each item *B* is in one-to-one correspondence with a data type *C* (*Item*). Figure 13 encapsulates this concept.

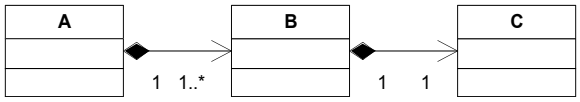

Figure 13: Type of data suitable for Overview Plus Details

It is this concept that forms the basis of the software tool: these pattern signatures, based on their datatypes, offer us a way of identifying which parts of a UML model may be suitable for a patterns representation, as long as we can match the datatype signatures on the actual UML model of the system, and the prototypical system.

4.1 Example system: an email client

Consider in Figure 14, which depicts a system model for an email client. This contains the usual things we would expect in such a client – multiple users, multiple mailboxes, folders that contain email, with messages listed by date, time, title, priority and sender, and so on. This model essentially states that we have a Mail Server that can have many users. Each user can have one or more mailboxes, and each mailbox has both an Inbox and a Local Folder. Each of these can be subdivided into more folders. These folders contain email, and this email is comprised of one or more identifiers with a variety of fields, and associated with each identified chunk is some content.

By examining the data types within the model, we can see that the part of the model including *email*, *identifier* and *Content* matches Figure 13, i.e. Overview plus Details, as marked by letters A, B and C, accordingly.

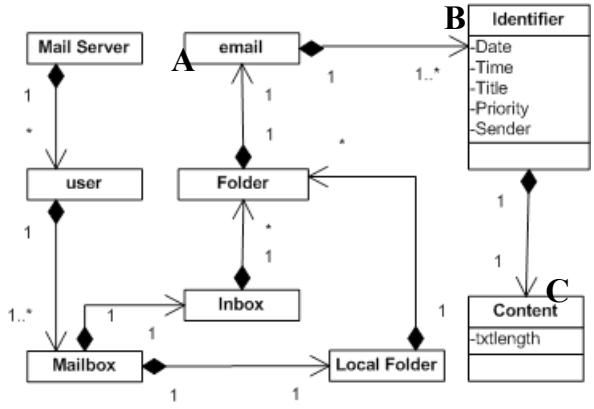

Figure 14: Simplified email client

Also matching is the user:Mailbox:Inbox and user:Mailbox:Local Folder – all three of these element sets are therefore potentially amenable for display using the Overview Plus Details pattern, as shown in Figure 15. We have seen examples of the data modelled as Part A of Figure 15 as both OPD and OWD. For example, OPD is used for popular desktop-based mailers such as Mozilla and Outlook – the right hand side of Figure 2 shows this, for example. OWD is used in shell-based mailers such as pine and mailers on the PDA, and mobile phone displays, which can't use OPD because of the size restrictions of the screen.

Part B is also seen in mail systems in which users have many email accounts and collect them together on one email server: by representing this in the display the user is able to identify which mail account they want to deal with. Part C is similar to B except that the user can see the Local Folder that is associated with the selected Mailbox. For parts B and C it is also clear from the diagram that, whilst these patterns are appropriate to display this information, only part of the system will be shown to the user, since Mailbox does have both an Inbox and a Local Folder.

As well as being able to correctly identify the correct parts of the system that can be modeled this way, it is equally important that the system does not incorrectly identify other parts that in fact cannot. This is the case, as the data signature does not map onto other aspects of the system. For example, Mail server – User, User – Mailbox: this has a different data signature (1:*, 1:1..*) and so is not identified as appropriate. We can undertake a thought experiment in which we try to envisage an interface to a mailer in which these parts did appear in an overview-detail representation: we have to modify our conceptions of the system – there would have to be at least one user, rather than possibly none, and we would have to consider the collection of users' mailboxes as one thing – and if we did that we are altering the model, giving it a signature of 1:1..*, 1:1 which would clearly then be suitable for OPD or OWD. These changes represent substantial changes in our conception of the mailer, and whilst it would be possible for a designer to want to represent this information in this manner, this approach ensures that the underlying model is modified to fit the new concepts introduced.

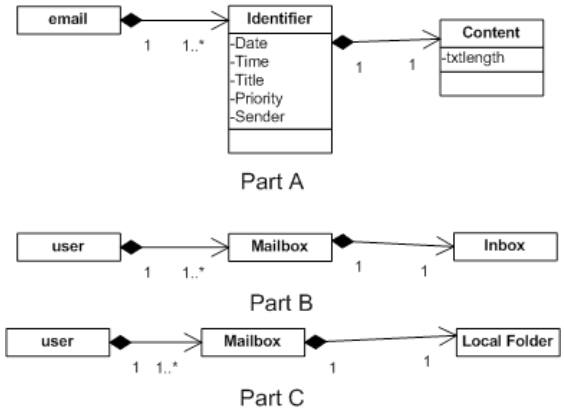

Figure 15: Parts of the email client that can be mapped to the Overview Plus Details pattern

There are two major observations from this. Firstly, it is possible to identify data models similar to Parts A-C automatically. In other words, it is possible to programmatically scan a UML class diagram and identify all parts of the model that can be refined via OPD and OWD. Identifying such patterns paves the way to creating tools that can prompt a designer, suggesting the application of suitable design patterns for relevant parts of the system. At present, using this approach does allow us to automatically identify all the design patterns that are potentially appropriate for representing different elements of the system, and, equally, rules out patterns that are unsuitable. This represents a step forward for the use of patterns – we do not have to be familiar with all the patterns in a library, but can rely on them being indexed and identified by the types of data that they can represent.

4.2 The design pattern tool

We have seen that the idea behind the method is to create a tool to identify fractions of the UML model within a system model that match a design pattern. More specifically, this means recognising instances of one of the diagrams (typically the smaller diagram, the HCI pattern in this case) that occur within the other diagram. Partial matches are of no interest, but multiple instances of the same pattern within a diagram are.

The basic algorithm for matching patterns in the diagrams' structure works by attempting to use each element of the system model as a start point for comparison with the HCI pattern. The algorithm then attempts to compare the structure of the start point with a designated `first' element from the HCI pattern. A comparison is successful if the two objects being compared have associations with the same properties pointing to them. For example, if the HCI pattern element under consideration has two associations pointing to it, then the system design element must also have at least two associations and, further, the multiplicity at the end of each

association must be the same in both models. It does not matter if the system model element has other additional associations that do not match the HCI pattern. A simplified example of this is presented in Figure 16.

The figure shows a UML representation of an HCI pattern and System design model. It is clear that the HCI pattern's structure is replicated in the system model. In fact, there are two instances in which such a mapping could occur. The two possible mappings are: "*Object X maps to Object A; Y to B; Z to D*" and "*Object X maps to object D; Y to B; Z to A*". The algorithm will recognize both of these instances, ignoring the fact that objects A and D in the system model have other associations that do not map to the HCI pattern. Using the approach outlined above, it is possible to recognize all of the *direct* mappings between the structures of HCI patterns and system models.

The above approach is successfully implemented as a prototype tool [23], which is an Eclipse plug-in working with Omondo [24]. First, the UML models of the system, which are captured in XMI format by Omondo, are transferred to collection of Java objects and the above algorithm is implemented in Java.

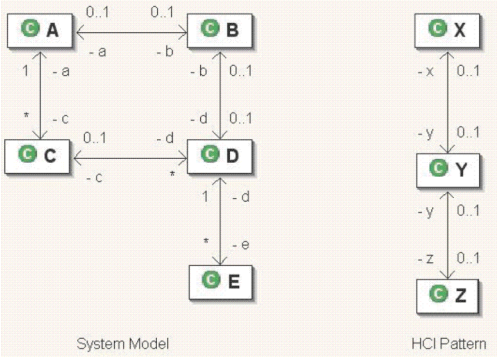

Figure 16: Matching UML and HCI pattern models

5 Discussion and Further Work

This approach is not a universal panacea. For example, a large number of the HCI patterns are subjective. It may not be possible to model such patterns in UML. For example, consider the ``Intriguing Branches" design pattern [12]. In the pattern's description, as part of the section on "how to implement the pattern", the following guideline is given: ``*Start with a deep understanding of your users. What might interest them? Where in the interface are they likely to take time to explore something further, and where do they just need to get something done?*" Formal modelling of such patterns is a far more complex task than that dealt with using UML languages. Hence, our approach can not deal with these forms of HCI patterns – though others have argued that these types of design approach should not be termed patterns anyway [13]. However, this should not be seen as a terminal problem for the approach – the

system aims to guide designers through a morass of patterns, and whilst not all aspects of designs may be contained within the pattern set, an extensive and comprehensive pattern library is already in existence, for which guidance can be highly valuable.

Another major issue that is the level of complexity the application designer has entered into when drawing up their system designs. It is possible for two designers to represent the same system, in the same format (a UML class diagram, for example) and still come up with vastly different output. One designer may wish to group small components together to form a single component that combines their functionality, whilst another may wish to express every component, however small, in their designs. As an example, consider and imagine if the system designer had, instead of representing component `B' as a single class, decided to represent `B' as two separate components `B1' and `B2', linked by a `one-to-one' association (Figure 17).

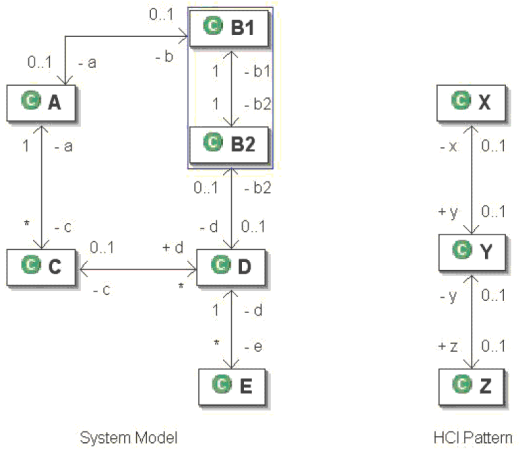

Figure 17: Alternative system model to Figure 9, containing B1 and B2 elements

Clearly, the components `B1' and `B2', if viewed as a single component (enclosed in the box) result in the same diagram as that in Figure 9 and thus the same output from the pattern matching tool. However, as the diagram is presented, the tool would come up with no *direct* matches between the HCI pattern and the system design. This can be rectified by including additional runs of the algorithm in which elements of the system model are permitted to be 2,3,4... actual classes in size. However, this feature is not implemented in our prototype version [23] and remains a subject for further work. The solution is known, however: this will be tackled by implementing a backtracking algorithm, in much the same way as the string searching utilities such as `grep` work when trying to find matches to complex expressions containing wildcard identifiers such as '*'.

6 Conclusions

The tool does offer identification of some relevant design patterns given a UML model of a system, and so takes us some way towards our goal. By having a high-level model, we can adapt our implementation to changing technological bases or to revisions in functionality relatively easily (and in some cases, automatically, using machine translations from platform independent models to platform specific ones). We can then use these models to give designers some guidance as to which parts of the model are able to be represented with which design patterns. Clearly, there will be parts of the system that require no such interface presence, and other parts in which multiple patterns will be identified, and so we see this as giving support and guidance to designers. By identifying potentially appropriate patterns, it reduces the barriers to wider use of design patterns, and so could promote more effective interfaces through the use of known solutions to problems. As well as producing better systems, this also speeds up the implementation phase, allowing even faster production of code, potentially keeping up with the pace of technological change.

References

1. Szyperski, C., D. Gruntz, and S. Murer, *Component Software - Beyond Object-Oriented Programming (2nd edition)*. 2002: Addison-Wesley / ACM Press.
2. Thimbleby, H. *The computer science of everyday things*. in *Second Australasian User Interface Conference Proceedings (AUIC 2001)*. 2001.
3. Beale, R. and B. Bordbar. *Using modelling to put HCI design patterns to work*. in *HCI International. 11th International Conference on Human-Computer Interaction*. 2005. Las Vegas, Nevada, USA: Lawrence Erlbaum Associates, Inc (LEA).
4. Alexander, C., *A city is not tree*. Architectural Forum, 1965. **122**(No. 1 pages 58-61, No. 2 pages 28-62.).
5. Alexander, C., *Notes on the Synthesis of Form*. 1964: Cambridge, Massachusetts: Harvard University Press.
6. Alexander, C., et al., *A Pattern Language*. 1977, New York: Oxford University Press.
7. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994: Addison-Wesley.
8. Dearden, A. and J. Finlay, *Pattern Languages in HCI: A Critical Review*. Human-Computer Interaction, 2006. **21**(1): p. 49-102.
9. Erickson, T. *Interaction Design Patterns Page*. [cited 2008 18th September]; Available from: http://www.visi.com/~snowfall/InteractionPatterns.html (was www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html).
10. Duyne, D.K.V., J.A. Landay, and J.I. Hong, *The Design of Sites: Patterns: Principles and Processes for crafting a Customer-Centered Web experience*. 2002: Addison Wesley.
11. van Welie, M. and G.C. van der Veer. *Pattern Languages in Interaction Design: Structure and Organization*. in *Human Computer Interaction (Interact 2003)*. 2003. Tokyo, Japan: IOS Press, IFIP.
12. Tidwell, J. *Designing Interfaces: Patterns for Effective Interaction Design*. 2005 [cited 2008 8th June]; Available from: http://designinginterfaces.com/ (was http://time-tripper.com/uipatterns/index.php).
13. Mahemoff, M. and L.J. Johnston, *Usability Pattern Languages: the "Language" Aspect*.

14. Gamma, E., et al., *Design patterns: elements of reusable object-oriented software*. 1995: Addison-Wesley Longman Publishing Co., Inc. 395.
15. Akehurst, D.H., W.G.J. Howells, and K.D. Maier, *Implementing Associations: UML 2.0 to Java 5*. To be published in Journal of Software and Systems Modeling, 2006: p. 1 - 33.
16. Sunye, G., A. Guennec, and J.-M. Jezequel. *Design Patterns Application in UML*. in *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*. 2000.
17. France, R., et al., *A UML-Based Pattern Specification Technique*. IEEE Trans. Softw. Eng. PU - IEEE Press, 2004. **30**(3): p. 193-206.
18. Kim, D., et al. *A UML-Based Metamodeling Language to Specify Design Patterns*. 2003;
Available from: http://www.cs.colostate.edu/~georg/aspectsPub/WISME03-dkk.pdf.
19. Mak, J., C. Choy, and D. Lun. *Precise Modeling of Design Patterns in UML*. in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. 2004.
20. Dong, J. and S. Yang. *Extending UML To Visualize Design Patterns In Class Diagrams*. in *Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 2003. San Francisco Bay, California, USA.
21. Dong, J. *Representing the applications and compositions of design patterns in UML*. in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. 2003.
22. OMG. *Unified Modelling Language (UML)*. 2005; Available from: www.uml.org.
23. Evans, M.J., *From System Models to HCI Design Patterns via the Model Driven Architecture*, in *School of Computer Science*. 2006, University of Birmingham: Birmingham. p. 44.
24. Omondo, *The Live UML company: Omondo Eclipse - Free Edition*. *Software package available at www.omondo.com*. 2006.