Context-aware UI Component Reuse

Kerstin Klemisch[1⋆], Ingo Weber[1,2], and Boualem Benatallah[1]

[1] School of Computer Science & Engineering, University of New South Wales
{kerstink,boualem}@cse.unsw.edu.au
[2] Software Systems Research Group, NICTA, Sydney, Australia
ingo.weber@nicta.com.au

Abstract. Adapting user interfaces (UIs) to various contexts, such as for the exploding number of different devices, has become a major challenge for UI developers. The support offered by current development environments for UI adaptation is limited, as is the support for the efficient creation of UIs in Web service-based applications. In this paper, we describe an approach where – based on a given context – a complete user interface is suggested. We demonstrate the approach for the example of a SOA environment. The suggestions are created by a rule-based recommender system, which combines Web service-bound UI elements with other UI building blocks. The approach has been implemented, as well as evaluated by simulating the development of 115 SAP UI screens.

Keywords: User Interface Development,UI Component Reuse, Ripple-Down Rules, Context-awareness.

1 Introduction

There are few ICT areas as economically and socially critical today as the provision of services over mobile and Web channels. This trend is impacting profoundly the effectiveness of services delivery in a variety of domains including health, banking, education, healthcare, meteorology, forecasting, media, and office tasks. For instance, over 577692 Android applications were accessible in December 2012 [1]. The number of iPad applications reached 275000 in March 2012 since its launch in 2010 [2]. From a software engineering point of view, while advances in mobile and Web technologies increased the ability to deploy simple services and devices, demand for effective development of mobile applications is rising even faster. More specifically, the development of user interfaces has been identified as one of the most time-consuming tasks in the overall application development cycle [26]. This is partly due to the fact that user interfaces have to be developed for a large number of different devices with different development platforms, such as XCode for Apple iPhones and iPads, Android SDK for Android mobile phones, or Carbide IDE for Symbian applications [13].

Modern Integrated Development Environments (IDEs) support UI developers by providing: reusable basic UI elements (e.g., text boxes, sliders) from generic

⋆ The majority of the work done while this author was working for SAP.

programming libraries (e.g. JQuery [22], VCL [25]), code generation / completion for both UI and functionality as in MS Visual Studio, UI to functionality binding specification, etc. However, even with sophisticated IDEs like MS Visual Studio or Bootstrap [24], there are still a lot of repetitive, time-consuming tasks that UI developers have to code manually. Examples of such tasks are the repetitive drag and drop of basic UI elements (e.g., text boxes, buttons) onto a canvas, creation of UI elements from scratch, configuration of UI element parameters (e.g., size), specify UI screen layouts, define labels and descriptions. In addition to describing UI elements, developers need to develop mappings from UI elements to backend functions and vice versa (e.g., mapping Web form fields to Web service (WS) parameters, or WS invocation results to Web widgets).

In this paper, we focus on improving productivity of UI development on top of Web services, as Web services are widely used to provide backend interfaces to all sort of applications ranging from ERP software to cloud services. A good example to illustrate the reuse of backend functionality via services is Twilio[3], a provider of various Application Programming Interfaces (APIs) for messaging services such as telephony and SMS. A large number of applications have been developed on top of Twillio in a variety of domains. For instance, recently Twilio announced that 150,000 developers used its APIs to support phone calls, video conferencing and chat features including the development team of eBay and Hulu. Application development over services such as Twilio is UI intensive. When offering a service in different contexts, its UI has to be adapted to these contexts, including, e.g., the adaptation to new consumption channels, new audiences, and the change of UIs as an outcome of usability testing [6].

In addition, in enterprise applications, adaptation to business scenarios is a major requirement in UI development. A business scenario is essentially a "complete description of a business problem" [23]. Examples of business scenarios are sales order management and supply chain management. A business scenario can be associated with a collection of reusable UI elements including UI templates, backend services, and mapping templates between the two.

Adapting UI code to a given context can be complex, time-consuming and resource-intensive. In major software development projects, various techniques are used to create appropriate UI prototypes before the development phase starts. These techniques include brainstorming, mockup design iterations by experienced UI designers, and usability testing. Existing UI development and adaptation techniques have not kept pace with our ability to deploy individual devices and services. They rely on human understanding of different contexts, platforms, devices and extensive manual programming to develop UI components. This is clearly unrealistic in large scale and evolving environments. Main stream code reuse techniques assist developers in finding and reusing general code artefacts, such as functions and methods [14, 18]. Although these techniques have produced promising results that are certainly useful, more advanced techniques that cater for effective and context-aware UI components reuse are

[3] `http://www.twilio.com/`, accessed 25-11-2012

needed. Whereas the backend functionality provided by code might be exactly the same on a desktop and a mobile device, the UI is likely to be different.

In practice, we observe that UI developers often have valuable knowledge for identifying UI components that are most appropriate in a given context (e.g. UI components appropriate for sales order entry screens). We believe that sharing this knowledge is beneficial for UI code reuse, and thus that integrating context-ware UI components reuse and SOA offers tremendous opportunities to increase productivity of modern application development frameworks. More specifically, based on the above observations we propose a novel approach for context-aware UI component reuse, making the following contributions:

1. We propose a UI artefacts representation model and a rule-based UI component recommender system. The recommender system suggests comprehensive UI components to developers. UI Programming knowledge is exploited in the form of programming recommendations provided during UI development. In essence, the recommender system uses a context description (e.g, device type, business scenario) to query a "UI programming knowledge base", which returns UI components that are "appropriate" for the given context.
2. We propose an incremental knowledge acquisition technique, by which newly created or modified UI components are associated to contexts and stored in the UI programming knowledge base for future reuse. The UI programming knowledge underpins knowledge-driven and context-aware UI components recommendations.
3. We discuss an evaluation of the proposed approach, including (i) a proof-of-concept prototype called UISE (UI Suggestion Environment); and (ii) a detailed feasibility and usefulness experiment based on 115 UIs from the SAP Business Suite CRM software.

The paper is structured accordingly, starting with how UI artefacts are represented. Section 3 describes how UI artefacts can be reused and how UI knowledge is incrementally acquired. The implementation and evaluation is discussed in Section 4, followed by related work (Section 5) and our conclusions (Section 6).

2 Representing UI Artefacts

The core idea of our approach is to support UI developers in their daily tasks by suggesting user interface components during the development process. These UI suggestions are based on context selected by the developer. Our approach derives suggestions for user interfaces from UIs created for similar contexts in the past. The suggested user interfaces consist of a number of UI widgets. These widgets might come with a specific service and mapping already linked to it. Where no service / mapping link is provided, the developer needs to perform service discovery – which is outside of the scope of this paper. The UI developer can modify the UI proposals according to her needs, or reject the proposal and create a new UI from scratch. The changes / new UI will be made available to other UI developers on its completion.

UI components, context and recommendations are stored in a rule-based knowledge base (KB) in our approach. Updates to the KB are triggered by any

change in context or UI. The KB structure consists of artefact types and relationships linking them, as shown in the entity-relationship diagram, Fig. 1. The main artefacts are **Rules** (for UI recommendations). The rules consist of **conditions** (when does the rule apply) and **conclusions** (what should be recommended when the rule applies).

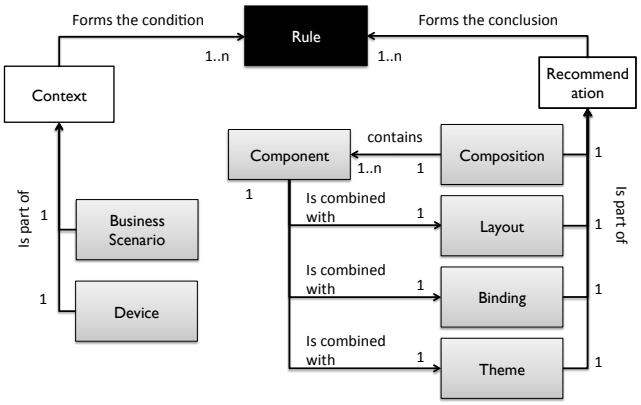

Fig. 1. ER-Model of Rule Components

2.1 Rule Conditions

The left hand side of Fig. 1 shows artefacts which make up rule conditions or the Context. **Devices** (different consumption platforms, e.g., Blackberry, iPhone, Windows Desktop, etc.) and **Business Scenarios** (e.g., Sales Order Management, Quotation Management, Campaign Management for Business Customers, etc.) describe our **Context**. Information on how this context was assessed can be found in Section 4.1. In this paper, we have restricted ourselves to two dimensions of context. The context could however be easily extended by other dimensions, such as corporate identity or development platforms.

These artefacts form the vocabulary used in the condition of a rule, e.g.: *If device=*"Apple iPhone" *and screen resolution* = 800x600(+/-20%) *and scenario=*"Sales Order Management-Standard Sales Order"*, then...*

2.2 Rule Conclusions

The conclusion of a rule is a UI recommendation, for a given context. User interface suggestions in our approach are composed of UI **Components**. These components can be of different levels of granularity, ranging from simple textboxes with labels to entire user interfaces. In order to capture relevant aspects of UI components in a generic way, we represent them through four basic dimensions (see right-hand side of Fig. 1).

– **Composition** specifies which UI components are combined to form the structure of a UI (component). For instance, a screen for the scenario "Sales

Order Management - Standard Sales Order" might be composed of the UI components "sales order header V1", "sales order date section V3" and "sales order item list V1".

– **Layout** describes how the components are arranged on the screen – see Fig. 2. We distinguish horizontal layout, where components are placed next to each other, and vertical layout, positioning components one beneath and above one another. Vertical and horizontal layouts can be nested, thereby allowing to model arbitrary combinations of horizontally and vertically ordered components. Layouts can be used to adapt a UI for different devices – e.g., a collapsible vertical layout may be suitable for mobile devices with small screen sizes, whereas a horizontal layout may be more suitable for wider desktop screens.

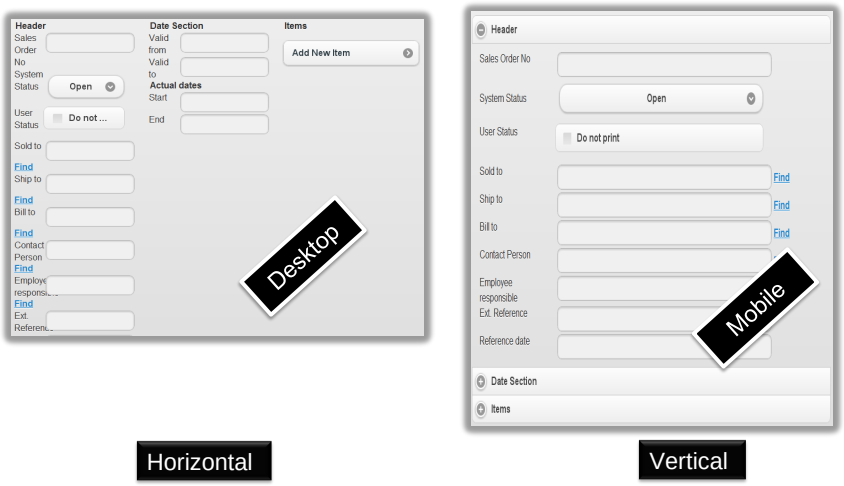

Fig. 2. Examples for different layouts

– **Binding** links a UI to back-end Web services, and describes how data should be transformed and exchanged. One UI might be used for different purposes, like accessing different back-end services, even without changing the UI itself. The flexible binding mechanism allows for changing the functionality in the background without changing the UI as such. An example is the creation of user interfaces for *creating* a sales order vs. UIs for *changing* a sales order. The UI may remain the same, whereas the Web service the data is submitted to may change.
– **Theme** refers to style sheets, which define the look-and-feel of UIs. Style sheets are a common method to adapt UIs to the look-and-feel of different operating systems (Windows, iOS, etc.) and to different corporate designs (e.g., Coca Cola vs. IBM). This is achieved by varying attributes like font size and style, colour scheme, background images, and even scripts.

3 KB-based Reuse of UI Artefacts

A central part of our approach is the reuse of UI artefacts. This reuse is enabled by a rule-based recommender system, which suggests user interfaces depending on the current context, as specified by the developer. The suggested UIs stem from previous development efforts. The high-level architecture of the proposed **Recommender System** is shown in Fig. 3. We explain its parts and aspects of its usage in this section, starting with the information used by the recommender system. Then we describe how reuse rules are structured, how they are entered into the system, and how they evolve over time.

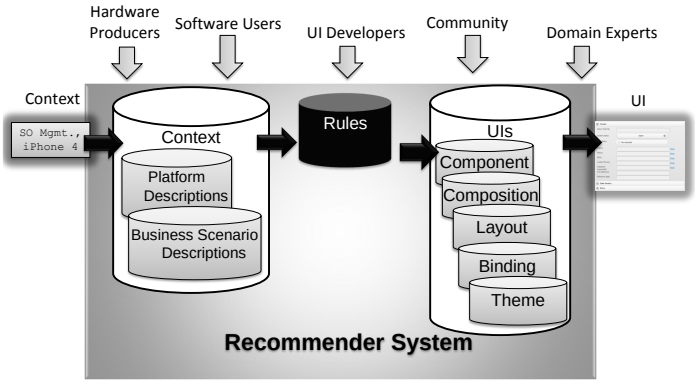

Fig. 3. Proposed Architecture

3.1 UI Recommender System

The recommender system stores **Context** data for platforms and business scenarios. The **Platform Descriptions** database (see Fig. 3) contains data about the characteristics of different platforms, such as brand, model, screen sizes, keyboard type, etc. The platform-related characteristics are stored using a hierarchical structure, from more generic characteristics such as "Mobile" vs. "Desktop" to more specific ones such as "Apple iPhone 3G" vs. "Apple iPhone 3G S".

Likewise, business scenarios are stored in a hierarchical structure in the **Business Scenario Descriptions** database. An example for such a business scenario hierarchy is "Sales Order Management" ⇒ "Quotation" ⇒ "Web Auction Quotation". The **UI** repository stores different types of UI components, as outlined in Section 2:

- UI **Components** of different granularity, starting with
  - basic UI components such as text fields, labels, sliders
  - more complex generic composite UI components such as calendar pickers
  - composite UI components which are specific to a certain domain, such as a sales order management UIs or purchase order lists

– **Themes** in the form of stylesheets
– **Layout** information as views (e.g. a view representing a horizontal layout loops over an array of UI components and places them in a horizontal manner on the screen), and
– **Composition** and **Binding**

Contextual knowledge can stem from different parties such as hardware and software producers, UI developers, domain experts or internet communities [2]. How these data sources are used in rules is explained next; how the UI knowledge is acquired follows in Section 3.3.

3.2 Rule Specification

A key point of the approach is the use of knowledge acquisition method Ripple-Down Rules (RDR, [19]) which has been successfully applied to other domains such as pathology reports, soccer simulations, duplicate invoices, but never before to the domain of UI creation. We decided to adopt this approach as it (i) provides a simple and easy approach to knowledge acquisition and maintenance [16]; (ii) allows for the incremental creation of new rules while processing example cases.

For the specifications of rules, we make use of a Single-Conclusion RDR approach (SCRDR, [19]) which provides exactly one conclusion per context. This is combined with an approval process as used in CRDR (collaborative RDR)[19], and the possibility to actually change rule conclusions (in traditional RDR approaches, only exceptions to rules are allowed, rules can never be changed or deleted).

A **Rule** in our approach specifies which UI components should be combined to a screen for a given context. Each rule specifies a condition (the context), for which a conclusion (a partial UI suggestion) is derived. In our approach, rules are described in the following way: Conditions are based on two attributes, the business scenario (Sales Order Management, Supply Chain Management, etc.) and the platform (Desktop, Mobile, PDA, iPhone, etc.). These attributes represent the context. The conclusion relates to a *subset* of the four dimensions described above: Layout, Binding, Composition and Theme. *The combination of the conclusions of the most applicable rules for all four dimensions results in the suggested user interface.*

Fig. 4 depicts a number of rules, some with partial conclusions, in the knowledge base. Rule 0 contains the default conclusion where the business scenario and the platform chosen are not defined. In our prototype, we do not suggest a specific UI for that case, but propose an empty UI screen with some default layout and theme. The knowledge base shows true (except) branches and false (if not) branches. Starting at the top node, the inference engine tests whether the next rule node is true or false. If a rule node is true, the engine proceeds with the child nodes and tests if they are false or true. The last rule node that evaluates to true is the conclusion given. This is done for each dimension individually. The

[2] in our approach, we are making use of device characteristics stored in the WURFL [11] database, see 4.1

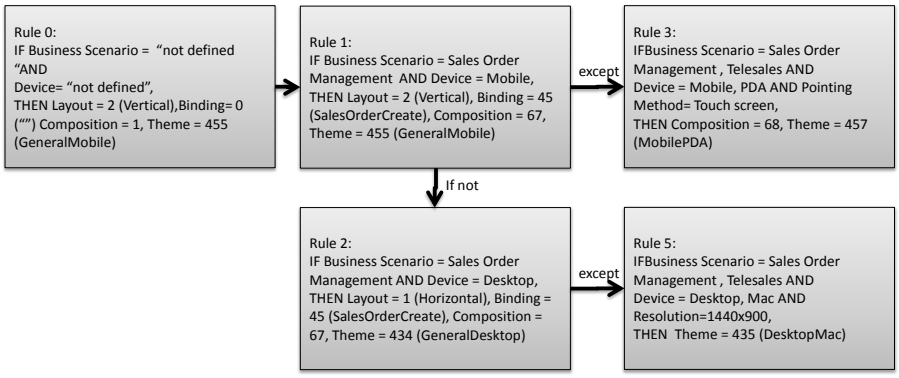

Fig. 4. Rules in our approach, featuring four dimensions in the conclusion

overall conclusion is then the combination of all partial conclusions. To give an example in Fig. 4, a UI developer wants to create a UI for the context "Business Scenario = Sales Order Management, Telesales AND Device = Mobile, PDA AND Pointing Method= Touch screen". The system finds a UI that was created for the context "Business Scenario = Sales Order Management AND Device = Mobile" (Rule 1). The developer modifies the UI suggested by Rule 1 by applying a new stylesheet for PDAs (affects the theme) and adding further UI components (affects the composition). An exception to rule 1 is created which results in rule 3. In a second scenario, the developer wants to create a UI for the context "Business Scenario = Sales Order Management AND Device = Desktop". Rule 1 is checked and results to "false". The system starts to check the rules down the "if not" path. Rule 2 results to true, and the UI with the dimensions "Layout = 1 (Horizontal), Binding = 45 (SalesOrderCreate), Composition = 67, Theme = 434 (GeneralDesktop)" is suggested to the user.

3.3 Incremental UI Knowledge Acquisition

In our approach, knowledge is added to the KB in the following situations:

(a) A new UI is created from scratch, for a given context.
(b) A UI suggested by the recommender system is modified for a given context.
(c) A UI is created/changed based on a new context.

Case (a) triggers the creation of a new rule for a context that was already stored in the database, but for which no associated UI existed. The KB is extended with a new component entry, and, if new layouts, themes, or new Web service bindings are used, these are stored in the respective databases.

In case (b), the rule condition would remain unchanged, but the conclusion would change. In our approach, such changes are not stored automatically: if an existing rule would be modified, it is unclear if this change would improve the KB content. Therefore, an approval process is triggered, where a suitable approver (e.g., development manager) has to confirm or reject the change.

Case (c) relates to a completely new context. Here, the condition and (likely) the conclusion of a rule are affected. This case arises when new devices are added

to the platform database, new business scenarios are added, or combinations of devices and business scenarios are chosen for which no knowledge exists (a new case is created and accordingly a new condition and rule is specified). If the new context is an extension to an existing context, an exception to a rule is stored. Otherwise, a new rule is stored in the if not branch of the rule tree.

After saving the UI in case (c), a difference list (similar to [19]) is presented to the user for every dimension of the UI, e.g. the UI developer is asked why the layout was changed, why the composition was changed, and a list of context-related differences is presented to her. The developer can flag the differences which were relevant for her decision to change the suggested UI. Relevant for a change in the layout might be e.g. that the UI was created for a PDA as opposed to a generic mobile device. The context specified as relevant by the user will be saved by the system as a new rule condition.

Our approach thus requires little user effort for maintaining the knowledge: Apart from specifying the rule conditions in a difference list, the user is not required to actively contribute to the rule base evolution – rules are created and stored by the system autonomously.

4 Evaluation

To evaluate the approach, we (i) developed a proof-of-concept prototype, and (ii) conducted an experiment simulating the development of UIs.

4.1 Prototypical Implementation

As a proof-of-concept of our approach, we built a prototype which implements the recommender system. The prototype instantiates the approach for a Visual Studio .Net MVC environment. User interfaces are rendered with the help of JavaScript and the libraries JQuery and JQuery Mobile. These technologies were chosen as they support the adaptation of UIs to a large number of different platforms. The prototype is implemented as a rich Web application, using JSON (JavaScript Object Notation) as a data-interchange format.

To fill the context side of the KB, we first integrated the Wurfl database [11] into our approach, which contains detailed technical descriptions of many devices and is updated on a regular basis. Second, we fed the KB with sample business scenarios derived from the SAP Business Suite CRM System via reverse-engineering.

The prototype's functionality is split into two main procedures: recommending UIs and acquiring knowledge. The recommendation starts with the user specifying the current context in a wizard. Using the hierarchies for the two context dimensions, the user selects from all available devices and business scenarios present in the repository. This is done with a set of dropdown boxes. The selected context information is sent back to the recommender system. As shown in Fig. 3, the system matches the context with the conditions of the available rules,

selects the best fit, and returns a UI suggestion. Technically, the UI suggestion is stored and transmitted as a HTML string.

If needed, the UI developer refines the suggested UI, deletes and adds elements, and possibly changes the theme, layout, or binding. Storing the changes triggers the knowledge acquisition procedure, depicted in Fig. 5. For each dimension, the system calculates difference lists as described in section 3.3.These are shown to the user, who selects the dimensions which were relevant for her decision to modify the UI. The case repository is updated with the new case, and the new UI is stored in the component repository. New rules and conclusions are derived from the difference lists and stored into the database for all four dimensions.

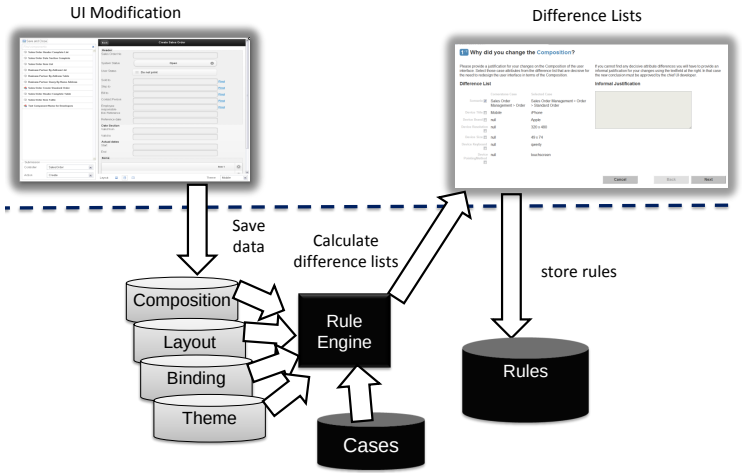

Fig. 5. Recommender system: knowledge acquisition (top: user; bottom: system)

4.2 UI Development Experiment

Experiment Setup. An important factor in any application of our approach is the amount of content and number of rules in the KB. On one end of the spectrum, the KB does *not contain any UI components* needed to create a user interface for a given context – all UI components have to be created from scratch first. In this situation, we hypothesis that our approach will not provide a benefit over conventional IDEs – it might even lead to a slightly increased effort. On the other end of the spectrum, *all UI components* required for the creation of a user interface are available in the KB. In this situation, we suspect that our approach is most helpful and leads to a significant increase in productivity. In all other situations, where only *some of the UI components* required are available, whereas others have to be created in order to complete the UI, we expect that an increase in productivity is observable – depending on the portion of UI components available, the improvement may range from minor to significant.

For the evaluation herein, we decided to focus on the end of the spectrum where UI components reuse enabled by using our approach should be most clearly observable – i.e., all UI components needed are already available, but no rules for the creation of UIs have been defined. In order to evaluate the added value of our approach in UI development productivity, we analyzed 115 UIs from the Customer Relationship Management system SAP CRM 700, specifically from high-level business scenarios including "Sales Order Management", "Service Order Management", and "Contracts/Service Contract Management". To facilitate the experiment, we reproduced the main screens for these highly complex UIs.

Using SAP CRM screens, our experiment was based upon a consistent set of designs with strong methodological design background. Such a setting is likely to occur in practice where development aims for UIs with a consistent look and feel throughout an application, thereby fostering learnability and usability (e.g. ERP and CRM software, e-government applications). The results of this experiment thus rather apply to applications where consistency of UIs is given or desired.

In the first step, we identified the UI components out of which the user interfaces were composed. This was done by comparing different UIs with each other, to detect common elements and identical screen parts – e.g., header elements, item lists, data sections. The UI components identified thereby were added to the KB of our prototype UISE. Furthermore, the business scenarios we encountered in the CRM system were integrated into UISE via reverse engineering. After this *boot-strapping* step, we reached a reasonably rich UI base. All UI components, themes, layouts and bindings to Web services required to create the UIs were available in the database – but no rules. By combining these elements to user interfaces during our experiment, new compositions, rules and new UI components would result.

In the following step, the *testing phase*, we recreated the SAP CRM screens with UISE. For the experiment, we varied the business scenarios, but kept a fixed platform (a generic mobile platform). By doing so, we simulated a large-scale UI development effort. Given the source of the screens, the simulation is strongly based on real-world application. Due to the fact that we recreated the screens, we knew what a correct result would look like. However, we cannot use this setup to realistically compare times taken for UI development. Instead, we counted the number of rules that had to be created during the recreation of all the 115 user interfaces with our approach. Only if the UI suggested by the tool already corresponded exactly to the desired UI, no changes were needed and accordingly no new rule was created. Likewise, we observed how the number of compositions evolved during the experiment. Since the experiment only involved recreating UIs for an SAP system and a fixed platform, the theme, layout and mapping were kept fixed. *An identical composition therefore indicates an identical UI.* Less new rules and less compositions thus mean less work for the developer. Therefore, both these numbers can be seen as an inverse measure of productivity.

Experiment Results. Figure 6 shows the number of rules that had to be created for the selected UIs over time. Over the course of the experiment, we can see a slight decline of the rules curve: for creating 115 UIs, 107 rules were added. That means, out of all suggested UIs, 8 were a perfect match. A much

more significant decline however is observable in the composition curve: the 115 UIs were reproduced with only 48 different compositions. Improvements through the reuse of UI compositions were clearly present in the experiment.

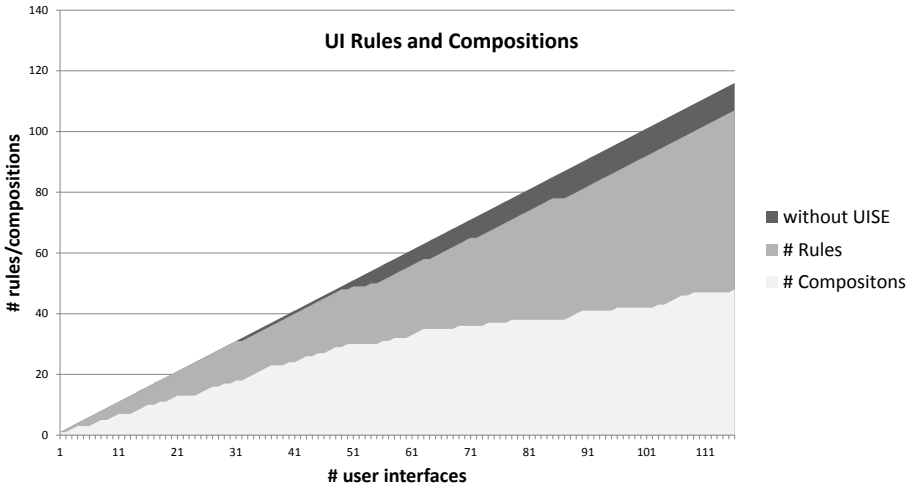

Fig. 6. Knowledge base size vs. number of UIs

On a more detailed level, we also counted the number of changes needed to recreate a UI. A user interface is suggested to the developer according to the context chosen. If the suggested UI corresponds already to the UI she wanted to create, no changes are required. Otherwise suggested UI elements have to be replaced or eliminated, or new elements added. We counted each add/delete action as one change, thus replacements were counted as two changes. Fig. 7 shows the results of this investigation for each of the high-level business scenarios.

Depending on the specific scenario, Sales Order UIs consist of 38-71 simple UI elements (like text boxes, labels and scroll-bars), contracts require 52-96 simple UI elements, and Service Orders between 54 and 108.

The frequency with which how many changes were needed using our prototype, UISE, are shown in 7. For all three scenarios, there were cases where zero changes were required – i.e., exact hits of the recommendation. For Sales Orders (see light grey columns in Fig. 7) up to 6 changes were needed in UISE, with an average value of 2.6 steps. Contracts (white columns) also required up to six steps; the average was at 3.6 steps. Service orders (dark grey columns) required up to 12 steps, with an average of 5.6 steps. Given the complexity of the target UIs, the number of changes required with UISE seems relatively minor.

5 Related Work and Discussion

In the following section, we will review different approaches in research related to the UI development. We differentiate between approaches dealing with the

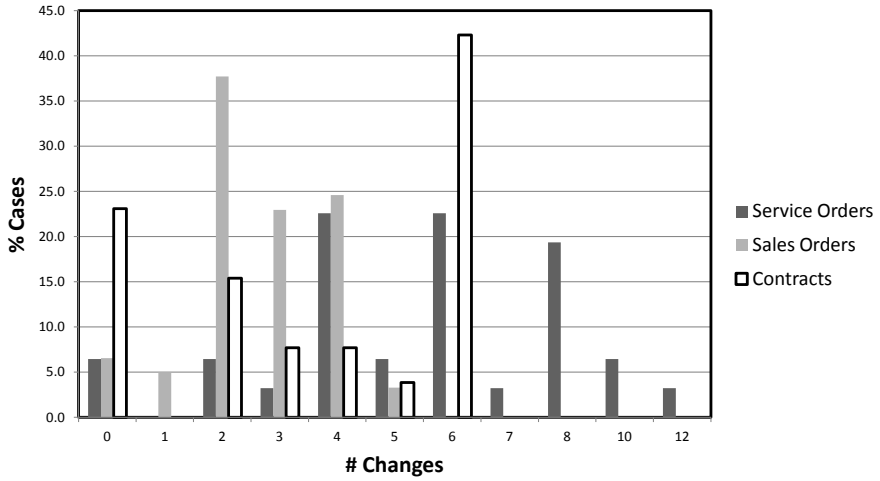

Fig. 7. Changes for Sales Order, Contract, and Service Order UIs

creation of user interfaces for Web services, and approaches which adapt UIs to various contexts.

GUI Development for Web Services. In this paragraph, we contemplate GUI development approaches for Web services in general, which do not take contextual factors of any kind into account. The ServFace project aims at enabling the non-technical user to create applications from annotated Web services (see [20], [9], and [5]) . The ServFace project aims at a multi-channel approach and provides platform-independent development models. [5] supports the creation of user interfaces as well as the modelling of user interactions in task-driven software development approaches. Like in [9], the concept is based on the introduction of annotations. Annotations are as well used in the Dynvoker tool [21]. All named works related to ServFace lack a feature which is provided in our approach: Context in terms of business scenarios is not taken into account. There is no guideline provided to the user how a UI has to look like in certain scenarios. [15] tackles the creation of user interface from independent, loosely coupled modules with a framework based on ontologies. It aims at reducing the complexibility of large UI development projects via modularization. The FAST project [8] establishes a concept for enterprise mashups. In contrast to traditional solutions, FAST does not allow for the mash up of heterogeneous data sources, but proposes a screen flow design resulting in so-called gadgets. A FAST ontology is introduced to describe the characteristics of complex gadgets (graphical elements, user interaction models, data flows), thereby the approach partly resembles the work presented in [15]. The disadvantage of these approaches is that the ontology has to be created before the solution can be used. Our approach

is based on an incremental acquisition of knowledge, starting with an empty knowledge base which is gradually built up.

GUI Development and Context-Awareness Various attempts to integrate UIs and context-awareness have been made in different research projects. The approaches are differing in terms of which type of context is taken into account and which devices they refer to. Context-awareness for mobile UIs during runtime is realized in [3], and during design time in [12]. In [3], user interfaces of mobile devices are automatically adapted to user context-changes, as well as to changes to different screen resolutions or orientations. UIs adapts to specified user contexts by the use of style sheets. [12] generates UI containers for mobile devices and adapts UIs to platform specific characteristics and user tasks making use of heuristics. [10]'s work aims at providing native user interfaces to mobile devices based on the context of the user, whereas the context-awareness in this work is not task-driven like in [27] or [12], but refers to environmental factors and user data. An approach not restricted to mobile devices called "CRUISe" [17] aims at the dynamic composition of user interfaces via Web services. The work differs from the other proposals presented insofar as it not making use of annotations like [9] , and is not based on predefined structures or preindexed documents, but is encapsulating generic, reusable web UI components as so-called User Interface Services. All approaches named rely on hard-coded rules development and do not support incremental UI reuse rules capture. There are assumptions made about what users prefer, and all knowledge about the system behaviour has to be captured before the implementation. Our approach provides more flexibility, as user interfaces can be adjusted to varying needs and preferences, and rules can be extended or changed accordingly. Our work is related to the multi-target UI framework presented by Calvary et al. [4], in that we provide an automatic method for Calvarys forward engineering step, using an RDR-based knowledge base. [7] adapts user interfaces to tasks and context. The context considered in this work comprises platform, user (such as user profile, level of knowledge) and environmental (noise level, luminosity) factors. Users' tasks point to related business components. Task-driven UI adaptation can be realized together with changing the UI according to different user contexts. In difference to our approach, the association between context and UI patterns is however fix, there is no way to change relationships between entities during run time.

6 Conclusion

In this paper, we presented a novel framework and a tool for reusing UI components in UI development environments. The approach has the following main characteristics. (1) It enables the automatic creation of UI proposals for (Web service-based) applications in a given context. The suggestions stem from previous developments. (2) The approach is based on ripple-down-rules (RDR), which allows for incremental acquisition of knowledge in a rule base. The creation of rules is performed semi-automatically, during development. (3) UI suggestions are matched to varying contexts by combining UI components, applying different

style sheets, changing the layout, and selecting the binding of UI fields to Web services. We implemented the approach in a prototype. Using the prototype, we simulated the development of 115 UIs from SAP's CRM system. By doing so, we evaluated qualitatively if using our approach is likely to result in productivity gains. The results from the experiment indicate that productivity increase from UI reuse is possible, and our approach is a step in the right way.

In future work, we would like to investigate if suggesting multiple user interfaces for a given context would be beneficial, and how an efficient combination of aspects from different suggestions can be achieved. Furthermore, we would like to evaluate our approach for mobile applications, and for the variation of multiple dimensions of context (business scenarios and platforms).

Acknowledgements

We would like to thank Jan-Felix Schwarz for his implementation work on the UISE prototype. The assistance and advice provided by Prof. Paul Compton in regards to RDR technologies was greatly appreciated. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. AppBrain. Number of available android applications, December 2012. `http://www.appbrain.com/stats/number-of-android-apps`.
2. Apple. Made for iPad. Ready for anything., March 2012. `http://www.apple.com/au/ipad/from-the-app-store/?cid=wwa-au-kwg-ipad-00001`.
3. T. Butter, M. Aleksy, P. Bostan, and M. Schader. Context-aware user interface framework for mobile applications. In *Distributed Computing Systems Workshops, 2007. ICDCSW '07.*, 2007.
4. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computer*, 15,3:289–308, 2003.
5. M. Feldmann, G. Hubsch, T. Springer, and A. Schill. Improving task-driven software development approaches for creating service-based interactive applications by using annotated web services. In *Proceedings of the 2009 Fifth International Conference on Next Generation Web Services Practices*, NWESP '09, pages 94–97, Washington, DC, USA, 2009. IEEE Computer Society.
6. Garnter. Hype cycle for context-aware computing. Technical report, Gartner, 23 July 2009.
7. A. Hariri, D. Tabary, S. Lepreux, and C. Kolski. Context aware business adaptation toward user interface adaptation. *Communications of SIWN*, 3:46–52, 2008.
8. V. Hoyer, T. Janner, C. Schroth, I. Delchev, and F. Urmetzer. FAST platform: A concept for user-centric, enterprise class mashups, 25-3-2009 2009.
9. P. Izquierdo, J. Janeiro, G. Hubsch, T. Springer, and A. Schill. An annotation tool for enhancing the user interface generation process for services. In *Microwave Telecommunication Technology, 2009. CriMiCo 2009.*, 2009.

10. T. Lange. *Dynamic Service Integration for Applications on heterogeneous Mobile Devices*. Master thesis, TU Dresden, 2009.
11. S. K. Luca Passani. Wurfl, 2011. `http://wurfl.sourceforge.net/`, accessed in April 2010.
12. F. Martinez-Ruiz, J. Vanderdonckt, and J. Arteaga. Context-aware generation of user interface containers for mobile devices. In *Computer Science, 2008. ENC '08.*, 2008.
13. N. McAllister. Mobile UIs: It's developers vs. users, 2012. `http://www.infoworld.com/d/application-development/mobile-uis-its-developers-vs-users-184472`.
14. C. McMillan. Searching, selecting, and synthesizing source code. In *ICSE '11: 33rd International Conference on Software Engineering*, 2011.
15. H. Paulheim. Ontology-based modularization of user interfaces. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '09, pages 23–28, New York, NY, USA, 2009. ACM.
16. P.Compton, L.Peters, G.Edwards, and T.G.Lavers. Experience with ripple-down rules. *Knowledge-Based System Journal*, 19(5):pp. 356–362, 2006.
17. S. Pietschmann, M. Voigt, A. Rmpel, and K. Meiner. CRUISe: Composition of rich user interface services. In *ICWE'09: Intl. Conf. Web Engineering*, volume Vol. 5648/2009, pages pp. 473–476. Springer Berlin / Heidelberg, 2009.
18. S. P. Reiss. Semantics-based code search. In *ICSE '09: 31st International Conference on Software Engineering*, 2009.
19. D. Richards. Two decades of ripple down rules research. *The Knowledge Engineering Review*, 24:159–184, 2009.
20. ServFace. Service annotations for user interface composition, 2010. `http://www.servface.org/`.
21. J. Spillner, M. Feldmann, I. Braun, T. Springer, and A. Schill. Ad-hoc usage of web services with Dynvoker. In *ServiceWave '08*, pages 208–219, 2008.
22. The jQuery Foundation. jQuery, 2012. `http://jquery.com/`.
23. TOGAF. Business scenarios, 2006. `http://www.opengroup.org/architecture/togaf7-doc/arch/p4/bus_scen/bus_scen.htm`.
24. Twitter. Bootstrap. Sleek, intuitive, and powerful front-end framework for faster and easier web development., 2012. `http://twitter.github.com/bootstrap/`.
25. Wikipaedia. Visual component library, August 2012. `http://en.wikipedia.org/wiki/Visual_Component_Library`.
26. J. Yu. *A UI-driven Approach to Facilitating Effective Development of Rich and Composite Web Applications*. Doctorial thesis, Univ. of New South Wales, 2008.
27. L. Zhang, B. Gong, and S. Liu. Pattern based user interface generation in pervasive computing. In *Third International Conference on Pervasive Computing and Applications*, volume 1, pages 48–53, Oct 2008.