UI Creation Patterns (using iTasks for DSL → GUI transformation)

Michaela Bačíková* and Jaroslav Poruban†
* † Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
{michaela.bacikova*, jaroslav.poruban†}@tuke.sk

Abstract—Domain-specific languages (DSLs) are becoming more and more popular. However, there is still a small number of DSLs when comparing to the number of existing applications. The results of our previous research showed that it is possible to speed up the DSL development process by aiding the first development phases (design and implementation). More specifically it is possible to generate DSLs from existing GUIs of component-based applications. We claim that based on these generated DSLs and their models, it is possible to generate new user interfaces or even whole new applications. To verify this claim, in this paper we use existing technologies which simplify the creation of web applications: iTasks. We also describe stereotypes of creating GUIs which we used to extract data from existing applications and to generate new applications. In the last part of this paper we limit the types of applications, which can be used for extraction.

I. INTRODUCTION

Domain-specific languages (DSLs) are computer languages tailored to a specific application domain [1], [2], [3]. Currently, DSLs are becoming more and more popular. However, there is still a small number of DSLs when comparing to the number of existing applications. Despite their advantages, the cost of developing a new DSL is usually high [1] because it involves development of language parsers and generators along with the language. The implementation phase of DSL development is well documented by many researchers [2] but the analysis and design phases are still dropped behind. The various DSL development phases and the tool support of each of them is described in the article by Čeh et al. [4].

In our previous work [5] we showed that it is possible to speed up the DSL development process by aiding the first development phases (design and implementation). More specifically it is possible to generate DSLs from existing GUIs of component-based applications.

In this paper we claim that based on these generated DSLs and their models, it is possible to generate new user interfaces or even whole new applications. To verify this claim we use existing technologies which simplify the creation of web applications: iTasks. We performed an experiment in which the DSLs and their models are automatically generated by our DEAL tool [6]. We implemented a new module into DEAL, which automatically generates new iTask application based on a previously generated DSL. Through this experiment we aim to prove the possibility of generating a new UI based solely on domain knowledge extracted from an existing application.

In addition, in this paper we describe stereotypes of creating graphical user interfaces which are used to extract data from existing applications and to generate new applications. Some stereotypes were identified in our previous works, some were identified during the work with the iTask system.

We realize that not every existing application can be used to extract domain information in the manner we describe in this paper. Therefore, in the last part of our paper we limit the types of applications, which can be used for extraction to ensure the highest possible degree of automation and the lowest amount of manual work needed after the generation process is finished.

A. Tasks and Goals

In our previous research we strived for verifying the validity of the following hypothesis:

(H1) It is possible in an automatized manner to extract a DSL from the interface of an existing component-based application.

The validity of this hypothesis was shown on an experiment [5]. In this paper, we define a new hypothesis based on the previous one:

(H2) It is possible to automatically generate a new GUI (or even a whole new application) from the previously extracted DSL.

A question is connected both with hypothesis H1 and H2:

(Q1) To what extent it is possible to meet the hypotheses H1 and H2? How relevant are the results?

We try to satisfy the primary goal of this paper, which is to show the validity of the hypothesis H2 and to answer the question Q1, that means to define the boundaries, for which H1 and H2 will definitely be fulfilled. We assume the results will be strictly dependent on the application to be analysed (target application), namely on the following criteria:

• the programming style of the target application,
• the presence of components in the GUI of the target application,
• the types of components in the GUI of the target application,
• the presence of domain terms in the GUI of the target application.

Based on the defined goals the following tasks arise:
• To design and implement a generator of GUIs. The input of the generator will be the DSL generated by the DEAL method and the output will be a new generated GUI.
• To perform a series of experiments on existing component-based applications using the implemented generator.
• To evaluate the results and to create a list of guidelines and restrictions for applications, which can be used for the extraction and transformation process of DEAL.

II. THE DEAL METHOD

We have performed a research in the area of automated domain analysis of UIs, described in detail in [7] and [8]. Our method for extraction of domain information from existing GUIs is called DEAL (Domain Extraction ALgorithm). The input of DEAL is an existing system programmed in a language, which provides the possibility of determining the component structure (introspection), reflection and/or aspect-oriented programming. The output of DEAL is a domain model and a DSL of the target application. The DEAL traversal algorithm was described in [8] and in [5] we introduced the method of the GUI \( \rightarrow \) DSL transformation.

The domain content of the target UI is extracted as a graph of terms, their relations and properties. We experimentally confirmed the possibility of extraction on Java and HTML applications ([8]). Both languages meet the presumption to have the component nature and provide the possibility to determine the component structure. We also experimentally confirmed the possibility of extracting a DSL based on the extracted domain model [5] on Java applications.

III. THE DEAL TOOL PROTOTYPE

The DEAL tool prototype\(^1\) is an implementation of the DEAL method on Java applications. Currently, DEAL uses YAJCo\(^2\) language processor to generate grammars. More about YAJCo can be found in [9].

The DEAL tool prototype proves that it is possible to:

• traverse the GUI of an existing Java application,
• extract domain information in a formalized form (domain model),
• generate a DSL grammar, model, and parser based on the domain information.

The DEAL prototype was tested on 17 open-source Java applications. Some of them are included in the DEAL project online. It is however still in development and we are improving it based on the test results.

An example of the DEAL output for the person form (fig. 1) is the grammar generated from the extracted model:

\[
\begin{align*}
Person &::= (\langle Person \rangle Name \ Surname \ Age \ Gender \ (Favourite\_color)*) \\
Name &::= (\langle Name \rangle \langle STRING\_VALUE \rangle) \\
Surname &::= (\langle Surname \rangle \langle STRING\_VALUE \rangle) \\
Age &::= (\langle Age \rangle \langle INT\_VALUE \rangle) \\
Gender &::= (\langle Gender \rangle (\langle man \rangle | \langle woman \rangle)) \\
Favourite\_color &::= (\langle Favourite\_color \rangle (\langle red \rangle \langle blue \rangle \langle green \rangle \langle yellow \rangle))
\end{align*}
\]

The grammar is in the YAJCo notation and the rules for not-mutually exclusive terms (colors) are supplemented by the \(0 \leq n\) version because YAJCo tool does not support the "?" operator. However the results show that generating DSL grammars from existing UIs is definitely possible.

IV. DSL \( \rightarrow \) GUI TRANSFORMATION

The transformation process is described in the fig. 2. The DEAL tool runs an existing Java application, traverses its components and extracts its domain model and a DSL. Based on the extracted DSL, a new application will be generated. The generated application is written in iTask.

A. Why iTasks?

The iTask system (iTasks) is a task-oriented programming toolkit for programming workflow support applications in Clean [10]. Workflows consist of typed tasks that produce results that can be passed as parameters. From these iTask specifications, executable workflow systems are automatically generated. An example of a iTask code for a Person form is in lst. 1.
B. Identified stereotypes

In our previous paper [8] we identified a number of stereotypes of creating UIs. Heuristic transformation rules for generating DSLs were later created based on these stereotypes [5]. In this section we will summarize all previously identified stereotypes and introduce a few new, identified during the work with the iTask System. Based on this list, we will define new transformation rules in the future, and then we will include them in the DEAL prototype. The list of stereotypes follows:

1) **Hierarchical arrangement** of components is usually very similar to (or the same as) the hierarchy of entities and properties in the domain model.

2) **Labels** have two functions:
   - determine the purpose of other components by labelling them,
   - display data.

3) For data entry, usually **textual**, or other components are used. They can be of different types and they can have restrictions depending on the type of information entered (or it’s length).
   a) Standard **text fields** are used to enter information of type **STRING**.
   b) To enter information of type **STRING** with greater length and containing new lines, **text areas** are usually used.
   c) To enter numeric values, **spinners** or **text fields with number restriction** are used.
   d) To enter passwords, **encoded text fields** are used.
   e) To enter a single selection from multiple options, **combo boxes, lists and radio buttons** are used.
   f) To enter a multiple selection from multiple options, **lists and check boxes** are used.
   g) To enter dates, **spinners, text fields with restriction to dates** and **special components of type calendar** are used.
   h) To enter a geographical location a map component is used (e.g. Google Maps).
   i) To enter a selection from a tree structure, a **tree component** is used.
   j) To restrict the data, restrictions on components are set:
      - upper and lower limits,
      - a list of values.

4) Forms usually contain **OK, Cancel and Reset buttons**, which represent the functionality of form confirmation, cancelling of entering values or cleaning up the form values.

5) Applications, which provide an additional functionality, use additional functional components such as **buttons or menu items**.

6) **Buttons** are usually located in dialogs or windows and the result of clicking on a button is an execution of a functionality, which is predefined on the button in the code and a possible change of the application state.

7) **Menu items** are usually structured in menus and like buttons, the result of clicking on a menu item is an execution of the predefined functionality and a possible application state change.

8) Functional components can trigger opening a new window/dialog or closing an existing window/dialog.

9) **Tab panes** are used to structure window content. Each tab has its name which identifies a sub-domain and all items located in the particular tab belong to this sub-domain.

Fig. 3. The example of a person form generated by the iTask System.

Listing 1. Example of a Person form written in iTask

```iTask
:: MyPerson =
  { name :: String,
    , surname :: String,
    , age :: Int,
    , gender :: MyGender,
    , favouriteColors :: MyColors
  }

:: MyGender = Male | Female
:: MyColors = {blue::Bool, white::Bool, yellow::Bool, green :: Bool}

derive class iTask MyPerson, MyGender, MyColors

taskPerson :: Task MyPerson

enterPerson = enterInformation
  "Enter your personal information:" []

The model of the application is represented by the MyPerson type along with its subtypes MyGender and MyColors. The task `enterPerson`, after adding it to the iTask System workflow, generates a web form which is displayed in fig. 3.

As can be seen in the example, the same relations and properties extractable by DEAL can also be expressed in iTask:

- **data types**: expressed by the task type or by the data type property,
- **mutually-exclusive relation**: expressed by alternatives (e.g. in the MyGender type),
- **mutually-not-exclusive relation**: expressed by a Set (e.g. in the MyColors type).

Similarly, a view of the person can be implemented (see listing 2).

Listing 2. Person view example in iTask

```iTask
viewPerson :: Task MyPerson

viewPerson = viewInformation "This is the person:" []
```

The provided examples clearly show the simplicity of writing a simple entry (or display) form only by providing a domain model and a few lines of additional code. Now we are only a small step from generating iTask applications: the domain model is automatically provided by DEAL, only its notation has to be transferred into the iTask notation.

---

**Note:** The code snippet provided in the image seems to be incomplete and contains multiple tasks, which might not be fully typed as expected. The listing numbers and some annotations are also not perfectly aligned with the description. The content is focused on summarizing and identifying stereotypes for UI creation, and the examples are presented in the context of the iTask System, a tool for generating web forms and applications based on a domain model.
To express some of the stereotypes mentioned above, iTasks contain the following predefined structures:

- **String** – for textual information,
- **Int** – for numeric information,
- **Date** – calendar for entering dates,
- **Note** – for a long textual information,
- **"\|"** operator – for a single selection,
- **Set** – to represent a list of values for multiple selection or to restrict the minimum or maximum value of data,
- **GoogleMaps** – for geographical location,
- **Programmer defined types** – to express structured information.

The layout of the resulting forms is automatically created by iTasks, along with default buttons (if they are necessary). In addition, iTasks also provides a possibility to add programmer defined functionalities by creating Action tasks, results of which are menu items or buttons. OK, Cancel or Reset buttons are generated automatically, if the form requires them.

V. EXPERIMENTS AND RESULTS

We created a prototype of the transformation module described in section IV for the purpose of experimental verification of the hypothesis H2 defined in the introduction. The transformation module was integrated into the DEAL prototype. The input of the transformation module is a DSL and domain model generated by DEAL. The output is an iTask application. We verified the transformation module on all 17 Java applications which are included in the DEAL prototype. The results can be summarized as follows:

- For the standard components, the transformation produced expected results.
- If the target application contained special components, they were not extracted and therefore they were not present in the generated iTask application. Additional programming would be needed to include the support for non-standard components into DEAL.
- The functionality of user defined buttons and menus could not be extracted and transformed into the iTask application.
- Components without any identifier (i.e. label or tooltip) were not extracted. If they represented a grouping node, a term with a randomly generated name (e.g. "Unknown123456") was created instead.
- The best results were produced from applications, which looked like simple forms, with a properly defined terminology and no special functionalities and components.

Based on the listed results we can conclude, that the highest possible degree of automation and the lowest amount of manual work after the generation process is finished can be achieved by selecting the target application which would:

- contain only standard components,
- have properly defined domain model,
- look like a simple form or dialog (or set of forms and dialogs) with standard functionalities (OK, Cancel).

The above defined characteristics represent recommendations for applications, which can be used as an input of DEAL. Other applications can be analysed and transformed too, but much more manual work will be needed after the transformation.

VI. RELATED WORK

Here we briefly summarize the different approaches related to: domain analysis, ontology extraction, GUI modeling, semantic UIs and reverse engineering.

1) Domain Analysis: The domain analysis (DA) was first defined by Neighbors [11] in 1980 and he stresses that DA is the key factor for supporting reusability of analysis and design, not the code.

The most widely used approach for domain analysis is the FODA (Feature Oriented Domain Analysis) approach [12]. FODA aims at the analysis of software product lines by comparing the different and similar features or functionalities. The method is illustrated by a domain analysis of window management systems and explains what the outputs of domain analysis are but remains vague about the process of obtaining them. DREAM (Domain Requirements Asset Manager) approach by Mikeyong et. al. [13] is similar to FODA, but with the difference of using an analysis of domain requirements, not features or functionalities of systems. Many approaches and tools support the FODA method, for example Ami Eddi [14], CaptainFeature [15], RequiLine [16] or ASADAL [17]. Other examples of formal methodologies are ODM (Organization Domain Modeling) [18] and DSSA (Domain Specific Software Architectures) [19].

There are also approaches that do not only support the process of domain analysis, but also the reusability feature by providing a library of reusable components, frameworks or libraries. Such approaches are for example the early Prieto-Díaz approach [20] that uses a set of libraries; or the later Sherlock environment by Valerio et. al. [21] that uses a library of frameworks.

The latest efforts are in the area of MDD (Model Driven Development). The aim of MDD is to shield the complexity of the implementation phase by domain modelling and generative processes. The MDD principle support provides for example the Czarnecki project Clafer [22] and the FeatureIDE plug-in [23] by Thüm and Kästner.

ToolDay (A Tool for Domain Analysis) [24] is a tool that aims at supporting all the phases of domain analysis process. It has possibilities for validation of every phase and a possibility to generate models and exporting to different formats.

All these tools and methodologies support the domain analysis process by analysing data, summarizing, clustering of data, or modelling features. But the input data for domain analysis (i.e. the information about the domain) always comes from the users, or it is not specified where it actually comes from. Only the DARE (Domain analysis and reuse environment) tool from Prieto-Díaz [25] primarily aims at automatic collection and structuring of information and creating a reusable library. The data is collected not only from human resources, but also automatically from existing source codes and text documents.

But as mentioned above, the source codes do not have to contain the domain terms and domain processes. The DARE tool does not analyse the GUIs specifically.
Last but not least, the approach most similar to ours is the one proposed by Čeh et al. [4]. They proposed a methodology of transforming existing ontologies into DSL grammars and they present the results of their Ontology2DSL framework. The disadvantage in comparing to our approach is the little amount of existing ontologies available when comparing to the amount of existing software systems.

2) Ontology Extraction: Many approaches are targeted to ontology learning. Results are almost always combined with a manual controlling and completing by a human and as an additional input, almost always some general ontology is present (a core ontology) serving as a guideline for creating new ontologies. Different methods are used to generate ontologies:

   i) clustering of terms [26], [27],
   ii) pattern matching [28], [26], [29], [27],
   iii) heuristic rules [28], [29], [27],
   iv) machine learning [30],
   v) neural networks, web agents, visualizations [27],
   vi) transformations from obsolete schemes [29],
   vii) merging or segmentation of existing ontologies [31], [26],
   viii) using fuzzy logic to generate a fuzzy ontology, which can deal with vague terms such as FFCA method [32] or FOGA method [33],
   ix) analysis of web table structures [34], [28],
   x) analysis of fragments of websites [35].

3) GUI Modelling and Semantic User Interfaces: Special models are designed specifically for modelling UIs or for modelling the interaction with UIs, whether they are older, such as CLASSIC language by Melody and Rugaber [36], or modern languages, such as XML, described in the review made by Suchon and Vanderdonckt in [37]. Paulheim [38] designed UI models of interaction with users. For UI configurations usually models such as configuration ontology designed for WebProtg tool in [39] are used.

The most complex UI model was designed by Kösters in [40] as a part of the modelling process of the FLUID method for combined analysing of UIs and user requirements. A part of Kösters model is a domain model and model of UI (UIA-Model). Our model was slightly inspired by Kösters work - however we use domain-specific modelling without the relation to user requirements, therefore our model is simpler.

An interesting work was made in the area of semantic UIs by Porkoláb in [41].

4) Reverse Engineering: Only a few works in the area of reverse engineering will be mentioned since our work is primarily focused on the area of domain analysis, not on reverse engineering. However, there are several works closely related to our work. Specifically, they are either reverse processes compared to ours (i.e. generate GUIs from domain models), or they produce other outputs than a DSL grammar:

   • a GUI-driven generating of applications by Luković et al. in [42],
   • generating of UIs based on models and ontologies by Kelschev and Gribova in [43],
   • deriving UIs from ontologies and declarative model specifications by Liu et al. in [44],
   • program analysing and language inference [45].

A very interesting process is also seen in [46] where authors transform ontology axioms into application domain rules however the results are not as formal as our DSL grammar.

VII. Conclusion

In our previous research we proved the possibility of generating DSLs from existing GUIs of component-based applications. Based on the DSLs, it is possible to generate new user interfaces or even whole new applications, which we showed in this paper. We designed, implemented and experimentally verified a DSL → GUI transformation module. As an input we used the outputs of our current project, DEAL: DSLs generated from UIs. To create the output, we used existing technologies which simplify the creation of web applications: iTasks. Through experimental verification we aimed to prove the possibility of generating a new UI based solely on domain knowledge extracted from an existing application. We also listed a number of previously identified and new stereotypes, which were used to create transformation rules for the DEAL extractor and generator. Although the generation process provided expected results for a form-based applications, for more structured applications with special components and programmer defined functionality an additional manual work is needed after the generation process. In the last part of the paper we limited the types of applications, which can be used for extraction to ensure the highest possible degree of automation and the lowest amount of manual work after the generation process is finished. In the future we plan to improve DEAL to support more component types based on the stereotypes defined in this paper. Primary, the DEAL tool will be used to evaluate domain usability of existing user interfaces.

Acknowledgment

This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the Artifacts Written in Domain-specific Languages Driven by Language Evolution.

References
