Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 963–969

Rapid Application Prototyping for Functional Languages

Martin Podloucký
Department of Software Engineering,
Faculty of Information Technology,
Czech Technical University,
Prague, Czech Republic
Email: martin.podloucky@fit.cvut.cz

***Abstract*—This work addresses the problem of automated graphical user interface generation for functional programs in relation to rapid application prototyping. First an analysis of current state in the field of automated GUI generation is performed. Based on the analysis, the concept of *functionally structured user interface* (FSUI) is introduced. Meta-data system for code annotation is then specified for the Clojure programming language and a transformation from this system to FSUI data model is implemented. Finally, a graphical layer for displaying the actual interface is implemented in Clojure.**

I. Goal

THE focus of this work is to investigate the idea of automated GUI generation for functional programs for application in software prototyping. The main goals are:

1) Analyse the current state in the field of automated GUI generation.
2) Based on the analysis, explore possible approaches to GUI generation for functional languages.
3) Design and implement an automated GUI generator for functional programs.
4) The generator should create the GUI from annotated source code of the program.
5) Both the generator and the GUI should be implemented in functional language.

II. Paper structure

Firstly, the basic concepts, such as application prototyping and functional programming along with the motivation for this work, are explained in section III. Then, the Clojure programming language is introduced in section IV. Analysis of current state in the field of GUI generation is performed in section V. Some possible approaches to a solution of the stated problem, based on results of the analysis, are described in section VI. Finally, the concept of functionally structured user interface is presented in section VII.

III. Introduction

Software engineering industry is evolving and expanding rapidly, constantly searching for better techniques and technologies allowing software to be created faster with lower cost and better resulting quality [1]. Great desire for innovations can be observed in the field of business applications where the cost of the development is a key factor [2]. In this domain, the advantageous fact is that business applications have many common characteristics. They are based on similar principles, work with similar data and they have to deal with alike limitations [2]. This situation creates a good opportunity to use techniques such as prototyping [3] and generative programming [4].

A. Prototyping and generative programming

Application prototyping focuses on creating incomplete versions of the software being developed [3]. The purpose of such prototypes may be to explore possible solutions or to provide a piece of working software to the customer for evaluation in early stages of development [3]. There are several approaches to prototyping [5]: some prototypes are developed only to be discarded after they served their purpose. Such approach is called *throwaway prototyping* or sometimes *rapid prototyping*. Another approach is to incrementally evolve the prototype to fully functional product. This approach is often called *evolutionary prototyping*.

The main concern of either approach is being able to create prototypes as fast as possible [3]. Some techniques that can be used to shorten the prototype development time may be those of generative programming [4] and model driven development [6]. When significant parts of prototypes can be generated from conceptual models or from annotated source code, developers may focus more on implementing application logic rather then, for instance, implementing graphical user interface.

Graphical user interface (GUI) seems like a good candidate for a functionality to be generated out of annotated source code. GUIs used in prototypes do not need to meet too strict requirements on user friendliness since their aim is primarily to be suitable for presentation to the customer or for quick testing of basic functionality of the demanded software [3]. Furthermore, generating the GUI from source code may help to shorten the development time since there is minimum additional and external information needed to generate the GUI. According to [7] the source code already contains enough information to create properly working GUI.

978-1-4673-4471-5/$25.00

B. Functional languages

The family of Lisp languages such as Common Lisp [8], Scheme [9], Clojure [10] and others are hereby taken as functional languages. This work is focused on generating GUI from source code written in such a functional language. There are several reasons for choosing functional languages rather than imperative ones as focus of this work.

1) Code in functional languages has simpler structure then a code written imperatively [11], [12]. Since thorough analysis of source code is needed this decision simplifies algorithms used in GUI generation.
2) Functional languages are growing in popularity even for writing business applications [13].
3) There is a number of similar tools for object-oriented languages such as Java or Smalltalk (discussed in section V). It is interesting to investigate which of its functionalities and approaches could be used in the functional world.

IV. The Clojure language

Clojure is a relatively new functional language based on Lisp [10]. It was created by Rich Hickey whose goal was to create mainstream functional language which can compete and complement present mainstream object oriented languages [10]. Basic features and characteristics of Clojure according to Rich Hickey [10] are

Functional programming
: The philosophy behind Clojure is that most parts of most programs should be functional, and that programs that are more functional are more robust compared to programs written imperatively. Clojure provides the common functional tools, however, doesn't force the program to be referentially transparent.

Lisp
: Clojure is a member of the Lisp family of languages, extending the code-as-data system beyond parenthesized lists (s-expressions) to vectors and maps.

Dynamic Development
: Programming in Clojure is interactive. It is not a language abstraction, but an environment, where almost all of the language constructs are reified, and thus can be examined and changed.

Hosted on the JVM
: Clojure is designed to be a hosted language, sharing the Java virtual machine (JVM) type system, garbage collector, threads etc. It compiles all functions to JVM bytecode. Clojure has simple syntax to reference and create Java classes therefore it can easily interoperate with Java and its libraries.

Runtime Polymorphism
: Clojure supports polymorphism at 3 levels. First, almost all of the core infrastructure data structures in the Clojure runtime are defined by Java interfaces. Second, Clojure supports the generation of implementations of Java interfaces. The final and primary language construct for polymorphism is the Clojure multimethod.

Concurrent Programming
: Since the core data structures are immutable, they can be shared readily between threads. However, it is often necessary to have state change in a program. Clojure allows state to change but provides mechanism to ensure that, when it does so, it remains consistent, while alleviating developers from having to avoid conflicts manually using locks etc. This is achieved by using the software transactional memory system (STM) [14].

Clojure is chosen here to represent a Lisp-like functional language. This work therefore focuses on generating GUI from source code written in Clojure. Clojure was chosen as a representative language for several reasons

1) Clojure can easily interoperate with Java and so the GUI can be created using standard Java GUI libraries such as Swing [15].
2) Clojure has very robust mechanism for state manipulation which allows to write the code for the GUI in Clojure itself.
3) Clojure is more suitable for writing business applications than other commonly used Lisps [13]. Therefore, it makes more sense to create rapid prototypes with graphical interfaces especially in Clojure.

V. Current methods for GUI generation

This section analyses current state in the field of automatic GUI generation. The author of this work did not find any current implementation of automated GUI generation for functional languages. Several implementations exist for object oriented languages such as Java, C# or Smalltalk. There is also work [7] which explores automated GUI derivation from programs written in term rewriting systems [16]. Let's first focus on object oriented frameworks.

Most of the object oriented GUI generators share a similar paradigm which is based on annotating domain data model using some kind of meta-data. This annotated model is then used as an input to the generator which then generates not only a graphical interface but often also application logic, database scheme and other components. This is essentially the whole application. In some cases a static generator isn't even necessary and the application's GUI accesses the annotated classes dynamically using reflection. Reflection is an ability of a programming language to change properties and behaviour of objects at runtime.

A. OpenXava

OpenXava [17] was hereby chosen as the main representative of the formerly described approach to object oriented GUI generation and a business application generation. This framework is capable of generating web applications in Java programming language with GUI based on AJAX technology. Input for the generator is a domain data model written in Java, annotated with combination of JPA annotations and

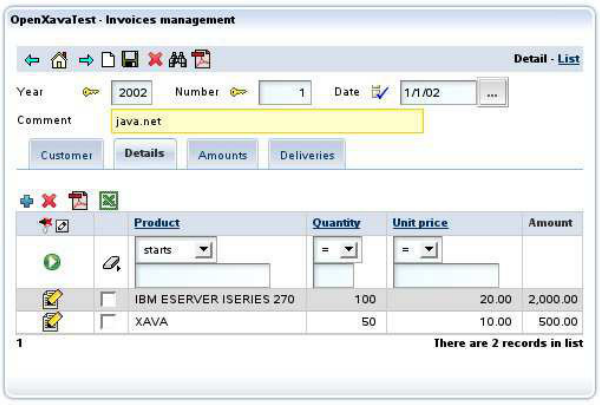

Fig. 1. Example of graphical user interface generated using OpenXava (taken from [17]).

OpenXava's own annotations. The output is a complete web application which uses Hibernate or JPA to store its data. An example of generated GUI is shown in figure 1.

OpenXava is a powerful technology which allows total separation of development of application's business logic and its user interface. Its advantages and disadvantages are more or less common to all hereby mentioned OO frameworks:

1) Independence of particular UI technology. It is possible to implement generators that produce code for different GUI libraries ranging from desktop UI to web or mobile interfaces.
2) Developers are freed from time consuming changes of the graphical interface when the model changes or when migrating to another platform.

However, the formerly described technology has some disadvantages as well:

1) Source code of the data model is often overloaded with meta-data which complicates orientation in the code and potential changes.
2) The user interface should better be derived from user behaviour and adapt to his intuition and habits rather than following the logic of the code underneath the UI [18].

B. Other OO frameworks

The OpenXava framework was used here as a representative of many others that function on similar principles. Some of them are listed along with their brief characteristics and differences.

NakedObjects [19]
: is a framework based on .NET platform. It works with domain model as well, however, the model is now written in languages for CLR – Common Language Runtime. In contrast to the OpenXava, NakedObjects uses reflection instead of code annotations and the resulting UI is created dynamically.

RomaFramework [20]
: works with domain model in Java. Besides code annotations, it makes it possible to use separated XML files to provide meta-data. Both annotation and XML approaches can be used together which can significantly reduce the amount of annotations in source code.

Magritte [21]
: is targeted at dynamic web application written in Smalltalk. Domain classes are annotated with so called description objects which add additional information using naming conventions. Such information is then used to generate GUI, database schemas etc.

There are other OO frameworks based on paradigm described above such as Tynamo [22] (previously named Trails [23]), JMatter [24], Apache Isis [25] and others. The nature of all frameworks mentioned so far is very close to the term of model driven software development (MDSD) [6] where the whole application is generated from some kind of model. As shown above, for decorating the model with additional UI information, a rich annotation language is often required. Even though, according to [7] a source code already contains enough UI information even without using expressive annotations. This idea is demonstrated in [7] by writing a small application computing an average of entered numbers. The application is written in term rewriting language [16]. Generating GUIs using this approach for functional languages is discussed in the next section.

VI. Possible approaches

Several possible approaches for generation of graphical interfaces from functional code were investigated during this work. Here are the main criteria according to which the final solution was assessed.

Practicality
: Useful and practical solution are favourized. For instance, code heavily burdened with vast amount of annotations is not considered practical since it corrupts readability and maintainability. These factors are key in prototyping since prototypes should be developed rapidly [3].

Separation
: The generated GUI should be separated from the application logic so that those two parts can be developed independently.

Generator extendability
: The solution should be extendable enough to allow creation of generators for other platforms such as mobile platforms or web.

A. IO oriented approach

The input/output (IO) oriented approach is based on the idea in [7]. It is hereby named IO oriented since its primary focus is on inputs entered by the user and outputs returned by the program in response. Term rewriting is in its nature close enough to the functional paradigm and thus the technique

described in [7] can be easily adapted for Clojure. The original tree rewriting program is shown in figure 2 and its Clojure implementation is shown by listing 1. This solution, however, has some serious disadvantages.

Listing 1 Program for computing an average rewritten to Clojure.

```
(defn add-number [average size]
  (do
    (label "Add:Result"
           (str "Average of " size " = " average))
    (action
      "Add:Add"
      (add-number
        (/ (+ (* average size) (number 0))
           (inc size))
        (inc size))
      "Add:Finish"
      (alert "Done" "Completed"))))

(defn compute []
  (add-number (number 0)
              (number 1 1 100)))
```

1) The code in Clojure is difficult to read since the GUI directives are heavily intertwined with the code for application logic. This violates our practicality requirement.
2) The GUI is tightly coupled with the application. It is not even possible to run the application without some kind of GUI. Moreover, the GUI directives cannot be easily removed from the program. Even the computational nature of the program has to take the GUI into consideration. This goes against the requirement of separation.
3) Another problem, which is more technical in nature, is interrupting the program. When the program is expecting some input from the user, it is blocking other possible actions such as termination.

Problems described above arise from the very nature of the idea used in [7] thus they cannot be easily overcome.

B. Action oriented approach

All problems described above have essentially the same cause. That is that functions contain directives for graphical interface inside their bodies. From certain point of view we can say that functions are actions that a user can execute. Such an action expects some inputs, produce some output, and as a an implicit side effect, it enables or disables some other actions. This information can be explicitly captured be annotating those functions before or inside their bodies.

The only GUI directive used inside bodies of functions would be a directive for enabling or disabling actions. The GUI generator could, by some kind of source code analysis, identify actions that belong together in the sense of their inputs. Graphical representation such as buttons and input fields belonging to those actions could then be grouped into graphical forms and windows.

According to the solution eventually presented in this article the above approach goes in the right direction. Even though, it is still rejected because serious difficulties are connected to it. The main issue is that the graph representing how actions are enabled and disabled through the flow of the program may heavily depend on the input of the program since we do not impose any restrictions on how the function should determine which actions will be enabled or disabled. Thus, constructing such a graph is in general an undecidable problem (explained in [26]). On the other hand, imposing restrictions that would make this problem decidable would seriously limit expressiveness of the functional language (further details in [26]).

VII. Functionally structured user interface

The above approaches to generating GUI from source code are not very useful still. The IO oriented approach has serious disadvantages and the action oriented approach forces us to overcome undecidability issues. This pushes us to look at the problem from another point of view. An interesting idea is to make the GUI less static and adjust its behaviour according to how functional programs actually work.

The question is how is the functional program essentially different from object oriented program from the user's point of view. By user is hereby meant the end user of the graphical interface of the program. One way to look at this is that when using graphical interface of OO program, the user creates and manipulates objects as primary entities. Functional program, on the other hand, is more about using functions as primary entities. These functions only then create, manipulate and transform objects. Thus, to reflect the different nature of functional programs, the GUI should be somehow function or operation (here we say *action*) centric. This is how we arrive at the term *functionally structured user interface* (FSUI).

A. FSUI specification

The main idea of the FSUI concept is that the UI has the same structure for all Clojure functional programs. Only the description of actions and their inputs and outputs is generated. GUI directives for enabling and disabling actions are not annotations. Instead, they are executable functions which then call the FSUI throw registered callbacks. Thus then, the program may be executed without any generated GUI just from a command line. In such a case, no callbacks are registered for the directives and so those directives do nothing.

Design of the graphical FSUI is shown in figure 3. The graphical FSUI is based on interaction of three basic kinds of components – *actions*, *invokers* and *value holders*.

Actions

An *action* represents a function from the source code of the underlying program. Each action is represented as a button on the left side of the UI. The program starts with some initial actions which can then enable or disable other actions by executing GUI directives inside the body of the function. These are the only directives that are placed inside function

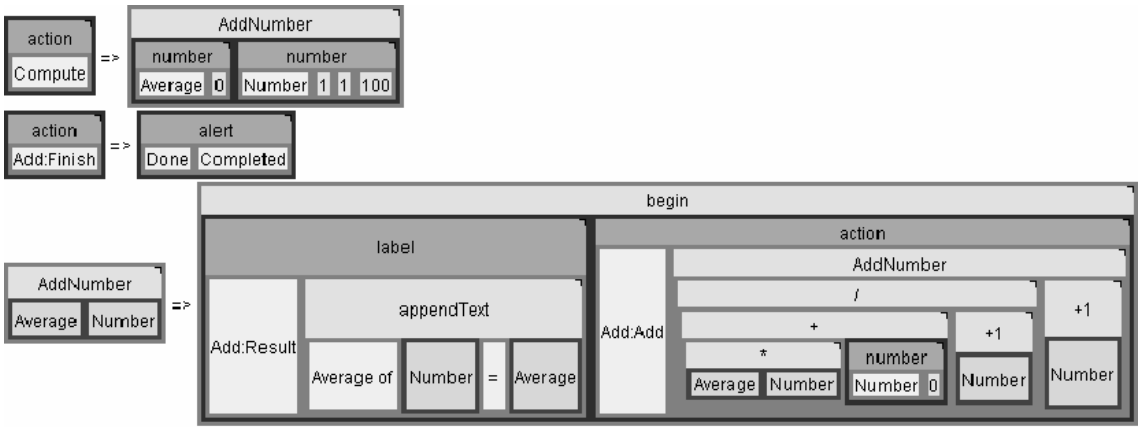

Fig. 2. Program computing an average written in tree rewriting scheme (taken from [7]).

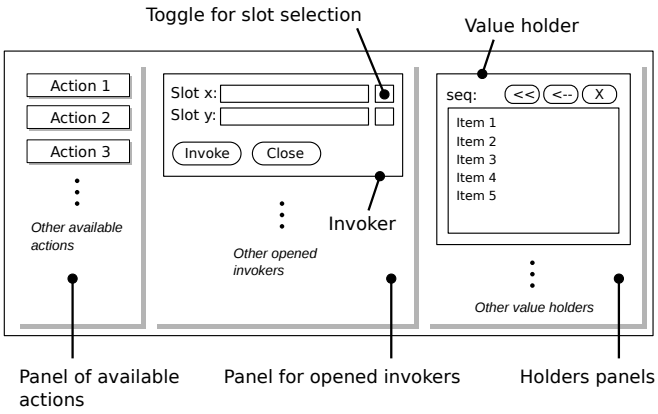

Fig. 3. Design of functionally structured user interface.

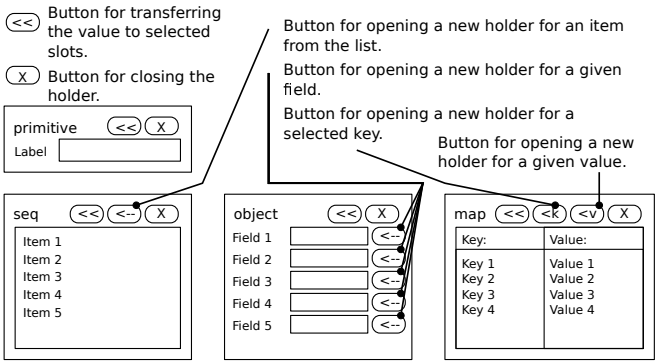

Fig. 4. Four types of holders for corresponding value types.

implementations. The key concept here is that actions do not execute functions right ahead. Instead, they open so called *invokers*.

Invokers

Invoker is a component used to collect inputs for particular action. The user can enter those inputs using *slots* (see figure 3). Slots are represented in the FSUI by text fields. Each slot declares its type so that the user cannot enter invalid values (more about the type system later in this section). The invokers panel in the middle of the UI can contain as many opened invokers as the user wishes. The actual invocation of the represented function is done by clicking on Invoke button. The same function can be executed multiple times since the invoker stays opened until the user closes it. When the function returns a value this value is inserted into so called *value holder*.

Value holders

Value holder displays a graphical component according to the type of value returned by the executed function. There are four types of holders at this time (see figure 4). These types are taken from the type system described later in this section.

Value from a holder can be transferred to a slot with compatible type. The fundamental idea here is that all values in holders are immutable thus they can be changed only by invoking a function on them. This preserves the immutability principle from the functional programming paradigm and makes the UI actually more function oriented than object oriented.

The flow of the program is controlled solely by enabling and disabling actions in the source code. Since all values are immutable, and the user has to execute a function even to create a non-primitive value, the programmer has a great control over what the user can and cannot do at a given moment.

The FSUI is implemented in Clojure in Swing GUI toolkit [15] with help of Seesaw library [27]. This library makes it easier to write GUIs in functional style.

B. The type system

Although the user is limited to create non-primitive values only using functions, he has considerable freedom in what

functions to execute and what values to pass them as arguments. To prevent errors and misunderstandings in the GUI, a function should clearly declare values of what types are expected as input and what is the type of the returned output. As the Clojure language is not statically typed, a type system for annotating functions had to be developed.

These requirements were imposed on the type system

1) The types system should be simple enough so that the user can understand it and be able to relatively easily determine which value can be passed to which function.
2) There is absolutely no need for type inference. All the type information is explicitly written in function annotations.
3) Primitive types as well as composed ones should be implemented. Lists, maps and objects are for now the basic composed types in FSUI. Objects are just heterogeneous key-value structures grouping different kinds of values together.

The type system meeting the above requirements was developed based on [28]. It is used for annotating function bodies in a way that each function is prepended with a type signature. Type signature is a list of type forms specifying value types. The first form in the list specifies the type of the return value and the following forms specify types of the arguments. The grammar for the type signature is shown in listing 2.

Listing 2 Structure of type signature.

```
<signature> ::= (<form> <form-list>)
<form-list> ::= <form> | <form> <form-list>

<form>      ::= <prim> |
                (seq <form>) |
                (map <form> <form>) |
                (object <name>)
<prim>      ::= :bool | :int | :float | :ratio |
                :number | :string | :file
<name>      ::= arbitrary symbol starting
                with a lower case letter
```

C. The generator

As it was previously mentioned, the GUI itself is dynamic which means that the code for the GUI itself doesn't have to be generated. The GUI needs only a description of actions and their respective inputs and outputs. This description is saved in Clojure data structure which is the output of the generator. Input for the generator is annotated source code in Clojure. The generator and the dynamic GUI are both implemented purely in Clojure.

VIII. Discussion

The solution developed in my master thesis [26] and described here contributes a new approach to the field of application prototypes development in functional languages hereby represented by Clojure. The main attributes of this solution, compared to object oriented approaches analysed in section V, are:

1) The UI generation is done using actual source code instead of domain data model.
2) The generator of the FSUI was developed to use more lightweight annotations then the OO solutions usually use.
3) The generated UI is uniform for all generated prototypes.
4) The type system was implemented with simplicity in mind so that the user can understand the GUI quickly.
5) Output of the generation was designed to simplify implementing the FSUI on other platforms.

Possible shortcomings of the whole FSUI and ideas for future development may be:

1) The FSUI is not flexible enough to support specific needs of particular prototypes. More control over the resulting UI layout could be integrated into source code annotations. Specific locations for invokers and value holders may be specified or actions could be nested and produce menu hierarchies.
2) The type system is not flexible enough either since it does not yet support recursive types.
3) This solution is built on top of Lisp-like languages which are dynamically typed. If statically typed languages such as Haskell are used, there will be no need for separate type system and the code annotations could reduce even further.

IX. Conclusion

Focus of this work was on the problem of automated GUI generation for functional languages in relation to application prototyping. Its result is, on the one hand, an analysis of current state in automated GUI generation in object oriented languages, on the other hand, a design and implementation of functionally structured user interface concept using Clojure and Java programming languages.

References

[1] I. Sommerville, *Software engineering*, 9th ed. Pearson, c2011.
[2] J. A. O'Brien and G. M. Marakas, *Management information systems*, 9th ed. McGraw-Hill Irwin, c2009.
[3] M. Smith, *Software prototyping*. McGraw-Hill, c1991.
[4] K. Czarnecki, *Generative programming*. Addison-Wesley, c2000.
[5] J. Nielsen, *Usability engineering*, 1st ed. AP Professional, 1993.
[6] T. Stahl, *Model-driven software development*. John Wiley and Sons, 2006.
[7] J. Jelinek and P. Slavik, "Gui generation from annotated source code," in *Proceedings of the 3rd annual conference on Task models and diagrams*, ser. TAMODIA '04. New York, NY, USA: ACM, 2004, pp. 129–136.
[8] D. S. Touretzky, *Common Lisp: a Gentle Introduction to Symbolic Computation*. Dover Pubns, 2013.
[9] R. K. Dybvig, *The Scheme Programming Language*. The MIT Press, 2009.
[10] R. Hickey, "The clojure programming language," in *Proceedings of the 2008 symposium on Dynamic languages*, ser. DLS '08. New York, NY, USA: ACM, 2008, pp. 1:1–1:1.
[11] B. J. Maclennan, *Functional Programming: Practice and Theory*. Addison-Wesley Professional, 1990.
[12] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs, Second Edition*. McGraw-Hill Science/Engineering/Math, 1996.

[13] L. VanderHart and S. Sierra, *Practical Clojure (Expert's Voice in Open Source)*. Apress, 2010.
[14] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.
[15] J. Elliott, R. Eckstein, M. Loy, D. Wood, and B. Cole, *Java Swing, Second Edition*. O'Reilly Media, 2002.
[16] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1999.
[17] (2013, Feb.) Ajax java framework for rapid application development: Openxava. [Online]. Available: http://www.openxava.org
[18] J. A. Jacko, Ed., *Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications*, 3rd ed. CRC Press, 2012.
[19] (2013, Feb.) Naked objects. [Online]. Available: http://nakedobjects.codeplex.com
[20] (2013, Feb.) Roma framework: The new way to conceive web applications. [Online]. Available: http://www.romaframework.org
[21] (2013, Feb.) Google project hosting: Magritte metamodel. [Online]. Available: http://code.google.com/p/magritte-metamodel
[22] (2013, Feb.) Tynamo framework. [Online]. Available: http://tynamo.org
[23] (2013, Feb.) Trails framework. [Online]. Available: http://trails.codehaus.org
[24] (2013, Feb.) Jmatter. [Online]. Available: http://jmatter.org
[25] (2013, Feb.) Apache isis: Domain driven applications, quickly. [Online]. Available: http://isis.apache.org
[26] M. Podloucký, "Automated gui generation for functional data structures," Master thesis, Charles University in Prague, 2012.
[27] (2013, Feb.) Seesaw. [Online]. Available: https://github.com/daveray/seesaw
[28] R. L. Akers, "Strong static type checking for functional common lisp," Doctoral dissertation, University of Texas at Austin, USA, 1995.