Information System Software Development with Support for Application Traceability

Vojislav Đukić[1], Ivan Luković[1], Matej Črepinšek[2], Tomaž Kosar[2(✉)], and Marjan Mernik[2]

[1] Faculty of Technical Sciences,University of Novi Sad, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
{vdjukic,ivan}@uns.ac.rs
[2] University of Maribor, Slomškov trg 15, 2000 Maribor, Slovenia
{matej.crepinsek,tomaz.kosar,marjan.mernik}@um.si

Abstract. Information systems are rapidly changing since new requirements are emerging frequently in business processes. When incorporating changes in the system you should not underestimate the usability and personal satisfaction of the user. There are many variables that influence the success of evolving an information system from the user's viewpoint. In this paper we outline the problem of information system traceability, the ability of users to verify the history of information system and with that a possibility to check the differences between information system's versions. Unfortunately, most of the systems support traceability only at the level of the document. The novel approach presented in this paper is integrated within WISL, using our information system generator, and supports versioning control inside information systems. WISL introduces application traceability at the level of information systems' domain concepts which deliver versioning information to the users in a seamless manner.

Keywords: Program versioning · Information systems · Dynamic graphical user interfaces · Human-computer interaction · Domain-specific modeling · Domain-specific languages

1 Introduction

Software engineers are constantly dealing with new requirements and extending Information Systems (ISs). Usually, the success of an extension is recognized by the proper functioning and efficiency of the IS. Rarely is the success factor related to usability and the personal satisfaction of the users. The latter are often forced to unconditionally adopt the changes introduced in the ISs. Activities that would ease the transition between two versions of the same software are often not supported. The above-mentioned problems are topics of research within field of human-computer interaction (HCI) more specifically as an interaction design discipline, which focuses on how to design computer software so that it is as simple, intuitive, and comfortable to use as possible [1]. In this paper we suggest

P. Abrahamsson et al. (Eds.): PROFES 2015, LNCS 9459, pp. 513–527, 2015.
DOI: 10.1007/978-3-319-26844-6_38

improving the user's usability with a software development approach often used in model-driven architecture [2], a domain-specific language (DSL) [3].

In rapidly changing systems there are many variables that influence the success of IS development from the viewpoint of user's efficiency. We will outline traceability. An example of a successful implementation at the level of source code traceability is the Git versioning system [4]. Unfortunately, it only enables traceability at the level of the text, rather than on the level of IS domain concepts.

The idea of traceability in ISs is not new [5] but is rarely being implemented since it takes precious resources. We can find examples of such functionality within some development frameworks but it is usually limited to changes inside the log file, the source code with its comments, and the updated documentation that is not linked to the context of history. Such documentation is often a good support for a software developer but unfortunately not available to the user to understand the evolution of IS, evaluating/determining the price of changes for a customer, finding potential security holes for the security engineer, etc.

In order to overcome some of these problems, we have designed a tool WISL (Web Information System Language)[1] that is based on the centralized integration of changes at the level of domain concepts. In view of that, the feature of traceability is available throughout the whole process that includes documentation, metamodel, source code, user interface, and history-enriched user documentation.

In order to support users with the capabilities of tracing the IS versions and their functionality changes, we use DSL [6]. DSLs have become an important Software Engineering (SE) field and one of the vital elements of several software development methodologies, like Model-Driven Engineering (MDE) [7], Software Factories [8], etc. In this paper we first used the power of DSLs to generate an arbitrary IS. In this manner the software engineer can describe any IS with writing specification in WISL. The novel approach that we present in this paper, is the extension of this DSL to support version control inside IS similar to the Git versioning system. We believe that such interaction design of IS improves the user's perception on changes introduced in evolved IS.

The organization of the paper is as follows. Motivation is discussed in Sect. 2. The general overview of the WISL framework is given in Sect. 3. The language behind the WISL framework is presented in Sect. 4. An extension of the language from Sect. 4 that enables the construction of system traceability is given in Sect. 5. Finally, concluding remarks with future work are summarized in Sect. 6.

2 Motivation

Each approach to software development has its advantages and disadvantages. The design of our approach is based on the actual needs of the industry on the one hand and our expert knowledge on the other. The environment where we

[1] The project source is available at: https://bitbucket.org/work91/wis.

operate an economy is changing rapidly. Economy is changing enterprises from large to small and micro. Budgets for information technology are getting smaller, although there is a growing demand for ISs. Economy also changes the dynamics of the IS life-cycle. The frequencies of IS changes are increasing constantly. In practice, we meet the requirements, which are expected to change the IS not in a few days or weeks but within a few hours. The expectations of customer are often unrealistic and do not foresee the consequences they bring.

IS software development is becoming similar to the software prototyping in many aspect. In order to support our needs, the IS development framework needs to meet the following requirements:

- A high level of interactive integration of the customer, the developer, and the user.
- Support of traceability not only for the developer but also for the customer and the user.
- Support for the incremental development of domain specific concepts.
- Support for rapid implementation.

Customer and user are often the same person (Fig. 1) but this is not essential. In a simplified scenario, the customer is the one who is ultimately responsible for paying for the IS changes. Supporting aspects of billing system is very important, because requirements can grow out of proportion very quickly. The user is one who usually interacts with the IS user interface. However, the developer is usually the software engineer who adopts and implements a customer's requirements.

The described procedure is similar to the well-known issue-tracking systems [9], where the subscriber or user opens a new ticket to which developer responds

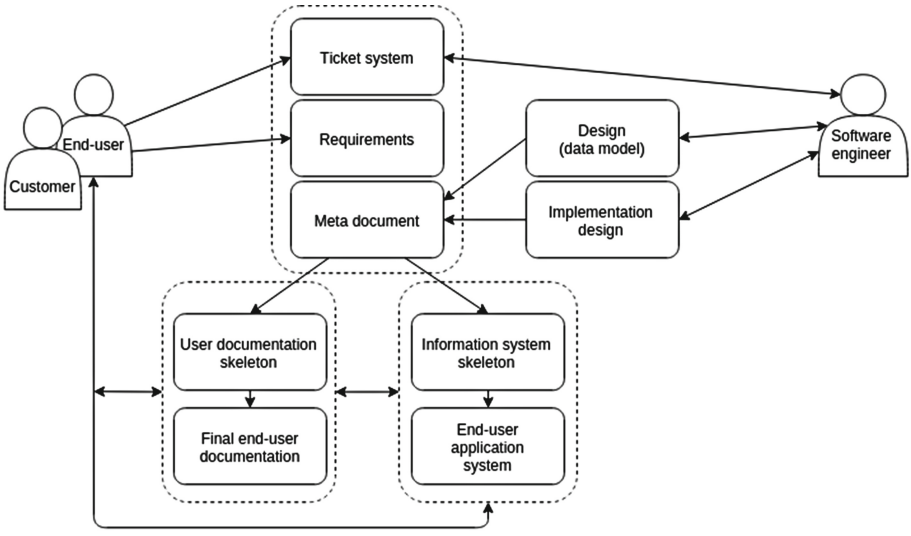

Fig. 1. Outline of WISL with traceability support

with a change in the IS or just a simple reply. Support of traceability is taking care of by including the ticket identifier in all documents of IS. The identifier of changes can be found in the description of the changes, over the source code to the user interface forms, or even reports.

In regard to the efficiency and transparency of incremental developing it is necessary to support the history of changes at the level of the domain concepts of IS not just at the level of text. Using such support IS can be returned to the previous version, review changes over time and link requirements not only on the level of specification but also at the level of graphical interface, database and associated documentation. These requirements of rapid development we have achieved with WISL by using a code generation, and concepts of DSL.

3 System Architecture

WISL is designed to be a comprehensive way of describing an IS in general (see Fig. 2). It combines the basic concepts from the entity-relationship model (ER model) but also includes some technical details about the end-system implementation. Comparing abstraction levels, WISL stays between ER and the relational data model. After creating the system description ("WISL Model" from Fig. 2), we have provided a process (see "IS Generation") for automatically generating a functional prototype ("IS Prototype" in Fig. 2). First of all, we used XText to implement WISL language and an editor as a plug-in for Eclipse IDE. Using the same tool we have defined a model to model transformation from WISL description to the WISL object model. The object model has been built within the Eclipse Modeling Framework (EMF) [10]. We used XTend to write code templates from the object model. Since we opted for Web-based applications, we have implemented two highly independent generator categories for the client and server sides of the application. WISL end application is built-up using some of the more widely used software frameworks mixed with our own extensions in order to support system extensibility. On the server side we used Spring Framework and Hibernate [11,12] together with a WISL data framework, in order to overcome problems with bidirectional relationships within the Hibernate framework. We have generated a JPA [12] data model, Spring repositories and Spring REST Controllers [11] from the WISL object model. On the client side we have implemented a highly decoupled Single Page Application (SPA) using the AngularJS framework [13]. In addition, we have provided the WISL with a dynamic user interface framework. In order to support the client side of the application, we have generated Angular routers, services, and menus but also WISL dynamic UI descriptions.

It is very important to mention some of the techniques for integrating a generated code with a hand written code. We are aware that WISL can build functional applications but it is not capable of satisfying all customer needs in commercial projects. On the server side we have relied on the options of the Spring Framework. Using configuration files it was possible to exclude some parts of the generated code or to include custom written code and combine it with

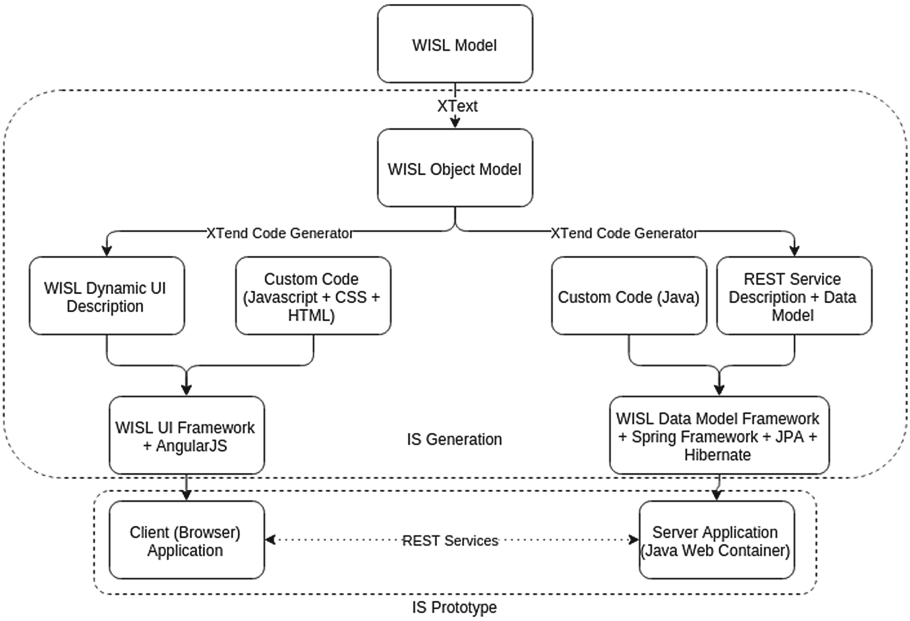

Fig. 2. WISL system architecture

the generated one. On the other hand, the WISL UI framework enables simple extensions of a client side application just by following the project structure.

3.1 WISL Backend Application

The WISL system follows the SPA architecture. It means that a server side does not generate HTML code, as in traditional web applications. The main idea is to write HTML templates that are going to be delivered to the client side as a bundle together with Javascript code. By using the Javascript code, it is possible to fill the templates with data retrieved by the server REST controllers. A big advantage of this approach is that it allows browsers to cache a client-side application. It leads to much more effective bandwidth usage and simpler server side code.

In order to support SPA architecture, the WISL backend is a REST application based on the Spring Framework (see Fig. 3). This framework has a well-defined code structure that allows developers to modify or to easily add custom code. What is more important, it separates the generated and custom code. Newly generated code will not overwrite the hand-written parts and violate a previous structure. On the other hand it is possible to use generated code within other custom parts using the Spring framework features. By changing the configuration files, a developer can substitute or exclude some parts of the generated code.

Fig. 3. WISL backend application

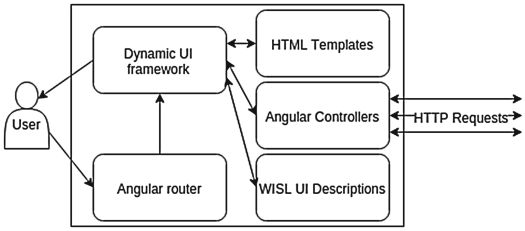

Fig. 4. WISL frontend application

The REST controllers by themselves are quite simple components that can be easily generated from the WISL system description. A much more serious problem is the data storage system. In order to achieve effective data persistence, we have used Java Persistence API (JPA) and Hibernate (see Fig. 3, again). JPA relies upon Java Database Connectivity (JDBC) drivers that allow usages of many different Relational Database Management Systems (RDBMS).

3.2 WISL Frontend Application - WISL Dynamic UI Framework

The WISL Dynamic UI framework is the most important part of the WISL client application (see Fig. 4). It is capable of generating the complete functional web user interface just by using the WISL UI description model. The framework is implemented as a group of AngularJS directives. They are independent components that contain HTML templates as static view descriptions and Javascript controllers that define the behavior of a component. We have implemented directives for the global entity view, detail entity view, entity create, entity update, show entity relationship, and edit entity relationship. There is also a general directive that encapsulate the previous ones. If a developer is satisfied with the default functionalities it is enough to call a general directive for each entity. Otherwise, it is possible to use only some of the directives or to write the entire user interface from scratch. For example, write a completely new global entity interface but after that just include buttons for entity editing.

Each of the UI directives is based on the WISL UI description model. It is a description of the user interface on an abstract level. The description is in the JSON format (see Fig. 5). The directives parse the description and use it to generate different user interface components dynamically. It is also possible to change this description during the run-time. The framework will recognize the changes and will reorganize the user interface.

```
angular.module('WIS.projectManagement')
.controller('ctrlProject',['$scope',
  function($scope){
    $scope.model={
      label:'Project',
      serviceName:'ServiceProject',
      fields:[
      {
        label:'Name',
        name:'name',
        type:'text',
        important: true,
        validation:{
          required: false,
          vType: 'stringsingleline',
          customValidation:[
          ]
        }
      },...
      ],
    relationships:[
    {
      entity: {
        label:'Employee',
        serviceName:'ServiceEmployee',
        fields:[
        {
          label:'First name',
          name:'firstName',
          type:'text',
          important: true
        },...
        ]
      },
      label: 'Project manager',
      name: 'projectManager',
      editable: true,
      type: 'one'
    },...
    ]
  };
}]);
```

Fig. 5. WISL UI description

The UI description provides technical details about UI components (e.g. type of component - see that the component "name" is presented as a "text" type in Fig. 5) and interfaces of Spring REST services on the server side of an application. The description could be custom-written or generated from a WISL model. The description in Fig. 5 corresponds to the WISL model described in the following Sect. 4.

4 WISL IS Definition

As was mentioned previously, WISL is a solution for a comprehensive description of IS. The WISL abstract syntax (metamodel [14]) has been created using ECore and EMF (see Fig. 6). The concrete syntax is textual and is implemented in XText. The WISL metamodel shares some of the basic concepts with the ER data model. As in ER, WISL contains entities (WEntity in Fig. 6) and relationships (WConnection in Fig. 6) with slightly different properties. Each entity, except its own name, contains zero or more attributes similar to ER. However, the attributes (WAttributs in Fig. 6) are much different. They play an important role for the most part in end-system generation. Hence, at the attribute level it is possible to configure name, type, label, validation rules, and much more properties which can affect the user interface and user experience. WISL also provides the possibility for defining a new enumeration type together with suitable UI component and validation rules and then assign it as a type to some attribute. The other very important concept is the link. The link can be defined between two and only two entities. Each side of the link has minimal (zero or one) and maximal cardinality (one or many). Unlike in the ER, n-ary relationships and categorization are not supported. Also, it is not possible to bind attributes to a relationship. If the concept of a gerund is not directly supported then it should

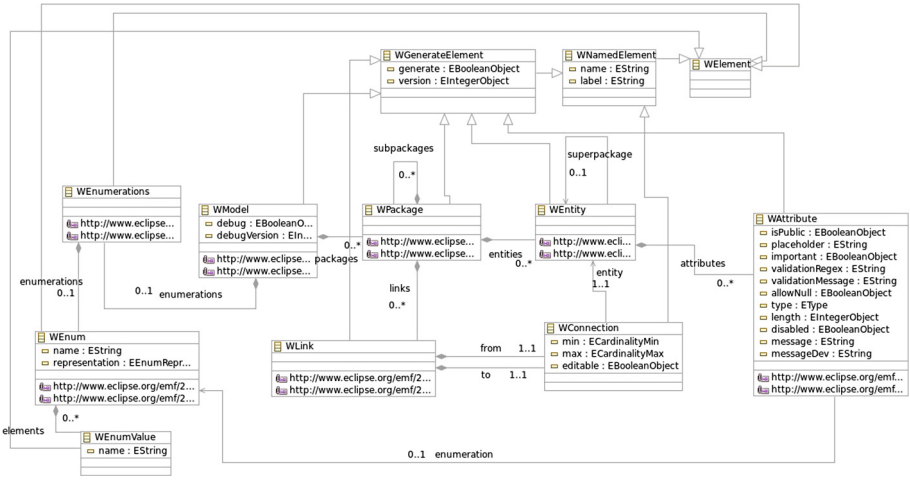

Fig. 6. WISL metamodel

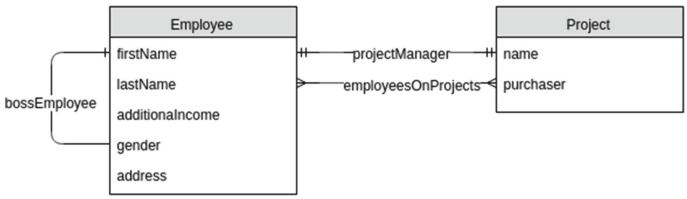

Fig. 7. Employee salary ER diagram

be implemented as a new entity. Also, the solution is the same for relationships with attributes. The inheritance is supported in the WISL metamodel but it is not implemented in the end system due to technical reasons. In addition to the mentioned concepts, WISL extends the existing ER data model with a concept of package (WPackage in Fig. 6). By using packages, the user is able to build a system as a hierarchical structure.

4.1 WISL Use-Case Scenario

As an example of WISL application we can consider a very basic use case - manager attaching employees to different projects. Each employee is described by the following attributes: first name, last name, wage, additional income, date of birth and gender. A project is defined with the name, and purchaser. A corresponding ER schema to this scenario is the one shown in Fig. 7.

The same semantic as described in WISL language is in Fig. 8. Note that entities and links are defined separately within individual blocks of code. Each link has a name (e.g. projectManager) and two related entities - the first entity (construct “from”) and second entity (construct “to”). It is necessary to assign

```
entities{
    employee{
        attributes{
            firstName,
            lastName,
            additionalIncome{
                type Double
            },
            gender{
                type Enumeration
                enumeration EGender
            },
            address
        }
    },
    project{
        attributes{
            name,
            purchaser{
                type StringMultiline
            }
        }
    }
}
links{
    projectManager{
        from employee as projectManager {
            min one
            max one
        }
        to project as managedProjects
    },
    employeesOnProjects{
        from employee as employees
        to project as projects
    },
    bossEmployee{
        from employee as boss{
            max one
        }
        to employee as subworker
    }
}
```

Fig. 8. Employee salary system in WISL

a role name for each entity (e.g. from employee as projectManager). This is particularly important in case of a recursive relationship.

Entities in the WISL can have type, length, placeholder and many other properties. Each of these properties has a default value. Note the difference between attributes "firstName" and "additionalIncome" (see Fig. 8). Using the default values wherever possible, we have tried to decrease learning time for the system developer.

4.2 IS Generation

The WISL is capable of generating fully functional prototypes. The generated system supports basic CRUD operations (create, read, update and delete) [11] for each defined entity and relationship. An entity has two views - a global and a detailed view (see Fig. 9). The global view shows only the important attributes - attributes that are marked as important in a model. This concept of a way to narrowing entities with a very extensive number of attributes. It makes user interface much clearer. Search queries are only possible regarding important attributes (by default).

In order to make the user more comfortable, the WISL provides a global search, entity pagination and sorting. These operations are usually very time-consuming for implementation, but they are an infallible part of every modern IS.

Each time a developer changes a WISL model, it is necessary to generate a system again. If a custom code is written properly, the process of generation will not violate a project's structure and functionalities. In regard to the simple example from Sect. 4 (employee <-> project), the generation process lasts only a few seconds and creates 17 files with 973 lines of code.

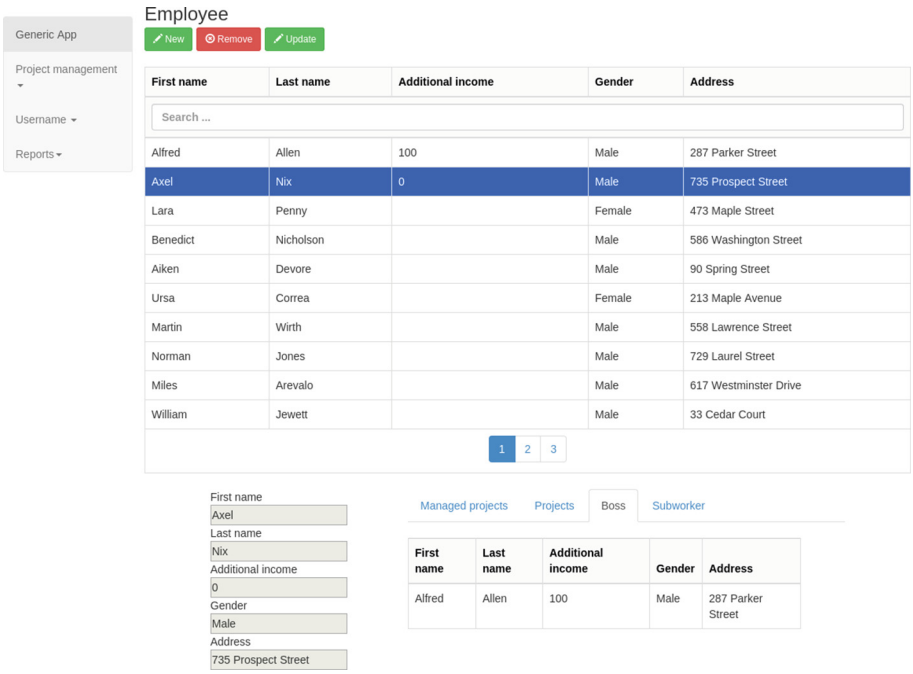

Fig. 9. Generated system - global view (upper part) and detail view (lower part)

5 IS History Support

In the previous Sects. 3 and 4 we introduced the WISL system, which is able to generate arbitrary IS. However, the original architecture does not have any kind of support for traceability. The implementations of these WISL functionalities are presented in the following section where advice is given on how we extended the metamodel behind WISL. The outcome of this extension is IS with integrated versioning support on the levels of domain concepts.

An IS model can be viewed as a state of a metamodel. Making changes to the model has similar properties as modifying the state (data) within database. Any change must be atomic, consistent, isolated and durable. In order to change the model we used the operations of deletion and insertion. While the concept of update is interpreted as a combination of deletion and insertion.

Each operation has parameters that describe the domain concept. So you could say that the version of the model changes after each operation but such an interpretation could lead to inconsistencies of the model. Therefore a set of operations is carried out during a transaction. All changes within a transaction are identified with the same id version. Update of the concept in a model is always performed as a transaction.

In order to support the suggested approach, we needed a relatively small change of the metamodel (see Fig. 10). All we had to do was to expand the

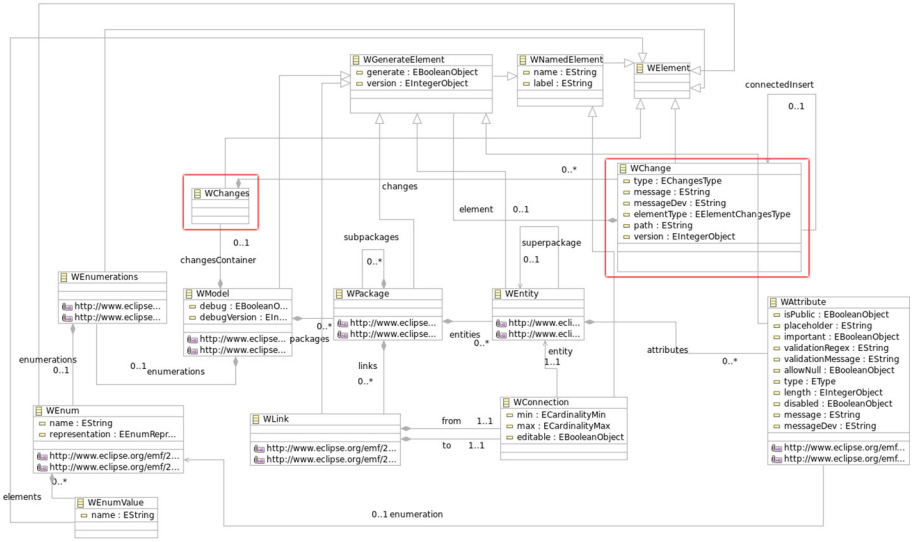

Fig. 10. WISL metamodel with history support included

metamodel with WChange and WChanges classes, where WChange includes the concept updates and WChanges is just a container for changes (Fig. 10). In order to achieve transparency of the changes arguments are introduced for the user (message), developer (messageDev), identifier changes, and the like.

Changing the metamodel has two aspect. The first aspect is changing the user interface and second is changing tts IS data. In order to achieve full history backtracking, data transformation needed to be provided (for example transforming integer to real data type). Modifications that need complex transformations will be the subject of future work.

5.1 Example

Imagine a real case-scenario which extends the IS defined with specifications from Fig. 8. In this scenario we would like to insert new attributes (e.g. “wage”, “date of birth”) and delete some attributes (e.g. “address”) inside entity “employee”. Note, that we would like that WISL automatically inserts/deletes concepts in the user interface, insert new entities in the database, etc.

Normally, one would insert just the attribute inside the specifications in Fig. 8. In this way the system would be fully functional but the history of the changes would be absent. Using the extension that corresponds to the new metamodel (see the concept WChange in Fig. 10, again) we are able for instance, to insert attribute “wage” in a way that will support traceability for the developed IS.

The concept “history” in Fig. 11 contains several block sections that correspond to individual change in the IS. Each block contains the same attributes.

```
{
    ticketId 2
    message 'Wage added'
    messageDev 'Wage added development'
    path 'package.projectManagement.entity.employee.attribute.wage'
    type Insert
    elementType Attribute
},
{
    ticketId 5
    message 'Address delted'
    messageDev 'Address delted'
    path 'package.projectManagement.entity.employee.attribute.address'
    type Delete
    elementType Attribute
    element attribute {
        address
    }
}
```

Fig. 11. Traceability specification in WISL

```
entities{
    employee{
        attributes{
            firstName,
            lastName,
            wage{
                ticketId 2
                type Double
            },
            additionalIncome{
                type Double
            },
            dateOfBirth{
                ticketId 3
                type Date
            },
            gender{
                type Enumeration
                enumeration EGender
            }
        }
    },
    project{
        attributes{
            name,
            purchaser{
                type StringMultiline
            }
        }
    }
}
```

```
links{
    projectManager{
        from employee as projectManager {
            min one
            max one
        }
        to project as managedProjects
    },
    employeesOnProjects{
        from employee as employees
        to project as projects
    },
    bossEmployee{
        from employee as boss{
            max one
        }
        to employee as subworker
    }
}
```

Fig. 12. History aware specifications

In our case-scenario attribute “ticketID” is set to value “3”, which means that this change has been done for user request id “3”. Attribute “message” contains the message that will be presented in user interface beside the concept that is going to change. Our system also supports messages that can be seen only by a software developer (attribute “messageDev”). The next attribute that needs to be set in history is the name of the concept that is going to change (attribute “path”). Here, the full path is expected - package name, entity name and finally, the name of the attribute. Next attribute is “type”, the value of which represents the operation of the change in IS. The last attribute is “elementType” which contains the information about the changing concept. Note, that deleting the concept is very similar to inserting a new concept, the only difference is in the value of the attribute “type” (see the second block inside the history in Fig. 12).

After generating a new version of IS, the user interface has been changed (see Fig. 13). The deleted fields are marked with red rectangles and new ones with green. For every changed field you can get additional information in the form of the hint (black bar with text).

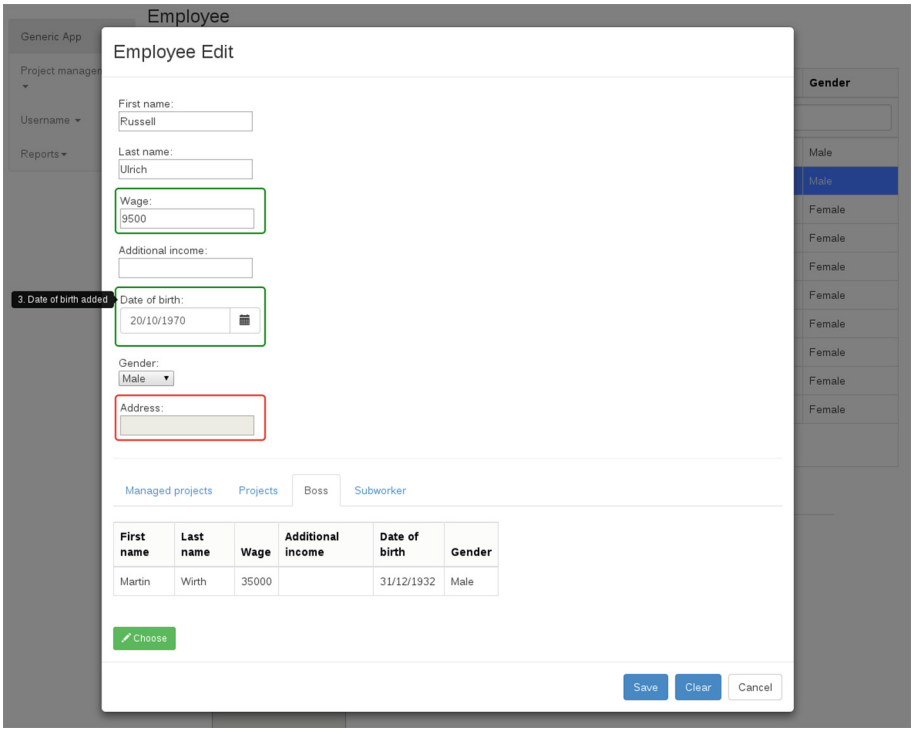

Fig. 13. Interface and help changes in IS (Color figure online)

5.2 Discussion

IS history support in WISL has been shown on a very simple use-case scenario where text fields have been added/deleted from our IS in previous subsection (see Fig. 13, again). The difference of the last two instances of IS is shown in this figure. But the essence and the real power is hidden behind this figure and WISL implementation. The user can always select two arbitrary versions of IS and check the differences between them. Imagine a real-case scenario where user might miss several versions of the IS. IS developed with WISL has a support for such user which can check the differences between the current and a specific version of the IS. Note that this is supported with keeping information of IS instances in our modified WISL metamodel (see Fig. 10, again).

Of course, the history-aware view in the IS is optional. User that understands the newly introduced changes in IS can easily turn off support for traceability. An interesting research question would be how long to show new changes visible to the user and when to automatically hide traceability support.

6 Conclusion

The importance of usability and efficiency for IS users should not be underestimated [15]. To conclude, we may say that the issue of perceiving software modifications by software users and how those modification influence human work and activities is a large and complex problem. WISL, our IS generator, is an attempt towards developing software with changeability awareness. In order to visualize changes in the IS, we have incorporated into WISL an extension, which in a seamless manner supports users with additional information about the changes in IS. The idea is somehow similar to versioning control systems, where we can trace the differences between two versions of text file. In WISL, this idea is incorporated on a level of domain concepts included in an IS. As future work of the current implementation, we plan to prepare several use-case scenarios and test WISL with different usability measurement models [16], to see how our framework affects the user's usability and efficiency. We also plan to extend WISL with automatically generated history-sensitive documentation that would facilitate users' understanding of the changes in ISs. The next desired feature for WISL is support for history management - traceability of the changes throughout history is currently still a plan for our future work.

Acknowledgments. Research presented in this paper was supported by Ministry of Education, Science and Technological Development of the Republic of Serbia, Grant III-44010, as well as the Project of Bilateral Cooperation of the Republic of Serbia and the Republic of Slovenia, Grant BI-RS/14-15-034.

References

1. Preece, J., Sharp, H., Rogers, Y.: Interaction Design-Beyond Human-Computer Interaction. Wiley, New York (2015)

2. Abrahão, S., Iborra, E., Vanderdonckt, J.: Usability evaluation of user interfaces generated with a model-driven architecture tool. Maturing Usability: uality in Software, Interaction and Value. Human-Computer Interaction Series, pp. 3–32. Springer, London (2008)
3. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
4. Lawrance, J., Jung, S.: Git on the cloud. J. Comput. Sci. Coll. **28**(6), 14–15 (2013)
5. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. J. Syst. Softw. **80**(6), 918–934 (2007)
6. Kosar, T.: Martínez López, P.E., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Inf. Softw. Technol. **50**(5), 390–405 (2008)
7. Stahl, T., Völter, M.: Model-Driven Software Development. Wiley, New York (2006)
8. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, New York (2004)
9. Aggarwal, A., Waghmare, G., Sureka, A.: Mining issue tracking systems using topic models for trend analysis, corpus exploration, and understanding evolution. In: Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2014, pp. 52–58, New York, NY, USA. ACM (2014)
10. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
11. De, A.: Spring, Hibernate, Data Modeling, REST and TDD: Agile Java Design andDevelopment. CreateSpace Independent Publishing Platform (2014)
12. Bauer, C., King, G.: Java Persistance with Hibernate. Dreamtech Press, New Delhi (2006)
13. Freeman, A.: Putting AngularJS in Context. Apress, Berkeley (2014)
14. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003)
15. Hering, D., Schwartz, T., Boden, A., Wulf, V.: Integrating usability-engineering into the software developing processes of sme: a case study of software developing sme in Germany. In: Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2015, pp. 121–122. IEEE Press (2015)
16. Shawgi, E., Noureldien, A.: Usability measurement model (umm): a new model for measuring websites usability. Int. J. Inf. Sci. **5**(1), 5–13 (2015)