

Technical Report: A UML Profile for Modeling User Interfaces of Business Applications

Dragan Milićev
University of Belgrade
Faculty of Electrical Engineering
Department of Computing
P.O. Box 35-54, 11120 Belgrade, Serbia
dmilicev@etf.rs

Žarko Mijailović
SOL Software, www.sol.rs
Kosovska 28, Belgrade, Serbia
zarko.mijailovic@sol.rs

Abstract

We present a novel approach to modeling and implementing user interfaces of large business applications. The approach is based on the concept of capsule, a profiled structured class from UML, that models a simple UI component or a coherent UI fragment of logically and functionally coupled components or other fragments with clear interface. Consequently, the same modeling concept of capsule with internal structure can be re-applied recursively at successively lower levels of detail within a model, starting from high architectural modeling levels, down to lowest levels of modeling simple UI components. Interface of capsules is defined in terms of pins, while functional coupling of capsules is specified declaratively, by simply wiring their pins. Pins and wires transport messages between capsules, ensuring strict encapsulation. The approach includes a method for formal coupling of capsules' behavior with the underlying object space that provides proper impedance matching between the UI and the domain object space layers, while preserving clear separation of concerns between them. We also describe an implementation of a framework that supports the proposed method, including a rich library of ready-to-use capsules, and report on our experience in applying the approach in large-scale commercial systems.

Keywords

Graphical user interface (GUI), Unified Modeling Language (UML), modeling, model-driven development, software architecture, business applications, data-centric applications, information systems

1 INTRODUCTION

Building user interfaces (UI) for complex interactive applications, such as information systems, is usually labor-intensive and time-consuming work. Typically, in a large-scale industrial project, building the UI takes a substantial part of time and resources¹. In addition, the complexity of design and the volume of artifacts such as source code, page templates, style sheets, configuration files, and the like, needed to define a UI in most popular technologies, burdens the construction and especially the maintenance of UIs.

The widespread orientation towards Web-based UIs of business applications in the last decade has definitely improved their availability through ubiquitous Web browsers and different client devices. However, a key feature of traditional approaches of building form-oriented UIs—tight coupling with backend relational data models—seems to have been abandoned in the widespread orientation towards

¹Different research papers estimate high percentages of code and time devoted to working on UI, as well as huge numbers of widgets in present-day applications [17, 47]. The authors' experience, corroborated by timesheets collected from a number of industrial projects, coincides with those findings.

Web-based UIs. The conceptual gap introduced between the UI and the business logic tiers, usually left to programmers to overcome, as we will argue later in the paper, causes considerable accidental complexity of yet immature mainstream Web-based technologies. Additionally, regardless of the scale of accidental complexity imposed by the chosen technology, dealing with the essential (inherent) complexity of large-scale UIs is always a challenge.

Recent trends in model-driven development [21, 37, 46] encourage the use of models made in highly abstract languages, such as UML, for building executable systems, without losing the advantages of tight and formal coupling between the abstract architectural and design views and executable artifacts. Da Silva and Paton [8] remark that “using models as part of user interface development can help capture user requirements, avoid premature commitment to specific layouts and widgets, and make the relationships between an interface’s different parts and their roles explicit.”

In this paper, we pursue this trend and propose a novel approach for modeling and building UIs of large-scale applications. Our motivation was to design a method for building UIs with reduced accidental complexity and with seamless coupling of the UI and business logic tiers. Besides, we tackle essential complexity of UIs by introducing concepts of higher expressiveness and with strong support for abstraction, encapsulation, and reuse. Our approach is fully formal and uses executable models, which means that a model is also an implementation and leads to a runnable UI without any manual transformation. Development and maintenance efforts are reduced by using declarative coupling of UI components, instead of the widespread imperative way of event handling. Finally, by strongly impeding mixing of business logic with the UI code, our technology improves the architectural design of the application’s UI.

Our technique is inspired by the concepts originally conceived in a modeling approach used in a completely different domain [38], but later adopted in the UML2 standard [29]. The proposed approach is part of an executable profile of UML named OOIS UML, described thoroughly elsewhere [21]. Although the applicability of the presented approach is not limited to that kind of applications, it is primarily conceived for large-scale, business and data-centric applications, such as information systems, where the adequate and coherent internal software architecture and design, efficiency of development and maintenance, as well as usability and accessibility to the data and functionality is of higher importance than pure visual appearance, contents and its attractiveness, as it is often the case with public Web sites and portals. On the other hand, public Web sites and portals have some specific demands and aspects that are successfully addressed by some other techniques that are orthogonal and unrelated to our approach, such as portlets [39] and mashups [45]. Although it is particularly suitable for Web-based UIs, as it is the case for our current implementation, our technique is not limited to that kind of UIs.

The paper is organized as follows. The next section gives a brief overview of the paradigms and techniques used in widespread industrial frameworks for building UIs, as well as of some more advanced approaches to modeling UIs. Section 3 presents the motivation and the idea of the proposed approach in a brief and informal manner. Section 4 describes the proposed concepts in more detail. Section 5 briefly describes our existing implementation, while Section 6 reports on our experimental and industrial experiences of applying the presented approach. The paper ends with conclusions.

2 OVERVIEW OF EXISTING SOLUTIONS

We first provide a brief overview of a few major categories of mainstream industrial approaches to building UIs, by analyzing some of their most prominent representatives widely adopted in the development community. We then comment on some UI modeling approaches. Our focus is on the set of criteria that we find relevant for each UI development technology: level of abstraction, architecture modeling capability, flexibility, usability, platform independence, and extensibility.

Mainstream Industrial Approaches

Form-oriented, relational database-coupled frameworks. The traditional approach to building UIs for relational database applications is to organize the UI into forms. The forms present values of fields of records in the corresponding form controls, such as text boxes, list boxes, check boxes, grids, etc. Tools and runtime engines support generic navigation through forms and direct coupling of controls with the backend data, so that the developer does not need to take care of data locking, transfer, transformation, and updates. For example, when the user switches to another record in the master part of a master-details form, the values in the details part are automatically refreshed by the mechanism incorporated in the generic form. Similarly, when the user modifies the value in a control bound to the backend data record, the value is immediately stored to the database record. No manual coding is needed for this kind of behavior except for declarative binding of a UI control or entire form with its backend data source. In other words, tools provide a uniform view to the relational database application development and tight coupling with data sources, regardless to the architecture of the distributed application. In all other aspects, these frameworks typically work as general-purpose event-driven frameworks. Among many frameworks of this category, we will just mention Microsoft Access and Oracle Forms as classical representatives.

Template-based approaches. These techniques base the design of UI pages or forms on templates that mix the target UI markup code (HTML, XML, or XHTML), delimited blocks of native language code (Java, C#, PHP, Python, or another), and possibly technology-specific tags for supporting various built-in tasks. These tags are either compiled into the native technology code or interpreted at runtime and are responsible for generating dynamic parts of the UI or for specifying server actions. A typical representative of this category are Java Server Pages (JSP) that mix markup with Java code and JSTL (tag library) tags for common tasks such as iterations and conditions, manipulating XML documents, internationalization, and data access. Mixing the markup and native Java code in JSP pages (for example, embedding the markup into Java “for” loops) results in frequent syntax switching which makes such JSP pages extremely hard to understand. This type of scripting was recognized as tedious and painful for programmers, so the Expression Language (EL) was introduced to make the scripting easier and script files more readable. EL also enables easier, more compact, and more readable access to Java objects than pure Java code. However, it introduces new syntax into an already linguistically heterogeneous environment. The problem of linguistic integration in general is well addressed by Groenewegen et al. in a recent work on WebDSL [12]. They show how confusing and detrimental effects on UI development can arise from such a linguistically heterogeneous environment. They have noticed that “programmers must know numerous frameworks and general-purpose and domain-specific languages, such as, HTML, cascading style sheets (CSS), JavaScript, Flash, Java, XML, XSLT, Hibernate, and JSF.” The majority of web frameworks in this category feature this same kind of heterogeneity: ASP, PHP, Struts2, Spring MVC, Spycy (Python Server Pages), and others.

Probably the most progressive template- and action-based framework is Ruby on Rails [35]. Rails implements the Model-View-Controller pattern to cleanly separate the model (business logic) and the view (user interface) [41]. Rail’s view files contain Ruby code interspersed with static HTML that is executed when needed to interleave dynamic content, just like in other template-oriented technologies. However, Ruby on Rails does contribute to UI domain significantly. It spares programmers from building and maintaining intricate configuration files lowering the technology threshold [26]. As Viswanathan remarked [41], “it favors convention over configuration; following standard conventions eases development and understandability while precluding complex configuration files.”

Object-oriented event-driven approaches. These techniques match well with basic object-oriented principles. Their main advantages are the ease of composing UI controls and widgets into larger

reusable components and intuitive and flexible definition of their event handlers. In particular, all behavior and interaction of UI controls is specified in their event-handlers, which are ordinary operations written in a hosting programming language. One of newer technologies in this category that has successfully provided a full linguistic integration in a platform-transparent way for the Web is Google Web Toolkit (GWT) [11]. GWT is a component-based framework with a very rich library of UI components (widgets). The UI is written in Java. The Java code is then compiled into optimized JavaScript by a GWT compiler. This way, Web UIs are built and maintained completely in a single programming language in a classical way, just like traditional desktop UIs. However, from the development paradigm perspective, GWT belongs to the same category of OO event-driven frameworks just as Swing, AWT, SWT, MFC, Delphi, and many others. Seaside is another interesting UI framework for building Web applications in a linguistically unified environment of Smalltalk.

Although it also falls into the same category, the Qt framework [36] adds some original concepts to the paradigm that have certain similarities to the concepts proposed later in this paper. It is originally desktop-oriented, component-based, C++ application and UI development framework with an extremely rich library of components. Although the complexity of UI subsystems, in particular, the complexity of UI component structure and interaction, is well known to be typically enormous, other UI frameworks have not recognized a need for special concepts (other than simple event handlers) to alleviate that problem. Qt offers a solution by means of *slots* and *signals*. A signal is emitted from a UI component when a particular event occurs. A slot is an operation that is called in response to a particular signal. A signal can be connected by declarations to as many slots as needed, and vice versa. A component which emits a signal does not know which slots receive it. This way, UI components are loosely coupled by signal-slot connections that are specified declaratively. This mechanism enforces a true information encapsulation on the component level. However, Qt does not provide any specific coupling with the domain object space whatsoever. In addition, our extensive experience in using Qt in industrial projects shows a tendency of proliferation of inter-component signal-slot connections, which in turn results in pretty complex networks of components and connections very hard to comprehend and maintain, due to the lack of proper architectural views and abstraction mechanisms.

Hybrid approaches. These technologies represent a newer trend in mainstream Web UI development. They combine templates and expression languages for easy access to server-side objects from the template UI markup as in template-based approaches, with an object API for access to UI widgets from the server-side application code as in OO event-driven approaches. One typical representative of this category, JavaServer Faces (JSF) [5, 14], is a component-based UI framework for Java-based Web applications. JSF consists of: (a) the API for representing UI components, managing their state, handling events, and input validation at runtime, (b) tag libraries for expressing UI components within a Web page, and (c) an expression language for binding of components to server-side objects, as well as of component-generated events to server-side application code. Other notable frameworks from this category are ASP.NET MVC, Apache Wicket, and Apache Tapestry that additionally (and similarly to Ruby on Rails) puts “convention over configuration” principle in practice. Adobe Flex is conceptually close to these technologies: templates are used for defining layout, OO programming is done in the ActionScript language, and there is also a valuable data binding mechanism.

Apart from other characteristics inherited from template-based and OO event-driven approaches, these approaches introduce a few other severe drawbacks. First, when a *post* request is made to the Web server by an action issued from a client, the server has to build up (or to keep it already available in memory) the entire object structure representing the hierarchy of UI widgets available on the page being viewed. In many cases, this structure is necessary just to provide the foundation for handling one or few events issued from a widget. In order to handle that event within an object’s operation, the programmer has to have the entire structure of other objects accessible in the scope of the event handler, and the server runtime has to provide these objects. Similarly, when a new

page has to be rendered, the same object structure for the new page has to be available to handle other application-specific tasks in the same manner. This represents a significant workload and memory demand for the server side, very often oversized for very simple interaction between two UI widgets, such as validation or changing a property of one when the selection or the contents in the other has changed. Such simple tasks can be much more efficiently performed on the client side in JavaScript, thus releasing the server from the unnecessary workload and distributing the workload to clients. However, in practice, most developers do not seize for such optimized solutions because they either do not know JavaScript or are not aware that this is even possible. Instead, they simply do it on the server side because it is much simpler that way, since it is done in a syntactically homogeneous environment of an OO programming language, through a convenient API, native in the given framework. Even when a developer occasionally implements widget interaction using JavaScript on the client side in order to improve responsiveness, this has bad implications on the overall organization of the code because different parts of UI's behavior are done in different languages and placed in different artifacts (template pages or OO programming language code). The authors have seen much code made that way, where the same task is performed on the client side in one case, and on the server side in another, causing complete confusion to those who try to understand and maintain the application.

UI Modeling Approaches

Model-based UI development approaches aim to provide an environment where developers can design and implement UIs in a more systematic and easy way than with using traditional UI development tools. Da Silva [7] gave an overview of the first two generations of model-based UI environments up to year 2000. As he said, apart from having raised the abstraction level, created methods to design and implement UI in a systematic way, and provided infrastructure required to automate tasks related to UI design and implementation, these technologies, unfortunately, have not succeeded in solving the problem of how to integrate UIs with their underlying application nor in reaching a consensus on which set of models is most suitable for describing UI.

A small but notable and (mainly experimentally) important subset of UI technologies are those based on generic UIs. There has been a rich history of research in the area of automatic UI generation, mainly in the model-based UI community [27]. A generic UI is a presentation layer of an application obtained directly from the application's model. The model can be a structural or behavioral UML model, a task model, a relational model, or a model of any other kind or a combination thereof. Orientation to generic UIs saves development time, is less platform-dependent, and decreases the number of bugs, while UIs are clear, coherent, uniform, and well structured. Generic UIs, however, fail on flexibility challenges and suffer from the "low ceiling" syndrome and unpredictability [26]. However, generic UI approaches show that with extensive and direct coupling with the domain object space, even applications with generic UIs can be quite successful. Pawson's Naked Objects [32], successfully confirmed in production, are one of the best known examples. Other interesting generic UIs can be found in literature [6, 19, 21, 25].

The SUIDT tool [2] uses the application domain model on one, and abstract and concrete task models on the other hand to get the resulting UI. However, it seems that in order to create a target UI, the developer must first conceive at least blurred pictures of the target UI, then invest a considerable mental effort to transform these pictures into the abstract and concrete models, and finally, to generate the target UI from these models. It would be much easier just to create the target UI directly from these mental pictures, instead of making these conceptually distant intermediate forms. Furthermore, the example given in the work on SUIDT [2] indicates how complicated these models can be even for tiny interactive applications with a couple of fields and buttons.

JUST-UI [24] is a UI requirements specification and modeling approach with UI generation ability. Its requirements specification phase is founded on the domain model. UI modeling is then

conducted using the information captured in the previous phase and a set of coarse-grained UI patterns. Each pattern can be expressed in terms of abstract interaction objects (AIO). In the UI generation phase, these abstract objects are translated into concrete interaction objects (CIO) using some rules and transformation tables, comprising the final UI. JUST-UI has a high abstraction power (due to presentation patterns and reliance on the domain model), extensibility (based on introducing new UI patterns), and enviable platform independence (due to platform-independent UI models). However, it is unclear how each UI presentation pattern maps into implementation (a set of AIOs), nor how interaction objects communicate with the domain object space.

In another approach [34], task models are used along with abstract UI models (AUI) and concrete UI models (CUI) in order to address multi-target (multi-device) user interface design, prototyping and development. The main contribution of this approach is in platform-independency. Flex-based desktop and small-device prototypes are generated as the final UIs. However, the relationship between task and AUI models is left unclear. It seems that a lot of work needs to be done in order to specify both types of these models.

UML*i* [8] is a profile of UML for modeling user interfaces. By providing stereotypes for modeling abstract categories of UI controls, like inputters, editors, and containers, it preserves the model from premature commitment to specific implementation-dependent widgets. In its structural part, it models the usual containment relationships of UI controls. The behavior and interaction of UI controls is basically modeled through UML activity diagrams. Consequently, even a very simple interaction of UI controls, like reaction on an event or coupling two adjacent UI controls in a non-trivial manner, requires significant modeling effort and thus does not scale well. In addition, it is not fully clear how UML*i* models map to an implementation.

EUIS (Enterprise User Interface Specification) [33] is another example of a recent domain-specific language, defined as a UML profile, for specifying layout and behavior of coarse-grained objects for business applications. It is based on a specific framework with a prescribed set of concepts and principles of building form-oriented UIs from annotated (stereotyped) domain models, enabling generic or semi-generic navigation and data manipulation, as well as declarative binding of UI commands with the backend business logic. It generates complete code for UI in a specific implementation technology from the annotated UML class model. Although the approach of constraining the way how UI is organized and deriving it from the domain class model has advantages, such as uniformity and completeness of the UI, it may suffer from limitations of flexibility for specific UI tasks.

Z-Based Object Oriented Modeling (ZOOM) [15] is a UML-based software modeling approach that uses structural, behavioral, and UI models, and integrates them formally to build a complete application. It provides formal semantics, ensuring consistency between these models, and UI design notations. However, its component library is relatively thin, containing behaviorally depleted components. UI behavior is dislocated in a separate layer between the UI and the domain object space. This has several negative effects: UI components are rather anemic, the conceptual distance between the UI and the domain object space is increased, and finally, trivial inter-component behaviors are over-modeled, because all UI events must be handled by the model of the behavioral layer. On the other hand, separation of concerns is clear and enough evidence of platform-independence is given.

Intellium Virtual Enterprise [13] is UML-based modeling approach that uses class diagrams for modeling conceptual domain and activity diagrams for modeling operations and business transactions. Initially, CRUD-enabled, web-based application skeleton with a generic UI is automatically generated from these models. Then, obtained generic UI is manually adjusted using special web-based WYSIWYG editor. The authors emphasize that not a single line of code is required to produce highly scalable multi-tier application, however, the level of flexibility in terms of different UI patterns and behaviors of such kind of approach is questionable.

WebRatio [42] is model-based agile development tool. It uses WebML, a domain-specific language for modeling business applications with web-based user interfaces. A part of the WebML for conceptual domain modeling is semantically, and in terms of notation, very similar to the UML. In terms of the modeling presentation and interaction in the UI, WebML exhibits concepts for structural decomposition of web application on sections (areas) and pages with autonomous access rights. Also, it allows for modeling each structural UI part on the level of UI components that present or edit domain object space or perform other specific tasks. Finally, there is a set of concepts for modeling navigation and parameterized inter-component interaction. The built-in UI component library contains a set of components that couple UI with the domain model and by that reduce the level of accidental complexity overly present in the mainstream technologies. On the other hand, the UI library can be easily extended by custom UI components, which is crucial in terms of flexibility. The layout is done using HTML-based templates.

AgilePlatform [1] is very successful and well established commercial agile modeling approach aimed at high productivity and flexibility. It supports both Java and C# target languages reducing the level of platform dependence. Its powerful and fully integrated modeling environment allows for modeling all aspects of both web and mobile application development, namely: databases, business logic, workflow and business processes, security rules, and user interfaces. The environment provides WYSIWYG editor for UI design, and encourage encapsulation, reusability, and modularity of UI artifacts. It provides built-in visual styling themes as well. However, we dislike user navigation flow charts because they can hardly suffice for modeling navigation of modern, desktop-like, web applications where page as the main concept of user navigation slowly disappears.

State machines are frequently used in UI modeling approaches. Two recent papers in this area [48, 49] propose applying state machines in a combination with aspect-oriented programming principles. Being a well-established tool for modeling behavior, state machines performed well for modeling internal behavior of UI components. Aspects, on the other hand, introduced to bring separation of concerns in UI development, come up with their own problems and issues. These are well addressed by Goderis [10], where she highlighted the problems of aspect-oriented programming generally, as well as its applicability to UI design.

Summary and Criticism

Flexibility, extensibility, and development tools' support are the true power and strength of the mainstream UI technologies. On the other hand, their main advantage, the flexibility and virtually endless freedom in defining behavior of UI controls, turns into their main drawback. It opens a huge space for design flaws, anti-patterns, and programming mistakes. Mingling business logic with UI code, especially in event handlers, or designing tangled interconnections and interactions between UI components in different event handlers often leads to unwieldy UI code that lacks any clear architectural view and is difficult or impossible to understand, debug, and maintain. As Myers et al. [26] observe, a respectable UI tool should lead implementers toward doing the right things, and away from doing the wrong things. Furthermore, a rather low level of abstraction used by the mainstream technologies, as well as the absence of reliance on the domain models, typically results in huge numbers of UI elements (widgets, tags, and others). With no architectural views to organize these complex structures and behaviors, this proliferation can get intractable. Due to a notation mess, some of these technologies have also poor usability and steep learning curve [26]. Although very popular, template-based approaches have been proven not to scale well for large and complex applications.

On the other hand, model-based approaches rely on the domain and UI models extensively. They also inherently enable platform independence. The number of models used in each approach varies from one (in case of generic approaches based only on the problem domain model) to several (in case of other approaches based on combinations of domain models, UI composition models, task

models, and interaction models). These abstract models can get pretty complex. This especially holds for interaction models, because they inherit the semantics of the implementation level. Kovacevic [17] remarked, and we fully agree, that explicitly specifying every tiny detail of each interaction through an activity or sequence diagram can be quite tedious. Building and maintaining these models can be difficult and error-prone [25]. In addition, the existing model-based approaches do not address the problem of organizing huge volumes of UI elements in large applications.

3 MOTIVATION AND IDEA

The motivation behind our paradigm is illustrated in Figure 1. A simple UI form taken from a business application and shown in Figure 1a works as follows. The tree view on the left is configured to render the hierarchy of departments in a company managed by the application. Each department is an object of the class *Department* from the underlying UML structural model of the problem domain.

The three UI controls on the right in Figure 1a display the properties of the department currently selected in the tree view. The first two controls are editable text boxes that render and edit the *name* and *responsibility* attributes of the class *Department*. The third control is a list box that displays all employees who are members of the selected department. It is assumed that the assignment of employees to departments is modeled in the underlying conceptual UML model as an association between the classes *Department* and *Employee*.

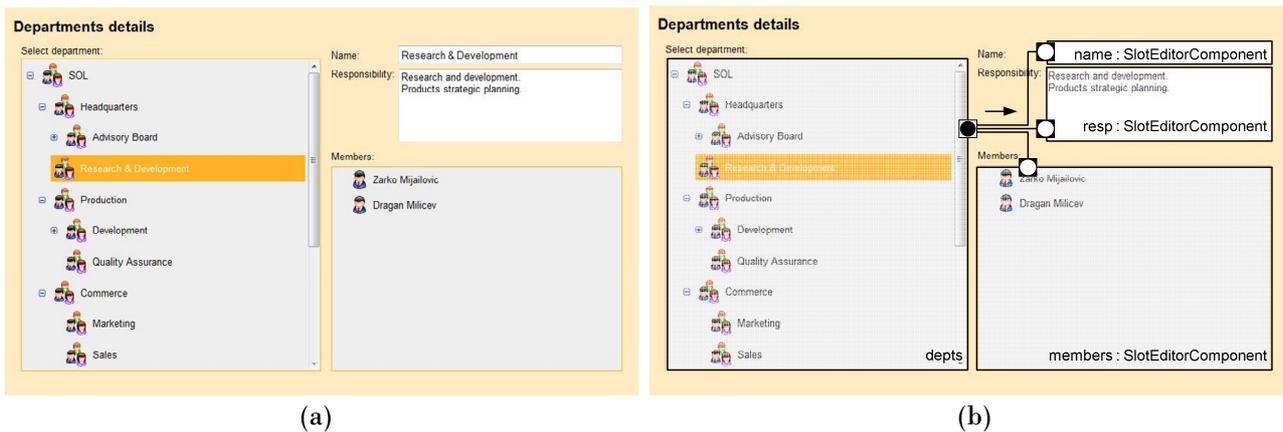


Figure 1: (a) An example of a form with coupled UI components. (b) The message passing perspective to the functional coupling of UI components.

The behavior of the UI controls can be abstractly described like this. Whenever an object of *Department* is selected in the left-hand tree view, each of the three right-hand controls has to display the value of the corresponding slot of that object (*name*, *responsibility*, or *members* – the collection of objects of *Employee* connected by links of the corresponding association). Obviously, the object of *Department* selected in the tree view has to be dynamically provided as the input parameter of each right-hand control. Thus, the property (attribute or association end) of the class *Department* whose value will be displayed by each of these three right-hand controls can be configured at design time, when the UI is being constructed. On the other hand, what changes dynamically, by user's interaction, is only the (reference to the) object of *Department* whose slot (as an instance of the configured property) has to be displayed.

By such observation we come to the perspective depicted in Figure 1b. The UI controls can be logically regarded as components with *pins* that form their interfaces. A *pin* is a connection point through which a UI component can send or receive signals or data to and from other components. For example, the tree view has one *output pin* through which it sends the (reference to the) object selected

in the tree view each time the selection is changed; of course, this selection is changed dynamically, by user's actions. Each of the three right-hand components has one *input pin* that accepts a reference to the object whose configured slot the component will display and edit. The internal behavior of the control ensures that each time a new value occurs on its input pin, the control accesses the underlying domain object space by reading the corresponding object's slot and reflects its current value on the screen.

In order to ensure the proper functional coupling of these components, the developer simply has to connect the output pin of the tree view component to the three input pins by *wires*. Wires specify the connections through which data will flow from an output pin to one or more input pins, whenever a new value is provided on the output pin, as shown in Figure 1b.

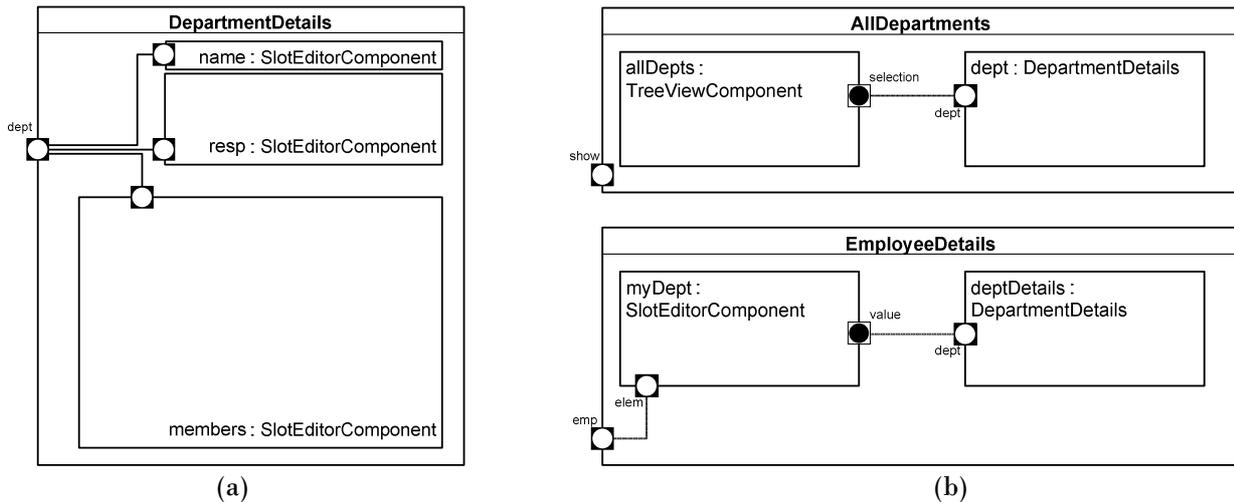


Figure 2: (a) A UI capsule is a structured class that abstracts a set of logically grouped and behaviorally coupled sub-components. (b) Once defined, a capsule can be reused in (referred to from) other capsules to appear as a fragment in various places of the UI.

Having taken the described perspective to UI components and their behavioral coupling, we have conceived an abstract and general modeling technique in which individual UI components or entire UI fragments of logically grouped and behaviorally coupled components represent abstractions modeled with what we call *UI capsules*. A capsule in a UI model represents a class with an optional internal structure. The internal structure consists of other capsules as its sub-components and wires between their pins. For example, the entire group of UI components that appear on the right-hand side of the form in Figure 1a can be regarded as a compact and reusable UI fragment responsible for displaying details of a department provided on its input pin; the fragment can appear in many other places in the application's UI. For that reason, the group of components forms an abstraction modeled with the capsule *DepartmentDetails* whose definition is shown in Figure 2a.² Once defined in this way, the capsule can be reused – referred to from other capsule definitions, as shown in Figure 2b. One such occurrence (reference) is in the capsule *AllDepartments* (Figure 2b) that appears at runtime as shown before in Figure 1. Another occurrence (reference) is in another possible UI form, modeled with the capsule *EmployeeDetails* in Figure 2b, which renders the properties of an object of the class *Employee* provided on its *emp* input pin. One of the properties of the employee is the department to which the employee is currently assigned, displayed by the *myDept* slot editor component. The value of that department is also propagated to the same input pin *dept* of the referenced capsule *DepartmentDetails*.

² The static labels that appear in the form are suppressed in the capsule definition for the sake of simplification and clarity.

As it can be concluded, the same UI fragment can be defined once and appear and behave the same in many parts of a complex UI. In addition, if the implementation of that fragment has to be modified or enhanced, the change can be done only in the definition of the internal structure of the *DepartmentDetails* capsule, that is, its implementation only. As its implementation is strictly encapsulated behind its interface (defined in terms of pins), it cannot affect any environment in which it occurs.

One obvious but important consequence is that such UI models are fully formal in terms of runtime semantics. This means, an executable UI can unambiguously be obtained directly and automatically from a model of UI capsules. For that reason, such a model not only forms a highly abstract architectural view to the construction of a large-scale UI that makes its management tractable, but also its real, executable specification and accurate design documentation. This fact has many important implications, such as avoidance of the so-called *rush-to-code syndrome* that many design and modeling approaches suffer from, as discussed in [38, 21].

In fact, the conceived paradigm and its concepts fully fit into and were inspired by a much older modeling approach invented for a completely different domain of applications – ROOM (Real-Time Object-Oriented Modeling) [38]. The perfect analogy is certainly not accidental, as ROOM was particularly designed and suited for modeling event-driven systems (although in the domain of embedded and real-time applications), while UI environments are inherently and traditionally event-driven as well. Since most of the fundamental concepts from ROOM have also been adopted in UML2, our paradigm and concepts are compliant with the definitions of UML2 too [29]. More formally, our modeling technique can be seen as a profile of UML2 particularly suited for modeling large-scale UIs. What we refer to as UI capsules³ are called *actor classes* in ROOM and *structured classes* in UML2. Pins and wires are called *ports* and *bindings* in ROOM, and *ports* and *connectors* in UML2. We have chosen different names to make a clear distinction of their profiled semantics and purpose, as it will be described soon. A complete and formal specification of the profile is provided in a separate document [23].

4 CONCEPTS

In this section, we elaborate the core concepts and principles of the proposed paradigm.

Capsule Structure: Interface and Internal Structure

UI capsule is a class that abstracts either an elementary (primitive, atomic) UI component, or an entire coherent group of logically and functionally coupled UI components or fragments that appear as an integral and reusable UI fragment. In the latter case, the capsule has its internal structure defined in terms of references to other already defined capsules and wires between their pins, as shown in Figure 2; these internal capsules form the parts of the enclosing capsule. For example, *deptDetails* as a part of capsule *EmployeeDetails* in Figure 2b is a reference to capsule *DepartmentDetails*. The semantics of capsule references is the same as the one of actor references in ROOM or properties (parts) of structured encapsulated classes in UML2. Namely, whenever an instance of capsule *EmployeeDetails* is created, an instance of *TreeViewComponent* and one of *DepartmentDetails* will also be created (along with their internal structure, if any) and interconnected via their pins. These two instances can be referred to from the scope of capsule *EmployeeDetails* over references *allDepts* and *dept*.

The described support for abstraction is a crucial feature of the proposed UI modeling technique for several important reasons. First, it provides the same concept of capsule for modeling UIs at high, architectural levels, as well as on low, detail levels. Second, defining a UI fragment as a capsule

³ The term *capsule* is borrowed from an intermediary form of the ROOM modeling concept of actor class that was used in Rational Rose for Real-Time modeling tool.

supports effective, large-scale reuse of repetitive parts of UIs. Third, modeling of capsule's internal structures is recursive, supporting a strategy to re-apply the same modeling concept and structure at successively lower levels of detail within a model [38].

The interface of a capsule is defined in terms of pins. As opposed to more advanced features of ports in ROOM, where ports can transport values of data types (called data classes in ROOM) according to defined protocols in both directions, or in UML2, where ports can provide and require any number of interfaces that can incorporate any kind of behavioral interaction (operation call or signal passing), we have simplified (profiled) the concept of port into the concept of pin as follows. A pin can transport only one reference or a collection of references to objects of classes or data types defined in the underlying UML domain model. An occurrence of a reference or a collection of references on a pin is treated as a signal (or message). A signal can also be a null value, meaning that its very occurrence merely matters, but that it does not carry any additional information. A pin can be typed with a class or data type from the domain model, meaning that it can transport only references to objects of that type (substitution is assumed). A pin can also be untyped or, alternatively, typed with a common generalization of all types in the model, meaning that it can transport references of any type; this is default. A pin also has a specified multiplicity, which constrains the size of the collection of transported values ([0..1] is default). A pin can be either input or output, meaning that it can transport signals in one direction only. Due to these semantic simplifications, we have chosen a specific name (pin instead of port) as well as notation for output (black circle on a white square) and input (white circle on a black square) pins as shown in Figure 2.⁴

A capsule's interface, defined as a set of pins, ensures a very strong and strict encapsulation. Not only is the interior of a capsule shielded from the exterior, but the encapsulation is also ensured in the opposite direction: the internal implementation of a capsule cannot directly access, rely, or depend on any component from its environment; it can only send or receive signals on its pins. This makes it fully pluggable in any environment that complies with its interface.

An output pin of a capsule's part can be connected to one or more compatible input pins of other parts within the same internal structure; compatible means that the type and multiplicity of the output pin must conform to the type and multiplicity of the input pin. At runtime, when the capsule instance with the output pin sends a signal to that pin, the signal gets broadcasted to all connected input pins of other capsule instances. Moreover, an input pin from a capsule interface can be connected to one or more compatible input pins of the capsule's parts from its internal structure; an example is pin *emp* connected to pin *this* in capsule *EmployeeDetails* in Figure 2b. In this case, any occurrence of a signal on the interface pin (*emp*) of an instance of the enclosing capsule gets immediately propagated to the input pin of the internal capsule instances (*this*). The similar holds for output pins of internal capsules connected to output pins of enclosing capsules. The precise semantics of handling signals in terms of flow of control and synchronization will be discussed later in the paper. Due to such semantic profiling, we call the connectors *wires*. The terminology (capsules, pins, and wires) has not been chosen arbitrarily: the semantics of UI capsules and their interaction (by sending signals over pins and wires) is very similar to those of digital electronic circuits, while our UI models in practice tend to resemble these circuits.

The described paradigm for modeling collaborative objects has many general advantages, as discussed in [38, 29]. In the context of UI development, it also eliminates the need for applying some design patterns that are commonly used to reduce dependencies among functionally coupled UI components and to improve their reusability in traditional event-driven frameworks, such as the Dependency Injection or Mediator patterns [9]. Design patterns are just a good practice whose proper application is fully up to the developer's skills and discipline, which can easily fail. As a result, it is a common outcome in usage of classical event-driven UI frameworks that UI components become very

⁴ This notation is adopted from ROOM.

tightly coupled by the code written in their event handlers, written as ordinary methods in classical OO languages. This often results in a tangled code from which it is difficult to understand how components interact and work. It is because features of components surrounding one component in a UI form are typically directly accessible (over references to those components) within the code of its event handlers. On the other hand, the proposed approach enforces encapsulation and loose coupling, as the only available mechanism for inter-component communication is through pins.

In addition, instead of writing boilerplate code of event handlers to functionally couple UI components, in our paradigm, the modeler simply connects the components (most of them readily available from component libraries) in a purely declarative way, visualizable through diagrams, while the message passing mechanism is implied and provides the necessary behavior. Such boilerplate code is a significant burden in building UIs with classical approaches. In our approach, the message passing mechanism is available out-of-the-box, as it is ensured by the implementation of the framework.

Capsule Appearance: Layout and Styling

The described perspective provides an abstract architectural and functional view on a UI construction. However, of the same importance in building a UI is its actual appearance on the screen. Our technique supports it with two aspects: layout of UI capsules and styling of UI.

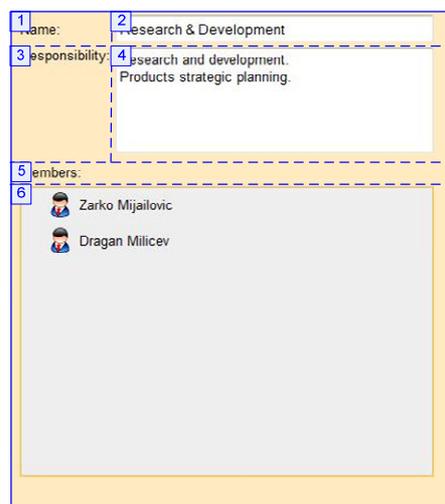


Figure 3: The layout view of the capsule *DepartmentDetails*.

Each instance of a capsule appears in the actual UI as a rectangular panel within an enclosing parent panel of its owner capsule instance (the outermost panel being the “main” panel of the application’s window). The layout of a capsule’s internal components is defined in the layout view of the capsule definition. The layout view provides a classical WYSIWYG approach to form design in the context of the proposed technique. The view allows the UI designer to lay out all components of the capsule on the rectangular panel of that capsule. An example of a layout view is shown in Figure 3 for capsule *DepartmentDetails* modeled in Figure 2a. Available to lay out are panels that represent the capsules immediately referred to by the former capsule; these appear as rectangular fragments with possible other nested UI elements.

This way, the layout view and the model view provide two different perspectives on the definition of the same abstraction: the former shows how it will actually appear on the screen, while the latter shows how it is conceptually structured and how its components interact. Of course, an appropriate modeling tool should allow the designer to toggle between the two views, or even to show

them at the same time, as indicated in Figure 1b: for the purpose of testing and debugging, the tool can interpret and run the functional definition within the layout view, let the designer test the UI components by providing values on their input pins, and show or even animate the dataflow between the components (over their pins and wires), initiated by user's events (clicks, entering values to input fields, etc.).

The level of support for concrete layout policies is left to the implementation. In our current implementation, any of the following layout policies, usual for Web-based systems, can be attached to a panel:

- Tab: the directly owned sub-panels overlay each other as tabs.
- Deck: the directly owned sub-panels overlay each other; any one of them appears on top when it receives a signal on its *show* input pin. Alternatively, the deck panel can receive an integer value *i* on one of its pins to display its sub-panel having the order number *i*.
- Vertical/Horizontal/Flow panel: the directly owned sub-panels are tiled one below the other (vertical panel) or one next to the other (horizontal panel), with optional overflow when the parent panel is shrunk (flow panel).
- Table: the directly owned sub-panels are positioned in the enumerated cells of a tabular layout.
- Tile: the panel can be arbitrarily tiled with horizontal or vertical separators, as shown in Figure 3. Its nested panels are positioned in its enumerated cells.
- Absolute panel: the sub-panels are placed in absolute positions in the *x-y* raster of the owner panel.

Altogether, these options allow very flexible and powerful design of UIs.

The described UI paradigm is applicable to classical desktop applications as well as to Web-enabled applications. Indeed, our current implementation described in a later section uses the described paradigm for Web UIs. Interestingly, this paradigm shifts the meaning of the traditional concept of a page in the context of an integral Web application, because the application's UI in our approach is, on the topmost level, a single panel. What appear to be "pages" (that fill in the entire browser's window or only part of it in turn) are simply child panels that overlay each other in a single parent deck panel.

The second aspect of an application's appearance is its styling. In our approach, it includes two elements. The first one deals with what is usually referred to as a "skin" or "graphical styling" of particular capsules. This includes, for example, colors, fonts, padding, alignment, and other elements of usual UI styling. For that purpose, a UI capsule can have UML's tagged values defined by the concrete implementation to refer to styling definitions supported by the implementation. For example, in our implementation, we use a tagged value to refer to a set of CSS classes applied to a capsule.

The second element deals with the look and behavior of single items that appear in the UI as icons with associated labels and typically render objects from the object space. For that reason, a separate UI model, orthogonal to the architectural UI capsule model is designed. It defines the so-called *UI item settings*, whereby each UI item setting specifies the elements of appearance and behavior of objects of a certain class from the domain model. For example, it defines how the icon, the label written next to the icon, the tooltip, and others are obtained (for example, as constant picture and text, or as a value obtained from the object), as well as its behavior on e.g. a double click on the item. Each UI capsule is associated with a *UI context*, whereby a UI context groups UI item settings that define how objects appear in that context. The same object, for example, can appear and/or behave differently in different UI contexts. A more detailed description of this aspect of our approach is out of the scope of this paper and can be found elsewhere [21].

Capsule Creation: Constructor and Parameters

The notion of capsule is an intrinsic impetus for abstracting recurring UI fragments of large-scale UIs into reusable structures. However, such fragments often occur as similar, but slightly different

structures in different places of a system’s UI. For example, the fragment abstracted into capsule *EmployeeDetails* shown in Figure 2b may be needed in different places with some variations: its *myDept* component may be needed as read-only (immutable) in some places, or its *deptDetails* component may be placed differently or even completely suppressed in some places. This calls for a method for defining parameterized and variable internal structures of capsules.

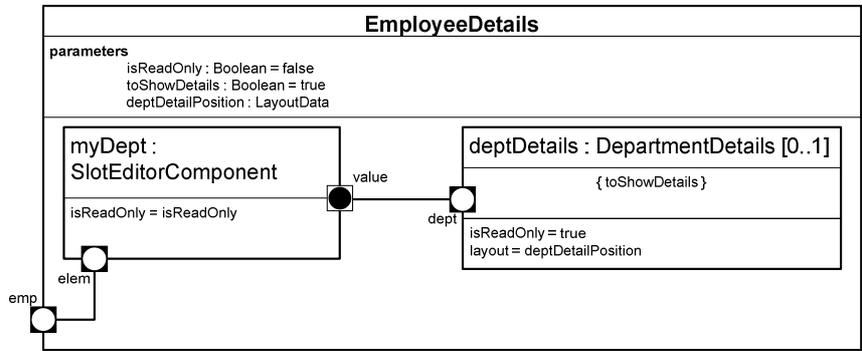


Figure 4: Parameterized constructor of capsule *EmployeeDetails*.

Figure 4 shows the specification of parameterized creation of capsule *EmployeeDetails*’ internal structure. The specification has the semantics of creational object structures described in [20, 21]. In fact, it is a specification of an implicit constructor method of the capsule. The constructor is executed to initialize the internal structure of a capsule instance on its creation. The parameters of the constructor are listed in the *parameters* compartment of the capsule definition. An element of the internal structure can be created conditionally, if a Boolean expression attached to the element’s specification evaluates to True. This is the case with the *deptDetails* part: the constraint `{toShowDetails}` attached to it is a trivial Boolean expression that refers to the value of the formal parameter of the constructor; upon creation of an instance of *EmployeeDetails*, the *deptDetails* part (and the wires connected to it) will be created only if the parameter is set to True. This is why the *deptDetails* part has `[0..1]` multiplicity (it is optional). The values of parameters passed to the constructors of components can be specified in the form *parameter = expression*, where the expression is evaluated in the scope of the enclosing constructor. For example, the value of *isReadOnly* parameter of *EmployeeDetails* capsule’s constructor is simply passed as the value of the same parameter of *myDept* part’s constructor. Although the parameterized internal structure definition can support many other sophisticated concepts of creational specifications described in [20, 21], conditional and parameterized creation suffices in most practical cases.

Capsule Behavior: State Machines, Operations, and Actions

In most cases of composite capsules with internal structures, those structures provide the necessary synergistic behavior of their composite capsules by ensuring proper connectivity and event and data flow via pins and wires; the composite capsule itself does not need to provide any additional behavior of its own. However, there are circumstances when it is necessary to define own behavior of a capsule.

The first such case is when the composite capsule has to manage its internal structure dynamically, by creating and/or deleting parts and wires between them after its own construction. In the ultimate case that will be mentioned later, the entire internal structure of a capsule can be polymorphic and dynamically constructed. The capsule can also dynamically manage the layout of its parts.

The second case is when the composite capsule has to do some logical transformation of messages received on its input pins before propagating the messages to its internal components, or the other way around.

The third example is the implementation of primitive capsules that have no internal structure (in terms of other component capsules), but whose implementation relies on native API of the underlying UI framework that is used as an implementation platform. These primitive capsules can be either generic (built-in, elementary) capsules provided by the framework, or complex, application-specific capsules.

Typically, in such cases, the capsule's own behavior has to orchestrate messages received from different sources. One source of messages are external capsules; such messages arrive through the capsule's interface input pins. Another source are internal capsules, such as UI widgets that handle asynchronous user's events (mouse clicks or key presses). Finally, messages can come from the underlying domain object space. These messages can be responses to the capsule's requests for reading the data that it renders, or notifications on changes in the object space. Obviously, in a most general case, all these messages may arrive asynchronously (at unpredictable moments) and independently (in unpredictable order). In particular, requests for reading from the underlying object space can be sent asynchronously to a remote server as it is the case in our Web-based implementation that uses AJAX calls for such requests.

Because of this unpredictability, state machines can be used for modeling internal behavior of UI capsules. We fully adopt the semantics of state machines from ROOM and UML2. In particular, we assume the run-to-completion semantics of event handling and transition processing: when one event occurrence has started being processed in a state machine, it cannot be preempted by another event in the same state machine until the processing of the first one is fully completed, that is, until the machine calms down in the target state. In case when behavior of a capsule is stateless, reactions to triggers can be modeled as simple internal transitions from the single topmost state to itself. Essentially, actions of such transitions play the role of methods of operations represented by the events. This is how the state machine model reduces to a simple behavioral model of operations and their methods (with inherent mutual exclusion).

Guards (conditions) and actions of transitions are statements written in a detail-level (action) language. This language can be any traditional object-oriented programming language selected as the implementation language for the framework. Typically, actions are simple invocations of internal operations defined in the capsule. Their semantics is the same as the semantics of classical operations in OO programming languages, except that they are always private or protected and inaccessible from outside the capsule; they can be called within the capsule's behavior only. This ensures full and proper encapsulation of UI capsules.

Methods of these operations are written in the detail-level language (action) language too. The code may include standard control flow statements (conditions, loops, operation calls) and actions. Actions particularly include access to the following features available to capsules' behaviors:

- 1) Maintaining the capsule's internal structure by creating or deleting parts and/or wires, according to the model of the internal structure. This is accessible through a specific API of the framework.
- 2) Sending messages to output pins. Each pin is accessible as an object in the scope of a method and a message is sent by calling a specific operation of that object. Sending can be asynchronous (non-blocking) or synchronous (blocking). For example, to synchronously send a message carrying the data value *val* to *value* pin, one can write:

```
value.send(val);
```
- 3) Reading or writing internal attributes of the capsule. Each capsule can have internal attributes whose values contribute to the overall extended state of the capsule. As opposed to attributes in

other programming languages, these attributes are always private or protected and thus fully encapsulated in the capsule. Attributes are directly accessible via their names in the scope of methods.

- 4) Accessing the domain object space in a restricted and controlled manner as it will be described in the next section.

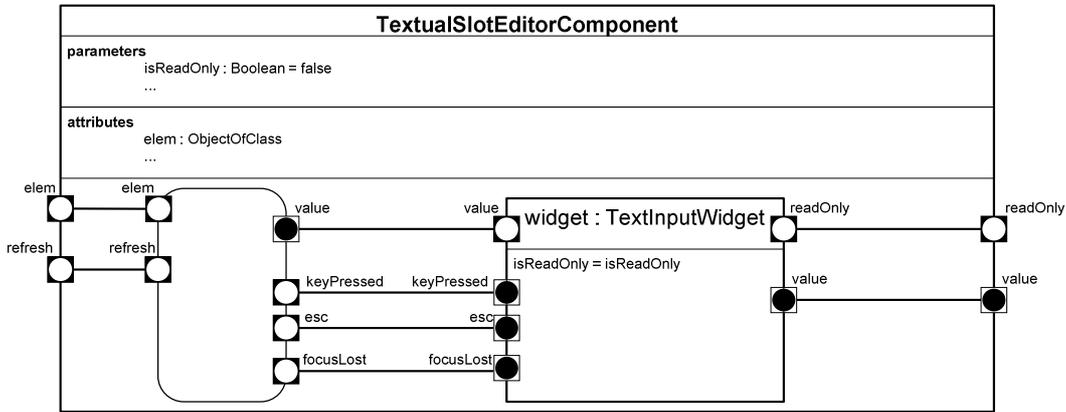


Figure 5: Internal structure and behavior with internal pins of a textual attribute editor capsule.

As illustrated in Figure 5, the behavior (shown as the oval symbol) of a capsule receives and sends messages only through its own pins, referred to as the *internal pins* of its owner capsule. These are pins that are private to the capsule and are not part of its interface. These pins can be connected to interface pins of the enclosing capsule that owns the behavior, or to interface pins of the internal component capsules. This way, even the internal behavior of a capsule is fully encapsulated because it can see its own pins only, and does not depend even on its owner capsule’s internal components or any other capsule directly. This makes behavioral models extremely flexible, because the internal implementation of behaviors, in fact, of action methods written in code, is independent of the internal structure of the capsule that owns the behavior.

As a consequence, interface pins of a capsule always operate as relays: they simply propagate messages from or to pins of either the internal behavior of the capsule or of its internal component capsules. The internal behavior itself operates as an ultimate “source” (generator) and “sink” (consumer) of messages.

Coupling with Object Space: Service Access Points

The UI of an application certainly has to access its underlying domain object space in order to display and update its state. However, the way the object space can be accessed from UI components is, in our opinion, a particularly sensitive issue, because it entails two somewhat opposing requirements.

On one hand, as we have already discussed, it is particularly important for efficient UI development that UI components (widgets) be tightly coupled with the underlying object space. These two have to have “matching impedance” and to be conceptually close to each other. That is, the UI components have to reflect the paradigm used for the underlying data representation. Otherwise, developers will have to write lots of boilerplate code to “match the impedance,” to fill in the conceptual gap and couple the widgets with the data.

We meet this requirement by providing a rich library of built-in UI capsules. These capsules, on one hand, wrap around usual UI widgets, such as textboxes, checkboxes, combo boxes, lists, tree views, pictures, and many more, providing a usual appearance and behavior to the user, as well as interface pins and other semantics of capsules to other capsules. On the other hand, these capsules raise

the level of abstraction and directly match with the underlying object space with the OOIS UML semantics [21]. As just one illustrative example, we present the UI capsule shown in Figure 6.

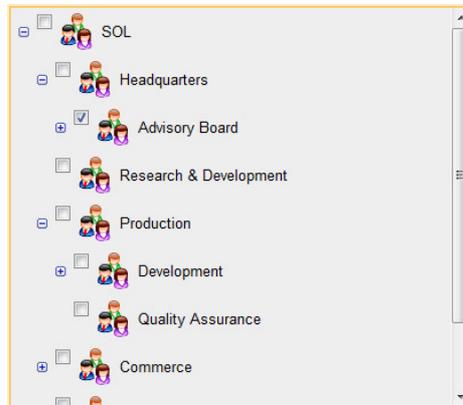


Figure 6: A sample UI capsule for editing an association end instance.

The capsule shown in Figure 6 is an editor of a slot of an object provided on its *elem* input pin. In the presented example, the object is an instance of the domain class *Employee*. Let us refer to it as the host object. The slot is an instance of an association end from the domain UML model. The association end is configured as a construction parameter (and stored in an attribute) of the capsule. In this example, this is the end *dept* that links an employee to its department. The capsule shows a collection of objects of a certain class that are candidates for creating links with the host object over the configured association end (*dept*). The capsule can be configured to show this collection of objects as a list, from a collection given on another pin, or as a tree, with the object given on another pin as its root and the tree defined by the styling configuration. Whenever a new *Employee* host object arrives on the *elem* input pin, the capsule fetches the link (or multiple links in a general case) of the host object on its *dept* slot from the underlying object space and updates the checkboxes to reflect this link (or multiple links in a general case). Whenever the user changes the state of the checkboxes, the capsule updates the underlying object space by creating and/or deleting the links. As a result, the developer has nothing to code in order to provide this typical behavior; everything is done declaratively, by configuring the (construction parameters of) the capsule and connecting its pins with pins of other capsules in its environment. Our library is full of sophisticated capsules that follow the same paradigm. Obviously, the reflectivity of the underlying UML object space plays a significant role in this paradigm, since the domain object space is accessed through UML reflection.

On the other hand, if the object space is fully and openly accessible to the UI layer, the application developers can easily fall into a trap of incorporating business logic in the UI layer. Virtually all traditional UI frameworks allow an unlimited access to the underlying object or data space. It is then exclusively up to the discipline of designers and developers to obey the clear separation of concerns and respect the well known and commonly advocated recommendations and design patterns that separate business logic from UI. However, without strict mechanical barriers, developers will always (at least unwittingly) break the discipline and make mistakes by intermingling code of different scope and level of details, especially business logic and UI-related functionality. It is a very common mistake, not prevented by classical UI frameworks, to write business logic code in event handlers of UI components (a symptom known as the “Magic Pushbutton anti-pattern”). As soon as event handlers of UI components (transitional actions in our case) allow (a) control flow constructs, such as conditions and loops and (b) full and easy access to the object space, such as reading and writing values and invoking operations, there will be an opportunity to incorporate procedures that logically belong to the business logic into UI event handlers.

To alleviate this problem in our approach, we strictly separate the UI and the object space into two different logical layers, as shown in Figure 7. The UI layer is the “upper” layer that consists of capsules that interchange messages “horizontally.” The lower layer is the domain object space that provides the proper data structure, such as a graph of linked objects, and the business logic, for example in objects’ operations. The two layers communicate by interchanging messages in the “vertical” direction. These messages have the same semantics and are treated in the same way by UI capsules as the messages interchanged in the UI layer. (Obviously, such logical separation and loose coupling allows for easy physical distribution of these tiers as well.)

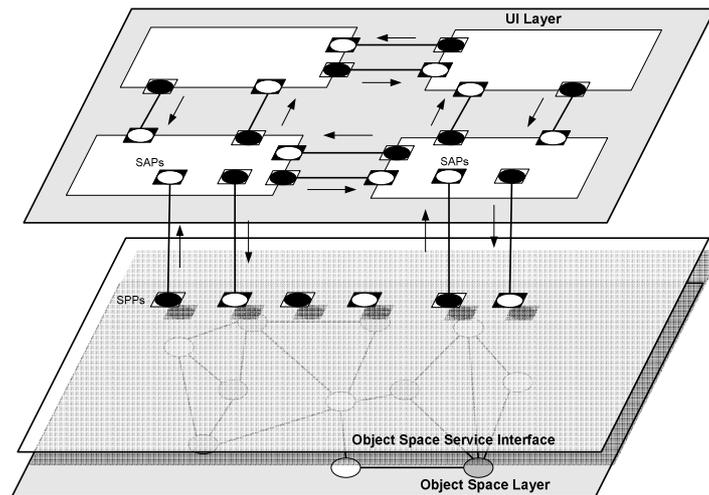


Figure 7: The layered architecture of the modeling framework

The Objects Space layer provides a partially predefined service interface consisting of pins that are called *service providing points* (SPP) as in ROOM (Figure 7). In order to connect with an SPP, a UI capsule has to declare a *service access point* (SAP) with a reference to an SPP. The SAP then appears and behaves within the capsule’s behavior as an internal pin connected to the corresponding SPP. By declaring an SAP, the capsule can receive or send messages to the Object Space layer via that SAP in the same manner as with other pins. As opposed to interface pins of capsules that can be connected only in the scope of the enclosing capsule, SPPs are available to the entire UI layer, that is, to all capsules.

Our Object Space Service interface provides a limited, but powerful set of services in terms of predefined SPPs whose detailed description falls beyond the scope of this paper. In brief, there are SPPs through which a capsule can request reading the value of an object’s slot (for example, retrieving the collection of objects linked over instances of associations or the value of an attribute), write an attribute value, create or remove a link between objects, and so on. The response to the request arrives on the *response* SAP that has to be declared in the capsule.

Another important inherent mechanism also incorporated in our paradigm is the mechanism of notifications of the UI layer from the Object Space layer. Namely, on each change in the object space, such as object created or deleted, or slot changed, the Object Space layer sends a notification message to one of the *notify* SPPs, depending on the kind of the change. By the same described mechanism of interlayer communication, the notification message is distributed to all UI capsules that have the corresponding SAP declared.⁵ They can thus react on notifications of a certain kind, provided that they

⁵ For practical reasons, in our current implementation, notifications are sent to a user session only for the changes made by a transaction issued by that same session. However, the concept of notifications is general and it is a matter of a particular system architecture and configuration to which user sessions the notifications are broadcasted.

are interested in the particular change, with updating the representation on the user's display. Essentially, this is a very simple-to-use surrogate (or manifestation) of the Observer (aka Model-View-Controller) design pattern [9]. In order to sign in a capsule for notifications of a certain kind, the developer simply has to declare the corresponding SAP and provide reaction on the notification messages in the capsule's behavior. In addition, our built-in capsules already embody such behavior. We have found this mechanism invaluable in practice, because it dramatically reduces the complexity of UI. It also removes the burden placed on the developer of taking care about refreshing parts of the UI depending on the updates in the domain object space. Instead, the developer is focused only on the domain structural (conceptual) model and business logic, while the UI is updated automatically, because the off-the-shelf widgets get notified on the changes in the object space. For example, whenever a domain object is deleted by any piece of business logic, all UI widgets that render that object in any form, or render any of its slots, will get updated automatically (by removing that object or its slot value from the screen).

In order to keep the UI layer free of business logic, we apply the following approach. The UI layer, more precisely, capsules' behaviors, deal with typed references to domain objects. These references can be stored in capsules' attributes or passed through pins or as construction parameters. They adhere to usual typing rules (subtyping, conversion, substitution). For example, a reference to a subclass (or subtype, in general) can appear wherever a reference to a superclass (or supertype in general) is expected. However, features of referenced objects, that is, their properties and operations, are inaccessible in the scope of capsules. (Features of instances of data types are fully accessible, because data types are considered as constant entity types, as discussed in [21, 22].) In order to reach or affect any feature of a domain object, a capsule's behavior has to send a request message to an SAP and possibly wait for a response. In fact, domain objects reside in a different "address space" than UI capsules – they reside in the Object Space layer. This way, the developer is strongly discouraged from intermingling business logic code with the UI code (capsule behavior). This is because it becomes extremely difficult to access an object's feature, such as to read one or more slots or call one or more domain operations, within the sequential code of a UI widget event handler (a capsule's method in our case). Instead, the developer has to issue a (typically asynchronous) call to a service of the Object Space Service interface and incorporate any complex business logic in the underlying object space, that is, in domain classes' operations. On the other hand, the UI layer is focused on the interaction between UI capsules, handling UI events, and interchanging signals and parameters, while the responsibility of handling domain behavior is kept with domain classes.

One notable example of a capsule from our library that meets both needs for impedance matching and for proper separation of concerns between the two layers is a command capsule illustrated in Figure 8. It can be rendered as a button or another widget such as a menu item or a hyperlink, for example. It has a number of universal input pins that can accept values provided from other capsules, as shown in Figure 8. It is configured with (a reference to) a prototype object of a command class as suggested by the Prototype and Command design patterns [9]. When the button is pressed, the capsule issues a special request to a specific SPP. The implementation of that SPP clones the configured prototype command object, passes the value given on the capsule's input pin(s) to a slot of the clone command, and executes the command. This is how parameterized actions are again easily configured with enforcing the proper architecture and separation of concerns. As there is no other way to invoke a domain behavior from the UI, business logic cannot be incorporated in the UI layer's event handlers, but is properly embodied within the domain object space and underpinned by design patterns.

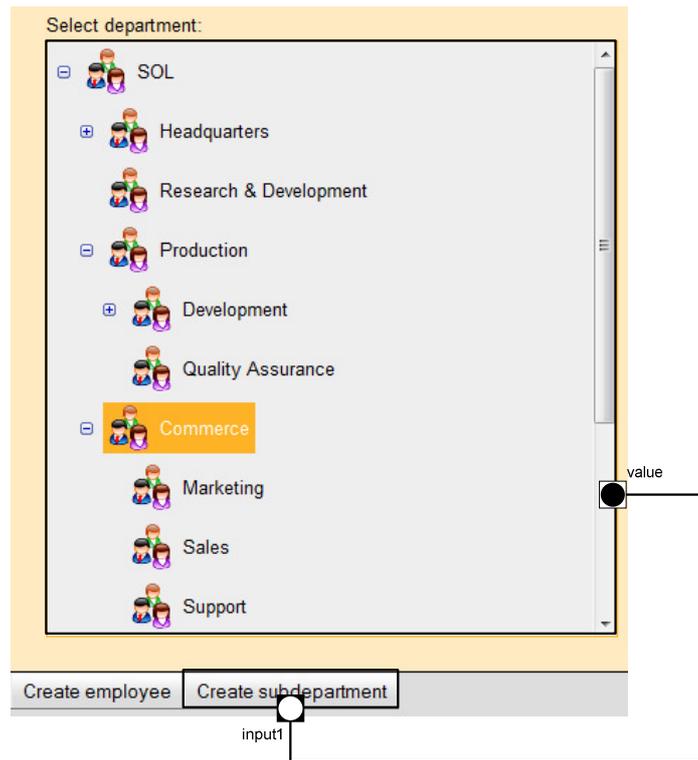


Figure 8: An example of a command capsule.

Sometimes, a UI fragment has to have its contents dynamically structured according to the current state of a piece of the domain object space or as a result of some specific business logic. For example, a number of certain UI controls have to be created, one for each object in a given collection obtained dynamically from a business procedure. For such ultimate purpose, we provide the concept of a capsule with dynamic internal structure. Such capsule has its interface and behavior defined as for any other capsule, but its internal structure definition in its constructor may be empty or may define just an initial or partial internal structure that may be changed later at runtime. One or more specific pairs of SPPs should be defined and implemented for each such capsule in the UI model, one for receiving a request, and the other for providing a reply. The behavior of a dynamic capsule can use an SAP to request the definition, that is, the creational specification of the internal structure of the capsule. This definition has to be constructed by the implementation of the SPP for that capsule. This implementation uses the API of the framework to construct the definition of the internal structure according to some specific business logic. When the definition is sent back to the requesting capsule instance, the underlying framework constructs the internal structure of the capsule instance in the same way as when it is being initialized. The same procedure can be repeated arbitrarily many times during the lifetime of the capsule instance, with the internal structure being replaced with the new structure arriving from the Object Space layer each time. By that means, the internal structure can be constructed dynamically at runtime and according to specific domain object structure or business logic, while the strict separation of concerns is still preserved: the business logic remains with the domain and does not get intermingled with the UI layer; that logic deals only with constructing well defined and encapsulated fragments of the UI model only, and does not deal with the dynamics of the behavior of the UI. The details of this mechanism fall beyond the scope of this paper.

Capsule Generalization/Specialization: Inheritance and Substitution

UI capsules are structured classes with interfaces that ensure their strict encapsulation. Being compliant with structured classes in UML2 and analogous to actor classes in ROOM, UI capsules can be related with generalization/specialization relationship. The rules and semantics of this relationship follow those from ROOM and are briefly summarized here for completeness.

The essential meaning of this relationship is the well-known substitution rule (aka the Liskov rule): an instance of a specializing capsule can always be a substitute for an instance of the generalizing capsule.

The specializing capsule inherits the interface of the generalizing capsule. The specializing capsule can extend and/or redefine that interface, provided that the substitutability is preserved. In particular, the specializing capsule can extend the interface by adding new pins, or can redefine one or more inherited pins by preserving their connectivity (covariant for output and contravariant for input pins).

By default, the specializing capsule inherits the internal structure, as well as the behavior from the generalizing capsule, but can extend them or redefine any of their parts. Obviously, redefining the implementation without changing the interface in the specializing capsule does not affect the substitutability.

5 IMPLEMENTATION

We have implemented a runtime environment for the presented technology within our SOLoist™ framework (www.soloist4uml.com)⁶. SOLoist is a framework for rapid model-driven development of information systems based on executable UML models [21]. The implementation supports the described mechanisms of capsule instantiation, inter-capsule communication via pins and wires, and access to the object space, including automatic notifications.

The implementation is Web-based. The Web client tier implements the entire UI layer described in the previous section, while the Object Space layer resides on the Web server. The client tier uses Google Web Toolkit (GWT) as its implementation platform. In particular, the off-the-shelf capsules from our capsule library wrap up GWT widgets with the semantics of capsules. According to GWT's philosophy, the programming language used for coding is Java. This is also our detail-level language used for implementing behavior of capsules. The Java code of the Web client tier is then compiled into JavaScript by the GWT compiler.

The server tier maintains the object space with the OOIS UML semantics. It also provides SPPs to the client tier implemented as Java servlets. The client tier accesses SPPs via AJAX calls. Notification messages are carried by HTTP responses to the clients.

It has turned out that the philosophy of GWT, being our lower-level implementation platform, perfectly matches our UI building paradigm. Our paradigm complements GWT by providing a higher level of abstraction. As a result of this match, the look-and-feel of Web applications designed with our framework is especially attractive. Namely, when such application is started (by entering its URL in the browser), the entire application – all capsule instances and their panels (except from capsules especially tagged as having deferred loading or with dynamic internal structures), get downloaded to the client. This normally takes one or a few seconds. From then on, the entire application's UI – all panels, forms, and controls, reside on the client all the time. Different panels are just hidden behind the currently displayed panel and appear on the screen by getting on top of the panel stack very quickly, as soon as they receive *show* signals on their pins. This ensures a feel of immediate response on user events, just as in desktop applications. This is because all the JavaScript code that implements the UI layer is completely executed on the client and there is no any communication with the server over the

⁶ The SOLoist demo site brings live demos of the implementation and of a prototype debugger for UI.

network or any load on the server for classical page retrieval. The only communication with the server happens within AJAX calls issued by individual capsule instances when they need to fetch and refresh the values they are currently displaying, modify these values, execute commands, or issue other calls to SPPs (including deferred and dynamic loading). Besides, this communication with the server may happen independently of what is currently displayed on the screen, in the background, because it is initiated by the message flow between capsules, regardless of whether the capsules are or are not currently visible on the top of the panel stack.

Our implementation encompasses many other technical solutions and optimizations that make it scalable and applicable to large industrial systems. For example, we optimize the communication between the client and the server by collecting multiple individual requests issued independently from different capsule instances into a single bulk HTTP request that is sent to the server. Such request is then unpacked and processed on the server. In addition, if a bulk request has multiple individual requests with the same contents, for example, reading of the same slot, only one access to the object space is issued and the same result is responded to all requests. This is all transparent to the application. The overall result is that the bulk of the workload related with the UI appearance and behavior is now moved from the server to the client (in fact, to JavaScript code executed within the browser), while the server deals only with what falls into its responsibility: maintaining the object space and executing business logic.

In addition to the runtime environment, we have implemented a very useful library of capsules. It consists of about thirty configurable and variable, ready-to-use capsules that provide easy and abstract access to the object space. For example, there is *SlotEditorComponent* capsule that is configured with the given class property and accepts the host object on its input pin, as described earlier in Section 3. Then it dynamically creates a concrete widget, such as a combo box, checkbox, text box, picture editor, etc., depending on the type of the attribute. We also provide a number of object collector capsules that provide single objects or multiple objects on their output pins according to different available configurations. For example, a collector capsule may retrieve all objects of a certain class, or all objects of a certain class having the given value of a certain attribute or satisfying a certain criterion, etc. Of course, the provided collections are updated dynamically on each significant change in the object space over the described notification mechanism. There are many other capsules in our library, such as command controls, tree views, lists, grids, query result tables, as well as invisible capsules that implement different logic such as negation of Boolean signals, multiplexing of input signals, polymorphic type resolution, and many others.

We implemented the inter-capsule communication mechanism, which is conceptually a message-passing mechanism, with simple synchronous operation calls between objects in Java (later compiled to JavaScript). These operation calls ensure that the executions of both the sender and the receiver capsules take place in the same thread context and address space of the Web browser, thus introducing no additional performance overhead in the client tier when compared to traditional event-driven systems, and no workload for the server. This is why the implementation ensures seamless reaction of UI components triggered by events issued from other components, with responsiveness, look and feel as in desktop applications.

It should be noted that the same efficient implementation approach can be taken even for message passing between the UI layer and the Object Space layer, when they reside in the same physical tier, as in desktop applications. In that case, the described paradigm ensures proper logical separation of the layers, while the implementation can use simple operation calls to efficiently couple the layers at no additional overhead and with good responsiveness. On the other hand, it allows for seamless distribution of the layers in case of tiered architectures, such as Web.

As a proof of concept and the applicability of the proposed paradigm to development of contemporary UIs for mobile device applications, we have made another prototype implementation for

Android smart phones. As expected, it provided the usual look and feel of typical Android UIs, including the responsiveness, but with using a different development paradigm.

A more detailed description of our implementation and its capsule library falls beyond the scope of this paper and will be published elsewhere. Partial descriptions of (an earlier form of) the library can be found elsewhere [21]. Our new modeling environment for the described modeling approach is under development.

6 EMPIRICAL EVALUATION

To substantiate the presented claims, in the three subsections that follow we present results from three campaigns that we have conducted to empirically evaluate our proposal. The first campaign was aimed at evaluating the practical applicability of the proposed technique to large industrial projects and its level of acceptability and perception by different developers other than the authors. The second campaign was oriented towards analytical comparison of different development-related technical aspects of the proposed technique and a set of selected mainstream approaches. The third campaign was aimed at evaluating performance aspects of our current implementation and its comparison with the selected mainstream approaches. The second and the third campaigns were thoroughly described in the appendix “Analytical Comparison with Third-Party Tools”.

Practical Applicability and Acceptability

Goals and Assumptions

The goal of this campaign was to evaluate the following:

- a) practical applicability of the proposed approach to large industrial projects; this included examination of:
 - whether the vast diversity of requirements and needs (in terms of UI) of real applications can be successfully met by the proposed technique;
 - whether the given technique can be applied to large applications, with providing the metrics of the scale of those applications.
- b) the level of acceptability of the proposed approach by developers other than the authors; this assumed examination of:
 - whether the proposed technique can be adopted and used by developers other than the authors;
 - the feedback from other developers.

Experiments and Reports

We conceived the perspective on how UI components interact described in the Motivation section as early as in 2004. Before the first version of a framework that supported the approach in a generic way was implemented some time later, the authors and the colleagues from their development teams had used the concepts of pins and wires for describing dynamic interaction of UI components just conceptually, for sketching and designing UIs, prior to implementing them in classical event-driven and other UI frameworks by ad-hoc techniques of manual coding. That approach was used in a few small to medium industrial projects. After that period of positive preliminary experience, the first version of the framework for desktop UIs was implemented. It was used by the authors and several other developers in another few mid-sized to large projects from very different domains, including commercial, administrative, and engineering applications. During that period, we improved our understanding of the concepts and enlarged our base of generic UI capsules.

Our current Web-based implementation is, therefore, the second implementation of the UI framework, developed in 2009. Since then, the technology has been used in four industrial projects of different size conducted in the period 2009-2012.

The first project dealt with an Event Management System (EMS), a highly customizable product for large social events of different kinds, like sport competitions, conventions, symposia, and the like. It supported registration of participants, approvals of registrations, accreditations, access definitions, pass layout configuration, pass printing and issuance, maintenance of venues, scheduling of events, and others.

The second project was to develop a governmental Human Resources Management System (HRMS). It supported central management of open positions in all governmental organizations nationwide, applications for posts, profiles and testing skills of candidates, management of employment processes, and many others.

The third project was about a national Real-Estate Cadastre System (RECS). It supported all standard concepts and functionalities for such a system, including real-estate entities (parcels, buildings, apartments), as well as legal concepts (ownerships, encumbrances, mortgages).

The fourth project was about a specialized Customer Relationship Management System (CRMS) built for a real-estate company to cover real-estate selling processes, including administration of involved parties, projects, houses, products, communication with clients, document management, online house configurations with standard and special options, change requests, and pricing. The project included integration with several different third-party UI-related products, challenging the integration potential of our framework.

The described applications all had very rich UIs with tens to hundreds of entry, display, and search forms. The diversity of the UI was significant and was primarily dictated by the customers' requirements and business domain needs. The applications included also several custom UI components, some of them incorporating third-party Web widgets. Figure 9 serves as a very rough illustration of the diversity of the applications, showing a few sample screenshots taken from these systems. By this experience, we believe we have proven that the vast diversity of requirements and needs (in terms of UI) of real applications can be successfully met by the proposed technique.

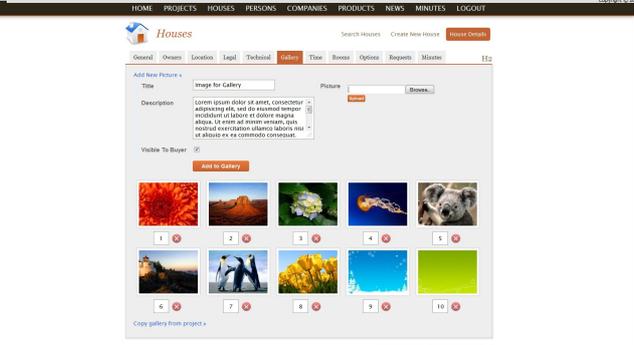
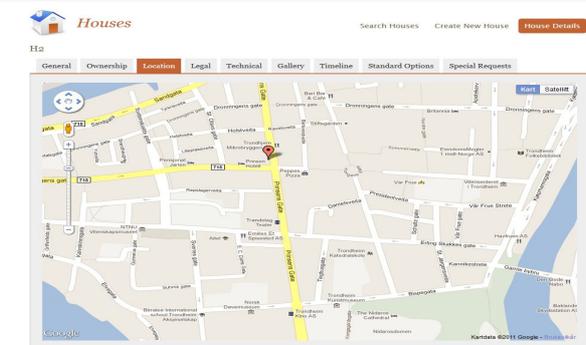
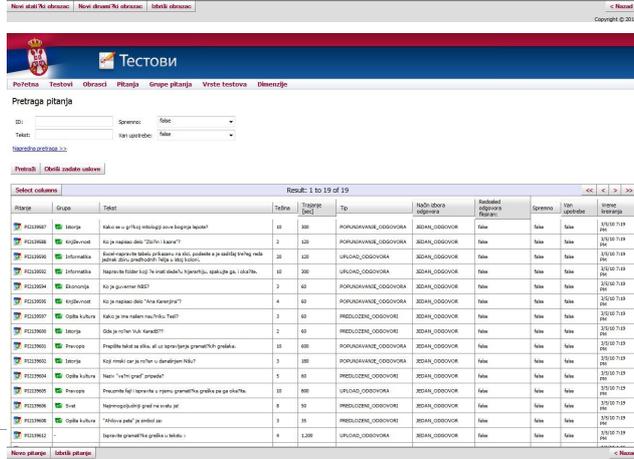
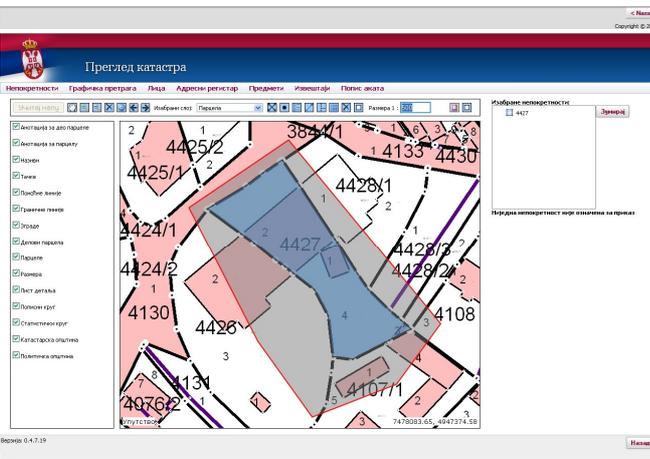
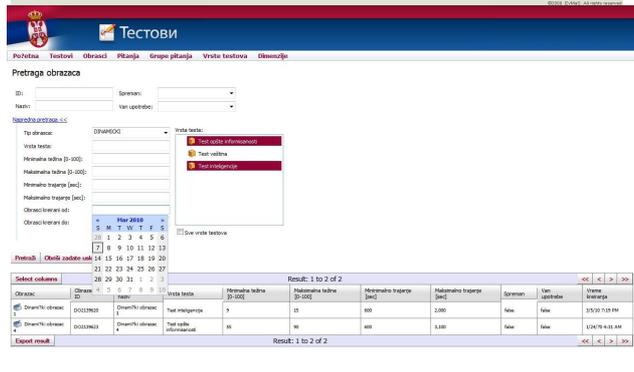
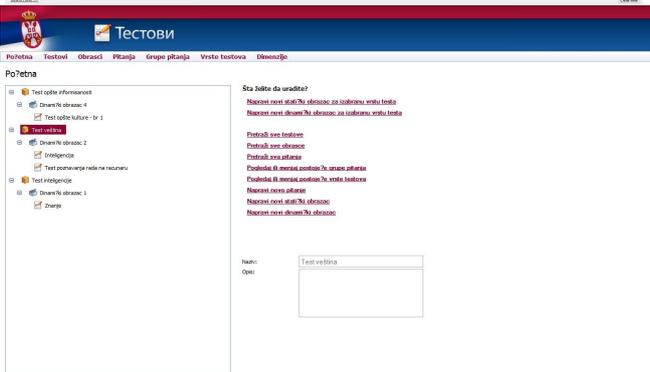


Figure 9: Sample screenshots from industrial systems.

Table 1 gives an overview of the size of UML domain models for these four systems. The metrics cover structural parts of the models (in terms of classes, attributes, and associations), as well as non-trivial business functional points of the systems (in terms of command classes). Note that the metrics cover only domain (persistent) classes from the conceptual models, not any of UI-related ones. In some sense, therefore, these figures give an impression about the essential complexity of the systems, free from the noise of accidental complexity.

<i>Metrics</i>	<i>System</i>			
	EMS	HRMS	RECS	CRMS
Number of domain classes – total/abstract/concrete	81/13/68	86/5/81	196/17/179	28/4/24
Number of domain attributes – total/max/avg. per class	218/13/2.69	566/52/6.58	280/17/1.43	130/22/4.64
Number of domain associations/association ends	79/158	132/264	254/508	32/64
Number of navigable domain association ends – total/max/avg. per class	139/11/1.72	228/18/2.65	336/29/1.71	50/6/1.79
Number of properties (attributes + navigable association ends) – total/max/avg. per class	357/18/4.41	794/54/9.23	616/46/3.14	180/27/6.43
Class hierarchy size – max depth/avg. depth	4/1.95	3/1.24	5/2.36	3/1.50
Number of domain enumerations	10	32	15	13
Number of command classes	63	186	193	70
Total number of classes (domain + command)	144	272	389	98

Table 1: The metrics of the UML domain models of four industrial systems.

Table 2 gives the metrics of the UI layer of the four systems. The figures correspond to one instantiation of the entire UI model of the system, typically the size of the UI for one user session that covers the entire scope of the system.

<i>Metrics</i>	<i>System</i>			
	EMS	HRMS	RECS	CRMS
Total number of UI components	16,883	6,925	59,448	1698
Component containment hierarchy – max depth/avg. depth	15/8.55	21/11.23	32/16.40	16/10.65
Number of container components	2,058	1,928	18,046	395
Number of primitive components	14,825	4,997	41,402	1303
Number of wires	12,104	4,255	31,285	1221
Max/avg. number of components owned by one structured component	101/8.22	65/3.60	57/3.28	28/4.29
Number of developers involved in UI development	5	7	10	3

Table 2: The metrics of the UI of four industrial systems.

We have concluded from this experience that the proposed technique works quite well for all sizes of projects, as the given results report. We also briefly discuss here some other lessons learnt from our experience.

In the first two projects (EMS and HRMS), our approach did not enforce the use of capsules as abstractions with internal structures and strict encapsulation. Instead, it only supported hierarchical containment of capsule instances, as well as connectivity of pins through wires, as described in [21]. Although the development of UIs was satisfactorily efficient, the maintenance of UIs was an issue: as soon as UI structures became more complex and bigger, the wiring became tangled and very difficult to understand, debug, and maintain in some cases. This was because wires were allowed to cross boundaries of UI fragments and it was often difficult to realize the entire wiring around some fragments.

This is why we introduced the concept of capsules as abstractions with strict encapsulation aimed for reuse. We used that approach in the last two described systems (RECS and CRMS) and immediately noticed a substantial improvement and solution to the described issue. In addition, the concept of capsule has dramatically advanced the potential for reuse of UI fragments, which was an important aspect in those systems.

We have also concluded that the notification mechanism is very important and convenient, as it makes the development much easier than in other conventional approaches. In addition, we have noticed that our UI models, as well as the way of thinking about their design, very often resemble those of digital circuits design. This is easy to understand taking into account their close semantics, and the developer’s conception that UI capsules behave as electronic circuits that send and receive signals over pins and wires. This effect supports the clear separation of UI from the business logic even stronger, because developers tend to couple capsules directly or over capsules that provide simple logic only, leaving more complex behavior to the object space layer.

In its last row, Table 2 gives the number of developers involved in the development of the UI of each of the systems (this is not the total number of developers involved in the project; there were other developers involved in non-UI parts only). The total number of persons that adopted and experienced the proposed technique in these projects, not counting in the authors of this paper, is 15. The included junior developers as well as senior developers very experienced in several UI mainstream development technologies. They were coming from three different software development companies.

We also asked these developers to give us their feedback on the proposed technique by filling in a questionnaire. In the questionnaire, they were asked to select another UI development framework they were most familiar with or they considered as the best of those they were familiar with, and to compare different technical aspects of that other framework with our technique. For each aspect, the other and our techniques were graded with marks 1 to 5 (1-unsatisfactory, 2-satisfactory, 3-good, 4-very good, 5-excellent).

We received the feedback from 9 developers, while one of them returned the comparison of ours with 5 other technologies. There were 8 different technologies among those selected for comparison: Swing, GWT+MVP+UiBinder, JFace+SWT, JSP 2, Ruby on Rails, C#.Net with Visual Studio, Android, and Spring. Table 3 presents a brief summary of the average marks we got for different aspects of our and the other technologies.

<i>Aspect</i>	<i>SOLoist Average (X)</i>	<i>Others Average (Y)</i>	<i>Difference (X-Y)</i>
Encapsulation support/encouragement/enforcement	3.38	2.55	0.83
Code readability and organization	2.89	3.17	-0.28
Support for UI fragment reuse	3.44	3.75	-0.31
Quality of built-in components	3.75	3.36	0.39
Convenience of implementing inter-component behavior	4.44	3.17	1.28
Convenience of coupling UI with data and business logic	4.44	3.75	0.69
Convenience of defining layout	3.11	4.25	-1.14
Convenience of styling	2.89	4.17	-1.28
Convenience of defining custom components	2.67	4.42	-1.75
Support for clear separation of UI from business logic (prevention of intermingling the two)	4.63	2.82	1.81

Table 3: A brief summary of the results of the poll among developers that have adopted the technique.

It is important to notice that the experience of the developers interviewed in the poll was for the most part gained from using the version of our technology available before we introduced the notion of capsule with the support for strong encapsulation, UI fragment reuse, and better organization of complex UI. This has obviously affected the assessment of the first three aspects in Table 3 and may partly explain somewhat lower average marks for code readability (2.89 vs. 3.17) and support for UI

fragment reuse (3.44 vs. 3.75). Even so, our technology got a significantly higher average mark for encapsulation (3.38 vs. 2.55). We have strong reasons to believe that the assessment would be much better with our improved technology as presented in this paper.

We got higher marks for the aspects that we described as the most relevant contributions of the technology: quality of built-in components (3.75 vs. 3.36), convenience of implementing inter-component behavior (4.44 vs. 3.17), convenience of coupling UI with data and business logic (4.44 vs. 3.75), and especially for clear separation between UI and business logic (4.63 vs. 2.82).

We got lower marks for the aspects that are practically important, but not particularly characteristic or original in our approach: for convenience of defining layout and styling, and for constructing custom components. We explain this with the immaturity of our implementation and the lack of proper tooling that does not yet support these technical aspects on the same level as other mainstream technologies do. We are going to improve these practical aspects in our future development.

In the freeform part of the questionnaire where the developers could freely comment on advantages and disadvantages of our technology, the developers commended the following features: easy and declarative way of interconnecting UI components, easy access to the data and business logic and good coupling with the UML domain model, strict separation of the UI layer from the business logic, built-in data refreshment (notification) mechanism and transparent AJAX support, and intuitive concepts and easy adoption. They criticized the immaturity of the implementation, especially in terms of supporting tooling, reduced flexibility in terms of customizing components and some limitations in performing simple, low-level, Web-specific programming tasks.

Analytical Comparison

Goals and Assumptions

In this campaign, we tried to compare some development-related technical aspects of the proposed approach with a set of selected mainstream UI development frameworks by assessing applications developed in these technologies. In particular, we were interested in assessing:

- a) quality of code; in particular, to what extent the technology prevents some bad coding practices;
- b) expressiveness of the programming technique, in terms of the size of code necessary to perform a certain task.

It is important to emphasize that our intention was *not* to show or imply that any technology is generally better in the sense that it ensures making good design with less code. In fact, we believe that every technology allows for making good programs: a well-trained, skilled, and disciplined developer can make a good program with more or less effort in any framework. Instead, we wanted to examine the quality of the code from *real* applications developed independently and in an unbiased way.

Therefore, in order to be as objective as possible and obtain as reliable results as possible, we did not want to develop applications in different technologies from scratch, or have them developed by us or somebody else involved in the experiment: that way, we would have run into a risk of biasing the comparison due to the potential insufficient expertise in other technologies of those involved in the experiment, or of obtaining perfect code because they would have been aware that the design and code would be examined. Instead, we have chosen the following approach.

Apart from our technology, we first selected three other very popular technologies to compare with, from different groups of technologies:

- 1) Django/Python, representing the template-based approaches,
- 2) GWT/Java, representing the object-oriented approaches,
- 3) ASP.Net/C#, representing the hybrid approaches.

For each of these three technologies, we have selected two open-source business applications already available on the internet and developed by teams completely unknown to and independent from the authors of this paper. By selecting two independent applications produced by different and independent authors or teams for each technology, we wanted to reduce the impact of group and individual skills and discipline on our analysis. In addition to these three other technologies, for our technology, we selected two our newest applications. In these applications, we analyzed the UI models translated to Java code: capsules are represented as Java classes, while Java initialization code was written for the constructors.

For each of the eight selected applications, we selected three UI modules (pages or files): except that we tried to select typical and average modules or pages, the selection was arbitrary. For our technology, the modules were developed by people different from the authors.

For all these modules, we collected different metrics by manually analyzing the code and counting the lines of code that belong to certain categories, as described in the following subsection (one line of code can belong to several categories). Although lines of code are disputable metrics, it is still reasonable to believe that they at least roughly or indirectly indicate some aspects of program complexity.

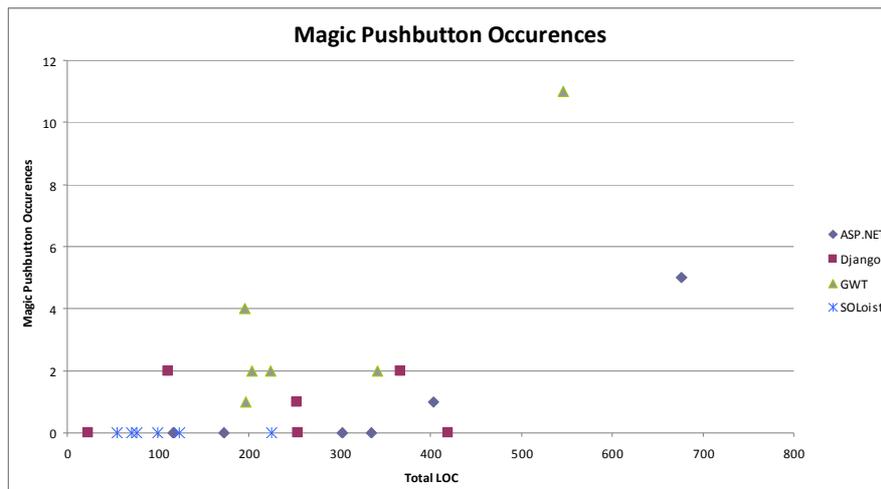
By taking this way, we believe that we ensured an unbiased examination of different aspects and phenomena that really occur in common industrial programming practice and reflect how the technologies are actually used and perform in practice, instead of theoretically analyzing their potential usage.

Experiments and Reports

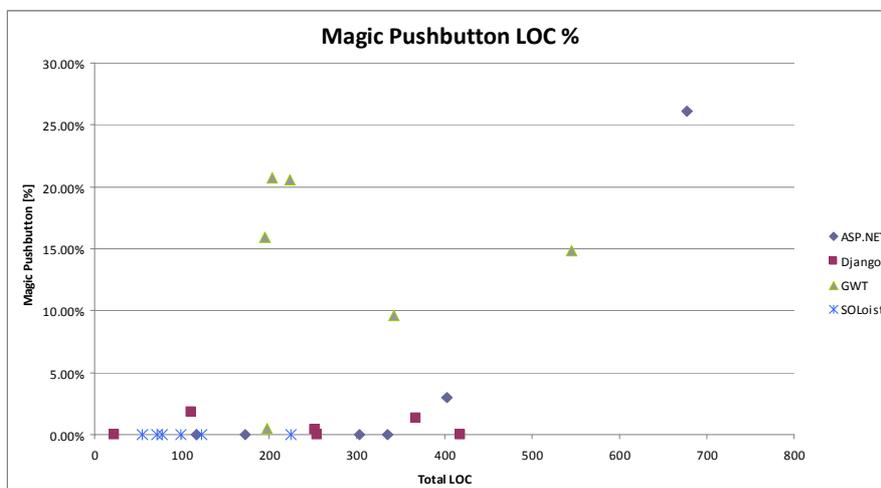
We will publish a detailed report on our extensive analysis in a separate article, while here we present only some results that are most relevant for the subject of this paper.

The first relevant analysis examined the level of presence of the bad practice of intermingling business logic code with UI code (the “Magic Pushbutton” anti-pattern). In each analyzed module, we counted the occurrences of this anti-pattern, as well as the lines of code involved in them. Figure 10 shows the results. In both diagrams, the x axis gives the total number of lines of code in a module (the size of the module). In Figure 10a, the y axis is the number of occurrences of the anti-pattern in a module, while in Figure 10b, the y axis gives the percentage of lines of code involved in the anti-pattern.

It may be concluded from this analysis that, unless strongly discouraged by the technology, developers really fall into the trap of this anti-pattern – it simply is a real phenomenon, even not exceptionally rare in practice at all. For all considered technologies except ours, the anti-pattern occurred in more than one analyzed module, while in the object-oriented one (GWT), it was found in all six analyzed modules. When it occurred, it was found from one to 11 times in a module, taking up from a few to even 20% of lines of code (26% in one case). In the samples made in our technology, it was not found at all.



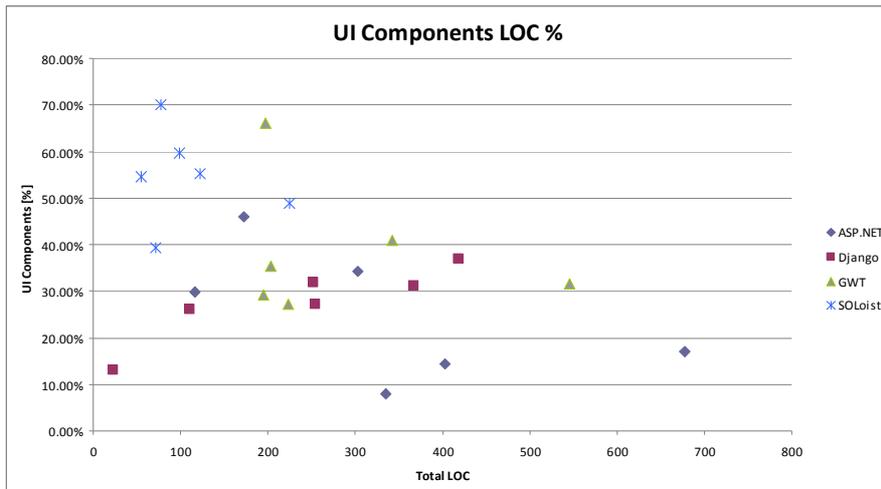
(a)



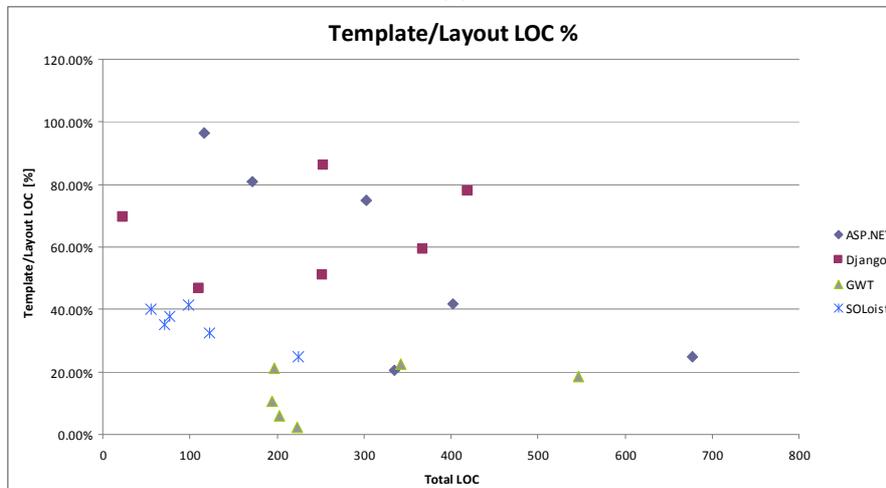
(b)

Figure 10: The presence of the “Magic Pushbutton” bad practice in analyzed applications. (a) Number of occurrences in a module. (b) Percentage of lines of code.

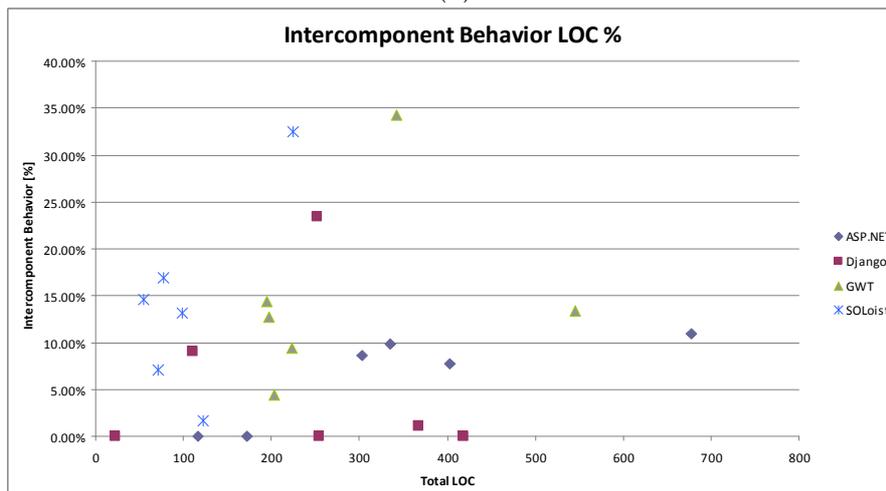
Figure 11 shows a few other interesting characteristics of the contents of the analyzed modules. The modules defined in our technology tend to have their biggest parts devoted to the definition of UI components (always above 40% of code, and above 50% in most cases), as opposed to other techniques that were below 40% in most cases. On the other hand, our technology requires much less code spent on defining other aspects: the layout (40% and less), inter-component behavior (typically 15% and less), and especially access to the data (less than 20%). In most cases, other technologies seem to require more code devoted to aspects other than defining pure UI components, especially to data access (at least some modules with more than 20% of code for each technology).



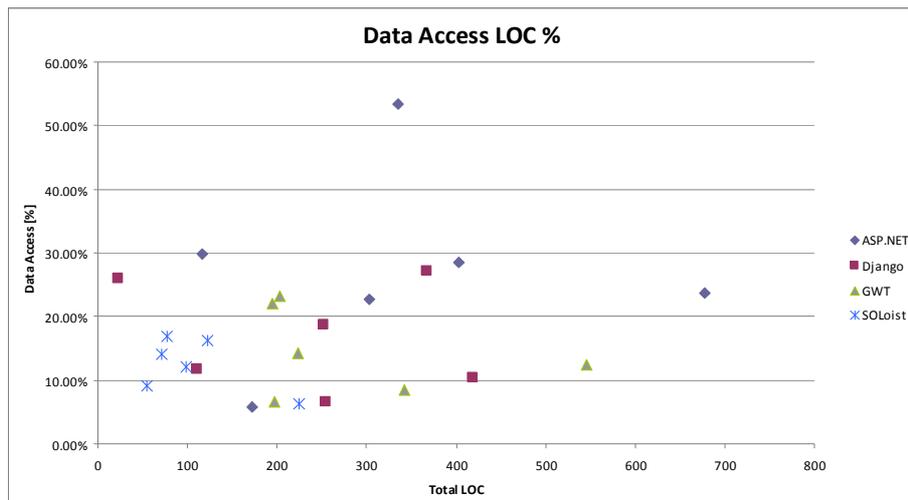
(a)



(b)



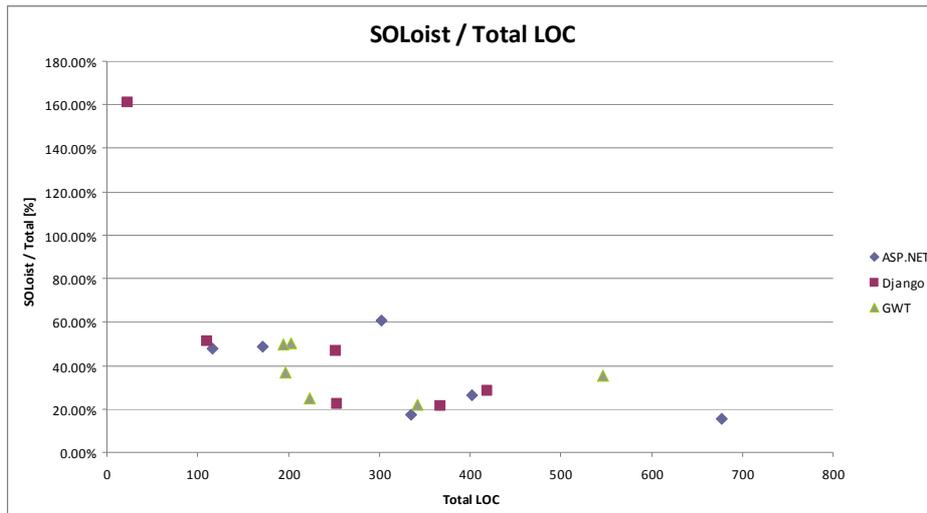
(c)



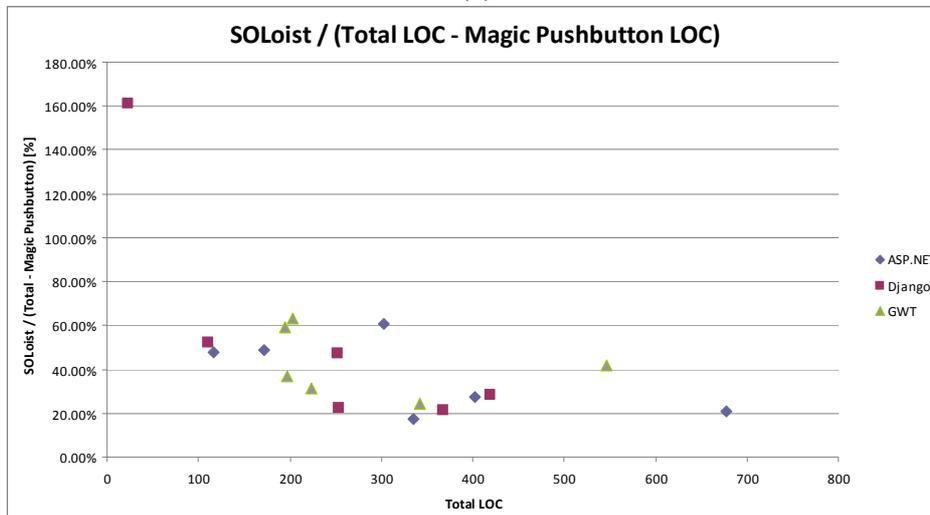
(d)

Figure 11: Comparison of different aspects of the analyzed applications. Percentage of lines of code spent on defining (a) UI components in a module, (b) layout, (c) inter-component behavior, and (d) data access.

Our last experiment was to create a functional and visual equivalent in our technology for each selected module made in another technology, and to compare the size of the code of the counterparts. While doing this, we assumed that we had the appropriate conceptual model and entire business logic implemented on the back end, while we focused on implementing the UI layer only. Figure 12 shows the ratio of the size of the equivalent counterpart made in our technology and the size of the original module (the percentage of the number of lines of our code out of the number of lines of code of the original module), depending on the size of the original module. It can be seen that except for one singular case of an exceptionally small module of 16 lines of code, the counterpart made in our technology was always between 20% and 60% of the size of the original module, with an apparent tendency of decreasing with the size of the module. As a conclusion, at least for the analyzed samples, our technique scaled better than the other technologies.



(a)



(b)

Figure 12: Comparison of the size (in lines of code) of each original module made in another technology with its functional and visual equivalent made in our technology: (a) when the original module contains the inappropriate code (“Magic Pushbutton”); (b) when only the appropriate code in the original module is counted in (with “Magic Pushbutton” code removed from the module).

Performance Evaluation

Goals and Assumptions

The goal of this campaign was to evaluate some aspects of our current implementation that affect performance and responsiveness of applications, and to compare them with other technologies. In particular, we wanted to measure the volume of communication between the frontend (the client) and the backend (the server) for our implementation and to compare it with other technologies.

To do this, we took the same set of sample applications and modules used in the previous campaign. For each of the modules (Web pages), we measured the total volume of communication between the client and the server, for a scenario (use case) that is characteristic for that page. The scenario started from loading the page, went on with some characteristic selections or data

modifications on the presented form, and ended with a characteristic command that has either refreshed the same page or loaded another page.

For each such scenario, we measured:

- a) the number of messages (requests-response pairs) interchanged between the client and the server,
- b) the amount of data (in bytes) transferred in both directions during the entire scenario; we have not counted in the data cached by the browser, as they were not transferred over the network.

For each module and its characteristic scenario, we have compared these parameters with the same parameters of the module's equivalent made in our technology.

Experiments and Reports

Figure 13 shows the summary of the results of our experiments.

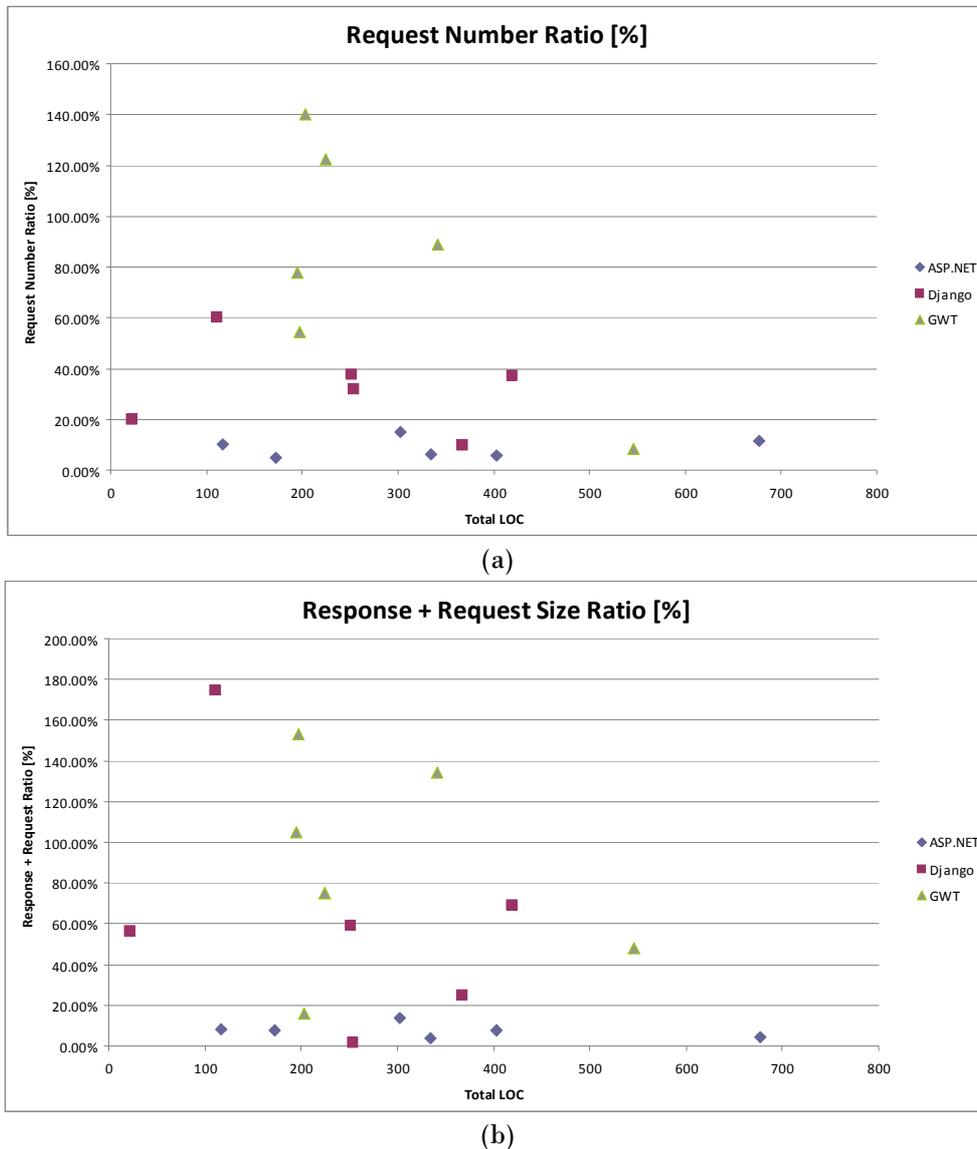


Figure 13: Comparison of the volume of communication between the client and the server of each original module made in another technology with its functional and visual equivalent made in our technology: (a) the number of request-response pairs; (b) the amount of data transferred over the network (without the data cached by the browser).

Figure 13a shows the ratio of the number of messages (request-response pairs) interchanged between the client and the server for the equivalent made in our technology and for the original module. It can be seen that in most cases, the number of messages is smaller in our technology (below 80%), while it is still acceptable (120% and 140%) for two cases of GWT modules. For ASP.NET, our equivalent generates more than 5 times fewer messages than the original for all modules.

Figure 13b depicts the ratio of the amount of data transferred between the client and the server. In most cases, our implementation still outperforms the other technologies (again more than 5 times for ASP.NET), while it is acceptably inferior to the originals in a few cases (between 100% and 180%). It should be noted that our implementation can further be improved in terms of compaction of the transferred data, because we still have some non-compact descriptors in messages.

We can conclude that, although it may look like that our approach introduces a flood of messages between the two tiers, experiments show that this is not the case: due to internal optimizations, the volume of communication is in most cases smaller than in other technologies, and when it is not, it is within acceptable boundaries. In some cases, it outperforms the counterparts several times.

7 CONCLUSIONS

We have presented a novel approach to modeling and implementing UIs of large-scale business applications. Our substantial experience in applying the method in large industrial projects makes us strongly believe that it has several outstanding features:

- The same modeling concept of capsule with internal structure can be re-applied recursively and coherently at successively lower levels of detail within a model, starting from high architectural modeling levels, down to lowest levels of modeling simple UI components.
- The proposed concepts are formally coupled, linguistically coherent, and thus do not suffer from scope, semantic, and phase discontinuities [38].
- The concept of capsule encourages proper abstraction of individual UI components as well as coherent UI fragments, with inherent strong encapsulation and support for reuse. This is one of the fundamental prerequisites for its scalability.
- Interaction between UI capsules is specified declaratively by simply wiring their pins. This feature significantly reduces development effort and accidental complexity of specifying functional coupling between UI components as in imperative code of event handlers.
- The method strongly impedes mixing business logic in UI code, e.g. in event handlers.
- Our capsule library provides easy coupling with object space due to proper conceptual matching.
- Developers are free to build flexible interfaces, either task- (wizard-) based, purely data-centric, or combined. Our paradigm does not prefer any interaction style, but supports all of them. In other words, tight conceptual coupling with the domain object space does not imply that the UI must always present objects as “naked,” although we agree that this kind of perspective may be appropriate in many cases. On the contrary, our paradigm is general and flexible enough to support development of both noun-verb (object-action) and verb-noun (function-oriented) UIs.

We are continuously working on improving our modeling tools, the runtime, and the capsule library, as well as on gaining new experience from applying the method in ongoing industrial projects.

REFERENCES

- [1] Agile Platform™, <http://www.outsystems.com/>
- [2] Baron, M., Girard, P., “SUIDT: A Task Model Based GUI-Builder,” *Proc. Task Models and Diagrams for User Interface Design (TAMODIA)*, July 2002, pp. 64-71
- [3] Bodart, F., Hennebert, A. M., Leheureux, J. M., Vanderdonckt, J., “A Model-Based Approach to Presentation: A

- Continuum from Task Analysis to Prototype,” *Proc. Eurographics Workshop on Design, Specification, Verification of Interactive Systems*, June 1994, pp. 77-94
- [4] Bodart, F., Hennebert, A. M., Leheureux, J. M., Sacré, I., Vanderdonck, J., “Architecture Elements for Highly-Interactive Business-Oriented Applications,” *Proc. EWHCI*, 1993, pp. 83-104
 - [5] Burns, E., Kitain, R., editors, *JavaServer Faces Specification*, Sun Microsystems, June 2009
 - [6] Da Cruz, A. M. R., Faria, J., P., “A Metamodel-Based Approach for Automatic User Interface Generation,” *Proc. 13th ACM/IEEE Int’l Conf. Model-Driven Engineering Languages and Systems (MODELS)*, Oct. 2010, pp. 256-270
 - [7] Da Silva, P. P., “User Interface Declarative Models and Development Environments: A Survey,” *Proc. Interactive Systems: Design, Specification, and Verification*, June 2000
 - [8] Da Silva, P. P., Paton, N. W., “User Interface Modeling in UMLi,” *IEEE Software*, Vol. 20, No. 4, Jul./Aug. 2003, pp. 62-69
 - [9] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley Longman, 1995
 - [10] Goderis, S., *On the Separation of User Interface Concerns*, PhD Thesis, Vrije Universiteit Brussels, 2008
 - [11] Google Web Toolkit, <http://code.google.com/webtoolkit/>
 - [12] Groenewegen, D., Hemel, Z., Visser, E., “Separation of Concerns and Linguistic Integration in WebDSL,” *IEEE Software*, Vol. 27, No. 5, Sept./Oct. 2010, pp. 31-37
 - [13] Intelliun Virtual Enterprise, www.intelliun.com
 - [14] JavaServer Faces, <http://www.java-serverfaces.org>
 - [15] Jia, X., Steele, A., Qin, L., Liu, H., Jones, C., “Executable Visual Software Modeling – the ZOOM Approach,” *Software Quality Journal*, Vol. 15, No. 1, 2007, pp. 27-51
 - [16] Johnson, P., Wilson, S., Markopoulos, P., Pycocock, J., “ADEPT – Advanced Design Environment for Prototyping with Task Models”, *Proc. INTERCHI '93 Conference on Human Factors in Computing Systems*, 1993
 - [17] Kovacevic, S., “UML and User Interface Modeling”, *Proc. 1st Int’l Workshop on The Unified Modeling Language (UML '98)*, 1998, pp. 253-266
 - [18] Landay, J. A., Myers, B. A., “Sketching Interfaces: Toward More Human Interface Design,” *IEEE Computer*, Vol. 34, No. 3, March 2001, pp. 56-64
 - [19] Li, N., Li, H., Wu, J., Zhong, X., Sun, Z., Bao, W., “AUTOSAR Based Automatic GUI Generation,” *Proc. 13th IEEE Int’l Symp. Object/Component/Service-Oriented Real-Time Distributed Computing*, May 2010, pp. 156-162
 - [20] Milićev, D., “Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments,” *IEEE Transactions on Software Engineering*, Vol. 28, No. 4, April 2002, pp. 413-431
 - [21] Milićev, D., *Model-Driven Development with Executable UML*, John Wiley & Sons (WROX), 2009
 - [22] Milićev, D., “Towards Understanding of Classes versus Data Types in Conceptual Modeling and UML,” submitted for publication
 - [23] Milićev, D., “The OOIS UML Profile for UI Capsule-Based Modeling,” internal technical document, available at <http://rcub.bg.ac.rs/~dmilicev/publishing/ooisumlui.pdf>
 - [24] Molina, P. J., Meliá, S., Pastor, O., “Just-UI: A User Interface Specification Model”, *Proc. CADUI 2002*, pp. 63-74
 - [25] Monteiro, M., Oliveira, P., Gonçalves, R., “GUI Generation Based on Language Extensions: A Model to Generate GUI, based on Source Code with Custom Attributes,” *Proc. ICEIS*, June 2008
 - [26] Myers, B. A., Hudson, S. E., Pausch, R., “Past, Present and Future of User Interface Software Tools,” *ACM Transactions on Computer-Human Interaction*, Vol. 7, No. 1, March 2000
 - [27] Nichols, J., Faulring, A., “Automatic Interface Generation and Future User Interface Tools,” *ACM SIGCHI Workshop on The Future of User Interface Design Tools*, 2005
 - [28] Nunes, N. J., E Cunha, J. F., “Wisdom - A UML Based Architecture for Interactive Systems,” *Proc. DSV-IS*, 2000, pp. 191-205
 - [29] Object Management Group, *UML 2.2 Superstructure Specification*, <http://www.omg.org>, February 2009

- [30] Olivé, A., *Conceptual Modeling of Information Systems*, Springer, 2007
- [31] Patternò, F., Sabbatino, V., Santoro, C., "Using Information in Task Models to Support Design of Interactive Safety-Critical Applications," *Proc. ACM Working Conf. Advanced Visual Interfaces (ACM AVI '00)*, May 2000
- [32] Pawson, R., "Naked Objects", Ph.D Thesis, Trinity College, Dublin, Ireland, 2004
- [33] Perišić, B., Milosavljević, G., Dejanović, I., Milosavljević, B., "UML Profile for Specifying User Interfaces of Business Applications," *Computer Science and Information Systems*, Vol. 8, No. 2, May 2011, pp. 405-426
- [34] Pitkänen, A., Hickey, S., "Implementation of Model-based User Interfaces for Pictorial Communication Systems Using UsiXML and Flex", *Proc. 1st International Workshop USeR Interface eXtensible Markup Language (UsiXML)*, 2010, pp. 65-74
- [35] Ruby on Rails, <http://rubyonrails.org/>
- [36] Nokia Qt, <http://qt.nokia.com/>
- [37] Selic, B., "The Pragmatics of Model-Driven Development," *IEEE Software*, Vol. 20, No. 5, Sept./Oct. 2003, pp. 19-25
- [38] Selic, B., Gullekson, G., Ward, P. T., *Real-Time Object-Oriented Modeling*, John Wiley and Sons, 1994
- [39] Trujillo, S., Batory, D., Diaz, O., "Feature Oriented Model Driven Development: A Case Study for Portlets," *Proc. 29th Int'l Conf Software Engineering (ICSE '07)*, 2007, pp. 44-53
- [40] Van Welie, M., Van der Veer, G. C., Eliëns, A., "An Ontology for Task World Models," *Proc. Design, Verification and Specification of Interactive Systems Workshop*, 1998, pp. 57-70
- [41] Viswanathan, V., "Rapid Web Applications Development: Ruby on Rails Tutorial," *IEEE Software*, Vol. 25, No. 6, Nov./Dec. 2008, pp. 98-106
- [42] WebRatio, www.webratio.com
- [43] Wilson, S., Johnson, P., "Empowering Users in a Task-Based Approach to Design," *Proc. Designing Interactive Systems: Processes, Practices, and Techniques*, ACM Press, 1995, pp. 25-31
- [44] Wilson, S., Johnson, P., Kelly, C., Cunningham, J., Markopoulos, P., "Beyond Hacking: a Model Based Approach to User Interface Design", *Proc. HCI*, 1993, p. 217
- [45] Wong, J., Hong, J.I., "Making mashups with marmite: towards end-user programming for the web," *Proc. SIGCHI Conf on Human factors in Computing Systems (CHI '07)*, 2007, pp. 1435-1444
- [46] Woods, E., Emery, D., Selic, B., "Point/Counterpoint," *IEEE Software*, Vol. 27, No. 6, Nov./Dec. 2010, pp. 54-57
- [47] Xie, Q., Grechanik, M., Fu, C., "Guide: A GUI Differentiator," *Int'l Conf on Software Maintenance (ICSM'09)*, 2009, pp. 395-396
- [48] Zhang, G., Hölzl, M., "Aspect-Oriented Modeling of Web Applications with HiLA", *7th Int'l Conf on Advances in Mobile Computing and Multimedia*, 2009, pp. 1-12
- [49] Zhang, G., "Aspect-Oriented UI Modeling with State Machines", *Proc. 5th Int'l Workshop on Model-Driven Development of Advanced User Interfaces (MDDAUI'10)*, 2010, pp. 45-48

8 APPENDIX 1: THE OOIS UML PROFILE FOR UI CAPSULE-BASED MODELING

Introduction

This document contains a specification of the OOS UML profile for UI modeling based on capsules, wires, and pins. The specification follows the OMG's UML Superstructure Specification 2.2 [1]. The motivation and applicability of the concepts and features of the profile are explained elsewhere [2, 3].

Overview

This specification describes the contents of the OOIS IML UI profile that references the UML2 metamodel (Figure 1).

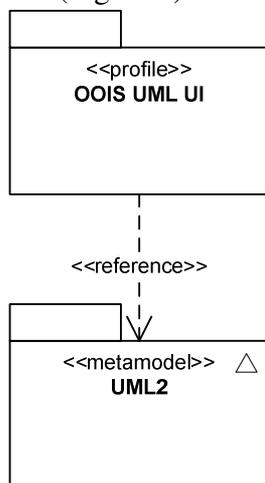


Figure 1

Capsule, Internal Structure, and Constructor

Capsule is an optional stereotype that can be applied to a Class (from `CompositeStructures::StructuredClasses`), as shown in Figure 2. A capsule has an optional presentation style (the `style` attribute), which is an implementation-specific style that defines the elements for formatting and appearance, such as colors, fonts, borders, padding, margins, etc. A capsule has an optional UI configuration context (the `guiContext` attribute), which is a GUI context as defined in OOIS UML [4]. If not specified in a capsule, each instance of the capsule inherits any of these from its owner capsule instance.



Figure 2

The internal structure of a capsule shown in a OOIS UML UI diagram, including parts (capsule references), ports, wires, conditional creation constraints, and the `parameters` compartment, is just a notational shorthand (i.e., a profiled diagrammatic representation) for the definition of a constructor operation with the given creational specification as in OOIS UML [4]. For example, the diagram in Figure 3 is a notational shorthand for the UML specification in Figure 4. Instance specifications, including links (wires), have the semantics of creational specifications in OOIS UML [4].

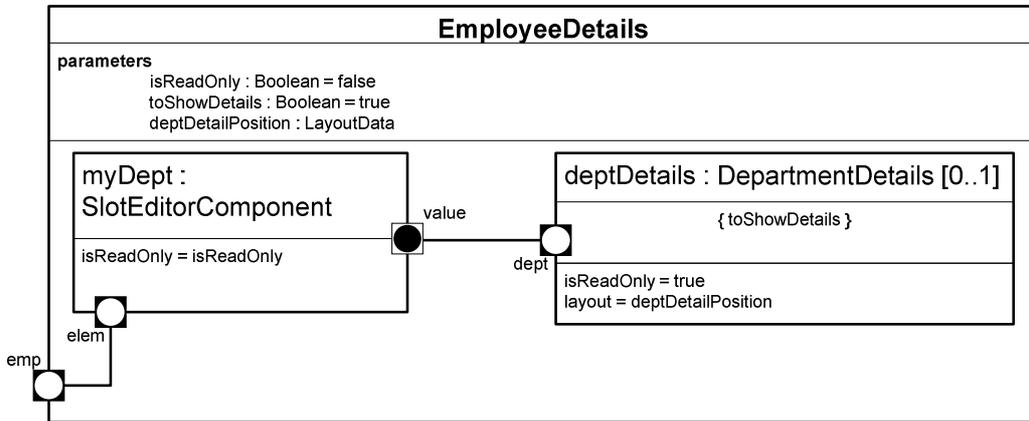


Figure 3

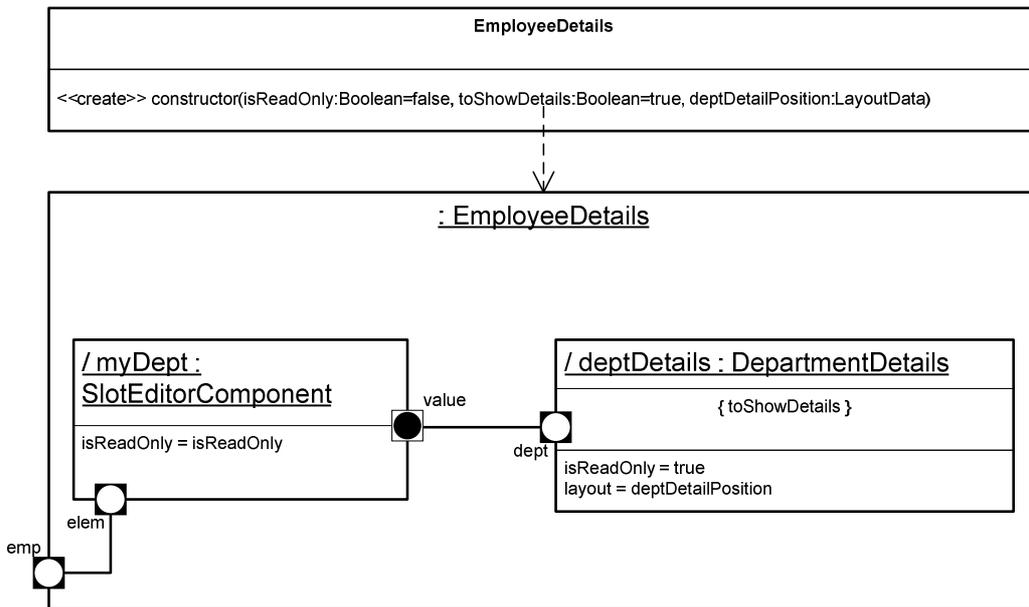


Figure 4

Additional Constraints

1. All owned attributes of a capsule, except ports, must be either protected or private:
context Capsule:

```

self.base_Class.ownedAttribute->
forAll(m|          not          m.oclIsKindOf(Port)          implies
(m.visibility=protected or m.visibility=private))

```

2. All owned operations of a capsule, except constructors, must be either protected or private:

```

context Capsule:
self.base_Class.ownedOperation->
forAll(m|          m.extension$_create->size()=0          implies
(m.visibility=protected or m.visibility=private))

```

Pins

Pin is a mandatory stereotype applied to UML ports (Figure 5).

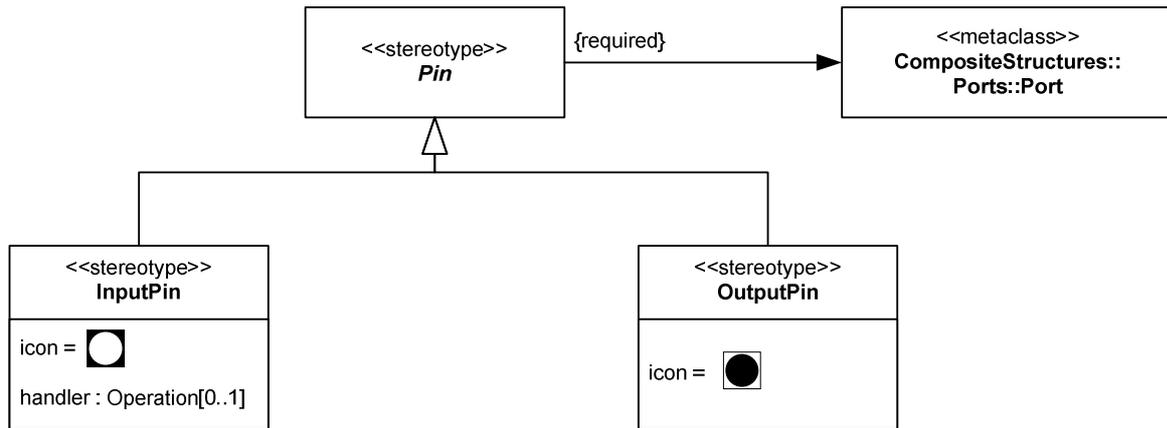


Figure 5

An input pin (**<<inputPin>>**), being a UML property, is typed with the parameterized (template) class `InputPin`, and an output pin (**<<outputPin>>**) is typed with the parameterized class `OutputPin` shown in Figure 6; these two classes are library classes of the profile. The template parameters define the type (T), multiplicity, ordering, and uniqueness of the (collections of) objects that can be transported over the pin.

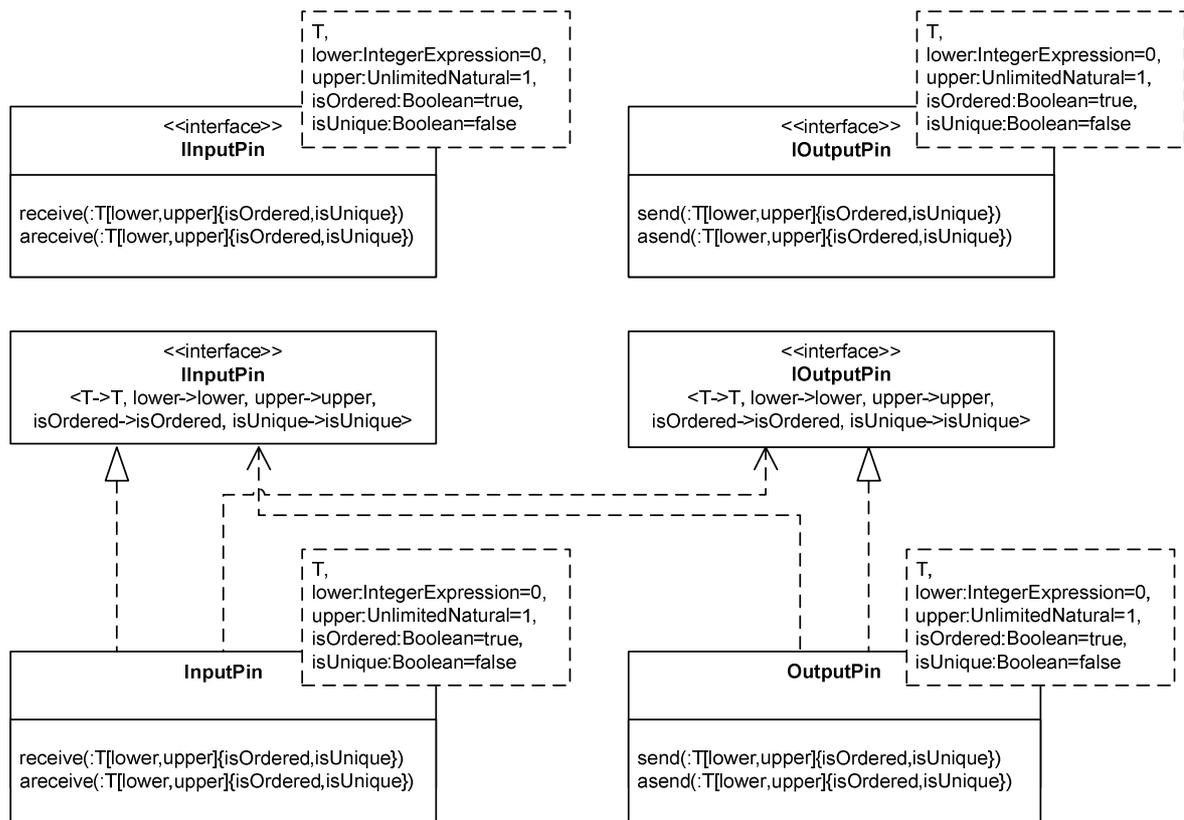


Figure 6

The class `OutputPin` realizes the parameterized interface `IOutputPin` (Figure 6). This is the (only) interface provided by an output pin. It requires the interface `IInputPin` provided by input pins. The operation `send` is available for sending (collections of) objects of type `T` over that output pin synchronously. The operation `asend` is available for sending (collections of) objects of type `T` over that output pin asynchronously.

Conversely, the class `InputPin` realizes the parameterized interface `IInputPin` (Figure 6). This is the (only) interface provided by an input pin. It requires the interface `IOutputPin` provided by output pins. The operation `receive` is called internally when (collections of) objects of type `T` are provided on that input pin synchronously. The operation `areceive` is called internally when (collections of) objects of type `T` are provided on that input pin asynchronously.

Interface pin is a synonym for a service port of a class. *Internal pin* is a synonym for a non-service port of a class.

The attribute `handler` of the stereotype `InputPin` is an optional specification of the operation of the class that owns the pin and that is called as the handler of the incoming message arriving on the pin.

A *behavior's pin* is a notational shorthand for an internal behavior pin. The notation in Figure 7a is a shorthand for the model in Figure 7b.

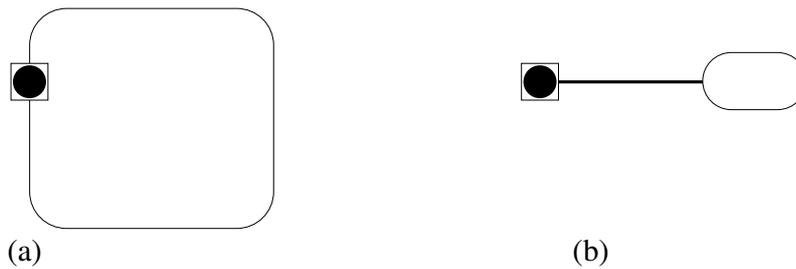


Figure 7

Additional Constraints

3. A pin must be a property of either a capsule or of a service provider (to be defined later), but not both:

```
context Pin:
  self.base_Port.class.extension$_capsule->size()=1 xor
  self.base_Port.class.extension$_serviceProvider->size()=1
```

4. An interface pin must have unspecified, public, or package visibility:

```
context Pin:
  self.base_Port.isService implies
  (self.base_Port.visibility.size()=0 or
  self.base_Port.visibility=public or
  self.base_Port.visibility=package)
```

5. An internal pin must have unspecified, private, or protected visibility:

```
context Pin:
  not self.base_Port.isService implies
  (self.base_Port.visibility.size()=0 or
  self.base_Port.visibility=private or
  self.base_Port.visibility=protected)
```

6. The type of an input pin must be InputPin:

```
context <<stereotype>>InputPin:
  self.base_Port.type=InputPin
```

7. The type of an output pin must be OutputPin:

```
context <<stereotype>>OutputPin:
  self.base_Port.type=OutputPin
```

8. Only a behavior pin can have a handler operation:

```
context Pin:
  self.handler->size()>0 implies
  self.base_Port.isBehavior
```

9. The handler operation of a pin must be a feature of the same class that owns the pin:

```
context Pin:
  self.handler->size()>0 implies
  self.base_Port.class->allFeatures()->includes(self.handler)
```

Additional Operations

1. `Pin::conformsTo(other:Pin):Boolean`: Does this pin conform to the other given pin? The conformance is defined according to the type of the pin, that is, to the type T , multiplicity, ordering, and uniqueness of the parameterized class typing the port (`self.type`), as defined in OOIS UML [4].

SAP and SPP

Input SAP is a kind of input pin that is always internal (Figure 8). *Output SAP* is a kind of output pin that is always internal. An input or output SAP has a specified SPP (the attribute `spp`) it is bound to. The SPP has to be compatible with the SAP.

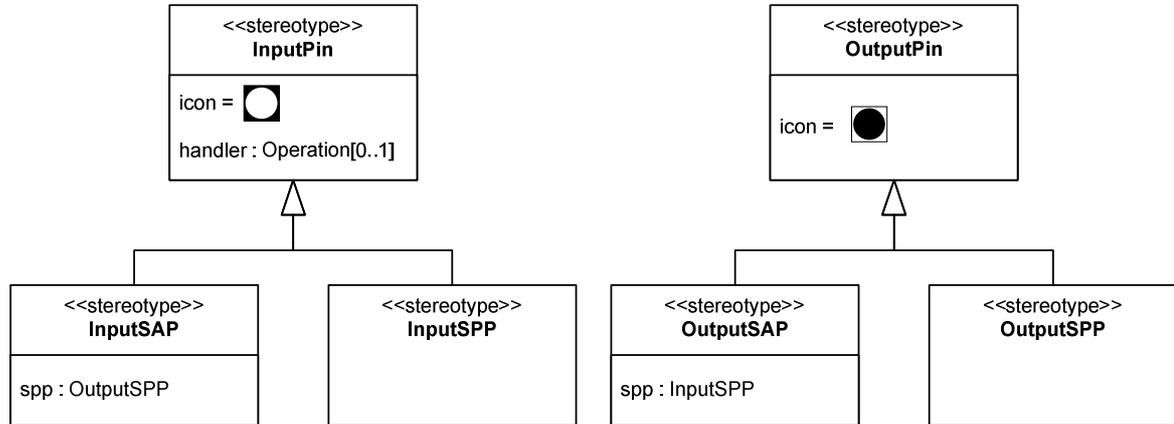


Figure 8

The type of an input SAP is the class `InputSAP` (Figure 9). The type of an output SAP is the class `OutputSAP` (Figure 9). These two classes are library classes from this profile.

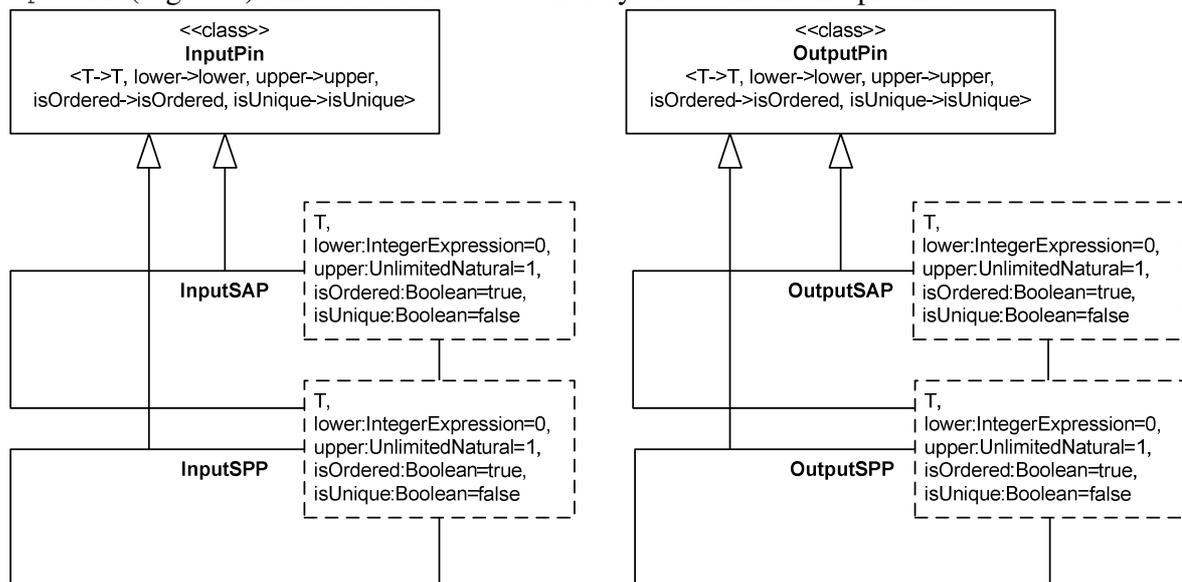


Figure 9

Service Provider is a stereotype that can be applied to a class (from `CompositeStructures::StructuredClasses`, Figure 10). A class that is a service provider provides access to its instance that serves requests from SAPs through its operation specified in `ServiceProvider::instanceOp`: for each incoming message sent via an SAP to an SPP that belongs to the service provider, the instance of the provider is obtained by calling the instance operation, and then the message is passed to the SAP of that instance.

Input SPP is a kind of an input port (Figure 8). The type of an input SPP is the class `InputSPP`, analogous to `InputSAP`. *Output SPP* is a kind of an output port (Figure 8). The type of an output SPP

is the class `OutputSPP`, analogous to `OutputSAP`. An SPP serves as an interface port of its owner service provider. It serves the requests sent from SAPs.

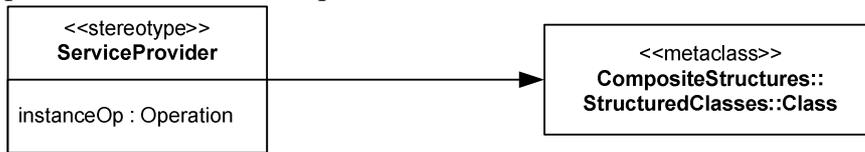


Figure 10

Additional Constraints

1. The owner of an input or output SAP must be a capsule:


```

context InputSAP:
    self.base_Port.class.extension$_capsule->size()=1
context OutputSAP:
    self.base_Port.class.extension$_capsule->size()=1
      
```
2. An input or output SAP is always internal:


```

context InputSAP:
    not self.base_Port.isService
context OutputSAP:
    not self.base_Port.isService
      
```
3. The type of an input SAP must be `InputSAP`:


```

context <<stereotype>>InputSAP:
    self.base_Port.type=InputSAP
      
```
4. The type of an output SAP must be `OutputSAP`:


```

context <<stereotype>>OutputSAP:
    self.base_Port.type=OutputSAP
      
```
5. The bound SPP and SAP have to conform one to the other:


```

context InputSAP:
    self.spp.conformsTo(self)
context OutputSAP:
    self.conformsTo(self.spp)
      
```
6. The instance operation of a service provider has to be a feature of the service provider's class, has to be static, and has to return an instance of that very class:


```

context InputSPP, OutputSPP:
    self.base_Port.class->allFeatures()->includes(self.instanceOp)
and self.instanceOp.type=self.base_Port.class and
self.instanceOp.isStatic
      
```
7. The owner of an SPP must be a service provider:


```

context InputSPP:
    self.base_Port.class.extension$_serviceProvider->size()=1
context OutputSPP:
    self.base_Port.class.extension$_serviceProvider->size()=1
      
```
8. An input or output SPP is always an interface port:


```

context InputSPP:
    self.base_Port.isService
context OutputSPP:
    self.base_Port.isService
      
```
9. The type of an input SPP must be `InputSPP`:


```

context <<stereotype>>InputSPP:
    self.base_Port.type=InputSPP
      
```
10. The type of an output SPP must be `OutputSPP`:


```

context <<stereotype>>OutputSPP:
    self.base_Port.type=OutputSPP
      
```

Wires

Wire is a required stereotype of UML connector (from `CompositeStructures::InternalStructures`, Figure 11).

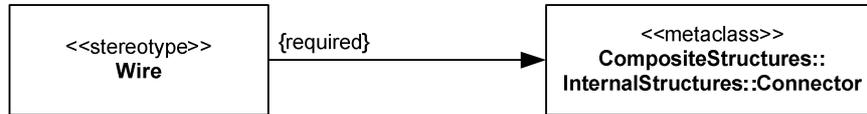


Figure 11

A wire can connect pins according to the basic UML rules, but also to additional rules defined in Table 1. This table specifies whether a pin *X* of a certain kind (input or output) can be connected with a wire to a pin *Y* of a certain kind. The label “Behavior” denotes an internal behavior pin, while the label “Part’s” denotes a pin of the owner capsule’s part (internal component). The label “N/A” means that the given connection is not allowed. The symbol “ $X \rightarrow Y$ ” means that the connection is allowed and that the messages flow through that wire from *X* to *Y*; in that case, *X* is called the *source*, and *Y* the *destination* pin. It can be concluded from this table that interface pins and SAPs behave the same (like pins for external communication), while internal behavior pins and part’s pins behave the same (like pins for internal communication) in terms of connectivity and flow direction.

Pin X		Pin Y							
		Input				Output			
		Interface	SAP	Behavior	Part’s	Interface	SAP	Behavior	Part’s
Input	Interface	N/A	N/A	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	N/A	N/A
	SAP	N/A	N/A	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	N/A	N/A
	Behavior	$Y \rightarrow X$	$Y \rightarrow X$	N/A	N/A	N/A	N/A	$Y \rightarrow X$	$Y \rightarrow X$
	Part’s	$Y \rightarrow X$	$Y \rightarrow X$	N/A	N/A	N/A	N/A	$Y \rightarrow X$	$Y \rightarrow X$
Output	Interface	$Y \rightarrow X$	$Y \rightarrow X$	N/A	N/A	N/A	N/A	$Y \rightarrow X$	$Y \rightarrow X$
	SAP	$Y \rightarrow X$	$Y \rightarrow X$	N/A	N/A	N/A	N/A	$Y \rightarrow X$	$Y \rightarrow X$
	Behavior	N/A	N/A	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	N/A	N/A
	Part’s	N/A	N/A	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$	N/A	N/A

Table 1

Additional Operations

1. `Wire::sourcePin():Pin[0..1], Wire::destPin():Pin[0..1]`: Return the source and the destination pin of the wire according to the definitions given in Table 1 above.

Additional Constraints

1. A wire must have exactly two ends:

```
context Wire:
    self.base_Connector.end->size()=2
```

2. A wire can connect pins only:

```
context Wire:
    self.base_Connector.end->
        forEach(ce| ce.role.oclIsKindOf(Port) and
            ce.role.oclAsType(Port).extension$_pin->size()=1)
```

3. A wire can connect pins only according to the definitions in Table 1:

```
context Wire:
```

```
self.sourcePin()->size()==1 and self.destPin()->size()==1
```

4. A wire can connect compatible pins only:

```
context Wire:  
    self.sourcePin().conformsTo(self.destPin())
```

References

1. Object Management Group (OMG), *OMG Unified Modeling Language (OMG UML), Superstructure*, version 2.2, OMG Document Number: formal/2009-02-02, Standard document URL: <http://www.omg.org/spec/UML/2.2/Superstructure>, February 2009
2. Milicev, D., Mijailovic, Z., „Effective Modeling of User Interfaces for Large-Scale Applications,“ submitted for publication.
3. SOLoist framework, www.sol.rs
4. Milicev, D., *Model-Driven Development with Executable UML*, John Wiley & Sons, 2009

9 APPENDIX 2: ANALYTICAL COMPARISON WITH THIRD-PARTY TOOLS

In this section, we present comparison of the SOLoist framework with major UI technologies. In our approach, we try to evaluate different aspects by classifying source code fragments based on the type of “work” (concern) they were intended to do. Although this analysis tends to be subjective, we have developed set of strict rules which help in the process of code classification.

In this campaign, we tried to compare some development-related technical aspects of our framework with a set of selected mainstream UI development frameworks by assessing applications developed in these technologies. In particular, we were interested in assessing:

- a) quality of code; in particular, to what extent the technology prevents some bad coding practices;
- b) expressiveness of the programming technique, in terms of the size of code necessary to perform a certain task.

UI Framework Technologies and Selected Applications

We selected three very popular technologies:

- 1) Django/Python, representing the template-based approaches,
- 2) GWT/Java, representing the object-oriented approaches,
- 3) ASP.Net/C#, representing the hybrid approaches.

We chose six third-party applications for the analysis, two developed in each of the three technologies. Additionally, two projects developed with SOLoist are analyzed. For each selected application we selected three UI modules. Figures 1 to 24 depict selected modules (fragments) using red rectangular frame. The first selection criterion was that the chosen application was open-source. Obvious reason is availability of the source code for analysis. Additionally, open-source projects tend to be developed by broader community of developers. In our opinion, this leads to better and more revised code, so every bad trait would imply that problems occurred due to weaknesses or limitations of the technology rather than inexperience or incompetence of the developers. The second criterion was that problem domain was not overly complex. The one who analyzes the code should easily grasp it. It is not necessary to understand completely problem domain, just to be able to understand a user interface code. Business logic code is not of the interest here. Final criterion is that the chosen open-source project is neither too large nor too small. In addition, we are avoiding niche projects and thus, the results of this analysis should hold for a wider range of applications.

Bellow is the list of our test applications:

- C#/ASP.NET
 - dashCommerce, an e-commerce application [1] (Figures 1, 2, and 3)
 - nopCommerce, an e-commerce application [2] (Figures 4, 5, and 6)
- Python/Django
 - Satchmo, online store framework [3] (Figures 7, 8, and 9)
 - LFS, online shop and e-commerce solution [4] (Figures 10, 11, and 12)
- Java/GWT
 - LogicalDoc, document management system [5] (Figures 13, 14, and 15)
 - OpenKM, document management system [6] (Figures 16, 17, and 18)

Additionally, two projects developed with SOLoist were included into the analysis:

- CRMS - real estate agency customer relationship system (Figures 19, 20, and 21)
- ICMS - emergency call center management system (Figures 22, 23, and 24)

Apart from the abovementioned projects, we also re-built every analyzed UI fragment from third-party technologies using the SOLoist framework. These SOLoist artifacts were analyzed in the same way and compared with others.

dashcommerce  Welcome Back srdjan.lukovic@sol.rs |  My Cart (1) | [My Account](#) | [Admin](#) | [Log Out](#) |

Home

Test	Billing Information	Billing Address
	Shipping Address	
Favorite Products Banana	Cancel Add New	Mikan Palidrvce
Favorite Categories Test	First Name: <input type="text"/>	Testera Avenue
VISA MasterCard DISCOVER PayPal AMEX Master JCB	Last Name: <input type="text"/>	Belgradensaft, Sorabi 7788
	Phone: <input type="text"/>	SN
	Email: <input type="text"/>	5454101
	Address: <input type="text"/>	miki@gmail.com
	City: <input type="text"/>	Shipping Method
	Postal Code: <input type="text"/>	Fedbox: \$ 241.00
	State / Region: <input type="text"/>	Payment Information
	Country: <input type="text" value="-- Select --"/>	
	<input type="checkbox"/> Use for Shipping?	
	<input type="button" value="Continue"/>	
	Shipping Information	
	Shipping Method	
	Payment Information	
	Order Review	

Powered by dashCommerce

Figure 1: Address submission fragment (dashCommerce).

Home

Test

Favorite Products

Banana

Favorite Categories

Test

Billing Information					
Shipping Information					
Shipping Method					
Payment Information					
Order Review					
Order Summary					
Sku	Item	Quantity	Price	Extended Amount	
111	Banana	5	\$ 2.00	\$ 10.00	
				Subtotal:	\$ 10.00
				Coupon:	\$ 0.00
				Tax:	\$ 0.00
				Shipping & Handling:	\$ 0.00
				Total:	\$ 10.00
<input type="button" value="Process Order"/>					

Billing Address

Mikan Palidrvce
 Testera Avenue
 Belgradensaft, Sorabi 7788
 SN
 5454101
 miki@gmail.com

Shipping Address

Mikan Palidrvce
 Testera Avenue
 Belgradensaft, Sorabi 7788
 SN
 5454101
 miki@gmail.com

Shipping Method

Payment Information

Powered by dashCommerce

Figure 2: Checkout fragment (dashCommerce).

dashcommerce

Welcome Back srdjan.lukovic@sol.rs ! | My Cart (0) | My Account | Admin | Log Out |

Home

Test

Favorite Products

Banana

Favorite Categories

Test

Test »

Banana | Edit

Retail Price: \$ 2.80
Our Price: \$ 2.00

You Save: \$ 0.80 (28.57 %)

Average Rating:

Fruit

Reviews

There are no reviews for this product.

Add To Cart

Review

[Review It!](#)

Powered by dashCommerce

Figure 3: Product fragment (dashCommerce).

49

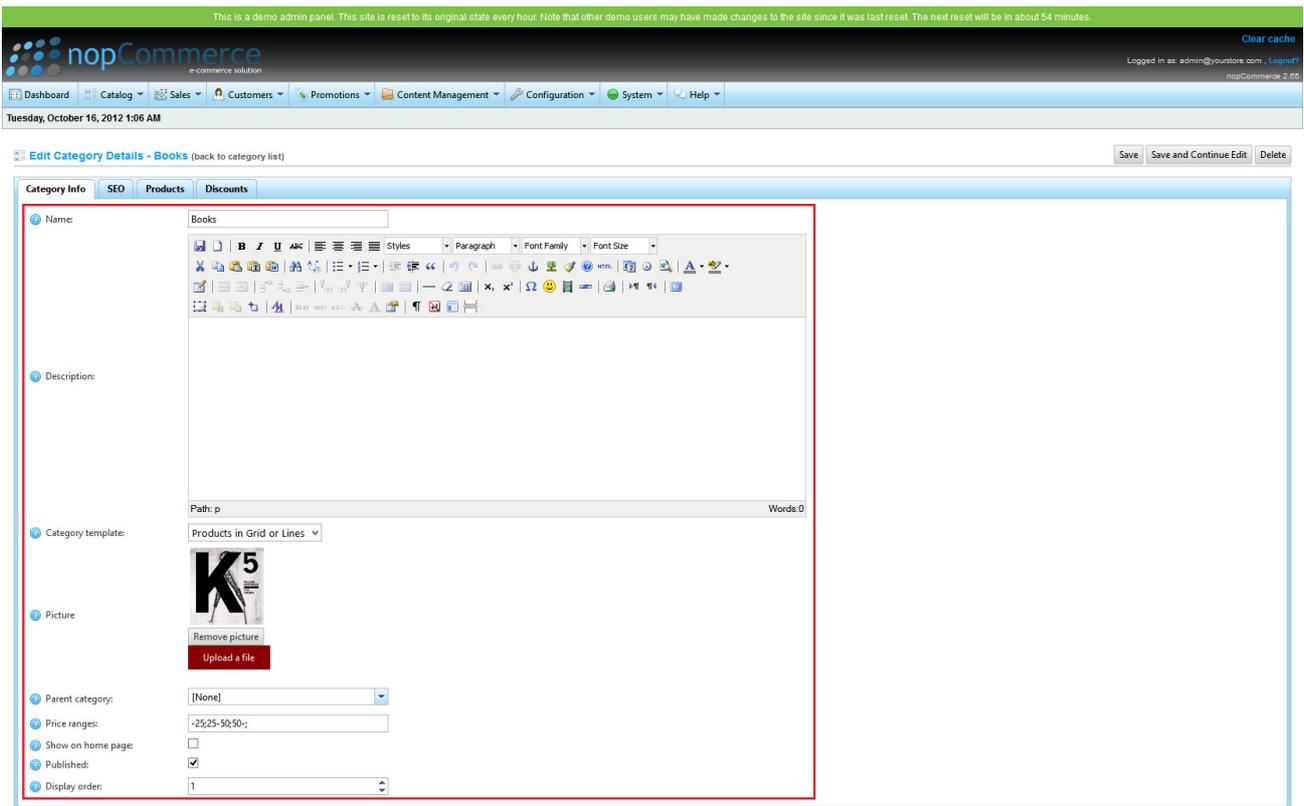


Figure 4: Category details fragment (nopCommerce).

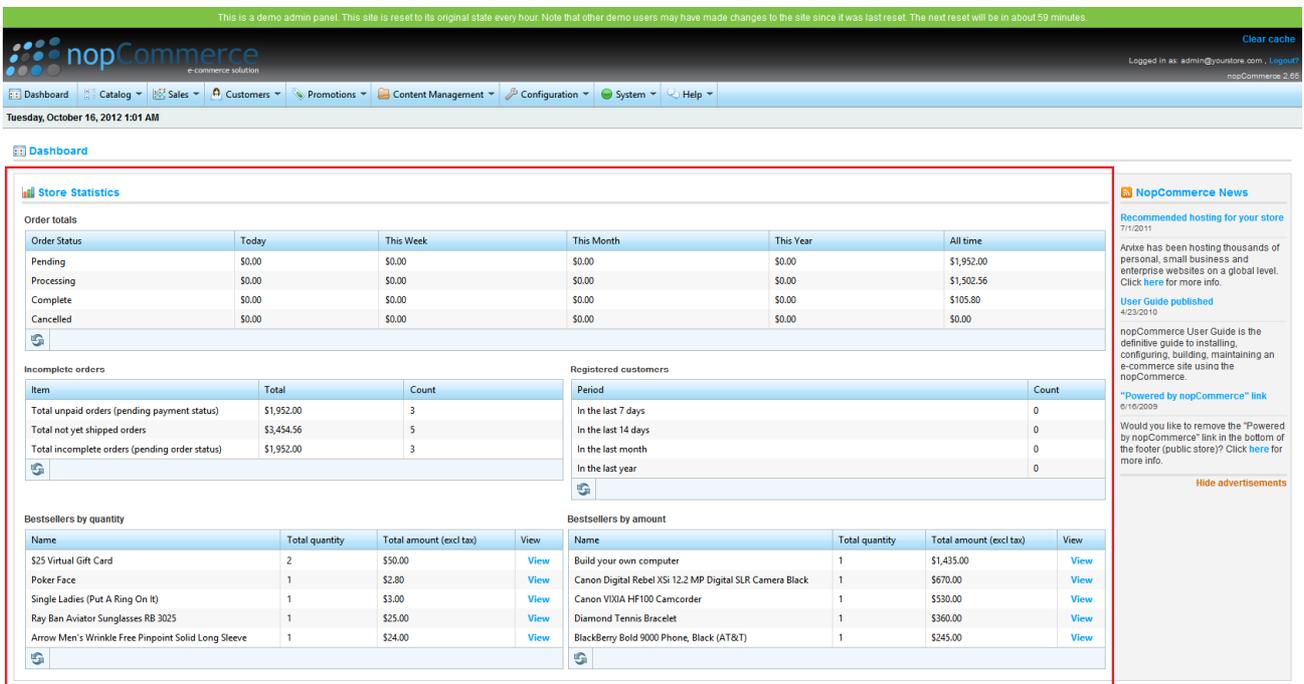


Figure 5: Dashboard fragment (nopCommerce).

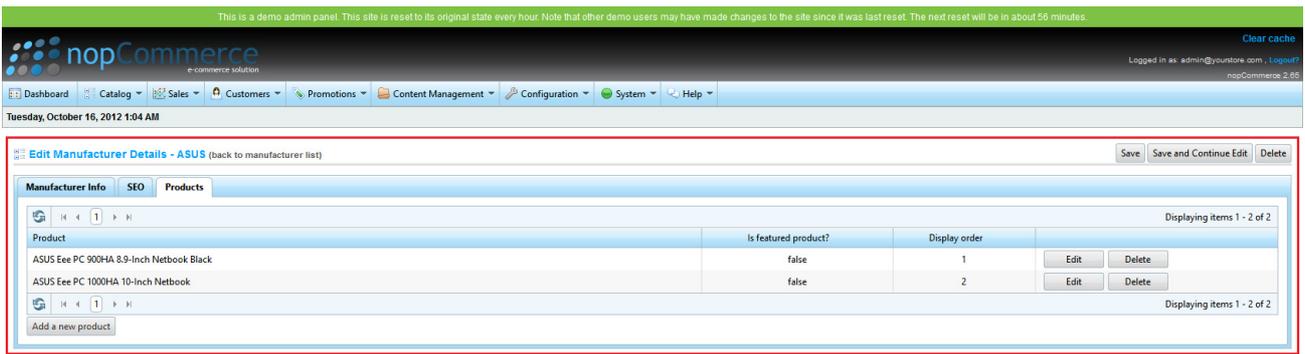


Figure 6: Manufacturer details fragment (nopCommerce).

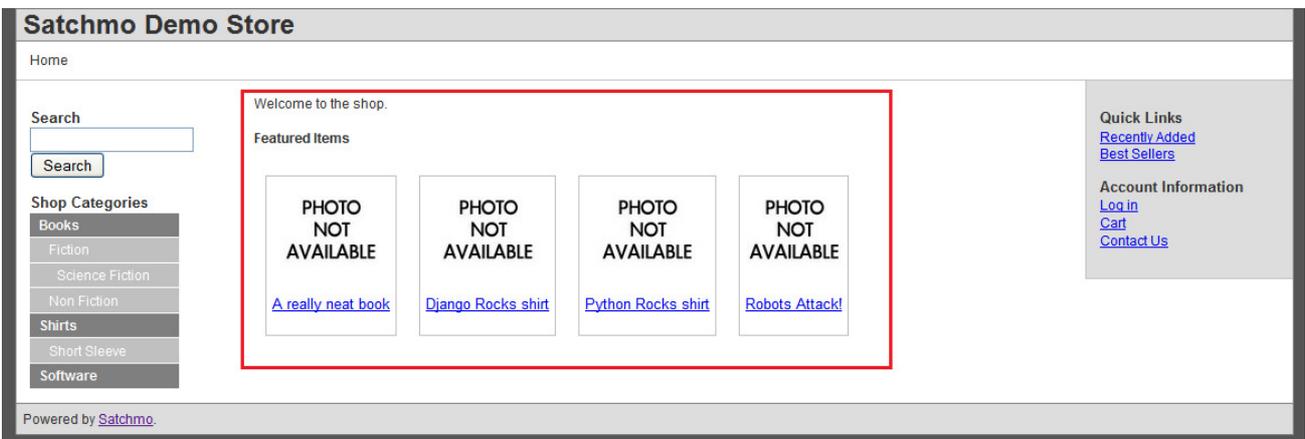


Figure 7: Bestsellers fragment (Satchmo).

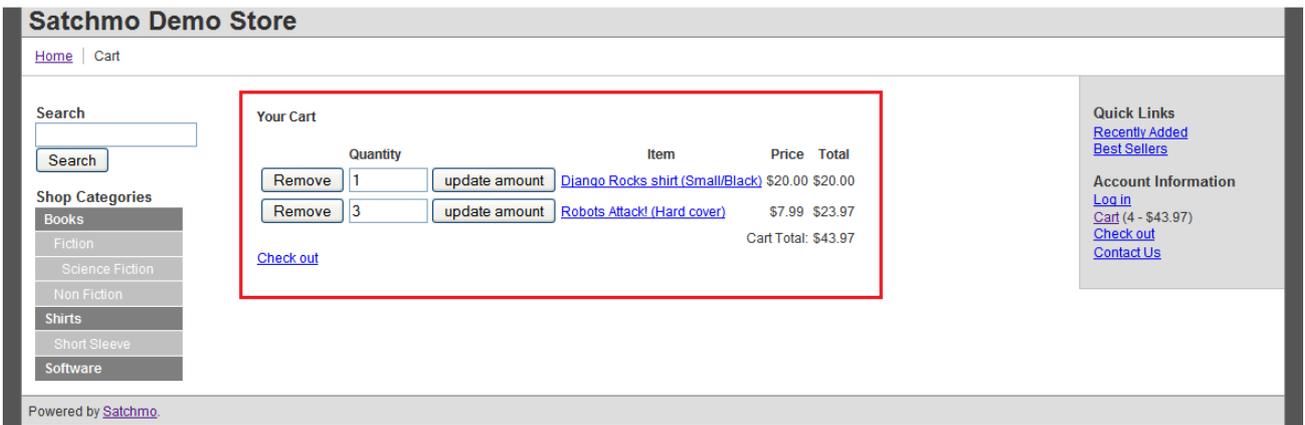


Figure 8: Shopping cart fragment (Satchmo).

Satchmo Demo Store

[Home](#) | [Checkout](#)

Search

Shop Categories

- Books
- Fiction
- Science Fiction
- Non Fiction
- Shirts
- Short Sleeve
- Software

Checkout

Ship/Bill » Payment » Confirmation

Please complete the following information in order to submit your order.

If you already have an account, you can login now to fill in most of the information below. This is just a convenience, no account is required!

Name:

Password:

You'll need to fill out at least the fields with a *

Basic Information

Email address*

First name*

Last name*

Phone number*

Billing Information

Bill To

If different from the first and last names above

Street*

Street

City*

State

Zipcode/Postcode*

Shipping Information

Shipping same as billing?

Ship To

If different from the first and last names above

Street

Street

City

State

Zipcode/Postcode

Discounts

Discount code

Quick Links

[Recently Added](#)

[Best Sellers](#)

Account Information

[Log In](#)

[Cart \(4 - \\$43.97\)](#)

[Check out](#)

[Contact Us](#)

Powered by [Satchmo](#).

Figure 9: Contact information fragment (Satchmo).

LFS admin | Logout

Shop ▾ Catalog ▾ Properties ▾ HTML ▾ Customers ▾ Marketing ▾ Utils ▾ Help

ADD PRODUCT DELETE PRODUCT VIEW PRODUCT GOTO PRODUCT Standard ▾ Change

1 of 1

ProductA

Data Categories Images Attachments Accessories Related Products **Stock** SEO Portlets Properties

Stock

Dimension

Weight: kg

Height: m

Width: m

Length: m

Stock Data

Deliverable:

Manual delivery time: ----- ▾

Manage stock amount:

Stock amount:

Order time: ----- ▾

Ordered at:

Packing

Active packing:

Amount per packing:

Packing unit: ----- ▾

SAVE STOCK

Powered by LFS — Lightning Fast Shop
© 2009 - 2011 by Kai Diefenbach and contributors — All rights reserved
Distributed under the BSD-License

Figure 10: Product stock fragment (LFS).

admin | Logout

Shop ▾ Catalog ▾ Properties ▾ HTML ▾ Customers ▾ Marketing ▾ URLs ▾ Help

ADD PRODUCT DELETE PRODUCT VIEW PRODUCT GOTO PRODUCT Standard ▾ Change

1 of 1

ProductA

Data Categories Images Attachments Accessories Related Products Stock SEO Portlets Properties

Data

General
 Active:
 Name:

URLs
 Slug:
 Redirect to:

Article numbers
 SKU:
 Manufacturer:
 SKU Manufacturer:

Prices
 Price:
 Price unit:
 Tax:
 Price calculator:
 For sale:

Quantity field
 Quantity field unit:
 Type of quantity field:

Base price
 Active base price:
 Base price unit:
 Base price amount:

Short description:

[Rich text editor toolbar]

Description:

[Rich text editor toolbar]

Appearance
 Static blocks:
 Product template:

Figure 11: Product general data fragment (LFS).

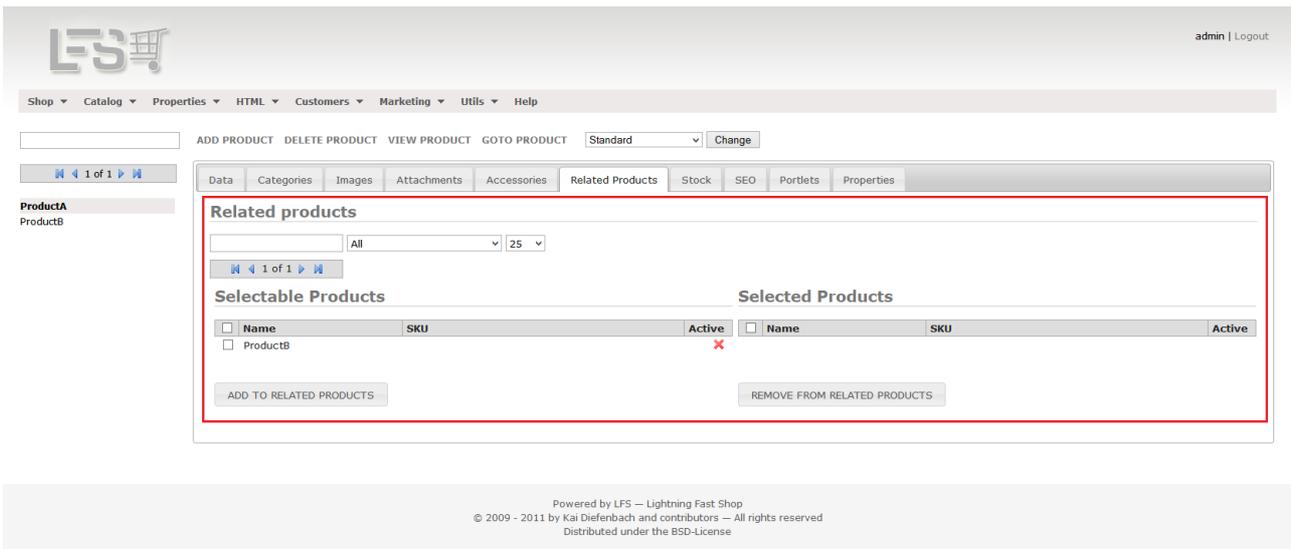


Figure 12: Related products fragment (LFS).

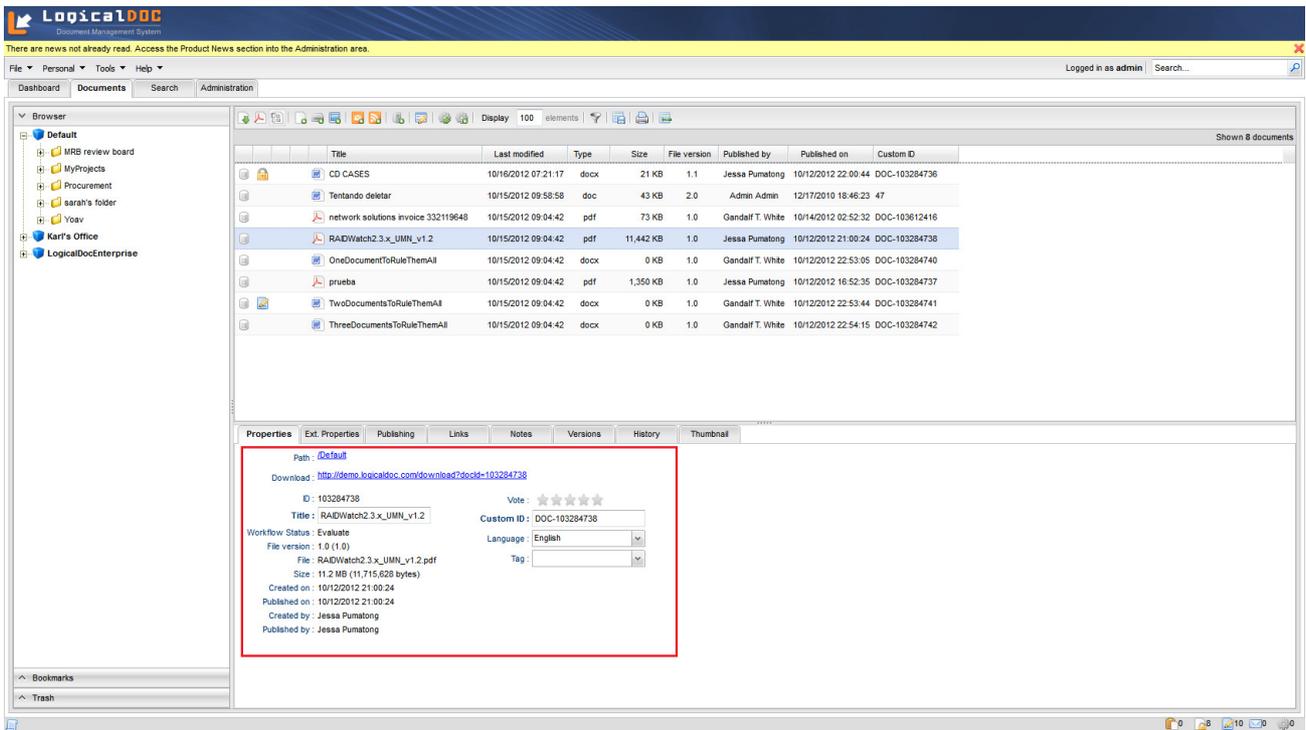


Figure 13: Document standard properties fragment (LogicalDoc).

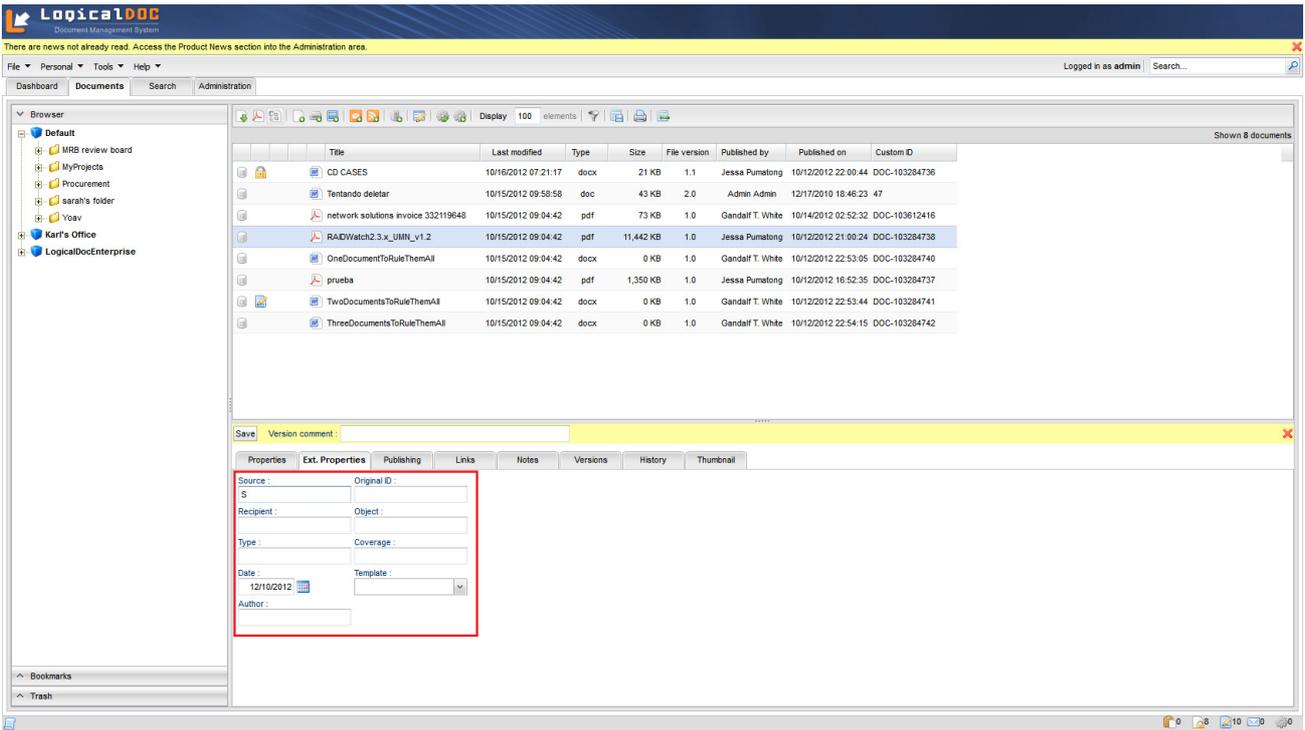


Figure 14: Document extended properties fragment (LogicalDoc).

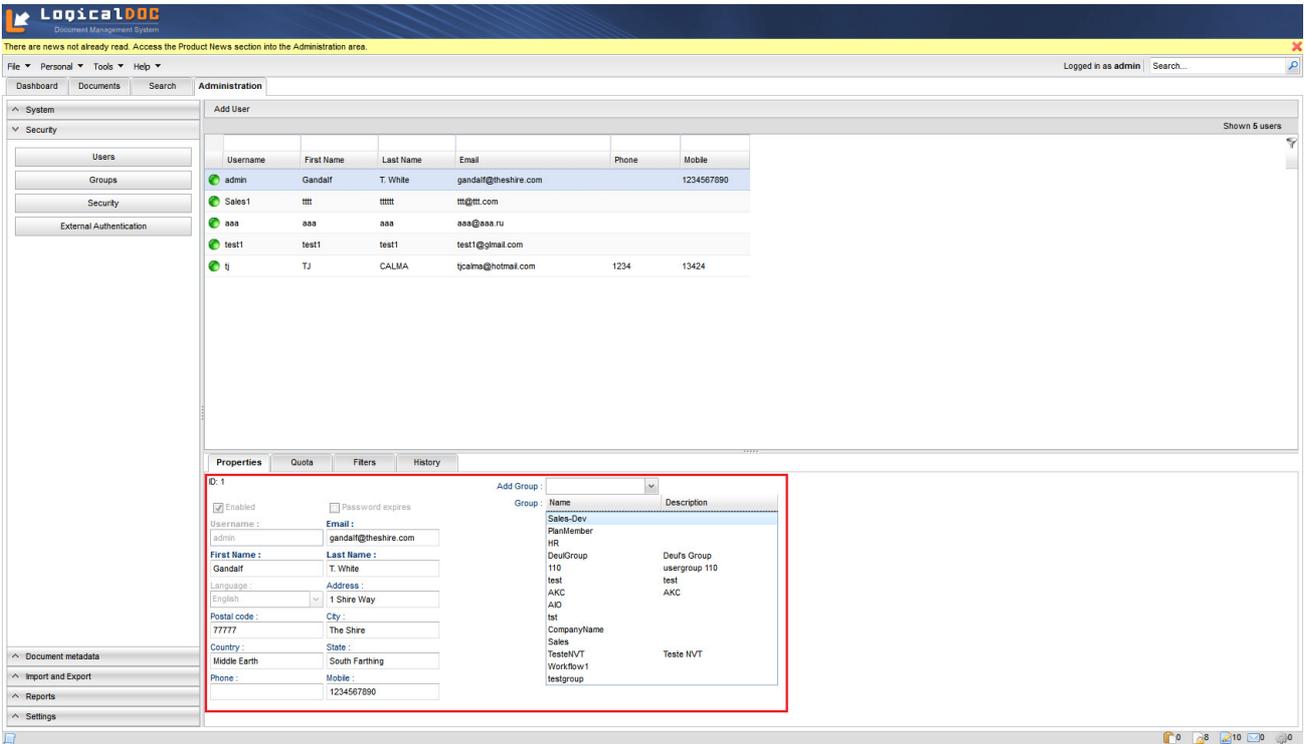


Figure 15: User properties fragment (LogicalDoc).

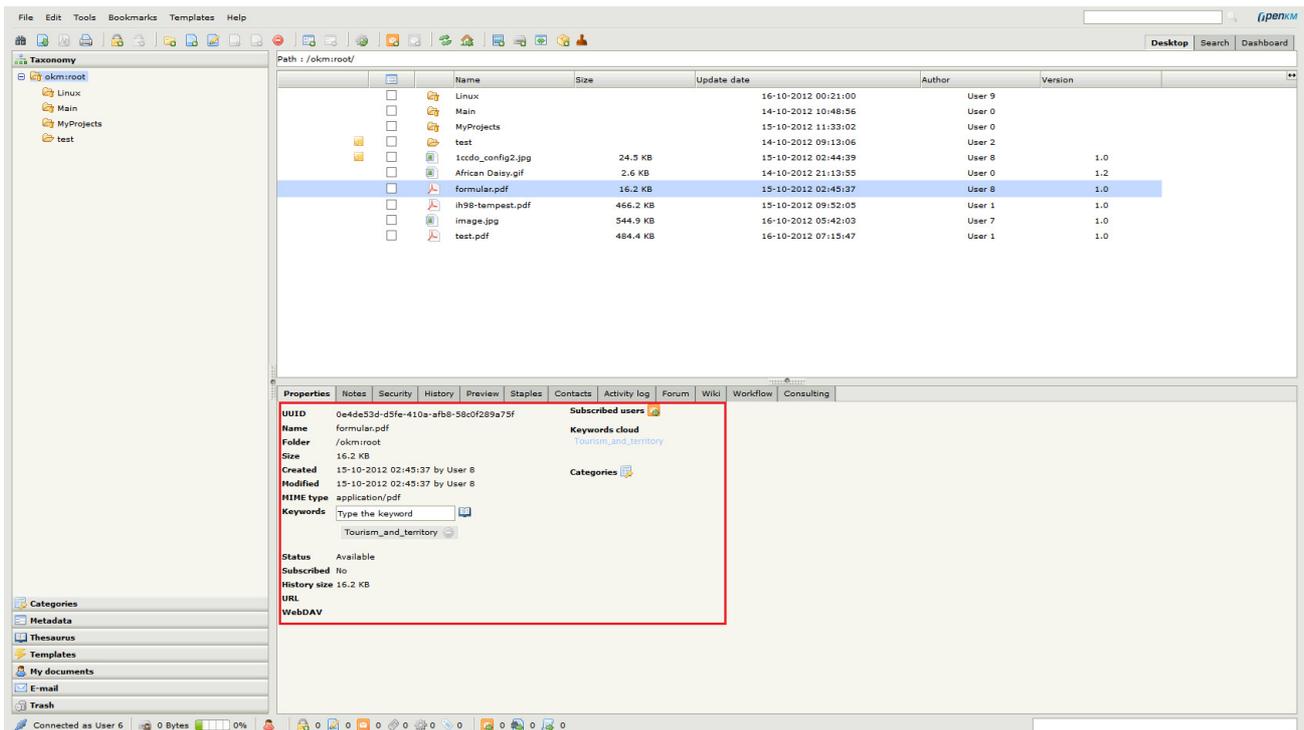


Figure 16: Document details fragment (OpenKM).

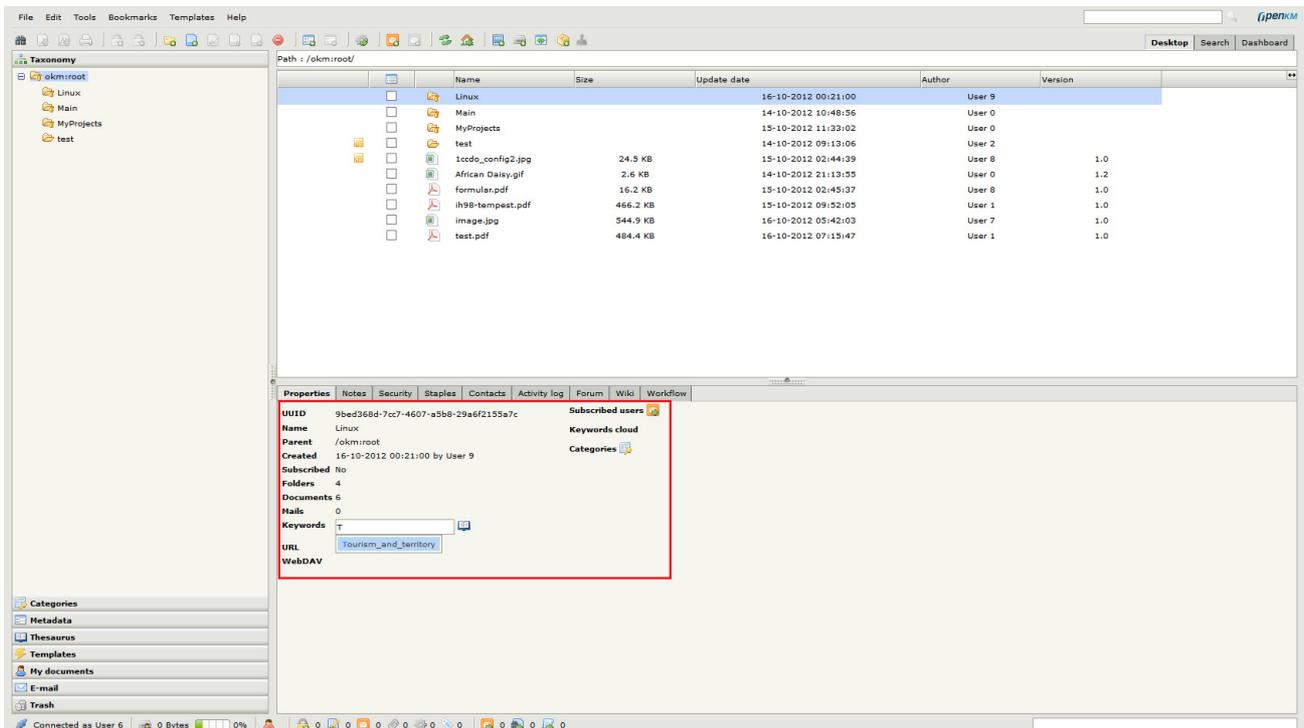


Figure 17: Folder details fragment (OpenKM).

User configuration

User account

User id: user6

User name: User 6

Password:

E-mail:

Roles: ROLE_USER

Mail account

IMAP server:

IMAP user name:

IMAP user password:

IMAP folder:

Figure 18: User config fragment (OpenKM).

Admin

Home Projects Houses Persons Companies Products News Minutes Documents Logout

Admin

Create General Ownership Password

Name:

Personal ID Number:

Gender:

Primary Role:

Description:

Impressions:

Miscellaneous:

Misc. File:

Company:

Email:

Phone 1:

Phone 2:

Address:

City:

Postal code:

Country:

Picture:

Copyright 2011-2012, HouseApp. All rights reserved.

Figure 19: Party contact details fragment (CRMS).

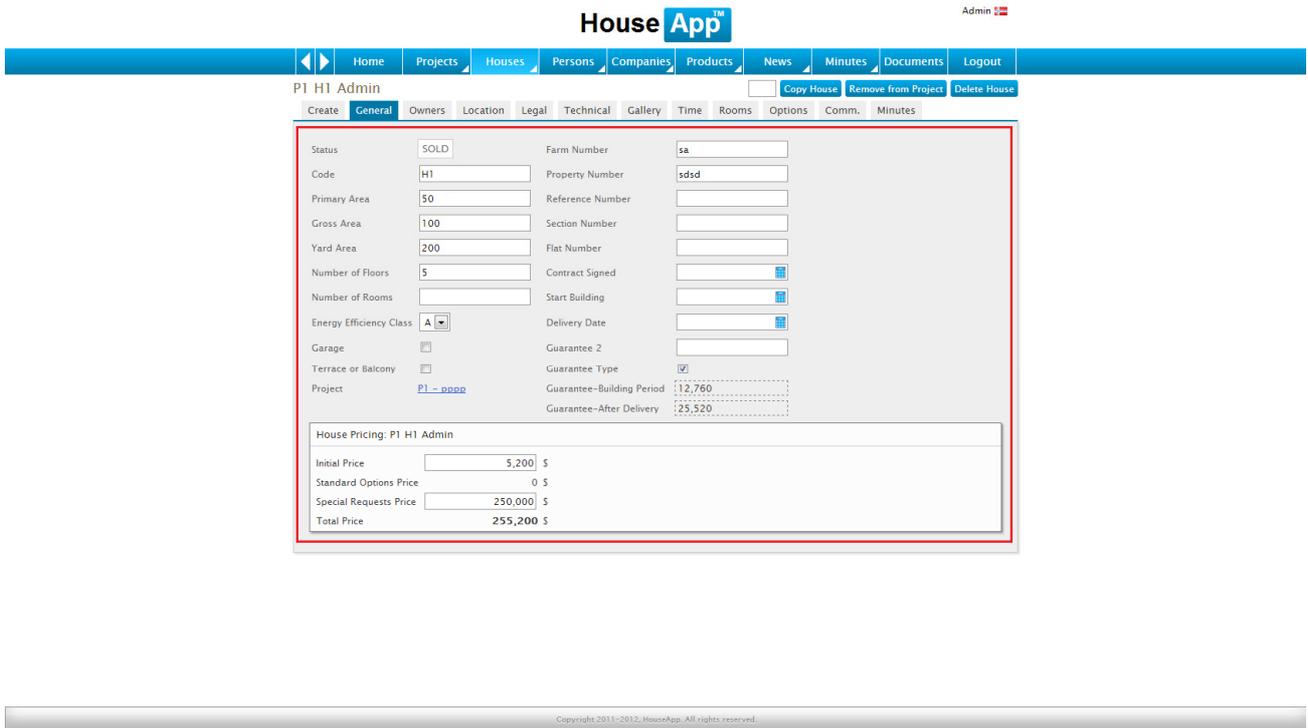


Figure 20: House general details fragment (CRMS).

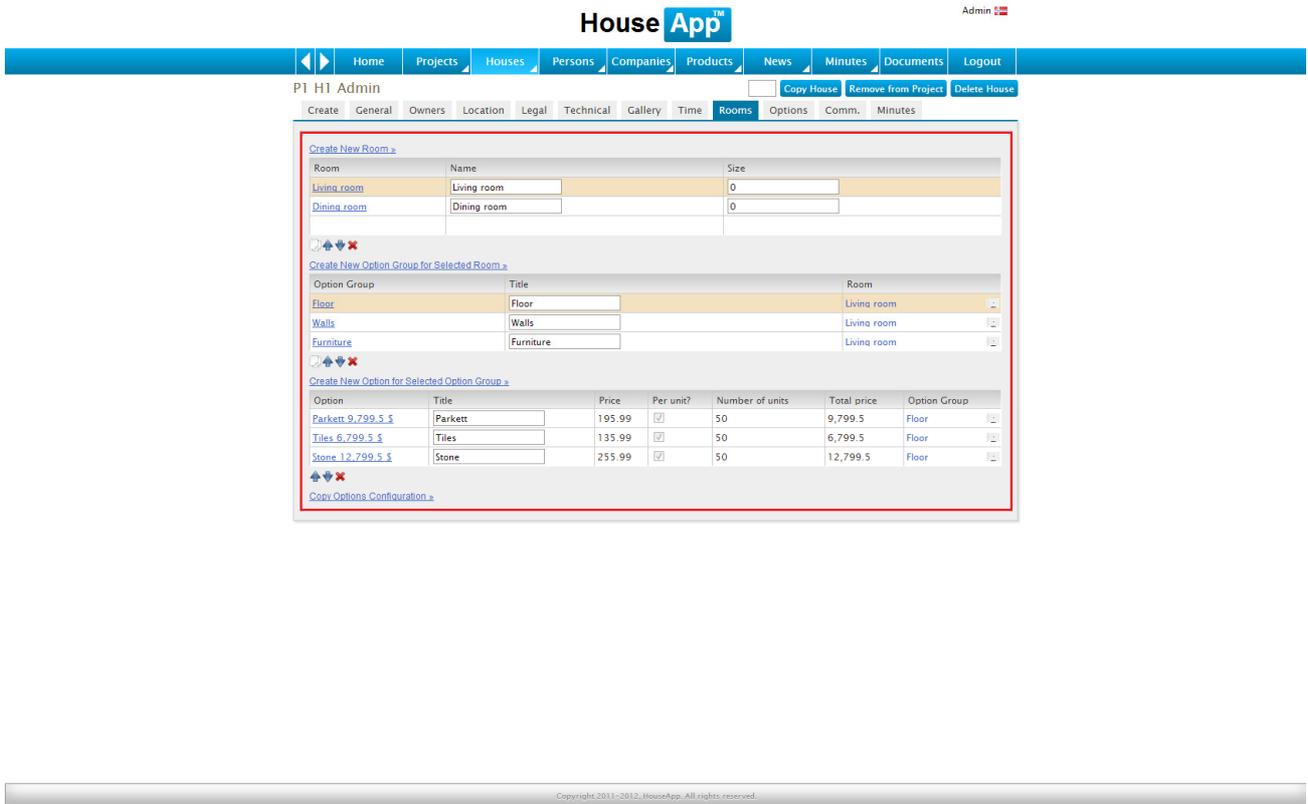


Figure 21: House rooms fragment (CRMS).

Skade, detaljer 3456654112

Id: 1

Skade status: Aviser behandling

Post: Post 1

Sendt av: 22.08.12 14.08

Behandlingstidspunkt: 22.08.12 14.06

Sendt til:

Ambulanse/bil reg.nr:

Sted:

UTM koordinater: 31 N 166021 0

Langde: 0.0

Bredde: 0.0

[Location on Google Maps](#)

Kontaktperson:

Aviser behandling:

Rapport: Erik engh

Lag rapport

Skader	Vedlegg	Viktige punkt	Pasient
22.08.12 14.08		Tidspunkt for registrering:	Ingen reaksjon
		GCS syns:	Ingen reaksjon
		GCS språk:	Ingen reaksjon
		GCS motorikk:	Ingen reaksjon
		Pust:	
		Puls:	
		Blodtrykk systolisk:	
		Blodtrykk diastolisk:	
		Venstre pupill:	
		Høyre pupill:	
		Temperatur:	
		Nedkjøling:	<input type="checkbox"/>

Figure 22: Vital signs reading details fragment (ICMS).

Skadessted Skadetype

- Head Injury
- Fracture
- Fire Burns
- Other
- Poisoning
- Spraining
- Lifeless
- Fall Injury
- Unconscious
- Cold Injury
- Blisters
- Stretching Injury
- Bleeding
- Heat Injury

Ny mappe

Id: 7

Navn: Head Injury

Beskrivelse:

Figure 23: Directory entity details fragment (ICMS).



Figure 24: Patient details fragment (ICMS).

Analysis

Identifying Sample Artifacts and Code Sections

We have decided to use at least three UI fragments (modules, pages, forms) for each selected technology. We were interested in fragments that resemble standard UI patterns. For obvious reasons, we picked test fragments by hand, browsing through applications and searching for the appropriate ones.

After deciding which UI fragments will be analyzed, the corresponding code had to be fetched from the codebase. This includes any template/layout related files for the given technology (.html, .aspx), JavaScript code, and server-side code (Java, C# or Python). We are interested only in code that is directly related to the selected UI fragment. In some cases, one file contains code for different parts of UI, so only relevant parts of file are used. Deciding where to cut files is a quite ambiguous process. A rule of thumb is to include every piece of code that developer(s) had to write to get desired user interface.

In the case of server-side code, e.g. UI widget construction, data access code, event-handlers etc., we considered both UI related and non UI related code. This is because we regard mixing these two as a bad coding style.

Preparation of the Sample Artifacts

Before the classification process, every code artifact is prepared to ensure consistent and fair results. Empty lines, inline comments and comment sections are ignored. The code fragments were already formatted appropriately for their respective languages. C# and Java formatting styles are almost identical (e.g. opening curly brace is in line with language constructs). Maximum line width was not limited.

Additionally, auto generated code was ignored. We were analyzing only the code that had to be written by developers. This does not apply to code generated by IDE helpers (getters and setters or similar), because it is difficult to recognize it, and secondly it still has to be initiated by developer. In our opinion, this does not favor any competitor. Unfortunately, we could not know if developers used any graphical design tools, so estimation of the developer effort by sheer number of LOC may be inaccurate. On the other hand, every other measured aspect is still valid.

Metrics

Software metric used in our analysis is physical source lines of code (LOC). Admittedly, the LOC measure is not a very reliable way to measure software size or other qualities, except that all the other ways are worse. Although involved UI frameworks use different underlying programming languages we are not comparing size and complexity between projects. We are more concerned with estimation of developer effort required in building UIs and trying to measure percentage of code concerned with the certain aspects.

For each UI fragment, every line of code is marked with a flag: template/layout flag, UI component flag, data access flag, business logic flag, “sticky tape” flag or incomprehensible flag. LOC could be left unmarked or marked with any number of flags. Results are then compiled in the following form:

1. Overall organization and clarity:
 - a. Total lines of code
 - b. Template/layout-related code
 - c. Comprehensible LOC: those that can be understood what they are supposed to do without particular domain and application knowledge, just with knowledge of the language and technology/framework
 - d. Incomprehensible LOC
 - e. Overall mark for structure (1-5) (subjective)
 - f. Overall mark for clarity (1-5) (subjective)
2. Pitfalls (anti-patterns):
 - a. “Magic Pushbutton” (business logic in UI code)
 - b. Others (other unclear or weird code)
3. Essential and Accidental Complexity:
 - a. Data access code in UI code
 - b. “Sticky tape” (accidental complexity due to unexpected tweaks and workarounds in code to fix weaknesses or limitations of the technology)
 - c. UI components (that have similar counterparts in SOLoist)
 - d. UI inter-component behavior

The very process of counting and classifying LOC was done in Excel. Excel sheet was prepared so that every LOC was placed in the first column of its row. Subsequent columns were flags, with possible values of one or zero meaning the flag holds or does not hold for that particular LOC. Lines were highlighted by different color for each flag. In the next section we describe the flag types more thoroughly.

Classification Based on UI Programming Concerns

Data Access Flag. Providing relevant information to the user and parsing user’s input is essential role of the UI. All contemporary software systems use relational databases to store data, and object-oriented paradigm to manipulate data. On the other hand, presenting data on the UI requires substantial additional effort on developer’s behalf. We used data access flag to mark any occurrence of storing or reading from the data source.

UI Component Flag. We used UI component flag to mark any instantiation and initialization occurrence of the UI structural elements (containers and widgets). In case of the template-based frameworks, we marked appropriate HTML elements, which roughly correspond to SOLoist components.

Template/Layout Flag. We used Template/Layout flag to mark any template (HTML) code and any procedural code used to define position or visibility of the components.

Inter-Component Behavior Flag. By inter-component behavior, we refer to any communication event between the UI components. For example, clicking on button clears input field or selection of item in the list enables buttons etc. In addition, user triggering some business action (e.g. clicking on delete button) was considered inter-component behavior, similar to execution of command in SOLoist. This usually corresponds to code in event-handlers (on server or client Java Script), but in case server pages frameworks (ASP.Net, Django) it can be more difficult to classify this behavior as will be discussed later.

Other Classification Groups

Business Logic Flag. The magic pushbutton is a common anti-pattern found in UI code. In essence, it is execution of business logic code in UI event handlers or initialization procedures. This overcomplicates implementation and can cause duplicate code, which is difficult to maintain. We flagged any code in the UI which manipulates with domain objects as business logic. Often business procedures must be called from UI, but ideally this should be just one line of code i.e. method call on domain object.

Sticky Tape Flag. Accidental complexity due to unexpected tweaks and workarounds in code to fix weaknesses or limitations of the technology is marked with sticky tape flag. For example, creating invisible components on the UI just to store temporary values is considered sticky tape.

Incomprehensible Flag. We marked any code that we could not understand its purpose with incomprehensible flag. Particular domain and application knowledge was not necessary. High occurrence of this flag is the indication of a system which is difficult to maintain.

Classification Problems

Due to difficult nature of this research, we encountered number of different problems while classifying lines in the source code. We will list them here as a reference and provide solutions or conventions we used to overcome the problem.

Language Constructs. We encountered the first problem in conditional statements, that is, how to count opening statements. Usually these statements are not flagged (neither is closing curly brace for C# or Java code). There is one exception to this rule, if conditional expression can be marked with any of the flags, the opening line is flagged. The same applies to *switch statement*. Loop statements (*for*, *do*, *while*) follow similar reasoning, as well.

Statements inside method bodies can be arbitrarily complex, and can be marked with multiple flags. Therefore, we have decided to left *method signatures* unmarked, except if there is absolutely no ambiguity (e.g. all statements inside body are layout related so is the method signature). This helps classifying method calls. It can be very complicated to analyze call hierarchy. If method definition is unequivocally classified we can flag *method call* with same flag, otherwise it is left unmarked. Additionally, if method call contains expression that can be separately marked we will do so. *Imports*, *class definitions*, *exception handling* and other language constructs are left unmarked.

Spatial Locality of Code. Lines of code usually cannot be analyzed independently from the neighboring lines. Logical blocks of code often perform task of the single UI programming concern. The statements that are grouped together in those blocks can be all marked with the same flag. That way some statements (e.g. declaration and/or initialization of *local variables*) can be classified even though that would not be possible otherwise.

SOLoist Counterparts

After initial analysis of the selected applications we were interested to compare analyzed UI fragments to fragments with the same functionality but developed with SOLoist. The only way to do it was to actually develop a new set of counterpart fragments. To make this simpler we assumed following:

- an adequate domain conceptual model exists,

- all business logic is implemented, located in the backend, and easily available (through one line of code) to the UI code,
- SOLoist counterpart fragment should be as close as possible to the original

Then we counted the SOLoist counterparts':

- total lines of code (LOC),
- UI components,
- wires (bindings),
- commands.

Results of the Analysis

The results of the analysis are provided in the separate Excel files as follows:

- C#/ASP.NET
 - DashCommerce.xlsx, e-commerce application
 - NopCommerce.xlsx, e-commerce application
- Python/Django
 - Satchmo.xlsx, online store framework
 - LFS.xlsx, online shop and e-commerce solution
- Java/GWT
 - LogicalDoc.xlsx, document management system
 - OpenKM.xlsx, document management system
- SOLoist
 - CRMS.xlsx, real estate agency customer relationship system
 - ICMS.xlsx, document management system
- Summary.xlsx, summary of results of the analysis

Each file contains analysis results for only one technology. For each technology, the results are based on a selection of three modules and corresponding code artifacts. The source code of each analyzed artifact as well as the classification of its LOCs are within the file on a separate artifact-specific sheet. The first sheet in each file contains the summary of the results for that file, that is, for one concrete application. The last sheet in each file contains the results of the performance analysis for corresponding application. The Summary.xlsx contains the summary of results of the whole analysis. Finally, SOLoist counterpart artifacts are in separate folders according to their names.

References

- [1] DashCommerce, E-Commerce Online-Store Application, <http://www.dashcommerce.org>
- [2] NopCommerce, E-Commerce Solution, <http://www.nopcommerce.com>
- [3] Satchmo, Online Stores Framework, <http://www.satchmoproject.com>
- [4] LFS, Online Shop and e-Commerce Solution, <http://www.getlfs.com>
- [5] LogicalDoc, Document Management System, <http://www.logicaldoc.com>
- [6] OpenKM, Knowledge Management, <http://www.openkm.com>