HAL
open science

Enhancing Use Cases with Screen Designs

Łukasz Olek, Jerzy Nawrocki, Miroslaw Ochodek

To cite this version:

Łukasz Olek, Jerzy Nawrocki, Miroslaw Ochodek. Enhancing Use Cases with Screen Designs. 3rd Central and East European Conference on Software Engineering Techniques (CEESET), Oct 2008, Brno, Czech Republic. pp.48-61, 10.1007/978-3-642-22386-0_4 . hal-01572553

HAL Id: hal-01572553

https://inria.hal.science/hal-01572553

Submitted on 7 Aug 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Enhancing Use Cases with Screen Designs*

Łukasz Olek, Jerzy Nawrocki, Mirosław Ochodek

Poznań University of Technology, Institute of Computing Science,
ul. Piotrowo 3A, 60-965 Poznań, Poland
{Lukasz.Olek, Miroslaw.Ochodek}@cs.put.poznan.pl

Abstract. This paper presents a language called ScreenSpec that can be used to specify screens at requirements elicitation phase. ScreenSpec was successfully applied in 8 real projects. It is very effective: average time needed to specify a screen is 2 minutes, and takes an hour to become proficient in using it. Visual representation generated from ScreenSpec can be attached to requirements specification (e.g. as adornments to use cases).

Keywords: Use cases, GUI Design, Prototyping, ScreenSpec

1 Introduction

Use cases are the most popular way of specifying functional requirements. A survey published in IEEE Software in 2003 [12] shows that over 50% of software projects elicit requirements as use cases. Use case is a good way of describing interaction between a user and a system at a high level, so maybe now the number can be even higher. At the same time many practitioners (in about 40% of projects [12]) draw user interfaces to visualise better how the future system will behave. This is wise, since showing user interface designs (e.g. prototypes [14,17,8,18], storyboards [10]) together with use cases helps to detect more problems with requirements ([13][1]). Unfortunately, user interface details would clutter use-case description and should be kept apart from the steps ([6,7]), but they can be attached to use cases as adornments [6].

Much have been said about writing use cases [6,7,16,11,9] (e.g. how to divide them into main scenario and extensions, what type of language to use), however it is not clear how to specify UI details as adornments. Practitioners seem to either draw screens in graphical editors and attach graphical files to use cases, or just describe it using a natural language. Both approaches have advantages and disadvantages. The graphical approach is easier to analyse by humans, however more difficult to prepare and maintain. On the other hand, the textual approach is much easier to prepare, but not so easy to perceive.

The goal of this paper is to propose a simple formalism called ScreenSpec to specify user interface details. It has both advantages of the approaches mentioned

* This research has been financially supported by the Polish Ministry of Science and Higher Education grant N516 001 31/0269.

[1] See experiment conclusions in section: "Mockup helps to unveil usability problems".

earlier: it is inexpensive to prepare and maintain, and can be automatically converted to a graphical form (attached to use cases as adornments can stimulate readers visually). Currently the language is limited to describe user interfaces of web applications.

It is easy to propose a new formalism, but it is much more difficult to prove that it is useful. ScreenSpec has been successfully used in 8 real projects. An investigation was carried out to find out how much effort is needed to use ScreenSpec, and how much time does it take to learn how to use it.

The plan of this paper is following. Section 2. describes some approaches to UI specification that are popular in software engineering. Section 3. describes the ScreenSpec language. Section 4. describes case studies that were conducted to check whether ScreenSpec is complete enough and flexible to specify real applications and how much effort does it take to specify screens at requirements elicitation phase. Section 5. describes a way to generate graphical files representing particular screens from ScreenSpec. Section 6. presents how the visual representation of screens can be embedded in requirements documents: as adornments, or as a mockup. The whole paper is concluded in Section 7.

2 Related Work

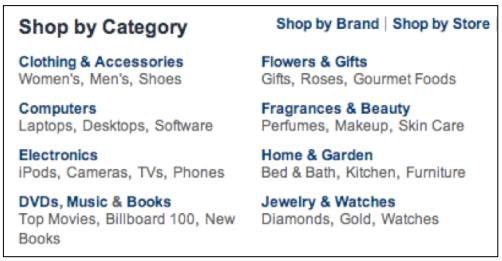
There are many outstanding technologies and formalisms used to describe user interface. One of them is XUL[2] (XML User Interface Language) from the Mozilla Foundation. This is an interesting language used to describe platform-independent user interface. It has many interesting features, e.g. separation of presentation from application logic, easy customisation, localisation or branding. However this language seems to be too much implementation-oriented, to be used at requirements elicitation phase. There are two important impediments to use XUL for this purpose. Firstly, one has to know component type before it can be specified (e.g. `<textbox id="username"/>`).Therefore it is not possible to start from specifying the structure of information connected with particular screen, and then add details about control types (see Section 3.3). Secondly, labels for components have to be defined explicitly. This means that for each component two declarations are required: one for a label, and second one for a control. However it would be interesting to generate (or round-trip-generate) XUL from ScreenSpec. Produced XUL could be easily used for implementation.

There are other technologies that are getting more and more popular nowadays, like for instance, MDSD (Model Driven Software Development) approaches, that provide an ability to generate a whole application from a set of models (e.g. WebML[3], UWE/ArgoUWE[5]). Unfortunately this approach still seems to require too much effort, to be successfully used at the requirements elicitation phase. There are companies using such approach, that Poznań University of Technology cooperates with. According to their experience it takes at least several hours to describe a single use case with MDSD models. It is definitely to long to be used at requirements elicitation phase, so they use generic text editors to specify use cases and screens.

3 ScreenSpec - Language for Screen Specification

The goal of the ScreenSpec language is to allow analyst to specify the structure of user interface very efficiently. Since this approach is supposed to be used at early stages of requirements elicitation phase, it would be wise to focus on the structure of screens and information exchanged between a user and a system, rather then on such attributes like colours, fonts, layout of components. This is called a lo-fidelity approach([19,15]) and is used in ScreenSpec.

It is best to explain how ScreenSpec specification looks like on a simple example. Let us imagine an Internet shop with a view of all categories (see Fig. 1). It contains a list of categories (each of them is represented by a link with its name) and lists of subcategories. There are also some links to other shopping-views (by brand, by store, etc).




```
SCREEN Shop_By_Category:
    Categories(LIST):
        Name(LINK)
        Subcategories(LIST):
            Name(LINK)
    Shop_by_Brand(LINK)
    Shop_by_Store(LINK)
```

Fig. 1. A screen showing a list of categories (screenshot taken from shopping.yahoo.com), and its corresponding specification in ScreenSpec.

3.1 Language Definition

Screens Each screen is specified with a set of lines that describe its structure. The definition starts with keyword `SCREEN` followed by screen's identifier (each screen must have an unique ID). The following lines are indented and describe components, that belong to the screen:

```
SCREEN Screen_ID:
    Component_definition_1
    Component_definition_2
    ...
```

Where `Component_definition_n` is a definition of a basic component or a group.

Basic components Basic components are mainly simple widgets taken from the HTML language. They are specified using the following syntax:

```
Component_ID[(CONTROL_TYPE)]
```

where:

- `Component_ID` is a string that identifies the component, e.g. `Login_name`, `Password`, `Author`. It has to be unique in the scope of the screen or parent group.
- `CONTROL_TYPE` is one of: `BUTTON`, `LINK`, `IMAGE`, `STATIC_TEXT`,`DYNAMIC_TEXT`, `EDIT_BOX`, `COMBO_BOX`, `RADIO_BUTTON`, `LIST_BOX`, `CHECK_BOX`, `CUSTOM`. This parameter is optional (`EDIT_BOX` by default) and can be defined by an analyst later (see Section 3.3).

Three controls do not come from a standard set of HTML controls, thus require additional comment: (1)`STATIC_TEXT` is used to display a text that will be the same on each screen instance (e.g. a comment or instructions) (2)`DYNAMIC_TEXT` is a block of text generated dynamically by the system (e.g. a total value of an invoice) and (3)`CUSTOM` is used to represent non-standard components (e.g. date picker).

Groups Groups of components are containers for widgets, usually providing additional features. They are defined using the following syntax:

```
Group_ID[(GROUP_TYPE)]:
    Component_definition_1
    Component_definition_2
    ...
```

where:

- `Group_ID` is a string that identifies the group.
- `Component_definition_n` - is a definition of a basic component or another group of components
- `GROUP_TYPE` - determines the type of the group:
  - `SIMPLE` or omitted - such group is just used to introduce structure to the screen, but it does not provide any additional semantics to its components
  - `LIST` - its components are repeated in a list, all child components form a single list item
  - `TABLE` - its components are repeated in rows, as a table (similar to `LIST`, but different layout)
  - `TREE` - similar to list, but its items can also contain another lists of the same structure (e.g. used to create tree of categories, or site map tree).

3.2 ScreenSpec Advanced Elements

Static values are used to specify, that the component will have the same values on each screen. For example, when we have a combo box that allows to choose sex of user, it would always have two values: "Male" and "Female". We can express such case in ScreenSpec using the following structure:

```
Component_ID(...):Value_1|Value_2|...
```

e.g. `Sex(COMBO_BOX):Male|Female`

Template mechanism can be used to build more complex screens. Two elements are needed to do this: a template definition and a reference to the template in the particular screen. This can be done using the following structure (shown in the example):

```
TEMPLATE Main:
   Header:
      Sign_in(LINK)
      Register(LINK)
   ...
   Contents(CUSTOM)
   ...

SCREEN All_categories(Main):
   ...
```

Each template is defined with keyword `TEMPLATE` followed by a template name. The structure of template is the same as the structure of a screen. The main difference is that it requires putting the additional component `Contents(CUSTOM)` - which is a placeholder for the specific screen which will use the template. When screen is supposed to use a particular template, we need to mention the name of the template in parenthesis, e.g. `SCREEN All_Categories(Main)`.

Include common screen elements When there are the same controls (functional blocks) used in more screens it would be wise to declare them once and then make references. We can do this by using `INCLUDE` keyword:

```
SCREEN Search_box:
   ...

SCREEN Search_results:
   INCLUDE Search_box
   Results(LIST):
      ...
```

Select clause It is common that a screen has different variants depending on situation. For example, a screen can display a list of all accepted papers or a message that no papers are accepted yet. When one of several variants can be used, it is specified by a select clause:

```
   SELECT:
      Component_definition_1
      Component_definition_2
```

This means that either `Component_definition_1` or `Component_definition_2` will be displayed on a real screen. More than two variants are possible (additional lines should be added within the `SELECT` clause). Components, that are specified here can be both simple components, or groups.

3.3 Iterative Approach to Screen Specification

ScreenSpec is designed to be used by an analyst at requirements elicitation phase. This phase is exploratory, which means that change-involving decisions are made frequently. It would be important to provide incremental approach to the screen specification process. It would be great if analyst could just roughly describe screen at the beginning (only the structure of information), and add more details later (when a customer confirms it is correct). Therefore ScreenSpec has 3 levels of details:

L1 *Component names* - need to be specified at the beginning.
L2 *Types of controls and groups* - specifies types of information connected with each screen.
L3 *Static values and templates.*

These levels can be mixed throughout the specification process: some fragments of screens can be written at one level of details, whereas other ones at another level.

4 Experience with ScreenSpec

4.1 Specifying Screens for the Real Projects – Case Studies

Analysts usually use word processors and sheets of papers to author requirements. Keeping it in mind, it seems that introducing formalised requirements models can be risky. It may happen that some of the developed-system features might be too difficult to describe.

To make sure that the ScreenSpec formalism is complete and flexible enough to be used for describing real systems, eight case studies were conducted. They included a large variety of projects. Some of them were internally-complex (large number of sub-function[2] requirements), with a small amount of interaction with a user (e.g. Project A, Project C). Others were interaction-oriented, with a great number of use cases and screens (e.g. Project D, Project G). First 6 projects were selected from the Software Development Studio course at Poznań University of Technology. These projects were developed for external customers by students of the Master of Science in Software Engineering. Students were successfully using ScreenSpec approach to specify screens. They also raised some minor suggestions for ScreenSpec language, and small simplifications were introduced afterwords. Then screens for two commercial projects were also written using ScreenSpec language. In both cases all screens were successfully specified.

It seems that number of lines of code (LOC) per screen may differ depending on the screen complexity. In analysed projects average LOC per Screen varies from 3.0 to 14.5 (see table 1).

[2] After Cockburn[7]: a sub-function requirement is a requirement that is below the main level of interest to the user, i.e. “logging in”, “locate a device in a DB”.

Table 1. Eight projects selected for the case study

Project	Number of Use Cases			Screens	ScreenSpec Screens	
	Business	User	Sub-functional		Average LOC/Screen	Total LOC
Project A	0	4	2	4	14.5	58
Project B	3	13	2	5	9.4	47
Project C	0	5	0	4	3	12
Project D	0	16	0	27	4.7	128
Project E	0	4	0	7	3.9	27
Project F	1	3	2	3	13	39
Project G	0	44	39	92	9.5	917
Project H	2	12	0	7	5.1	36

4.2 ScreenSpec Efficiency Analysis

Although an average amount of code required to specify a screen with ScreenSpec seems to be rather small, two important questions arise:

- Q1: how much effort is required to specify[3] a screen?
- Q2: how much time is required to learn how to use ScreenSpec?

The second question is also important because practitioners tend to choose solutions, which provide business value and are inexpensive to introduce. If an extensive training is required in order to use ScreenSpec efficiently, there might be a serious threat, that the language will not be attractive to the potential users.

In order to answer these questions, a controlled case study was conducted[4]. Eight participants were asked to specify sequence of 12 screens coming from the real application (provided as the series of application screenshots). The time required for coding each of the screens was precisely measured (up to the seconds). The code was written manually on sheets of paper. Participants were also asked to copy a sample screen specification, in order to examine their writing speed. Before they started to specify screens, they had been also introduced to the ScreenSpec during the 15-minute lecture, and each of them was also provided with a page containing the ScreenSpec specification in a nutshell. All materials provided to participants are published at [1].

[3] The term "specifying" is understood here as the process of transcribing the vision of the screen into the ScreenSpec code.

[4] The case study is labeled here as a controlled, because the methodology was similar to that used in case of controlled experiments, however the nature of questions being investigated refers rather to the "common sense", then to some obtainable values (e.g. compare average learning time, to the one which is acceptable for the industry).

Descriptive analysis and data clearing. During the completion of each task (single screen specification) two values were measured:

- time required to finish the task
- lines of code developed to specify the screen

Screens specifications, developed by participants, differed in respect to their size, because they were specified only on the basis of the screenshots, which were be perceived slightly differently by different people. What is more, some of the ScreenSpec structures might be used optionally. The detailed results of the case study is presented in table 2. Before proceeding to the further analysis

Table 2. Effort and lines of code for each participant and task (*sample screen refers to the task measuring participants writing speed*)

	Time [min]										
Participant	**Sample Screen**	**1**	**2**	**3**	**5**	**6**	**7**	**8**	**9**	**11**	**12**
P1	0.9	4.3	2.0	2.0	6.2	2.3	4.7	7.7	3.3	5.5	6.3
P2	1.6	2.8	2.0	3.6	3.6	1.2	2.0	4.1	3.1	3.1	4.2
P3	1.2	1.8	2.0	3.0	4.6	1.5	2.4	6.2	4.8	3.9	4.4
P4	1.3	1.7	1.4	5.7	2.5	1.8	1.9	3.3	3.5	4.7	3.9
P5	1.0	2.4	2.2	1.0	2.5	1.4	1.8	3.6	4.3	3.0	4.2
P6	1.0	2.5	1.6	1.5	3.7	1.3	1.4	4.1	3.0	3.3	3.8
P7	1.0	1.9	1.5	1.6	2.0	1.1	1.3	2.3	1.8	3.2	3.5
P8	1.6	5.4	4.2	3.2	5.8	3.7	2.5	4.7	5.6	4.0	6.7
Mean	1.2	2.9	2.1	2.7	3.9	1.8	2.3	4.5	3.7	3.8	4.6
SD	0.3	1.3	0.9	1.5	1.6	0.9	1.1	1.7	1.2	0.9	1.2
	Lines of code - LOC										
Participant	**Sample Screen**	**1**	**2**	**3**	**5**	**6**	**7**	**8**	**9**	**11**	**12**
P1	8	14	6	8	17	7	20	35	20	25	32
P2	8	9	7	9	12	6	8	21	16	18	19
P3	8	8	6	9	10	6	7	22	16	18	25
P4	8	7	6	9	10	6	7	15	17	20	19
P5	8	11	8	8	11	6	8	16	16	15	23
P6	8	9	7	8	9	7	8	17	14	17	23
P7	8	10	7	9	10	6	7	17	15	21	21
P8	8	5	7	7	10	6	7	14	16	13	18
Mean	8.0	9.1	6.8	8.4	11.1	6.3	9.0	19.6	16.3	18.4	22.5
SD	0.0	2.7	0.7	0.7	2.5	0.5	4.5	6.8	1.8	3.7	4.5

results for all tasks were carefully analysed in order to find potential outliers. The task was marked as a suspicious if the variability in lines of code provided by participants was high (or there were outlying observations). According to the box plots presented in figure 2 tasks 1, 4, 5, 7, 8, 9, 10, 11, 12 were chosen for further investigation in order to find out the reasons for the LOC variability. It

turned out that tasks 4 and 8 were ambiguous, because in both cases there were two possible interpretations of the screens semantic. What is more, the amount of code required to specify each of two versions differed significantly. Therefore those tasks were excluded from the further analysis.

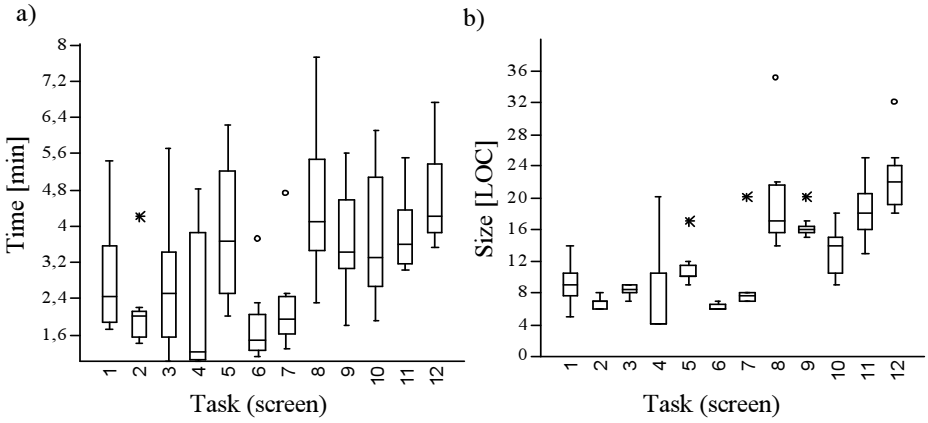


Fig. 2. Effort and size of code (LOC) variability for each task, *a) box plot presents effort variability for each task, b) box plot presents lines of code variability for each task*

Productivity analysis. Based on the effort and code size measured for each task performed by each participant, a productivity factor can be derived. It will be defined here as a time required to produce a single line of code. It might be calculated using equation 1.

$$PROD = \frac{Effort}{Size} \tag{1}$$

Where:

- $PROD$ - is a productivity factor understood as a number of minutes required to develop a single line of code
- $Effort$ - is an effort required to complete the task (measured in minutes)
- $Size$ - is the size of code developed to specify the screen (measured in LOC)

Effort measured during the case study consists of two components: (1)time required for thinking and (2)writing down the screen. It would be difficult to precisely measure both of them, however knowing the writing speed of each participant (see equation 2) it is possible to calculate the approximate effort

spent only on thinking. It can be further used to estimate cognitive productivity factor (see equation 3), which can be understood as a productivity of thinking while coding the screen. It is independent from the tool (effort needed mentally produce the screen-specification code).

$$V_{writing} = \frac{Size_{sample}}{Effort_{sample}} \tag{2}$$

Where:

- $V_{writing}$ - is a writing speed (measured in LOC per minutes)
- $Effort_{sample}$ - is an effort required to copy the code for the sample screen (measured in minutes)
- $Size_{sample}$ - is the size of the code for the sample screen – 8 LOC

$$PROD_{cognitive} = \frac{Effort - (Size/V_{writing})}{Size} \tag{3}$$

Where:

- $PROD_{cognitive}$ - is an estimation of cognitive productivity factor understood as a number of minutes spend on thinking in order to produce a single line of code
- $Effort$ - is an effort required to complete the task (measured in minutes)
- $Size$ - is the size of code to specify the screen (measured in LOC)
- $V_{writing}$ - is a writing speed (measured in LOC per minutes)

Cognitive and standard productivity factors were calculated for each task performed by participants. The chart presenting mean values for each task is presented in figure 3.

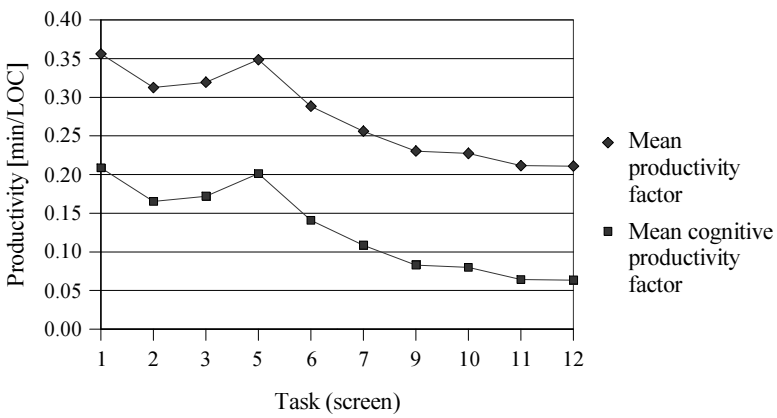


Fig. 3. Mean (cognitive and standard) productivity factors for each task (screen)

Q1: how much effort is required to specify a screen? If the mean productivities from the first and the last task are compared, it would mean that average beginner produces around 2.81 LOC / minute while person with some experience 4.7 LOC / minute (this of course may very depending on the screen complexity). That means that total effort of specifying all of the screens for the largest project included in the case studies – Project G (92 screens with total of 917 LOC of screen specifications) would vary from 3.2 to 5.4 hours depending on analyst skill. What is more, an average screen size is around 8 LOC (average from table 1), which could be specified in less then 2 minutes (for experienced analyst, and less then 3 minutes for beginner). Therefore it seems that the SpecScreen notation might be used directly during the meetings with customer. It is also worth to mention that if there was an efficient editor available (with high usability), the productivity factor for potential user, would be closer to the cognitive one. This means that a 8 LOC screen would be specified in about 30 seconds.

Q2: how much time is required to learn how to use ScreenSpec? By looking at the productivity chart presented in figure 3, the learning process can be investigated. The ratio between productivity factors calculated for the ending and beginning task is 1.69. In addition it seems that after completing 8-10 tasks, learning process saturates. Therefore it seems that participating in a single training session which includes a short lecture and ten practical tasks (about an hour), should be enough to start using ScreenSpec effectively.

An interesting observation is regarding the task number 5, because the productivity factor suddenly increased at this point (more time required to produce one line of code). This issue was further investigated, and the finding was that the screen for that task contained interactive controls, which appeared for the first time in the training cycle (edit boxes, check boxes etc.). Thus an important suggestion for a preparation of the training course would be to cover all of the components available in the ScreenSpec language.

5 Visual Representation of Screens

ScreenSpec can be authored using a dedicated tool. This is a simple editor, that detects each change, and automatically regenerates graphics files (PNG) that can be attached to requirements documents. The generator uses simple rules to transform ScreenSpec to visual representation:

1. For each component:
   - `EDIT_BOX`, `COMBO_BOX`, `LIST_BOX`, `CUSTOM` - a label (equal Component ID) is displayed on the left side of the control, the control's value is taken from the defined static value, or it is left empty. CUSTOM component is displayed as the `EDIT_BOX`.
   - `BUTTON`, `LINK` - displays a control with a caption equal to the defined static value, or component ID.

  - `STATIC_TEXT`, `DYNAMIC_TEXT` - displays a piece of text equal the static value or component ID.
  - `RADIO_BUTTON`, `CHECK_BOX` - displays a control followed by a label (label's value equals the static value or component ID)
  - `IMAGE` - displays a label on the left (equal to component ID) and an empty image frame on the right.
2. For each group:
  - `SIMPLE` - a header and a frame is created, all children components are placed inside this frame.
  - `LIST` - a header and a frame is created. In the frame 3 rows are displayed (this visualises that a list can have more elements): two rows having the child components, and the third one containing "..."
  - `TABLE` - is similar to a `LIST`, however a new table column is created for each child component. Its label is displayed in the table header rather then on the left (near its control).
  - `TREE` - is similar to a `LIST`, but for each row a nested and smaller list is displayed.

Following example (figure 4) shows a visual representation of a simple screen specified in ScreenSpec.

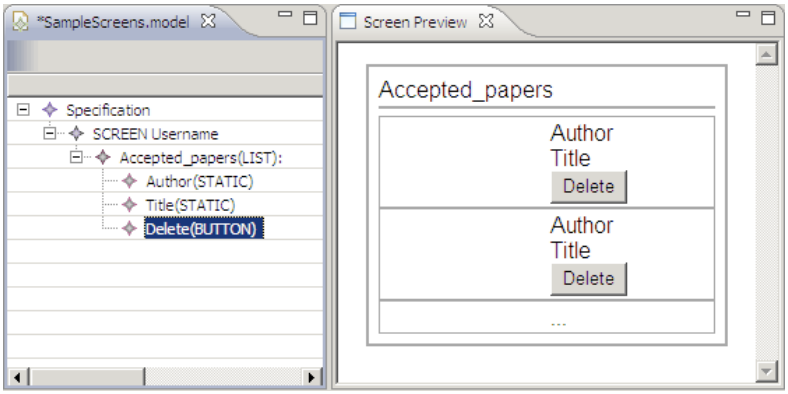


Fig. 4. An example of visual representation for a LIST group component.

6 ScreenSpec Meets Use Cases

Visual screens generated from ScreenSpec can be directly inserted into requirements specification in adornments section of particular use cases. Having up-to-date graphic files allows to update the specification very easily, because many modern text editors allow to link with external files, and update them each time the document is opened (e.g. Microsoft Word, OpenOffice).

6.1 Mockup

Mockup is an interesting artefact created by connecting screens to particular steps of use cases. It is rendered as a simple web application that can display both: use cases and screens at the same time. Use case (displayed on the left side) shows the interaction between an actor and a system (see figure 5). After selecting particular step, an according screen is displayed (on the right side). This artefact seems to be useful in practice, initial feedback from commercial projects using mockups is very positive.

It is difficult to connect screens to use case steps in generic text editor, so a dedicated tool called UC Workbench [4] was developed at Poznan University of Technology.

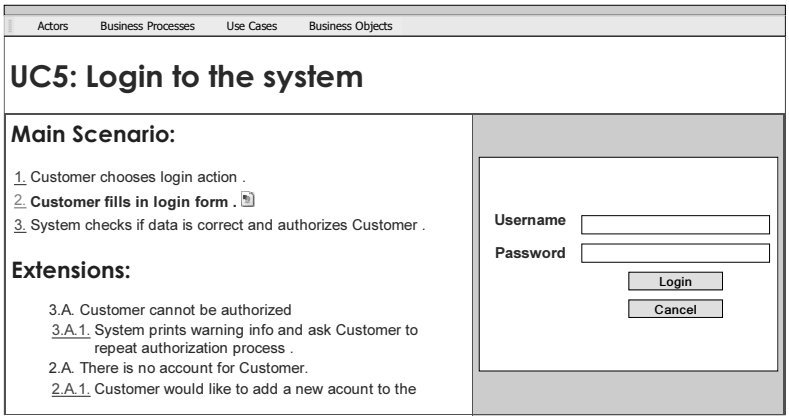


Fig. 5. An screenshot of Mockup - showing use case with corresponding screens at the same time.

7 Conclusions

User interface designs are often attached to use cases as adornments, because it helps to understand the requirements by IT laymen. However, it is not clear how to specify UI details. This paper proposes a language called ScreenSpec, that can be used for this purpose. ScreenSpec is a formalism that was thoroughly validated. It was used to describe UI in eight real software projects. ScreenSpec allows to work incrementally on screen designs, starting with the general structure of information at particular screen, and then adding more details about widgets. It is very efficient, it takes on average about 2 minutes per screen. ScreenSpec is also easy to learn, it takes about an hour, for a person that has never seen ScreenSpec, to become proficient in using it.

Although it is interesting to use ScreenSpec at requirements elicitation stage, it could be even more interesting to use it at later stages. One can think about generating skeleton user interface code (in XUL, SWT, Swing or other technologies), that could be refined during implementation. Appropriate research will be conducted as a future work.

Acknowledgements

Authors would like to thank companies which cooperate with Poznań University of Technology: Polsoft and Komputronik. They found time and courage to try our ideas in practice and provided us with a substantial feedback. This research has been financially supported by the Polish Ministry of Science and Higher Education under grant N516 001 31/0269.

References

1. A web page containing all materials for a ScreenSpec evaluation case study: http://www.cs.put.poznan.pl/lolek/homepage/ScreenSpec.html.
2. Home page for Mozilla XUL: http://www.mozilla.org/projects/xul/.
3. The Web Modeling Language Home Page. http://www.webml.org/.
4. UC Workbench project homepage. http://ucworkbench.org.
5. UWE - UML-based Web Engineering Home Page. http://www.pst.informatik.uni-muenchen.de/projekte/uwe/index.html.
6. Steve Adolph, Paul Bramble, Alistair Cockburn, and Andy Pols. *Patterns for Effective Use Cases.* Addison-Wesley, 2002.
7. Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2001.
8. Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.
9. Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
10. James A. Landay and Brad A. Myers. Sketching storyboards to illustrate interface behaviors. In *CHI '96: Conference companion on Human factors in computing systems*, pages 193–194, New York, NY, USA, 1996. ACM Press.
11. Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Use Case Approach, Second Edition.* Addison-Wesley Professional, May 2003.
12. CJ Neill and PA Laplante. Requirements Engineering: The State of the Practice. *Software, IEEE*, 20(6):40–45, 2003.
13. Łukasz Olek, Jerzy Nawrocki, Bartosz Michalik, and Mirosław Ochodek. Quick prototyping of web applications. In L. Madeyski, M. Ochodek, D. Weiss, and J. Zendulka, editors, *Software Engineering in Progress*, pages 124–137. NAKOM, 2007.
14. Roger Pressman. *Software Engineering - A practitioners Approach.* McGraw-Hill, 2001.
15. Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, 1996.

16. G. Schneider and J. P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1998.
17. Carolyn Snyder. *Paper Prototyping: The Fast and Easy Way to Define and Refine User Interfaces*. Morgan Kaufmann Publishers, 2003.
18. Yan Sommerville and Pete Sawyer. *Requirements Engineering. A Good Practice Guide*. Wiley and Sons, 1997.
19. Miriam Walker, Leila Takayama, and James A. Landay. High-Fidelity or Low-Fidelity, Paper or Computer? Choosing Attributes When Testing Web Applications. In *Proceedings of the Human Factors and Ergonomics Society 46th Anuual Meeting*, pages 661–665, 2002.