Interplay Between Requirements, Software Architecture, and Hardware Constraints in the Development of a Home Control User Interface

Michael Sørensen Loft, Søren Snehøj Nielsen, Kim Nørskov, Jens Bæk Jørgensen
Mjølner Informatics A/S, Aarhus, Denmark
{*mls, ssn, kno, jbj*}*@mjolner.dk*

***Abstract*—We have developed a new graphical user interface for a home control device for a large industrial customer. In this industrial case study, we first present our approaches to requirements engineering and to software architecture; we also describe the given hardware platform. Then we make two contributions. Our first contribution is to provide a specific example of a real-world project in which a Twin Peaks-compliant approach to software development has been used, and to describe and discuss three examples of interplay between requirements and software architecture decisions. Our second contribution is to propose the hardware platform as a third Twin Peaks element that must be given attention in projects such as the one described in this paper. Specifically, we discuss how the presence of severe hardware constraints exacerbates making trade-offs between requirements and architecture.**

I. Introduction

We, the authors of this paper, work for the Danish software company Mjølner Informatics A/S (Mjølner), which develops custom-made software solutions for Danish and international customers, both in the private and the public sector. Mjølner has expertise in development of a broad range of system types. For an industrial customer, we have recently developed a new graphical user interface for a home control device.

Our customer is a large company, which sells its products all over the world, has more than 10,000 employees and a yearly turnover of billions of euros. The product that we are contributing to is a strategic product, which is seen as very important to strengthen our customer's market position in the near future. The product is a home control that can be used to control various things in houses and apartments. For confidentiality reasons, we do not have permission to mention the name of our customer or to describe the product in detail.

The purpose of the project was to analyze the needs of the users, design a new graphical user interface (UI) that was aligned with the users' needs, and implement the new UI. The customer's vision for the UI was "to look and feel as good as the iPhone". This meant a UI with consistent interaction design, beautiful graphics, smooth scrolling, responsive touch interaction and anti-aliased text. Figure 1 sketches the main screen of the UI (anonymized).

In the UI, the user can browse a coverflow containing various products that can be controlled, e.g., as it is known

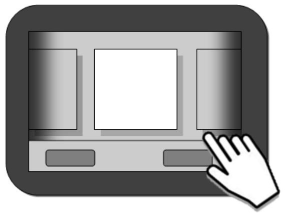


Figure 1. Coverflow

from presentation of music albums in iTunes. Additionally, a wide range of advanced features such as automatic programs to operate products, and grouping and arrangement of products to fit the user's home, are supported. The new graphical UI has approximately 200 different screens.

The project was finished in the spring 2012 and had been running for approximately two years. The project team size varied between 5 and 10 developers, and the total time consumption has been 18,000 hours. The hours are roughly distributed with 50% used for implementation, and 50% used for other activities including analysis and design. 100,000 lines of code have been written. In the first year, focus was on requirements elicitation and user experience in the form of analysis and design of various proposals for the style of the UI. In parallel, certain overall architectural decisions were made, such as the choice of a specific graphical library, developed in-house by Mjølner, to be used.

Requirements can be classified in the sense of [1] and the activities carried out the first year identified the goal-level requirements and yielded a reasonable stable requirements specification that also captured the domain-level and the product-level requirements. However, the design-level requirements have been volatile (which is quite natural). In the rest of this paper, when we say "requirement", by default we mean design-level requirement.

In the last year of the project, focus has been on implementation and realization of the UI. We have used an iterative, Scrum-like approach to development, with iterations of length approximately 4-5 weeks, and we have detailed, expanded, clarified, and prioritized requirements continually. There has been a very close coordination between the project's user experience designer - in this context, this is the same as the project's requirements engineer and will be referred to as such in this paper - and the project's architect. The choices made by the former have continually been discussed with and validated by the latter, and vice versa.

While the overall hardware specifications for the new home control were known from the beginning, the hardware platform was under development by another team at our customer and samples on which we could run the software were not available until the last months of the project's development.

Nusebeih's Twin Peaks approach [2] consists of high-level recommendations to software engineering in which requirements and software architecture are developed separately but concurrently. In the literature we have not seen many applications of Twin Peaks in practice. Therefore our first contribution is to provide a specific case study, in which Twin Peaks is discussed in relation to a real-world project.

In the development of the home control UI, we have experienced how constraints made by the hardware platform had substantial impact on this relationship, causing architectural challenges late in the project and thereby underlining the importance of a mutual understanding between the requirements and architecture domains. Therefore our second contribution is to explicitly propose the hardware platform as a third Twin Peaks element that must be given attention in projects such as the one described in this paper. We describe how requirements, architecture and hardware constraints have been dealt with simultaneously, and how the choices made about one of these have influenced the others.

II. Requirements Engineering

In eliciting requirements, a user experience (UX) approach was used. A primary goal was to gather information about the product's users and their ways of using the product (our customer has had a similar product on the market for years). Therefore, we started by conducting field studies and focus groups in two of the product's main market countries. We analyzed the domain, the competitors and conducted user tests of the existing product. All the research information gathered in the user research activities was scrutinized, discussed with the customer and consolidated.

Among the most important results from this process was the definition of personas that represent key user profiles, whose wishes should have proper priority in the design of the new UI. In this process, we also discussed and identified the goal-level requirements. Along with the personas, a list of the main usage scenarios for the home control was created in order to focus the subsequently developed wireframes and prototypes on the main usage and user needs.

Some of the wireframes were further elaborated into graphical versions. This being a high-end consumer product to be placed in people's homes, the graphical design and overall appearance was of very high importance. Different graphical styles were explored, still without knowing exactly what the limitations of the hardware platform would be.

It was at this point that the first discussions about the interplay between requirements engineering and software architecture started, the obvious goal being that the most evident software and hardware limitations could be taken into account before proceeding with and ultimately presenting for the customer interactions styles and graphical layouts that would be impossible to realize.

During the entire project, the requirements engineer had his desk in the same room as the software architect and the developers. This arrangement made it straightforward continually and on a daily basis to clarify mutual dependencies between architecture and requirements/user experience.

The UX approach was supplemented by a more traditional requirements process, where a comprehensive requirement specification was written - this specification focused on domain-level and product-level requirements, and, unlike the UX deliveries, did not say anything about the design of the UI, only its functionalities; see [3] for more details.

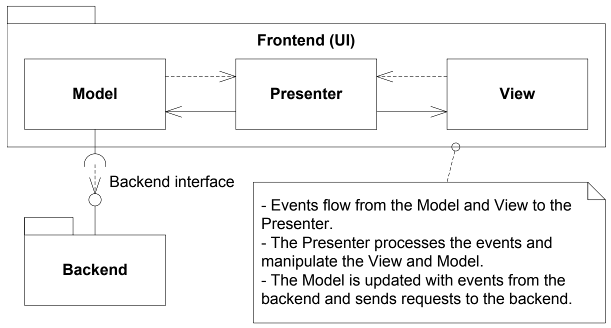
III. Software Architecture

There were three major characteristics of the project that greatly influenced the software architecture.

First the software development was done by two physically separated teams, with our team being responsible for the graphical UI, the frontend, and another team, consisting of software developers from the customer, responsible for the rest of the system, referred to as the backend in this article. This called for a very clear separation of these two components in order to facilitate parallel development. This was achieved by defining a backend interface which was essentially a set of structs communicated between the components asynchronously. Through this interface, the frontend can query product information and various configuration data for displaying in the UI. Similarly, the operation and management of products in the home control is dispatched to the backend through this interface.

Second, we knew the hardware platform was not going to be available until late in the project, necessitating a hardware abstraction layer and the implementation of a PC-based simulator. We were probably going to implement the simulator regardless, since our experience shows such a simulator proves a valuable tool in many aspects of the product cycle including test, rapid prototyping of features, input to the user manual and training of support personnel. But since the backend implementation done by the other team was target platform only, we had to simulate parts of the backend software as well. This provided the additional benefits of easy loading of various test data, easier debugging of the messages flowing across the backend interface, but perhaps most importantly it reduced the dependency on the development schedule of the other team. Our own development schedule was already to some extent dictated by the requirements engineering process, which did not necessarily align with the optimal schedule for implementing the backend.

Last, we wanted a UI architecture where the logic, view and data model were separated and chose a Model-View-Presenter (MVP) approach inspired by Presenter First [4].



Frontend (UI)
Model
Presenter
View
Backend interface
Backend
- Events flow from the Model and View to the Presenter.
- The Presenter processes the events and manipulate the View and Model.
- The Model is updated with events from the backend and sends requests to the backend.

Figure 2. Software Architecture—Model View Presenter

The architecture is illustrated in Figure2. Each separate area of functionality in the UI would be handled by a Presenter-View pair, thereby partitioning the code according to the various usage scenarios available in the home control. Only one Presenter-View pair would be active at a time, depending on what the user was currently doing.

The data model would act as a cache for certain backend information needed across all presenter-view pairs as well as filtering backend interface messages and dispatching relevant events to the active Presenter. The Presenter would then in turn populate the View according to the current Presenter state. Likewise, user input events such as touch or swipes would be processed by the presenter, which would then dispatch the appropriate commands to the backend. We assumed that by doing this, we could implement and test the Presenter, later create the final UI in the View and in this way implement Presenters for Views that were still in the specification state by the requirements engineer. However, as will be described later in this article, several obstacles arose during the project which turned out to impede this approach in practice.

IV. Hardware Platform

We were developing a consumer product for the mass market to be produced in large quantities. Therefore, the production price per unit was crucial and it was important that the hardware costs was low. Consequently, we had to accept that the hardware platform for our project would be low-end (eventhough the user experience should be high-end).

To give an idea of the challenges of the hardware platform, it can be compared to an iPhone 3GS (released in 2009). We were tasked with implementing high-end graphics on a processor running at less than 1/12th the speed and using around 1/1000th the memory of an iPhone 3GS. Also we had no dedicated graphics processor so the microcontroller itself needed to handle the display updates. The microprocessor, display and connected memory units all shared the same data bus. The target frame rate was 21 updates per second, meaning that we had less than 50 milliseconds to both render the screen and flush the frame buffer to the display. The theoretical throughput of the bus was only slightly higher than that, meaning it was important to ensure that no unnecessary pixel transfers took place.

Equally challenging was the amount of RAM available for holding the widgets, state information and various other data structures. This placed a severe restriction on architectural maneuverability since memory usage was a permanent concern.

V. Introduction to the Interplay Examples

With the requirements engineering, software architecture and hardware platform thus presented as background information, we proceed giving examples of their interplay in the next three sections. The examples have been specifically chosen to illustrate three different variations of the Twin Peaks relationship, where the first example resulted in major architectural changes, the second example yielded partially unsatisfied requirements and the third example resulted in a redesign. In all three examples, hardware constraints played a substantial role and limited the options available for ensuring coherence between requirements and architecture.

This is illustrated in Figure 3, which is an augmented Twin Peaks model that includes the hardware constraints as a third element. The hardware element is drawn as a box, instead of a peak, to hint that hardware properties are given, beyond the control of the software team. In contrast to requirements and software architecture, hardware is not elaborated and expanded during the software project in consideration in this paper (even though the software team's access to knowledge about and understanding of hardware properties may expand).

The annotated iterations of Figure 3 represent the iterations of Example 1 in the following section. Similar figures could be drawn to illustrate Examples 2 and 3, but space does not allow us to do so.

VI. Interplay Example 1: Presentation of Products in the UI Accommodated by a New List Design in the SW Architecture

We will use the implementation of lists as an example of how the architecture changed during iterative development.

We knew that the UI would have different types of lists. An example was a list used, when the user wanted to rename a product. The list should show all available products as list elements and let the user select one of them. This knowledge is show as (*A*) in Figure 3, which marks the beginning

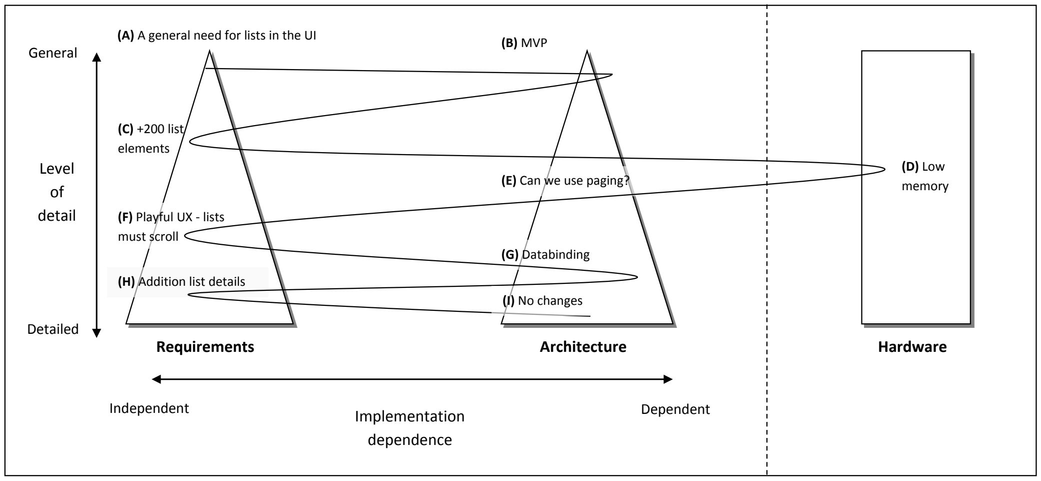


Figure 3. Twin Peaks and Hardware—and Interplay Example 1 illustrated

of the first iteration from requirements considerations to architecture considerations

At point (*B*), our initial architecture was based on the Presenter First variant of MVP, where the Model and View are isolated from each other and all UI events are processed by the Presenter. Our approach was to let the Presenter iterate over the products in the Model and for each product invoke a method on the View to add a list element to the View. The View had a virtual screen size much larger than the actual screen size and when the user scrolled the screen, the actual screen was updated to show a different part of the virtual screen. The Presenter was informed, when the user made a selection and an element index was used to identify the selected element.

In the second iteration (starting at point *C*), the list design became more detailed. The UI list was required to hold up to 200 products, since the home control was not only to be sold to private homes but also to hotels, schools and other larger institutions with many products. The user could scroll and use a swipe gesture to navigate to the product to be selected, to have a seamless interaction and to follow the de facto interaction standard. Our initial MVP implementation had no limitation on the number of list elements and thus supported the new UI design.

At point (*D*), it became obvious that the hardware constraints would force us to change direction. Following the architecture, we would iterate over the model and create 200 list elements, but we did not have enough RAM to allocate all the list elements.

One solution would be to keep the MVP architecture unchanged and add a paging mechanism, where only a limited number of list elements were shown at the same time (*E*). This would give a working product, but it would be less attractive, not meeting the goal-level requirement of creating a playful user experience with a seamless flow, because the user could no longer just swipe to the desired product.

This observation marked the starting point of the third iteration (*F*). The solution chosen was to modify the MVP architecture to include databinding for lists (*G*). The Presenter would provide a data collection to the View and the List widget in the View would bind to the data collection. The only list elements allocated was the number of elements, which could fit on a single screen. When the user scrolled or swiped the list, the list elements were reused by repositioning them, e.g. by taking the top element and placing it at the bottom of the list if the user scrolled upwards. It was now the responsibility of the View to modify the list elements to make them display the correct elements in the data collection.

In the fourth iteration (*H*), list design was continued with different colors for odd and even numbered list elements to make it visually less flat; it also gave a nice effect showing the movement when scrolling through a list. An icon for each list element was also included to indicate the type of product. The software architecture remained unchanged by this (*I*). Based on the data collection, the View would determine if a list element should be even or odd, determine the product type and modify the list elements correspondingly.

These iterations illustrate how a central part of the software architecture changed through iterations of the UI design and under the constraints given by the hardware. The initial MVP Presenter first architecture was modified to allow data binding between the View and a data collection from the model, an approach which gave us the look-and-feel required and at the same time minimized the memory usage to a level where it was possible to implement on the hardware.

VII. Interplay Example 2: Interleaved Scenarios Limited by the SW Architecture

As mentioned earlier, we had a Presenter-View pair for each logical grouping of functionality in the system. Because they each represented distinct user scenarios, there were not initially any reasons for the architecture to support advanced transitioning between Presenters such as seamlessly jumping to the middle of another scenario and back again. We did, however, keep all Presenters and Views loaded in RAM at all times should such a need arise. This consumed a lot of memory since the Presenters and Views contained a substantial amount of information, most notably the layout and widgets needed by the scenario.

As the implementation progressed, it became clear that we used too much memory. At this point, the design-level requirements were not finalized, in particular for some of the most complex scenarios. But there was no concrete evidence that advanced transitioning was going to be needed, so reworking the memory allocations such that only the active Presenter-View pair was loaded in RAM at any particular time was an obvious choice, especially since this would free a substantial amount of memory. In addition, even though this represented a somewhat large change in terms of architecture it could be implemented relatively easily, so we went ahead with this. As a consequence, when navigating away from a presenter, all its state information was lost so when the presenter was later reactivated, it would be in its initial default state.

When specifying the remaining scenarios, it became apparent that sometimes it was desirable to be able to jump between otherwise unrelated scenarios. Most prominently at the occurrence of errors or other exceptions causing a secondary scenario to be activated, after which the original scenario should be resumed. This would require the ability to switch to another presenter-view pair and afterwards restore the complete state of the previous screen (the previous Presenter-View pair). There were also examples where the initial state of a Presenter-View pair would vary situationally, and this was not supported by the architecture.

Faced with these requirements, it was obvious that some of the assumptions on which the architecture had been based were no longer true. Reverting the architecture to keep all state information in RAM would have helped alleviating these problems, but that was no longer an option, as we were already approaching the limit of RAM usage without it. In fact, this would have to be solved without any increase at all in RAM usage, which was not a trivial problem. In essence, we would need to rethink our core definitions of Presenters and Views, requiring substantial amounts of rework across the code base. It was simply too late in the project for that kind of change.

With that realization, the requirements engineer and the architect collaborated on reaching a practical compromise. Some of the offending requirements were not essential for the user experience of the product and could be removed or tweaked to fit the architecture. Conversely, some requirements were indeed important and would degrade the quality of the product if omitted. Some of these could be implemented by applying local changes (hacks) to the architecture and in those cases we chose to do so. However, there were also requirements which ideally should have been implemented but ended up being rejected due to limitations of the architecture and hardware platform.

VIII. Interplay Example 3: Smooth Transitions in the UI Redesigned with Respect to Hardware Limitations

We will now describe an example of how the hardware limitations influenced the design of the product.

For most of the project time we did not have any hardware. This meant, that we could not test the performance of different solutions, so instead we did a performance study on similar hardware and calculated how much of the screen we would be able to update and still have a proper refresh rate. We searched the Internet for guidelines for the update frequency of the screen and concluded that we needed to draw animations at a minimum of 21 frames per second to make them smooth. We wanted to use alpha blending for parts of the UI, because that could give a fading effect for instance in the coverflow and anti-aliased edges on icons and text, which would improve the look of the home control. Alpha blending uses more of the databus bandwidth and to take this into account, we created a spreadsheet that allowed us to calculate how much we could blend and move at the same time.

As described earlier, the main user interface is a coverflow where we show a number of product icons at the same time and highlight the one in the middle of the screen to show that this is the product, with which the user interacts.

The requirements engineer had suggested a spotlight effect, where an alpha blended gradient in both directions was used to create the spotlight. We used the spreadsheet calculations to get hardware constraints into consideration early, and the calculations showed that we would not be able to alpha blend on such a large area and still have an acceptable framerate and thus the swipe effect in the coverflow would stutter.

Instead, the software architect suggested an alternative solution. If only a gradient in one direction (vertical) was used and the UI was designed to scroll in the other direction (horizontal), the blending of background and icons could be pre-rendered and the only thing we needed to do in the product was to fade out the icons when they moved away from the center of the screen. The requirements engineer used the feedback from the architect and now had to find another way of living up to the goals and keeping a consistent interaction design. A few different solutions were considered to get the right effect at an acceptable frame rate and a good looking solution, which did not change the mental model of the coverflow's behavior and interaction. It was later implemented with the swipe effect running smoothly. The coverflow is an essential part of the home control, so we needed both to get the maximum effect out of the hardware, and be sure that it would work on the final hardware.

IX. Discussion of Three Main Twin Peaks Issues

Below we discuss three management concerns [5] that are said to be addressed well by the Twin Peaks approach [2].

"I'll know It When I See It": it is recognized that effective requirements engineering often demands that partially developed systems or prototypes are presented for various stakeholders during a project. We have to a high degree taken advantage of that, both in the initial UX approach with wireframes and graphical mockups and through development of the UI, where the simulator was used for presentations. We have been running an iterative project with frequent demonstrations and frequent releases to the customer. Had requirements and UI design been made without consideration of software architecture and hardware simultaneously, there would have been a risk that stakeholders would have experienced a number of disappointments, because the UI vision potentially would have been quite different from what was possible to implement given the hardware constraints. Conversely a fixed software architecture made with the given hardware constraints as a prime concern, but before before requirements and UI design, might have resulted in a solution that was less attractive and less user-friendly.

"Commercial off-the-shelf software": has not been used to build the new home control UI. Instead we have used a general-purpose graphical library, developed previously by our company, because it was seen as a suitable component to be applied in this context. The architectural decision to use this library was taken very early in the project, with only high-level requirements known, and this decision entailed a project risk, because it is an example of an architectural property that would have been very difficult, if not impossible, to change late in the project. In accordance with our expectation, it turned out that there was a good correspondence between what was needed in the UI, and what was made available by our graphical library.

"Rapid Change": with our development approach, we have been in a position, where it has been possible to deal with rapid change, as we have seen in the interplay examples. It would not have been possible with a non-iterative, waterfall-like approach, because we don't think that the many stakeholders' various viewpoints could have been captured and resulting design-level requirements specified before a software architecture was developed, and before implementation work began.

X. Conclusion

In this paper, we have considered an industrial project that has applied a development approach which is compliant with Twin Peaks. We have illustrated this in three interplay examples.

To sum up, in example 1, a number of iterations through requirements considerations, software architecture considerations, and hardware considerations resulted in a more detailed specification of the requirements, accommodated by extensive changes to the software architecture, and respecting the given hardware constraints. The original requirement was satisfied in full; it was an important requirement, and the solution demanded a major rewrite of many lines of code. In example 2, the requirements engineer had to accept that, because of limitations dictated by the software architecture and the hardware, it would not be feasible to satisfy the original requirement fully; compromises had to be made, and were made. Ultimately, this is an example of the risk of making architectural choices early, which ended up causing both partially unsatisfied requirements and deviations from the architecture. We believe that such situations are unavoidable in practice but it underlines the necessity of close collaboration between the requirements engineer and the architect. It also illustrates the benefits of these parties being able to understand the concepts and problem domains of the other to solve or even avoid these situations. In example 3, the requirements engineers' basic requirement got fulfilled, but in a different way than he had imagined from the beginning, and with no changes to the software architecture, but with a detailed analysis of the possibilities, given the hardware constraints.

We believe that the three interplay examples are of general interest to serve as a specific instance of Twin Peaks applied in practice. Moreover we think that the inclusion of hardware as a third Twin Peaks element will be relevant in many other projects, which are similar to ours.

References

[1] S. Lauesen, *Software Requirements - Styles and Techniques*. Addison Wesley, 2004.

[2] B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, pp. 115–117, Mar. 2001.

[3] J. B. Jørgensen, K. Nørskov, and N. M. Rubin, "Requirements engineering and stakeholder management in the development of a consumer product for a large industrial customer," in *RE'11*, 2011.

[4] M. Alles *et al.*, "Presenter first: Organizing complex GUI applications for test-driven development," in *AGILE 2006*. IEEE Computer Society, 2006, pp. 276–288.

[5] B. Boehm, "Requirements that handle IKIWISI, COTS, and rapid change," *Computer*, pp. 99–102, Jul. 2000.