Risky and rapid design spaces

Developing a digital ticket sales cooperative

Thesis to get the degree of cand.it

Risky and rapid design spaces: Developing a digital ticket sales cooperative

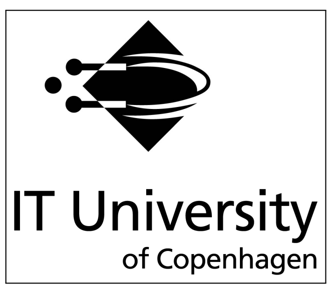
Expanding design space and reducing development complexities by means of rapid application development and co-realization

Benjamin Bach - benb@itu.dk

Supervisor:

Brit Ross Winthereik - brwi@itu.dk

June 3, 2013



IT University of Copenhagen
Digital Design and Communication
Technologies in Practice dept.

Software development is an activity of overall design with an experimental attitude

Peter Naur, 1985

Contents

Abstract **1**

1. Introduction **5**
1.1. Background 5
1.2. Problem statement 7
1.3. Contributions of the study 8
1.4. Overview 8

2. Theory **9**
2.1. Overview 9
2.2. Synthesis definition 10
2.2.1. Supporting definitions 11
2.3. Ethnography and software development 11
2.3.1. Cautious ethnomethodologists 13
2.3.2. Design and implementation: A continuum 14
2.3.3. Co-realization 17
2.4. Rapid Application Development 18
2.4.1. Technological acceleration 21
2.4.2. Changing organizational structures 22
2.5. Risk 23
2.5.1. Assumptions of agile development 24
2.5.2. Cost of user involvement 24
2.5.3. Lowering development costs 26
2.5.4. Complexities in software development 26
2.6. Entrepreneurship 28
2.7. Case and research methods 30
2.7.1. Overview of activities 31
2.7.2. Project establishment 31
2.7.3. Software development and sketching: A reflective process . . 33
2.7.4. Case study methods 35
2.7.5. Research methods 36
2.8. Summary 36

3. Outcome **39**
3.1. Case study: Organizations, environment, decisions 39
3.1.1. Project establishment 40

3.1.2. Participants . . . 43
3.1.3. Time line . . . 45
3.1.4. Nature of the Cooperation . . . 46
3.1.5. Risks . . . 48
3.1.6. Finding solutions to risks within the meeting space . . . 50
3.1.7. Ethnographic bias and IT facilitation . . . 53
3.1.8. Planning the project . . . 55
3.1.9. Moving towards design-in-use . . . 56
3.1.10. Arriving at design-in-use and rapid development . . . 60
3.1.11. Hidden decisions . . . 65
3.1.12. Testing the system in its real environment . . . 67
3.2. Software product . . . 68
3.2.1. Application functionality . . . 69
3.2.2. Technologies in use . . . 72
3.3. Summary . . . 75

4. Discussion and Analysis 77
4.1. Overview . . . 77
4.2. Case study practiced as research . . . 77
4.2.1. Research methods . . . 78
4.2.2. Case study practices . . . 78
4.2.3. Missed points of inquiry . . . 79
4.3. Results of the case study . . . 80
4.3.1. Contemplating alternatives . . . 82
4.3.2. Reducing costs and complexity . . . 83
4.3.3. Technological enablers . . . 84
4.3.4. Participation vs. non-participation . . . 85
4.4. Risky and rapid design spaces: Scope and applicability . . . 87
4.4.1. Design-in-use; Co-realization, PD, and agile methods . . . 87
4.4.2. Project establishment . . . 90
4.4.3. Assumptions and limitations of agile methods . . . 91
4.4.4. Prototypes and in-production artifacts: Convergence of design 91
4.4.5. Risks . . . 93
4.4.6. A general recommendation for future work . . . 94
4.5. Summary . . . 94

5. Conclusion 97
5.1. Findings . . . 97
5.2. Limitations and future work . . . 98
5.3. Recommendations . . . 98
5.4. Epilogue: A bit of normativity . . . 99

Acknowledgments 101

A. Communication and data **102**
A.1. Project plan 102
A.2. Project description (attached document) 102
A.3. Diary (not attached) 102

B. Application **103**
B.1. Data model, first revision (February 21) 104
B.2. Function list 105
B.2.1. Purchasing / frontend 105
B.2.2. Management / backend 106
B.2.3. Check-in 108

C. Misc **110**
C.1. Popularity of rapid application development 110
C.2. COCOMO estimate 110

Bibliography **111**

Abstract

I present a novel software development approach which is a synthesis of co-realization and rapid development methods. Analysis of this hybrid through a case study reveals several strengths when applied to smaller software projects. The method demonstrates cost efficiency, but implies a high degree of uncertainty regarding how to limit and facilitate the pre-analysis process and thus how to arrive at a construction phase. In the case study, the risks for the participants, implied by the experimental development approach, did not cause failure of the software project. Rather, the design space, created by rapid development and IT facilitation, meant that information for design decisions was easy to obtain, and even resulted in features outside of the initial project scope. Furthermore, my participation as IT facilitator at a strategic level of the organization meant that the new IT system and organizational practices were aligned.

Danish abstract

I følgende specialerapport præsenteres en eksperimenterende tilgang til softwareudvikling, som er en syntese af 'co-realisering' og 'rapid development' metoder. Analysen af denne hybrid gennem et case study viser flere styrker ved anvendelse i mindre softwareprojekter. Udviklingsmetoden demonstrerer omkostningseffektivitet, men den indbefatter usikkerhed vedrørerende bl.a. begrænsning af 'IT facilitering' af pre-analyseprocessen, hvilket vanskeliggjorde at nå frem til en konstruktionsfase. I case study'et lykkedes det at levere et fungerende stykke software, der opfyldte organisationens mål – på trods af de risici, som deltagerne løb ved projektet, som resultat af en ustruktureret, improviserende udviklingsmetode. Til gengæld gav udviklingsmetoden udslag i et rigt 'designrum', som hjalp til at informere design-beslutninger og skabte endda funktionaliteter, der lå ud over projektets oprindelige formål. Endvidere betød deltagelsen som IT facilitator på et strategisk niveau, at det nye it-system og organisationens praksisser blev samstemmet.

Prologue

Risks, unknowns, and opportunities

Often the unknown outcome of software design is perceived as a risk while it could also be explored as opportunities. We expect this from a good researcher, so why not from software developers and designers? By expecting a certain outcome of a software project, the space for creativity and design is reduced. Consequences of the complex nature of the social realities we seek to address by means of good design become risks of unpredictable outcomes. They are, however, also opportunities to do good and explore the nature of design and use.

The case study, subject of this thesis, started with a pretty cool idea of a platform that could help the local musical venues, but the project and its participants were tentative. As the project moved on, it proved possible to both create a usable product and at the same time find a hopefully unique synthesis within a very broad and inclusive theoretical background. After wrapping up the project and having seen it perform in its real context, I am relieved that it succeeded to an extend where the product became more than an academic experiment intended for immediate deletion. It promises to provide value to the participants and to be performing as an IT system in real organizations.

The achievement, however, is not sole credit to my own effort, but rather as it is commonly said in open source circles: *Standing on the shoulder of giants.* I owe credit to all the great FOSS (Free and Open Source Software) that has been the foundation of the project, though sadly not within the scope of the academic mission. FOSS is both subsidizing entrepreneurship all over the world and truly an enabler of rapid development, because it does not require economic, individual or organizational costs. FOSS is a tool to get stuff done and build upon other projects. It means that a developer acting to assist the needs of a particular organization can boldly brag to solve their needs with little effort and transcend the organization's needs by sharing their solutions with similar organizations on a global scale.

1. Introduction

In the present thesis, we explore a synthesis of co-realization and rapid development by means of a case study. What happens when the developer, acting as *IT facilitator,* makes use of rapid development methods? To shed light on this, the study uncovers examples from its case work and adds analysis, including the interplay between the IT facilitation, development process, real software as design artifacts, users, and organizational settings.

The study also intends to contribute to an understanding of the role of contemporary rapid development tools and techniques in software development methodology, and how they help reshape the role of IT practitioners. From the case study, we find that co-realization and rapid development can be especially relevant for smaller, risk-willing organizations, such as typical entrepreneurships.

1.1. Background

In the early 90s, Rapid Application Development (RAD, Martin (1991)) was coined and described as a formal software development methodology and has slowly died out in terms of research. Since then, the arena has been given to methodologies in the broad family of agile development. Once methodologies have been adapted to complex organizations, social interplays, and a wish to manage and control the outcome, they seldom seek to address a scenario like:

> *Software developer:* What do you need?
> *User:* I need a function which does 'this'.
> ...
> *Software developer:* I've implemented 'that'.
> *User:* Thank you, it does 'this' as I expected.

If we remove all the noise of communication channels, conflicts of interest, technological limits, economy, human resources, dynamic social contexts, etc., we are left with a very simple 'demand and develop' loop. Is the real world like that? No! But can we make a setting that at least puts a limit on the noise that clutters our otherwise simple and clean development model? That is what RAD was initially trying to achieve, claiming that the low complexity of the model would result in faster development and yield better results. Following the emergence of RAD, many

software development methodologies have based themselves upon a rapid and iterative feedback loop between users and developers. Even though these development methodologies all have qualities, their complexity remains a fundamental problem.

> *Methodologies are inherently complex; even methodologists who try to be scientific and professional in their approach to defining their processes too often end up giving too little or too much detail at the wrong level. (Ramsin and Paige, 2008)*

The inclusion of social and organizational reality is easily the main reason that methodologies become complex. This natural complexity of software environments creates uncertainty in the outcomes of any software project. Moreover, uncertainty is just as much a risk of failure to comply with the goals of the project or even do harm to organizations and users. To reduce complexity in software development, we have to let go of the inclusion of many formal and structured practices, ultimately something that may impact guarantees or predictability, but on the other hand leave room for improvisation and adaptivity.

Co-realization is a term and a set of methodological guidelines centered on the function of IT facilitation. As a method, Co-realization assumes a role of IT facilitator played by an individual who understands both the development of software, the design process, and aims to understand the user and organizational practices. Furthermore, co-realization suggests that we use the role as a means to address that of organizational change, i.e. that *the adoption of a system for a particular purpose often has wider consequences, which the IT facilitator may be called upon to play a role in identifying and formulating* (Hartswood et al., 2002). In its ideal form, co-realization's idea of IT facilitation should establish a direct connection between users and developers, lowering complexity and making way *to move from intermittent and over-formalized participation to a situation where informal interaction between users and IT professionals becomes a part of everyday experience* (Hartswood et al., 2002). By concentrating such 'facilitation' functions in a single entity and including a more direct approach to software development, i.e. rapid development, we can achieve a less complex development process.

1.2. Problem statement

Study and aim

Study A single developer, who is also the researcher, designs and implements a web application using a combination of rapid development and co-realization. The main outcome of the study is the process of building the application and the concurrent formation of an organization owning and using the application. This includes access for the developer and researcher to observe and influence decision making, social context, organizational structures, and daily work routines.

Aim The study should present and test a synthesis definition of co-realization and rapid application development. From the specific settings of the case study, the outcomes of placing a developer in the role of IT facilitator and researcher are analyzed and discussed. The analysis should also address the use and outcome of applied software for rapid development and how this software targets and shapes the development process in a real and concrete environment. From the outcome, we seek to understand if the approach of using rapid development and co-realization holds true with respect to complexity reduction and how it aligns software design and organizational practices.

The case and product

A common practice in the field of culture is to outsource ticket sales to private third-parties. In the case study, I address musical venues who would be interested in cooperatively owning and running such a system to lower their ticket fees and furthermore develop a system to target their own needs. Seen in a broader aspect, a private third-party's economic profit takes up economical space for cultural consumption, which creates a tension and a divergence of interests between the third-parties and the venues. In some cases, the fees of ticket sales were so high that it created a disproportion between the cost of the ticket itself and the fee. This motivates a need for an internalized ticket sales system that is owned by the venues, and furthermore would mean that they would be able to open up presales for less costly shows. By means of addressing this issue, the case study is also a real and beneficial project to its participants.

Scope

The study puts a great amount of work on the researcher acting as a developer, who is intended to conduct field notes for later study and at the same time participate in the project, playing a natural role. This means that the study can achieve neither to

perfection. The researcher is subject to the bias that follows from striving towards methodological guidelines. Furthermore, the case study does not include any holistic or external observation to its process. Naturally, I do not claim objectivity in the descriptions of the outcome.

When I address the value and success of the software, I seek to promote an understanding of basic software properties as compared to the goals of the project. It is not meant as an evaluation study, but a simple observation from the outcome of testing the software in a real environment.

1.3. Contributions of the study

The present study has the following primary contributions:

- A promising synthesis definition of co-realization and rapid development
- Outcomes of the case study: The synthesis put to work in a real environment
- An analysis and discussion of the outcome: How the synthesis performed as a design strategy

1.4. Overview

The thesis report is structured into the following main chapters:

2 Theory: The scientific work that has laid the ground for the case study and the analysis and discussion hereof. This is mainly targeted at an understanding of a synthesis definition (section 2.2) that can be related to a broad variety of works that are both separated in time and research fields.

3 Outcome: A presentation of the case study, including the most significant observations and descriptions of functionalities and technologies.

4 Discussion and Analysis: Firstly, I analyze the case study, relating it to concepts introduced in the background. The latter part of the chapter discusses how the case study and synthesis can be understood in a broader perspective with some focus also dedicated to alternative and related methodologies.

5 Conclusion: Finally, in the conclusion I briefly state my findings, their limitations, and a recommendation for future work along the same path.

2. Theory

2.1. Overview

The following chapter outlines the theoretical background of the three main areas of interest to the case study: Software development methodology, ethnomethodology, and an applicable scope. More specifically the focus of the study is Rapid Application Development (RAD, Martin (1991)) and co-realization (Hartswood et al., 2002), two closely related strategies for software development and design (Figure 2.1) and the application of a synthesis of both in a risk-willing setting, such as entrepreneurship.

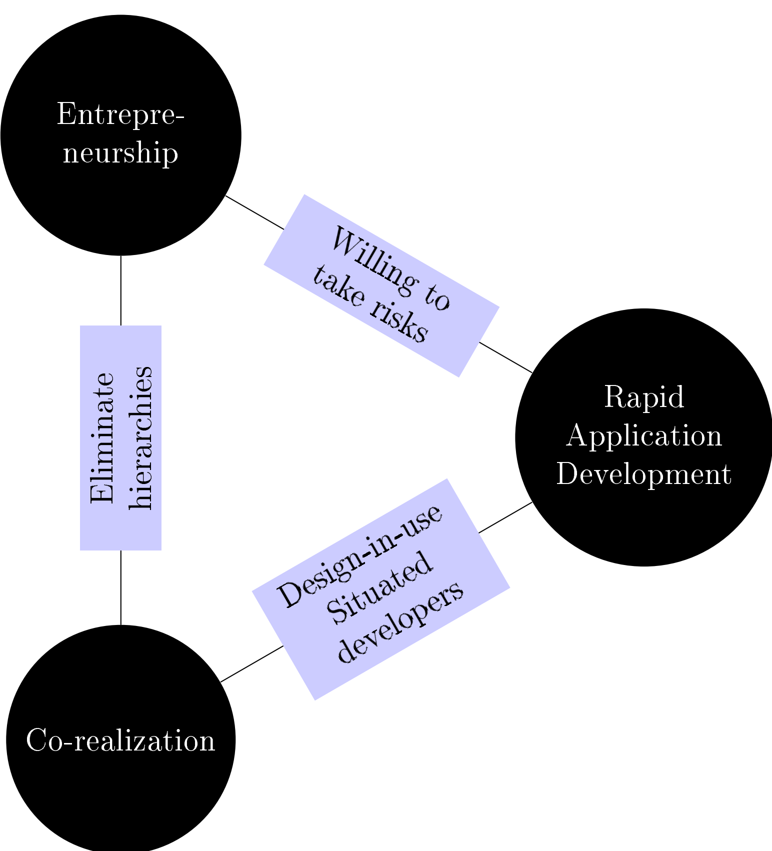


Figure 2.1.: The three most important theoretical areas and their relations which will be explored in this chapter.

The common problem and criticism which RAD and co-realization respond to will be explained later as the state of having a *bounded* design-process (see section 2.3.2), but each argue from separate perspectives. The origins of RAD are its criticism of top-down waterfall models as they are subject to environmental and temporal forces, making software projects based on these models unlikely to succeed. This is a case that RAD promises to resolve by introducing more efficient and effective design-in-use methods and by utilizing new software tools (note that this was 1991). Co-realization raises a criticism to prior ethnomethodologies by challenging the limits of what can be observed by ethnographers and what their role should be.

The overall aim of this chapter is to establish a relevant theoretical foundation for the case study, specifically since this case is conducted around the parallel development of an organization and an IT system. Furthermore, as the system is developed by a single developer with access to nearly all processes, I dedicate attention to theoretical works that are relevant through an appreciation of the sociotechnical nature of software development.

2.2. Synthesis definition

Drawing up the theory outlined in this chapter, we envision a synthesis framework of ideas from co-realization, design space, rapid application development, and the risks involved in these. We will refer to this synthesis as *risky and rapid design spaces* and give the following definition:

Risky and rapid design spaces: An IT facilitator (Hartswood et al., 2002) with knowledge of technological tools for rapid development, is situated in the context of an organization. A design space (Botero et al., 2010) is available to the IT facilitator and the relevant members of the organization. The IT facilitator supports participants with the use and design of new technology. Moreover, the IT facilitator guides the participants through organizational changes caused by new technology. The IT facilitator conducts meaningful ethnological observations to inform further design-in-use and development of the technology through iterations reoccurring to a degree such that design and deployment become a continuum. The practice of this should respect the risks involved by subjecting a software design methodology to a preference of day-to-day innovation over coordination by software requirements.

The framework definition is not meant to achieve precision in a way that limits its scope or turns it into a methodology. It intends for a broad analysis and further discussion of its outcomes. One example of an aspect that is not covered by the definition is the degree to which an organization can afford to be defined by its IT system. Another example is that the definition does not mention the nature of the risks involved – such examples are left for further discussion and analysis (see chapter 4).

2.2.1. Supporting definitions

In addition to the synthesis definition, the following definitions are given:

Rapid development: Rapid Application Development (RAD, Martin (1991)) refers to a specific application development *life cycle*. When referring to the study of and arguments of Martin (1991), I use the term RAD. But *rapid development* is seen as a broader methodology, simply a strategy or some set of methods to implement new design as fast as possible and push back the result to users for further iteration. No one in particular have defined the term, yet the term exists all over the Internet.

Co-realization: Although no singular definition of co-realization is at hand, I find the core understanding of the term to be that of the following. *Through being there, co-realization's aim is to achieve a situation where users and IT professionals can spontaneously shift their attention between the different phases of the system/artifact lifecycle, even to the extent that these cease to exist as distinct and separable activities. Co-realization means that IT professionals must help users realize their needs by playing the role of 'facilitator' in the broadest sense of the term.* (Hartswood et al., 2002)

IT Facilitator: (continued from above quote) *...this involves interalia acting as design consultant, developer, technician, trouble-shooter and handy-person. To support user-led innovation and design-in- use effectively, IT facilitators need to be able to shift their efforts smoothly between these various tasks as circumstances dictate.* (Hartswood et al., 2002)

Design space: *1) A design space is always actively co-constructed and explored by multiple actors through their social interactions with and through technologies and 2) the participating actors, resources, conditions and supporting strategies frame the design space available.* Botero et al. (2010). See also section 2.3.2. The term is pluralized to put emphasis on the different nature of design space and how they occur in dispersed in time and space.

2.3. Ethnography and software development

Ethnographers have long been describing technology's effect on organizations, social life, culture, management and vice versa. Ethnography tends to have exotic or unintuitive observations. In early eras, such exotic studies took place abroad, but would later find the exotic on the local scene (Neyland, 2007). An example of this is science and technology studies (STS) where the interplay and interactions between people, organizations, technology and society is the main focus. STS was made famous by works such as Latour (1987), uncovering the true origins of scientific work, not as pure science but a social construction. This has provided an understanding of ethnography not just as an observational research method to

examine and explain the world as is, but as an important tool to inform design decisions. Especially the field of Computer Supported Collaborative Work (CSCW) has adopted *ethnomethodology*, where many have argued that due to the complexity of such systems and their environments, ethnomethodology was a key tool to unlock the problems that arise from very complex social settings. A great example is an early 1992 study:

> *Ethnographic methods are introduced, and applied to identify the social organization of this cooperative work, and the use of instruments within it. On this basis, some metaphors for the electronic representation of current manual practices are presented, and their possibilities and limitations are discussed. (Hughes et al., 1992)*

Hughes et al. (1992) presents a study of ethnomethodology used to foster design ideas for a software project (an air control system) with respect to the social organization. The study is conducted in light of the achievements of HCI and cognitive psychology, but with an objective of unlocking perspectives of social organization, hinting that this is definitely not within reach and tradition of their field's (computer science) conventional HCI methods. Their study contained multiple ethnographic methods (as this case study also will – see section 2.7) with a non-interventionalist approach, observing only already existing technological 'artifacts'. To illustrate just how strictly they treated their observational concepts and how they saw them applicable, they envisioned that ethnographic results should be compiled such that some expert could extract and transform the data generated from observations:

> *These precepts cannot be simply 'applied to' or 'mapped onto' a domain in such a way as to yield, in mechanical fashion, a set of results. Rather – and in a manner which mirrors the overall relationship between ethnographic studies and systems design – they act as a resource, as a set of alerting mechanisms, and as a means of orientation. As a hermeneutic, interpretive or phenomenological approach, the aim is to produce a 'thick description' (Geertz (1973)) as a means to discovering the sociality of these activities in context, or situated actions. In the end – and somewhat elusively – the aim is to grasp 'what is really going on' in the course of a piece of work, 'what is really the problem' about doing it, and what instruments might be devised to help resolve these problems. (Hughes et al., 1992)*

Most notably, their work seeks to give input to the development of a new technology, but fails to mention how this technology will change its environment once deployed. They conclude that results gathered from ethnographic work are complex and hard to operationalize:

> *There is no formal modeling of functions or requirements, no analysis of data flow, no tabulation of viewpoints, no separation between function, implementation and interface. There are only descriptive, interpretive, incomplete quite subjective accounts of the and seemingly and the trou-*

> *bles involved in socially accomplishments organized action. (Hughes et al., 1992)*
>
> *While we suggest that the ethnographic approach offers an important additional resource in the artful and creative work of designing systems, the setting is far too complex for its findings to be simply predictive. (Hughes et al., 1992)*

Onwards, a whole set of methods in the field of Participatory Design (PD) (Kensing and Blomberg (1998), Tjørnhøj-Thomsen and Whyte (2008)) have been developed in a more hybrid fashion than *technomethodology* (Hughes et al., 1992), thus actually finding more specific ways to yield the results of ethnographic work in the design process. The schools of participatory design and contextual design (Beyer and Holtzblatt, 1999) can be seen as *hybrid disciplines*, or more pragmatic fields, in which the practical circumstances under which software has been developed are taken serious:

> *Compromise is essential in a design context it is argued and so efforts have been made to integrate a softer, more user-friendly version of ethnomethodological inquiry with other approaches to design, such as Participatory Design (...) These hybrid forms seek to cherry-pick and combine core disciplinary concepts and practices for the practical purposes of design. (Crabtree, 2004)*

The argument then follows that such hybrids have caused a segregation between those that create a system, and those that need or use a system, and that the design is an objective to *reach out* and *meet*.

2.3.1. Cautious ethnomethodologists

The many notions we find of how the limitations of scoping ethnography to the design process or finding ways of studying the true nature of technology design fits perfectly with the core of this thesis' objectives.

Work on the aforementioned air traffic control system (Hughes et al., 1992) revealed that the ethnographer should be informed by systems requirements, and that systems requirements should be re-iterated through debriefing meetings Bentley et al. (1992) through a notion of *ethnographically informed systems development*. Their work also had one very significant decision:

> *Because of the time constraints on our project, we could not carry out an ethnographic study then develop the software system; rather the ethnography and the systems development had to be concurrent activities. (Bentley et al., 1992)*

Even though earlier history's technological constraints and development costs could be considered a higher cost than modern day technology allows, the interplay between ethnographic findings and system design and development are found in these

early works. Similarly in Kensing (2003) on participatory methods, a principle is advocated under the notion of *Co-development*, outlining an interplay between development of IT, organizational development and development of user's qualification. Kensing (2003) quotes previous sociotechnological approaches as inspiration, discussing that in its essence, the parallel development of both social and technical systems was desirable, but had fostered a critique that Kensing (2003) responds to. The response is a principle stating that we should include the development of user's qualifications as an offset for the limitations introduced by the parallel development of social and technical systems. Yet, the principle only concerns informing the design process and generating the necessary tools, documents and measures for deploying the final product.

While the ethnographic tools from PD are indeed very useful, it is the theoretical principles, scope of design, and the outcome of PD that has fostered a criticism which the next section of this chapter is dedicated to outline. This includes both arguments from the PD literature, and from subsequent new theoretical models that aim at a combined understanding of *not only* social reality as an informer of technological design, but a combination of both technological design and organizational interventions, alterations and consequences of design once deployed.

2.3.2. Design and implementation: A continuum

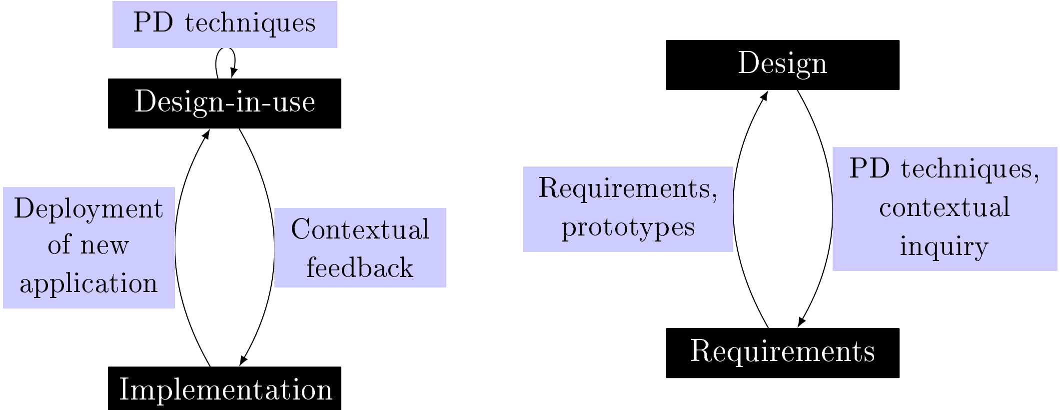


Figure 2.2.: Two main paradigms for iterative application of ethnographic methods: design→implementation and design→requirements.

Design has a very critical position in the application of ethnomethodology and the many different attempts of studying and unlocking the emergence of both organizations and technological systems. The reason for its critical position is, that if design→implementation is the (often sublimated) discourse, it means that systems must first be designed, then implemented and then changed through a new design process. This will be referred to as *bounded design*. It relies on a theoretical understanding that design is separable from implementation or deployment of some

bounded product, i.e. in retrospective to the design process, it would make sense to answer the question "What were we designing?", and the answer would then contain the product and all of its properties. However, contrary to whether this discourse is significant or not, we could repeat the design→implementation process enough times to make the question of the finished product unbounded by the mere fact that it is subject to constant re-iteration (see Figure 2.2) with no guarantees of theoretical saturation. The logic is so fundamental, it was already formulated by Peter Naur in 1965:

> *... we are dealing with three elements: problems, tools, and people, and the essence of the situation is the interplay between them. More specifically, we cannot, as people, think of a problem without at the same time implying some kind of tool. Stronger yet, when the tool changes the problem is not the same any more. On the other hand, our opinion about what is a proper, or desirable, tool surely depends on our understanding of the problem. In any case the problem and tool are nothing if they are not recognized as such by a person – that is where the people come in. (Naur, 1965)*

It is stated in literature on Participatory Design that iteration is desirable. In the description of MUST, Kensing (2003) suggests iterations as a principle to refine the results of ethnographic work, for instance by confronting users with what they have said versus what they have been observed to do by the ethnographer or *IT professional*. In this way, the principles that guide MUST acknowledges that as a new understanding of the requirements is reflected between the IT professional and users, design can the be consequently altered. This iteration could continue to refine design documents into eternity, but MUST suggests to move on to (rapid) prototyping after a couple of iterations. The belief that iterations are the solution is stated as such:

> *It would seem that a thorough design founded on data collection, analysis and prototypes and perhaps several iterations would be key to a high-quality IT system. Indeed, there are also several levels of what it means to collect data and inform the design-process which are discussed in detail in many works on PD. (Kensing, 2003)*

Without regarding whether ethnomethodology should be seen as a subjugated tool of software development methodologies or as an individual player, it is safe to say that software development methodologies have abandoned the thought of a design→implementation scenario through adaptation of agile processes (already from the beginning of the 90s!), while the field of ethnomethodology has been a little slower at renouncing the principle of software development through bounded, informed design.

Design space

Design does not, however, have to be understood as a monolithic figure, and is expanded in Botero et al. (2010). Instead of talking about *a* design of *a* system, they define *design space* as the space of possibilities for realizing a design, seeing participants in a continuum of consumption to active creation.

> *We argue that: 1) a design space is always actively co-constructed and explored by multiple actors through their social interactions with and through technologies and 2) the participating actors, resources, conditions and supporting strategies frame the design space available. (Botero et al., 2010)*

In their discussion of expanding the *conventional* design space, they also choose to define roles of designers more loosely *in order to avoid unproductive user-designer dichotomies* (Botero et al., 2010), a criticism that should be adapted by any facilitator of a design process. In presenting the foundation for the design space expansion, they admit to a technological reasoning, as the affordability to alter a design is established on technological tools. In other words the design space is actively sought and expanded as part of an organization and the technical tools that are present within. One example that the authors give is the use of wikis, which allows users to design their own information system, creating content with vast formatting options, categorizing, uploading etc. In the definition of design space, it means that all members of the organization are given the tools to evolve social practices, configure, integrate, remix, assemble components, use software modules and libraries and program and write modules freely, as this is the design space that a wiki offers.

This view is shared by Hartswood et al. (2002):

> *Co-realization means that IT professionals must help users realize their needs by playing the role of 'facilitator' in the broadest sense of the term. (Hartswood et al., 2002)*

What they say is largely unifiable with Botero et al. (2010)'s notion of expanding the *design space*, as the role of facilitator is required in the *broadest sense of the term* to help users realize their needs, however there are no direct concerns of *unproductive user-designer dichotomies.* Unifying the idea of design spaces and IT facilitation could mean that facilitators must make themselves redundant by providing a technological platform that can be evolved by the users themselves or to offer their assistance such that bounded design vanishes:

> *Through being there, co-realization's aim is to achieve a situation where users and IT professionals can spontaneously shift their attention between the different phases of the system/artifact lifecycle, even to the extent that these cease to exist as distinct and separable activities. (Hartswood et al., 2002)*

Both Botero et al. (2010) and Hartswood et al. (2002) share the notion of design-in-use, and they refer to the same idea: That design can be conceptualized by users and

observers in the process of using a system, and that designing in artificial contexts such as prototyping is inferior. This relates to design ideas both on a refinement level and on an innovative level:

> *[O]ne important aspect of design-in-use is recognizing and supporting the innovative processes of adoption and reconfiguration to ensure those functions meet the demands of professional adequacy. (Hartswood et al., 2002)*

2.3.3. Co-realization

Co-realization is a means to expand design space (see section 2.3.2), but this property holds more value than simply including more users and eliminating the notion of *bounded design*. It puts *emphasis on tightly coupled, 'lightweight' design, construction and evaluation techniques*. Co-realization does not abandon techniques from inspirational ethnomethodology but seeks a new setting, in which the role of the facilitator is key to build a better bridge between the results of ethnographic work and the implementation in the technological system. This is done by *the boundary between IT production and use* which addresses the situated consequence of having IT developers performing their work in the same context as the users, receiving direct feedback and being able to directly access the feedback necessary to both revolutionize and optimize aspects of the technological system. *Being there* is fundamental to development of the system, and *lightweight* is fundamental to the principal that guides design decisions, meaning that decisions are made with an intervention-like approach in which they are subject to change as rapidly as they have been deployed.

The emphasis put on the IT facilitator also means that a number of properties or skills have to be possessed by this individual. This is a debatable topic, which this study only investigates by discussing how IT facilitation ventured in the case study. Furthermore, there is no concrete set of methods defined for the IT facilitator, just an overall guideline:

> *Not only does the IT facilitator assemble the technologies, the technologies are the main focus of his daily activity, and a focus for his observations of and interactions with [the organization's] team members. Furthermore, the IT facilitator will seek to keep up with relevant technological developments. So, while the technologies are a constant factor in the life of the IT facilitator, this is not necessarily the situation for [the organization's] team members. (Hartswood et al., 2002)*

Foci of users and the IT facilitator are different, and thus the critique from Botero et al. (2010) remains that while team members from the organization are concerned with having the IT system solve their objectives, the IT facilitator may be concerned with the costs and possibilities of satisfying those objectives. But part of this discrepancy is addressed by having users develop and express their understand-

ing of the IT system, a principle resembling Kensing (2003)'s *development of user's qualifications*:

> *This focus enables the generation of a corpus of understanding comprising each member's specific experience with using the system that can feed into various forms of documentation, crib sheets, and advice to users in a dynamic and incremental way. (Hartswood et al., 2002)*

Furthermore, the idea of having users reflect on the IT system also means that they are able to share knowledge on how to alter and configure an IT system, which exemplifies an expansion of their design space. One of the case studies of Hartswood et al. (2002) also illustrates how the co-realization of an IT system does not only relate to design and changes to the IT system, but also touches upon the scope of use and consequences that the IT system has on the organizational system: By *being there*, the IT facilitator is able to give feedback to the users about their usage of the system, and to make suggestions to organizational changes that may both be an intended use of the IT system or a reconfiguration or redesign of the organization to the benefit of organizational objectives.

As this study seeks to give space to an organizational establishment all the way from scratch, the synergy between an IT system and organizational system is essential and the theoretical foundation laid out by Botero et al. (2010) and Hartswood et al. (2002) are fundamental to describing the case and illustrating their work.

2.4. Rapid Application Development

To the top-down waterfall models or requirements modeling of software development which were dominant until the 90s, the key criticism has pertained requirements modeling as subject to change by environmental and temporal forces and thus unlikely to succeed. This view was not only held by the founders of RAD, but cheered on as agile software development methods started to emerge in the mid 90s (Wikipedia, 2013a) and to this day, the adoption of agile methods is reported continued growth (VersionOne.com, 2012). RAD entered the stage as an ancestor of agile methods with an intention to redeem *higher speed, better quality and lower costs* (Figure 2.3), and indeed in spite of different views of whether RAD is a set of tools for rapid prototyping (Computer Assisted Software Engineering, CASE) or a methodology, the impact compared to previous methodologies is rated in *order of magnitudes* (Agarwal et al., 2000).

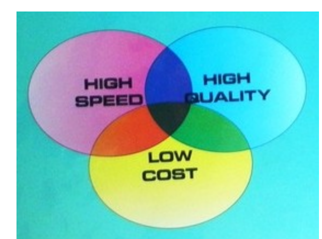


Figure 2.3.: Rapid Application Development, 1991. The book cover of Martin (1991).

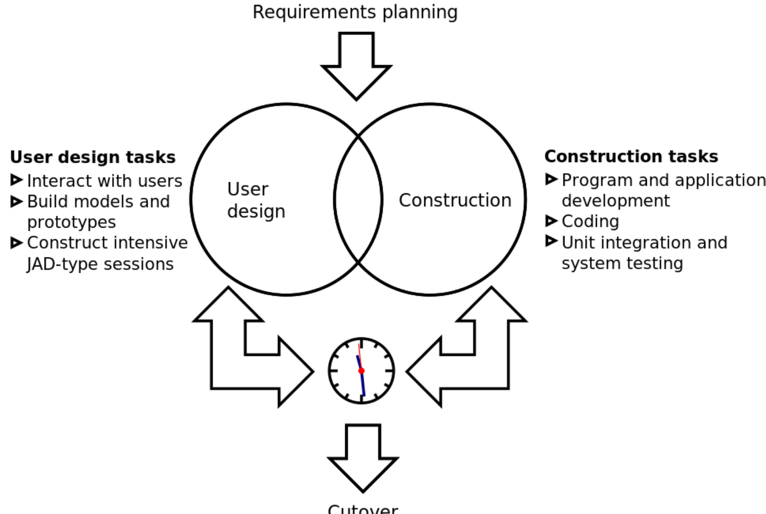


Figure 2.4.: RAD model in Shelly et al. (2011), the requirements phase and cutover phase have been shortened.

Since its emergence, RAD has faded as a software development methodology (Howard, 2002), but certain properties have remained although *no universal definition of RAD exists*, Agarwal et al. (2000):

Technological design tools: Rapid prototyping became heavily researched and also meant that the notion of "rapid development" was adopted by many software projects that did not address RAD itself (Django Software Foundation (2013), Wirdemann and Baustert (2008), McConnell (2010)).

Iterative development: Design is no longer seen as a one-step process, but takes place continuously. This is both the case in ethnomethodology, for instance Participatory Design, and agile development .

User Involvement: User participation is principle to design and development both in-context (RAD and co-realization) and through meetings, focus groups, and design panels (PD, SCRUM, and other agile methods).

In Shelly et al. (2011)'s model of RAD (see Figure 2.4), they show two overlapping and interchanging processes, namely user design and construction. These activities are shown to overlap, as the construction and user design may take place in the same context, and in the same sequence. The processes can be merged to the point where users are constructing the program themselves or directly instructing the developers while at work.

An illustration of the broad adaption of rapid development is found in Howard (2002), stating that *RAD is still a misunderstood concept*, and the introduction of McConnell (2010)[1] starts:

[1]Source is not an academic book, but is highly popular in its field.

> *To some people, rapid development consists of the application of a single pet tool method. To the hacker, rapid development is coding for 36 hours at a stretch. To the information engineer, it's RAD – a combination of CASE tools, intensive user involvement, and tight time boxes. To the vertical-market programmer, it's rapid prototyping (...). To the manager desperate to shorten a schedule, it's whatever practice was highlighted in the most recent issue of Business Week. (McConnell, 2010)*

Even though RAD has been proceeded by multitude of understandings of *rapid development* (which I use as a broader and hopefully less controversial term), we can recognize a number of the original ideas in newer concepts. Firstly, RAD relies on CASE tools, and while the initial tools that the literature was based upon have found replacements long ago, the same type of tools are still in use. CASE is supportive to the idea that an *expanding design space* should employ technological tools to support users in taking part in the design process. CASE tools were, however, created for system designers' needs, for instance providing aid for rapidly developing data models, user interfaces and prototypes. On top of that, it allowed a quicker transition from software models to actual software, for instance UI developing tools would directly feed the application building process, and software modeling tools (such as the UML scheme) would export their models to a native programming language. Only a very little or none of such software's output would result in visualizations intended for normal users.

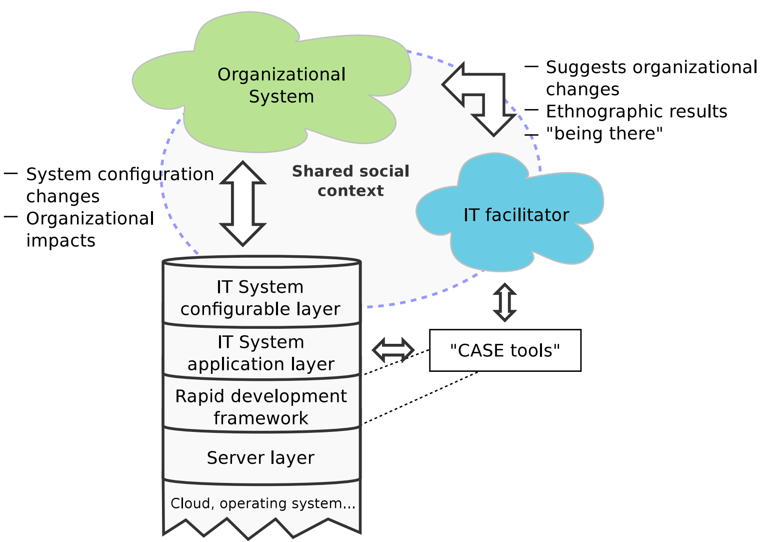


Figure 2.5.: A conceptual model of different layers of software.

An objection against its capability of expanding design space, is that CASE does not give users direct access to perform system development, but the lines can be more blurry in practice, as many platforms are making CASE tools redundant. For instance, a CMS (content management system) may allow its administrators (who can be a member of an organization with some training) to install plugins and extra

modules to gain increased functionality. This is given that a system developer has installed the CMS to allow for such behavior. In this example, software is altered and configured by users and administrator roles (users with special permissions), which is the case with most software, especially database-driven web applications (Cloud.com, 2011[2]).

Conceptually, we can use the layers introduced by Botero et al. (2010) to fit in the CASE tools from RAD, but with an understanding those tools as the possibilities that are supported by the underlying rapid development framework (see Figure 2.5). The abstraction introduced by the layers means that we can point to an *IT system configuration layer* and an *IT system application layer*. All layers together can be understood as the IT system, but parts of the system are more and less relevant to the organizational reality, for instance the server layer may introduce technical faults and economical costs, but it should be simple to the "design" of this layer what the desired criteria are.

> *Indeed, the tools and methodology are inextricably linked: the tools enable the methodology and circumscribe what is accomplished during a development project. (Agarwal et al., 2000)*

Or: The better we understand the reality of the tools of our methodology, the better we can define the methodology. The expansion of the design space and the co-realization effort are both grounded in the tools at hand, and how we choose to use them. In theory, those tools could entirely transcend the design capacities of the IT facilitator and the users configuring the IT system to achieve whatever organizational goals, but in reality we need to describe exactly what those tools are capable of in their context, because their limits and possibilities will define organizational changes.

2.4.1. Technological acceleration

A critical part of rapid development, is the time it takes to implement a change and complete a loop or an iteration of development→feedback. The faster the iterations, the more adjustments and features can be afforded. This is the obvious role of technology as an enabler of rapid development, and its relations to expanding design spaces and entrepreneurship.

> *[Data] Models can play an important role here, especially if tools exist for generating significant portions of the code from the models. (Turk et al., 2002)*

Data models are at the heart of software architecture and agile processes (Turk et al., 2002), a key example of how the choice and opportunities of technology serve as an

[2] A 2011 survey reveals that only 20% of 521 businesses using enterprise software did not have any plans to move into cloud computing. Furthermore, most use cases were related to CMS, document handling, ERP and CRM.

active enabler of the development process. The important point here is that data models serve on one hand as a development tool to efficiently build an application, but at the same time it becomes a manifestation of decisions and encapsulation of the problems and solutions that an IT system is addressing. Because there are tools to convert data models (especially object-oriented) to diagrams that can be subject to discussions with non-technical participants and conversely to programming code, the models become an efficient multi-dimensional tool, supporting both the design and development process.

Other important framework tools are the programming paradigms and automation of typical processes. Rapid frameworks are usually built to support object-oriented data models (Wikipedia, 2013b) and support typical manipulation tasks such as CRUD (Create/Update/Delete) scenarios, accelerating the development of typical user interfaces necessary. Other functions include "drag-and-drop" or wizard based development, automation of common tasks such as project deployment, easy integration of third-party applications / plugins, and encouragement of efficiency-focused programming principles, for instance *Don't Repeat Yourself* (DRY) and Model-View-Controller (MVC).

Since most applications are web-based, and this case study in particular, it's worth noting that both technologies for creating web applications (HTML, CSS, JavaScript) have improved in aspects of development efficiency, and many popular libraries have been created to support development processes. These include Twitter Bootstrap (Twitter, 2013), a *sleek, intuitive, and powerful front-end framework for faster and easier web development* and jQuery (jQuery, 2013) for efficiently manipulating and accessing elements on a web page:

> *It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript. (jQuery, 2013)*

The raison d'être of frameworks are that they simplify or provide functionality that is common in application development, and furthermore they often fuse performance and design issues guided by best practice discussions. But frameworks differentiate on aims and functionality, and arguably also on to which degree they are well-designed. They also require learning. Therefore, the technological acceleration of rapid development in a software project is dependent on the developer or IT facilitators familiarity with particular frameworks.

2.4.2. Changing organizational structures

One of the most important arguments pro RAD and other agile methodologies concern the changing settings of a software project. This was how RAD was inspired, to solve a problem of how demands, social contexts and project specifications would

change before the original project specifications could be implemented (Shelly et al., 2011, see Figure 2.6). This led to criticism, arguing that RAD is actually "Rough And Dirty" (Howard, 2002), but we need to understand that software delivery time and quality are not opposite sizes, but rather understand that RAD addresses actual issues that arise do to bounded design (see section 2.3.2):

> *What RAD tries to avoid is the bureaucratic nature of current quality control and assurance practice. (Howard, 2002)*

But while understanding RAD as a response to lack of coherence between organizational goals and some waterfall model, we should also keep in mind, that RAD is still a software design methodology, ie. its aim is to create technology that aligns with organizational strategies.

2.5. Risk

In this section, we shall look at risks relevant to rapid development and ethnomethodological tools (see also section 2.7). Risks can be seen as a central aspect when using a sparsely managed approach as rapid development, where the project is established with little assumption or assessment of costs and outcomes due to the cut-away of requirements and project management. Alternative development methodologies are prone to risks of their own, which is saved for the discussion of the likely benefits of *risky and rapid design spaces*. Other risks are general to software development, especially in cases of complex organizational structures and collaborative features.

Software risk can be defined as the cases of outcome in which software is not successful (Addison and Vallabh, 2002), i.e. the chances of software failure. Studies of such failure from 80's (Grudin, 1988) and up to 00's (Addison and Vallabh, 2002) agree that risks of failure can be related to the following key areas: Extra work and lack of benefit, favoring of management needs, and problematic/impossible evaluation and analysis. It can be argued that rapid methodology is either more or less risky than its alternatives.

When risks are taken deliberately, the action also relies on some criteria of success. However, if the requirements are loosely defined, the risks also become more abstract.

Figure 2.6.: Dilbert

In Addison and Vallabh (2002)'s study, risks are measured against failure to comply with requirements, but even though this measure is weakened by having little or no explicit requirements, the risk of failure can still be grasped. Even without explicit requirements and a day-to-day take on design, the aims and objectives of a project may persist. Risks are still that the project fails to meet aims and objectives, even if they are changed underway.

Software development may be risky in its nature, and rapid development and/or participatory methodologies may introduce more risks and relieve others, but the scenario fits well in that of entrepreneurship, which shall be elaborated in section 2.6.

2.5.1. Assumptions of agile development

There are a number of risks which are directly related to using rapid application development seen as a type of agile development method. This is caused by the assumptions that agile development makes, some of which are hard to avoid without breaking with the agile scheme:

> *The unavailability of customers was frequently the highest risk identified.*
> *(Turk et al., 2005)*

The above is a risk well-worth considering before choosing a rapid development method: If the users or customers are not available to give feedback, a defining part of the design process is removed. As described in Turk et al. (2005), agile development makes a number of assumptions (see Table 2.1). These assumptions apply to rapid development, and will be given consideration when analyzing the outcome of the case study.

2.5.2. Cost of user involvement

It is not always the case that user participation in the RAD cycle (see Figure 2.4) yields constructive results. For instance, users may repeat already known views, present views that defeat the purpose of the project or ask for features that the IT facilitator cannot grand. Or the results from user participation could potentially be derived more efficiently with other methods.

> *Questions must be asked about: the aims of participation, the forms of participation that are being advocated, and the skills and strategies required of practitioners. Dearden and Rizvi (2008)*

To rapid development, the degree of participatory design is not, however, a fixed quantity. Neither is it to co-realization (Hartswood et al., 2002) which does not rule out the option of non-participatory design decisions. This leaves a space for the IT facilitator and management to still make changes and priorities within the cycle of design and construction (see section 4.3.4). The discussion of participation is not,

Assumption	
Visibility	Project visibility can be achieved solely through delivery of working code.
Iteration	A project can always be structured into short fixed-time iterations.
Customer interaction	Customer teams are available for frequent interaction when needed by developers.
Team communication	Developers are located in time and place such that they are able to have frequent, intensive communication with each other.
Face-to-Face	Face-to-face interaction is the most productive method of communicating with customers and among developers.
Documentation	Developing extensive (relatively complete) and consistent documentation and software models is counter-productive.

Table 2.1.: Turk et al. (2005)'s list of common assumptions in agile development, which extend to our understanding of rapid application development. They list a total of 14 assumptions, these are just some of them.

however, all or nothing, but takes a direction of solving the issue through mediation or participatory figures that are not necessary users or designers exclusively:

> *While there is discussion in the CSCW literature about how to construct productive relations between those doing work analysis and those designing CSCW systems, there is no explicit commitment to direct user participation in design. In fact, some have argued that it is too costly and logistically problematic to have users directly involved in design (see Bentley et al., 1992; Hughes et al., 1993). As an alternative, social scientists and others may act as user surrogates or representatives in design discussions. In PD direct user participation in design is one of the hallmarks of the field although as Mambrey et al. (1998) suggest, sometimes it is valuable to augment direct user participation with what they refer to as user advocacy. (Kensing and Blomberg, 1998)*

In the sense that *surrogates or representatives in design discussions* are the same as Hartswood et al. (2002)'s IT facilitator except for capabilities of understanding IT development, the argument in Kensing and Blomberg (1998) supports that an IT facilitator can mend or avoid being *too costly and logistically problematic to have users directly involved in design.*

2.5.3. Lowering development costs

As argued in section section 2.7.3, more and more rapid development frameworks are showing up, increasing the IT facilitators and developer's ability to meet new design ideas and reiterate. Supposing that software development has become cheaper or the support of an expanded design space has deliberately been built into a development process, experimenting with different ideas and alternatives or changing direction to avoid some perceived failure becomes more likely. An example of a middleware targeted at making rapid development of a user specified application is given in Weaver et al. (2012):

> *This paper presents several case studies of rapidly implemented, audience-specific applications (...) By tailoring each application to the audience and the task at hand, the required learning curve for new users was greatly reduced. Each case study used OpenStudio, the U.S. Department of Energy's middleware software development kit. (...) OpenStudio dramatically reduced the effort and risk required to develop the building energy modeling applications described in the four case studies. (Weaver et al., 2012)*

2.5.4. Complexities in software development

Complexity can be seen as a negative term: Complexity understood as a multitude of factors at play can make decisions become clouded. Complexities can pose a threat to such a degree that some methodologies have even resorted to *complexity management* (Ramsin and Paige, 2008).

..and a positive term: Since the true nature of the organizational and social structures that an IT system should support is complex, then the uncovering of these complexities should not cause failure, but rather be an opportunity to secure the project's success.

In order to frame the discussion of complexities and reducing them, we should make clear what we mean by *reducing complexity*. Naturally, we mean reducing those complexities that cloud design decisions and add to unnecessary articulation work and not the complexities that should be accounted for and play part in the design.

Some risks due to complexity are avoided by choosing co-realization or rapid development. For instance, the risk of requirements not truly reflecting a complex environment or requirements becoming too complex to be managed, thus defeating the purpose of using requirements specifications.

> *Software development is difficult to manage if the requirements cannot be fully and accurately defined at the beginning of a project or if the requirements are so complex that changes are inevitable during development. (McQuaid, 2001)*

Researchers of CSCW have stated the need to *incorporate social complexity into CSCW design* (Procter and Williams, 1996), but further explaining that this complexity lead to a new *complexity of the social interaction between end-users, systems designers and implementors*, which we will refer to as complexity in the design space (see section 2.3.2):

> *In CSCW, the problem is not the complexity of the technology (though it may well be complex), but complexity of the social interaction between end-users, systems designers and implementors (Greenbaum & Kyng 1991). And one important reason for this complexity is that end-users and designers don't inhabit the the same environment and share a common practice. (Procter and Williams, 1996)*

The parallel, I wish to draw here, is once again to co-realization, namely that since CSCW researchers find the problem of complexity within the design space, they are advocating what can be found Hartswood et al. (2002), stating that *one important reason for this complexity is that end-users and designers don't inhabit the the same environment* (Procter and Williams, 1996).

On a more universal level, methodologies and the discussion of when, why, and how lead to all sorts of self-inflicted complexities in the research field (see Ramsin and Paige, 2008). The problems caused by this are, to a practitioner, that the choice of methodology becomes difficult. To this case study in hand, it means that framing the analysis becomes deadly critical.

> *Methodologies are inherently complex; even methodologists who try to be scientific and professional in their approach to defining their processes too often end up giving too little or too much detail at the wrong level. (Ramsin and Paige, 2008)*

Methodology complexities are very much also a reality in entrepreneurship research, which we shall look at in the next section (section 2.6), and also caused by the arguable necessity of including constructionist discourses:

> *Much of Entrepreneurship Theory development has assumed positivist methodologies, but many of the models defied rigorous empirical testing due to their complexity (for example Bygrave, 1995: 5). This prompted a number of scholars to consider just how Entrepreneurship Theory might be redirected. (...) Sociological approaches focus on structure and 'agentic' aspects of entrepreneurial behavior; this has led to consideration of how signals from the environment may influence entrepreneurs' actions (...) (Chell, 2007)*

We leave the discussion of complexities in entrepreneurship theory, but it would seem that an argument could be made on the same grounds as co-realization, for instance that an entrepreneurship should design itself with the context that it wishes to operate, and that's not very far from grasp, if it is a software entrepreneurship.

2.6. Entrepreneurship

Introducing the generally business-minded research field of entrepreneurship is a two-sided measure. On one hand, there are lessons to be included from this topic, but more importantly, it helps to define the scope of *relevance* to the outcome and discussion of rapid application development, co-realization, and expanding design spaces towards the final synthesis (see section 2.2).

Entrepreneurship – similar or identical to the likes of small-businesses, startups, and software ventures – is a popular term, and it's argued by *innovation economics* that entrepreneurship is a core part of the economic model. Over time, tools necessary to create new inventions, products and business models have become cheaper and more accessible and likewise, there has been an emergence of an organizational structure targeted at new opportunities. Some see it as the second era, the one following the dot-com burst. The Internet is hard not to mention, especially as it is key to global markets, work-forces, knowledge, and software building blocks being at the hands of a software entrepreneurship. Zutshi et al. (2006) have built upon preceding theory on entrepreneurship with an objective of finding attributes to characterize *e-entrepreneurship*:

> *...E-entrepreneurship [is defined] as a concept which principally uses the Internet to strategically and competitively achieve vision, business goals, and objectives. (Zutshi et al., 2006)*

The case study has many characteristics of an entrepreneurship, and especially of an e-entrepreneurship. First of all, it is an online ticket sales system and thus requires the Internet to exist and function, but moreover, the idea of having presales is a core business goal and objective to the venues (see chapter 3). During the study, many critical functions of the participating organizations turned out to be adjacent to the ticket vending system, putting both risk and opportunity at stake.

Many of the assumptions and positive findings in the settings of RAD and co-realization studies hold true in entrepreneurships. Shelly et al. (2011) states that RAD is indeed suitable for those cases when new business functions are developed:

> *Because it is a dynamic, user-driven process, RAD is especially valuable when a company needs an information system to support a new business function. (Shelly et al., 2011)*

The case study consists of a project that initially holds many unknowns and in a normal economical setting would need a startup investment. In actuality, the project participants have all risked their time (see section 3.1.5), and if the project had not been a research project, they would have had to risk an economical investment. In addition, the software product has been a highly open-minded process, rather than a precisely modeled plan. The risks that are inherent from the nature of software projects and more specifically the ones using rapid development are outlined in section 2.5. One of the reasons to specifically distinguish e-entrepreneurship is that of economical risks, ie. setup costs, are generally lower.

Entrepreneurship and organizational structures

For an organization to label itself as an entrepreneurship, or rather for someone to consider an organization as having entrepreneurship properties, there are a few normative results to draw from the literature. Properties which to an e-entrepreneurship could be fueled by the development methodology that they choose. One of them is the definition of *dynamic capabilities*, after a long a controversial discussion:

> *[T]he abilities to re-configure a firm's resources and routines in the manner envisioned and deemed appropriate by its principal decision maker(s). (Zahra et al., 2006)*

The view held by Zahra et al. (2006) is focused upon leadership and management, which could be disjoint from the ideas of co-realization and rapid development if it was to be understood as a methodology that weakens management control. However, the definition actually could be an approach towards such methodology, as *dynamic capabilities* asks for management and leadership to be capable of reconfiguration, e.g. to reconfigure from the changes that arise from a *dynamic* (rapid and participatory) development methodology.

> *...from these definitions, it can be inferred that successful entrepreneurs need to possess attributes such as vision, opportunity-seeking, leadership, and management skills. (Zutshi et al., 2006)*

Another argument strengthening the bond between rapid development and entrepreneurship organizations, is that of the adaption of the entrepreneurship to its environment and stakeholders. Because such organizations are required to adapt to survive, they should be able to *exploit serendipity*:

> *The exploitation of serendipity necessitates flexibility with regard to the start-ups' existing product or service concepts, strategies and business plans because in the serendipitous mode these are often re- and co-designed with newly encountered stakeholders. (Tahvanainen and Steinert, 2013)*

The rather normative discussion of how an organizational structure *should be* in order to successfully support entrepreneurship or innovation is not only relevant in an economical sense. If we were to suggest that co-realization and rapid development are essential to such environments, we should eventually account for what exactly we mean by such. Given this idea of organizational structures supportive of change and a specific practice of software development adds another layer of complexity to our understanding of social and technological interplay. Not only should organizations adapt and adapt to new IT systems, they should do the same to the development of them!

2.7. Case and research methods

The case study is very conventional in its overall structure: In the former sections of this chapter, we have explored a theoretical synthesis bond between co-realization and rapid development and now aim to explore its validity and add reality to our theory through analysis of the outcome (see for instance Eisenhardt (1989) on building theories from case studies and using theoretical constructs during case research). We use a few classic ethnomethods to inform software design while on the other hand also informing the case research (recall the double-role of the researcher). Explaining the methodology takes off with a definition of the study *as is*:

Case 1. In the study, a software developer initiates a ticket sales cooperative comprised of two different organizations, namely two music venues. The developer participates in organizational decisions, interviews stake-holders, keeps a diary, and conducts field observations. The software development takes place continuously.

The reason for seeing the methodology *as is*, is due to the nature of the case – the study does not afford any further observational resources, so I try to define the case methodology in a deliberate fashion with reference to item 1. As I've placed myself as the IT facilitator (see section 2.3.3), we should take note of the bias present. The outcome of my own participation is explained in section 3.1.7. Furthermore, I'm also a stakeholder in terms of academic pursuits, but this behavior can be seen as a convenient replacement for the fact that I'm not being paid or professionally affiliated as would normally have been the case for an IT facilitator, i.e. in terms of defending my development methods and not seeing my hard work go to waste, I would naturally like to see case project succeed.

In order to gather inputs, the role of IT facilitator uses a free range of ethnomethodology tools as deemed necessary and affordable. The tools are used in classic ethnography to gain the observer a more natural role, in which he can conduct fieldwork and make observations in natural surroundings. Many ethnographers argue that an active role is necessary to gain real insights, the most extreme being perhaps the field of *Militant Ethnography* (Juris (2007)) where an ethnographer adopts the same views and sympathy of the subject group. As I have no other similar participants subject of study, i.e. no other IT facilitators, it would be an exaggeration to hint to such theories as Militant Ethnography. My position can be seen to unlock access to the organization that I want to study, undertaking a naturalized role. But in the case of understanding the software design and development process, the goal is not as to reach an in-depth understanding, empathy, or even sympathy of *other* participants, rather it is to reflect upon and later reconstruct *my own* decisions. As such, the extreme degrees of sympathy becomes a given, as long as I can truly reflect upon my own actions in the development process.

The case study resembles to two levels of ethnographic practice: Embedded within the case's project itself, is the interest of gaining knowledge to guide the software

design. This also includes ethnomethodology! Since I'm both facilitating the design process *and* the succeeding study of the case itself, the embedded ethnographic observations become input to both the process of the case study itself, and the research.

Most of this section and its subsections are dedicated to theory on gathering ethnographical insights for the design and development process, except the latter (section 2.7.5) which touches upon the overall research methodology targeted at accounting for the case study.

Ethnomethodology of software development The specific selection of tools and methods is subject to the circumstances under which the project is carried out (see Sanders et al. (2010) for a discussion on selecting such tools). Even though rapid development may seek to direct design observations to materialize in application development as directly as possible, the developer does not necessarily enter a conflict with such discourse by choosing a participatory approach, seeing that participation is a consequence of co-realization. Rapid development can simply be seen as a factor when choosing tools and methods. The framework of Sanders et al. (2010) presents 4 criteria dimensions for the *purpose* of choosing tools and techniques for participatory development:

> *1) for probing participants, 2) for priming participants in order to immerse them in the domain of interest, 3) to get a better understanding of their current experience or, 4) the generation of ideas or design concepts for the future (Sanders et al., 2010)*

When rapid development becomes a factor for choosing tools and techniques, it directly relates to 3) and 4). Having a better idea of current experience is similar to performing observation of how users act in the realm of the RAD cycle. The generation of ideas or design concepts relates to the feedback from users during the development cycle. 1) and 2) are more indirect consequences, for instance priming participants would be necessary if they are not actively playing a part by providing feedback.

2.7.1. Overview of activities

Table 2.2 is an outline of activities that the case study has adopted. The theoretical reasons for choosing them will be elaborated in the succeeding sections. They are not a conceptual or principal set of steps, but circumstantial elements of this project.

2.7.2. Project establishment

Before any practical work can be carried out on a software development project, before the design can take place, however lightweight it may be, and before the range

Activity	Tools
Project establishment	Meetings negotiating aims and expectations, project plan, developing statutes
Application data model	Development, management meeting, object-relational diagram
Baseline application implementation	Development, management meeting, improvisation
Ticket purchasing module	Development, management meeting, real testing, improvisation
Administration module	Development, management meeting, real testing, 'being there'
Door check-in module	Development, field observations, key personnel interviews, development, real-life testing
Holistic re-iteration	Development, real-life testing, meetings

Table 2.2.: Main activities and tools, by "development" is meant software design programming.

of participants has been uncovered, an establishment process is inevitable, both in theory and practice. Elaborating the *project establishment* we need to uncover how detailed and structured it should be, and what tools in it should include. Kensing (2003)'s principles outline a number factors that should be considered in project establishment. It can be seen as encompassing a possibly binding negotiation of basic goals and constraints of the project for instance visions of how the software could be made and discussions of necessities (Kensing, 2003). It is suggested to create a *project charter* to manifest those decisions. To leave room for opportunities be explored later on in the process (see section 4.4.2), we need to locate a balance such that the establishment of the project does not confine itself too much, ie. that we do not make binding agreements that limit the scope of freedom to act upon later findings.

The project establishment can also be an opportunity for management, users, and IT professionals to brief each other on the organizational impacts that they are willing to accept, their choices of ethnographic tools, creation of business models, and negotiate the risks that they are willing to take.

Preceding processes of the project establishment may be intended or circumstantial, but via our understanding of existence and size of design space (see section 2.3.2). If the goal is to maximally extend the design space, then the implication may well be that we should allocate as little design decisions in the establishment process as possible.

Establishing a project is not solely about preparing a design phase. There are other issues that may need to be established or researched such as legal, economic, and ethical aspects.

The case study aims to establish the basic objectives and terms of a cooperative, partly the organization that would be contextual to the IT system, as well as the already existing organizations that were entering the cooperative, including their targets and policies. The formation of our cooperative has legal and economical consequences, and being initialized from an external player (me), it requires an initial effort of convincing stakeholders. We can refer to the *establishment of the entrepreneurship*, part of the entrepreneurial process. And even though we do not intend an in-depth analysis of the case's entrepreneurial process or business model, we refer to the consensus that entrepreneurship does not exclusively bind itself to exact economic goals, but values to an extend that there is a common agreement:

> *There does appear to be more of a consensus that 'opportunity recognition' is an entrepreneurial attribute (Gaglio, 1997, 2004; Hills, 1995; Kirzner, 1979, 1985) as is the goal-oriented behavior that may be summed up in the phrase the 'creation of something (of value)'. Chell (2007)*

Thus, when we talk of the establishment of a software entrepreneurship, our attention should be directed to the value that we are trying to create by use of the IT system in an organizational context where it is supposed to integrate with overall achievement of goals through a consensus of *opportunity recognition*.

2.7.3. Software development and sketching: A reflective process

In *Software Development as Reality Construction* (Floyd, 1992), two paradigms were made subject of criticism, namely software production and software engineering, in which the process of software engineering was viewed as a consequence of software production. Floyd viewed software as a construction of reality, and the development process was described with constructivist discourse. Her characterization sounded:

> *Software exhibits an extreme degree of complexity, this calling for equally complex construction processes. It consists of a uniform, abstract building material, is therefore plastic and, in principle, of unlimited revisability. It must be machine-processable, i.e. complete down to the last detail, consistent and formally free from error. It is not amendable to sensory perception and can therefore, in the last analysis, only be evaluated once in use. It creates social contexts for human actions, which are shaped by the technical properties of the product. (Floyd, 1992)*

Floyd (1992)'s illustration of the complexity and construction of software urges that we take careful note that the developer is constructing and expressing her version of reality, in some cases subject to an agreed upon design, but thus still expressing

a construction of something relative to its perception. What we should note in regards to this construction process is that non-decisions are also expressed, for instance because the developer is not aware of alternatives. In cognitive discourse, we could also add that subconscious decisions are also at play. It highlights the role of the IT facilitator, being a developer, having to convey a personal experience of reality to programmatic artifacts.

What is important to this study, is the many parts of an application code base that are never discussed, and parts which are never even used or tested because they refer to theoretical cases of reality that have been constructed and expressed by the developer. In this study, at least, with help from autoethnographic methods, we can uncover the examples and come to grasp the significance of these decisions seen as a part of the design, though often unmentioned. To put it in another way, what the developer decides to do without any prior design decision or even contradictory to design decisions, is not a feature of the development methodologies found in this study.

Another process of interest takes place while the developer is at work: Sketching. Works on design processes promote sketching and other means of expression through materialization or creation of new artifacts to further a design process. Sketching makes exploration and sharing of ideas easily affordable, and however more affordable rapid software creation becomes, the better access is gained to sketching. The sketching process can take place between the software developer and the artifacts that are given life through creation of new application structures, but it might as well be seen between the artifacts and the users, subject to how the artifacts are communicated. A rapidly implemented artifact may both be a true function of the application and a sketch, however suits the view (see section 4.4.4). The process of sketching is highly related to Kensing (2003)'s principle of anchoring visions, however they assume almost opposite aims, since the aim of sketching is to introduce design change, while the aim of anchoring in PD has more to do with manifesting and fixing design decisions in order to make them visible to key participants.

This subject could be expanded forever, but what is important to notice is that whatever the impact on software methodology, the developer will be performing an active role that specifically expresses itself as a reality construction through programming code. The interesting role of the developer is not only pertaining the construction of reality in software code, but also the idea generation that happens at the stages of development, namely that the developer possesses a unique insight:

> *Developers who are personally more innovative with respect to information technology and have greater exposure to OO technology are inclined to respond to RAD with more alacrity. These individuals can serve as key change agents in diffusing the technology more widely. (Agarwal et al., 2000)*

2.7.4. Case study methods

During the case study, the developer will be engaged in a number of processes which shape to be either direct field work such as interviews and observations or indirect field work such as meetings and screenings (showing something on the screen). To the extend of the convenience in which the developer is situated, he can choose and combine different field work tools as needed.

All field work is performed in order to aid the development process. To reduce the costs, the developer, being just a single developer, can reduce work by sketching results in the programming code base instantly, or more specifically: Keeping the field work within the RAD cycle of user design $\rightleftarrows$construction.

The following tools and techniques are included, with no particular reference to preliminary works. Since these methods are not fully understood, we should describe their impact in the order of a *thick description* (Geertz, 1973).

Unstructured interviews: To gain insights prior to development, the developer can observe and inquire into the work performed by users of interest. Context is very important as it gains the developer more insights and inspires the interview situation. Interviews should be unstructured and open, because the developer has little knowledge of the work domains, but the more prior knowledge, the developer has, the more direction can be included in the interview. As the software progresses and can be made usable, testable or otherwise visible to the user, the developer can ask for specific input while the user is trying or using the software. Kensing (2003)'s principle of anchoring visions can be well-tested by anchoring what the developer perceives as a decision in software and affirm this anchoring through processes of interviews.

Observations: Participant observation can take place by watching users using the old IT system, a development version of the new IT system – or perhaps no IT system at all. To the developer, this can hold various values, either to explore or suggest specific design, or to make a quick improvement in development. Having the developer observe users at work is key to co-realization.

Being there: Another thing that's key to co-realization, is the basic principle of *being there*. It needs to be included as a technique, because the choice may in some cases arise when the develop for some reason is *not being there*.

Meetings: Making decisions visible and take place with consensus of affected users and management is often necessary in the design process. Meetings can also serve as a screening space, i.e. for the developer to showcase system changes since last meeting and gain feedback. Some methodologies such as SCRUM seek to structure meetings with regards to frequency and participants. Co-realization ad Hartswood et al. (2002) does not target meetings as a specific method, but it is inherent that the IT facilitation has to participate.

Real testing: Testing can mean many things, and the study makes use of isolated testing of functions. But it also seeks a coherent and holistic test scenario,

which is why we establish a *real test* of a new IT system by preempting cases where the system can fail, and making everyone aware that the system is being tested for the purpose of improving the design. Thus still using the system for real, normal purposes.

Regular contact: Regular contact means that the developer and users do not have to go through repetitions of previous processes due to loss of information.

Improvisation: Being able to quickly respond to new situations by introducing new problems, new application functions is one aspect of improvisation. Another could be the spontaneous deployment of other tools and techniques when a sudden opportunity arises. The term "improvisations" can unlock a range of easily affordable unplanned design and development decisions.

2.7.5. Research methods

Since I'm the research ethnographer and participate in organizational work and IT facilitation whilst also doing research, the observational work aimed at research will be limited.

The extra observation and diary keeping done for research purposes can also have an effect on project case work itself, though. Either because it results in extra work and observations being done, or because it adds bias to the regular case fieldwork, for instance such that observations are directed at the ones that seem interesting to research.

In this case study, as the developer is also the researcher, it is important that the process is documented in a diary and reflected upon. Furthermore, as all development is carried out with specific development tools, it is possible to use those for retrospective analysis (Krogstie and Divitini, 2010).

> *The objective of this paper is to investigate the potential to support project teams' retrospective reflection by the use of historical data in lightweight collaboration tools. (Krogstie and Divitini, 2010)*

Since I'm just a single developer, the notion of *collaborative* becomes an exaggeration, however the idea remains that historical data collected by a revision control system can aid the information from diaries in a retrospective analysis. Similar to the study of Krogstie and Divitini (2010), we use emails and changes to the software code base to *recall and reflection, aiding the reconstruction of the project trajectory.*

2.8. Summary

Firstly, in this background chapter, I have laid out the synthesis definition. In order to support it, I have listed a number of related studies, mainly nourishing the ongoing discussion of software design and development methodology. The aim has been

to fuel the understanding of software design and development as a sociotechnological practice, and as a consequence why development methodology should make use of ethnomethods, co-realization, and how rapid development can achieve this.

Finally, I have listed a number of loosely defined methods for the case study and research. They are an honest reflection of what has been practiced during the case study, and as a result the outcome chapter gives further insights to the course of events, and does not claim a perfect understanding of its methods.

3. Outcome

This chapter explores the empirical outcome of the case study, namely the process of developing a cooperative consisting of two music venues. The primary responsibilities of the cooperative are to develop, own, and run an online, digital ticket sales system. This cooperation will be referred to as the *ticket coop* in the following. The IT facilitator and researcher, i.e. the author, are essential in this development process and, as such, I will share the autoethnographic observations that I have found to be key insights for the case study. To quickly establish an understanding of the kind of software developed, Figure 3.1 shows a screen shot of one of many functionalities in the web application.

The chapter is divided in two main parts. Firstly, I describe the case as it happened. This includes key events, decisions, participants, environment, and how the rapid development process progressed. The case is described in chronological order and as situated as could be reconstructed. This does not entail all events being described with the same level of detail, but it is intended to give a clear understanding of the process in a similar fashion as an *ethnographic story*. Whenever there is uncertainty or processes hidden from my knowledge, I try to make sure that it is clear.

Secondly, I unfold more technical details about the programmatic design, development process, software libraries, and technologies. This is with a special focus of course given to the rapid nature of the development. All of these are technologies that I was familiar with before the study, and which has given me the experience required to assume the role of IT facilitator.

3.1. Case study: Organizations, environment, decisions

The story given in this section describes the process of forming the ticket coop. The story takes place between the initial phases starting from November 2012 and leads to the actual system being run for the first time in April 2013. The story has been captured through written diaries, meeting minutes and personal accounts. The ticket system itself has been finalized over this period. However, due to precautious behavior from the cooperative partners and overall slow decision making, the statutes of the coop have not been finalized. Thus, the coop itself concluded as a work-in-progress, but with a good sense of the final actions necessary.

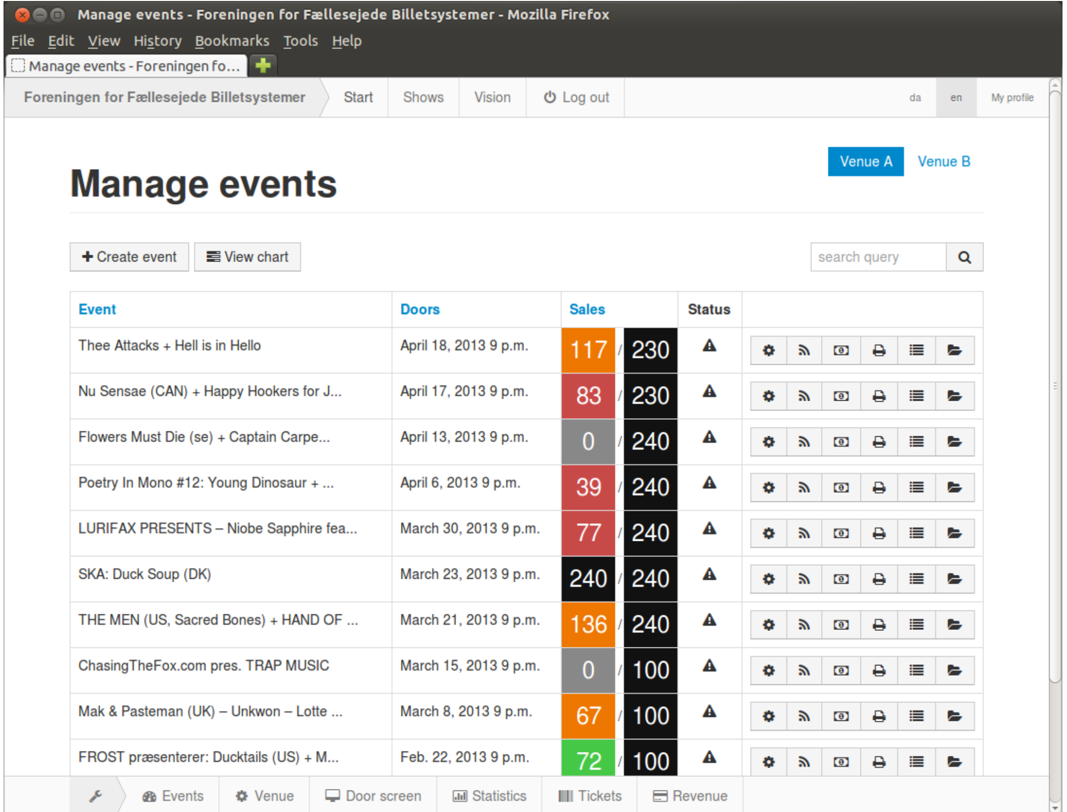


Figure 3.1.: A screenshot of one of the venue management screens. Data is fictive and the name of the ticket coop is tentative. Almost every aspect of this screenshot has been subject to discussion and nothing has been developed with the aid of design artifacts. All functions in the screenshot were fully implemented and tested. For more on application functions, refer to section B.2, the screenshot pictures the `Event list` of the Management Backend.

The focus of the stories is both to gather a few insights on the project itself and, just as much, to document the results and insights from being a developer on board every aspect of an organization – in aspects of strategy, management, work-life and organization. In other words, I try to tell the story with a specific goal of highlighting what I have found relevant to risky and rapid design spaces, both of negative and positive nature.

3.1.1. Project establishment

The origin of the ticket coop came from my own initiative (see section 3.1.7 about my own role). The functionalities of a ticket system are, however, heavily inspired by the multitudes of similar platforms[1].

[1] Before the coop started, we did not encounter any similar organizations, but during the process, a ticket coop for theaters was curiously enough established and launched.

Personally, I saw the ticket coop as an opportunity to start a project from scratch with a complicated and unknown context and environment. This also meant that I was taking an initiative that I would have to share with someone in order to really have a case study with *participants* (more on this in section 3.1.5).

3.1.1.1. Formation of the Ticket Coop

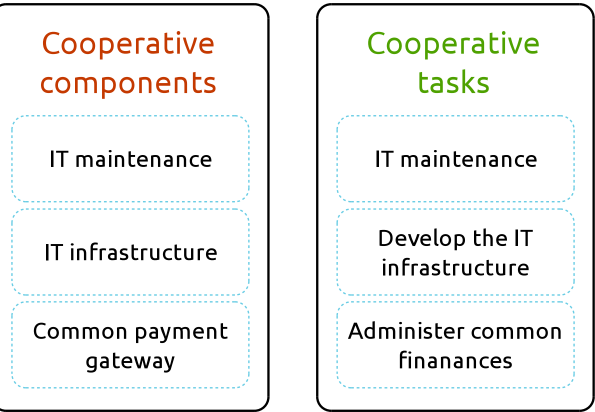


Figure 3.2.: Cooperative components and tasks proposed in the initial project draft

From the beginning, this project was to be a cooperation, a shared ownership of a ticket system. The idea was sketched out in an 8-page project description (see also section A.2). It was emailed to three different smaller Copenhagen music venues, attached with an optimistic email text. The venues were fairly similar in size, had a non-commercial culture and featured lesser known musical acts and low ticket prices. I invited to a meeting and two of the three venues responded with some interest.

I wrote the project description with the aim of capturing the interest of relevant music venues. It was not a technical description of the ticket system, a system design nor a project plan. A ticket system would have to be developed if they were interested. The project plan had a level of detail in areas of economy, technology and organization. The project description suggested that a lesson should be learned from the never-decreasing high fees of commercial ticket vendors which had made it difficult to offer ticket pre-sales on less lucrative shows, as low ticket prices meant a disproportion between the actual contribution to the musicians+venue and the third-party ticket system owner. The sizes of current ticket fees were also deemed unreasonable, since the function of a ticket system seemed within reach of a single developer acting on the knowledge of venues. Sharing a belief that we could create our own ticket system and acting upon it could thus as a final outcome liberate the venues from commercial ticket vendors, and at the same time create a digital system adapted closely to their needs. This refers to the *establishment of the entrepreneurship* and the sharing of *opportunity recognition* (see section 2.7.2). The

level of practical detail revealed components and tasks of the cooperative. Summarized (see Figure 3.2), they could be split into 3 components of shared ownership and development and 3 components of tasks to be undertaken by cooperative members.

More specifically, the project description proposed that the venues should share the costs of maintaining a ticket system, and that these costs would be significantly lower than a third-party ticket vendor. The two basic costs of the system would be the payment gateway[2] provided by a typical payment broker (there are many!) and the renting of a cloud server. It also emphasized the possibilities of co-developing an IT system to target the specific needs of the venues, and that there could be room for improvements through the devotion of the development methodology of the study.

Initially, the project description contained arguments about the prospects of targeted design and cooperation, but also complained that the current level of fees was unreasonable. It outlined economical figures, arguing that ticket fees could be as low as 1/10th of the current standard. The project description also argued that a ticket system should in principle be owned and operated by the venues themselves and seen as a core functionality. However, my own arguments were not the same as the venue's representative daily managers: 2 of 3 venues joined in, but they all had different sets of arguments:

Venue A wanted to join the project because it was non-profit and in the spirit of the venue itself, i.e., being run by volunteers and not for profit. The venue was very interested in cooperating with the other venues and saw this as a benefit and possibility. Venue A already has ticket sales provided by a third-party provider, which had found a way to provide sales without direct fees, but profited by printing commercials on tickets and possibly by returns from banking revenues. Venue A is non-profit run by volunteers.

Venue B already has presales for most of their shows, but each presale is done through an expensive vendor, which makes it unattractive to smaller and cheaper shows. Hence there was almost never presales on smaller shows. Venue B would like to try a system with little or no fees and is open to cooperate with Venue A. Venue B is non-profit and run by volunteers.

Venue C already has a presales solution, although it is twice as expensive as the solution proposed in the cooperative. The venue does not believe that a new ticket system can benefit them enough to change the current system. The venue is generally running well and they are not very desperate for change. The venue is not interested in attending the meetings. At a later informal meeting, a manager, when informed on progress of the project and the support gathered from the other venues, noted that “money talks” about their own interested in joining. The statement does not necessarily relate to any money made or lost

[2] A subscription to a third-party provider of secure online payment, price of which could easily impact the goal of no-fee ticket sales

from ticket vending, but rather reflects that time spent by the management on non-essential projects can be costly. Venue C is privately owned.

The project started at the end of November 2012 with the first meeting with the managers of Venue A and B. Afterwards, work was cut out for investigating fundamental parts of the coop: The statutes, the economy, and legal issues.

Similarities between Venue A and B: Both Venue A and Venue B are music venues with the sole purpose of hosting show nights. They concentrate mainly on underground music with audiences of up to 230 and 400, the music genre being for instance alternative rock, punk, metal, reggae. Both venues are almost exactly the same age, formed some 40 years ago. The difference, however, being that Venue A recently went bankrupt and was reopened with a new, more professional management, but on the premises of most functions being handled by volunteers. This is almost identical with Venue B's setup, except that Venue B's volunteer platform and vibrant structure contains individuals with experience reaching back to past decades, thus being significantly more established.

3.1.2. Participants

Before turning to the summary of events, here are the people who were key participants in the study. Participant 'D' and 'G' were by far the most influential to the project. Other participants mentioned made important contributions during the process. Participants who were only present at a few informal occasions are not included.

Participant		Role
'D'	Venue A	'D' restarted the venue from after its collapse and bankruptcy 5 years ago. 'D' is full-time daily manager with tasks like office work, economy, volunteer coordination
'E'	Venue A	'E' is part-time manager, working at the office with promotion, volunteer coordination, floor management. 'E' is a professional event manager with several years of experience
'F'	Venue A	'F' is floor manager and the venue's main booker. 'F' works on volunteer-basis. 'F' is probably the most regular volunteer in daily activities.
'G'	Venue B	'G' started at Venue B after the ticket coop project had started. One of 'G's first tasks was was to be regular contact and meet-up person for the project.
'H'	Venue B	With around 20 years of experience as daily manager and booker at Venue B, 'H' was perhaps the most experienced and professional individual in the project and participated closely in the beginning, but then gave over initiative to 'G'.
'J'	Venue B	'J' did accounts, statistics, and daily planning. Saw an early possibility to give input to the backend management of the IT system.
me	IT Facilitator	see section 3.1.7

3.1.3. Time line

App. time	Activity	Description
November 5	Meeting	Project description presented and discussed
November 30	Meeting	Establishing project, statutes draft
December-January	Emails	Discussing statutes, stalled
February	Development	Data model drafted, core system explained
February 12	Meeting	Discussing data model, project establishment. Consensus on splitting all coop costs equally.
February	Development	Ticket purchase
February 26	Meeting	Discussing purchase phase and next phase, project plan, deadlines
February	Development	Management backend, smaller meetings
March 12	Meeting	Meeting with 'G', discussed and reviewed some administrational functions. Venue B wants to drop coop payment gateway.
March	Development	Several days of development at both venues, complexities have been uncovered
March	Development	Switched to contextual development, difficult decisions made easy by sit-down session with the live development system.
March 22	Development	Test data added
March 28	Deployment	Participants are given user logins to development server
March+April	Interviews	During contextual development met many of the volunteers
April 3	Development	Starting to develop door screen
April 6	Observations	Gathered observations specifically targeted at door screen
April 11	Reflection	After several attempts decides that email communication cannot be used to gather feedback
April 15	First RAD cycle	Setting up first event for sale in the ticket system. Ticket sales kick off.
April 17	Live testing	Full test
April 18	Live testing	Testing doors and guest lists, even rock stars have used the system now
April 19	Part-conclusion	Development has been halted, most aspects of the system has been tested, except online payment.

Another way to express the progress of the project comes from observing the development of the software itself measured as lines of code (see Figure 3.3). From

this perspective and from my own experience, the project became very active and productivity rose once I started working in a contextual environment during the end of March. Normally, pace would decrease as a result of testing and feature stability, however the ever-increasing development in the code base size expresses that the project was cut short at a stage of high productivity. The method of post-analyzing the project from commits to a version control system could have been much better supported and used, though, and thus the graph lacks data points.

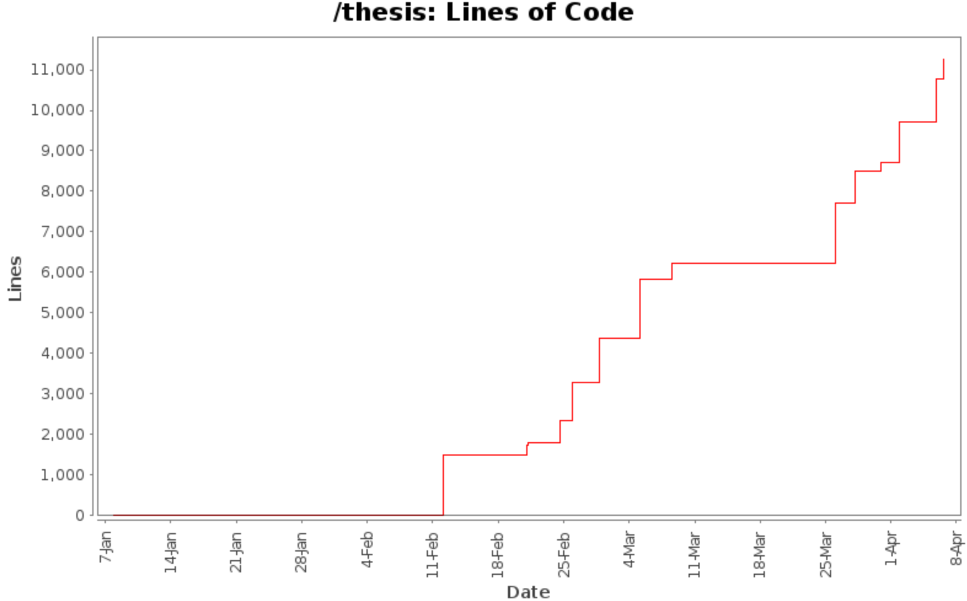


Figure 3.3.: Development of the code base as a function of time. The final count read 10,723 lines, excluding over 70,000 lines of code included from third-party open source projects, and additionally hundreds of thousands lines from frameworks etc.

3.1.4. Nature of the Cooperation

The following section gives an overview of the sense of cooperation that was established and the processes are elaborated in section 3.1.9.3. The study has not targeted an understanding of how music venues work, nor did the project seek to draw any preliminary work on similarities between participants and the nature of their tentative cooperation, it was simply *assumed* that the known similarities between the venues would exist at a practical level. However, out of necessity, such insights have occurred in the particular cases of the venues participating in the ticket coop as it established itself. There was no plan of how the coop should function before the establishment or during the development, but having the two venues cooperate seemed natural, because of their organizational similarities, music genre and audience.

At the end of the project, both parties had engaged in both common decision-making with mutual understanding, and as such, the coop was cooperating – even though it was not finally constituted.

The staff at both Venue A and B largely consists of volunteers, except for sound engineers and some of the door staff who on the other hand also participate as volunteers. The managements are professional and have very hectic daily routines, mainly office routines such as coordinating schedules, doing promotion work, volunteer shifts, socializing with volunteers, music booking, and managing inventory. Both venues have a full time head of management and a couple of part-time roles that also supplement the management on a professional level. Furthermore, both venues receive cultural benefits, requiring that the professional management satisfy a number of terms, most of which did not directly affect the ticket coop. For instance, both venues are headed by a board, membership constitutions, subject to general assemblies, professional accounts keeping, and documentation of activities.

Both venues are fairly accustomed to online technology, since music booking has been shifting towards this for a long time. Communication with touring bands and their management, promoters and touring agencies is mostly done by email, and the booking of touring acts, local support bands, DJs etc. is highly dependent on a digital and shared calendar. Not to mention that promotion is heavily dependent on social media.

There is a mutual interest in the cooperation, and meetings have resulted in an exchange of experience that went beyond the scope of a ticket coop. The venues acknowledge that they are in some ways competing for the same market, but the number of audiences to be choosing between a show at Venue A or B is always likely to be low. So rather than keeping secrets from each other, they prefer sharing experiences and goals, and the forum created by the ticket coop is valued in itself.

The cooperative mindset has been established over time. During the first meeting, I was regarded as a third-party, and the project was my own. One way that this played out was the mentioning of “your project”, “our [Venue A] needs” etc. which slowly but not entirely changed to “our project” and emphasis on common interests such as advertisement of the other venue’s shows.

Disintegration of cooperative interests was strongly linked with perceived risks of criminal or grave misunderstandings arising from the initial project description: Mis-management of a shared bank account, from which all ticket revenue would be stored, and channeled to the individual venues by some management figure. Since presale revenues are of a considerable sum, it was not an intuitive decision to have one venue to manage the other venue’s revenue, given the slightest feeling of competitive nature.

By the time we had reached the second meeting of February, enough talk and perhaps trust had been build for the manager at Venue A to make a fundamental suggestion in the light of the cooperation: To share the costs of the coop instead of estimating and accounting complicated fees on every ticket transaction. This was immediately

acknowledged by the representative of Venue B who had in the meantime been assigned the responsibility of the coop project. The outcome of the second meeting meant for the cooperation, that both time and effort had been vested in the project, and a more cooperative nature was established, i.e. that trust and solidarity should outweigh and give advantages over the complicated nature of tracking the usage of common resources.

3.1.5. Risks

To describe the risky nature of the project, we observer its risks at two connected and continuous levels: On the higher level are the risks as perceived by the participants, hence their commitment to the project. As mentioned earlier in section 3.1.1 the existence of such risks also supported that the case study itself had taken on a realistic nature. The other level is gradually more embedded in the project as software development risks. Ultimately, these are the risks of failure to deliver a successful IT system with some given measure. A mashup of the risks can be seen in Figure 3.4.

In the early stages of the project, the risk which was sensed and later confirmed through a debriefing, had to do with the nature of the study: *Student project* (see also section 3.1.9.2). No money was being spent and the costs had to do with time. During a phone conversation in January, I was prompted by a participant that the time schedule of the project was too tight, and from the participant's point of view, that my thesis plans were the true reason. Similar incidents happened throughout the project, in which the participants operated with a radically different time frame from my own, and I felt I had to over-step a border to bring about enough action to avoid collapse of the project. Typical suggestions were "after summer", "at the next board meeting" etc. On February 21, my diary concluded:

> *This project is only happening still because I'm pushing through. Had I been relying on the initiative of the venues, nothing would have happened at this point. (Diary, February 21)*

What I did not know at that time, was that things would get slightly better and support would increase as the system started being developed. Results, observations, and meetings were still possible as long as the intentions did not include statutes and legal decisions.

During a meeting at the very early stages of development, we went through the perceived risks of creating and having a ticket coop. At this point in time, the nature of the perceived risks had become more specific and targeted at the software:

- Errors in the payment gateway, as the venues were familiar with the heavy cost of credit card terminals in the bar not working
- Hackers breaking into the system and stealing information
- Flooding of server capacity, e.g. from popular shows

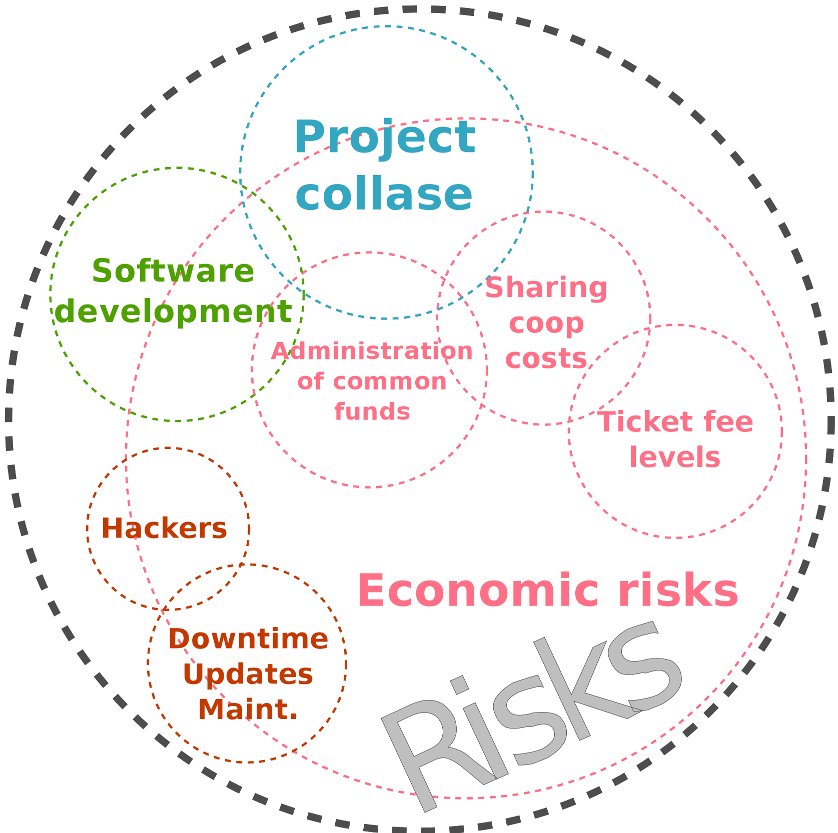


Figure 3.4.: Several risks were perceived by participants and brought up during development meetings. Risks are shown here to illustrate that there is *some* space which defines the riskiness of the project overall, and that the risks contained are related by definition or implication (hence the overlap between elements). For instance, hackers could pose a risk, which could cause downtime – but they could also pose some other unknown risk, which the participants did not know of, they merely sensed it. The risks include both the IT system and the ticket coop.

- Errors caused by system updates
- Errors from specific browsers
- Other errors - ie. the risk of unknowns, inadequate system functionalities, human errors etc.
- Single person maintenance, single point of failure, lack of backup

Furthermore, the initial establishment meetings in November showed a particular worry about the common administration practice of coop ticket sales. As fundamental parts of the system were developed, namely the ability to generate tickets, purchase them in an online setting, and administrating the tickets and revenues, the discussion of risks became more grounded. This paved the way for the participants and facilitation to give more decisive statements and look for solutions that could be measured.

Participant 'D' stated that from assessing the costs of downtime, they were willing to go as far as paying for a reliable service provider to ensure the uptime of the system. This was especially because I was the only system maintainer taking part, and 'D' wanted that as much money as were risked on system downtime should also be spent on preempting it.

Perceptions of risks were guided by a sense of what threat it actually posed to the venues, and facilitation was made easy by managers being very proactive about raising issues and explaining consequences posed to their own organizations. For instance, the discussions yielded a necessity for backup procedures as both venues could be prone to internet outage (as experienced before), and the system would only work when online. In case of a popular night being hit by system faults or internet outage, tickets would not be scanned, and entries could not be registered, and the venue would risk a faulty door entry process causing delays. In cases where presales could not be properly processed and managed, this would influence the cash sales negatively, and possibly postpone concerts, prompting both immediate harm to the venue and damage to its reputation.

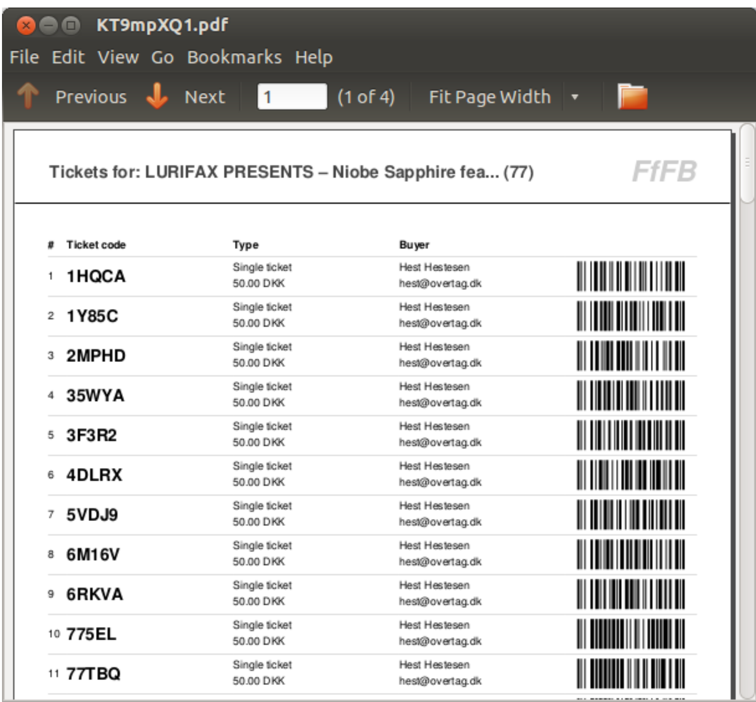
As a strange and paranoid conclusion to the February 12th meeting's discussion of particular risks, the manager of Venue A reassured that their concerns of system failure were actually *not* particularly strong. The statement indicated that the participant's perception of the risks were subject to change. Not from concrete factual grounds, but as an outcome of meeting room discussions, atmosphere, and intuitive reassurances.

3.1.6. Finding solutions to risks within the meeting space

Not all risks were laid aside for later analysis or good fortune. During the discussions, we sought ideas of how technology could practically lower risks or produce viable solutions. As IT facilitator, knowing the tools, and having theorized the specific data model of our system, made it easy to name realistic solutions within the meeting space, rather than reacting to issues raised by participants by setting up new investigative processes.

System and internet outages

One example of solution finding within the meeting space, was an issue raised at a meeting with managers from both venues. The need for a backup system for handling either internet outage or other system failure was discussed. The ideas for handling it felt short at the problem of internet outage, as this would cut off any redundant online backup systems. Instead, we agreed that a backup copy of tickets should be mailed and kept on a separate server. Since paper lists of ticket holders could possibly be kept in the door for other purposes, such as guests who had forgotten their tickets, I did my part as IT facilitator and suggested that the overlap



#	Ticket code	Type	Buyer
1	1HQCA	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
2	1Y85C	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
3	2MPHD	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
4	35WYA	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
5	3F3R2	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
6	4DLRX	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
7	5VDJ9	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
8	6M16V	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
9	6RKVA	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
10	775EL	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk
11	77TBQ	Single ticket 50.00 DKK	Hest Hestesen hest@overtag.dk

Figure 3.5.: Ticket lists as PDF. Each ticket has one row in the table, firstly the necessity of the list was discussed at a meeting, then the list was made, and finally the format and the contents were debated once while looking at the PDF itself.

and these interests called for a functionality of generating *PDF* lists of ticket sales containing information of ticket holders and barcodes (see Figure 3.5). At the next meeting, I presented the ticket lists generated by an added feature in the system. We reviewed the ticket list and found that it could conveniently be marked with a pen and scanned using normal procedures once the system was back up, also on the day after. The PDF file was easy to both email and upload to an external backup server. Once deployed, the perceived process of a backup system would be able to either stand the test or be adapted. Furthermore, the risk posed by developing this solution was proven low, due to the fact that it was implemented in just a few hours (see also 3.2.2), which I sensed already at the meeting from knowing a tool to easily generate and structure dynamic PDF documents.

Hackers and sensitive data

As stated from the February 12 meeting, Both participants from Venue A and B were concerned with the likes of hackers breaking into the system, which prompted a discussion on what kinds of data we would be handling. Thereby, we knew that even though the system might not be a lucrative target for hackers, it would still be necessary to maintain passwords properly hashed and not to include any sort of APIs that could pose a risk of financial theft. Facilitating this part was more

difficult for me, as I am not a computer security expert. From the data models, and from using external payment providers, we could jointly establish a conclusion that the information contained within the system was at least not prone to data theft. The same discussion also lead us to talk about permissions and roles in the system, ie. that ticket sales could in theory be sensitive data, for instance to cause someone to wrongly perceive their cut of entry sales.

After the discussion of hackers and data sensitivity, I decided to spend time implementing a developer-wise easy, self-contained API for securely handling permissions and authentication throughout the system, and even though this *should* be standard practice, the discussion of user permissions helped to guide the level of detail contained in the permission handling and to make it an early feature.

Sharing the coop revenues

Perhaps the risk that was hardest to grasp was the risk posed by the coop's handling of common funds, known as the *administrative role*. It related to the handling an transfer of financial assets on the coop's bank account, and the initial project description had naively stated that such a role would easily be dealt with by a couple of staff hours every month, time-shared by the coop's member venues. To the point of constructively dealing with the issue, I introduced several measures early on after sensing that the members were raising the issue almost at every occasion possible, especially in light of the coop ideally being open to new venue members. Firstly, I assured that all system data linked to calculation of revenue was calibrated with events such as price changes, ticket refunds, data deletion and such. Data would be kept safe through lower level process signal that would be fired at every occasion, essentially being the core business logic of the system remaining intact and topped of with revision handling of every object (e.g. ticket) in the database. I demonstrated the implications of these signals by showing participants at a meeting how even changes done by myself in the *backend* were captured by this mechanism.

Finally, together with managers from both venues, I reviewed how to deal with cooperative funds on the screen. At this stage, there was no bank account and no funds to handle, so we simply went through the process of buying a ticket, and tracing the added revenue to the screen on which revenue was summed up. The screen had only one button to indicate that action had been taken to transfer this revenue to the respective venue. Thus, there were no manual actions to sum up revenue, and we noted how additionally generated venue after the transfer would be counted in a new revenue stream. We discussed that the on-screen feedback should bring up an additional confirmation step, which was included, and the discussion enlightened me to bring about an extra check on amounts being marked transfered in case of concurrent ticket sales in order to avoid the complex situation of a concurrent ticket sale messing up an on account transfer.

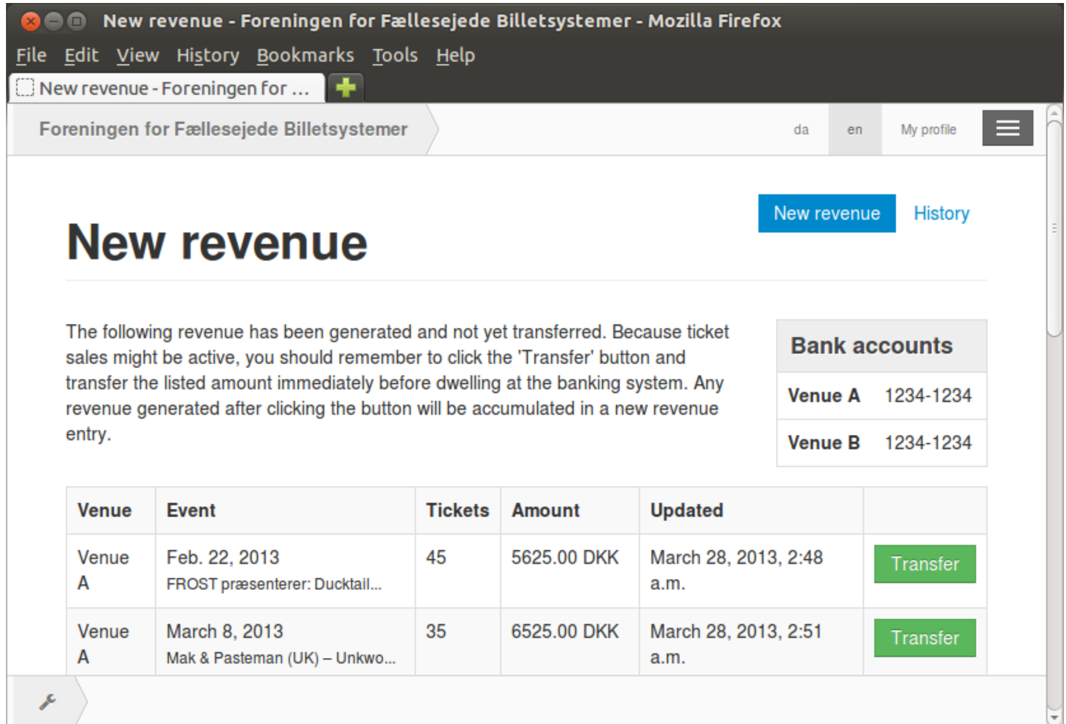


Figure 3.6.: Management screen for revenue administration. Curiously, this screen almost never changed, but the intended functionality of the screen was adapted from managing a single venue's revenue to being the accountants tool at each venue.

3.1.7. Ethnographic bias and IT facilitation

The following section is meant to capture a few observation on my double-role as IT facilitator and researcher. From my diary, it is obvious to see that it does not simply lay out observations of events and environments. Instead, it has functioned more as a daily reflection space, and to gather sparse notes for later reconstruction of events. This has made it easier to reflect on my bias throughout the process at a point in time when emotions and attitudes are hard to recall.

Bias

As a developer and professional, I have worked on web related software projects for a decade. I almost exclusively worked through informal hierarchies and mainly with contextual development. The work that I have done for the past years has especially been with rapid web frameworks (?) and database-driven applications, and because of that I started out from a position that the software should be an online application, running from a platform that I already knew the qualities of.

From my own experience, I wanted the data model to be the starting point of

development, because I have experienced the consequences of changing a data model later in the process. Changing the data model can result in much refactoring of code and as a concequence introduce many new errors.

The origins of the ticket coop came from my own initiative: I found it to be a convenient combination of interests (music, programming, volunteer organizations) and that it would be worth testing my new motto "source code should be free, and systems should be cooperative". I wanted to experience disjoint organizations cooperating on their IT with open source as an enabler, and to follow a familiar development process with a deliberate and introspective cause.

Briefly put, the idea of the coop had been conceived by wanting to engage in a structured and carefully noted process of doing rapid development, which I saw as a basic requirement for this sort of cooperative. Costs would have to be kept low, and rapid development was my answer to this problems. I wanted to formalize and study what I had previously been doing, to get *socially considerate* applications created and running with little bureaucracy and costs, and out of what I honestly have to say is a despise of contracts, planning and software with too little concern of actual needs, and a fear that incentives from third-party development often served to complicate and generate orders rather than target and serve customer needs and progress upon open platforms. I wanted to develop software while having full access to its context and have people feedback directly, enabling the kind of rapid change that I miss whenever organizational hierarchies would cut off direct to design-critical information.

Developer's diary

One of the intentions of keeping a diary (see section 2.7) was to maintain auto-ethnographic insights on the development process to retrospectively guide the storytelling on how the development took place.

After development took off, I worked both from home, and later at the venues themselves. It was not always possible to get hold of participants, namely the management at the venues, since they were busy. Much of the functionality especially at early stages was fairly fixed, so I could manage to work for up to several days before meeting to discuss results. Often, we would also talk on the phone and communicate by email, but those communication methods never became dominant.

The diary was not always well-maintained, as I experienced a limit to my own resources. It was slowly improvised towards a level of observation and reflection that suited my work habits, and I had to learn to take notes before the point of forgetfulness. Often, I worked until I was too tired to write a meaningful diary entry. One entry read:

> *The last two days, I should have written a diary entry, but I was too caught up coding (Diary, March 6).*

Some of the personal reflection on the process also had to do with frustration of lacking results from participants, especially on the work of drafting statutes (see section 3.1.9.3). I find that from reconstructing the process, that such entries have been especially beneficial and during the process itself, they have served as an emotional vent that becomes necessary when the work is lonesome.

3.1.8. Planning the project

In order to finish the project within a time frame and to have common aims, managers of both venues agreed on a project plan in February (see section A.1). The plan spanned roughly two months of development with 6 targeted milestones. The final aim was to be able to test the full system at the beginning of April. Total time allocated for development would be roughly three months including the time before the project plan was conceived.

The necessity of a time plan was brought up by 'G' who wanted a clearer view on deadlines for clearing and completing the statutes with his own board. 'D' acknowledged the schedule without any particular aims, except for the venue's own general assembly and board meetings setting the course for some of the activities, such as aiming for a completion of statutes or aspects of the system to be presented to the board at future meetings. In the end, 'D' offered that the venue could host a night for testing the system with full support of all the staff, a date that would have to be set according to the venue's concerts.

The project plan was created from my own understanding that the system would most likely be chunked in the following *modules*, not system modules with some API between them, but loosely coupled components in the sense of user roles, test cases, and data models:

Ticket purchase: This module would require the basic data models to be completed, e.g. `Event`, `Ticket`, `Venue`, `Revenue` etc. We did already anticipate that certain management functions such as refunding/canceling/changing tickets and changing ticket sales or supporting different sales for the same event would have to be supported at later steps.

Management backend: Logged in managers should be able to monitor ticket sales, create events, add ticket sales, create new user accounts, refund tickets, cancel shows.

Door check-in: Finally, we assumed that the check-in screen would have to be created through an entirely different interface for support quick and simple ticket scanning and sales.

3.1.9. Moving towards design-in-use

In this section I present the process and obstacles of arriving at a development phase that allowed design-in-use.

3.1.9.1. Using data models

Modeling data types correctly is important to application development, especially if it happens in an object oriented language and even more so, when it relies on database functionality. To a developer, it is therefore desirable to have the database planned at an early stage, but to a system designer, a fixed database schema can be a constraint (see also section 2.4.1).

The usage of an object relational mapping (ORM) framework, gave the opportunity to create real application models immediately and automatically transferable to both diagrams for easy visualization and to the database schema. A further discussion is presented in section 3.2.2.

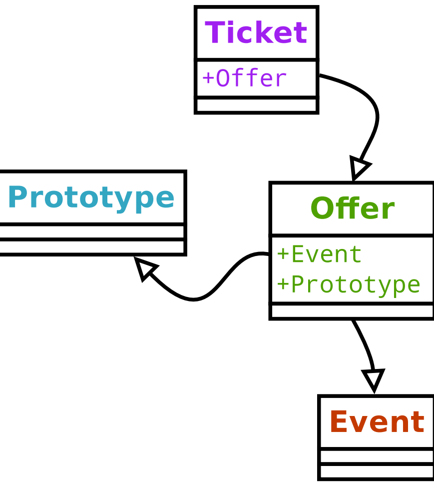


Figure 3.7.: A sub-set of the models, but perhaps the most unintuitive set of relations: Rather than a ticket being for an event, a ticket is related to an offer, that is for a specific event. The offer is based on a prototype (prototype for instance being “normal presales” or “2 person ticket”.).

After the preliminary meetings concerning the project description document and statutes, I had gathered enough insights to lay out the foundation of the system. The input for the data models were both issues stressed by the venue management (such as show cancellations, system permissions, ease of use, revenue calculations, and on account cash-outs) and functionalities of the three system components from the project description. Since these contents had not been contested at the meetings, I felt confident enough to start developing models for the application.

The initial draft of the model (see section B.1) was presented at a meeting, and we ran through all of the model fields, discussing their relevance and adding a few new fields such as phone number (venues may provide the service of calling ticket holders in case of sudden cancellations). In the diary, I stated:

> *Explaining the data model went fine. Everyone seemed to understand clearly the foundation of data, and the questions I put forward were answered with a common understanding and agreement. Apparently, a technical diagram is not that bad. (Diary, February 12)*

The data model should not be perceived as straight-forward, as it actually had a few catches, namely that of modeling ticket sales for an event (see Figure 3.7). We spent some time going through the model. My motivation for creating ticket prototypes was from a normalization perspective in database theory, ie. that if something was to be called a "single ticket" and have the same properties in multiple occurrences, it would have to be a separate entity. I devised a "presales" and "cash" ticket from this model, and explained the concept. The manager at Venue A told that they had many different ticket types already as they were using the ticket sales to register different kinds of entrants, e.g. journalists, promoters, band friends, volunteers etc.

3.1.9.2. Showing the first system screen and artifact

After the final full system test, participant 'D' told me that the initial reception had been skeptical as a standard precaution against student projects, but that this had changed at the meeting where the first system-generated ticket was shown and the purchasing procedure was demonstrated. No further system artifacts were presented at this meeting. The agenda I had laid out was concerning a range of issues on topics that I had found during development of system models and the first functional screens. Examples of issues discussed were:

- Should the system be multilingual, and which languages should we support? Outcome: English and Danish.
- When should tickets be refundable? Outcome: Never, only when shows are canceled or by management backend.
- When do ticket sales close, on a fixed or customized time? Outcome: Standard 4 hours before, always customizable.
- Edge case: What do we do if a payment arrives after sales are closed? Outcome: We let it pass.
- Edge case: How do we handle concurrent purchasing processes, ie. tickets are not bought through a single, atomic transaction. I suggested that some sites simply reserved the ticket(s) for at limited time and freed them if they were not bought. How to deal with malicious users? Outcome: Simply limit the number of tickets in a single purchase. Participant 'G' suggested to keep it a single-digit (1-9) because he had seen it somewhere - I think it had to do with fraud and not being able to add zeros.

We did not get to discuss all edge cases of the system, for instance handling of a login-free system in which authentication was handled through unique URLs sent in emails. This elaborated in section 3.1.11. Furthermore, I regretted that the participants did not show much enthusiasm in seeing the work-in-progress administration backend. I acknowledged that it would mean little to the immediate design decisions, but trying to convey an understanding of the nature of software development, the kind of extra curiousness could have been a positive sign.

> *One of the most disappointing reactions were that both parties did not want to see the [system administration] backend at the end of the meeting. Perhaps a bit of fatigue, but after discussing the data model, I was also quite certain that the simple functionalities of the basic backend would serve little to their understanding and my next target in the development. (Diary, February 12)*

From having been a long and sparse process in the November-February build-up, in which we mainly discussed statutes and strategy, the level of input and commitment had risen from the first meeting containing a work-in-progress IT system.

3.1.9.3. Changes to the cooperative platform and its statutes

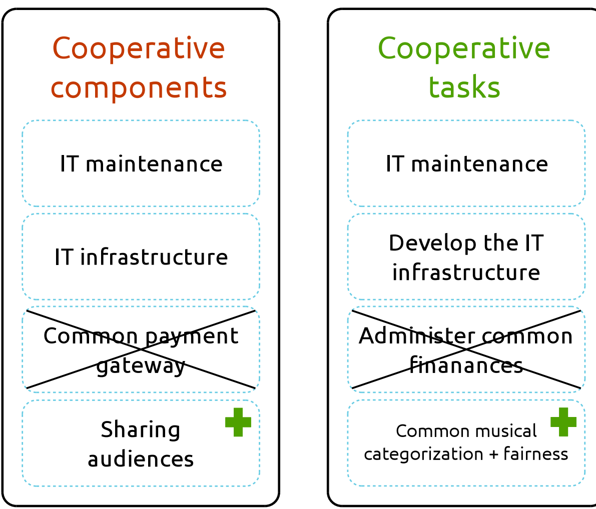


Figure 3.8.: At the conclusive stages of the project, these were the components of the ticket coop.

From the beginning of the project in November and until its conclusion in April, there had still not been a constitutional meeting nor a revised draft of the statutes from neither of the two venues. Consequently, the ticket coop only exists through intentions and its infrastructure, but not through a final mandate. In November, both venues stated their commitment to participate in the coop. During the process, we met two significant obstacles:

- The *administration role* of common financial assets, i.e. the revenue collected from ticket sales to be channeled through to each respective coop member.
- Splitting the costs of the coop, either through fees on every ticket or some other measure.

Even though both issues were finally solved by means of workarounds, the process of co-authoring the statutes draft that I had handed out in December failed to take place, even though several attempts were made to facilitate the process with

promises from Venue B to deliver inputs. I see the explanation for this mainly in the delay of finding solutions to two key issues quickly enough. Uncovering the solutions told a story of how the design of the organization was dependent on the design of the IT system, and how IT facilitation played a part. As a final outcome, and perhaps very obvious, we simply dropped a component and a task. During the process, the coop also gained a new cooperative component and task (see Figure 3.8).

Administrative role

The workaround solution to the administrative role came from an idea fostered from a draft addressing the perceived administrative role (see section 3.1.6). Furthermore, it was established that government regulations on financial activities ruled that in order for the ticket coop to handle assets on behalf of its members, it would have to register every in and outgoing transaction, which would be subject to auditing. The final solution thus became that the venues would ditch part of the cooperative foundation, ie. the sharing of a payment gateway. The idea of ditching this part, however, had also come from a quick and random encounter with the manager 'H' at Venue B:

> *Before the meeting, I spent 10 minutes by chance with J, the daily leader of the venue. He was still focused on the administrative part of the ticket system, the potential for a breach of trust in the government of funds. This was one of numerous occasions where he would explain the amount of revenue being generated by events, especially the more expensive ones where 400 tickets would be sold [economical figures removed] and the importance of having secure legal boundaries within the coop. [...] more players are not necessarily an advantage to the coop, since the economic part of running the ticket system is quite low and he was even doubtful if it made sense to share it with Venue A once it was up running. (Diary, March 12)*

The decision furthermore turned out to be harmonious with the already-running individual payment systems integrated in both venue's bars, and we would only need to extend already existing contracts with payment authorities in order to process online payments. Furthermore, the cost of having the administration of coop finances would very likely exceed the cost of an individual payment gateway, which would enable all presale funds to be handled on an individual bank account.

'D' had already talked to the accountant at Venue A, and he explained carefully how bank accounts and system accounts had to be aligned and chunked into periods and balanced. The accountant who had overheard part of my conversation with Participant 'D' jumped in and was briefed on the system's functionality and how historical data was preserved. He said that it could be necessary to generate a different kind of export for his account keeping system, and was on par with 'D's understanding that a ticket coop could not be handling financial assets on behalf

of the venue, but that the screen built for the administrative role was perfect for marking transfers and balancing the ticket system's accounts with bank accounts.

Sharing costs

One of the less elaborated cooperative tasks[3], would have been to either budget or post-calculate the number of tickets sold to estimate an individual cost for each ticket. This would either impose a fee on every ticket or an individualized membership fee for the participant's in the coop. It could be argued, however, that as solidarity and trust is established in a cooperation, the necessity and utility of such metrical cost cutting is shadowed by the simplifications of the cooperative effort. Practically speaking, two members owning a cooperative would find it easier to simply split costs 50/50 than calculating thousands of ticket sales and forecasting individual ticket fees.

The understanding that equal cost sharing was the way to go came completely unprovoked from participant 'D' in February at a meeting with management from Venue B.

> *Both parties want to split costs and simply do a no-fee ticket system, and run a coop in which all costs are seen as fixed, annual costs. This is a new development that requires revisiting the statutes and project description. (Diary, February 12)*

3.1.10. Arriving at design-in-use and rapid development

The first meeting in February constituted the first review of artifacts. This, however, had nothing to do with design-in-use, though we were starting something that resembled rapid development. Inputs were given from the participants at meetings and actions were immediately taken to implement those on the same code base that was intended to comprise the final product.

The most design-in-use that had been achieved until then, had been my own development method: To develop the system functionalities directly targeted at screen images that I envisioned as the final result by constantly seeking a *final-like* representation through layout and user text messages.

Adding test data and deploying

After the first meetings' presentation of the ticket purchase process, the next component to be implemented was the management backend. It became clear that

[3] It is not shown in Figure 3.8 because to my knowledge it never played a part in any discussions, however at meetings we discussed the possibility and usefulness of ticket fees as if they would occur without ever identifying any mechanism at work

functionalities that were to compute thousands of tickets and gather meaningful statistics would need data in order to fully understand and verify correctness (see also section 3.1.11).

Once test data was available, I sent a screen shots to the participants picturing fictive ticket sales for shows that were actually in the calendar. A week later, I deployed the full work-in-progress system and handed out logins. At this stage, the purchasing function and management backend could be tested. To my knowledge, non made the effort, however, and even if they had, the outcome would have been undesirable for the communication of feedback, since a user experience hidden from my perception would lead to no further improvements. Strategically, I knew that deploying the system on a live test server was a necessary means to gain experience with maintaining a live and production ready environment.

Guest lists

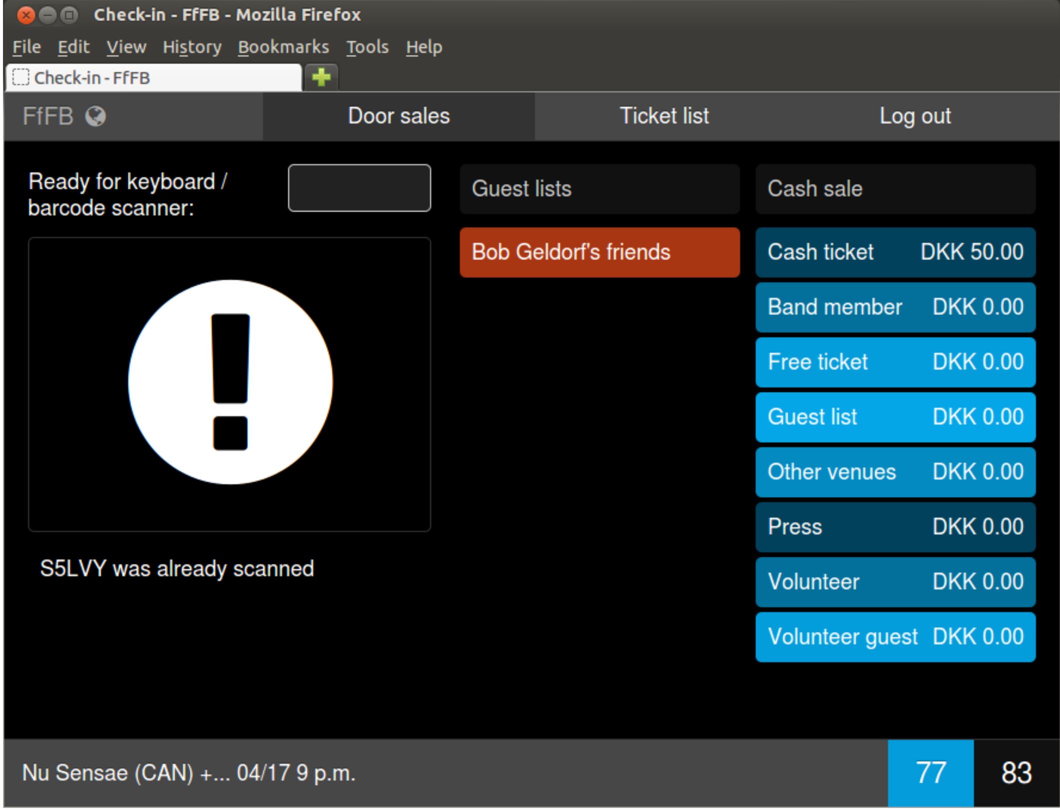


Figure 3.9.: Check-in screen for ticket scanning, cash sales, and guest lists. Curiously, Venue A had implemented their own ticket definitions, meaning that the cash sale also contains “guest list”, denoting that a guest has entered from a paper guest list.

Once situated at Venue B for a couple of days, inquiring into and having interested

inquiries from the volunteers, a volunteer working the doors at Venue B asked about the possibility to do guest lists. Not just as a modest request, but actually the first idea that sprung to his mind. The problem was that guest lists were difficult to manage as it usually meant multiple sheets of paper being handled, one for each band, one sheet for volunteers, staff guests etc. Once passed to the door staff an hour before opening, the guest lists would often be changed anyways through many lines of communication.

This prompted an immediate response from my rapid habit, and I drafted a guest list feature that would make it possible to exchange unique links to guest lists such that the venue could simply email a link to a digital guest list to a band. The band could then fill in their slots, and the data would automatically end up in on the door staffs screen.

However non-ideal the pen & paper solution for guest lists would seem, the management at Venue B after a short discussion immediately concluded that guest lists could never be digitized, and that I should not bother continuing down this path. This decision lasted for a couple of frustrating minutes in which I was trying to swallow the decision of deleting yesterday's work, but then a new volunteer stepped into the office by chance just to greet and let me know how fantastic a feature a guest list would be! As the management at Venue B had not heard the story directly from the volunteers, we took some time to go through the features on screen which were ready for guest lists. In this way, the management could both grasp the functions and options for a digital guest list, while the volunteers could make their feature requests heard. In this way, we arrived at an adjusted version of the guest lists, where management's wish to have the guest lists close and have multiple entries on the same line were granted and implemented while we were still sitting, talking, reading emails and I was writing code in the same room.

Guest list observations

A worst-case scenario of using current paper guest lists could be: A guest telling a band member about another guest, the band member telling the booker, the booker telling staff at the venue, and the staff at the venue then looking for where the sheet of paper might have gone, then finding that there are no more slots, after which the response would propagate back. And the scenario was not just cooked up, actually this was a situation I encountered while field observing a night at the door. And not only that, my sparse observation easily concluded that the guest lists were taking up huge amounts of time, requiring action from an authorized person (*floor manager*, *booker* etc.) or a band member to emerge from the back stage etc. Floor managers also complained that often bands would try to add more people than they were allowed. Even worse, once people were checking in, the guest lists were often prone by errors such that guests would have to dial or even use their smart phones to show email evidence of their claims to enter.

In the end, guest lists were merged into the ticket system's management backend,

check-in frontend, and a front-end for bands and bookers to fill in. Even though practices from the two venues were slightly different, the final result was supported by both managements. The management at Venue A was so happy that they stated they would setup a computer in their backstage. Band members, that I had interviewed briefly about their test of the guest list, had prompted that they found it stressful to have to locate a computer in order to fill in the guest list, and the management – always skeptical of *luxury* demands from bands – at the venue found their problems to be solvable by simply putting a computer backstage.

Door check-in: Scanning tickets, cash sales, and guest lists

Before starting to develop the door check-in screen, I turned to a couple of observational nights at each venue. The results were quite different as both venues used different ticket systems, each with their own significant technical issues to learn from. Furthermore, the volunteer staff was very different. At Venue A, the volunteers were new and unexperienced, and at Venue B, they had up to 10 years of experience and had a very professional mindset. Both experiences gathered one major conclusion: That the user interface would have to be very user friendly, efficient, and absolutely free from errors.

I sought this error-free, user friendly, and efficient interface with all the technical experience, I could gather. A further, significant factor came from the very likely introduction of hand held devices, and the already existing difference in the platforms that were in place.

Observation at venue doors The door staff at Venue A used an old touch screen computer with awful responsiveness, culprit in multiple errors. Furthermore, as they had one system for cash entries, another system for ticket scanning, and a third system for guest lists, the volunteers often became confused. Tickets were not scanned because the single computer could not handle two systems at once, and the poor touch screen interface of the cash sale system resulted in several errors from different volunteers in a short space of time. For instance, they would register the wrong price because they had to type in the price every time, and unknowingly push the same button several times, resulting in wrong ticket counts registered. At Venue B, there were errors during observations, but the more experienced staff informed me that the ticket system's user interface often resulted in wrong registrations by, briefly put, indicating success on screen even though a scanning had failed. Several guests showed up at both venues with a self-printed tickets that the scanner could not read, a smart phone image of the ticket, and one guest even had a story about how they she forgotten the ticket at home, 100 km away, such that the door personnel would look up the ticket by name and email.
I also inquired volunteers about the fun of work. They were happy about sitting in the door, because it gave them contact with guests. Complains about the ticket system was that it would defer them from real human contact. One volunteer enjoyed technical problems scanning barcodes, because it gave a pause in work to talk to a guest by asking for details and looking them up in the system. However, looking at the computer screen was painful, because it was a normal, bright Windows user interface in stark contrast with the dimly lit room. So we decided to have a dark-themed user interface. Furthermore, the response from the scanning procedure was a large icon indicating success or failure such that the screen could be perceived with the corner of the eye.

The final solution was a responsive, web based interface that used asynchronous callbacks to allow for efficiency, even on a slow or periodically failing internet connection (see Figure 3.9). The scanning itself could be done either with a barcode scanner or by typing in the unique ID for the ticket. Furthermore, it presented all three functions on the same screen: Cash sales, ticket scanning, and guest lists.

From the observations, I gathered many confirmations of what I had already sensed; but there was one irreplaceable lesson, namely the time allocated for each ticket to be scanned. I had thought a bit about how many seconds there could be spared for every scan with up to 400 people entering through the same counter within 1 hour. In theory, it gave 9 seconds for every guest. But theory is very far from practice, and being present with the staff gave me a perfect experience of just how frustrating even the slightest software idiosyncrasy can be. It meant that I chose a full setup while developing, i.e. sitting with a barcode scanner and a paper ticket to perceive the system at its fullest function and not leave anything to chance.

Ticket purchase

At the very end of this description of design-in-use and rapid development comes the ticket purchase mechanism. It was developed as the first system component, but development lay dormant until full system tests arrived at the end of the process. Until then, only sit-downs at meetings had introduced the functions of the ticket purchase. However, the ticket format itself took on quite a different shape once deployed and handled as an artifact by the managers and through inquiries about print-it-yourself tickets in general:

- Managers at both venues were interested in sharing a platform for communicating their shows. In their view, the more promotion, they could get, the better. And because of their mutual similarities in music genre, they got the idea of putting a "related shows" list on every ticket.
- During observational inquiries, it was found that many concert goers arrived in groups, with as much as 10 tickets individually printed on a full A4 sheet. This was inconvenient to both guests and door staff, and we devised an option to print multiple tickets on the same sheet.
- Managers wanted ticket buyers to spread their shows on social media, and suggested that a ticket buyer could link to a show on Facebook. This was implemented during the first stage, and when the final system test came, one of the ticket buyers had mentioned appreciation of the option to share the ticket purchase on social media.
- I had made emails to be sent with tickets attached in PDF format. Even though the emails contained both text, show date, ticket ID, and more, it never generated any reaction. However, participant 'D' noted that ticket buyers sometimes complained that tickets had not been sent in the current system and wanted a feature to resend tickets.

3.1.11. Hidden decisions

During the development process, I quickly came to realize that many decisions were made with little interest or influence from the participants – security, performance issues etc. Far from every aspect of the software had been directly influenced by a discussion or decision. I had to do my own reasoning on the cost/benefit of further inquiry.

> *While working tonight, I came to the idea that most likely, many of my decisions will never be told to A and P. They are technical decisions, and there will never be enough time for such details. Even some of the design decisions are probably not going to be mentioned, although I think we covered the whole data model diagram at the last meeting. (Diary, February 21)*

Some decisions even illustrated an internal conflict, reflecting a lack of coordination with project participants:

> *[I am currently] Doing a sprint towards a deadline [to satisfy the project plan], and I see a contradiction in the sense that working hard on programming code will become a choice defeating the need to build/distribute the software such that it may be tested. Any software needs building, in this case deployment on a web server. I had promised this last week, but still failed to prioritize it, rather doing more features and fixing known errors before actually releasing the code. (Diary, March 6)*

The following cases illustrates various hidden decision making within the development process:

Deploying a test environment: During mid-stages of development, it became necessary to setup an environment that was freely accessible for participants, such that they could experience the system, although it had not yet been applied to any real tasks, i.e. events and presales. However, to gain insights, I wrote a test module to generate test data. These data were not a true reflection of historic events and merely simulated ticket purchases for a number of past concerts. At this point, participants were not saying *we want to see the system*, rather I was saying *you need to see the system*. Furthermore, the decision was clouded and postponed by decisions to continue working on known errors and features and several days passed from my initial promise to grant access, until I released the test environment (see Diary, March 6).

Fiddling with test data: Since the system would have to handle huge amounts of test data, it was obvious to myself that every screen artifact had to be envisioned with thousands of tickets and a long history of concerts. However, I found that to most discussions and sit-downs, participants did not mention any long-term perspectives regarding data handling.

Non-PD ideas: Many system functionalities were a result of my own perception that they would be necessary, i.e. that the system would be useless without them. There are the basic cases of CRUD (Create, Update, Delete), in which every object needs basic manipulation functions. Such features were never mentioned by participants, but at times we had discussions on *who* should have access to *which* object manipulation function. Another set of ideas which occurred outside of PD-like sessions were the idea of statistics, such as showing time series graphs of ticket sales figures, and comparing different shows. Even though I repeatedly inquired for demands for these statistics or alternative ideas, I never had any input.

Edge cases: During implementation, I had to analyze several edge cases of the purchasing process. What should happen if a show was sold out during a purchase etc. These questions were laid out and found response. However, probably due to lack of resources, I found myself noting (Diary, March 24) that I had to make decisions regarding the validity of tickets price at 0.0, since

they could have ambiguous properties such as `is_paid`. In order to solve these, I made a lower-level design decision which in turn influenced the behavior of issuing free tickets.

Security level: I decided to log the remote IP address in case of invalid system calls, e.g. when a URL for a non-existing ticket is requested. After a number of attempts, the IP address would be blocked. This was necessary to block brute force attempts to discover valid ticket IDs.

Reconfigurability: All aspects of the system have been designed to handle localized data. Dates, currencies, even VAT is regarded as a localized setting. I do not expect participants to make such observations, but I found it to be a good idea to inform that the system would be able to handle these. After all, such reconfigurability comes at a low cost, development-wise, but discussing the risk of a VAT change for cultural events in Denmark would be *very* hard!

Remaining risks: Some edge cases were simply not solved or discussed. They should be, but even though I made note of them, I never brought them up at a later point. Examples are: 1) A show that is canceled and refunded while a ticket purchase is pending and 2) Tickets are sent or refunded to an invalid email address, i.e. a customer does not receive receipts or notifications.

3.1.12. Testing the system in its real environment

Once the full system was made operational, it was time to test it. The test included the full range of processes, except for the payment step in the ticket purchase, which was replaced by a dummy form for typing in credit card data. The first test night did therefore not sell real tickets, but invited concert goers to test the system, bring the free ticket which in turn would grant them a free beer.

1. Firstly, the venue management would create the event and guest lists. Then, they would announce the event with a widget on their own web page and a link on Facebook.
2. Concert guests would use the purchase process to buy a ticket, receive it buy email or print it on screen. 22 people participated, most showed up with a printed ticket, a couple showed up with a ticket to be scanned from their mobile phone.
3. During the first night, the cash sales and ticket scanning process was tested. 83 people entered, and 16 of the 22 free ticket testers showed up. At the second night, 117 people entered through cash sales, and 20 of the 24 guest list slots filled in showed up. 3 bands filled in guest lists.
4. After both shows, the *floor manager* would conclude the night by reviewing the numbers of the report and send any corrections to the venue management.

> However, no corrections were amended, and after inquiring, both floor managers said that no adjustments that they were aware of had to be made. This is the best evidence found that the check-in process had worked flawlessly.

One of the more controversial issues had been how to verify guest list data. By freeing the entering process for guest lists, and having bands communicate the guest lists directly to the door staff through digital media, the floor manager and venue management were afraid of loosing control. For instance, one band would enter "John Doe +2" in the name, which the door staff would interpret as legal data since it showed up on their screen. For this reason, we immediately banned numbers in the validation of name field data.

> *After creating the event, we found that the widget to be put on the venue's homepage was defunct. I fixed that instantly. After that, we had an issue because L had clicked Buy several times. This lead to tickets being included in the same purchase, an intentional feature, however something that did not play along with the max-1-per-email restriction. I fixed that instantly as well. (Diary, April 15)*

One of the features of the first version of the ticket purchase system had been that users could add several tickets to the same purchase. During the *free beer* test we had imposed a restriction that only one ticket could be bought per email. However, already at the first test, we found that the first feature conflicted with the second feature, so we introduced an option that a ticket offer (recall Figure 3.7) could have a customized maximum number of tickets per purchase.

Many insights were created from the full system test, and some were implemented immediately. Unfortunately, the system tests were late in the process, and the check-in screen could not be iterated again after the tests. Observations from the check-in process were never implemented, but they gave insights to further improvements, although the overall experience had made both management members of Venue A happy and optimistic of the full realization of the ticket coop within a few months.

3.2. Software product

In the following section, I explain the functionalities of the final ticket system, and what I found to be the most important aspects of the tools I used to develop the software platform using rapid development and IT facilitation.

I will not describe all functionalities of the application. Examples are limited to what I find to be most important to the ticket coop or to illustrate points of the methodology. The tools and libraries described in section 3.2.2 are all very recent open source projects, and the final application can be seen as a product of its time and age, using what I see as state-of-the-art techniques for web applications.

3.2.1. Application functionality

In the following section, I will describe the overall functionality of the application, as to give a sense of what has been designed and ultimately developed. For a full list of application functionality, see section B.2.

Recalling that the system was divided in three components (Ticket Purchase, Management Backend, and Door Check-in – see section 3.1.8), it should first of all be noted that the description of these components held true to the final result. The system was implemented in a fashion that resembled the objectives from its planning stages, and deployment in a real environment was finally done in all aspects, except that it did not include payment processing, and real usage was limited to just 2 practical cases, i.e. concerts.

All of the application's components were operational from a web browser. The interface implemented modern responsive layout techniques, making the UI agnostic to both handheld devices, touch screens, and PCs. Data processing respected all kinds of *state of the art* validation, the full application was localized, and the application was readied to scale for deployment in a fairly large production environment.

That the application was a web application means it was not defined by a singular application interface. It could conceptually be divided into several modules, for instance by using *server side* and *client side* paradigms. Whatever the modular divisions, it is hard to define the causality of each module's emergence, nor to describe a functionality as an individual entity. To maintain a focused discourse and keep this description short, the we turn to a more user-oriented focus on application functionality. If we chose a more technical discourse, for instance a description of a *ticket scan* could follow likes of data transmission, application state, and functional input and output, and we could trace through keyboard input passed on to the browser application, markup language presentation, network layer transmissions, server side application layers, and database manipulation – and then all the way back again.

The kind of application description that is laid out in appendices, and in this section, is meant as to guide the analysis with a perception of *what kind of* application was developed, but not to dismantle the whole software architecture.

Responsive layout One of the features of the application was the broad adaption of responsive layout techniques (see Figure 3.10). As a feature, it meant that many scenarios of usage would be satisfied already from the beginning, and without having to create multiple independent UIs, adding complexity and further testing. Scenarios count:

- Ticket holders can order and show their ticket on mobile devices.
- Managers can use the backend for creating and updating events on the go. Furthermore, they can check up on statistics (a requested feature during development).

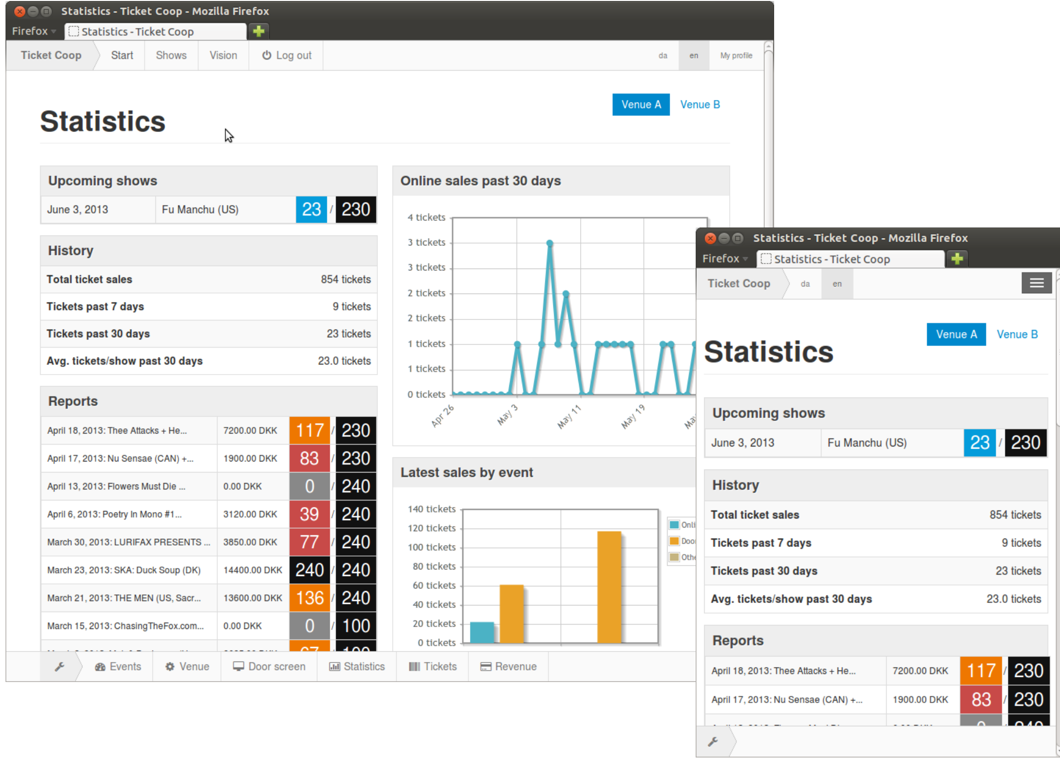


Figure 3.10.: Responsive layout techniques were used throughout the UI. Screen shots depict the same web interface, but seen with a normal PC size and mobile size screen, respectively.

- Check-in screen can be operated from touch screens, and even smart phones, with or without a barcode scanner.
- Printer friendliness, especially many management backend pages, such as event reports, are also relevant for printing.

Statistics and visualization

In order to guide the venues in their promotion and reflection on past achievements, the backend comes with a number of figures and statistics. The implications of selling tickets were that the venues had to know:

- How many tickets were sold at any given point in time: Displayed on both event lists, reports, and statistical overview
- How much revenue was generated and to be able to balance accounts between the ticket system and book keeping systems: Is displayed in the event report and a special page for extracting and fixing revenue counts.
- How ticket sales progressed over time: Can be compared with other events on the events list and seen as the latest figures on a statistical overview page.

Social media

By clicking a button for every different popular social media platform after completing a ticket purchase, the ticket holder can easily inform peers on Facebook and Twitter about their concert plans. Event pages are made using *meta tags* for integrating on social media platforms, such that links can easily be posted by event promotors.

Venue managers have access to copy and paste a special embed widget, which they can place on their own respective websites, and bands etc. can use it on theirs. The widget has a dropdown menu for selecting the number of tickets to buy and a *Buy* button, taking the user directly to the purchasing process.

Check-in screen

When scanning tickets, selling cash tickets, and validating guest lists, the door staff can operate a dark-themed and very simple UI. At its maiden voyage, it was used without any complications or known mistakes.

The UI puts default focus on the Ticket ID input field, meaning that a keyboard or barcode scanner will by default feed input to this field. In this manner, tickets can be continuously scanned without the need for any other interaction with the UI than with a standard USB barcode scanner.

If problems arise, such as un-scannable barcodes, people who have forgotten a ticket, or cash tickets that need refunding, the door staff can navigate to a screen that lists all tickets.

Furthermore, the screen displays the number of current entrants and the number of total sales, including presales. This gives the door staff an insight in the number of people present inside the venue, such that they are informed in a way that they can easily update other staff members, and safely know when to close the doors in case an event becomes sold out.

The whole interface for scanning and selling tickets and marking guest list entries is asynchronous in its communication with the web server. This means that broken internet connectivity does not cause data loss or system freeze. The staff can continue to scan tickets etc. without pausing. The UI will warn the user that she is working offline.

Easy UI

All application components are made with a maximum focus on minimum interaction. Feedback is always given upon interaction, for instance every form update results in a success or error message consistently placed at the top of the following screen.

Form fields in the backend are made using dropdown menus whenever choices are limited, and for instance all date and time fields are based on a calendar widget. After form submission, if a form field is invalid, it is marked red, and the error message appears next to it.

Navigation and language has been adapted to common phrases used by the venues themselves during the process. For instance, guest lists are modeled to be almost identical to normal guest lists. As such, the software does not attempt to introduce any new unnecessary terms, the most severe exception being the use of ticket prototypes and ticket offers (see section 3.1.9.1).

3.2.2. Technologies in use

In the following section, I review some of the upper-layer technologies that have been fundamental to the development and design. By upper-layer, I refer to software which has a direct interface to my application project or development process, for instance a software library that is imported into the project's source code or a tool that I use for writing, building, testing, or deploying the project.

Django, a web framework for *rapid development and clean, pragmatic design*

The most important tool of the project was Django (Django Software Foundation (2013)), an open source, Python-based web framework for writing the server side part of the application. Core to the principles of Django, is the slogan *Don't Repeat Yourself* or *DRY*, adding intent to the efficiency of work. Django is based on an object-oriented paradigm, and the easiest way to understand the framework's role is by observing the flow of web server request and response, i.e. that the web framework handles a request and returns a response:

```
django(http_request)->http_response
```

Anyone familiar with the *Model-View-Controller* (MVC) can imagine Django as the controller that calls the programmer's view function, which invokes a template (usually an HTML file), and fills it with data from the database. Inside the view function, the programmer does not indulge in parsing HTTP requests, creating response objects, or writing SQL for database queries. The utilities of Django and its layered handling of HTTP requests result in an architecture in which different programming languages can be handled separately with clean interfaces. Functionalities can be divided in a logical, testable, and reusable manner. Furthermore, Django can be seen as a toolbox, from which common tasks are standardized, and based on long-standing community discussions and best practice. I found the following properties of the framework to be indispensable to the outcome:

Class and model inheritance: Data models and classes can inherit properties of ancestors. As DJANGO comes with a lot of common functionality which can be extended, it has an immediate impact on functionality, and in most cases, a usable interface and database architecture can be sketched and deployed without minutes. An example of such inheritable structure is the authentication system, which gives any project a full scale, but basic authentication system, ready for further adaptation.

Decorators: In PYTHON, which is a *higher order* programming language, it is possible to wrap functional calls easily around other functional calls. In this manner, functional calls can be added and reused in an explicit manner, suitable for e.g. authenticantion mechanisms that need to be implemented in a uniform and strict manner across a software platform. In our case, we used it to easily and explicitly check for venue administration rights on the management backend. In the application, I tried to immediately acknowledge authentication functionalities that would be reused across several class structures, making subsequent features easy to implement and authentication mechanisms easy to alter at a later point.

Class based views: CBV is a relatively new design pattern of the MVT arena. It makes it possible to piece together web page functionality such as showing a list of database objects, creating a new object in the database, logging a user in etc. Core to CBV is that compound functionality has been split and parameterized to allow for class inheritance. All views of the project extend highly similar common functionality from the built-in CBVs of DJANGO.

Object-Relational Mapping: ORM makes SQL statements redundant and translates database results into native programming language types, a popular feature of DJANGO and similar frameworks. Many critiques argue that using ORM may result in lower performance, but to the rapid developer, and to this project, it meant for fast paced development, and I did not observe any performance issues.

Gettext translation: Having a simple habit of wrapping all UI messages in gettext calls, means that an application becomes easily translatable. At the end of developing, and iteratively, I could translate the UI in a separate environment from the application.

Forms: DJANGO comes with a library for generating forms for data models. This includes the ability to convert the forms into HTML markup and process the HTTP response object, safely validating data and saving it to the database. I used this for all forms throughout the project, cutting away needs of writing validation, passing data from the form the database etc.

Admin backend: Any system needs an administration or superuser backend. In Django, this comes almost for free, as it is automatically generated from the data models of the application. Automatically generated means that the application will have all CRUD web interfaces readily available.

Documentation: A guiding principle and in some cases culture of DJANGO and its many models, is the style of documentation, which often features copy-paste ready examples and thorough explanations. Documentation has a focus on user friendliness over technical precision, as the latter is preferred to be found by reading source code.

Debugging and feedback: Whenever a web page fails, it will output a precise stack trace. Furthermore, additional debugging tools make it easy to spot performance issues by simply adding debugging content to the web page. If an error occurs on the production web server rather than the developer's own machine, the server can send an email directly to the developer, containing full debugging information. In this way, errors can be discovered and fixed promptly. I found this very useful, as I would deploy new functionality that failed in its first user tests, but instead of debugging on the user's machine or logging into the web server, I would have all debugging information readily available in my inbox.

PDF handling

The reason for mentioning this, is that the PDF format is complicated and often prompts a lot of know-how and effort to produce in a parameterized fashion. Using XHTML2PDF[4], I found a way to generate PDF responses from the web server, which perfectly integrated with the rest of the DJANGO framework. Since the library needed pseudo CSS/XHTML structures as input, the usual DJANGO MVT pattern was applicable and able to feed the exactly the input that XHTML2PDF needed, harnessing from template inheritance, and various template generation utility functions. Put in simple terms, PDF tickets could be customized by simply editing a simple `.html` file.

Responsive layout

Responsive layout is almost impossible to create using bare CSS and HTML. The combinations of browsers and devices makes it impossible for a developer, and on the other hand there's no need to comply to the same standards in every application. Therefore, a lot of responsive layout frameworks have emerged, and the open source project TWITTER-BOOTSTRAP[5] is one of them, giving the developer a grid-based layout technique with widgets, highly optimized typographic settings, icons, and more. The framework itself is written on top of another open source project, LESS[6], a domain specific language targeted at efficiently generating CSS and enabling code reuse. Both TWITTER-BOOTSTRAP and LESS are applied in all

[4]http://www.xhtml2pdf.com/
[5]http://twitter.github.io/bootstrap/
[6]http://lesscss.org/

aspects of responsive layout, making development for several devices and browsers highly efficient, and minimizing the level of layout logic embedded in the HTML code, which thus becomes more of a container for data structures.

Development tools

I regrettably did not employ all known tools for rapid development, but these are examples of my own subset, i.e. the development environment and the tools that I enjoyed:

- Changing application code, spawns an automatic reload of the development server
- Altering data model structures can auto-generate database migration scripts
- Static web server contents (such as CSS files, images etc.) are automatically gathered and deployed in a statically served environment
- Changing and saving a LESS file automatically built a CSS file without invoking the LESS compiler.
- Mozilla Firefox made it possible to test different layout sizes and introspect and manipulate CSS and HTML structures, making it possible to experiment, sketch, and debug without altering source code.

Deployment One of the key features of Django is its focus on deployment which follows naturally from the rapid development aim. The use of automated *delivery* tools become key to the developer, and rather than maintaining control of a case-by-case based deployment process, the rapid developer gathers all these processes in scripted programs. This is also known as *Continuous Integration* and *Continuous Delivery*. The development process, however, lacked features of Continuous Delivery, such as automated test cases and automatic deployment from a master development branch in the version control system. The reason for this was indeed that my own habits fell short of writing test cases. Controversially speaking, I did not see the utility of such test cases or test-driven development in the settings of a rather small project that did not suffer from any immediate external dependencies or to deliver functionality or data to external interfaces.

3.3. Summary

Measured in terms of product quality, the development process has been successful in establishing a software system able to generate and sell tickets and support all known relevant aspects of the venues participating in the ticket coop. Meanwhile, the full scope of the project failed, as the ticket coop was never officially launched

within the time frame of the project. Participants remain committed to establish and launch the cooperative in the immediate time to come.

In terms of risks, I have mentioned the process of establishing the project, and the objectives and intentions behind. Following from this, we have seen that problematic structures in the organization have been dealt with continuously, and both the organizational structure and IT system have been adapted to resolve those issues. This was a gradual development, involving several meetings between management of both venues, and the technological insights from IT facilitation and steps from rapid development.

The greater number of decisions and discussions made through the development process have not been mentioned in this chapter, but the variety and quantity of examples should illustrate that a process of co-realization has indeed taken place. As with the description of the application, I have not mentioned all aspects, but sought to illustrate a variety of goals achieved and properties of the development process that are of interest to the successive analysis chapter.

In my experience, the outcome was not predictable, from which I infer that that the process has been complex enough to reach beyond the banal and to become subject of interest to the analysis. Unpredictable elements of the final organizational structure and application include the discarding of joint payment gateways and common financial administration. On the other hand, and perhaps more of a more innovative nature, participants were able to formulate several ideas and have them included and tested throughout the process. Finally, the ticket coop's IT system came to include and combine functions of two otherwise fragmented systems (paper and IT system, respectively), which I suspect from the two test cases can be of great benefit to both venues.

4. Discussion and Analysis

In this chapter, I present a discussion that focuses mainly on the insights from the case and relates them to the synthesis framework, *risky and rapid design spaces*. Moreover, I suggest improvements for a hypothetical future study. Since the framework is a synthesis, I first analyze the extent of which it already resembles components of the synthesis. Furthermore, I discuss how alternative methodologies contain elements that fall short of the case study and framework. Finally, I discuss the scope and applicability of the framework seen as a methodology and how the framework definition falls short in the sense of leaving questions open.

4.1. Overview

In order to address key questions and thereby draw a final conclusion in chapter 5, the current chapter is divided into 3 main parts:

Case study practiced as research: A reflection on the upper-level methods employed in the case study, and their shortcomings, i.e. the research methods, *not* the ethnomethodology targeted at the embedded design process.

Results of the case study: A discussion of the result of the case study in a broader scope: Did it cohere with the synthesis framework, what did it achieve, and could the outcome have been achieved through alternative methods?

Risky and rapid design spaces: Scope and applicability : Finally, I review the framework and discuss its role as a guiding methodology in order to see if it may contribute to software and organizational development in other settings than the case study.

4.2. Case study practiced as research

A major goal of establishing a discussion and analysis of *risky and rapid design spaces* grounded in the case study, is to see that the synthesized components address the outcome without missing substantial parts of the process. As the very first, therefore, we turn to the methods applied in the case study, and their effectiveness in addressing the synthesis framework (for a definition, see section 2.2). There are a number of shortcomings, which shall be elaborated in this section.

4.2.1. Research methods

Firstly, time constraints meant that it was difficult to maintain both IT Facilitation, development, and ethnographic studies all at once. I tried to address the necessity of concurrent activities of software development and ethnography as Bentley et al. (1992) stated (see section 2.3.1), but found a further time constraint issue of ethnography. As a result, the case study is a mixed reconstruction of my own observations and *experience*, highlighting those themes that I find relevant for the analysis, which in itself is problematic. Furthermore, as it took considerable share of the time frame to arrive at a full-scale usable platform, our design-in-use concept was only applied to limited system functionality, and therefore only gives as mere sense of what rapid development means. This could be seen as if the study just needed more time to develop, but expanding the study time-wise would not resolve issues of a single-person IT facilitator, developer, and observer. Requiring more effort from one role would take away resources from another, ultimately harming how the case study could perform as it is intended. For instance, if I was to spend more time observing and inquiring into the organizational structures, I would not be able to respond rapidly to issues raised by participants, and the day-to-day information flow of the RAD cycle would be lost.

Ironically and contradictory to the above, I was actually able to generate more data in my diary, than I could include in the outcome. This autoethnographic nature of the study and its biased prioritizing of issues alone calls for an objectivity check. Firstly, we need an external part to add such objectivity, i.e. someone who is not participating in the project. This supports the previously mentioned shortcoming by relieving the developer and/or IT facilitator from such tasks and puts the facilitator in a more natural position. Some of the interesting issues that could have been addressed through a more thorough debriefing or observational studies are elaborated in section 4.2.3.

Counter to the argument of third-party objectivity remains the insights generated by participatory involvement. Because of these insights, I find it reasonable to say that autoethnography has turned out as a highly efficient tool for reconstructing a complex process.

4.2.2. Case study practices

During the case study, I have come across deviations from the synthesis definition, or rather, methods that should have been applied to a larger extend but saw practices outside of the methodological scope take place. This relates to my own initial absence from a real situated context, but also that the participants often failed to respond and participate. There are several reasons for this.

As a typical case of *ivory towerism*, I was reluctant, especially in the beginning, to engage with the normal working day of the venues. Naturally, they have no office

space for a programmer and even worse, a programmer doing research. Because of this, I wrote the initial structures of the software assuming that it would be of limited benefit to situate myself in the everyday of live music production and promotion. Even worse, I thought that it would be harmful to productivity, being assumptive of the stability of design decisions and that they would remain unchanged by the involvement of any further context. I have chosen to depict this as a range of hidden decisions (see section 3.1.11) – a reconstruction of diary contents – but the speculation remains that they could have been avoided by engaging in a more situated context. This does not necessarily speak against the framework seen as a methodology but is to be seen as a flaw in the case study. Otherwise, the discussion should be from a point of view that developer absence from context and co-location is a natural element and should be thought as such – which is a limitation of this study.

The lack of participation is already described as a natural precaution taken by venue managements (see section 3.1.5). As the framework seeks to address design processes by acknowledging the risks and moreover targets such risky circumstances as organizational establishment, entrepreneurship etc., we should anticipate such precautious behavior. By debriefing with one of the venue managers, I found that such reflection on precautious behavior was accessible as subject of research and quite beneficial to the understanding of the process. To bring about more certainty and less speculation, a better case study should explore and reconstruct events together with participants afterwards, rather than only gathering data while regretting a lack of participation.

A less controversial and more obvious retrospective grievance has been a lack of version control commit logs during development of the software. As the plot in Figure 3.3 reveals, there have not been enough commits to create a smooth representation of the project's quantitative development. Moreover, commit messages could have told a detailed story of decisions made and cross-referenced to the diary. This could have been subject of a supplementary retrospective analysis (Krogstie and Divitini, 2010). As a practitioner of programming and a regular user of version control systems, I would have found it natural to explore the past through a version repository, and in this light, I find it problematic that it was not included as an explicit method of the study.

4.2.3. Missed points of inquiry

In order to get a better understanding of obviously contested core methods of the framework – co-realization, design-in-use, and rapid development – the case study and the background theory could or should have addressed further issues. Some suggestions are:

1. How does participation change once the design process is concerned with the actual product rather than abstract artifacts, prototypes or other subjects of

discussion?

2. Do participants perceive the difference between prototypes and the actual product, or is this distinction merely a technical one that the developer or IT facilitator should not concern participants with in the design process?
3. What strategies of IT facilitation can be employed? In Hartswood et al. (2002) *the broadest sense of the term* could be countered by more studies coming up with a set of strategical guidelines.
4. In what way does the speed of development affect the design process when there are no design artifacts and ideas are exchanged informally? I.e. just how crucial is the process flow to the risk of stalling and repetition?
5. How would participants have envisioned the ticket system without facilitation? I.e. in the case study, how would the design have been formulated by the venues independently of the IT facilitator?
6. Does the inclusion of non-technical users in hard technical decisions have any diverted effects?
7. How can we *avoid unproductive user-designer dichotomies* Botero et al. (2010)? I.e. which circumstances and methods bring about such problems for the IT facilitator when acting out the role of facilitating a design process in front of users?

4.3. Results of the case study

In the following section, I first revisit some key practices of the study that resemble a synthesis of co-realization and rapid development, and the design spaces that were available to participants and IT facilitation. This argues the coherence of the case study and risky and rapid design spaces.

After that, I revisit the project establishment process and discuss the goal of reaching the RAD cycle. This was perhaps the weakest point of the case study outcome and the overall discussion addresses whether it should be seen as a natural decision or consequence of my methods or as an unguided mistake.

The final parts of this section describe the complexity reductions and the added qualities that I perceive the development methodology has resulted in compared to how other methodologies may have ventured in this specific setting, e.g. agile methods and Participatory Design. This part puts special focus on the aspects of the case study that were successfully executed and finally claims that the software project itself has the quality of a larger production-ready project, credit to the use of development tools and software libraries applied.

Co-realization Throughout the process, I managed to place myself as IT Facilitator in the *broadest sense of the term* as described by Hartswood et al. (2002). Without using the term, it could be argued that I acted as project manager and developer, but with no economical stakes or attachment to the organizations. During all phases, I uncovered information to design and develop the application by participating in the organizational formation and meeting directly with management, and users at all levels. Moreover, I was able to inquire into and suggest changes to organizational structures of the ticket coop and ensure a consistency and interplay between organization and IT system.

Rapid development The software development process emerged without a requirements specification or an elaborate set of design goals, but simply an overall project description being discussed at meetings and then transformed directly into software artifacts. This is perhaps the strongest evidence that the approach resembled rapid development. Furthermore, those software artifacts were not prototypes, but as a developer, I consistently sought to continue evolving the same software platform with the need of greater software refactoring. This was aided by an incredible range of software development tools and libraries that supported the rapid development approach. What was lacking in this process, was time and resources allocated for design-in-use and other such refinements that could have been said to take place within the iterative cycle of design-development. Rather than refining the same functionalities and adding insights and participation, the iterative cycle of this particular project (see section 3.1.10) was mostly adding new features, with a few exceptions.

Design space From Botero et al. (2010)'s definition of design space, we see that they unfold two main points (see section 2.3.2), briefly summarized that design space is a exploration and co-construction through social interactions based on technology, and the design space is framed by a number of conditions, many of which are deliberate. In the case, we can envision the design space in a number of cases:

- The strategy of IT facilitation, i.e. that the facilitation has supported participants in their understanding of technology and actively inquired into design issues. This was the case at every meeting.
- Rapid development tools aided the developer to sketch and transform ideas into usable software, creating a much more clear and deep understanding of the system.
- Communicating the high affordability of rapid development tools, i.e. that ideas merely needed to be communicated informally and would then be sketched out in the product itself. In the most illustrative cases of the case study, it meant that ideas could be implemented and altered quickly thereafter.

4.3.1. Contemplating alternatives

In the following section, I contemplate the possibility for having applied alternative methods for design and development, seeing that the ticket coop was a construction of both social and technological nature.

Because of doubts concerning both technological outcome and the practices occurring from these, the establishment of the ticket coop took up considerable amounts of the project, and resulted in an unfocused and at times stalled emergence and establishment. Because of this, it seems unlikely that any top-down approach could have been applied. The organization supporting the software system, i.e. the ticket coop, simply would never have existed if rapid development or some other form of extreme programming had not been applied to uncover controversial and fundamental design issues. Furthermore, we can speculate that in our particular setting, it would have required additional resources to construct design artifacts and prototypes within the time frame, as rapid development resulted in a final product without reiterating or refactoring any significant elements. Such incidents could have argued that a preceding design process was in place, but they remained absent.

The mere technological platform (see section 3.2.2) of the development process highly resembled any other Extreme Programming (XP) platform, and many advocates may be identified for this isolated part of the project. However, seeing that there was no project management in the project, it is hard to say which other entity in the project could have guided the design. The lack of understanding of how, for instance, the venue managers would have designed the ticket system independently of the facilitation can only be speculated, but it is fair to say that none of them had any previous experience of creating IT systems and mostly sought to describe their wishes through perception of the systems they were already using. The innovative parts of the project occurred through facilitation or rather, a deliberate expansion of design space. Furthermore, since the process showed how the influence of IT facilitation at an management/strategic level was important to the establishment of the ticket coop, it is possible to say that the presence of facilitation at this level was even necessary[1].

Without any further elaboration, I find it fair to conclude that rapid development was the only realistic design and development strategy for this particular case study. Most real-life projects, however, need funding or some other guarantees or assessment of risk. This is where I find co-realization to be an indispensable part of the project. Since most established and well-defined methodologies have built a hierarchy (take for instance SCRUM), in which the programmer has been isolated from organizational and strategic planning, it is hard to access the overall grounds of decision making. Had my participation been secluded from the establishment of the ticket coop, it would have been likely that the rapid development of features

[1] I find it hard to completely exclude the possibility of an agile method targeting such flexible and improvised IT facilitation, yet I have not found one that fits.

had concerned a limited scope of the overall project. This criticism is also raised in Turk et al. (2002), in which they touch upon the problem of failure to reuse software functionality by generalizing *correctly* in accordance with the some social reality (see also the *Don't Repeat Yourself* principle, section 3.2.2):

> *Agile processes such as Extreme Programming focus on building software products that solve a specific problem. Development in "Internet time" often precludes developing generalized solutions even when it is clear that this could yield long-term benefits. (Turk et al., 2002)*

I can only guess, but in my own experience, the lack of proper access to organizational and social context and decision making, often means that these scenarios arise. Instead of seeking a top-down methodology, which is already argued as unrealistic for this case, we can avoid this pitfall by emphasizing this certain aspect of co-realization.

4.3.2. Reducing costs and complexity

From the knowledge gathered by performing the IT Facilitator role, I found it easy to access information necessary to make design decisions together with the participants to meet their needs. The final outcome exceeded their prior commercial platforms in terms of the user interface, functionality targeted at venue needs, and it even joined otherwise fragmented systems. Holistically speaking, this came at a low cost: Just a couple of months of effective design and development, and the system was able to go through testing without any issues or failures. This should be in stark contrast to what is achievable with a top-down approach, as illustrated by an ancient times COCOMO calculation, estimating the man hours needed to complete a software project of our size to be over 12 months (see section C.2).

Over two decades after Martin (1991)'s ground work on Rapid Application Development, the goals of reducing complexity and meeting user needs still support each other (see e.g. Martin (1991) p. 80 and recall Figure 2.4). Even though this ancient work on development methodology has many out-dated descriptions of concrete practices, the 30 recommendations for *I.S. Methodologies* (Martin, 1991 p. 80) still hold true to the vision of this study. Most design decisions could be taken in an ad hoc fashion, inquiring when needed, and letting users give feedback as they tested the system.

During the case study, I was able to do inquiries without any prior appointments or agenda planning. I could stay at the venues and work while people or users with expertise were available for resolving issues or questions at hand. As a counterbalance to constant inquiry or an overly problematizing design process, I could use induce my own resolution to design questions. When issues involved design decisions with little relation to organizational or social structure, I would turn to more hidden decisions (see section 3.1.11). As the project developed, I was able to improvise as I deemed necessary, which proved to be valuable, as the venues themselves were

dynamic or unreliable partners. Put in another way, the venues had a matching situated and improvised nature.

The software architecture following from rapid design frameworks meant that programmatic structures were made to adapt for future expansions and develop in an organic way. By keeping focus on re-usability, I was able to develop each function on the grounds of previous work (see section 3.2.2). This rationale does not necessarily follow from working from a fixed set of requirements, nor from working with prototypes and throw-away code.

Finally, with regards to the risks of development, the pieces really come together: If there are risks involved in the match between unknown organizational consequences or technological complexities, it seems like a viable strategy to explore both at the same time. In some cases, we may see rapid prototyping as a better option, being a close relative to rapid development (for a general discussion, see section 4.4.4), often building on the exact same software tools and frameworks. In this study, the need for prototyping was eclipsed by the certainty of supporting structures which made it realistic to lay out a foundation and keep design experiments and sketching at a production level.

The view is supported by Crabtree (2004) with the demand for ethnomethodology to be properly incorporated into design and construction phases, what they see as a *hybrid mixing-pot* of design practices and technological development. Something that Co-realization and Rapid Application Development covers for very obvious reasons, i.e. design-in-use.

> *This is a highly economical and efficient use of what is often considered an expensive and time-consuming approach. It requires only short periods of study. (Crabtree, 2004)*

The adaptive and improvised nature of the development process can thus be seen as cost efficient. The very minimal methodology guiding the process and the very simple and free role of IT facilitation still acted out a hybrid mixing-pot of design practices and technological development – in this light, it can be said that the synthesis development approach had managed to reduced complexities which would otherwise be imposed by the multitude of more structured methodologies.

4.3.3. Technological enablers

The case study can be seen as a bridge between contemporary technological practices and development methodology, although we may find that in some cases the methodology creates the demand for such tools. Even critiques of XP practices are acknowledging the technological discourse (see Turk et al. (2005) quote below). But their fear of heavy refactoring was empirically speaking never a real risk in this particular case study. Throughout the facilitation, I did not fear a surprise hat-trick of new demands, rather I tried to pro-actively inquire in cases where refactoring

was a risk, for instance regarding permissions and authentication. This was also the reason why the data models were laid out (see also the section on my own biased habits of such, section 3.1.7).

> *This assumption allows developers to do less than thorough analysis and design in the early phases and, instead, make improvements throughout the course of the project by refactoring the code. There is no objective evidence that this assumption is valid in general, but it can be argued that the cost of change curve can be flattened by using reusable design experiences in the form of architectural and design patterns, and capitalizing on new technologies supporting rapid program development (e.g., libraries, components and frameworks, and more powerful compilers that enable short and incremental compilations). (Turk et al., 2005)*

Summing up on the discourse set out by early likes of Martin (1991), the study testifies of a development that has greatly reduced the complexity and cost of the design process through technological enablers and an understanding and deliberate use of these.

A last point to be made regarding technology for rapid development, is the possibilities which were *not* reaped. First of all, I did not use automated test cases, which could have been beneficial, had the project increased in complexity, but also to ensure its future development. Related to this, but not that test cases are fundamental to such a method, is the idea of Continuous Delivery and Integration. I developed the project in a simple code base, which could have been split into production and development, automating processes of delivering the production ready features.

4.3.4. Participation vs. non-participation

In the following section, I observe the choice between participation and non-participation. Core to the role of an IT facilitator is the choice of inquiring into specific cases of design, the agenda setting of meetings, choosing what to observer, and the overall usage of PD methods. In this elaboration, these are all seen simply as *participation.* As the study did not compare any levels of participation or establish some range of choices or alternatives, it leaves a room for discussing and hypothesizing any such practical guidelines. On the other hand, the study was a perfect example of *not* formalizing or intending any specific levels of participation. In any case, I pointed out deviations from the synthesis framework (see section 4.2.2).

In order to enlighten the discussion of costs/benefits of participation vs. non-participation and guide the IT facilitator in this choice, I have devised a very simple list of classes of situations that speak in favor of each (see Table 4.1), but the final decision will most likely be a mixture of all, i.e. a gray zone.

With these guiding principles, I do not seek to establish anything more than a mere call for intuition. Many of the choices will become impossible, and from the

Participation	Non-participation
1. Uncertainty of social or organizational behavior	1. Fair certainty of social or organizational behavior
2. High value or consequences for organizational objectives	2. Little relevance to organizational objectives
3. Lack of understanding of organizational consequences	3. Conventional solution exists
4. Necessity of understanding technical issues: Behavior and development of user instructions	4. High technical value: Security issues
5. Added value of explaining technical choice and/or low cost of implementation	5. Technical choice with high cost of exemplification and explanation

Table 4.1.: Deciding between participation and non-participation: Guiding principles drafted from the case experience.

experience of the case study, almost all of the choices of non-participation (see the list of hidden decisions, section 3.1.11) could have been made partly more visible to participants. Although the nature of the case's non-participatory aspects was mostly technical, we should still ask to the result of of opening up the discussion to the users. Mainly because this could guide us towards a *purely* participatory approach and relieve the IT facilitator of hard choices.

From the lack of a real in-production iteration of the RAD cycle, I found that users were at times reluctant to give any real feedback. This is strongly supportive of the Table 4.1 guiding non-participation principle 2. In order to create statistics of ticket sales, I used my own perception of what might be necessary, of course guided by previous participatory activities, but not attempting to inquire any further into the specific ideas of visualizing statistics. I found after the test runs of two ticket sales that the users had nothing further to add to this, emphasizing that some cases need more usage (with real data!) before any real design-in-use can take place.

An understanding of the choice of a participatory elaboration of design decision in the continuum of development is fundamental to co-realization. The design space is thus not always expanded by setting out a participatory discourse if this becomes a limit or hindrance of utilizing the IT facilitator's own judgment or intuition.

4.4. Risky and rapid design spaces: Scope and applicability

In the final section of this chapter, I turn to a broader discussion fueled by the analysis of the case study. This includes criticism of agile methods and Participatory Design, keeping the nature of the case study in mind and drawing upon the analysis of the previous section. Onwards, I point out the most critical issues from the case study if the synthesis framework was to guide other software projects, and try to forge relevant criticism to build a case in favor of *risky and rapid design spaces*.

4.4.1. Design-in-use; Co-realization, PD, and agile methods

A number of alternative methods could have been employed in the case study, or was in fact by definition employed (see section 4.3.1), and we pickup the discussion of these to draw upon the existing body of criticism that has been raised in former works. One of the major discussions would be the question of where design should place, or rather where and when to upon up design space, recalling the criticism of bounded design in section 2.3.2.

Co-realization and agile

Risky and rapid design spaces is targeting a development strategy that is incremental and to some degree acknowledges an iterative process. These are elements of agile methods that were initially inspired by RAD, and thus not very difficult to show as highly similar. But what about Co-realization? Since Co-realization sees users participate in system design, adding situated guidance from the developer, two very specific entries of the Agile Manifesto (Wikipedia, 2013a) would seem very similar:

- Close, daily cooperation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)

Adding guidance from the developer without any contextual placement would both violate Co-realization's principles and the Agile Manifesto. It is hard to say, though, that co-location is the interpretation of many agile methods, for instance SCRUM, which deliberately targets distributed development by separating development processes into a hierarchical team structure.

Co-realization is also (indirectly) supportive of agile methods, in the sense that it objects against the participatory design scene for not moving into the realms of design-in-use:

> *With few exceptions, the focus within participatory design projects seldom moves beyond the design phase or the construction of early prototypes, and onto development and use (Hartswood et al. (2002) quoting Dittrich, 1998).*

But this is perhaps where the similarities end, because the Agile Manifesto does not address any organizational consequences of technology, and as such it fails to meet one of the core criticisms of Hartswood et al. (2002). Symptomatic of this, agile methods need an *acceptance test* allocated as a final step before deploying a new feature, the necessity of which must surely be closely related to the absence of user involvement during coding, unit testing, and system testing. Apart from the nature of acceptance tests, we can discuss their location in a design and development flow: Users should be able to agree to and *accept* changes to as far an extend as possible. However, by explicating the location of such tests, we would loose out, both on smaller design issues or holistic acceptance, as agile methods seek to modularize and decouple the experience and acceptance testing. This is highly analogous of bounded design. By means of IT facilitation and an improvised attitude, we can add acceptance tests anywhere *necessary* (see the discussion of participation and non-participation, section 4.3.4).

Co-realization, participatory design, and technomethodology

The criticism from Co-realization towards participatory design takes offset in the practice of these design activities and that they seem to be too supportive of IT professionals and their perceived agendas. The criticism especially targets that the methods of PD are abolished once construction begins, which is something we can be argued to be to the consequence of a heavily managed geography/time dispersed agile process where user input is discarded in favor of management and business.

> *We must conclude that, despite its declared intentions, participatory design continues to privilege the role and expertise of IT professionals over that of users. (Hartswood et al., 2002)*

This is not, however necessarily an unintended outcome of PD, and we need to take into account the very specific arguments by the PD advocates that have deliberately chosen to prioritize an initializing design phase before use:

> *We do acknowledge of course that there are also design activities later on in a systems development process, and that users find new ways of utilizing an application after it has been put to use. Further, they might come up with additional demands, which in turn leads to "continuing design in use" (Henderson and Kyng, 1991).* ***But for the type of IT applications addressed here****, such later design activities do not eliminate the need for a good first approximation. Rather, we see a competent initial design as one of the prerequisites for such subsequent design activities to be successful endeavors. Our method is intended for these early design activities. (Kensing, 2003)*

First of all, participatory design ad Kensing (2003) does not overlook to the opportunities that arise from design-in-use. Secondly, PD is open to its later adoption,

but thirdly, PD methods are not intended for this stage. The latter is quite surprising as there is still much benefit to be gained from participatory design techniques, *especially* after an IT system has been implemented. PD techniques at such a stage would merely give the discussion more ground, for instance such that users could create more insightful ideas from their understanding of the system they have been exposed to. And perhaps this is where PD falls victim of a wrong perception of the nature of the IT applications that are relevant, i.e. that it intends to limit design processes for some *type of IT applications addressed* which probably does not know of the technological development tools that hold the qualities of rapid development.

Furthermore, and to add to the criticism of PD's self-inflicted limitations, PD activities in a contextual environment can establish a way of thinking or culture that paves the way for day-to-day innovations. Requiring participation in a Co-realization and rapid fashion also requires that participation takes place on a rapid basis and steadily re-occurring fashion. It would mean that the IT Facilitation and users could realize the close coupling of their goals: To *gather* enough information to create successful system design, and to *deliver* enough information for successful system design.

Where we can see Co-realization as a mechanism that pushes methodologies such as PD further into the development phases, uniting users and developers even closer through IT facilitation, there is a need for a more direct address towards the development of new functions and new products. When an IT system is not in use and its functionalities are radically different from preceding systems, it becomes harder to gain qualified insights. Real technological artifacts arising from rapid development help to establish this understanding. But using an undeployed system, we would be studying a hypothetical usage, subject to a dynamic context, which would ultimately be altered by deploying a new IT system. The case study has argued that design-in-use became a necessity in cases where participation was clearly lacking due to tentative deployment, missing real life test data, and direct stakes for the users giving feedback about system functionalities.

> *Co-realization involves: attending to the evaluation of technologies; appreciating the benefit of active user participation; adapting to a particular organisational setting; the explicit connection of studies of work and system design; and commitment to a 'long-term engagement'. Hartswood et al. (2002)*

The strategy to adopt from Co-realization, should be that of deploying a system as quickly as possible in order to reap the results of the *explicit connection of studies of work and system design*, keeping in mind that a commitment to follow up on this is critical, and deployment (on whatever level[2] or sub-system) is not an achievement in itself, or as Crabtree (2004) puts it:

> *Recognition of the real-world uses of ethnomethodology in design practice opens up the possibility of devising a hybrid methodology that actively supports the invention of the future. (Crabtree, 2004)*

[2]see for instance *canary channels* or *canary testing* and *user-driven testing*

Participatory redundancies The case study did not make use of a range of PD tools and techniques from the *in-depth analysis* and *innovation* phases. Although the study did make use of observational inquiry (during a different much later phase), the study did not invest resources in any further pre-development design. Why draw? Why act? Why have workshops? As the previous discussion states, if the affordability of sketching software is very low and reflects the realistic realm of the software developer's skills, these activities may not be crucial to the outcome. They rather reflect an economic model, in which the developer of the software does not even exist after a project description and legally binding contract has been signed. In light of the progression of rapid development technologies, the need for PD tools can be discussed in terms of resource allocation versus the qualitative outcomes of their alternatives, i.e. rapid development and rapid prototyping. In this discourse, the software designers and their organizations can observe PD tools in the in-depth analysis and innovation phases from a utilitarian perspective and such planning as *nice to have*.

4.4.2. Project establishment

Firstly, what is meant by *project establishment* in this section, is the project work done prior to any particular design of software. In the description of Participatory Development, Kensing (2003) lays out project establishment as a negotiation process in which common objectives and structures of the succeeding design process is established. However, there are no limits to the obligations made in the project establishment in PD. In the case study, I found that there was a lack of not only organizational structure, but even the a set of common deadlines, obligations, and identification of roles.

As an IT facilitator, I did reach a satisfactory level of established structure. But the establishment process was not explicit (rather it was a posteriori reconstruction of the overall process), nor did it set a deadline or impose any sort of obligations from the participants. This did not go easily, as there were many cases where the project was either delayed or space was left in which participation could have occurred. As such, this is not a one-sided critique of PD, but an acknowledgement that the lack of structure left room for improvement. Counter to this and PD's idea of the necessity of an establishment phase, is the fact that the project slowly gathered more support from the participants as it yielded results. The case study thus showed that it is possible to re-negotiate obligations and promises of participants as the project evolves and the risks perceived by users, management and IT facilitator are eliminated or more clearly assessed.

On an organizational level, the case study suggested a connection between the co-operative's statutes and the system development phases: Once the system existed, management was willing to take further decisions, and resolutions to fundamental problems were found.

In light of the circumstantial outcome of the case study and a lacking understanding of what *project establishment* should really mean, the results at least show that the understanding of what project establishment means should *not* exclude concrete development of software artifacts. Instead, it should acknowledge that new software artifacts can influence objectives, organizational structures etc. In order to convey this in a meaningful way, I say that project establishment should be *minimized*. In this manner, I do not mean that there should be a specific phase with a planned time limit, but that perhaps the IT facilitator should seek solutions where software artifacts can help guide important decisions, rather than seeing such decisions as fundamental and preceding software development.

4.4.3. Assumptions and limitations of agile methods

In two highly related papers, Turk et al. (2002) and Turk et al. (2005), the authors go through a number of limitations and assumptions, respectively, of agile methods. In this section, I mention a number of ways that the case study addresses these, moreover how they are addressed in a wider perspective by *risky and rapid design spaces* and its more direct approach to design-in-use.

By dealing with the assumptions of agile development (see Table 4.2), the case for risky and rapid design spaces is founded in strategic manner. The perception of agile methods is highly guided by the agile manifesto and the more rigid structures that it imposes on development methodology. Co-realization does not deal with this in a very direct way, but once adding rapid development to the mixture, we can see the IT facilitator role as a remedy to many of the assumptions, although we need to directly discard of both the idea of distributed development and team coordination. However, this follows suit to the intention of small, entrepreneurship-like scenarios, which does not necessarily set the scale for the software project (remember Google started in a garage).

4.4.4. Prototypes and in-production artifacts: Convergence of design

Risky and rapid design spaces does not advocate the abolishment of prototypes, but simply argues a case where they may not be necessary, and suggest that rapid development tools are perfectly able of creating a transition between in-design and in-product. Suggesting that the continuously improving software artifact which is exposed to the user during Co-realization is *just* a prototype would be wrong, but suggesting that it is *the product* would similarly convey a wrong impression as it might as well be removed, re-done or radically changed.

Convergence already exists In reality, many open source projects have found a convergence between prototyping and rapid development. They evolve code-first,

discussion after. For instance, if we observe the development nature of the open source tools that this project is based on, most of the development is done through constant re-iteration, in which patches are sent in, discussed, and merged into the development branches or further modified. In this sense, the same artifact first performs as a prototype, then as a product.

In the discussion of how ethnomethodology should subject real technology to user's design inputs, Crabtree (2004) consistently avoids using the term *prototype* and instead gives a full description of the design phase without hinting as to whether a design artifact should be discardable or transferable to the final product, instead they talk about a loose transition from *quick and dirty* to *further technological exploration:*

> *Furthermore, in advocating the exploration of topics seen in the breach through quick and dirty study rapidly followed by further technological exploration, it is a strategy that puts technology at the centre of things. It is a strategy in which technological innovation is driven incrementally through the development of technology and the subsequent study of its essential social properties, and so provides for the development of future technologies that are well grounded in and responsive to the social circumstances of their use. (Crabtree, 2004)*

With the prototypes refined and re-iterated, one could argue that the final prototype could be considered the final product. However, I have not found any works during the study that seemed to support such an idea, and for obvious reasons. In order to distinguish the terms, we see prototype's main property as being discardable in favor of a full-quality implementation.

Through discussions with other developers of the DJANGO framework, it was especially argued that this framework was their champion of rapid prototyping. One lead developer at a larger global web application company advocated that prototyping should be used whenever possible, but within limits. The notion of prototyping vs. the final product was easy to deal with by *always* throwing away the code, because the doubt-filled choice of writing *real* code vs. *prototyped* code risked leaking prototype intended code into production. To the lead developer, prototyping was especially essential because "code wins arguments" was a view held in this company.

From the view of sketching designs, and however we may implement PD methods, it should be clear, that this process cannot continue forever, ie. we cannot continue to refine our prototypes and design documents. But many projects may not be able to emerge into the a construction phase without some guidance or informed design decision. Using the case study, however, I have sought to find an empirical base to say that *not all* systems need such guidance.

The inevitable notion of iteration, however, results in a restraint on the design phase. Both in theory and practice. In theory, it means that we have to redeploy a new understanding and then re-gather results within limits to which there exists

an upper bound, and in practice it becomes a decision to be made by someone, for instance the IT facilitator, management or some user acceptance test.

Using risky and rapid design spaces is definitely to be contested on the perception of what costs and complexities are. Even the tools that it relies on are used for many purposes, one of them being rapid prototyping. The possibility of rapidly developing does not only apply to design-in-use but also project establishment phases. In this setting, it is common to use prototyping in order to explore technical aspects of a solution. But even in later phases of development, i.e. after deployment, rapid prototyping is still used to explore new features and changes to be added.

The fact that the same tools are used both in design-in-use settings and rapid prototyping testifies of the overlap between these two methodologies, alleviating us from the burden of dogmatic choice or distinction.

4.4.5. Risks

Entrepreneurship can both be seen as a genre of organization or as some innovative goal of an organization. The scope of the the case study at hand resembles very much that of an entrepreneurship (see also the background pre-analysis, section 2.6 and the analysis of risks in the project venture, section 3.1.5), moreover that of an e-entrepreneurship. For instance, in the case of music venues, they have transformed most parts of their ticket sales and promotional functions into the digital realm and their business functions will rely greatly on the ticket coop's IT system.

As stated before, we are discarding conventional aspects of the establishment of software development, such as a set of requirements, but even the sort of pre-analysis to guide software design suggested by PD methods. Rather, we prefer a common understanding of fundamental goals, and thus the whole nature of the project requires risk-willingness, and in our case these risks were never estimated or explicated. This is not, however, a bad thing, as such assurances may in themselves be a hindrance of change:

> *First, there is the problem of inflexibility; many companies find it difficult to change. This may not be true of small, creative organizations but change will not be easy for those in the global market category. Organizational change is often stimulated and reinforced by companies that take the lead and are prepared to take risks. (Mumford, 2003)*

The assessment of risks through pre-analysis can have a considerable size compared to that of rapid development. Once development begins and participants gain a better perception of risks, the latter can be deconstructed and re-assessed. Because of lowering costs of development and the possibilities of integrating development and assess its organizational impact, the alternative to pre-analysis can in theory have a lower cost. And as the case study shows, also in practice. Moreover, as systems have organizational consequences and technology changes rapidly, there is a risk of uncertainty by trying to preempt risks through pre-analysis.

Addressing risks through rapid development During the study, I found (see Risks, section 3.1.5) that facilitation was necessary on one hand to guide the participants when insecure about technology-related risks, but also as a developer to sense the depth of risks on an economical and organizational level such that inquiries could be guided and prioritized by this. In the case study, the ability to quickly introduce new solutions based on managements' perception of a risky issue, served to re-assure that issues could really be handled. As an IT facilitator, it was possible to target rapid development at issues that brought about concerns of risks in order to solve or move on from such issues, because many aspects were speculative and nourished by lack of a concrete digital artifact.

4.4.6. A general recommendation for future work

Guided by a vision that *technological innovation is driven incrementally through the development of technology and the subsequent study of its essential social properties* (Crabtree (2004)) and the fact that the technology that is supporting these processes is changing as rapidly as we can design and develop our own new technologies, we should *refactor* our design process in that very same fashion. By this I mean that future work should both adapt to the increasingly rapid development tools and make suggestions of how they can achieve aims to support the needs of designers and developers seen as a collective entity.

4.5. Summary

To sum up this analysis of the case study and discussion of the synthesis framework's applicability, I present a list of the most important conclusions of the discussion and analysis.

The Case Study: An organization and an IT system simultaneously adapted to each other. Within a short span of time, a promising IT system was developed. The case study closely resembled the synthesis framework, and the outcome shed light on interesting properties of choosing this development approach.

Risky: The case study was relevant as it contained risk-willingness in an entrepreneurship-like fashion. We saw that rapid development helped to resolve and reassessing risks during the construction process, such that the riskiness can be said to decrease as actual software development occurs. In our discussion, I emphasized that risks are not necessarily assessable through pre-analysis, especially since affordance of rapid development is high, making pre-analysis unattractive.

and Rapid: I have discussed how to minimize the establishment process and why this is critical. Furthermore, I showed how, in the case study, I could efficiently and iteratively add new features while pro-actively seeking inputs through IT facilitation and utilizing contemporary rapid development tools.

Design spaces: By studying and including the organizational context, adding participation needed to guide the optimal outcome, using the affordance of rapid development, the term of design space was explored and beneficial to the analytical process. The outcome showed that design spaces were not necessarily maximized in some ideal sense but rather circumstantial and constantly shifted. Not least, they were guided by IT facilitation and expanded by a user-developer context shared through means of co-realization.

From the overall research methodology applied, we see that there are still many aspects to be covered and explored, both empirically a analytically. I have presented a list of the missed points of inquiry that would serve to enlighten the discussion further (see section 4.2.3). Having analyzed the outcome of applying risky and rapid design spaces and its intended scope, I find its synthesis components worthy of further exploration.

Assumption	Risky and rapid design spaces (as a strategy)
Visibility	Working code is delivered through possibility of rapid development tools and methods. This is done through a proper delivery mechanism. If such does not exist, there should be no claim of a proper design space.
Iteration	Through project establishment and mutual understanding, the project arrives at a construction phase that makes iteration possible either by design-in-use or IT facilitation
Customer interaction	IT facilitation should actively decide between participation and non-participation, having full access to and acknowledging the overall strategy of the project
Team communication	Development is always co-located, the project never assumes a size where teams are dispersed in time and place
Face-to-Face	In cases where face-to-face interaction becomes unnecessary, the IT facilitation may choose other methods of communication or decision making
Documentation	There is no intention of deferring documentation, rather there are tools to make the production of documentation as rapid as the development itself.
Self-Evaluation	By extending an understanding of organizational objectives and values and by means of Co-realization, the need for self-evaluation becomes largely redundant or as critical as any other self-evalutation.
The Quality Assurance	There is no restriction on the scope of testing or fragmenting of the subject of tests. Through rapid delivery, the system should always be perceivable to its fullest extend. If not, IT facilitation should schedule later acceptance tests. Original assumption: *Evaluation of software artifacts (products and processes) can be restricted to frequent informal Assumption interviews, reviews and code testing.*

Table 4.2.: Turk et al. (2005)'s list of common assumptions in agile development: These are the responses of the study.

5. Conclusion

Firstly, the study presented theoretical grounds of a more direct approach to software development, for instance how design and development can be perceived as an iterative continuum. This included an understanding that software design should adhere to its implications on organizations and social reality, and vice versa.

Through a specific small-scale project, I have applied a synthesis software development practice of co-realization and rapid development. The outcome of this process is a description of the concrete, situated use of the synthesis, while making use of design spaces for analytical purposes. Furthermore, because of the particular successful application of the synthesis, I have discussed it in a broader context. The initial definition of *risky and rapid design spaces* was thus subjected to a practical and analytical test.

5.1. Findings

Firstly and most fundamentally, the *risky and rapid development* synthesis was carried out in a real environment and resulted in real software. This process showed an improvised and less rigidly structured development process, reliant on co-realization's concept of IT facilitation. Through experience with the facilitation process, I was able to exemplify how ethnomethodology was useful and design information was easily obtainable to the IT facilitator.

The outcome of the case study showed an interplay between the IT system development and its users and organization. This included an ability, as IT facilitator, to observe and influence organizational aims and practices and design the system with an anticipation of its sociotechnical nature. Furthermore, it was possible to adapt organizational practices and IT system functionalities concurrently, and that this practice relied on participation from the IT facilitator in the shaping of the organization.

By observing my outcomes and discussing the role of design and development as a continuum, I have shown that design as a separate, preceding activity of development can be made redundant under the right circumstances. Furthermore, that design decisions may arise from artifacts that are instantly deployed or easily transferable to production as opposed to prototypes and other abstract artifacts.

My findings show that it is possible to use IT facilitation to guide participation using a number of ethnomethodological activities. The ethnomethods were closely linked

to rapid development, which can only take place as a result of the technological enablers attributed. By gaining information for design decisions by playing the role of IT facilitator and at the same time utilizing rapid development, the development model was simplified and efficient, yet the outcome indicated its qualities through successful testing. The choice of IT facilitation, however, results in a dependency on the facilitator's technological tools, but conversely a design space occurs by putting them to work using a rapid approach.

5.2. Limitations and future work

The case study presented the outcome of a very specific setting. Decisions were circumstantial, technological tools were specialized, IT facilitation relied on a single character etc. Thus, turning over the development approach of the case study to a broader perspective and creating a methodology from the discussion and results, is not straight-forward. The role of the IT facilitator needs much more elaboration, and the problem of project establishment needs more guidance. Once the iterative process of rapid development has arrived, the findings indicated a more straight-forward process which could yield results that are already available from other studies. However, the path of minimizing project establishment or pre-analysis to make room for rapid development remains sparsely described.

Whereas it might be easy to suggest a set of methods that work under some perfect social and organizational settings, it would be harder to generalize and transfer these settings to a useful category. I have tried to define the likes of entrepreneurship with respect to risk-willingness, but there are other organizational and social factors that need elaboration.

Since I was working with online open source communities and made use of their products, the project was in fact influenced by practices shared in this global community. Seeing IT systems and organizations, such as this case study, as a consequence of a global culture of sharing knowledge and software is also highly relevant. It could be embedded in the IT facilitator's persona, but since this persona is under the influence of these global trends and the technologies they produce, we should be highly alert that this has a great deal of impact on projects that choose this path – of course, most likely a positive kind of impact.

5.3. Recommendations

This study was motivated by a need for a supportive methodology to my own practices. Likely, there are more areas of the technological and innovative community that are in the same situation.

The research was highly autoethographic, collecting all observations in a personal diary, through memory, programmatic artifacts and their version control system. I found the role of participant and researcher a bit distracting from a natural behavior, but more critically, I was unable to observe the process from outside. There is a need for further studies of the IT facilitator role in combination with rapid development. Great insights such as this study can be gained from playing the part as both researcher and IT facilitator, but an observing counter-part could easily add to this while addressing many more issues.

With that said, I am sure that both the socio-technological (STS) and software development arenas and can greatly benefit from technological practitioners acting out an autoethnographic role to uncover the social nature of their work.

5.4. Epilogue: A bit of normativity

Seeing that software development methods such as Continuous Delivery are gaining support from more and more efficient technologies, software development can benefit from this new affordance by moving closer to the social realities that it plays part in. Aided by more efficient delivery methods, the fields of agile and rapid development help to bring about iterative cycles of design-in-use, uniting the developer with users and organizations in an increasingly closer bond. From my own experience in this study, I find that work is needed to explicate and promote the role of the developer in the building of organizations and their strategies. When technology is the main tool, value, and trade of an organization, the developer needs to be able to participate at a the highest levels and not to be managed and distanced by software methodologies. By failing to acknowledge and target this, software development methodologies will ultimately fail to support a socio-technological reality.

Acknowledgments

Thanks to everyone who gave me an education, Computer Science dept. at Copenhagen University, Computer Science dept. at Utrecht University, and the IT University of Copenhagen.

Thanks to the Django community for being AWESOME!

Thanks to Github for having built the greatest open source sharing platform imaginable.

Thanks to the venues who participated.

Thanks to everyone, especially all my roomies, who sent me smiles while I was self-indulged and preoccupied with this report.

Thanks to mor & far, and rest of family.

A. Communication and data

A.1. Project plan

In an email dated February 27th, the following time schedule was laid out. Notice that the first test was scheduled for the 13th, then moved due to venue issues to the 17th. All-in-all the plan was delayed by 10 days.

Plan for February 21 - April 3:

Phase 1 (21/2-27/2): Purchase process (already reviewed)

Phase 2 (28/2-3/3): Management tools, statistics, event reports

Phase 3 (4/3-10/3): Door check-in, barcode scanning, printed ticket format

Phase 4 (11/3-17/3): Fine adjustments, visual identity, texts (optional, eventually never happened)

Phase 5 (18/3-24/3): Creation of payment provider subscriptions, payment gateway interaction, bank accounts, virtual private server (constitution and general assembly pending, so did not happen)

Phase 6 (25/3-3/4): Testing the full system (happened, but without payment modules)

A.2. Project description (attached document)

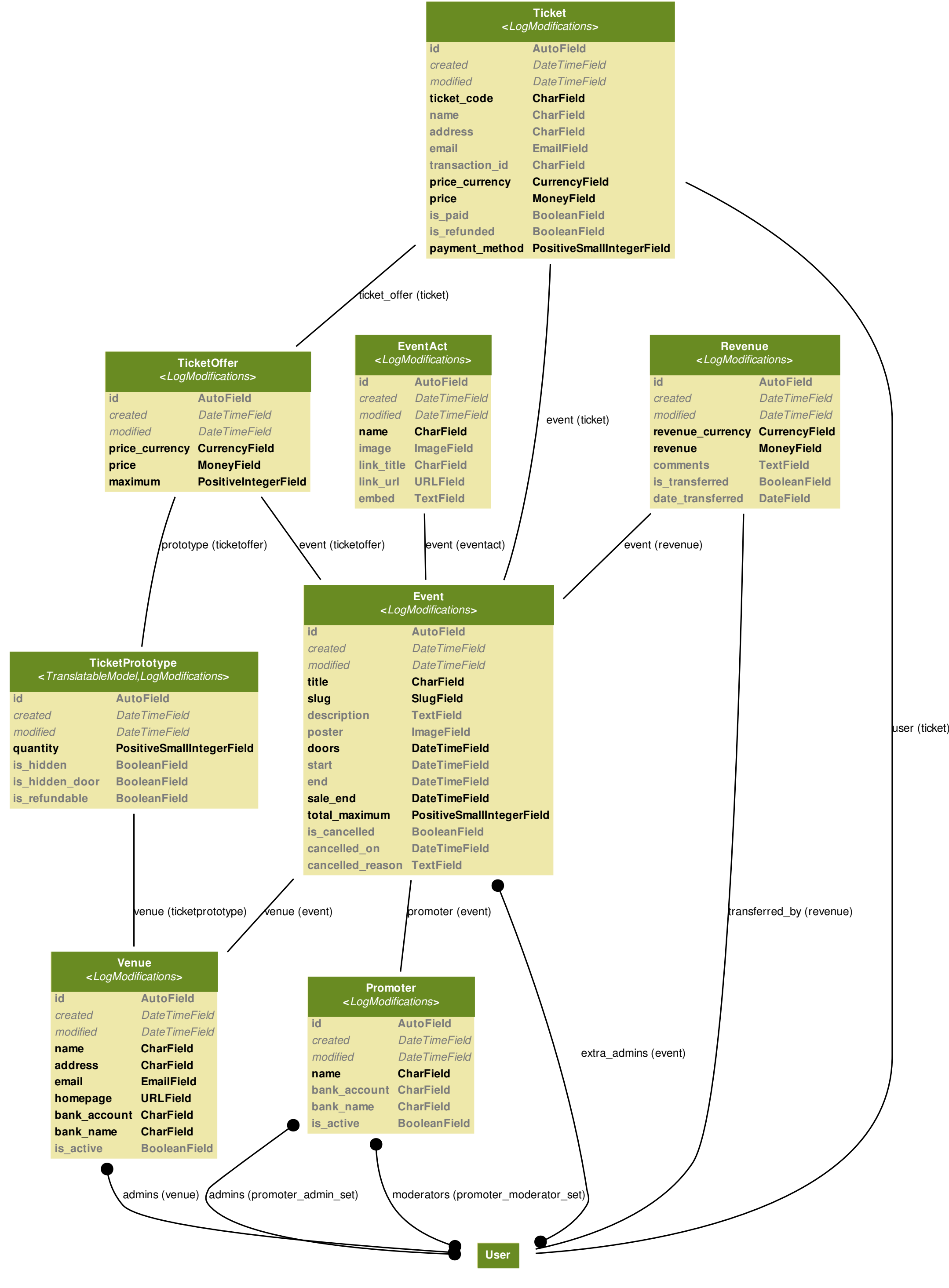
An 8-page project description, drafted, shared, and revised in November/December 2012, is externally attached to the hand-in of this thesis.

A.3. Diary (not attached)

The diary exists as a 31 page document spanning some 37 entries, in Danish and English, obtainable by request. It is a highly reflective autoethnographic work, every entry was written either situated or shortly after events occurred, with exception of the first months of the process, which were written just before the rest of the entries started flowing in on an approximated daily basis.

B. Application

B.1. Data model, first revision (February 21)



B.2. Function list

The following list names and describes key functions of the ticket system, all of which resemble a piece of web user interface. The programmatic details are abstracted, since they do not serve any specific purpose to the analysis. All pages are made with responsive layout techniques and function on both mobile, tablet and PC devices.

All pages are fully translatable (currently Danish and English) and localized – for instance, currencies can be changed, and dates are displayed in local format.

B.2.1. Purchasing / frontend

Function	Description
`Event list`	Lists all events for a given venue, descriptions, dates and sales status (sold out, few tickets left).
`Event view`	Displays an event with description and a buy form, from which users can select the number of tickets desired.
`Purchase initialize`	Creates a temporary ticket in the system and redirects to the purchase cart form. If the event is sold out, it redirects to a sold out message. If the event is temporarily sold out, it redirects to a screen telling the user to try later. If the concert is canceled, it displays a cancellation message.
`Purchase form`	Displays a list of tickets added to the purchase and a form to fill in name, email, address and phone. Also displays optional fees.
`Payment form`	Displays the payment form or redirects to a payment gateway.
`Ticket download`	Is display through a unique, reusable URL which is also sent via email. Displays the option to download the ticket. If more tickets are bought, it displays an option to download all tickets in one PDF. Also displays social media widgets.
`Ticket PDF`	Displays the name of the event, date, address of venue, description of event, price of ticket, name of ticket holder, Ticket ID, barcode, and a list of related events. If more tickets are requested, it displays all tickets in a compact list with big enumeration of each ticket.

Function	Description
`Ticket email`	Displays the name of the event, date, address of venue, price of ticket, and the unique ticket ID. Can be brought to the venue in place of the ticket PDF. Also contains the PDF as an attachment.

B.2.2. Management / backend

Function	Description
`Login screen`	Username and password form. Users are afterwards identified by their role, such as user with management rights for Venue A and/or user with rights to view/change financial figures.
`Event list`	The main view of all events, future and past. All columns can be sorted: Title, date, and sales figures. Search field to filter by name of event. Every event has a status indicating if all required fields have been filled in, and if the event is published. Sales figures are shown with easy color codes to indicate if they represent a positive development. The list is paginated, and each page has the option to display a graph showing ticket sales of displayed events compared. Every event has a small menu: Settings, social media, sales, print ticket list, show tickets, report
`Venue settings`	Displays a form to set the data for the venue: Email, address, homepage, logo etc. Displays a form to add new users and to change user permissions.
`Venue ticket prototypes`	Displays all ticket prototypes created for a venue, and a link to create new prototypes or delete/edit existing.
`Create/update ticket prototype`	Form to create/edit ticket prototypes: Name, description (or usage instructions for ticket buyer), quantity (how many people are allowed entry), template (indicates that the prototype should always be present at newly created shows, and that it has a price of '0'), online presale (indicates that it can be purchased online) and door sale (indicating that it will be shown as an option on the door screen).

Function	Description
`Statistics`	Main dashboard for viewing the progress of future or unfinished ticket sales. Contains overall figures for the venue, history of the past 30 days, links to recent reports, and the a graph of the latest sales, and a graph indicating ticket sales the past month.
`Ticket list`	List of all tickets. Can be filtered by event name, payment status, and free text search by name, email etc. The list can also be sorted. Actions are available for each ticket: Change the ticket data, resend the ticket email, and refund the ticket. The full list can be displayed as a PDF with barcodes.
`Revenue list`	Two different lists are avaiable, almost similar: One displays historical revenue, and another displays new revenue. New revenue can be marked “Transfered”. This will move the revenue figure to historical revenue. The transfer button results in a confirmation dialog. Every revenue figure contains: The event origin, amount of tickets, date which is was last updated, and the total financial figure.
`Event create`	Displays just a few basic fields for creating an event: Name, description, musical category, poster upload, total tickets for sale, and date and time for the doors to open
`Event settings`	Displays several individual forms and warning messages about data that needs to be filled in. Firstly, the basic form, but more fields are available, adding sales end time, event begin time, and event end time (for events that span several days) Publish/unpublish button Add/remove ticket offers Add ticket offer modal dialogue: For form selecting ticket prototype, price, maximum sales, maximum per email, deadline for purchase List of guest lists. Unique URL for each guest list. Total number of slots per guest list + slots filled in (by the band, promoter etc) Form for adding a new guest list. Event access: Grant additional users access to administer this particular event (promoters).

Function	Description
`Social widget`	Displays the widget to be included on a venue website and the embed code to include it.
`Adjust sales`	Displays a form to cancel the event or adjust sales after the event has taken place. The cancellation form includes a reason for canceling the event and redirects to a page that starts a process on the server that sends out confirmation emails for every ticket holder and sends feedback to the browser for every email sent, i.e. the overall progress. Adjustments can be made for every ticket offer available for the event, e.g. normal presale tickets, door sales etc.
`Ticket list PDF`	An enumerated, printable list of all tickets for an event. Sorted by Ticket ID. Every ticket has a barcode, name and email of ticket holder and a field indicating if the ticket has been used (for filling in with pen and paper). The list can be used to post-process an event check-in that has been conducted on paper.
`Event report`	States all data of an event: Sales figures, revenues, and added taxes on events such as VAT and Danish KODA fee. Sums up overall profits after taxes. A report can be LOCKED by clicking a button, thus all data for the event become locked throughout the system.

B.2.3. Check-in

Function	Description
`Log in`	First screen: Prompts for username and password, even though a user is already logged in. This ensures that management does not grant their privileges to door staff.
`Select event`	If more than one event are available or the user has management rights, a list of events is displayed to select which event to activate for the check-in screen.

Function	Description
`Check-in`	Main page for checking in guests. Displays the active event, and the number of presale tickets pending + the number of people granted access for the event. 3 columns for action: Column 1 has an input field where ticket codes can be entered either through keyboard or barcode scanner. When all digits of a code is registered, the code is verified from the list of valid codes. If it is valid, the overall checked in count is incremented. An icon displays whether the scan resulted in a valid or invalid ticket. All scans are logged on screen and can be reviewed. Column 2: Displays guest lists. With a click, each guest list is expanded, and every entry name+guests can be checked in by means of a confirmation dialogue. Once an entry is checked in, it is clearly marked with a strike-through. Column 3: Cash sales. All ticket offers are displayed, and by clicking a ticket type, a dialogue prompts the user, if he/she wants to sell 1 ticket of this type. If connection is lost, an icon at the top will indicate that data is processed offline and has to be synchronized in order to avoid data loss.
`Ticket list`	All presale tickets are display with name and email of ticket holder + ticket ID. Cash ticket sales are displayed and can be refunded individually.

C. Misc

C.1. Popularity of rapid application development

	Rapid Application Development	SCRUM
Amazon book search	46	635
Google scholar	12,900	61,400
Google scholar since 2009	3,470	11,700

C.2. COCOMO estimate

Using some simple settings and really just a guide to perceived old fashioned, top-down approaches, this is the outcome of an estimate of a 10,000 line software project with a requirement of a large database and high availability, using the COCOMO model:

Effort: 19.2 Person-months (using 1 person for most work, and 2 persons for construction)

Person-months Schedule: 12.7 Months

Software Activity Distribution (Person-Months)

Phase/Activity	Inception	Elaboration	Construction	Transition
Management	0.2	0.6	1.5	0.3
Environment/CM	0.1	0.4	0.7	0.1
Requirements	0.4	0.8	1.2	0.1
Design	0.2	1.7	2.3	0.1
Implementation	0.1	0.6	5.0	0.4
Assessment	0.1	0.5	3.5	0.6
Deployment	0.0	0.1	0.4	0.7

Bibliography

Addison, T., Vallabh, S., 2002. Controlling software project risks: an empirical study of methods used by experienced project managers. In: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology. SAICSIT '02. South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, pp. 128–140.
URL http://dl.acm.org/citation.cfm?id=581506.581525

Agarwal, R., Prasad, J., Tanniru, M., Lynch, J., 2000. Risks of rapid application development. Communications of the ACM 43 (11es), 1.

Bentley, R., Hughes, J. A., Randall, D., Rodden, T., Sawyer, P., Shapiro, D., Sommerville, I., 1992. Ethnographically-informed systems design for air traffic control. In: Proceedings of the 1992 ACM conference on Computer-supported cooperative work. CSCW '92. ACM, New York, NY, USA, pp. 123–129.
URL http://doi.acm.org/10.1145/143457.143470

Beyer, H., Holtzblatt, K., Jan. 1999. Contextual design. interactions 6 (1), 32–42.
URL http://doi.acm.org/10.1145/291224.291229

Botero, A., Kommonen, K.-H., Marttila, S., 2010. Expanding design space: Design-in-use activities and strategies. In: Proceedings of the DRS Conference on Design and Complexity.

Chell, E., 2007. Social enterprise and entrepreneurship towards a convergent theory of the entrepreneurial process. International small business journal 25 (1), 5–26.

Cloud.com, 2011. 2011 Cloud Computing Outlook. Online.
URL http://www.citrix.com/content/dam/citrix/en_us/documents/products/cloud_computing_survey.pdf

Crabtree, A., 2004. Taking technomethodology seriously: hybrid change in the ethnomethodology–design relationship. European Journal of Information Systems 13 (3), 195–209.

Dearden, A., Rizvi, H., 2008. Participatory IT design and participatory development: a comparative review. In: Proceedings of the Tenth Anniversary Conference on Participatory Design 2008. PDC '08. Indiana University, Indianapolis, IN, USA, pp. 81–91.
URL http://dl.acm.org/citation.cfm?id=1795234.1795246

Django Software Foundation, 2013. djangoproject.com.
URL https://www.djangoproject.com/

Eisenhardt, K. M., 1989. Building theories from case study research. Academy of management review, 532–550.

Floyd, C., 1992. Software development as reality construction. Springer.

Geertz, C., 1973. The interpretation of cultures: Selected essays. Vol. 5019. Basic Books (AZ).

Grudin, J., 1988. Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces. In: Proceedings of the 1988 ACM conference on computer-supported cooperative work. ACM, pp. 85–93.

Hartswood, M., Procter, R., Slack, R., Voß, A., Büscher, M., Rouncefield, M., Rouchy, P., Sep. 2002. Co-realisation: towards a principled synthesis of ethnomethodology and participatory design. Scand. J. Inf. Syst. 14 (2), 9–30.
URL http://dl.acm.org/citation.cfm?id=782686.782689

Howard, A., Oct. 2002. Rapid application development: rough and dirty or value-for-money engineering? Commun. ACM 45 (10), 27–29.
URL http://doi.acm.org/10.1145/570907.570925

Hughes, J. A., Randall, D., Shapiro, D., 1992. Faltering from ethnography to design. In: Proceedings of the 1992 ACM conference on Computer-supported cooperative work. CSCW '92. ACM, New York, NY, USA, pp. 115–122.
URL http://doi.acm.org/10.1145/143457.143469

jQuery, 2013. jQuery.com.
URL http://jquery.com/

Juris, J. S., 2007. Practicing militant ethnography with the movement for global resistance in barcelona. 2007) Constituent Imagination: Militant Investigations/-Collective Theorization, 164–178.

Kensing, F., 2003. Methods and practices in participatory design.

Kensing, F., Blomberg, J., 1998. Participatory design: Issues and concerns. Computer Supported Cooperative Work (CSCW) 7 (3-4), 167–185.

Krogstie, B., Divitini, M., 2010. Supporting reflection in software development with everyday working tools. In: Proceedings of the 9th International Conference on the Design of Cooperative Systems (COOP).

Latour, B., 1987. Science in action: How to follow scientists and engineers through society. Harvard university press.

Martin, J., 1991. Rapid Application Development. The James Martin productivity series. MacMillan.
URL http://books.google.dk/books?id=o6FQAAAAMAAJ

McConnell, S., 2010. Rapid Development: Taming Wild Software Schedules. Microsoft Press.

McQuaid, P. A., 2001. Rapid application development: Project management issues to consider.

Mumford, E., 2003. Redesigning human systems. Irm Press.

Naur, P., 1965. The place of programming in a world of problems, tools, and people. In: Proceedings of the IFIP Congress. Vol. 65. pp. 195–199.

Neyland, D., 2007. Organizational ethnography. SAGE Publications Limited.

Procter, R., Williams, R., 1996. Beyond design: Social learning and computer-supported cooperative work - some lessons from innovation studies. Human Factors in Information Technology 12, 445–463.

Ramsin, R., Paige, R. F., Feb. 2008. Process-centered review of object oriented software development methodologies. ACM Comput. Surv. 40 (1), 3:1–3:89.
URL `http://doi.acm.org/10.1145/1322432.1322435`

Sanders, E. B.-N., Brandt, E., Binder, T., 2010. A framework for organizing the tools and techniques of participatory design. In: Proceedings of the 11th Biennial Participatory Design Conference. PDC '10. ACM, New York, NY, USA, pp. 195–198.
URL `http://doi.acm.org/10.1145/1900441.1900476`

Shelly, G. B., Cashman, T. J., Rosenblatt, H. J., 2011. Systems analysis and design 9th edition. Cengage Learning.

Tahvanainen, A.-J., Steinert, M., 2013. Network! network! network! how global technology start-ups access modern business ecosystems. ETLA Working Papers 4, The Research Institute of the Finnish Economy.
URL `http://EconPapers.repec.org/RePEc:rif:wpaper:4`

Tjørnhøj-Thomsen, T., Whyte, S. R., 2008. 4. fieldwork and participant observation. Research Methods in Public Health, 91.

Turk, D., France, R., Rumpe, B., 2002. Limitations of agile software processes. In: Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP 2002). pp. 43–46.

Turk, D., Robert, F., Rumpe, B., 2005. Assumptions underlying agile software-development processes. Journal of Database Management (JDM) 16 (4), 62–87.

Twitter, 2013. Twitter Bootstrap - Sleek, intuitive, and powerful front-end framework for faster and easier web development.
URL `http://twitter.github.io/bootstrap/`

VersionOne.com, 2012. 7th State of Agile Development Survey.
URL `http://www.versionone.com/state-of-agile-survey-results/`

Weaver, E., Long, N., Fleming, K., Schott, M., Benne, K., Hale, E., 2012. Rapid application development with openstudio.

Wikipedia, 2013a. Agile software development — Wikipedia, the free encyclopedia. [Online; accessed 26-April-2013].
URL `https://en.wikipedia.org/w/index.php?title=Agile_software_development&oldid=550817798`

Wikipedia, 2013b. List of graphical user interface builders and rapid application development tools — Wikipedia, the free encyclopedia. [Online; accessed 26-April-2013].
URL `http://en.wikipedia.org/w/index.php?title=List_of_graphical_user_interface_builders_and_rapid_application_development_tools&oldid=551345360`

Wirdemann, R., Baustert, T., 2008. Rapid Web Development mit Ruby on Rails. Hanser Fachbuchverlag.

Zahra, S. A., Sapienza, H. J., Davidsson, P., 2006. Entrepreneurship and dynamic capabilities: a review, model and research agenda*. Journal of Management studies 43 (4), 917–955.

Zutshi, A., Zutshi, S., Sohal, A., 2006. How e-entrepreneurs operate in the context of open source software. Entrepreneurship and innovations in e-business: an integrative perspective, 62–88.