

# Introduction of Innovation M2

Christopher Oezbek  
[christopher.oezbek@fu-berlin.de](mailto:christopher.oezbek@fu-berlin.de)

2009-01-23

## Abstract

This document provides an overview of the progress made with my research into the introduction of innovation between May and September 2008. It continues where the first milestone [49] left off and discusses first the methodical advances regarding Grounded Theory methodology. Second, it presents the first set of results regarding the introduction of innovation such as the concepts of hosting, partial migration, enactment scopes, adapter innovations and decision making. Lastly, it closes with ideas about how to conclude this dissertation.

## 1 Introduction

How to define *innovation* for this research, has proved surprising difficult again and again. As the demand for a definition has persisted, this milestone will revisit the problem of defining the term, despite the fact that a definition of a phenomenon still being under study might fail to include all essential aspects.

The *purpose* of defining the term innovation is to describe precisely what we intend the term to mean within the boundaries of this research. It should be noted that thus our purpose is explicitly not lexical in nature, as it does not try to capture the “conventional or commonly understood meaning” [65] of the term innovation, but rather is *stipulating* as to describe an intended meaning within a given setting and *precising* as to outline which objects are innovations and which are not [65].

Given this purpose, we can also give some general criteria a “good” definition should fulfil [65]: A definition should (1) capture the essential (as in necessary) attributes of innovations (from which follows that it should not be too broad or narrow, nor defined negatively), (2) not be circular, (3) not have an agenda

on its own or try to convey results, (4) be concise and readable and (5) avoid using terms as definiens that are themselves not well defined.

The first milestone presented one large ostensive example of an innovation introduction in which an Open Source project migrated from the centralized source code management system *Subversion* to decentralized *Git*. It seems worthwhile to expand on this by giving an *enumerative definition* of the three most common types of innovation seen so far in this research (see also Section 3.7 for a more formal treatment), which certainly is neither complete nor operational enough to be suitable to our purpose of a precise definition, but might still provide a good sense of what we deem an innovation.

The largest group of innovations found so far are improvements to processes and process phases such as (1) using a *merge window* period during which new features may be added to the repository of the project, (2) asking new project members to introduce themselves, (3) redesigning the user interface based on a competition between several designs, or (4) cleaning of bug trackers from outdated entries after releases. Second are those innovations which are tied to the use of software for developing software collaboratively on the internet. This includes the use of systems for (1) managing source code (e.g. CVS), (2) compiling source code to programs (e.g. autotools), (3) managing bug reports (e.g. trac), (4) communicating synchronously (e.g. IRC) and asynchronously (e.g. mailing-lists). Third are tools used by individual project members to such as (1) API documentation tools (e.g. Doxygen), (2) buildtools (such as autotools) or (3) helper scripts to deal with whitespace conventions.

With these examples in mind we repeat the definition from the first milestone [49]:

**Definition 1** (Innovation). *An innovation is some-*

*thing that is intended to change any kind of process in a project.*

This definition only differs in one significant count from the definition for software process improvements (SPI) given by the Software Engineering Institute in [51] as:

The changes implemented to a software process that bring about improvements.

The difference is that innovations are about change, no matter whether this change is negative, positive or even without implications. This avoids taking an explicit side on the question whether a change is good or not, but rather assumes the viewpoint of the innovator who has an agenda and intention for it.

The definition which most closely matches ours can be found in Fichman's study of assimilation and diffusion of software process innovation [23] in which he defines software process innovations as

*C*

changes to an organization's process for producing software applications - changes in tools, techniques, procedures or methodologies [24]

The primary difference to our definition is that we explicitly extend the definition to include processes that are not necessarily performed primarily to produce software. We do this because we think that all processes occurring inside a project which follows the Open Source development paradigm are interesting at this early stage for understanding this mode of software development. If for instance an innovation improves the process of new user acquisition, we do want to include it under this definition, because it might turn out that the number of new users in an Open Source project is correlated with the number of contributions.

## 1.1 Literature

This section discusses literature read during the second milestone that contribute ideas to the question of how to introduce innovations.

Gunter Dueck describes the introduction of a Wiki as a knowledge management innovation in IBM [18].

As Dueck narrates the story of the introduction, he also notes the key practices employed ("just do it", "keep underground") which map very nicely to the ones proposed by Manns and Rising [45], and enumerates two lists of blocking questions from management and technical personnel.

In an article from Microsoft Research India, researchers in information and communication technologies for development (ICT4D) [17] explain that they found in several instances that an iterative approach to innovation introduction is necessary. They report that they used action research to approach an acceptable solution since their initial understanding of the problem often was insufficient.

Regarding Open Source, another useful static typology might be found in [57], albeit founded in theoretical reasoning alone: Sawyer distinguishes three perspectives by which to look at software production: a.) a sequential view stemming from industrial production, b.) a group view from social psychology and c.) a network view from network-actor theory. The differences between the views might be useful when discussing why innovation introduction is an interesting research topic in the realm of Open Source. For instance, in a sequential view on software production, control is said to be the central mode of belief, while in the network view it is upon interaction. Designing prescriptions for dealing with introduction in Open Source, yet giving only control mechanisms might thus fail.

## 1.2 Overview of Milestone 2

The last milestone ended with the question which step to take next and offered four alternatives: (1) To pursue the most interesting memos, (2) to investigate a single innovation introduction episode in a single project in depth, (3) to compare the introduction of a single innovation across several projects, or (4) to keep an unrestricted viewpoint and stick to open coding.

In discussions with my doctoral adviser it was deemed most interesting to concentrate on option 1, i.e. to distil from the memos found so far the five most interesting concepts and continue coding with them specifically in mind.

The last report enumerated a series of such concepts, from which the following two were initially selected and analysed during May and June:

1. The *partial execution of migrations* occurring

with several source code management innovations (see Section 3.1).

2. The *enactment scope* used in the proposals of process innovations (see Section 3.2).

On June 24th I presented the results of this analysis in the research meeting. While interesting new concepts could be discovered by this mode of investigation, the choice of the two concepts had one severe drawback: The terms were not central to the innovation introduction but rather peripheral phenomena, occurring in only a small set of innovation episodes. To address this shortcoming, I started looking for concepts that are more essential to the idea of innovation and found two which had occurred repeatedly to warrant a closer look:

1. The role the *hosting* infrastructure plays within the innovation introduction process.
2. The impact of *effort* on the success of innovation introductions.

I decided to focus on the concept of *hosting*, because its scope was more clearly defined, and presented the results of this analysis (see Section 3.3) in the research meeting on July 22nd.

As this analysis was successful, I next moved to a term even more closely related to innovation introduction, namely the concept of *innovation discussions*. Targeted for the next research meeting on August 26th, this turned out to be too ambitious for several reasons: The concept was too broad, covering too many sub-concepts such as *argumentational tactics*, *decision making* or *resource acquisition*, and it was dependant on too many concepts that were still insufficiently defined such as *types of innovations*, the *types of outcome* of an episode or an understanding of the *phases of the innovation process*.

This provided the trigger to look at several of these concepts as a last iteration before the end of this milestone. I looked at *decision making* (see Section 3.5), *innovation types* (see Section 3.7) and *episode outcome* (see 3.6), before starting to write this report on September 22nd.

While writing this report, it turned out that much of the analysis I had already done for presentation needed to be redone for writing it down. I presented the results of this milestone once at the Forschungswerkstatt Pfadkolleg on December 4th [48] and in the

Seminar “Beiträge zum Software Engineering”<sup>1</sup> on December 11th.

## 2 Methodology

During this second milestone, *Open Coding* as described in [49] using *searching* and *scanning* has been replaced by *Axial Coding* as the main mode of operation. Thus, instead of focussing on individual mailing-lists associated with an Open Source project, the focus is now on developing and understanding individual concepts scattered across mailing-lists.

The typical workflow of an Axial Coding session with regards to data scattered in many primary documents has been the following, which I call *focused coding*:

1. Select a topic, idea or concept as the one under scrutiny (the concept of interest). For example one might decide that *hosting* deserves further study as an interesting aspect of innovation introduction, after chancing upon it time and again in Open Coding. Note, that while this first step might appear similar to choosing a core category in selective coding, the situation in which focused coding and the associated goal are different: In focussed coding one is picking up undeveloped concept to uncover its relationship to other concepts and give it substance in explanation, definition and meaning. In selective coding one is trying to pick the concept with the richest analytical founding in data and the strongest relationship to other concepts to arrange the whole presentation of the research results around this concept. Thus while selective coding presents the last step of a long journey of analysing a concept, focussed coding is merely the first step beyond having identified the concept to be of interest.
2. From the list of codes built in Open Coding select those that are related to the concept by searching or scanning through the list of codes. In many cases, codes will already exist that are named in close relation to the concept of interest. For instance we might look for all codes that contain the word ‘host’ and find that we have coded

<sup>1</sup><https://www.inf.fu-berlin.de/w/SE/SeminarBSE>

*forces.hosting* for occurrences of hosting as a decisive force leading to the adoption or rejection of an innovation.

It is not necessary to execute this step with high accuracy or look for all codes that are related to the concept of interest. Rather one is looking for codes with (1) the highest relevance to the concept of interest and (2) a medium number of occurrences of the code. The second criterion for selecting codes is aimed to ensure that we do not drown in a flood of messages that need to be read. Continuing with the example of *hosting* as the concept of interest, we will not pick a broad concept like *innovation*, which - while relevant - has hundreds of messages attached to it. Instead we rather pick *forces.hosting* or *activity.sustain*, which are closely related but only contain five to ten messages.

3. For each code in the set derived from the previous step, all messages are revisited and re-read with regards to the concept of interest. In many cases this will entail as a separate step the rereading of the whole enclosing episode to understand the context the message was written in. Such an expansion in the focus of reading - from single message to a whole episode - is often very time-consuming and the researcher will be tempted to use an opportunistic strategy to read only as much of the episode until sufficient context has been established. Using this workflow during the last months has shown that this is bad practice. Instead, the researcher should rather take the time to write a descriptive summary of the episode, which can be reused when revisiting individual emails later. Indeed the researcher should be aware that the development of a narrative summary of an episode is one of the primary strategies for making sense of an episode [38, p.695] and should always complement the application of Grounded Theory methodology.
4. The goal of re-reading is to gather insights about the concept of interest or, as it is called in the lingo of Grounded Theory methodology, *to develop the concept*. There are two types of such insights:
  - (1) We might find that the concept we have been looking at exists in different kinds along some attribute or property. When discovering such a

property, we can improve the *dimensional development* of the concept. For instance with regards to the example of *hosting* we might discover that there are different types of *locations* that provide hosting such as private servers and community sites.

- (2) The second kind of insight relates the concept of interest to other concepts, thus help with the *relational development* of the concept. For instance, with the concept of *hosting* we might realize that there is a relationship between our possibilities to have *control* over a server and the amount of *effort* required to maintain this server.

While this describes the workflow that has been followed for most of the analysis in this second milestone, we intentionally left several aspects of it open: (1) How do we store such insights systematically for further development? (2) How do we use these insights in our further work? (3) How do we present it to interested parties such as the research community, Open Source participants or corporate stakeholders?

The first and second question lead us to revisit our tool for qualitative data analysis GmanDA and improve it with regards to concept development, which is described in the next two sections. Additionally GmanDA was released as Open Source software during this milestone<sup>2</sup>.

## 2.1 Coding Syntax

The first major improvement implemented during the second milestone was an improvement of the *coding syntax* to be able to handle hierarchical annotations. To achieve this, the syntax of milestone one, that consisted of lists of key value pairs, was changed to allow nesting of sub-codes in the following way:

```
code: {
  subcode1,
  subcode2: {
    desc: "...",
    def: "...",
  }
}
```

The semantics of this syntax are similar to those of the key-value notation: An email annotated with

<sup>2</sup>GmanDA is available for download from <http://gmanda.sf.net>

a certain code implies that this email contains the description or reference to an instance of the type of objects labelled by this code. For instance, when coding an email in which the author discusses the question of whether to move the project web-site from one server to the other, the code *hosting* might be used to denote that this email contains a discussion about an instance of type hosting. This instance can be further qualified by describing it with further sub-codes. Two special sub-codes exist: *desc* (short for description) is used to describe informally the instance categorized by a concept, while *def* (short for definition) is used to give an informal definition of the concept attached to the instance.

Using this syntax it becomes possible to conveniently store information related to *dimensional development*, such as properties and their values. To this end, GmanDA uses the convention that codes starting with a hash symbol (#) are properties and all sub-codes of properties are values. For instance, if an email contains a proposition with a large enactment scope, the following coding might be used:

```
activity.propose: {
  #enactment scope: {
    def: "Enactment scope is...",
    large: {
      def: "A large enactment scope
            is...",
      desc: "This is scoped...",
    }
  }
}
```

For *relational development* no convenient mechanism has been developed yet. At the moment relationships between concepts are represented by informal descriptions as memos.

This syntax draws heavily from *YAML Ain't Markup Language (YAML)*<sup>3</sup> and *JavaScript Object Notation (JSON)*<sup>4</sup> and can easily be converted into both notations. The syntax was designed to be easily writable and intuitive to users of other notations.

## 2.2 Tabulation and Code Details

Using this syntax in GmanDA has caused a series of functional requirements to appear to help with the

<sup>3</sup><http://www.yaml.org>

<sup>4</sup><http://www.json.org>



Figure 1: The Code Detail view in GmanDA. Its purpose is to provide information such as definitions, properties and occurrences for a given code.

question of how to work with *relational* and *dimensional* data.

A first development in this regard was the *Code Detail View* which outlines each occurrence of a code in further detail. Given a certain code, this view allows to inspect all properties, definitions and descriptions and quickly jump to each email to which the code has been added. A screenshot of the view can be seen in Figure 1.

While the code detail view can already display all properties of a code and their values, it often became necessary to see how two properties are distributed with regards to a concept of interest. For instance, given the outcome of an episode as a property ranging from failure to success, one might be interested to see how large an enactment scope has been used in the proposal of the episode.

To enable such operation, the *Tabulation View* has been created, which is shown in Figure 2 with data relating outcome and enactment scope. Similar to the visualization view, it allows first to filter for messages of interest, then it provides a way to define the concept of interest and two properties of this concept. It will then plot each instance of this concept along the values of the given properties.

#outcome	value.high	value.high and low	value.low
???			
failed	<a href="#">design approval@bugzilla</a> <a href="#">bug milestone@xfce</a>	<a href="#">work groups@grub</a>	<a href="#">gsoc@xfce</a> <a href="#">bug tracking@grub</a> <a href="#">branch for patches@argouml</a> <a href="#">gpl3@geda</a>
success		<a href="#">git@geda change log creation</a>	<a href="#">branching@geda</a> <a href="#">bug tracking cleaning@xfce</a>
unknown	<a href="#">ask smart questions@bugzilla</a> <a href="#">integrate document changelog@xfce</a>		

Figure 2: The Tabulation view in GmanDA. It is used to analyse the relationship between two properties. In the figure, the properties *abstraction level* and *episode outcome* are tabulated after filtering and grouping for messages tagged with the *episode tag*.

## 2.3 How to Present Results

This is the first milestone in which any direct results are to be reported in written form. Much thought has been spent thus on the question of how to present the results of a Grounded Theory analysis as outlined in this section on research methodology.

The first question that needs to be answered in this regard is what actually constitutes a result of a Grounded Theory analysis. Corbin and Strauss offer three aims of doing qualitative research: (1) Description, (2) conceptual ordering, and (3) theory [13], which are distinct, but still intertwined. All analysis has to start with description, because “there could be no scientific hypotheses and theoretical [...] activity without prior or accompanying description” [13]. This becomes more apparent when considering “description” to be more than just textual rendering of raw data, but rather as the mapping of this data into the domain of things understood by the researcher. Only what is understood and grasped by the researcher can be used to construct a conceptualization of the data. Conceptualization for computer scientists probably is most closely related to object oriented modelling, where we try to group sets of related objects under the label of a certain class and then put up hierarchical subtype relationships between classes to form a taxonomy of their associated objects. Well-developed categories then become the base for constructing relationships between them, which constitute theory.

How can we present each type of research result given the format restrictions of a paper-printed dissertation? We are primarily restricted to text, with figures and tables only as supportive means and links only possible to sections, figures and tables. From the literature we gather that *discussion with intertwined narrative* is the main presentation form for Grounded Theory studies, with page-sized tables giving examples of developed core concept serving as support (see for instance [34, 3, 10]).

This report is a first attempt to achieve such a narrative form, since all intermediate results presented so far have been presented in the same order in which they were uncovered from the data. For instance with the concept of hosting, the presentation in the research meeting has focused on developing the concept together with the participants, step by step, just as I discovered them. While such form along the *analytical development* of a concept is easy to follow for the reader and easy to report for the researcher, it might miss a concluding condensation into results which are comprehensible and valuable on their own. For this milestone report, I have used the following supportive means of presenting results:

- **Links into data** I have added a large amount of references to the actual data from which I have derived my results. Since all emails are available via Gmane.org, I was able to put hyperlinks into the electronic version of this paper that take the reader directly to the ref-

erenced email. For the paper-printed version, the links are given as `<id>@<project>` - for instance `[1839@kvm]` - which can be manually resolved by looking up the exact name of the mailing-list in the Appendix A.6 and then browsing to `http://article.gmane.org/<list>/<id>` (for the above example the resulting URL would be `http://article.gmane.org/gmane.comp.emulators.kvm.devel/1839`).

- **Links to concepts** For each concept referenced and discussed in this report, the detailed definition as stored in my GmanDA database is given in the **Glossary** (see Appendix A).
- **Graphical representation** Where possible and applicable I have tried to create a graphical representation of the data and resulting theory developed (see for instance Figure 8).
- **Descriptive overviews** Since the analytical development of a concept often is terse with regards to describing the actual episodes from which the theory was derived, this report offers *Info-Boxes* that try to give more detailed accounts of important episodes (see for instance Info-Box 1). The reader is free to explore the episode in more detail, but it should be possible to follow the theoretical development without it.

## 2.4 Visualization of Temporal Data

The first milestone [49] presented a screenshot of the visualization support (see Figure 3 for a new example) built into GmanDA without explaining its underlying mechanisms. This section is to rectify this by providing a treatment of visualization of temporal event data such as log file data or emails as used in GmanDA.

We begin by defining  $E$  as the index-set of all events  $e$  of interest. An event  $e$  is at the moment a completely opaque object, without any internal structure. We then define  $\tau$  - the timing function - as  $E \mapsto \mathbb{R}$  that maps events  $e$  to a real number that we take to represent a timestamp (for instance think of the number as the number of milliseconds since 1970-01-01).

To attach meaning to the events, we define  $T$  as the set of all tags  $t$  that we might attach to an event  $e$ . For instance we might use a certain  $t_1 \in T$  to represent a certain author or another  $t_2 \in T$  to represent a certain type of output which appeared in a log

entry. A coding  $C \subseteq T \times E$  then is used to represent which tags have been attached to which events. We found it useful to also define a partial order  $<$  on top of the set of tags  $T$  to denote a hierarchical sub-tag relationship (if a tag  $a$  is semantically a sub-tag of a tag  $b$ , then  $a < b$ ).

The first operation we then introduce is the *filtering* function  $f : 2^E \mapsto 2^E$ , where  $\forall_{x \in 2^E} f(x) \subseteq x$ , which takes a set of events and returns a subset of them. In particular, a *tag filtering* function  $f_{t,C}$  is a *filtering function*, which returns the set of all those events that are tagged with  $t$  in the coding  $C$ . Formally,  $\forall_{x \in 2^E} \forall_{y \in x} y \in f_t(x) \Leftrightarrow \exists_{s \in T} s \leq t \wedge (s, y) \in C$ . Another useful filtering function uses the timing function  $\tau$  to reduce the set of all events to those which occurred in the interval  $[start, end]$ . Using such filtering, we can reduce the amount of events we see on screen comfortably by naming the tags and time-periods we are interested in.

The next operation we found useful is *partitioning*, which we define to be a function  $p_C : 2^E \times 2^T \mapsto 2^{T \times 2^E}$ , which receives a set of events and a set of tags and splits the set of events into subsets filtered by each tag (the *partitions*), i.e. for  $x \in 2^E$  and  $y \in 2^T$  we define  $p_C(x, y) = \{(t, f_{t,C}(x)) | t \in y\}$ . This operation is useful for splitting events along a second dimension (the first being time) such as for instance the authors of emails, or the program from which a log entry originated. In many cases, the most common application of this partitioning function is in combination with a tag  $t$ , for which we are interested to see a partitioning along the sub-tags. This can be achieved by the set of all sub-tags for a given tag  $t$ , which we define as  $S_{\leq t} \subseteq T$ , with  $\forall_{x \in T} x \leq t \Leftrightarrow x \in S_{\leq t}$ . Also often useful is the direct sub-tag function  $\bar{S}_{\leq t} \subseteq T$ , which does only contain those tags which are direct sub-tag of the given tag  $t$ . For  $\bar{S}_{\leq t}$  the following needs to hold  $\forall_{x \in T} x \in \bar{S}_{\leq t} \Leftrightarrow x < t \wedge \neg \exists_{y \in T} x \neq y \wedge x < y < t$ .

We can also *nest* partitions further by taking each partition and applying the partitioning mechanism again. For a similar mechanism by which partitions can be built, see the table algebra in [60].

As a last processing step before being able to draw the event data, we use a *ranking* function  $r : T \times 2^E \mapsto N$  to define an order by which to print the *partitions* from a *partitioning* induced by a set of tags. Ranking functions we have found useful have been:

- The starting time rank function  $r_s$ , which ranks

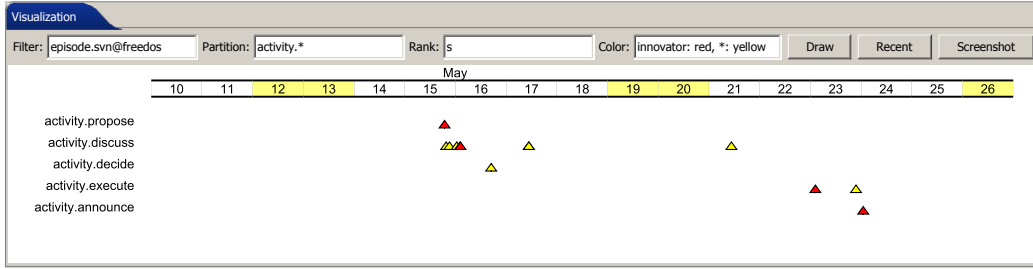


Figure 3: Example screenshot of the visualization built into GmanDA. A four-stage visualization pipeline with the operations filter, partition, rank and color is used to generate the resulting visual display.

partitions in the order of their earliest event, i.e.  $(t_1, e_1), (t_2, e_2) \in T \times 2^E$  holds that  $r_s(t_1, e_1) < r_s(t_2, e_2) \equiv \exists x \in e_1 \forall y \in e_2 \tau(x) < \tau(y)$  (and respectively end time, median time etc.)).

- Ranking based on the number of events in a partition function, which was used to find does episodes containing a sufficient amount of events. Formally we can define  $r_{|\cdot|}$ , where for  $(t_1, e_1), (t_2, e_2) \in T \times 2^E$  holds that  $r_s(t_1, e_1) < r_s(t_2, e_2) \equiv |e_1| < |e_2|$ .
- Alphabetical ranking based on the name of the tag of the partition.

Given such filtered, partitioned and ranked data, one could in addition *align* the partitions according to the first/last/median occurrence of a certain tag [63]. If interested in finding patterns it might make additional sense to *normalize* the total time displayed in each partition (or the time between two events) to allow for easier detection of patterns.

As a last step in the processing pipeline, we then take events and map them to visual marks. Currently only coloring based on tags and a single type of marks is supported, but one could extend this to support changes to shape, size, orientation, color and icons/textures [60] by tags as well. If quantitative information is available about an event this information can be used for styling the visual mark as well. Consider for instance the analysis of quotation depth of email messages by Barcellini et al. [7]. The authors defined quotation depth as the maximum number of replies leading from a source email to a direct or indirect reply which still contains part of the source email as a quotation. Using this definition for a function  $q : E \mapsto \mathbb{R}$  which returns the quotation depth regarding an event

representing an email message, we can use  $q(e)$  for any given  $e \in E$  to scale the size of the visual mark appropriately. Thus emails which are deeply quoted draw more attention.

Concluding, we want to relate this way of visualizing event data to the visualization of activity data that Salinger and Plonka are using in their analysis of pair programming sessions [56]. Instead of having atomic events without any temporal extent, their visualization consists primarily of activities with beginning and end times. To map events to activities we found two basic strategies: (1) Given two tags, we can interpret occurrences of events tagged with one of them as flagging the start of an activity and the other as flagging the end of an activity. Repeated occurrences of start events are ignored until an end event signals the end of an activity. For example, when working on email data regarding the release process of an Open Source project, we can define the period in which features may be added to the code base (the *merge window*) as the time between any number of consecutive emails stating that features may currently be added to the code base and the first following message stating that only bug-fixes and localization may be added to the code base (the *feature freeze*). (2) We can interpret the time between the first and last event marked by a given tag as the time during which an activity occurred. For instance, given three email messages tagged to belong a certain episode, we can define the period during which the episode occurred as the time between the first and last email tagged with the episode.

An overview of the operations for dealing with event data is given in Figure 4.

Once the events have been rendered to screen, GmanDA supports interaction using the zoomable

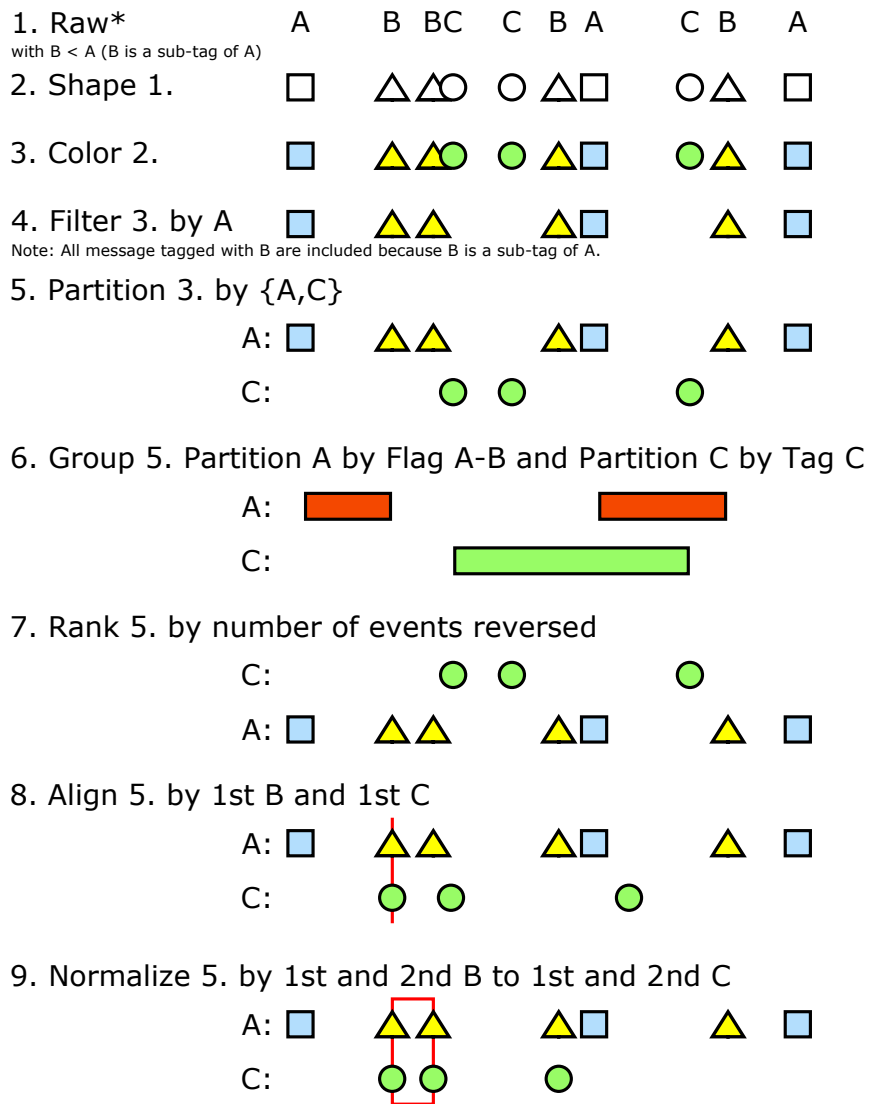


Figure 4: Operations for dealing with event data such as email messages. The given example data uses three tags A, B and C, where B is a sub-tag of A. Note\*: The letters in the first row (1. Raw) should be interpreted as events with one of the tags A, B or C and positioned in time relative to their placement on the horizontal axis.

user-interface paradigm based on the Piccolo Toolkit (successor of Jazz [8]). Most notably this includes the use of a semantic node for presenting the time-line, i.e. when zooming-in a time-line for months and a time-line for individual days is only displayed at adequate zoom-ratios.

## 2.5 Reasoning about Grounded Theory Methodology

The use of Grounded Theory methodology (GTM) in the Software Engineering research at AG Software Engineering has become a corner stone of our daily research work and thus continued thinking and exploring of the associated concepts is necessary to avoid running into methodological dead-ends or taking long detours where swift progression to the goal of research would have been more appropriate.

In this section, I thus want to discuss the following two issues related to GTM which this milestone provided some insights about: (1) Understanding the paradigm as a tool in Grounded Theory methodology, and (2) the use of a foundation layer or set of base concepts and its implications.

First, we want to discuss our understanding of the paradigm as a tool used in Grounded Theory research.

Strauss and Corbin describe the paradigm as “a perspective taken toward data [...] to systematically gather and order data in such a way that structure and process are integrated” [61, p.128]. To achieve this end, the paradigm suggests to take a concept of interest (called the phenomenon) and analyze for each occurrence of the phenomenon in data (a) the set of conditions that have caused the phenomenon and those that manifest its properties (those are called the context<sup>5</sup>), (b) the set of strategies employed by persons as reaction to a phenomenon, and (c) the consequences of such interaction. Strauss and Corbin note explicitly the danger of becoming stuck in the structure of the paradigm and following its prescribed model of analysis too rigidly and reducing the analysis to simple cause-effect schemes that lack deeper analytical insights [61, 13].

I have argued before in the research meeting that

---

<sup>5</sup>The term context is thus not used in the sense of “a context in which the phenomenon appeared”, but rather the context is the concrete instance of an abstract phenomenon in the data. The term highlights the importance of all phenomenon only being of interest if they serve as a context to further action and events.

the paradigm as given by Strauss and Corbin primarily improves our analytical abilities regarding the reaction to a phenomenon (as can be easily seen in the graphical representation of the paradigm shown in Figure 5) and is static towards the causes of the phenomenon. I have hypothesized in the research meeting as well that this alignment towards reactive analysis is caused by the kind of phenomena that Strauss and Corbin investigated such as pain and dying, which are emergent from a context rather than the products of actions by people. To extend our ability to understand the dynamic aspects that led to the phenomenon, I think it makes sense to split each causal condition analytically into three aspects, which closely mirror the reactional aspects into which the consequences of a phenomenon have been divided: First we need to know which actions and strategies have caused the phenomena to occur (action/causing strategy), second we need to understand from which starting point these actions were taken (the starting context), and third which conditions influenced the actions taken in a way relevant to affecting the phenomenon (intervening conditions). The resulting paradigm model can be seen in Figure 6.

A second important insight with regards to the paradigm is related to its ability to aggregate several occurrences of a phenomenon. In this milestone I have used the paradigm only as a tool that can be applied to a single occurrences of a phenomenon in the data as can be seen in Figure 7. I have not seen any indication that the paradigm provides any help in integrating several such *frames of analysis* or occurrences of the phenomenon to construct higher conceptual abstractions beyond providing a comparable structure for each frame. Grounded Theory methodology does not provide any explicit tool for these kinds of abstractions to my knowledge. My doctoral advisor has suggested to take the resulting paradigm frames and integrate them by perceiving them as data on which GTM can be performed again. We should investigate literature further for uses of the paradigm to learn how people have used it or any other mechanisms for reasoning about how process can be integrated into the GT being developed.

As a second point of reflection on Grounded Theory methods as used by the AG Software Engineering, I want to discuss my experience with the usage of foundational concepts or a set of base concepts: When I first started coding emails for this research project, I started developing my own coding scheme

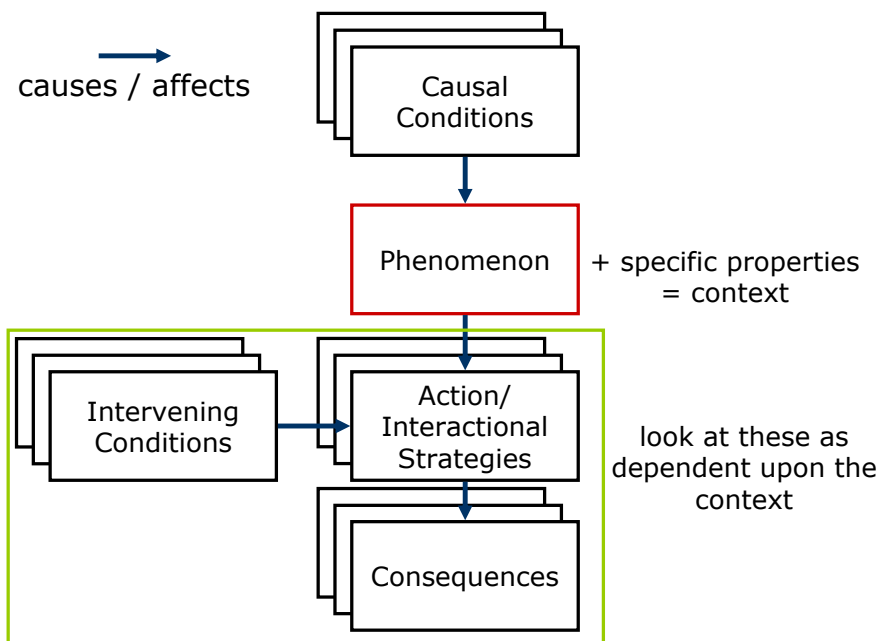


Figure 5: The paradigm model [61] graphically represented using a logic frame. Arrows pointing between elements represent that the pointed-to element is caused or affected by the pointing-from element.

to exhaustively and orthogonally conceptualize and describe what I saw happening in the emails I was reading<sup>6</sup>. When looking back now at these concepts (super-concepts such as *activity*, *argumentation*, *innovation* of these were discussed in the milestone 1 [49]) and looking at the way that results from axial coding were derived, these base concepts served the following purposes:

- Base concepts are useful as pointers into and marks in the data to find all occurrences of a certain phenomenon of interest. For instance when trying to explore how hosting played a role in the process of introducing an innovation (see Section 3.3), the codes for sustaining and executing activities were two of the most important candidates for finding emails from which insights about hosting could be derived. We did not expand *executing* so that hosting was a property of it, but rather *executing* became an associated concept of the more central concept hosting.

<sup>6</sup>I intentionally use the words *conceptualize* and *describe* at the same time, because there is no precise boundary between these terms, rather they have different values along the dimensional property of abstractness.

- Base concepts serve as starting points to develop properties. For instance, when looking at two episodes related to proposals of bug-tracking processes, I found that they resulted in different outcomes. To explore this difference, I started to investigate the properties of the proposals made, and arrived at the concept of enactment scopes of a proposal. During the course of this analysis, the importance of the original concept (proposal) became increasingly smaller and the embedded concept (enactment scopes) developed a life as a concept in its own right. I believe that such a concept could have emerged even in Open Coding already, but using base concepts offers a more systematic approach and less strenuous way to uncover it.

On the other hand, the danger of becoming stuck in base concepts should never be underestimated, because some concepts can be more easily derived as properties than others. For instance, if we consider the concept of partially migrated innovations (see Section 3.1) as those innovations that replace only part of an existing innovation, it is unclear where we would attach a property

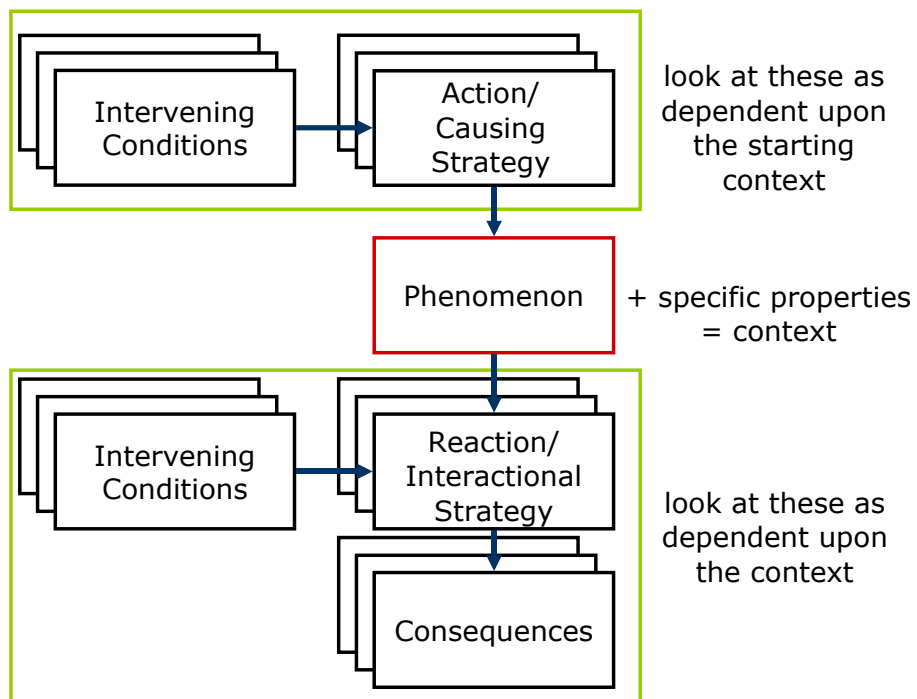


Figure 6: The paradigm model extended with three more elements to emphasize that the phenomenon is caused by *actions* that occur in a *starting context* and are influenced by a separate set of *intervening conditions*. Arrows pointing between elements represent that the pointed-to element is caused or affected by the pointing-from element.

such as partiality when we are coding using base concepts.

- Base concepts help to prevent concept explosion. By providing more clearly defined concept-boundaries than the ad-hoc concepts that appear in GT left and right while analyzing data, base concepts restrict the numbers of concepts that are in use. To that end, the base concepts' orthogonality and precise definitions are of high importance.

As a third point regarding GTM, I want to include some hints from literature. In particular, Suddaby makes a series of good points regarding Grounded Theory methodology [62] of which especially one shows some weaknesses this work has so far: High level of ignorance regarding existing literature both from the Open Source research community and the underlying social sciences or economics literature. There is no real excuse for this behavior, except that the goal was not to get stuck in preexisting views before having a good hard look at the data itself. Each time I looked at theory, it seemed that it fit only insufficiently well to our data and thus was not included. While being aware of the problem, the use of literature to me has the following main disadvantage that I have been unable to resolve: If literature is in line with the results, then the results are merely confirmatory; if literature is in disagreement with results, then the results are cast into doubt. I have added a more rigorous treatment of literature to the further work and believe that our visit to the Pfadkolleg workshop [48] has made the need sufficiently clear.

It remains an unsolved problem how the whole AG SE can deal with the task of boundary spanning between computer science and the underlying organizational and psychological topics. I would suggest the following: (1) Make interdisciplinary ideas an explicit topic in either the BSE or research meeting and devote 10 minutes each week to talking about theories and models from other domains. (2) Create stronger ties with other research disciplines and try to write papers with colleagues from them. (3) Continue to work out our methodology and vision for the results of the research group.

As a last point and to complement both the theoretical reasoning about GTM above and the practical application of GTM in the result section (3) below, I decided to perform small explorations into literature,

trying to uncover reports and results regarding the use of Grounded Theory methodology in areas related to my own research on innovation introduction.

With the first of such an exploration, I chanced upon *Grounded Theory and Organizational Research* by Martin and Turner [46], which explains how Grounded Theory methodology can be used for data from OR.

That Grounded Theory methodology is used in many different ways can be seen in *Planned Organizational Change: Toward Grounded Theory* by Dunn and Swierczek [19], who perform a retrospective case analysis on a set of 67 change episodes. They first built a Grounded Theory based upon existing literature which they then used to perform content analysis of the given episodes. Using the number of occurrences of each concept, they then looked at a series of hypotheses from literature rejecting most of them.

## 3 Results

During this second milestone the following concepts have been further expanded as outlined in section 1.2: (1) **Partial migrations**, (2) **enactment scopes**, (3) **hosting**, (4) **adapter innovations**, (5) **decision making and compliance enforcement**, (6) **episode outcomes** and (7) **innovation typology**.

### 3.1 Partial Migrations

The discovery of this concept in the data during open coding is closely bound to a single episode occurring in the project KVM at the beginning of 2007. This project conducted the introduction of a novel source code management system, while at the same time retiring its existing system.

**Definition 2 (Migration).** *A subtype of the introduction of an innovation, where an existing innovation is replaced by the newly introduced one.*

Yet, instead of migrating its existing setup completely from old to new innovation, the project chose to conduct the migration only partially, moving certain parts of the code-base of the novel system, while retaining others in the existing system. Not surprisingly this approach of concurrent systems led to a series of problems such as duplicated effort in maintaining branches and increased complexity of determining which revision contains a certain change [1839@kvm].

Similar situations of partially introduced innovations occurred in several other projects, which prompted the following questions:

- What causes partial migrations to occur, i.e. in which situations, because of which actions by project participants, and why does a partially migrated state arise?
- What are the consequences of partially migrated innovations?
- How and to which effect do projects and their members deal with these consequences? In particular, if the consequences are negative, why is the migration not completed?

These questions map directly to the development of a paradigm centered around the phenomenon of partial migrations as discussed in the Section 2.5 on Grounded Theory methodology. We thus set out to understand the phenomenon of partial migrations by trying to understand the actions causing the phenomenon, the starting context that they occurred in and any augmenting conditions influencing the actions taken so that the phenomenon occurred.

### 3.1.1 Partial Migration at KVM

The first episode in which we encountered a partial migration occurred in the project “*Kernel based virtual machine*” (KVM), which is a commercially-backed Open Source project of the start-up Qumranet, on which this company bases its main offering of a desktop virtualization product. KVM is a virtualization solution on top of the Linux kernel for x86 hardware, so that virtual machine images can be run on a single computer running Linux. The commercially dominated background is important to understand the history of KVM’s use of source code management tools. The demand for making KVM’s internally kept code repository publicly available first appeared at the end of 2006, when the project became more and more popular. After discussing internally, access using the centralized source code management tool *Subversion* was provided. This sets the starting context in which the partial migration occurred.

In February 2007, the project leader announced that the parts of the project code interfacing with the Linux kernel had been migrated to be managed with the decentralized source code management tool *Git*, yet

that the user space parts would remain in the existing system *Subversion*. The primary reason given for this change was the inadequacy on the side of *Subversion* to deal with changes in the requirements when compared to the advantages that *Git* provided. First, *Subversion* was said to be unable to scale to handle a full Linux kernel tree, which had become necessary when KVM code extended into places in the kernel beyond a single sub-directory. Second, *Subversion* requires an account on the Qumranet server for each developer who needed write access to manage long-term branches. These reasons are perfectly fine reasons to migrate, yet they do not explain why the project did migrate only partially and retained a large portion in *Subversion*. The only reasons we can find is in a discussion related to the initial offering of *Subversion* access to the project:

“Git’s learning curve is too steep for me. I may dip into it later on, but I’ve got too much on my plate right now.” [187@kvm]

So, abstracting from this quote, we see that learnability and associated effort to learn a new technology in combination with an already high work load might be the primary impeding factor to a full migration. Moving another step up in abstracting these insights, we can subsume these augmenting conditions that have caused the maintainer of the project to stay clear of a complete migration to two antagonistic ones: While certain reasons call for a migration, others make it more difficult to do so, and the project maintainer escaped the full impact of both by migrating only partially.

Because KVM is driven by a company, we did not see any emails as part of the migration itself that could have provided further insight. What we did see on the other hand is that, one month later, the maintainer complains about the negative implications of the partial migration:

“Managing userspace in *subversion* and the kernel in *Git* is proving to be quite a pain. Branches have to be maintained in parallel, tagging is awkward, and bisection is fairly impossible.” [1839@kvm]

So given the phenomenon of an existing partial migration, we see that within a month the context has changed: (1) The situation that was created just a month ago is now undesirable to the maintainer. (2)

The maintainer has learned how to use *Git* sufficiently well to be comfortable with it [1853@kvm]. These changes in the context create room for new interaction, which the maintainer thus uses to propose to migrate the remaining part of code from Subversion to *Git* by putting them into the *usr*-directory of the Linux kernel repository. At the same time though, the maintainer directly adds an intervening condition: The proposal feels “slightly weird” [1839@kvm], which sounds like a petty reason at first sight, but should probably rather be interpreted as the maintainer’s hunch that the proposal would cause pain and awkward situations in other regards. The second intervening condition that acts upon this proposal is the knowledge discrepancy the other project members have regarding using *Git* in comparison to Subversion [1853@kvm]. Thus while everybody seems to agree that the proposed solution is weird (for instance [1840@kvm]), all alternative solutions proposed involve undoing the migration to *Git*. Instead developers offer tools and processes for being able to use *Subversion* alone, such as scripts for managing patch sets [1845@kvm]. Given this situation, the maintainer withdraws his proposal and *postpones* a decision to think about it some more [1869@kvm]. If we try to interpret this “wait and see” strategy of his and deduce its consequences, we see that the maintainer is most probably balancing the effort necessary to adopt one of the proposed solutions against its effects on reducing his “pain” and the chance that the effort could be wasted, if a better solution arises.

It takes another four weeks before this better solution can be found. Instead of unifying the user space Subversion and kernel Git, the revision content of the Subversion repository is migrated to the new Git repository [2336@kvm]. The context in which this completion of the migration was done is given by the request of a user to enable http-access to both the Git and Subversion repositories [2184@kvm]. Thus, what the user is asking is to invest effort into maintaining both systems. If we interpret the partial migration as an effort management technique, it loses more and more appeal, if such request continue to arise. Thus the maintainer takes hold of the opportunity and migrates to a separate Git repository. After two months of using Git, this even draws a cheer from a project participant [2434@kvm].

A graphical summary of this analysis of the partial migration occurring at the project KVM can be found in Figure 7.

To conclude the discussion of the partial migration at KVM, we want to come back to our initial questions:

- *What causes partial migrations to occur?*  
Antagonistic forces such as shortcomings of existing tools and lacking knowledge of novel ones can be the cause of partially migrated innovation introductions.
- *What are the consequences of partially migrated innovations?*  
Partially migrated innovations do cause painful duplication of effort to manage and sustain both innovations, yet at the same time enable the project to become comfortable with both innovations.
- *How and to which effect do projects and their members deal with these consequences?*  
The project actively engaged in discussion when the maintainer mentioned his pains with managing both innovations in parallel. It was quickly agreed that it was important to resolve the situation. It is important to note that the project members did only provide solutions based on the old innovation and that it remained the task of the innovator to resolve the issue.

### 3.1.2 Partial Migration at ROX

To discuss the concept of partial migrations, we choose as a second episode, a migration occurring in the *project ROX* which focuses on developing a desktop environment that is closely tied to the file-system. Again this is a migration away from centralized *Subversion* to decentralized *Git* and again the decision to migrate is made unilaterally by the maintainer of ROX, so that we do not see any decision processes regarding the migration, but rather only get informed by the maintainer about his rationale to migrate the core component of ROX - the ROX-Filer - to a decentralized version control system (see Info-Box 1 for a detailed discussion of this rationale). Reasons why all other of the many sub-projects, such as the window manager OroboROX, the session manager ROX-Session or the library for shared ROX functionality ROX-lib remain in SVN, are not given directly as part of this proposition, but hide in the ensuing discussion during the execution of the migration.

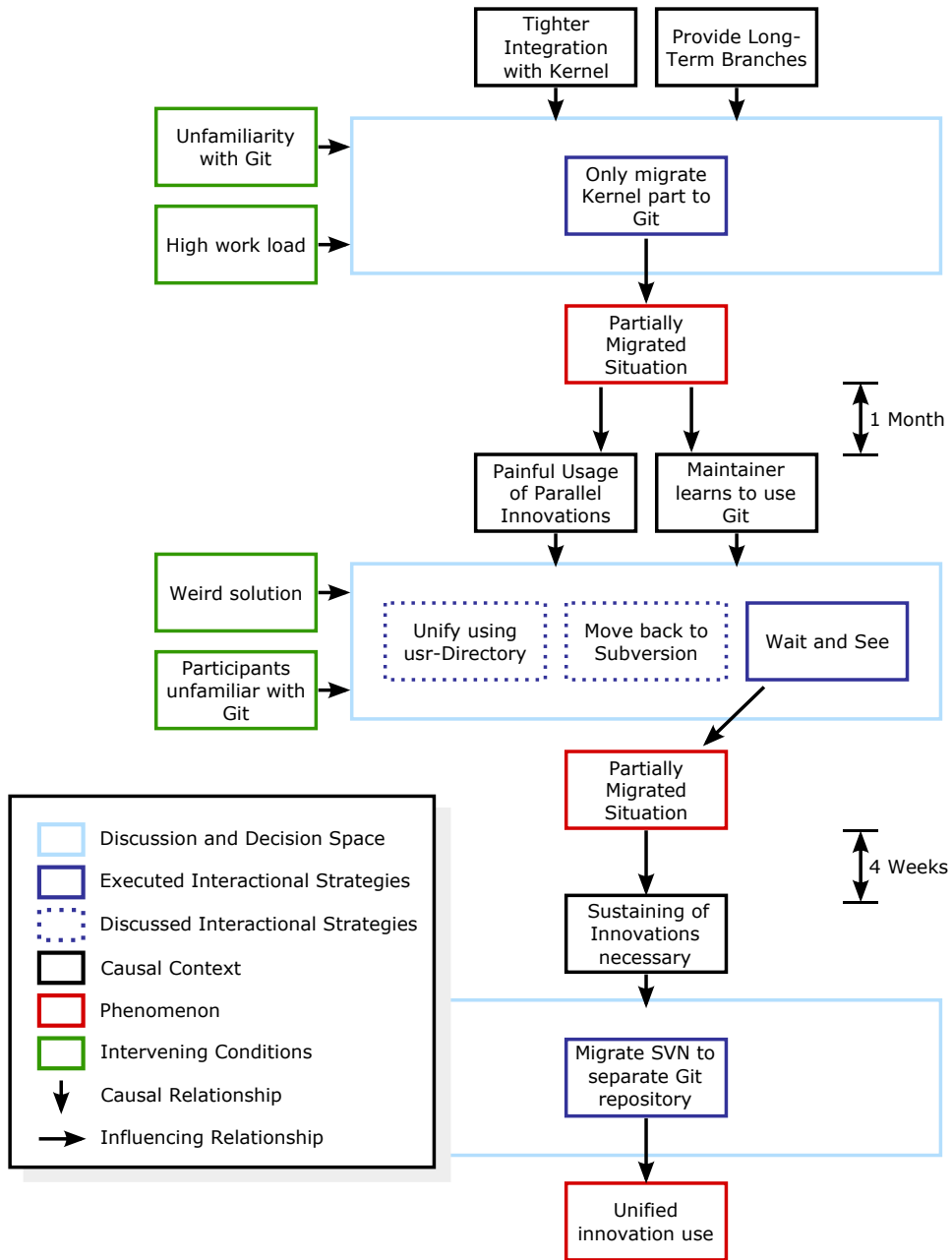


Figure 7: A graphical representation of our investigation into the concept of *partial migration* with an episode occurring in the project KVM using the paradigm. The process by which the phenomenon of a *partially migrated* source code management situation arose and was resolved is depicted by its three discussions and decision frames.

A first indication why the strategy to migrate only partially was chosen can be found in the email by the maintainer in which he retrospectively *narrates the conversion* to Git: The maintainer describes that the technical capabilities of both SCM tools favor the partial migration; Subversion by making it easy to extract parts of a repository, and Git by making it easy to combine two repositories. While this is certainly no causing condition, it clearly is an augmenting one, making the strategy more appealing. So in addition to the *module boundaries* and *boundaries of responsibility* (Kernel code vs. User Space in KVM) we can add *technologically indicated boundaries* as possible candidates along which a partial migration might occur.

When the maintainer of the project ROX proposes to migrate the repository of the central component ROX-Filer from Subversion to Git, his argumentation on the highest level of abstraction revolves around *enumerating disadvantages and problems with the existing and advantages of the proposed innovation*. He describes these from the view point of (1) the contribution processes for non-core-developers as a whole (*process centric view*) and (2) certain important individual tasks (*capability centric view*) such as keeping up to date with an evolving repository, collaborating without the presence of a project core member and merging a set of changes with full history. His argumentation also elegantly separates the discussion about a solution in the abstract (using decentralized version control) from the discussion about a solution in the concrete (using Git) by first making sure that he has convinced to use a centralized version system and then providing an additional set of arguments favoring Git over other DVCSs (all these arguments are based on external validation, such as being in use at the Linux Kernel).

**Info-Box 1:** Rationale given for the *migration from Subversion to Git in the project ROX*. [9368@rox]

If we take the freedom to look at an answer of the maintainer to the question whether there were plans to complete the migration [9380@rox] and take the answer - no there weren't any - as indications why the migration occurred only partially, then we can find two more conditions that are involved: (1) Finishing the migration would require a lot of free weekends [9384@rox]. Abstracting from this, puts *lack of time* and *associated effort* on the table of causing

conditions for partial migration. (2) He wants to see how using Git for the ROX-Filer will work out first, from which we can deduce an *unfamiliarity* with the new technology (similar to the one occurring during the introduction of Git in KVM) that is causing the maintainer to *run a trial* first.

If we next look for the consequences of the partial migration, these are not that clear as they were with KVM, because nobody is really complaining or being vocal about having pains with the migration being partial. Rather, we see several problems that were caused by the migration being only partial: (1) Because not all content managed in Subversion is deleted after the migration to Git, there are some commits that end up in the wrong repository and need to be manually transferred to Git [9404@rox], which we can abstract as *increased usage problems*. (2) A more complex problem might also be caused by the *duplicate learning effort*. In [9424@rox], a developer asks for some patience with providing a patch using Git, because he is still in the process of migrating in his personal development from CVS to Subversion and has not had time to learn how to use Git. If a migration thus is only partially, he cannot just stop learning about Subversion and start learning to use Git, because he still needs the related skill to use the parts not migrated.

More consequences probably exist, but they remain unspoken of, so that we are left to wonder whether they would cause anybody to want to complete the migration or whether other strategies are adopted. As noted before, the maintainer has opted to use a *wait and see* strategy of first *running a trial* with ROX-Filer and Git [9384@rox], before considering to migrate the rest of the project. We do not get any answers to the question why this strategy was chosen, and what it accomplishes, yet the most likely reasons seems to be *effort management* on the side of the maintainer: Not investing effort in a new technology or innovation, unless there is a clear indication to do so. Another *conservative* strategy can be seen when a developer adds new features to a sub-project remaining in Subversion and creates at the same time a new sub-project to complement the existing-one in Git without any obvious problems [9499@rox].

We find that the only developer who seems to be interested to extend the migration to more sub-projects is the one who initially asked the maintainer whether he had plans to complete the migration. When he gets the negative response from the main-

tainer [9384@rox], he pursues two interesting strategies (we must note this developer is already experienced with the use of Git [9370@rox] and has access to hosting [9380@rox] as two important intervening conditions): First is his offer to the project to provide an *adapter* that allows to access an existing Subversion repository using a Git client (a git-svn mirror) [9385@rox]. While such a system cannot copy the workflows enabled by distributed version control system, it causes its users to have a *homogeneous tool environment*. Such an offer is interesting strategy, since (1) it allows each developer to choose a central tool according to personal preference and (2) it is highly independent from being legitimized by the project, since it does not interfere with existing innovations. In fact the concept of an *adapter* has not only proved useful with regards to making partial migration seem like a completed one, but also as a powerful way to introduce an innovation (see Section 3.4).

The second strategy he employs is to migrate individual parts of the Subversion repository to Git using his own hosting infrastructure [9399@rox]. While he manages to migrate several such repositories [9403@rox], looking back one year later, we see that his efforts must have been in vain, because the repositories have disappeared, are still being managed in Subversion or have been migrated again by the maintainer [9647@rox]. The only effect of his activities seem to have been that he and the maintainer learn how to migrate repositories to Git more effectively [9409@rox].

To sum up this investigation into *partial migration* of Subversion to Git in the project ROX:

- *What causes partial migrations to occur?*  
Lack of time and lack of expertise are the most likely reasons why the maintainer decided to conduct a *trial* and not complete the migration despite compelling advantages in favor of the new innovation.
- *What are the consequences of partially migrated innovations?*  
ROX did not suffer any mayor negative consequences like KVM from the partially migrated project state, except a couple of minor *usage problems* and possibly some *duplicated learning effort*.
- *How and to which effect do projects and their members deal with these consequences?* Be-

cause there are no negative consequences, we see *conservative* strategies with the goal to *manage the expedited effort* regarding the migration of existing sub-projects, such as (a) keeping the *status quo* of existing sub-projects and only migrating novel ones and (b) using *adapter* technologies to enable *homogeneous tool environments* without migrating all sub-projects.

I believe that this analysis was able to convey some of the reasons and implications of partial migrations in Open Source projects. While not saturated with sufficient cases, we have seen that partial migrations are not without merit and that they provide the innovator with opportunities for strategic action.

### 3.2 Enactment Scopes of Process Innovation

Two proposals occurring within two weeks on the mailing-list of the desktop manager *xfce* triggered this second exploration into a concept. Both proposals, at first, seemed to involve almost the same process innovation, yet their outcome was completely different. The first proposal asked to define a target release for each outstanding bug, so that planning and tracking of progress towards the release becomes easier. The second proposal asked to close bugs already fixed and outdated with the current release. The first proposal - thirteen days before the second - failed to attract any attention of the rest of the development team, while the second proposed process was accepted by the project and enacted successfully by its innovator. What might have caused this difference in reception by the other project members?

A first look at the reputation of both innovators in the project does show the reverse of what we would expect: The successful innovator is not a developer in the project and has written only a third as many emails as the innovator who failed. Thus standing with the project cannot explain the differences in outcome.

What did strike us during analysis was that the successful innovator proposed only a single enactment of the process innovation for the current release, while the failed innovator proposed vaguely implying all future releases. This seems like a plausible difference by which we could explain different behavior of other project members: Being asked whether one wants to

do the activities being part of the process once is certainly a less difficult decision than deciding whether one wants to do its activities now and for all future releases.<sup>7</sup> More formally we can define (derived from [20]):

**Definition 3** (Enactment). *The execution of activities mandated by a process by one or several agents.*

And based on this:

**Definition 4** (Enactment Scope). *The set of situations in which a process should be enacted.*

Given these definitions we can hypothesize:

**Hypothesis 1.** *Restricting the enactment scope of an innovation proposal is positively correlated with introduction success.*

Which gives rise to the following strategy to be used by an innovator:

**Strategy 1.** *To increase introduction success for process innovations, an innovator should limit the enactment scope of the innovation.*

A directly visible advantage of this strategy is that after the initial enactment of the process, the innovator can assess much better whether the introduction is worthwhile to pursue. If not, the effort spent to convince the project of adopting the process for an abstract set of situations is not in vain.

To evaluate whether the hypothesized relationship between the probability of failure to be accepted and size of enactment scope of the proposition holds and can be used to formulate the strategy above, I looked at ten other episodes in which *process innovations* were proposed.

To begin with, I found two more episodes which exhibit the proposed correlation: First, is an episode in which an innovator proposes to adopt a stable/unstable branching scheme and focuses his proposal on the current situation. He describes the implications of branching now, nominates a certain person to be the maintainer of the new stable branch, chooses version numbers for each novel branch and gives detailed instructions for how development would continue. This proposition with small enactment scope

<sup>7</sup>It should be noted that proposing such a correlation only is interesting if we also assume that the *expected effort* of adopting a process innovation is a central factor correlating with the acceptance of this innovation. While I think this is a reasonable proposition, it is not yet grounded in coded observations.

is accepted and a stable branch is created within a month. Following this first enactment, yet without any renewed discussion or involvement of the original innovator, the branching procedure is enacted twice more over the next half of a year, thus strengthening the idea that an innovator might want to focus on getting a first enactment of a process innovation introduced and using its success to extend to a larger enactment scope such as at each release or every two months.

A second episode then shows the rare sight of a lead developer failing to gather support for a proposition of his, when he keeps the proposal too vague and thereby largely scoped. This is interesting because even though he proposes only an optional step to be added to the quality assurance process (in his case an optional design review before the mandatory patch review), his proposal is challenged on the notion that the proposal would cause effort and complexity:

“And why a comment from us is not enough?” [6944@bugzilla]

So again, we have found that a large or vague enactment scope correlates with fear of expected effort.

When looking now at the cases that do not fit the proposed correlation, we first find three episodes in which an enactment scope of one still leads to rejection.

In the first case, a maintainer proposes to two core members to use a separate branch for collaborating on a feature, when they start having problems with managing their patches via the issue tracker. This proposal is rejected by the two core developers arguing in the following way: (1) The first developer directly states that he deems the use of branches too much effort [4773@argouml], while the second says so indirectly by noting that branches have not been used for years to his knowledge [4784@argouml], which I have interpreted as primarily stating that the use of branches would have to be learned again. If we interpret both developers as pointing towards effort as reasons for rejecting the process innovation, we can clarify our current hypothesis that a reduced *enactment scope* can help to alleviate fears of overburdening *expected effort*: If even a single enactment of a process innovation is expected to be too cumbersome, the proposal will still fail:

**Hypothesis 2.** *When reducing the enactment scope of a process innovation proposal, the innovator still*

*needs to achieve acceptance for a single enactment.*

(2) The second argument used against the process innovation is not associated with effort expectancy, but rather picks up on the small *enactment scope* of the proposal: Even though the innovator asks the two developers whether they want to use a branch in the *current* situation of collaborating on a given feature, the second developer enlarges the *enactment scope* by asking whether the innovator was proposing to start using the branches again in *general*. He then goes on discussing the appropriate situations for branching and possible dangers. This can be done implicitly by assuming that a single enactment of the process innovation can become the default case for the future. Doing so, he counteracts the strategy of using a small enactment scope by directly discussing the implications of a larger scope:

**Strategy 2.** *To attack a proposed process innovation, an opponent can expand the enactment scope of an innovation, i.e. extend the discussion about consequences of adopting an innovation to additional situations in which the process could be enacted.*

Interestingly, while the proposal to use branches for collaboration on patches is rejected by these two arguments, the maintainer continues to advocate the reinstatement of branches as a means for collaborating. He does so by making them the default development approach for students under the Google Summer of Code program two months later. Taking this indirect approach, he achieves that within a month the use of branches has been adopted by the two previously opposing developers, and that over time even one of them explicitly suggests to a new developer to use branches for the development of a new feature [5681@argouml].

The second and the third case of a rejected proposal - in separate projects each - with small enactment scope extend these two insights. In the second case, we see an innovator proposing the participation of the project in the *Google Summer of Code*, and see this proposal fail because of *lack of time* of the maintainer and bad *previous experiences* with participating [13244@xfce]. This is in line with the hypothesis derived from the previous case, that reduction in *enactment scope* still needs to achieve acceptance of a single case. The third case shows another example of attacking a small enactment scope. When the innovator suggests to use the project's *Wiki* as infrastructure

for keeping track of bugs found just before a release, the opponent challenges this proposal by expanding the enactment scope to bug tracking in general:

“btw, I think the wiki is not very convenient, something like a forum would be much better.” [3268@grub]

From there the discussion quickly loses any association with the initial process-oriented question and becomes tool-centric.

When looking now at the episodes in which the enactment scope was large (i.e. should be used continuously or enacted in all future cases of a certain kind), we find that all of these proposals were (a) proposed and decided in favor of, but that we could not determine whether they were adopted as intended and thus successful, (b) proposed by the maintainers of the project and (c) all try to influence general behavior such as being more disciplined with keeping a changelog [13533@xfce], being more friendly towards new developers [6190@bugzilla] or being more careful when asking questions [6540@bugzilla]. Since we cannot say whether these proposals successfully changed the behavior of the project members, we can also just speculate whether smaller scoping - for instance such as praising an especially friendly developer or reprimanding another one for a missing changelog - could be more successful.

The two last episodes presented here show another interesting aspect related to *enactment scoping*: Both proposals contained small and large scoped aspects at the same time. In the first case, an innovator proposed that developers interested in a common feature should work more closely together as *work groups* (in the literature on team effectiveness such a work group would be called a self-managing work teams [12]) as to speed-up development on these features. He first did so with a large scope, targeting a possible wide variety of such work groups:

“Therefore I would propose that we would setup a group of people that would concentrate on specific issues related to implementation (like a work group or something).” [3236@grub]

Only later on does he reduce the scoping by naming two such “specific issues” that would need work groups right now. Similarly, in the second episode, a developer proposes a new scheme for managing

changelog messages, first by describing the general consequences of not adopting his innovation and then becoming very concrete in describing how to execute his new scheme for the first time [4330@geda].

This second episode of combined small and large enactment scoping was accepted and successfully used by the project, while the idea of work groups fails in an interesting fashion: When the innovator proposes to establish them, the project members in the discussion directly start doing so by talking about their desire and abilities to participate in the two concrete work groups that had been proposed. Such a *skipped decision* related to adopting an innovation is an interesting concept we saw for the first time in this episode. Since these work groups - formed without a decision - failed to go beyond an initial declaration of interest and their momentum quickly fizzled out, the verdict is still out whether the possibility of a skipped decision is an advantage of a certain type of innovation or it rather leads to *premature enactment* when the consequences of the innovation are not yet well known. To remain on the cautionary side, we might leave this as two open hypotheses to investigate:

**Hypothesis 3.** *When proposing a process innovation with too small an enactment scope, this can cause the constituents of the project to skip decision and directly enacting the innovation.*

**Hypothesis 4.** *To skip decision and directly adopt/use/enact an innovation is negatively correlated with introduction success.*

As a last remark on enactment scopes, we want to note some similarities between the concept of *enactment scopes* and *partial migrations*: With partial migrations we have seen that tools are not introduced for all parts of the project, but rather that some parts remain under the old technology and some are migrated to the new one. In a way we could also interpret this as a reduction of enactment scope along the dimension of modules in contrast to the reduction of scope in the dimension of future enactment situations. To discuss this relationship in further detail remains for future work.

### 3.3 Hosting

While studying the concepts of *enactment scopes* and *partial migration*, another concept began to stir my interest: Some participants in the commercially-backed

project *KVM* had repeatedly asked for a wiki to be introduced by the sponsoring company and eventually created one themselves when their calls remained unanswered. Despite this wiki being perfectly usable and available, the sponsoring company then hosted their own *official* wiki shortly thereafter on a server run by the company and never acknowledged the system set-up by the developers. This official wiki turned out to be so popular, that within a week the network bandwidth of the server running the wiki was insufficient, which caused problems with the source code management system being run on the same machine.

This case highlights that the question of where and how an innovation is *hosted* and why this choice has been done, has been left unexplored so far. We define:

**Definition 5 (Hosting).** *Provision of computing resources such as bandwidth, storage space and processing capacity, etc. to the use by an innovation.*

We ask the following questions:

- Which kind of strategic and tactical choices regarding hosting do innovators have when introducing an innovation?
- What are the consequences of a certain kind of hosting for the introduction and usage of an innovation?
- How is hosting related to the innovation introduction process?

I thus started a series of 14 coding sessions totaling 19 hours and 40 minutes focused on the concept of hosting. In contrast to the two previously discussed concepts in which I had used the paradigm (*partial migrations*) and tabular diagrams (*enactment scopes*), this section employs a literary style, because the analysis is more exploratory with regards to conceptual development than suitable for the other two approaches. Tabular diagrams as used for the discussion of enactment scopes in particular require a small set of alternatives, while the paradigm focuses too much on cause-effect analysis rather than embedding a concept of interest into a larger set of related concepts. Analysis was conducted using *focused coding* as described in Section 2:

First, I looked through the list of existing codes for the ones that were related to hosting such as “innovation.hosting”. I then looked at each email given this code and wrote memos in which I tried to develop

the concept of hosting and attach properties to it. After having written 36 memos regarding hosting in this way, the memos were re-read and a mind-map linking the concepts created. The following discussion describes the properties of the concept hosting and its relationship to other concepts.

First, we were interested to learn about the different options that an innovator has with regards to hosting, to possibly derive recommendations based on these options how the innovator should act strategically. The following rough *types of hosting* can be distinguished:

- *Forges* such as SourceForge.net or Berlios are platforms for collaborative software development that offer a set of popular development tools, such as a SCM, a mailing-list, bug-tracker and web-space. The projects hosted on such forges are usually not thematically related to each other.
- *Service hosts* such as repo.or.cz provide specialized hosting for a certain type of innovation (in the case of repo.or.cz, Git hosting) [2893@geda].
- *University servers* often provide web-space and compute capacity to students and faculty free of charge. For instance in the project gEDA, a participant used the *Student-Run Computing Facility* (SRCF) to host an adapter innovation making the Subversion repository of the project available via Git (see Section 3.4) [2799@geda].
- *Private servers* owned, rented or operated in the name of and by individual project members. For instance in the project *Flyspray*, one of the maintainers sets up a service running the latest developer snapshot of the project's software on his private server gosdaturacatala-zucht.de [5422@flyspray].
- *Affiliated hosting* occurs when a project asks a thematically related project to use its already available server or service infrastructure. For example, some developers in the project *KVM* at one point considered hosting a *Wiki* on kernel.org, which is the umbrella web-site for all Linux kernel related projects.
- *Foundation hosting* occurs when a project is part of a larger group of projects that are joined under a shared legal entity. These groups or their umbrella organization then often operate servers

for the use by their member projects. An example would be the hosting used by the Bugzilla project, which is part of the infrastructure provided by the Mozilla Foundation, from which the Bugzilla project originated.

- *Federated hosting* is a lesser kind of foundation hosting, where the ties between the individual projects is not based on a legal entity but rather on a common goal or friendship and cost-sharing between the maintainers. For instance, the project *gEDA* is hosted by the *Simple End-User Linux* (SEUL) project, which provides a home to projects which want to do “development for user friendly software for Linux, and more generally for high-quality free Linux software of all kinds”<sup>8</sup> [2899@geda].
- *Private PCs* are privately owned computers, which do not have a permanent internet connection.

Once these types had been found, the question arose which attributes make them unique and important with regards to the introduction of innovations. We identified five concepts that seem to influence most of the decisions people make towards hosting: These are (1) effort, (2) control, (3) identification, (4) cost and (5) capability. As the concepts all relate to and depend on each other, we will try weave their explanation and definition together in the following paragraphs.

We need to start with the concept of effort, as this has been initially our only variable to describe why a decision towards a certain type of hosting would be made or not.

**Definition 6** (Effort). *The amount of time, physical or mental energy necessary to do something.*

In the case of hosting, such effort can include (depending on the innovation) the time necessary to install the software on the machine, migrate data to be used by the innovation, configure network settings, watch for security updates and apply them in time or deal with hackers and spam attacks, create back-ups, take care of overfull disks and hardware or software failures, move the server to other virtual or physical locations, inform the project about the status of the innovation being hosted and answer questions about

<sup>8</sup><http://www.seul.org/pub/hosting.php>

any of the these. That maintainers should strive to minimize these activities should come as no surprise and maintainers find strong words how they feel about the effort hosting can cause:

“I dont need to devote my time administering yet another mail server, doing so is a no option unless someone pay for my time” [5411@flyspray]

“People often tend to think about software quality, but the really important thing in services is the maintenance. It is surely easy to set up a server, but it is very tough to maintain such a server for years (especially without being paid) [...] Please consider my words carefully, before proposing not using savannah.” [3273@grub]

Yet, obviously given the above example of the introduction of a Wiki in the project KVM, less effort cannot explain why the maintainers reject hosting the Wiki using affiliate hosting. If effort is to be avoided, why is it that many projects host their services on their own servers and not on big forges like Savannah or SourceForge.net which have dedicated staff for administrative tasks? In the example of KVM, the decision to not use the Wiki at kernel.org, caused a lot of effort to troubleshoot the software failures reported by users, and eventually required a possibly costly increase of bandwidth for the server [1168@kvm].

To answer this question, the following episode at the project ROX provided two more insights: Here, the project was experiencing such a slowdown of the web-site, that beside proposing to update the version of the content management system used, the maintainer also suggested to migrate the web-page away from SourceForge.net to a private project server [9507@rox]. A caveat: as hosting on a private server costs money, the maintainer asked the project whether they would accept advertisements to be shown on the project web-site. This reveals two more variables which influence hosting choices: *cost* and *performance* (or rather more abstractly *capability* of the hoster). While hosting which is “free as beer” [5411@flyspray]<sup>9</sup> seems to be preferred widely, in this episode, the members of ROX were not op-

<sup>9</sup>A play on words on the explanation given for the term Free Software: “you should think of free as in free speech, not as in free beer.” [59]

posed to trade dependence on some form of income such as advertisement for a faster web-site.

As for capability, several other aspects of the concept have been found extending beyond performance. Most basically, if the existing hosting does not provide a certain service, or the offered set of features with regards to a service is inadequate for the needs of the project, then project participants need to look elsewhere. Consider the following exemplifying quote from the migration of Subversion to Git in the project ROX (see Section 3.1.2):

> Does SF provide git hosting now?

No, and given how long it took them to support svn I'm not holding my breath ;-) [9373@rox]

Other attributes of the capability of a hoster include the *bandwidth*, as seen negatively in the episode, at KVM, *storage space* [31112@rox] and the *quality of service* such as uptime of a host [9428@rox]. The latter for instance is a drawback of private PCs which are not connected to the internet permanently.

Yet, neither capability nor cost can explain the KVM episode as kernel.org is both free of charge and a perfectly fine Wiki. Upon further searching for episodes related to hosting, I found the following episode in the database project *MonetDB*: One developer found that the results from the nightly regression test-suite were provided on the web-page without an expiration time, such that a browser would cache the pages and not provide the latest results [121@monetdb]. When notified about this, the maintainer replied that a fix “must be done in the server” [122@monetdb]. Abstracting this remark, we arrive at the concept of *control* which the project members exert over a certain type of hosting:

**Definition 7 (Control).** *The degree to which the potential capabilities of an innovation can actually be used by the project.*

In the above example, the server is certainly capable of sending out appropriate HTTP headers, but somebody still needs to be able to perform the corresponding changes.

To have sufficient control to perform certain tasks on a server, such as configuring the innovation, backing-up its data, troubleshooting it, installing updates and more, is then the first concept that can explain why the KVM maintainer rejected the Kernel

wiki: This wiki - while offering all necessary capabilities at a low price - is not controlled by KVM but rather by Kernel.org.

Control only refers to the potential to perform an activity, but not that it will actually be performed. In the above episode, the maintainer does not configure the server to correctly set the expiration time and the proposal fails [122@monetdb]. Also one should note that control is often bound to individuals who are authorized to perform certain activities. For instance, when one developer set up a Git-Web system to show changes to the Git repository in the project *gEDA*, he alone was in control of this system and thus had to ask project members to be patient with regards to updates which he needed to perform manually [9428@geda].

If control is lacking, then the project members become dependent on hosting staff, are unable to resolve issues with hosting or have to exert higher levels of effort:

- When the maintainers of the project *Flyspray* lost the password to administrate the mailing-list, their hosting provided them so little control that they did not have any other choice but abandon their existing platform and migrate to another hoster:

>So far, the old list was a better option for me.

for us, is no option, we had zero control over it [5411@flyspray]

- In the database project *MonetDB*, one of the core members made a mistake while committing to the project repository, which rendered the repository unusable for the project. The situation could only be resolved by filing a support request with the staff of the hosting provider and waiting for them to handle the issue [468@monetdb].
- During the introduction of *Git* into the project *gEDA*, the innovators ran a Git-CVS adapter innovation to show Git's usefulness to the project. In retrospect the innovators reported that this had not been easy "due to lack of git servers we control" [3068@geda], giving us a relationship between lack of control and increased effort.

Such instances of lack of control highlight its implications for the use of innovations and give us another variable by which we can understand why not

every project is using Forges such as SourceForge.net with their low level of associated effort. Forges, we learn, provide less control than privately owned servers, on which the owners can work at will. A notable exception to this seems to be the Forge operated by the GNU project - Savannah. As the software used to operate it - Savanne - is Open Source itself, the users of the Forge have the possibility to extend the capabilities of Savannah by contributing to Savanne [3273@grub].

Until now I have discussed control mainly as pertinent to administrative tasks, but the definition given above also explicitly includes control over the usage of an innovation. For instance in the context of centralized source code management, it is common to employ a user-based access control scheme to only allow changes to the repository by project members with commit rights [6@monetdb]. If we consider a project member who has sufficient control to change who has commit rights or not (sometimes called *meta-commit rights*), we can deduce that hosting (via its associated control) can be one of the sources for power in the project.

When looking to saturate the set of concepts that relate to the concept of hosting, we found one more interesting phenomenon to join the previously found cost, effort, capability and control:

In the project *Flyspray* the maintainer had become dissatisfied with the low level of user-feedback he received on a new experimental feature he proposed to the project. When repeating his request for feedback, he offered to the project that he would be willing to set up a server on which the latest development reversion could be run. This would allow each project member to beta-test the software without having to install an unstable developer version themselves before [5395@flyspray]. When the users responded positively [5399@flyspray], because many of them did only run stable versions [5409@flyspray], the maintainer set up such a development demo system on his own private server [5422@flyspray], yet marked it as temporary:

I have now set up a temporary development BTS (until we get one on our server) at

<http://gosdaturacatala-zucht.de/devel/>

What this hosting option was lacking was none of the four previously mentioned concepts, but rather

it only failed to fit the *identification* of the community. Conversely, a participant in another episode hosting at a private server as well explained that he did so to avoid being seen as “official” and rather give the hosting location the notion of being “experimental” [9428@rox]. Identification with the social norms of the Open Source community and adhering to its standard can also be subsumed in this concept: When the maintainer in Rox asked the project whether he could run advertisement on the project homepage to cover the expenses of moving to a paid hoster, we might interpret this as him being sensitive to the underlying norms in the project.

That *identification* is something that projects seek for their hosting options, might also explain why *federated* and *foundation* hosting exist at all, when *private server hosting* (high control, high capability) and *Forges* (low cost, low effort) are available.

The analysis of these five concepts related to hosting constitutes the main insight generated by this investigation. We have identified cases in which each concept was an important factor during a discussion related to choosing an appropriate hosting: (1) In the project KVM, the project maintainers favored a hosting with high control, even though they were presented with an innovation hosted at a site with little effort, no cost, sufficient capability and high identification. (2) In the project Grub, the maintainer used harsh words to remind the innovator that abandoning the existing Forge for a certain service would cause a lot of maintenance *effort*. (3) In the project Rox, insufficient performance of the existing host made the project members trade their existing low-cost solution for paid hosting with better capabilities. (4) When forced to look for a new mailing-list host, because losing the password caused lack of control, the project Flyspray decided that no cost would be a central requirement for the new service. (5) Despite being cheap and fully functional, the maintainer in the project Flyspray realized that hosting on a private server of his would provide insufficient identification with the project and thus he declared the service as only temporarily hosted there.

It should be noted that these concepts are just one possible way to categorize the variables that influence a complex topic like hosting. This dissertation did not spend sufficient scientific energies to validate the completeness or practicable usability of the developed categorization, but rather feels it reasonable adequate for innovators and researchers alike to use

it as a framework for understanding hosting. If and when additional concepts arise, one can add them or try to explain their appearance as a combination of the given ones. For instance, during the introduction of the source code management system *Subversion* in the project *FreeDOS*, one of the project veterans noted that one should host the project repository on SourceForge.net to “keep all FreeDOS-related resources in one place” [4826@freedos]. While defining a concept such as *co-locality* or *unification* of hosting resources makes sense, it is best to explain its motivation as a combination of the other five concepts. In this case, unified hosting is driven by reduced maintenance effort, more centralized control and improved identification, since all services are kept in one place. Capability is affected by co-location as well, but it is unclear whether it would cause higher risk as a single point of failure or enable synergies. Cost is not likely to matter, as SourceForge.net is free of charge.

### 3.3.1 Strategies for Hosting

Because each identified concept has appeared as important during at least one innovation episode, it is difficult to derive any *strategic* advice for an innovator with regards to hosting. The following should be regarded as speculation and is not backed by a stringent inquiry.

It seems that there are two essential strategies for non-maintainer innovators: (1) To convince people in control over the project’s current hosting to enable a novel innovation or (2) to use private hosting such as a university server to host the innovation outside of the project’s current hosting and over time achieve integration into the project.

We have seen both the first and second strategy succeeding and failing, but have extracted little insights so far. The first strategy, it seems, is particular dependent on the attitude of the person in control over hosting resources (often the maintainer) and the innovator should explore this attitude in advance by looking at innovation cases in the past. For the second strategy a common problem seems to be that the innovator considers his job done after setting up the innovation. Unfortunately in many cases, the innovation is not able to overcome adoption barriers by the advantages it provides alone and the innovation introduction then dies a slow death.

### 3.3.2 Relating Hosting to Innovation Introduction

So far we have interpreted hosting as a concept on which the innovator can have some slight strategic influence by making a choice on the location where he hosts an innovation. Implicit to this was the conceptual relationship that hosting is (1) a *requirement* for the usage of an innovation and (2) a *task* during execution (execution is the set of activities that are necessary to make the innovation usable by the project). In fact, it turns out that in many introduction episodes, the acquisition of hosting and the subsequent announcement of availability of a service constitute the only activities that the innovator will be involved with. For instance in the project *Xfce*, one developer set up an instant messaging server to provide low-cost and secure communication to the project by installing a software on his own private server and announced this via the mailing-list [13133@xfce]. There was no public discussion, decision making or advertising the innovation, just the provisioning of the service resource and the announcement to the list. Even though the introduction in this case failed to find any adopters, we can take note that the achievement of hosting is the central activity of executing and possibly the *minimal introduction* scenario for a service innovation. Conversely, if the innovator fails to acquire hosting that is acceptable, then the introduction as a whole will probably fail.

We found four other relevant conceptual relationships between hosting and individual aspects of an innovation introduction:

The first such connection to innovation introduction can be derived through the concept of control. We have discussed in the last section that control is often an important factor when making a decision against or in favor of a certain type of hosting, because it determines the ability to utilize the capabilities of the server. When considering these capabilities with regards to using the innovation, we found that control can lead to power, for instance when only certain people in the project can assign commit rights or access certain functionality in the hosted software or content. Conversely, control can be a *barrier* to democratic access in the project.

Next, we found that because achieving hosting is vital to achieving a successful introduction, it also often features prominently as an *argument* during the discussion and decision making. For instance, when a

developer proposed to switch from CVS to Subversion in the project FreeDOS, the only restriction brought up was the demand to keep resources of the project unified on SourceForge.net [4826@freedos]. Thus, it might happen that instead of discussing the potential effects of an innovation, the controversial matters might be related to hosting.

Fourth, hosting can be a *trigger* to a new discussion and decision process. In particular, when hosting breaks or changes its status, this seems like a plausible point in time to consider alternatives to the current system. For instance, during the introduction of Git in the project gEDA, the innovators hosted an adapter innovation to allow project members to get to know Git. When this adapter got broken by the administrators, it was a natural point in time to ask the project members whether there was any interest in moving the whole development to Git [2889@geda].

Lastly, implicit to the discussion about hosting has always also been that hosting is the source of countless *tasks* during the sustaining of an innovation (much of the discussion about effort of a hosted innovation derives from these sustaining tasks).

An overview of the proposed relationships can be seen in Figure 8.

To conclude this section, we want to note which types of innovation (see Section 3.7) have been found to use hosting:

Predominately we found that innovations with a *service* component, such as a centralized source code management system, a bug-tracker, mailing-list, or the project web-site require hosting. As an interesting sub-case we found that projects which develop web-applications also frequently host their own software for two reasons: (1) Their product is an innovation such as a bug-tracker and is used by them directly as part of their development process. (2) The product is hosted as part of another innovation such as providing a demo system to facilitate easier beta-testing.

Second were innovations that require *documentation* to be accessible by users. For instance in the boot-loader project *U-Boot*, the maintainer wrote a design document which outlines the primary principles to follow when developing for U-Boot, and then hosted it on the project server [29737@uboot].

Third, we also found that *tool* innovations are also frequently hosted by the project, so that developers can easily download a designated or customized version of the tool [4909@argouml].

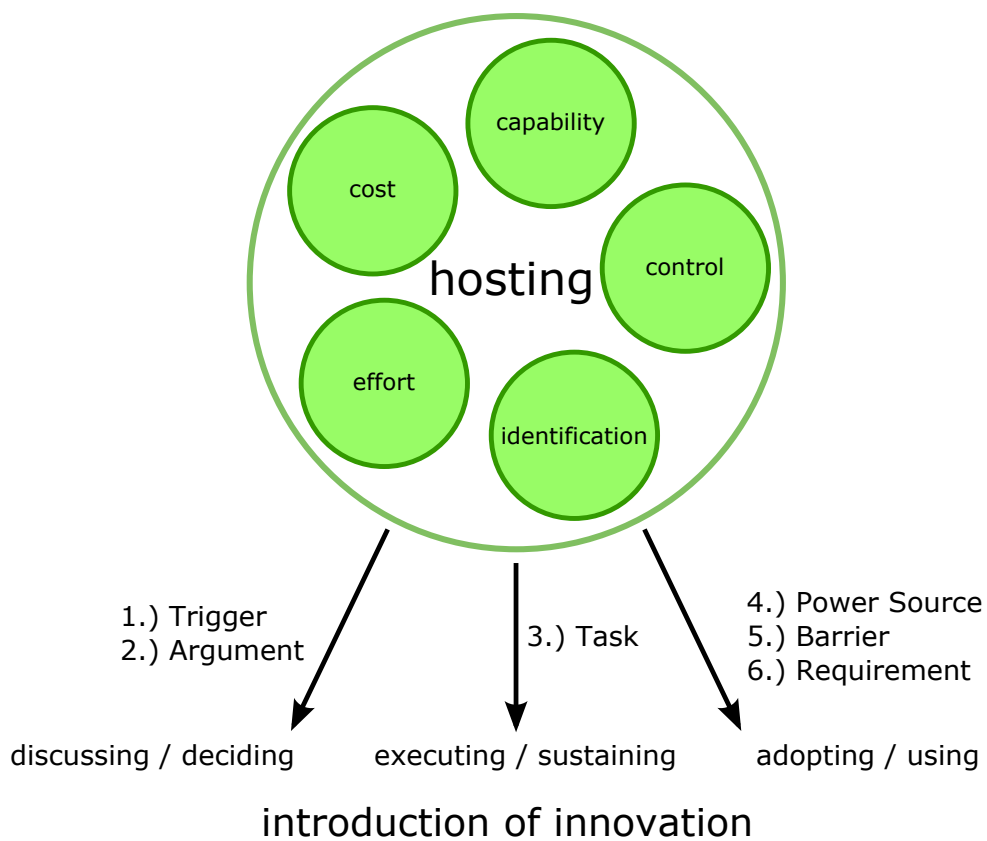


Figure 8: The conceptual relationships between the concept of *hosting* and the central activities performed during the introduction of an innovation. Hosting can serve as a *trigger* for or *argument* in the discussion and decision phase of an innovation introduction, it serves as a *task* during *execution* and *sustain* of an innovation, and can act as a central *requirement* for the *usage* of the innovation, as *source of power* for its controller and as a *barrier* to adoption.

### 3.4 Adapter Innovations

During the analyses of [partial migration](#) and [hosting](#) another concept appeared which relates to both of them but is interesting enough to be discussed independently:

**Definition 8** (Adapter Innovation). *An innovation used to make an existing innovation accessible by a third innovation.*

For instance in the project KVM, one developer used the adapter tool [Tailor](#) to import changes from the official [Subversion](#) repository into his private [Mercurial](#) repository [[997@kvm](#)]. We use the same terminology as the Gang of Five design pattern with the same name: The existing innovation (Subversion in the example) is called the “adapted innovation”, the innovation adapted to (Mercurial) is called the “target innovation”, and the innovation performing the adoption (Tailor) is called the “adapter” [[28](#)].

If we ask why such an adapter was used, we find a first set of cases in which the adapter can be seen as increasing the *independence of tool choice*, i.e. it gives each developer more choice regarding the set of tools he wants to use during development. We found three motivations for such independence: (1) As with the example just given, some developers have a *personal preference* for certain tools over others and thus need to adapt the officially offered system to be usable by theirs. [[997@kvm](#)] In some cases such a preference might be unqualified, in others it might have practical considerations such as additional features only found in the adapted innovation [[6157@bugzilla](#)]. (2) Partial migrations might fragment the landscape of innovations being used by a project and thus some developers feel the desire to use adapters to make their tool environment *homogeneous*. For instance in the case of ROX, an adapter allowed to access all parts of ROX using Git, even though only parts had been migrated away from Subversion. [[9385@rox](#)] (3) Certain tools might have high *entry barriers* for some project participants if they require complex installation or certain platforms. For instance, in the project Bugzilla the maintainer installed an [IRC Gateway](#) “for people who can’t easily get on IRC” [[6263@bugzilla](#)] to join the developer discussions from within a browser.

Beside this set of reasons why an adapter was used, we found one case in which an adapter is used as a strategic device: In the observed [episode in the project gEDA](#) it was part of the many activities by the two in-

novators to get CVS replaced by Git. Six advantages could be identified for using an adapter innovation as an intermediate step to achieving an introduction: (1) They were able to set up the adapter without needing access to the project hosting or getting the decision to set it up legitimized by the project. Thus adapters can help to *avoid decision and execution barriers* (see Section [3.6](#)). (2) The use of the adapter enabled them to start using Git in their daily work and thereby become familiar with the technology and *learn* about its problems and advantages. (3) Using Git enabled them to *demonstrate* first hand to others which impact the innovation could have if adopted project-wide [[2918@geda](#)]. (4) Instead of having to convince all project members at once to introduce Git for the project, they focus on individual developers and incrementally convince, demonstrate, and teach one after the other [[3635@geda](#)]. (5) Because Git was available to them, they could use it to introduce or propose additional innovations on top of it. For instance they installed a [Git-Web](#) repository browser [[2799@geda](#)] for looking at the source code changes in the project and proposed Git as a staging system for the work of students in their summer projects [[3068@geda](#)]. This made the target innovation even more interesting or plain necessary. And (6), the existence of the tool provided repeated *opportunity* to talk about the innovation. In the observed case, these opportunities were even given by negative events, such as the adapter innovation being currently broken [[2889@geda](#)] and the data in the target innovation out of date [[3930@geda](#)].

These were not the only reasons why the innovators achieved an introduction, but it highlights that the use of an adapter innovation can be a supportive strategy.

### 3.5 Forcing, Compliance and Decisions

This is the first section in which we try to explore the decision making processes involved while introducing an innovation. We have found that decision making about an innovation introduction can be divided into two major components: The *organizational innovation decision* and the *individual innovation decision* (Fichman calls this distinction the *locus of adoption* [[22](#)]). The organizational innovation decision is given by the decision of the project as a whole to acquire and commit to an innovation which then offers the chance to each individual to adopt this innovation. This distinction is important because the individual’s adoption is not automatically implied by the organiza-

tion's decision. But rather *assimilation gaps* between the organization's commitment to an innovation and the individual uptake might occur [25]. While in many cases the organizational decision has to precede the individual one because implementing the innovation is often only possible via the organization, it is also possible to see individual adoption precede and promote an organizational decision.

To understand both of these decisions better, we want to discuss them in turn and begin by looking at the different ways by which we have observed projects to make a decision on the organizational level:

### 3.5.1 Organizational Innovation Decisions

Well known from hierarchically structured organizations is the *authority* innovation decision which is "made by a relatively few individuals in the system who possess power, high social status, or technical expertise" [53]. Such authority to unilaterally decide on the organizational adoption of an innovation is usually located with the project leader, maintainer, or administrators of the project infrastructure. For instance, in the project gEDA the maintainer decided to migrate to *Git* as the source code management system using the following announcement:

Here's the plan going forward:

- \* I do want a stable/unstable branch/release arrangement going forward.
- \* I do not want to use CVS to maintain this though.
- \* So, I am going to go ahead and setup a git repository as the official repository of gEDA/gaf. [4123@geda]

In many cases the *maintainer's might* is so strong that the maintainer never questions his authority to make a decision in such a unilateral way. If maintainers involve their project participants, it is often only to ask for *opinions* or even only for *objections* about the plan. The second most frequent way we have seen organizational innovation decision being made is by a *collective* or more appropriately by a *representational collective* innovation decision. Rogers defines the former as "choices that are made by consensus among the members of a system" [53], but we feel that consensus is inadequate to describe that in most cases with Open Source projects a large part of

the project is absent from the discussion and decision making process. Rather, the subset of the project participants who are involved with the discussion assume the role of *representing* the project as a whole. This seems to be a necessary mechanism because of the decentralization of and fluctuation in participation and is directly connected to *meritocratic* principles: If a project member does not participate, this should not stall the decision making.

For instance, in the project gEDA one of the maintainers of a sub-project proposes to clarify the licensing scheme used for graphical symbols used in the project by posting an updated licensing disclaimer to the list [3122@geda]. Of the top twenty participants in the project by number of emails written in 2007 only nine participated in the discussion. The others (in particular two from the top five, four from the top ten) were absent from the discussion in which the license text was discussed. The innovator then takes the results from this discussion to revise the document and present it again for a final round of feedback, in which he receives four additional positive comments, two of which come from missing top 10 members.

A possible danger involved in invoking a representational collective arises if this collective is too small or staffed with project participations too low in the hierarchy that the collective lacks the *legitimacy* to actually make the decision. This problem is particularly pronounced in projects with strong maintainers. For instance, in the project Grub a set of project participants had already decided to adopt a new versioning tool and, after a waiting period for objections, began to talk about executing the innovation. It was only then that the maintainer voiced his concern [4116@grub] and thereby withdrew sufficient legitimacy to cause the innovation introduction to be pushed back into a discussion about whether to adopt the innovation at all.

The third mechanism we have found to drive an organizational innovation decision is *voting*. Famously known in the Open Source community is the Apache style minimum quorum consensus, where each project member can vote "+1" (in favor) or "-1" (against) for a proposed change, and at least 3 votes in favor with no vote against are necessary to achieve a code change [26].

The most interesting example of the use of *voting* occurred in *an episode* in the project *U-Boot* and provides many interesting hints about decision making. The vote occurred when the project leader rejected

Episode  
Licensing  
Schemas

Episode  
Git@Grub

Episode Merge  
Conflicts

Episode  
Git@gEDA

a change to the code format in the build files of the project [30660@uboot]. One developer who had not participated in the discussion so far replied to the rejection with the following sentence, which then informally triggered a vote to be started:

I vote for the new, single-line version suggested by Kim & Jon. [30667@uboot]

Note that this way of starting a vote does neither define who can vote, how the result of the vote is going to be determined, nor how long it will be run. Rather all aspects are implicit, as are the no-vote of the project leader and the yes-votes of the innovators Kim and Jon. It then takes only two more votes in favor, which are equally terse, to make the maintainer accept the decision.

We have seen not enough comparable episodes to derive strategies, but think it would make sense to hypothesize the following:

**Hypothesis 5.** *Voting can be an effective tool against individual high-ranking opponents such as a maintainer if the general opinion in the project is in the innovator's favor.*

**Hypothesis 6.** *Voting speeds up the decision process by reducing opinions to yes and no.*

Lastly, we found a series of innovation introduction episodes in which organizational innovation decisions did not occur, which we categorized as *just do it* innovation decisions.

Such *omitted* organizational decisions were particularly common when the innovator could execute the innovation to a large degree independently such as setting up mirrors of existing systems [9385@rox], *writing a new piece of documentation*, or *a tool for cleaning up white space issues*. It is a good indication of the pragmatic nature of Open Source projects and the flat hierarchies that project members feel empowered to invest time and effort to execute an innovation even though the project has not given them a mandate to do so.

To sum up, we have seen several different types by which innovation decision can be made at the organizational level and discussed some implications for the innovator.

### 3.5.2 Individual Innovation Decisions

After the project as a whole has adopted an innovation at the organizational level, we typically saw a set of ac-

tivities performed to enable the use of the innovation by the project such as migrating data [4835@freedos], installing software [2799@geda], or writing documentation [5069@geda] and we have labeled these *executing activities*. Once these activities are completed, each project member can now start using the innovation. The question of whether a project member does indeed adopt and use an innovation then, is called the individual innovation decision. In this section we discuss the results of looking at the extrinsic factors affecting individual innovation decision, namely the concepts of *forcing effects* and *compliance enforcement*.

Following Denning and Dunham's model of innovation introduction and its central concept of adoption [15], we were surprised to see that some innovations do not incur any noticeable adoption periods. Instead, adoption happens fluently and without any active resistance. How can we explain that for other innovations the adoption is the most central aspect of the whole innovation? One answer that goes beyond relative advantage of one innovation over the other is the concept of *forcing effects*.

**Definition 9** (Forcing effect). *A property or mechanism of an innovation that promotes the use of the innovation itself.*

Consider for example the introduction of the legal innovation GNU General Public License (GPL) v3 in the project GRUB. The maintainer proposed in July 2007 to switch the subproject GRUB 2 from GPL Version 2 to Version 3, because of a recommendation issued by the Free Software Foundation to switch all software of the GNU project to the latest version of the GPL. After letting the project participants discuss a little bit about the proposed license migration, the maintainer unilaterally decided [3380@grub] and executed the switch [3385@grub]. This was possible because the Free Software Foundation holds the copyright to all parts of GRUB and thus can execute such a change. If we now consider the adoption of this innovation by the project members, we see that the nature of the GPLv3 forced them to adopt the GPLv3 implicitly. This is because the GPL is a viral license and ensures that only code may be contributed to the project, which is under a license, which is compatible with the latest version used. In this case project members could no longer contribute GPL v2-only code to the project without being in violation of the terms of the GPL. In other words, their participation in the

project was now *contingent* on their adoption of the GPL v3 or any compatible license for their contributions. As such, the innovation itself has forced its own adoption based on a *legal* mechanism and the ultimate ratio of excluding project members from participating.

Such forcing effects, which make participation contingent on adoption, have been spotted to derive their power from two other mechanisms beside this *legal* mechanism: First is the existence of a *data dependency*, which is best explained by the example of migrating one source code management system to another. Because the repository from which the “official” project releases are made is changed to the new system, project participants cannot continue contributing towards such a release unless they contribute to the new system. If somebody continues to commit to the old repository, his changes are forfeited. In the project ROX, for instance, parts of the existing Subversion repository were migrated one after the other to separated Git repositories [*episode.git@rox*]. During this migration, one developer accidentally committed to the old Subversion repository after the Git repository was announced. Because he had not had time to adopt Git as client tool yet, he needed to ask the innovator to move his change from Subversion from Git for him unless he wanted his work to go to waste [*9404@rox*].

The second mechanism by which innovations can achieve strong forcing effects is by their use of systems restricting participation. An example for such a *code is law* [43] mechanism is the change of the mailing-list in the project *Bochs* to a subscriber-only mode. By changing the configuration of the mailing-list to accept only those members who have registered with it, the maintainer forces all project participants to sign up, because the mailing-list will otherwise reject their emails [*7272@bochs*]. Again we see that the underlying mechanism uses the desire of participants to participate to achieve its forcing effect.

It should be noted that forces are not necessarily perceived as negative by the project participants. Rather, and in line with results regarding the motivation for participating in Open Source projects [29, 37, 32], participants note that force has positive effects such as promoting learning of new technologies [*9369@rox*].

To sum up: We have found three mechanisms - legal, data dependency, code is law - by which an innovation can generate a strong forcing effect to be adopted. In all cases these mechanisms cause partic-

ipation to be contingent on adoption.

If we regard other innovations which do not contain such strong forcing effects, we find two more classes of innovations: Those with *expected* use and those with *optional* use:

First is the class of innovation for which the adoption is *expected* by the project members but not automatically forced by forcing effects as above. For instance, in the project U-Boot it was decided to change the coding standard used in the build-files to list individual build items on individual lines and no longer four items on a single line. This change was introduced to reduce merge conflicts, when individual items in the lists changed [*30646@uboot*]. By adopting this innovation on the organizational level, it puts only an expectation on the developers to maintain this file-format. Such expectation can come in various degrees of formalization, ranging from informal *pleas* to formal *guidelines* that primarily work via *social* mechanisms.

If developers do not comply with the given expectation, additional mechanisms outside of the innovation are required to detect and correct such violation which leads us to the concept of *compliance* and *compliance enforcement*:

**Definition 10** (Compliance). *The act of following a given norm or standard of using an innovation.*

**Definition 11** (Compliance Enforcement). *The act of ensuring that a given norm or standard is complied with.*

In the above example, the project U-Boot uses a *gate keeper* strategy to ensure compliance with the norm of single items per line: To contribute to the product, the contribution needs to pass through the watchful eye of a module owner (called ‘custodian’) and/or of the project maintainer. If a violation is detected, the custodian or maintainer in their role as gate keeper will reject the patch (or correct it themselves) [*32452@uboot*]. If we reason about the underlying mechanism from which the gate keeper as a *compliance enforcement* strategy derives its power, we can see that it also uses the data dependency on the project repository and the underlying contingent participation.

A second compliance enforcement strategy that again uses the contingent participation as the ultimate source of power was found in the project MonetDB, where the maintainer sends an email to the list

after the opening of each release branch, reiterating the rules of which branch to commit to. To enforce these rules, the maintainer then states:

Any violation of these rules might be "punished" by a forced undo of the respective changes. [54@monetdb]

The *forced undo* of a commit to the repository is thus another way to invalidate effort and sanction those activities that are in violation of the rules set by the project.

An overview of the forces affecting the adoption of an innovation by individuals can be seen in Figure 9.

If we try to abstract again from the strategies, we see that both strategies are similar in that they work primarily to ensure some quality of the code. It is unclear how a compliance enforcement strategy can help in assisting in the adoption of innovations that do not manifest in code.

Indeed it might turn out that process innovations which lack any such relation to code are the ones that require strategies from the innovator that clearly go beyond forcing effect and compliance enforcement. To stress this point, consider the following quote from a discussion about how to encourage the users of the project to report bugs directly in the bug tracker:

I don't think we could be more explicit or efficient in discouraging the use of the mailing list for bug reporting without the risk of being perceived as discouraging in this. [4752@argouml]

Put more abstractly, while the forces to achieve adoption draw their power from restricting participation, the innovator has to carefully balance the use of this force, because if forced too much, the participants in the project can turn their participation to be a power source as well. If we consider the forking of the project GCC, XFree86, and TWiki, we see that in each case the project lead had restricted or threatened to restrict participation so much that project members quit and forked the project<sup>10</sup>. For instance, in the project TWiki the project's backing company

<sup>10</sup> Unfortunately there is only anecdotal material on forking in the Open Source world such as Rick Moen's "Fear of Forking" [http://linuxmafia.com/faq/Licensing\\_and\\_Law/forking.html](http://linuxmafia.com/faq/Licensing_and_Law/forking.html) or Wikipedia's article on forking [http://en.wikipedia.org/wiki/Fork\\_\(software\\_development\)](http://en.wikipedia.org/wiki/Fork_(software_development))

locked out all developers from participation and required that they agree to new terms and conditions before resuming their work. The community got so upset by this and the associated transfer of rights to the company and its leader that a large number of project participants including many of the long-term core developers forked the project and created FosWiki.<sup>11</sup>

As a last point of discussion, we want to relate the insights about the concepts of forcing effect and enforcement to our model of decision making discussed at the beginning of this section.

If we make the distinction between organizational and individual innovation decisions, then the forces - as discussed above - primarily affect the individual's decision to use the innovation or not. But if the forces are anticipated by the individual, they can also shape the discussion leading up to the organizational decision. By this, forces can have ambivalent effects as a strategic device of an innovator:

**Hypothesis 7.** *While forcing effects and mechanisms for compliance enforcement can increase the speed of individual adoption, they can increase resistance with regards to the organizational adoption.*

### 3.6 Episode outcome

This and the following sections are very much only static typologies and not strongly analytical. Yet, understanding the different types of outcomes and different types of innovation (see Section 3.7) is a prerequisite for talking about innovation introduction.

We start by discussing the different ways that we have seen *episodes* end. Because episodes are our primary way of aggregating all messages belonging to the same innovation introduction, a successful episode is the primary goal for an innovator. Unfortunately *success* is hard to define in the context of Open Source projects, as very little external success measures such as share holder returns, margins or profitability make sense [14]. For within this study, we define *success* in the following way:

**Definition 12 (Success).** *An innovation is successfully introduced, when it used on a routinely basis*

<sup>11</sup>An account given by the forking developers can be read at <http://blog.wikiring.com/Blog/BlogEntry28>, while no official statement by the TWiki project leaders can be found about the fork.

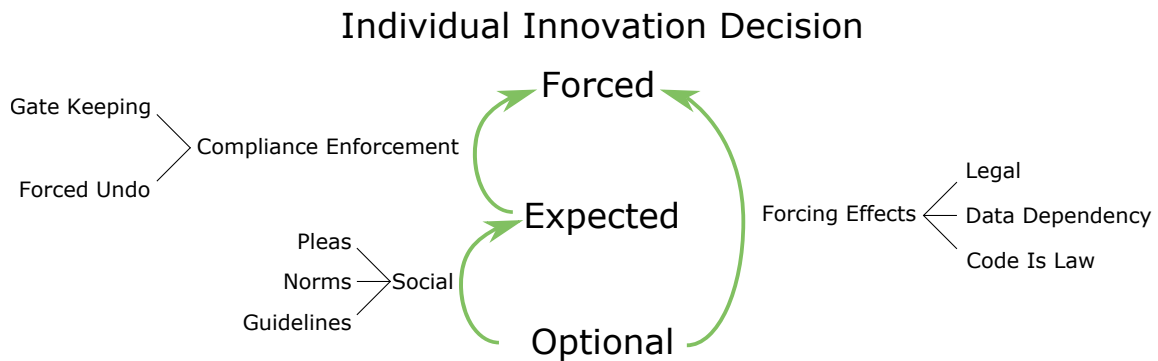


Figure 9: The proposed model explaining how an individual innovation (adoption) decision is shaped by the social actions (left side) and attributes contributed by the innovation (right side).

and it has solved the problem it was designed to solve or attained the goal it was designed to attain.

The first part of this definition regarding routine usage is a direct adaptation of Denning and Dunham’s key measure to innovation success - “adoption of a new practice by a group” [15]. The second part was added, because we found some innovations which did not aim for adoption but rather tried to increase the likelihood of a certain event.

Given this definition, we can sort episode outcomes into three big categories: *success*, *failure* and *unknown*. The *unknown* category is necessary, because we can not always determine whether adoption occurred (coded as *unknown adoption*) or whether the innovation achieved its goal (coded as *unknown success*).

As an example for the difficulties to determine the success of an episode consider the following episode in the project *ArgoUML*, which will serve as a source of examples for most concepts in this section. The maintainer had proposed to join the *Software Freedom Conservancy (SFC)* with the goal of avoiding some of the hassle of handling money as an Open Source project. Shortly thereafter the legal paperwork was filed and *ArgoUML* was part of the SFC. When *ArgoUML* participated in the *Google Summer of Code* and a money transfer from Google to the project had to be handled, the project used the SFC to take care of the transaction.

Using the Conservancy to handle money is definitely a usage of this innovation and a first step towards adoption. Unfortunately it is hard to decide at which point we could say this has been incorporated sufficiently to be a practice or routine, because handling

money in *ArgoUML* remains such a rare event that our sample period only includes this single case.

Similarly we find that measuring the success of the introduction by its goal - less hassle - is similarly difficult, since (1) the usage-interactions with the innovation are separated from the mailing-list and (2) measures for goals such as less hassle are hard to come by. Since the maintainer never reports on reduced work for him, we are left to ask whether we can call the introduction a success.

To determine innovation success, we have thus tried to be conservative in our judgment. We have tried our best to gather evidence with regards to goal attainment and the use of the innovation becoming routine and established behavior. Despite our efforts, we cannot guarantee that an innovation is abandoned shortly after our sample period ends or side effects appear that prevent it from achieving its intended goal.

With these difficulties regarding the notion of success in mind, we can come back to the viewpoint of the innovator, who is interested in these outcomes, because they might provide him or her with insights about why an introduction failed. The rest of this section therefore discusses those effects which had a substantial impact on an introduction becoming a failure.

First, it makes sense to use a simple phase model of innovation introduction that distinguishes between (1) *discussion* leading to an organizational innovation decision (see Section 3.5.1), (2) *execution* leading to a usable innovation and (3) *usage* of the innovation. These phases can then be used to group reasons for failure. We find that each phase provides opportunity for the introduction to fail and want to give an ex-

ample for a failing episode in each phase before we discuss all the different types of reasons found. For simplicity, all examples were chosen from the project *ArgoUML*, which is developing an UML CASE tool in Java:

An example of a proposal that fails in the discussion stage can be given by *an episode* in which the maintainer of *ArgoUML* proposes the use of *branches for cooperative bug fixing* in contrast to the existing use of patches attached to bug-tracker items. This proposal is very tightly scoped (see Section 3.2) and directed specifically at the two highest ranking core developers (beside the maintainer himself). In the ensuing discussion these two core developers *reject* the given proposal by (1) a series of arguments such as inefficient operations in their development environment [4773@argouml] or the lack of an established tradition of using branches for experimental work [4784@argouml] and (2) by enlarging the *enactment scope*. The innovator tries to counter these arguments, but fails to draw the core developers back into the discussion, a phenomenon we call a *dead end*. The episode thus ends rejected.

Examples of failures to execute are the rarest category by phase and the example I give is somewhat lacking: In the project *ArgoUML* a *translator* proposes to join forces with the translation teams of distributions like Debian or Ubuntu to get *ArgoUML* translated more rapidly [4691@argouml]. The idea is well received by two project members [4694@argouml][4696@argouml] up to the point that one of them declares that he will contact some other distributions about the idea. That neither he nor the innovator ever report back on their attempts to get in contact with the translation projects of any distribution (or presumably never contacted them), is a pointer to a first class of reasons why innovation introductions fail: lacking commitment by the innovator. If we dig a little deeper, we learn another reason why this introduction failed: The second proponent - a core developer - explores the state of *ArgoUML* packages with the major distributions and finds that they are in such an out-dated state that he *abandons* the discussed *episode regarding translating ArgoUML by using distributions* in favor of starting a new *one regarding packaging*.

As an example for an episode which failed in the adoption and usage phase, we find a very short episode in which the maintainer defines the new role of *an observer on the mailing-list*, responsible for wel-

coming new project members and managing duplicated or incorrect information in the bug-tracker. He decides to create the role by his power as a maintainer and defines (executes) it within the proposing email. Yet, he *fails to get the innovation adopted* by the project, because nobody volunteers to take up the role.

These three examples should provide a sense of the wealth of ways an innovation introduction can fail and that it is useful for the innovator to be aware of mechanisms that might prevent him from being successful. In this second milestone we thus have started to categorize the failures to introduction an innovation and concentrated on the first phase (discussion), because it provides the most examples.

We found not surprisingly that *rejection* by project members is the basic case for episodes to fail: During discussion, the proposed ideas meet with resistance and the innovator cannot convince the other participants to decide in favor of his innovation. We found that such rejections contain two special cases: (1) If a proposal is rejected by a high-ranking project member, then the end of the discussion is often so strong and abrupt, that I use the special term of a *killed* proposal. These drastically stopped introduction attempts might provide central insights when trying to overcome *maintainer might*. (2) As a second special case, some proposals are not outright rejected, but rather *postponed* for revisiting the proposal later. The innovator should carefully consider in such a situation whether time is working in or against his favor. In the project *KVM*, for instance, the maintainer waited four weeks until participants had become more comfortable with a partially introduced innovation to complete the migration (see Section 3.1).

While this first group of failures highlight the ways that the other project members can make an introduction fail, a second big group of failures highlight the importance of the innovator for each introduction episode. We call an episode *abandoned* if the innovator fails to continue with the episode, despite no obstacle to innovation success being present except the innovator's own ability to invest time, execute or explain the innovation. In fact, many such episodes highlight that ideas are easier proposed than followed through. An archetypical example can be seen in the project *Bugzilla*, where the maintainer proposes an *optional design review process* to be added to the development process. Such a review was meant to be an optional process step for any developer who is unsure

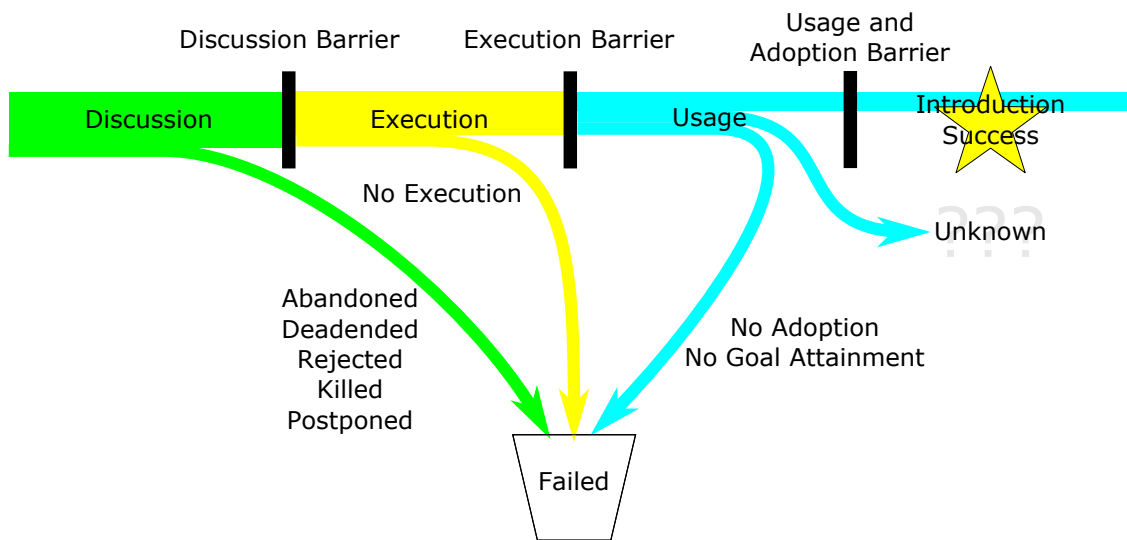


Figure 10: The reasons identified by which introduction episodes fail given a simple phase model of innovation introduction. Failure reasons can be interpreted as barriers which the innovator needs to overcome to proceed into the next phase of an introduction.

whether a contribution they want to work at would be accepted based on its design. The maintainer’s goal with this proposal was to reduce extraneous effort spent on contributions which are rejected after implementation because of a “fundamental design problem” [6943@bugzilla]. The maintainer receives two clarifying (if somewhat critical) questions just six minutes after he sent his proposal to the mailing-list, to which he fails to reply. Given that he as the maintainer has demonstrated time and again that he is able to convince others of his proposals, we are left to conclude that he has *abandoned* this proposal for reasons unknown to us. For an innovator we can conclude from this that his planning of available time and resolve to stick to the episode is a central prerequisite to introducing an innovation.

A third type of failure reasons during the discussion phase origins again in the behavior of the innovator. We have seen several cases in which the innovator failed to attract the interest of the project with his proposals or arguments. We call such situation *dead ends*, in which the last email in an episode is written by the innovator. An episode ending in *dead end*, is always also an *abandoned* episode, because the innovator could have picked up on his unreplied-to message. A dead-end on the other hand gives us different insights about the difficulties of capturing the interest of

the other project members and the tactical dimensions of an innovation introduction, such as when to write a proposal (not shortly before a release<sup>12</sup>), how many proposals a project can digest at once, how to scope a proposal (see Section 3.2), whom to address [54], how much effort to invest before proposing [52], etc.

Out of a total of 72 episodes, all 38 failing ones fit into one or more of these categories. The categories are not exclusive, because each associated failure reason implies a significant contribution to the outcome of that episode, of which there can be several. For example consider the following episode at ArgoUML that was *triggered* during the *discussion of joining forces with the translation teams at distributions like Debian*. One of the core developers explored the versions of ArgoUML shipped in distributions and proposed to discourage shipping unstable versions which might harm the reputation of the project [4697@argouml]. He received two replies, the first of which proposed to change the versioning scheme from *odd/even for unstable/stable releases to a milestone based naming scheme* to make unstable packages more ex-

<sup>12</sup>This is well exemplified by two nearly identical propositions shortly before and shortly after the release of version 4.4 in the project *Xfce*: The proposition shortly before the release fails to even draw a response [12700@xfce], while the proposition after the release by a peripheral developer is a success [12949@xfce].

plicit [4698@argouml]. The second email rejected both this proposition to change the naming scheme (as in line with Open Source community norms) and discounts the problem in itself [4701@argouml]. Given that the innovator does not return to the discussion, we can identify two significant contributions to the failure of this episode. The first being *rejection* by the project peers and the second being *abandonment* by the innovator.

A concluding overview of the reasons for failure of Innovation introductions can be seen in the Figure 10.

### 3.7 Innovation Typology

In this and the previous milestone [49] we have encountered a large number of innovations during the discussion of the phenomena associated with introducing them (see the preceding Sections on *hosting, enactment scopes, partial migration, adapter innovations, forcing effects, and innovation decisions*). What we want to accomplish in this section is a classification of the innovations we have seen being introduced. The goal of this classification is to provide us and eventually the innovator with a typology by which we can select appropriate *introduction strategies* according to the type of innovation to be introduced.

In many of the previous sections a general understanding of the different innovation types from the software engineers' standpoint were expected. For instance, when discussing hosting (see Section 3.3), we noted that this applies mostly to service innovations. This section is to expand on and formalize this implicit understanding.

It should be noted that this typology has never been in the dedicated focus of analysis during the last milestone. Thus, its result might not be without wrinkles.

For the following discussion keep in mind our definition of innovations as *changes to any kind of process in a project* (see Section 1).

We have found the following four central categories for innovations:

- Process Innovations: Innovations that modify, add, remove, or reorder process steps of the processes used in the project. As an example, consider the use of a *merge window in a fixed interval release scheme*. Instead of releasing the project after a set of features has been completed, a fixed interval release scheme aims for releases at given points in time (two to six months per release are

common). To achieve such regularity, a merge window is often introduced to restrict the time in which new features can be contributed. After the merge window closes, commits are often restricted to bug-fixes and localizations with the intent to encourage testing and debugging of the new features (this intention is not necessarily achieved, as short merge windows make it often necessary to directly start working on new features to be included during the next merge window). The use of this innovation thus primarily changes the process by which software is developed in the project.

- Tool Innovations: Innovations that involve the use of software by an individual developer. A typical example of a tool innovation is an *inline documentation tool* such as Doxygen.<sup>13</sup> Such a tool first defines a set of keywords to be used inside of source code comments. Given source code annotated with such keywords, the tool is then able to generate documentation of the application programming interface (API) of the annotated software. Note that the introduction of an inline documentation *tool* is in many cases combined with rules and conventions that determine when and how to add such documentation.
- Service Innovations: Innovations that involve the shared use of software or data by several developers typically over a network connection such as the internet. The important difference to tool innovations is that the piece of software or the shared data is not run or managed individually, but rather that several developers need access to the same running instance of a piece of software or the same set of data. In contrast to tools, service innovations thus require hosting on a server which is available to other developers (see Section 3.3).

As an example, consider a *continuous integration service* which is being run on a project server to detect failures to build the project and pass an associated test suite [27]. This innovation could be a pure process innovation if each developer would by convention build and test the project locally before each commit with an up-to-date version or a tool innovation if a developer used

<sup>13</sup><http://www.doxygen.org>

tool-support for this. Yet, not sharing a continuous integration system reduces the set-up effort for each developer and more important reduces the reliance on each developer to actually perform the integration (this reliance might not hold if build and test take considerable time or effort). The price the project has to pay to achieve these advantages of running this innovation is the costs for operating a public server and maintaining the continuous integration software on this server. Note that it is an attribute of this innovation that the decentralized approach to continuous integration is possible at all and that not all service innovations can be transformed into a tool or process innovation.

- **Legal Innovation:** Innovations that affect the legal status of the software being developed or of the project as an organization. This category targets in particular the choice of licenses used for releasing the produced software to the public. As a non-license example, consider the example of joining the *Software Freedom Conservancy (SFC)* - a non-profit organization that provides legal, fiscal, and administrative support to member projects. A member project can, for instance, make use of the SFC to receive money when participating in the *Google Summer of Code*.

As can be seen in the examples, these categories are not exclusive and many innovations are composites of these primary types. For instance, the *source code management system Subversion* consists of client applications for use by the individual developers, a server application which is shared by all project participants, and process steps for using a centralized versioning tool like Subversion.

In particular, all innovations contain at least a minimal aspect of a process innovation (otherwise they would not be innovation at all). For instance, when introducing a tool which can check and correct coding style issues, it must include at least the new process step to use this tool. To alleviate this problem, we have tried to determine a *dominant innovation type* to distinguish innovations. In the above example we have chosen to model Subversion primarily as a service innovation.

Beside these major categories, we found the following minor categories, which occur less frequent:

- **Documentation** - Innovations that put knowl-

edge, processes definitions, rules, guidelines, or instructions into a document form.

- **Conventions** - Innovations that define rules or schemes for the execution of process steps such as how to select names for releases, how to format source code, or which kind of programming language features to use and which to avoid.
- **Social** - Innovations that are based on social mechanisms such as meeting in real life or discussing hobbies and interests beyond Open Source.

Another set of innovation types discusses the relationship between an innovation and other innovations, which might be useful for (a) devising strategies to introduce a certain kind of innovation by means of intermediate steps and (b) as a starting point for thinking about how to devise novel innovations to be used in Open Source projects (compare with the innovations devised in [16]):

- **Adapter Innovations** - An adapter innovation allows the use of an existing system or innovation by another system or innovation. For example an adapter might make it possible to access a *Subversion repository* by means of a *CVS client*. As we have discussed in Section 3.4, adapter innovations provide many strategic advantages for innovation introduction.
- **Follow up innovations, Optimizations, Tunings, Configurations, Reinventions** - An innovation that is triggered or made specifically possible by a preceding innovation.
- **System split** - An innovation that is created by taking an existing innovation and creating separate accounts/domains/fields of application and possibly storing these in a novel system.
- **Unifications** - Innovations that subsume the data or capabilities of two or more innovations.
- **Preventive Innovations** - An innovation that tries to prevent a problem with an existing innovation from occurring.

## 4 Further Work

The research into the aspects that influence innovation introduction into Open Source projects has made some important progress during the last 9 months. Concepts have been developed by axial coding to a degree that it makes sense to provide summaries of their implications for the innovator and embed them into our model of innovation introduction. On the other hand, the analysis remains shallow in many aspects and its links to related work stay relatively weak, as our talk at the Pfadkolleg research workshop [48] has shown. The following tasks thus remain open for the next and probably last milestone:

- Deepen our understanding of the most important existing concepts.
- Broaden our conceptual base by adding analyses of additional important concepts such as tactical and strategic elements, maintainer might, the role of tool independence, etc.
- Link our results to the related work, especially in relationship to the literature on organizational decision making and system theory, such as the Garbage Can Model [11], Path Dependence [1], Self-Organization [44], Heterarchical Organisations [33, 66], Communities of Practice [39, 64], Social Network theory [35] (considering effects and principles such as the “0-1-2 effect” [2], “preferential attachment” [47] or “triadic closure” [36]).
- Explore if different perspectives or views on our data using conceptual frameworks such as Actor Network Theory [40] might help to reveal additional insights from already explored data.
- Include a revised treatment of the previous work done in our work group with regards to innovation introduction, such as the mediator experiment [58, 50], insights with regards to zero-acquaintance approaches [54], and the role of gifts in proposals [52]. This should also include the work which is still ongoing, such as the JUnit introduction at FreeCol and the introduction of security enhancements at Wordpress.
- Discuss the introduction of one or two important innovations such as *Git* across several projects.
- Provide an in-depth discussion of one complex innovation introduction at one project.
- Perform theoretical sampling to consolidate the existing insights and achieve saturation of concepts.
- Transform some results into representations suitable for publication.
- Explore the connection of innovation introduction to software design discussions. In particular Flore Barcellini et al. have extensively explored this space of decision making, which has a direct impact on the evolution of the software product (while innovation introduction focuses on the evolution of the software process) [5, 7, 4, 6, 55].
- Discuss the implications of radical change vs. evolutionary change in innovation introductions and associated concepts such as capacity for change, and relate these insights to the capacity of rewriting, reengineering, refactoring the software product of an Open Source project.
- Extend and improve the innovator’s guide (see Section 5) and consider getting it validated.

### 4.1 Proposed dissertation outline

The following list represents a possible outline for the dissertation from the most optimistic stand-point. Since only those items in the following outline that are marked by an asterisk\* have been analysed at all and those marked with a dagger† are analytically complete to be included in the dissertation without much further thought, it will be necessary to trim this outline to some sub-set of the proposed points:

- Motivation\*
- Definitions
  - What are Innovations?\*
  - What is Open Source?†
  - Success with regards to Innovation Introduction\*
- Methodology and Data source
  - Grounded Theory\*
  - Gmane and GmanDA\*

- Unit of analysis, aggregation to episodes, central concepts.\*
- Visualization\*
- The Process of Introducing an Innovation
  - View 1: Stage Model of Innovation Introduction\*
  - View 2: Resource acquisition view
  - View 3: Network-Actor view
  - View 4: Garbage Can view
- Understanding Decision Making in Non-hierarchical Organizations
  - Organizational and Individual Innovation Decisions\*
  - Skipping Decisions and Premature Use
- Special Concepts in the Introduction of Innovation
  - Maintainer might
  - Timing issues of innovation introductions, such as *participation sprints* in volunteer organizations, the use of appropriate triggers, times to avoid.
  - Partial Migrations\*
  - Tool Independence, Adapters\* and the absence of IDEs
  - Enactment Scopes for Process Innovations\*
  - Radical vs. evolutionary change
  - Strategic (between threads) and tactical (within threads) Dimensions in Innovation Introduction
  - Learning of Innovations and the role of narration\*
  - The effects of force, enforcement, policies or mandated usage on innovation introduction (see for instance [41] for a corporate perspective on the same issue). This can include discussion on gate-keeping vs. watchman styles on enforcement, social pleas vs. Code Is Law strategies and the role of adapters to circumvent enforcement.\*
  - Networking effects such as increasing returns for innovation introduction.
- Strategies and guidelines for creating new innovations and an innovation typology. Concepts might for instance include the *democratization of access* that is the driving mechanism behind wikis and decentralized source code management.
- Hosting of innovation. Includes an overview of the available choices for the innovator and the factors that affect adoption.†
- Case Studies
  - Mediator
  - JUnit at FreeCol
  - Security Innovations at Wordpress
- Common Innovations and their Introductions
  - Git, SVN and Revision Control in General
  - Status Updates
  - Bug Tracking Procedures
  - Version Naming
  - License Switching
  - Goggle's Summer of Code
- Conclusion, Outlook and Further Work
- Bibliography
- The Little Innovator's Guide
- Glossary

## 5 The Little Innovator's Guide

### Abstract

This guide should help everybody who is interested in achieving changes in an Open Source project.

#### 5.1 Prerequisites

Before setting out to make changes to the Open Source project of your choice, you first need to make sure that you - the innovator - are aware of the following:

- Introducing an innovation is a time-consuming matter and can and should not be attempted without dedication and sufficient time in the next couple of months.
- Introducing an innovation can harm a project, cause tension, stress, and conflict. It is your obligation as an innovator to assure that you do not cause such harm unnecessarily.
- Achieving change requires an open mind to the concerns, interests and attitudes of the other participants in the project. If you are too set on your own private problems and goals, then you will likely fail to achieve anything positive.
- Being open includes being honest about your goals and your affiliation (in particular if you are paid to do the job), so that the project can judge you based on what you really are.
- Changing the hearts and minds as a stranger is very difficult and probably dangerous as well, because you do not understand the reasons for the status quo. Making a call for change without being part of the project is thus likely to fail. Instead, start by contributing to the project. If you are established and have proved that you are capable and committed, changes will be much easier. If you really want to achieve a change as a stranger, you must get in contact with the maintainer or a high ranking person in the project and convince him explicitly.

#### 5.2 Getting your goal straight

Before you approach a project with your new idea or concern, you should first take a step back from your

email client and make sure that you understand your own goal or problem with the project sufficiently.

To this end, try to write down what you want to achieve (your goal) or what you believe the problem currently is. For instance, if you are fed up with the poor security track record of your project resulting in many vulnerabilities being published, you might want to jot down as a goal:

I want my project to be more secure!

Or if you like to think more about the issue as a problem, you might say:

The number of vulnerabilities found in my project is too high!

In many cases you might already have a pretty good idea what to do about this problem, but I suggest you take ten minutes and consider breaking your goal down in more manageable chunks, so that from one single sentence you get a more substantial break-down of the problem or goal at hand. To achieve this, write down the causes of the problems and what could achieve the goal.

In our example of achieving more security or dealing with too many vulnerabilities, we might conclude that the reasons for too many vulnerabilities is the result of using dangerous API functionality and lacking peer review, while ways to achieve the goal could be to use static type checkers or use a framework with validates input.

You can continue this approach several times by looking at each resulting cause or way to achieve a goal and look for its causes or ways to achieve it. Like a root-cause analysis this will help you understanding the problem better and better.

A word of caution: You will notice that beyond a certain point of splitting problems into causes and goals into sub-goals, your refined points will start to become more and more abstract. So, if you arrive at reasons that are bound to human strengths, weaknesses, or characteristics, such as discipline to sanitize inputs, motivation and time to perform peer-review, or knowledge about security topics, then you have probably gone one step too far. Do not consider these as problems that you can fix, but rather think about them as the boundary conditions within which you have to operate.

At the end of this process, you should have a good sense for the problems at hand and a vision for what you want the project to be like.

If you get stuck in the process and feel that you do not understand the problem sufficiently, jump two sections down to Proposing.

### 5.3 Getting a solution

During the last step, you will have noticed that some of the problems and goals will have tempted you to go right ahead and get them solved or achieved. In the above example, when finding the problem of dangerous API-access, you might have feel tempted to dive right into the code and get rid of all of them. I hope you have resisted that temptation and stayed a little bit longer with the problems and goals to see that there are many, many possible ways to improve your project.

If you now start looking for solutions - i.e. concrete things that could be done to reach a goal or solve a problem - keep the following things in mind:

- Using proven existing solutions from other projects is much easier than inventing solutions yourself.
- The better you understand the solution, the more likely is it that you can be a good advocate for it.
- The less work a solution causes, the more likely it is to be accepted by your project. A direct implication of this is that radical changes, which require a lot of heavy lifting before the first results can be seen, are less likely to succeed than anything which can be achieved incrementally by investing a little bit of effort over a long time.

If you found a solution that is promising to you to achieve the goal or solve the problem, the next step should be to think quickly about the consequences of your ideas. Who is affected? Which existing processes need to be changed? How would these people feel? Are you prepared to convince them that the change is appropriate for the goal?

At the end of this step, you should feel confident that your solution will help your project in getting the problem solved or the goal achieved.

If you get stuck in this step, you need to go to the next section.

### 5.4 Proposing

In the best case you can now approach the whole project with a good idea for a solution and a good sense of the problem that you are trying to solve. You might use an opportunity in which the problem is especially apparent (for instance when another vulnerability was found in your project) or when your project is discussing the future anyway.

Since this is the first time that you approach the project with a new idea, it is important to stay open to the way the other project members feel about the problem and the solution. Do not try to force a decision. This is also the reason why you do not have to think too hard about the problems and possible solutions before proposing them and why it is not a big problem if you got stuck in the previous two sections. Use the discussion with your project to understand the problem and find even better solutions with them.

There are many ways that such a discussion can evolve and it is not always easy to keep it within constructive boundaries. The following hints might help:

- Try to keep the discussion focused on the immediate future. If discussion becomes too abstract, many people quickly lose interest. This includes giving concrete descriptions what would be the implication of adopting an innovation.
- If the discussion becomes overtly complex and more and more participants do not follow the technical arguments, restart the discussion in a new thread by providing summaries of the proposed options and their pros and cons.
- External validation from other projects about how an innovation worked out for them can provide strong arguments and good sources of technical expertise in a discussion.
- If discussions become deadlock between a number of alternatives, conducting a trial of the alternatives can help make the advantages of each option clear.

Once you have identified a promising candidate for a solution, the next step becomes to achieve a decision and agreement with your project peers that you and the others really want to make the proposed change.

This should be of interest to you even if you feel like you have sufficient influence in the project to just go ahead with the idea, because it strengthens your

possibilities to point back to the decision and the legitimacy you got to make the change if you encounter resistance. Here some hints:

- Getting the maintainer to be in favor of the innovation is in most cases the most important step. Thus react to his objections, questions, and criticisms in the best possible way.
- It is not necessary to achieve a consensus with all project members, but rather you should strive to have a substantial set of supporting members and nobody who is fundamentally opposed to the change (think about how Apache projects vote - three +1s and no veto are sufficient - for a change to understand the basic motivation behind this<sup>14</sup>).
- If participation in making the decision is low, you can push the discussion by asking directly for objections and telling the project that you would start working on implementing the change in a couple of days or at the next weekend.
- If there are many proponents, very few outspoken opponents, and thus lengthy and tiring discussion, then *calling for a vote* can speed the decision process up considerably. This can be initiated by a simple single line reply such as "I vote for the new [...] suggested by [...]".

If you fail at this step because people decide against your ideas, then you should not be sad, but rather have a look at the reasons why your ideas were rejected. Are there things that can be changed the next time you propose something or were your ideas incompatible with the realities of your project? Change is hard, try smaller, more local, less abstract ideas next time.

## 5.5 Executing

If you achieve a decision at the organizational level, then you most probably will have now the possibility to put your ideas into reality. This can be a lot of work depending on your innovation and the resources available to you. Some suggestions:

<sup>14</sup>As an easy read from Roy Fielding <http://doi.acm.org/10.1145/299157.299167>

- If you need to do some work on a server, it is best to ask for an account rather than splitting responsibilities. The time available to project participants usually causes too many synchronization issues.
- Proceed timely with executing, unless you want your project to forget that you actually decided on performing the change.
- Use all opportunities you get to tell people about what you are doing. Write emails on how you installed the software, migrated the data, broke the server, switched system A offline, upgraded tool B, etc. Every chance you get to talk about the innovation and spread the word should be used to create awareness and let others understand the technology that will become available shortly.
- Involve everybody who is interested, so that people can start learning to use the new technology and become your co-innovators.

The end of this stage should be crowned by an email to the project that announces the availability of the new system, process, document or solution.

## 5.6 Adoption and Sustaining

After the announcement has been made, the primary usage of the innovation can start, but your job as the innovation is still not over. Helping your peers to get used to the innovation and seeing it well adopted into the daily processes of the project can still be a challenge and might take several months. Do not try to force anybody but provide assistance to everybody struggling.

Every couple of months you should take some extra time to check back with the innovation. First, take a step back and have a look at the innovation. Is it still used appropriately? Is it still appropriately at all? Did changes occur in the project that could allow easier and better solutions? Think about this and then, as a second step, let the project know about it. Write a short status report which takes note on the progress and successes of the new innovation, hands out praise, and encourages those who have trouble adjusting. Third, have a look whether there are any administrative tasks to be taken care of, such as cleaning-up stale bug-tracker tickets or orphaned wiki-pages, and then apply updates and security patches.

## 5.7 Write an episode recap

No matter whether your innovation introduction failed or was successful, we would love to hear from you about your experiences with changing an Open Source project. Write an email to [Christopher \(oezbek@fu-berlin.de\)](mailto:oezbek@fu-berlin.de).

## A Glossary

### A.1 Activities

**Def.:** A conceptualization of the things the author of a message has done and is reporting about or is now doing by writing an email.

An example of a type of activity that is typically reported about is *the execution of an innovation*: The innovator usually works in the background on setting an innovation up, and after he is finished he will report about the result of his work. Conversely, *proposing* an innovation typically is an activity that occurs within the email.

- *execute* (activity.execute)  
An activity which is concerned with making an innovation usable for within a project. Such an activity might include installing software, migrating and converting data, or writing documentation.

**Disambiguation:** It is not always possible to distinguish activities that are executing the innovation and activities of *using the innovation* (activity.use). To illustrate the difficulty to disambiguate to *using*, consider for example that setting-up (and thus *executing*) the *source code management system Git* involves both installation of a Git server, which is clearly *executing*, but then also often includes the migration of existing data, which involves *using* the system as well.

- *narrate execution* (activity.narrate execution)  
Giving a detailed account of the steps involved in executing a task.

**Disambiguation:** This activity is meant to distinguish messages that tell that an innovation was *executed* (activity.execute) and those that explicate *how* and following which steps this was done (activity.narrate execution). Narration seems to be such a distinctive literary form employed in email based communication, that it warranted its own code. Yet, often these two aspects become mixed so that the following way to disambiguate can be used: A narration involves at least a minimal sequences of steps that could be used to reexecute the setting-up of an innovation.

- *propose* (activity.propose)  
To propose an innovation is the activity to start a discussion about an innovation. Important properties of a proposition are its intention (which might not always be possible to figure out) or for instance which kind of perspective on innovation is being taken.
- *sustain* (activity.sustain)  
Performing an activity aimed at supporting an innovation which is already in use. Such sustaining can include performing software upgrades to a service innovation [4417@flyspray], clarifying the capabilities of an innovation in a document [1372@kvm], granting commit rights to new developers [5320@flyspray], deleting log-files [31112@uboot], etc.
- *use* (activity.use)  
Code given to emails in which an innovation use is described or which constitute innovation use (for instance posting a patch for peer review).

### A.2 Concepts

**Def.:** A concept is every abstract, theoretical entity relevant to the introduction of an innovation that is not covered by a more specific code.

- *compliance enforcement* (concept.compliance enforcement)  
*Compliance enforcement* is an attribute of an innovation introduction related to the question how its usage or the quality of its usage is ensured. For instance, we have seen a maintainer *plea* with other members of the project to improve their changelogs [13533@xfce].

Lawrence Lessig's *Code and other Laws of Cyberspace* comes as a good source for how we can enforce compliance (in his terms 'to regulate freedom'): Law, Norms, Market, Code [43].

- *control* (concept.control)  
The degree to which the potential capabilities of an innovation and activities related to it, can be actually used or performed by the project.  
For example, in the *Flyspray* the password to the mailing-list software had been lost so that the project participants could not execute any administrative operations [5411@flyspray]. This

lack of control prompted them to migrate to a new mailing-list service.

**Disambiguation:** Note that the *concept of control* does not cover the question of who can actually perform a particular task. This question is covered by the concept of (*access*) *rights*.

- *effort* (concept.effort)  
The amount of time, physical or mental energy necessary to do something.  
Effort is a key component when trying to understand how anything gets done in a project
- *enactment scope* (concept.enactment scope)  
The set of situations in which the activities mandated by a process should be executed.
- *forcing effect* (concept.forcing effect)  
A property or mechanism of an innovation that promotes the use of the innovation itself.
- *hosting* (concept.hosting)  
Provision of computing resources such as bandwidth, storage space and processing capacity, etc. for the use by an innovation.

We identified five concepts that seem to influence most of the decisions people make towards hosting: These are (1) *effort*, (2) *control*, (3) identification, (4) cost and (5) capability (see Section 3.3).

Hosting is an aspect of innovation that applies especially to *service innovations*, since they need a centralized system to work, which in turn requires hosting. Yet, one should not forget that also other types of innovation might need some minor hosting (for instance, *process innovations* are often formalized and the resulting documents then need to be hosted somewhere).

- *maintainer might* (concept.maintainer might)  
Concept discussing the role of the maintainer with regards to the introduction of innovation. In particular we found that the maintainer has considerable influence on the course of an episode that goes well beyond what would be expected.
- *partial migration* (concept.partial migration)  
An innovation introduction in which only parts of an existing innovation are replaced by a novel one and thus two innovations co-exist side-by-side.

- *rights* (concept.rights)  
Rights (or access control) refer to the degree to which a project participant can use an innovation. For example a project might grant anonymous read access to its *source code management system* to everybody but restrict write access to those who have contributed several patches. Rights and the rights to assign rights are one of the primary sources of structural power in Open Source projects [31].
- *time* (concept.time)  
A code used for gathering all insights related to the influence of time on the introduction of innovations. This is probably a salient concept, because of the variability of available time of the Open Source project participants. Surveys indicate that more than 70% of participants spend less than ten hours per week on Open Source [30], thus setting upper boundaries for how much time can be reasonable expected to be spend adopting and using an innovation.

### A.3 Episode Outcomes

**Def.:** An abstraction of the result an attempt to introduce an innovation into an Open Source project can have. The episode outcome thus is the central measure by which the innovator would measure the success of his or her efforts.

- *failed* (outcome.failed)  
An episode fails if the innovator cannot achieve the intended goal, get the associated problem solved or convince the project members to use the proposed innovation. Such failure might occur at many different points during the introduction of an innovation: The innovator can fail to gather an *organizational innovation decision* in favor of the innovation (i.e. he is ), *fail to execute* the innovation or *fail to achieve adoption* of the innovation.
- *abandoned* (outcome.failed.abandoned)  
An abandoned episode is one, in which there is no apparent reason why the episode did not continue except for a failure of the innovator to continue to engage the rest of the project or execute the innovation himself.

**Disambiguation:** An abandoned episode contrasts best with a *rejected* episode in that the

latter contains active opposition to the proposed innovation, where an abandoned episode might even contain support for the innovation.

- *deadend* (outcome.failed.deadend)  
An *failed* innovation episode is called a deadend, if the last person to write an email in this episode in support of the innovation is by the innovator himself. Thus he was unable to draw a response from the project. Such episodes should give us insights about tactical mistakes that can be made.

**Disambiguation:** A *deadend* most naturally is always also an *abandoned* episode, because the innovator could have always come back and restart the discussion by new email. So, to disambiguate we always code a deadend if the innovator is the last person to write a message with regards to the innovation.

- *no adoption* (outcome.failed.no adoption)  
The introduction of the innovation got to the point that it was executed and announced, but no adoption or usage beyond the innovator himself can be seen.
- *failed.rejected* (outcome.failed.rejected)  
One of the primary reasons why this episodes *fails* is because the proposals being made are actively rejected by the participants in the discussion.
- *killed* (outcome.failed.rejected.killed)  
A killed innovation introduction is one, where a high ranking member rejects the innovation and this has a substantial impact on the episode ending in *failure*.

**Disambiguation:** If it becomes difficult to deliniate a substantial and a non-substantial impact on the episode outcome, I would suggest to code rather a *plain rejected* than a *killed* outcome.

- *postponed* (outcome.failed.rejected.postponed)  
An episode ends postponed if the project rejects the innovation but says that the proposal will be revisited sometime later.
- *success* (outcome.success)  
An innovation is successfully introduced, when it used on a routinely basis and it has solved the

problem it was designed to solve or attained the goal it was designed to attain.

- *unknown* (outcome.unknown)  
Used if it was not possible to determine by looking at the available data whether the innovation was successfully introduced in the enclosing episode.
- *unknown adoption* (outcome.unknown.adoption)  
An episode outcome is unknown with regards to adoption if by looking at the mailing-list and other publicly available sources of information it is not obvious whether the innovation was adopted by its target audience or not.
- *unknown.success* (outcome.unknown.success)  
Used for episodes in which we could not determine whether the intended goal of the innovation was achieved or not.

**Disambiguation:** This code is used in contrast to *outcome.unknown.adoption*, when the idea of 'success as adoption' does not seem fitting. For instance, we have seen an innovation which does not aim for adoption as regular use, but rather aims to increase the likelihood of users becoming developers by putting links into strategic places [6190@bugzilla]. This innovation cannot be adopted, but rather we would like to call it successful, if it achieves additional developers to join the project.

## A.4 Episodes

**Def.:** A central means of aggregating message associated with the introduction of one innovation or an related set of innovation into a project.

Because the unit of coding is a single message and not the set of all messages associated with a certain introduction, we use the code `episode.<episode name>@<project name>` to aggregate those messages first.

Introduction episodes (and not innovations) are the primary mechanism of aggregation, because the episode will also contain mentioning of problems with existing innovations, discussions leading up to the idea for an innovation, the retiring of an superseded innovation, problems related to the use of the new innovation, etc.

Constructing episodes from the raw stream of messages is primarily a method to (a) organize the data

and (b) serve as backdrop for a narrative strategy to understand event data [38].

- *ask smart questions@bugzilla.integrate document* (episode.ask smart questions@bugzilla.integrate document)

This is a follow-up episode from *episode.ask smart questions@bugzilla.write document*, where a core developer condensed a guide for how to ask good questions and is now asking for inclusion of the resulting document into the process by which users can write support requests to the project.

This proposal is not discussed but rather another core-developer with control over the mailing-list service directly executes it by adding a link to the document from the subscription page of the mailing-list.

It is unclear from the data whether the innovation achieved its intended effect of reducing the number of low quality support requests.

- *ask smart questions@bugzilla.write document* (episode.ask smart questions@bugzilla.-write document)

One of the core developers in the *project Bugzilla* offers to the project a condensed version of a guide by Eric S. Raymond on the question of how to ask smart questions<sup>15</sup>. The core developer executed this innovation before asking the project for feedback, which - after the fact - is positive. When he offers the document, the innovator also proposes to integrate the document more closely into the process by which users come to write support requests to the project. This is covered in the episode *episode.ask smart questions@bugzilla.integrate document*.

- *branch for patches@argouml* (episode.branch for patches@argouml)

A small episode in which the maintainer and the top two developers in the project *ArgoUML* discuss the possibility of using branches for collaborating on patches. The maintainer brings the topic up, when he notices that the two developers are using attachments in the bug-tracker to collaborate on a feature. Both developers *reject*

the given proposal by (1) a series of arguments such as inefficient operations in their development environment [4773@argouml] or the lack of an established tradition of using branches for experimental work [4784@argouml] and by (2) enlarging the *enactment scope* of the proposal: The maintainer had proposed the use of branches primarily in the given situation but one of the developers brings up the implication of using branches in general as an effective argument against the proposition.

The episode is restarted two months later, when the maintainer proposes to use branches for the contributions of students as part of the *Google Summer of Code*. Taking this indirect approach, he achieves that within a month the use of branches has been adopted by the two previously opposing developers, and that over time even one of them explicitly suggests to a new developer to use branches for the development of a new feature [5681@argouml].

- *git@geda* (episode.git@geda)

This episode is about the introduction of the *decentralized source code management system Git* into the project *gEDA*. Because it is a very complex and long episode, continuing over more than one hundred messages, the following summary is naturally a drastic simplification of the real events, while trying to include the essential parts of the episode:

The successful introduction of *Git* into the project *gEDA* can be traced to a period of about six months in which the two innovators - both core-developers - continuously supported the introduction. Operating within their bounds of available resources and independently from the project itself, they set up an infrastructure centered around Git for working with the project source code and repeatedly promoted the use of Git for instance by linking to external opinion pieces on SCM choice [3954@geda].

With these activities they prepared the stage for the switch to Git, which occurred when the project *decided to adopt a new branching scheme*. The maintainer unilaterally makes the choice in favor of Git to support the new branching and release arrangement [4123@geda] and the actual conversion of the *gEDA* project repository

<sup>15</sup><http://www.catb.org/~esr/faqs/smart-questions.html>

takes place within two weeks.

There are numerous sub-episodes, which have only partially been explored, for instance about the resulting processes of using a decentralized version control system or the role which individual tool preferences can play during an introduction.

- *git@rox* (episode.git@rox)

I think I am finished with understanding this episode (at least I have coded all emails that contained the word GIT in this mailing-list).

Thomas proposed (one could say that he even let the project know) to migrate Rox-Filer from SVN to Git, and executed this proposal within 24 hours and two messages of support (In other words we did not have any decision process to understand here).

This episode is a good example of a partial migration, since Thomas only migrates the core application (Rox-Filer) to git and then explicitly waits what will happen.

The next large sub-episode to consider is centered around Lucas Hazel as pushing for migrating more and more applications to Git (episode.git@rox.migration.Lucas). This episode fits well with our discussion about partial migration. But has a couple of interesting gotchas to consider: Even though he migrates a lot of repositories, none of these ever end up in being official, as it seems, but are abandoned and migrated later on again, this time officially.

We then have several sub-episodes:

...translations - A discussion how to handle translations, which is decided to be done using Git in the end.

...adoption - Several episodes, showing how Git was adopted by the project participants (I tagged all episodes in 2007 regarding the use of Git)

...migration - Three episodes, in which the migration is continued regarding several aspects such as enabling nightly builds and correcting release scripts.

- *git@uboot.merge conflicts* (episode.git@uboot.merge conflicts)  
See also concept.discussion

This episode is about changing the fileformat of the MAKEALL file to be more resistant regarding merge conflicts by placing each item on an individual line.

I could not trace this back to the original proposition of the idea that is mentioned in this email ('as suggested by jdl'), but maybe one could interpret this another strategy, that from a diffuse set of emails an execution could be generated directly.

Wolfgang is against this change and thus blocks it.

Stefan in reply to this supports the change and implicitly starts a voting by voting in favour.

It gets one immediate response which is also in favour and another vote in favour which is more technical in nature and tries to argue about the advantages of the change (static aesthetics and aesthetics of change).

All three votes in favour are not replied to by Wolfgang.

Then the discussion gets sidetracked by another related but not identical request to modify the makefile to allow for personalization.

It is only when another core developer picks up the issue 30 hours after the initial vote, that Wolfgang accepts the overwhelming opposition to his preference.

I really did try to find out which email this 'as suggested by jdl' originates from.

- *packaging@argouml* (episode.packaging@argouml)

This episode is triggered by *an preceding episode* in which a translator

- *translation@argouml* (episode.translation@argouml)

Hans only proposes to cooperate with Debian on translations in between his message, but notes that one prerequisite for this would be that a newer version of ArgoUML would be distributed by Debian.

This prompts Tom in ref.4697@argouml to look at the current situation regarding the versions distributed.

- `white space@uboot.tools` (episode.white space@uboot.tools)  
Small episode in which one of the *custodians* (developer responsible for managing contributions to a certain module) offers to the project a script which can clean-up whitespaces in violation of the *U-Boot* coding conventions.

With other tool innovations similar to this script, we have seen that it is common to include them into the repository of the project. This does not occur in this episode, rather first one suggestion is made to make the source code of the tool more readable. Then two suggestions for how to solve the problem of trailing whitespace in different ways are made by project members. One of these makes use of an option in the *source code management system* used by the project and is integrated into the project's wiki page for custodians. Thus instead of using the proposed innovation, the project seems to have found a preferred solution in the discussion.

We can understand this episode as an example of (1) how execution and discussion can be reversed and combined to solve a problem or achieve a goal and (2) how innovations can relate to knowledge about innovations.

## A.5 Innovations

**Def.:** From IMM1: An innovation is something that is intended to change any kind of process in a project.

This definition easily includes tool, service, social and process innovation, but does not easily excludes day-to-day business like new software design propositions, detected bugs in the software, etc.

- `bug tracking.evaluate patches in branches` (innovation.bug tracking.evaluate patches in branches)  
A third possibility beside evaluating the patches on the mailing-list (classical peer review) or as bug tracker attachments is this innovation: Take a patch and put it into a separate branch. There it can be discussed and modified.  
  
In this episode Linus wants to figure out, what the disadvantages of using branches would be.
- `communication.irc.irc gateway` (innovation.-communication.irc.irc gateway)

A Internet Relay Chat (IRC) gateway is a web-application by which people can participate in IRC discussions of the project.

Project participants can use the gateway if they cannot or do not want to install a client-side tool or if they are blocked because of network issues.

- `Observer Role` (innovation.communication.observer role)  
A person in charge of managing all user contributed sources of information such as mailing-lists, bug-trackers and wikis. In particular the person is in charge of looking for duplicated or incorrect information, create FAQ entries of repeatedly asked questions, welcome new developers and help them get accustomed to the project, and take note of decisions and information relevant for project management and put them into appropriate places.
- `coordination.work groups` (innovation.coordination.work groups)  
A way to coordinate work in a project by creating groups of people who work together on a specific issue.
- `design.approval` (innovation.design.approval)  
A design approval is a process innovation, by which people can request their design ideas to be taken under scrutiny so that they do not waste effort on producing a patch, which is rejected on design reasons.
- `documentation.api docs` (innovation.documentation.api docs)  
A *inline documentation tool* such as Doxygen<sup>16</sup> is a typical example of a *tool innovation*. Such a tool first defines a set of keywords to be used inside of source code comments. Given source code annotated with such keywords the inline documentation tool is then able to generate documentation of the application programming interface (API) of the annotated software. Note that the introduction of an inline documentation tool which can be run by each developer individually to generate documentation is in many cases combined with rules and conventions that determine when and how to add such documentation which implies changes to the process.

<sup>16</sup><http://www.doxygen.org>

- *external.GSoC* (`innovation.external.GSoC`)  
From the Google Summer of Code (GSoC) homepage: “The Google Summer of Code is a program which offers student developers stipends to write code for various open source projects.”<sup>17</sup>.

Participating in the GSoC consists of several phases:

1. To introduce the innovation a project needs to apply to Google with a set of project ideas. This means that the project needs to execute at least two things: a.) Find and write down ideas and b.) apply to the program.
  2. After being accepted (this is a rare case in which the innovation success is not purely in the hands of the project itself), the project needs to attract applicants to these proposals.
  3. When students have been accepted by Google the project needs to mentor them to complete the tasks and most importantly integrate their work into the project code base.
- *legal.foundation* (`innovation.legal.foundation`)  
To found or join a foundation which can be used as a entity representing the project in legal, representational, monetary matters.
  - *qa.continuous integration* (`innovation.qa.continuous integration`)  
Continuously checking out the project source code from the repository, building it and running test cases. If any problems occur, the developers are notified.
  - *release process.merge window* (`innovation.release process.merge window`)  
The use of a *merge window* is a *process innovation* to increase the quality and regularity of releases. After each release the maintainer allows for new features to be merged into trunk during a period which is referred to as the merge window. After the merge window closes, commits are often restricted to bug-fixes and localizations with the intent to encourage testing and

debugging of the new features (this intention is not necessarily achieved as short merge windows make it often necessary to directly start working on new features to be included during the next merge window).

A merge window is often combined with a fixed interval release scheme and a single maintainer or *custodian* who performs the role of a *gate keeper* to enforce the merge window.

- *release process.naming of version.milestones* (`innovation.release process.naming of version.milestones`)  
A milestone naming schemes for releases (sometimes also called alpha-beta scheme) is one, in which immediately after a stable release, consecutive milestone releases m1, m2, m3,... are being released until a sufficient amount of new features has been added or it has been long enough since the last stable release.
- *scm* (`innovation.scm`)  
A source code management system (SCM) is one of the most central innovations for the use by an Open Source project because it enables the efficient cooperation of project participants on the project files. We can distinguish centralized SCMs such as *Subversion* and decentralized SCMs such as *Git*, which describes how the system stores the managed data.
- *scm.adapter* (`innovation.scm.adapter`)  
An adapter for a *source code management system* is a tool which adapts an existing *SCM* to be used by another type of client tool.  
Examples for SCM adapters include *git-svn*<sup>18</sup> and *tailor*<sup>19</sup>.
- *scm.custodian* (`innovation.scm.custodian`)  
An innovation used in particular with decentralized repositories in which the role of sub-project maintainer is created to coordinate contributions to this sub-project.  
This model of development draws directly from the way the Linux kernel is developed and is particularly suited when development can be split along architectures or modules.

<sup>17</sup><http://code.google.com/opensource/gsoc/2008/faqs.html>

<sup>18</sup><http://www.kernel.org/pub/software/scm/git/docs/git-svn.html>

<sup>19</sup><http://progetti.arstecnica.it/tailor>

- *scm.cvs* ([innovation.scm.cvs](#))  
Concurrent Versions System (CVS) is a *source code management* system which pioneered the use of branches in revision control and was from the beginning of the Open Source movement until roughly 2005 the most popular revision control system for developing Free Software. With the advent of *Subversion* this uncontested preference for CVS has subsided and today distributed version control systems such as *Git* are growing in popularity.

- *scm.dvcs* ([innovation.scm.dvcs](#))  
A distributed version control system (DVCS) is *source code management system* which does not require a central repository to *host* the revision control data. Rather, this data is replicated by each developer who is sharing in development. In particular, a DVCS commonly allows each user how has obtained such a replica (also called clone) to work independently from the original master copy. Since the replica is truly identical, what defines *the official repository* is rather a social convention. For instance in the Linux kernel this naturally is Linus Torvalds' which is being tracked by other.

This principle of *primus inter pares* (first among equals) has many important implications on power relations which existed with centralized *SCMs* because of the need to assign commit-rights to the single central repository.

- *scm.git* ([innovation.scm.git](#))  
*Git* is a *distributed source code management system* developed by Linus Torvalds<sup>20</sup> as a SCM for managing the source code of the Linux kernel. *Git* has gained widespread popularity with Open Source projects in the year 2007.

Like all other distributed versioning tools, it allows a mode of operation (pull based) that does not require anybody to be granted commit access while retaining versioning capabilities for all developers involved.

*Git*'s advantage over other distributed versioning tools is primarily its performance with the daily tasks of Open Source project maintainers.

<sup>20</sup>A good overview by Linus Torvalds himself can be found in the following video lecture Torvalds delivered at Google <http://www.youtube.com/watch?v=4XpnKHJAok8>

- *scm.git.gitweb* ([innovation.scm.git.gitweb](#))  
*Gitweb* is web application for browsing *Git* repositories. It is similar to *ViewVC* for *CVS* or *ViewSVN* for *Subversion*.
- *scm.svn* ([innovation.scm.svn](#))  
*Subversion* (abbreviated as *SVN*) is the most popular successor to the centralized Concurrent Versioning System (*CVS*). *Subversion* improves on several issues with *CVS* such as renaming and moving files, while keeping the workflow as compatible as possible.
- *wiki* ([innovation.wiki](#))  
A *WikiWiki* system (from the Hawaiian word 'wiki' meaning 'fast') is a content management system which is especially tailored towards collaborative and easy editing.

## A.6 Projects

**Def.:** An Open Source project is a cooperative endeavour to develop a software product. Unlike the term "project" in a business sense, Open Source projects are most often not closed under time and resource constraints and goals are often vague or undefined. Often Open Source projects are necessarily characterized by their use of an Open Source license (see [42] for an overview of licenses and license choice), but more importantly for is their use of an Open Source model for development. The central ingredients for this model of developing software for this research is *open participation*, i.e. the possibility of developers to participate in the project, which in turn implies many secondary attributes that must be achieved to varying degrees (these are derived from [9]):

- Open communication and Open Process: The possibility to communicate efficiently within the project and observe the development and decision processes in the project.
- Open release of code base: The possibility to work with the latest source code in the project. This should not be confused with the requirement to release the source code of a published binary software product, but rather implies that it is possible to track the development. Apple's handling of the KHTML codebase is an example of how open release can be prevented by

infrequently releasing the whole codebase as a unversioned archive and the frustration this can cause.<sup>21</sup>

- **Open environment and Open deployment:** The use of tools for development that are Open Source and the targetting of an open source platform. Both factors are not ensured by an Open Source license alone and are commonly referred to as possible “traps” by the Free Software Foundation.
- *ArgoUML* (project.ArgoUML)  
ArgoUML is a UML CASE tool written in Java.
- *Bochs* (project.Bochs)  
Bochs is a simulator for x86 hardware supporting many Intel processors up to the Pentium 4 architecture.
- *Bugzilla* (project.Bugzilla)  
A web-application for bug-tracking written in Perl. Bugzilla is part of the Mozilla foundation.
- *Flyspray* (project.Flyspray)  
A web-application for bug-tracking written in PHP.
- *FreeDOS* (project.FreeDOS)  
FreeDOS is an operating system mostly compatible with MS-DOS.
- *KVM* (project.KVM)  
Kernel based virtual machine (KVM) is an Open Source project backed by the start-up Qumranet. KVM is a virtualization solution on top of the Linux kernel for x86 hardware, so that virtual machines images can be run on a single Linux setup.
- *MonetDB* (project.MonetDB)  
MonetDB is a Open Source database for high performance applications with SQL and XML backends. The project was founded and is led by the database group at the Centrum Wiskunde & Informatica (CWI) in the netherlands.
- *ROX* (project.R0X)  
ROX (RISC OS on X) is a desktop environment like KDE or Gnome following the “everything is a file”-metaphor and is thus centrally built around the file manager ROX-Filer.

- *U-Boot* (project.U-Boot)  
U-Boot is a boot loader specialized for use in embedded systems. It supports a wide variety of architectures and motherboards.
- *Xfce* (project.Xfce)  
Xfce is a desktop environment similiar to KDE or Gnome, that tries to be lightweight and fast. The project encompasses a window manager, desktop manager, panel for starting applications, a file manager ('Thunar') and many others.
- *flyspray* (project.flyspray)  
No Definition found
- *gEDA* (project.gEDA)  
GNU EDA is a suite of tools for electronic design automation. The project produces one central set of software tools - gEDA/gaf (tools for schematics capture) - and several smaller pieces of software, the three biggest one's are PCB (software for designing printed circuit board layouts), Icarus Verilog (Verilog simulation and synthesis tool) and GTKWave (electronic waveform viewer).

## A.7 Uncategorized Codes

- *argumentation* (argumentation)  
This top-level code tries to capture all the aspects of how an innovation is argued for. For instance we might see that the structure of the discussion is very open and branches into many different threads and topics or discussion participants might use tactical devices such as ignoring particular reponses or using external validation, cross references or humor.
- *compliance enforcement.forced undo* (compliance enforcement.forced undo)  
A possible interpretation of the consequence of a forced undo might be that it invalidates efforts by somebody. Thus compliance becomes related to effort management for a violator.
- *compliance enforcement.gate keeper* (compliance enforcement.gate keeper)  
A gate keeper enforcement strategy is one, where we distinguish two areas such as outside and inside, and have a dedicated mechanism by which access to the inside area is granted.

<sup>21</sup><http://www.kde.developers.org/node/1001>

For many innovations, quality might be assured by this model.

We might further distinguish by contributor or by contribution gate keeping as for instance with centralized and decentralized source code management tools.

- *compliance enforcement.social.plea* (compliance enforcement.social.plea)

A plea in compliance enforcement is the a very slight form of moral enforcement. In this case we can read this from the sentence 'Please be kind...'.<p>

Interesting as a second point in this plea, is the use of the paranthesized plural s in 'the release manager(s)' when talking about the implications of not-compliance. He seems to indicate that is causing difficulties not only for him but other release managers as well, thus increasing social pressure for anyone who gets his subtle point

- *forces.hosting* (forces.hosting)  
This code is used if *hosting* is having an influence on the introduction of an innovation.
- *hosting type.forge* (hosting type.forge)  
Savannah is a little bit of both, as it is clearly a Forge but also a foundation hosting, because only projects as part of the GNU project are hosted
- *hosting type.private server* (hosting type.private server)  
A public server owned by a person privately.
- *innovation decision.just do it* (innovation decision.just do it)  
This innovation introduction was executed without being decided by the organization.
- *innovation decision.representational collective* (innovation decision.representational collective)  
An representational collective innovation decision is one, where the decision to adopt or reject an innovation is made by the participants in the discussion by implying that the project members present in the discussion \*represent\* the project at this moment. A typical set-up of such a representational collective is a discussion of the maintainer with a set of developers who agree or reject an innovation. The full consensus of the project then often is not invoked.

A representational collective might call upon other project members for opinions. This is typically between high-ranking project members.

The representational collective is another strategy for achieving legitimation for one's action. This also points to the weakness of the collective: If legitimation fails to be achieved, the collective will not work.

An example of such failure is the episode.git@grub, where a representational collective is invoked, but torn down by the maintainer Okuji in ref.4116@grub.

- *innovation decision.vote* (innovation decision.vote)

This decision against or in favor of an innovation is made by project participants casting a vote. We have found that such voting is highly informal, without rules regarding who can participate, how long the polls are open and typically even without a counting the resulting votes. Typical voting methods such as Apache's 3 votes in favor with no votes against [26] point towards the interpretation of voting in Open Source systems as a means to condense discussions into a decision.

- *innovation type.documentation* (innovation type.documentation)

The artifact form of the innovation is primarily a document

- *innovation type.process* (innovation type.process)

To introduce a process into a project.

Process for this matter is any generalizable course of action intended to achieve a result.

- *innovation type.process.conventions* (innovation type.process.conventions)

An innovation of types conventions, is an innovation that is primarily used to set rules or guidelines for certain activities/steps in the project that derive their advantage from being uniformous across the project.

For a coding convention our terminology can be read like this:

execute = a.) change existing artifacts to conform to new standard b.) formalize convention in a document

announce = declare the convention to be in effect  
adopt = new contributions by individual conform to the convention

- *innovation type.service* (`innovation type.service`)  
A service innovation is one, where there is a centralized or dedicated host on which a tool is installed that can be used by projects members. A service innovation is a special kind of tool innovation and often requires also some form of client-side tool to be installed.
- *innovation type.social* (`innovation type.social`)  
A social innovation is a *process innovation* that uses a change in the social interaction of the project members to achieve a certain goal. For instance, a yearly meeting at an Open Source conference uses the social ties that can be created by having met in real life to strengthen trust and cooperation between participants in a project.
- *innovation type.tool* (`innovation type.tool`)  
Tools are innovations that each developer runs on their private computer.  
Thus such an innovation will always require an individual adoption decision.  
Some tools do have dependencies or requirements that still make them require collection adoption decisions though (for instance when using a certain build-tool, then build-files need to be created for the whole project).
- *role.translator* (`role.translator`)  
A project participants who is primarily working on translating the project to one or more languages. Since translators are often only involved before releases, their influence and presence in the project is usually not very large.  
**Disambiguation:** A person who is both translating and contributing code, is more appropriately marked as a developer, if the contributed code exceeds minimal amounts.
- *trigger* (`trigger`)  
A trigger is something that causes a proposal to be made. For instance a roadmap discussion calling for future directions for development might trigger a proposal.

## References

- [1] W. Brian Arthur. *Increasing Returns and Path Dependence in the Economy*. University of Michigan Press, 1994.
- [2] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, New York, NY, USA, 2006. ACM.
- [3] Stephen P. Banks, Esther Louie, and Martha Einerson. Constructing personal identities in holiday letters. *Journal of Social and Personal Relationships*, 17(3):299–327, 2000.
- [4] Flore Barcellini, Françoise Détienne, and Jean Marie Burkhardt. Cross-participants: fostering design-use mediation in an Open Source software community. In *ECCE '07: Proceedings of the 14th European conference on Cognitive ergonomics*, pages 57–64, New York, NY, USA, 2007. ACM.
- [5] Flore Barcellini, Françoise Détienne, and Jean-Marie Burkhardt. User and developer mediation in an Open Source Software community: Boundary spanning through cross participation in online discussions. *International Journal of Human-Computer Studies*, 66(7):558–570, 2008.
- [6] Flore Barcellini, Françoise Détienne, Jean-Marie Burkhardt, and Warren Sack. Thematic coherence and quotation practices in OSS design-oriented online discussions. In *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 177–186, New York, NY, USA, 2005. ACM.
- [7] Flore Barcellini, Françoise Détienne, Jean-Marie Burkhardt, and Warren Sack. A socio-cognitive analysis of online design discussions in an Open Source Software community. *Interacting with Computers*, 20(1):141–165, 2008.
- [8] Benjamin B. Bederson, Jon Meyer, and Lance Good. Jazz: An extensible zoomable user interface graphics toolkit in Java. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 171–180, New York, NY, USA, 2000. ACM.
- [9] Alan W. Brown and Grady Booch. Reusing Open-Source Software and practices: The impact of Open-Source on commercial vendors. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 123–136, London, UK, 2002. Springer-Verlag.
- [10] Larry D. Browning, Janice M. Beyer, and Judy C. Shetler. Building cooperation in a competitive industry: SEMATECH and the semiconductor industry. *The Academy of Management Journal*, 38(1):113–151, February 1995.
- [11] Michael D. Cohen, James G. March, and Johan P. Olsen. A garbage can model of organizational choice. *Administrative Science Quarterly*, 17(1):1–25, 1972.
- [12] Susan G. Cohen and Diane E. Bailey. What makes teams work: Group effectiveness research from the shop floor to the executive suite. *Journal of Management*, 23(3):239–290, June 1997.
- [13] Juliet M. Corbin and Anselm L. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 3rd edition, 2008.
- [14] Kevin Crowston, Hala Annabi, James Howison, and Chengetai Masango. Towards a portfolio of FLOSS project success measures. In *In Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering, International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, May 25, 2004*.
- [15] Peter J. Denning and Robert Dunham. Innovation as language action. *Commun. ACM*, 49(5):47–52, 2006.
- [16] Stefan Dietze. *Modell und Optimierungsansatz für Open Source Softwareentwicklungsprozesse*. Doktorarbeit, Universität Potsdam, 2004.
- [17] Jonathan Donner, Rikin Gandhi, Paul Javid, Indrani Medhi, Aishwarya Ratan, Kentaro Toyama, and Rajesh Veeraraghavan. Stages of design in technology for global development. *Computer*, 41(6):34–41, 2008.

- [18] Gunter Dueck. Bluepedia. *Informatik-Spektrum*, 31(3):262–269, June 2008.
- [19] William N. Dunn and Fredric W. Swierczek. Planned Organizational Change: Toward Grounded Theory. *Journal of Applied Behavioral Science*, 13(2):135–157, 1977.
- [20] Peter H. Feiler and Watts S. Humphrey. Software process development and enactment: Concepts and definitions. Technical Report CMU/SEI-92-TR-04, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., September 1992.
- [21] Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani, editors. *Perspectives on Free and Open Source Software*, Cambridge, MA, July 2005. The MIT Press Ltd.
- [22] Robert G. Fichman. Information technology diffusion: A review of empirical research. In *ICIS '92: Proceedings of the thirteenth international conference on Information systems*, pages 195–206, Minneapolis, MN, USA, 1992. University of Minnesota.
- [23] Robert G. Fichman. *The assimilation and diffusion of software process innovations*. PhD thesis, Massachusetts Institute of Technology, Sloan School of Management, 1995. Chair: Chris F. Kemerer.
- [24] Robert G. Fichman and Chris F. Kemerer. Toward a theory of the adoption and diffusion of software process innovations. In *Proceedings of the IFIP TC8 Working Conference on Diffusion, Transfer and Implementation of Information Technology*, pages 23–30, New York, NY, USA, 1994. Elsevier Science Inc.
- [25] Robert G. Fichman and Chris F. Kemerer. The Illusory Diffusion of Innovation: An Examination of Assimilation Gaps. *Information Systems Research*, 10(3):255–275, September 1999.
- [26] Roy T. Fielding. Shared leadership in the Apache project. *Commun. ACM*, 42(4):42–43, 1999.
- [27] Martin Fowler. Continuous Integration. Online, May 2006. <http://www.martinfowler.com/articles/continuousIntegration.html>, visited 2006-11-16.
- [28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [29] Rishab Aiyer Ghosh. Understanding Free Software developers: Findings from the FLOSS study. In Feller et al. [21], pages 23–46.
- [30] Rishab Aiyer Ghosh, Ruediger Glott, Bernhard Krieger, and Gregorio Robles. Free/Libre and Open Source Software: Survey and study – FLOSS – Part 4: Survey of developers. Final Report, International Institute of Infonomics University of Maastricht, The Netherlands; Berlecon Research GmbH Berlin, Germany, June 2002.
- [31] T. J. Halloran and William L. Scherlis. High quality and open source software practices. In Joseph Feller, Brian Fitzgerald, Frank Hecker, Scott Hissam, Karim Lakhani, and André van der Hoek, editors, *Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering*, pages 26 – 28. ACM, 2002.
- [32] Alexander Hars and Shaosong Ou. Working for free? – motivations of participating in Open Source projects. In *The 34th Hawaii International Conference on System Sciences*, 2001.
- [33] Gunnar Hedlund. The hypermodern MNC - a heterarchy? *Human Resource Management*, 25(1):9–35, 1986.
- [34] Lynn A. Isabella. Evolving interpretations as a change unfolds: How managers construe key organizational events. *The Academy of Management Journal*, 33(1):7–41, March 1990.
- [35] Jon Kleinberg. The convergence of social and technological networks. *Commun. ACM*, 51(11):66–72, 2008.
- [36] Gueorgi Kossinets and Duncan J. Watts. Empirical Analysis of an Evolving Social Network. *Science*, 311(5757):88–90, January 2006.
- [37] Karim R. Lakhani and Robert G. Wolf. Why hackers do what they do: Understanding motivation and effort in Free/Open Source Software projects. In Feller et al. [21], pages 3–22.

- [38] Ann Langley. Strategies for theorizing from process data. *Academy of Management Review*, 24(4):691–710, 1999.
- [39] Jean Lave and Etienne Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press, September 1991.
- [40] John Law. Notes on the theory of the actor-network: Ordering, strategy and heterogeneity. *Systems Practice*, 5(4):379–393, 1992.
- [41] Dorothy Leonard-Barton. Implementation Characteristics of Organizational Innovations: Limits and Opportunities for Management Strategies. *Communication Research*, 15(5):603–631, 1988.
- [42] Josh Lerner and Jean Tirole. The scope of Open Source licensing. *Journal of Law, Economics, and Organization*, 21(1):20–56, 2005.
- [43] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, New York, July 2000.
- [44] Niklas Luhmann. *Soziale Systeme. Grundriß einer allgemeinen Theorie*. Suhrkamp, Frankfurt am Main, January 1984.
- [45] Mary L. Manns and Linda Rising. *Fearless Change: Patterns for Introducing New Ideas*. Addison-Wesley, September 2004.
- [46] Patricia Yancey Martin and Barry A. Turner. Grounded Theory and Organizational Research. *Journal of Applied Behavioral Science*, 22(2):141–157, 1986.
- [47] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [48] Christopher Oezbek. Innovationsführung in Open Source Projekten. Forschungswerkstatt Pfadkolleg, December 2008.
- [49] Christopher Oezbek. Introduction of Innovation M1. Technical report, Freie Universität Berlin, April 2008.
- [50] Christopher Oezbek, Robert Schuster, and Lutz Prechelt. Information management as an explicit role in OSS projects: A case study. Technical Report TR-B-08-05, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, April 2008.
- [51] Timothy G. Olson, Watts S. Humphrey, and Dave Kitson. Conducting SEI-assisted software process assessments. Technical Report CMU/SEI-89-TR-7, Software Engineering Institute, Pittsburgh, February 1989.
- [52] Luis Quintela García. Die Kontaktaufnahme mit Open Source Software-Projekten. Eine Fallstudie. Bachelor thesis, Freie Universität Berlin, 2006.
- [53] Everett M. Rogers. *Diffusion of Innovations*. Free Press, New York, 5th edition, August 2003.
- [54] Alexander Roßner. Empirisch-qualitative Exploration verschiedener Kontaktstrategien am Beispiel der Einführung von Informationsmanagement in OSS-Projekten. Bachelor thesis, Freie Universität Berlin, May 2007.
- [55] Warren Sack, Françoise Détienne, Nicolas Ducheneaut, Jean-Marie Burkhardt, Dilan Mahendran, and Flore Barcellini. A methodological framework for socio-cognitive analyses of collaborative design of Open Source Software. *Computer Supported Cooperative Work*, 15(2-3):229–250, 2006.
- [56] Stephan Salinger, Laura Plonka, and Lutz Prechelt. A coding scheme development methodology using Grounded Theory for qualitative analysis of Pair Programming. In *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group (PPIG '07)*, pages 144–157, Joensuu, Finland, July 2007. [www.ppig.org](http://www.ppig.org), a polished version appeared in: *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 4(1):9-25, May 2008.
- [57] Steve Sawyer. Software development teams. *Communications of the ACM*, 47(12):95–99, 2004.
- [58] Robert Schuster. Effizienzsteigerung freier Softwareprojekte durch Informationsmanagement. Studienarbeit, Freie Universität Berlin, September 2005.
- [59] Richard M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. GNU Press, October 2002. With an introduction by Lawrence Lessig.

- [60] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.
- [61] Anselm L. Strauss and Juliet M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 2nd edition, September 1998.
- [62] Roy Suddaby. From the editors: What Grounded Theory is not. *Academy of Management Journal*, 49(4):633–642, August 2006.
- [63] Taowei David Wang, Catherine Plaisant, Alexander J. Quinn, Roman Stanchak, Shawn Murphy, and Ben Shneiderman. Aligning temporal data by sentinel events: Discovering patterns in electronic health records. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 457–466, New York, NY, USA, 2008. ACM.
- [64] Etienne Wenger. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge University Press, December 1999.
- [65] W. Kent Wilson. *The Essentials of Logic*. Research and Education Association, Piscataway, New Jersey, 1997.
- [66] Arnold Windeler. *Unternehmensnetzwerke: Konstitution und Strukturation*. VS Verlag, 2001.