Design and Implementation of A Flexible Workflow Engine with XQuery

Author: Kajetan S. Abt

Mentor: Prof. Donald Kossmann

March 13, 2009
Abstract

Process management systems have become an important topic in computer science. Many software engineering companies sell specially tailored systems to all sorts of customers, because most businesses model what they do as processes or workflows. To give an example: Due diligence. Investors want to know if a fresh start-up company has potential and if they should invest money in it. For this reason, they ask the start-up to write a so called ‘business plan’ which is essentially a document that describes how the enterprise plans to make money at some point.

This involves asking experts on their opinion about the chances of success a certain product could have, researching the competition, figuring out the details on financing, advertisement, employees and more. If the institution that carries out the due diligence (such as [17]) does this on a regular basis, this process usually is structured, because it for example does not make much sense to ask the expert to review some document before the owner of said enterprise has written that document.

At first glance, such a process should be relatively easy to model on a database management system, commonly used through a GUI in a web browser. However, even if such systems are in place, these documents are often printed out and stored in old-fashioned folders on a shelf and every start-up easily fills a thick folder. Not only does this defeat the purpose of having a system, it also makes some interesting questions all but impossible to ask, such as ‘Which reviewer gave the highest/lowest average scores?’ since going through thousands of sheets of papers does not work. But a database could answer such a question easily. On one hand, companies want such systems and could draw great benefits from them, on the other, they do not use them. Why is that?

Of course, there is a reason for this contradiction. Most of the problems with current systems can be attributed to very few, but very systematic flaws in the underlying design. This work tries to find those flaws and solve them in an elegant way, using cutting edge technologies.
## Contents

1 Introduction 3  
  1.1 Background and motivation 3  
  1.2 Problem statement 5  
  1.3 Overview 6  

2 State of the Art 7  

3 Architecture and Technologies 9  
  3.1 Technologies 9  
  3.2 Architecture 9  
    3.2.1 Flex 10  
    3.2.2 Database 13  
    3.2.3 Between the layers 14  
  3.3 Features 16  
    3.3.1 Roles 16  
    3.3.2 Meta data as data 16  
    3.3.3 Inheritance 17  

4 Use Cases 19  
  4.1 Creating and configuring the system 19  
  4.2 Working with the system 24  
  4.3 Use cases on flexibility 28  

5 Evaluation 34  

6 Conclusion 35  
  6.1 Overview 35  
  6.2 Local flexibility 35  
  6.3 Global flexibility 35  
  6.4 Drawbacks and limitations 36  

7 References 38
1 Introduction

1.1 Background and motivation

This work will mostly focus on very few recurring examples for processes. The most important one of these is due diligence. But what is due diligence? Very abstractly spoken, it is a process which in the end results in a business plan and an answer to the question ‘Should we invest money into this start-up?’. But to give a more concrete meaning to this description, this is the due diligence process (see figure 1) that eValueScience [18] uses. For brevity’s sake, not every step is explained.

But first, the notion ‘business plan’ must be clarified. A business plan is a formal statement of a set of business goals, the reasons why they are believed attainable, and the plan for reaching those goals. It may also contain background information about the organisation attempting to reach those goals.

Assignment of Expert: A person is chosen to be the expert for an instance of this process. That person needs to be neutral to the start-up in question and must have knowledge about the field it tries to have success in. Ideally, the expert is also well-informed on the market and economics.

Signing Agreements: The expert (and later on, peers too) have to sign non disclosure agreements (NDA), as the information about the company in a due diligence process usually contains trade secrets. Of course, this NDA has to be written first.

Handing over Documents: These are the first draft of the business plan, which is given to the expert for review.

First Opinion of Expert, Decision: The expert studies these documents and decides whether the ideas presented stand a chance or if the proposed business idea could not imaginably work. In case he or she approves, the process is continued, else, it is terminated.

Full Due Diligence: At this point, more people are required to get more input, and the documents have to be refined and clarified, as they rarely are perfect in their first draft.

Developing List of Peers: Candidates are chosen for the peer group. This group of people reviews the documents and supervises the process. Ideally, they are from different fields, such as a computer scientist, an economist and a doctor for a business that wants to sell software to hospitals.

Discussing Peer List: Meetings are held to find the best list of peers. These people are then required to sign a CDA (confidential disclosure agreement) to get access to the confidential data.
Figure 1: SDD process used by eValueScience [18]
1.2 Problem statement

Review of DDC Documents: The confidential due diligence documents are reviewed by the peers who then have a normal meeting or video conference.

Report: The peer group writes a report, which first goes through a draft stadium, is refined and then results in a final report.

Approval of Final Report: If eValueScience is not content with the report, it will have to be improved. But if eValueScience deems it finished, they come to a decision whether investing money is sensible.

Of course, this example is rather specific, but most due diligence processes (even if they have a different focus, be this science or economics) have significant similarities, such as the most typical steps involved. We found that these are meetings, documentation, approving of something and employing people into roles.

1.2 Problem statement

As mentioned, there already are solutions (e.g. [15], [9]) for process modelling, in fact, there is an abundance of different software packages for all kinds of different businesses. But they all suffer from the same two problems:

- They are written from scratch on demand. Every time anyone wants such an application, they have to hire someone to write a big piece of software. If you think about a company producing yoghurt and compare it to a business offering a service on the internet, some information is very important to one and of no interest to the other, such as where the company buys its raw material (milk, or in the second case: nothing). This directly leads to a very different process which results in reinventing the wheel every time.

- They are inflexible. Even though it is assumed that those processes do not change much, in reality, they rarely do not evolve. What works as a due diligence process for the first start-up might not be feasible for the second, or the fifth or twentieth. As soon as these differences become major, software usually has to be rewritten and recompiled, database tables redesigned and people retrained, all of which is expensive, time-consuming and prone to introduce bugs. And if in the end the old data is not compatible with the new data anymore, more damage has been caused than benefits gained.

But from a more distant point of view, these two problems are both grounded on the same underlying limitation. If the system was flexible enough to be ported between customers, adapting it to another workflow might be easy, and if it supported any kind of workflow, making it also support a completely different process should be simplified. Therefore, the primary goal of the proposed design is flexibility. There must be strong support for different classes of workflows, small differences in how data is treated in one case or the other, and if changes occur late into the process they must be handled gracefully.
We will give a clearer distinction of the different kinds of flexibility that are required.

**Global flexibility:** This is the kind of flexibility that allows the software to be deployed in multiple companies. Having ‘enough’ of it means that one can model a process for producing cars or a lecture plan for a school with the same system. Even though that example might be exaggerated, this also is useful on a smaller scale, such as the distinction between two kinds of documents (a letter is not a contract, but both are a document). These are differences that define the building blocks of a process and generally answer to the question ‘What?’.

**Local flexibility:** The second kind of flexibility allows us to change parameters of the process during runtime, for example switching one person for another, or swapping the order of two steps. This answers to the question ‘How?’.

Note that this separation is not clear cut. Some examples can be fit into either category. The chapter on use cases gives a list on what kind of changes can likely be expected, both in the small and in the large, and how those are handled.

### 1.3 Overview

The paper is structured as follows: Chapter one gives a brief overview over the situation, including an explanation of due diligence and the major difficulties that have to be overcome. In chapter two, an outline of the current systems, their advantages and limitations is given. Chapter three lists the technologies used, the architecture employed, how the data is modelled and why these methods overcome the difficulties mentioned in chapter one and two. Chapter four is a list of detailed use cases with a focus on how the requirements are met, how the software works and why a traditional system would be inadequate. Chapter five is a summary of the results of the discussion on use cases. Chapter six gives a conclusive overview over what has been done, if and how well it works and what could be improved.
2 State of the Art

The commonly used technologies are a relational database system processing SQL and a GUI on top. Very often, Java or PHP are chosen, but many other modern object-oriented languages can be employed as well. For briefness, Java is used to give examples. There will be a brief discussion of how such a system would be constructed and maintained and what its strong and weak points are.

Starting from the example given in the chapter on due diligence, it would make sense to model those four different steps each as a table in a star-schema and a central fact table to link everything together, as seen in the corresponding ER-diagram in figure 2.

This data model will distribute some information, such as when steps are executed, what kind of prerequisites they have and what kind of forms the users are working with in different places. Some of it is in the database (the date of a meeting), some of it is in Java (the form) and some things could be done either way (such as prerequisites). Having forms there is only sensible, checking whether or not some piece of data was already committed is easily written with a small query called from Java. In fact, all SQL code is embedded into the Java client anyway, so trying to find a separation seems superfluous.

But there is a problem coming up: What happens if a single project needs a slightly different schema than the others? For example, documentation for a very security-critical project might have to be (digitally) signed by someone else except the original writer. The only way to do this would be to add a new column to the table containing the documentation. But that means that now all documentation steps in all projects have
such a column, most without requiring it. The front-end application has to know when to ask for these columns, and when not to, effectively requiring the Java code to include information about differences in projects in its own source code.

Or assume two steps are done in a different order for a new project. Again, the Java code must be altered and recompiled. The more refined the process becomes and the more changes happen, the bigger and more intransparent the code base gets. At some point, the idea of splitting up the fact table will start to look good, as the tables have accumulated too many columns which are only meaningful for a minority of the projects.

Splitting up the fact table by project, or rather, by class, seems reasonable. Similar projects are grouped together in the same fact table, whereas heavily distinct projects are stored in different tables. Technically, even a separate table for each project is feasible. Of course, multiple tables make cross-project queries neigh impossible, produce a gigantic amount of duplicated code and means that every project would need its private set of queries. It would be the ‘folders on a book shelf’-approach all over again, and is exactly what happens in reality. This is only a short-term solution, as the Java client becomes very hard to maintain and people start to switch back to pen and paper or have to buy another system.

The flaw is hidden underneath all the code bloat: Assuming a static model in the beginning is a mistake. Workflows change too often and too rapidly for this to make sense. But even realising that does not help much. Relational databases require a relational model, or else they do not work. Those models are the reason for the success of RDBMS, and they are also the reason why a problem which cannot be modelled well is best tackled without one.
3 Architecture and Technologies

3.1 Technologies

The proposed system uses a similar two-tiered architecture like established systems with a web client layer on top and a database underneath. But instead of the more traditional relational database, a cutting edge XQuery [11] engine is employed, called Sausalito [5]. The prototype was first developed at ETH [19] and is now being improved by a spin-off company called 28msec [10]. At the time of writing, they have published a first beta release (0.9.1), which is the required version for the system to run on. A short feature list of the sausalito package follows:

- It supports Windows, OS X and Linux.
- It can be deployed on Amazon Web Services [16], or it can be run on any web server.
- The XQuery Update Facility [13] standard is fully supported which allows to edit collections easily and efficiently.
- Zorba-Collections [12] are implemented.
- It runs as a RESTful [8] service, allowing nearly any other modern technology to seamlessly integrate with it.

The GUI layer is written in Adobe Flex [6], a Flash-derivative which targets a different niche in the market than Flash usually does. It is slimmed slightly, sacrificing some medial functionality, but instead has open-sourced libraries and an easy-to-use GUI editor as its main selling points. The same languages as for Flash are used, which are MXML (XML-dialect to declare a GUI) and ActionScript (an object-oriented language similar to Java). In Flex, it is incredibly simple to produce GUIs with buttons, tables and similar items. Flex also offers XML-transformation-functionality similar to XQuery, which makes processing XML very convenient. The two technologies communicate via asynchronous REST-calls, which again is one of the strong points for both.

3.2 Architecture

Even though the ‘same’ two-tiered architecture is used, there is a striking difference: Whereas the traditional systems have all their business logic written in Java, it has been taken great effort to separate business logic and the user-interface. What this means in detail will follow. The explanations given here are general and abstract, detailed information on how single methods are implemented is given in the use cases section.
3.2 Architecture

3.2.1 Flex

This is the upper tier of the software, which can be seen in figure [3]. Note that this does not show all classes, but rather the more interesting ones. Function calls and attributes are also very abbreviated.

![UML diagram of the flex classes](image)

**Figure 3: UML diagram of the flex classes**

**Step**: The step-class is the heart of the application. It contains most of the code that becomes somehow visible to the end-user. If a user wants to complete a step in the workflow, he or she is presented with a panel to do that. This panel is generated by a step instance and then populated with data by the subclasses. All behaviour that all the different types of steps have in common is offered by this class. The StepDB is a small database-interface. It basically consists of a list of interfaces for the REST-calls for the step object to use. A step object binds an event-listener to the StepDB-instance to catch results, since that process is asynchronous. This
3.2 Architecture

separation makes changes to data format, server-addresses or ports easily possible. When results are caught, the appropriate receive-functions in the subclasses are called with dynamic binding.

Employment, Documentation, Meeting, Approval: These four classes are the most basic steps in a (due diligence) workflow process. Of course, these are by no means a complete set of all kinds of steps that can ever come up. Rather, they demonstrate the principles that future work should follow: They are all subclasses of Step to prevent code duplication, and they implement the AbstractStep interface to make sure they offer all required functionality.

Review: This class does not really belong to the minimum requirements. It is a demonstration of how inheritance can be used, read more on this in section 3.3.3 and in the use case on review.

AbstractStep: This interface specifies how steps integrate with the system. Any class which inherits Step and implements AbstractStep is sufficiently equipped to be used as a new kind of step. The interface specifies six functions, four of which are concerned with receiving data from the database and two with committing data into it. Any time information arrives (asynchronous) from the database, the matching receive-function is called with dynamic binding, and analogous for the commit-functions (though these are commonly called by mouse clicks, if not necessarily so). All receive-functions have to first call their super-class’ function to allow them to create a panel and fill it with generic step data (such as id or mode) however commit-functions have to do the opposite: calling the super after finishing, to make the super class pass the new values on to the database.

The four different receiving functions show the typical four states a step can be in. One can just look at it (About), one can try to complete it (PreData), one may try to edit its meta data (Workflow) and lastly, one can just look at the committed data (Data).

- receivePreData
  This receives and displays the information required by a user to do work. For example, if a person had to be chosen to be expert, this would be the list of candidates, or if a document has to be written, this is a required content summary. It is supposed to make the work as easy as possible, which means it not just displays some plain XML, but rather a sophisticated form. In the first example that would mean to have a list of check-boxes for all people.

- receiveData
  Committed data is always of interest. This function will populate a panel
3.2 Architecture

with it.

• receiveWorkflow
  Since we expect details to change more often than not, there is a GUI for changing such things, such as the room for a meeting, or the order of steps.

• receiveAbout
  Essentially the same information as receiveWorkflow, but without the capability of editing it. Used for steps that are not ready to be worked on yet, or for users who do not have the privileges to do so.

• commitWorkflow
  This function is called with a mouse-click and writes the changes to a step (such as the time people want to meet at) into an XML which is then passed to the superclass and committed to the database.

• commitData
  When a user has filled out a form, this function writes the data back into storage.

StepFactory (not in the picture): It implements the factory-pattern [7]. The rest of the code should not need to know how to distinguish between the different subclasses of step and there are multiple ways of instantiating them (creating a new step or reading an existing step from the database), having a factory class is a very convenient solution. Adding new types of steps requires small changes to the factory, but no searching through other sources.

Main: The main application, a set of code that is often called ‘glue code’. The Flex application that can be started in a web browser. A lot of it is designed to produce a useful GUI and of course, a multitude of REST-calls. It is roughly state-based with states such as log in screen or data display.

SVGViewer: A library [2] which displays scalable vector graphic files [4] and can load them at runtime (which Flex cannot). This is very practical as is allows the software to display a picture of the graph which can be clicked to open the different panels. SVG-files support hyper-links, but this functionality has been replaced and the event is now caught in Flex itself.

Utility classes (not in the picture): These are small classes which contain support for often needed shorthands, such as creating a combination of label + text-box contained in a container, error reporting functions or search functionality for sets of steps, since flex does not support generics.
3.2 Database

The database layer contains at least three collections. Two of these are straight-forward: Users and projects. Following a brief display of how these collections are structured, nothing of which is particularly unusual.

```
<people>
  <person>
    <name/>  # unique id for login and identification
    <password/>  # password for the system
    <...>  # more data, such as address, AHV, age, etc.
  <person>
<people>
```

```
<projects>
  <project name="">
    <general>  # node for general project information
      <company/>  # data about the company, or a FID
    </general>

    <role kind="peer">  # the role peer
      <user/>  # a single user link (from people DB)
    </role>

    <completion stepid="" finished="">  # completed steps
      <data>
        <user/>  # committed data for a step
        <role/>  # user that committed dat
      </data>
    </completion>
  <project>
</projects>
```

One might wonder why sometimes attributes are used and sometimes child nodes. The reason is simple: There can only be a single attribute of a certain name per node. Since a certain meeting can only ever have a single ID, it makes sense to do it this way. A few other values right now are technically unique too, but one can reasonably expect them to change in the future, see the use case on multiple roles.

One can imagine nearly any database system having similar collections (or tables) like the two just introduced. Possibly the reader was bewildered by the utter simplicity of the projects collection or might have assumed it was harshly abbreviated. This it not
the case. Of course, there is a reason for this: Usually, such a collection contains a lot of information about the workflow that it models, for example what kind of data is expected at what time (e.g. dates for meetings) or what kind of type the data has. In a relational system, one could tell quite a lot from reading column names and types. In our system, this information is separated from the data. The reasons will be explained in the features section in chapter 3.3, and more specifically in section 3.3.2. The third collection, which will be explained in detail in said chapter, looks like this:

```
<workflow project="">
  <step id="">
    <work>
      <type/>
      <role/>
      <.../>
    </work>

    <prerequisites>
      <id/>
    </prerequisites>

    <mode/>
  </step>
</workflow>
```

### 3.2.3 Between the layers

Lastly, there is the question of how the two layers interact. Even with the already mentioned flexibility of meta data, businesses are vastly different and have different needs which will definitely require alterations. How this has been accomplished in the GUI-tier was already explained. If the reader takes a look at the schematic (figure 3.2.3) on how the database code is arranged, it should be impossible to miss the close mirroring of the class layout in Flex.

**main.xq:** This part of the system handles requests that span more than a single step, such as finding work in progress (WIP). If necessary, it will pass on the request down to *step.xq*.

**svg.xq:** All code in this file concerns itself with handling the scalable vector graphics [4] file to display a graphical representation of the process. It can translate between the different formats used in the GraphViz [1] library and the system itself, and it can find nodes in the graph and process them, to change their colour or edit their behaviour when clicked on.

**AbstractStepXQ:** It was stated that adding a new type of step only requires the interface
to be implemented in Flex. The same holds true for the database system. Of course, this an implicit interface, since XQuery is not object-oriented.

**step.xq:** It is the sibling of the **step** class in Flex. It offers the database interface that **step** needs to work, such as data retrieval or commit-functionality. Analogous to Flex, calls are passed on to specialised handlers if necessary.

If one wants to add a new kind of step, it is necessary to write specific commit-functionality, and possibly something for retrieval of preData. Since this is separated nicely from all other code, it is really only necessary to add an additional .xq-file with that code to the system, and add the calls to the switch-statement in step.xq. Changes to any existing features, such as how steps are written or read is also contained to a single place. And lastly, if a new feature spans across a far range of data, such as aggregates, this will most certainly be called by the main class in Flex and therefore is always handled in main.xq.

---

**Figure 4:** Diagram of the database code structure
3.3 Features

One of the main difficulties in designing the system was the broad problem domain. Instead of a specific requirement one could describe the issue as a ‘general notion of more flexibility’. What is worse, ‘flexibility’ in itself is already incredibly unclear. If one knew which change to predict, one could prepare for it and essentially incorporate the solution. Since prophecies do not belong into the world of science, all we can do is to prepare for an arbitrary choice of possibilities. Of course, one can still try to choose wisely. In general, this means that one prepares most for outcomes that are either very likely, very important or (ideally) both. This chapter shall give an overview over measures taken to catch problems that fall in these categories.

3.3.1 Roles

The attentive reader will have noticed that there are roles mentioned in the XML for the collections. This is an abstract way of grouping people. Since steps in a due diligence process can (and often must) be completed by different people, such as the owner of the start-up writing a financial document, but the person designated expert then reviewing and rating it, this is an important function. However, being an expert for project A does not qualify for project B. For that reason, roles are per-project (and also stored there), instead of per-user. This relation is always considered to be 1:N:N. Any project can have N roles associated with it, which each can have N users, with any amount of overlap (any users can be in more than one role, which happens often in practice), but a single role belongs to a single project only.

There are clear benefits to this approach. First, adding additional users to a role does not break the system. In relational databases, having a single-valued field become multi-valued (see the use case on this) often leads to problems. Assuming from the beginning that there are no single-valued fields solves this issue and can be done without any complication in XML, since set operations do not care whether one or many entries are in a node. Relational systems on the other hand would require an additional table and additional joins to model multiplicity.

3.3.2 Meta data as data

Relational databases work on top of a model, such as an Entity Relationship. This model is then mapped to tables, which can be done in different ways. Depending on other requirements, such as certain search-features in data-warehouses or easier distribution on multiple machines, this results in a lower or a higher normal form, respectively. But most important, the model is always designed and fixed in advance. It is a sometimes overlooked fact that relational databases do not just store data (or information), but also meta data (and therefore meta information). The model mentioned is a representation
3.3 Features

of that meta data. And this brings us to the crux of the problem. Changing this model
during use is all but impossible. Even if it might be possible to apply small changes, such
as new formatting or an extra column, a full rewrite of the model cannot be accomplished
without starting from scratch and manually porting all old data into the new model. In
practice, the easiest solution is often to declare the old system legacy and read-only, set
up a new one and start over.

Of course, this is highly impractical. As mentioned earlier, being able to change the
data layout on the fly is of tremendous practical use. This of course means changing the
meta data. And if one wants to be able to change meta data, one has to treat it just
like any other data. With this in mind, the workflow collection is designed. It does not
store the data, but instead, it stores the meta data.

When data is stored, retrieved or transformed, the meta data is used to understand what
is happening. It gives meaning to data, or rather, it makes data become information.
At the top of the workflows collection we have a list of workflow nodes. Each of these
nodes contain the meta data for an instance of a process. A process is broken down
into steps which have to be completed in some (partial) order, and this information is
exactly what can be found in the multiple <step> nodes. For example, the <work> part
explains what kind of work (see the use cases on basic steps) is supposed to happen,
whereas <prerequisites> lists all steps that have to be completed first, establishing
this order (which can be displayed as a directed graph).

With this technology in place, new options open up. Suddenly, altering part of a workflow
becomes far from difficult. In fact, the current GUI already offers a simple point and
click interface for doing so. One can easily add steps, change their order or any of
their parameters. This covers local flexibility well, but will not do much for global
flexibility.

3.3.3 Inheritance

Complementary to the former feature solving most issues with local flexibility, a tech-
nique to deal with global flexibility is required. As a reminder, global flexibility deals
with the problem of deploying the same system in two different environments.

Inheritance is used in two ways in the system. Firstly, by moving as much functionality as
possible into the step class, adding new types of steps is simplified a lot. The developer
altering the system does not need to write something from scratch, nor does he have to
learn the old code. By adhering to the interface abstractStep given, the new type will
fit perfectly. That means that rewriting parts of the system to a completely new kind of
process is mostly about rewriting the GUI classes, and not the internal workings. This
solves a lot of problems concerning global flexibility.

But even if that would technically be sufficient for most purposes, there is a second
possibility hidden in the idea of inheritance. Instead of only inheriting step, a class
could also inherit a subclass of `step`, such as `documentation`. This allows one to refine parts of the system without altering its core. A subclass of `documentation`, say `review` still responds to any query launched for documents and will always behave at least like it, but offer additional functionality on top of that. To translate that to **global flexibility**: Instead of recreating the model when the process is refined it is now possible to add the new classes on top without breaking the old ones.

Now if a query asks for all steps of type `documentation`, most systems would return all instances of that class and also all instances of subclasses, but it would treat them all as the superclass, `documentation`. This means that any additional functionality or information from subclasses is hidden from the user, or in the worst case, discarded or damaged. But the written system behaves slightly different: After retrieving that data, Flex-classes are created by `StepFactory`, which will always return the most refined type. In the former case, it would return an instance of `review` with its more complex panels than the plainer `documentation`. But since both adhere to the same interface and use dynamic binding, the system can treat them exactly the same later on.

This begs the question on how the database handles sub-typing. The implementation is actually rather simple: The database does not distinguish between sub- and super-types (most of the time), instead it just treats such objects as a set of types. Since XQuery is about set-arithmetic anyway, this is handled very gracefully. The only limitation to this approach is the lack of override functionality. A subclass cannot override how a superclass retrieves or commits data, which (ironically) is an advantage, since it prevents data loss. In case this feature is requested, sub-classing was the wrong choice to begin with.

See the use cases on `review` and new forms for practical demonstrations of this.
4 Use Cases

This chapter is meant to list the most interesting and most important use cases for a workflow management system. A useful system has to be able to successfully support all the mentioned use cases well. If nothing is mentioned on the relational system, it is assumed to be able to fulfil the requirements similarly well without major differences. In case the relational systems performs significantly better or worse, this is described explicitly. Many examples of these use cases have been briefly mentioned in earlier chapters.

4.1 Creating and configuring the system

These uses cases are about creation of workflows and what building blocks are indispensable for any such system. Some do not really show anything unusual or new, but they are necessary for the sake of completeness.

**New workflow:** At the beginning of a new project, the responsible person has to set up the first workflow. He or she has such workflow written down on paper (just like the example in figure 1) and enters this information into the system.

The workflow has to be written down in the schema of the specified XML for workflows. The software already demonstrates that is is possible to do this with an easy GUI, which offers the most basic functionality, such as creating new steps and editing them and should be easy to improve on. While doing this, the system will constantly update the graphical representation of the process. Ellipses (or boxes in the example) symbolise steps, arrows dependencies and labels the details of a step, similar to the example given in the first chapter.

**Multiple classes of workflows:** A company has two workflow classes: Full due diligence and short overview. These are very different but they are built from the same set of steps. The individual projects are then again slightly differing from each other, but are still very homogenous compared to the other class.

The system does not distinguish between classes and instances. Every workflow is an instance. It is of course possible to design a couple workflows for use as templates and never use them for productive work, and just create (modified) copies of them every time a new project launches, but typical ‘classes’ do not exist. Having classes of workflows is a band-aid for relational systems (because they cannot work with an arbitrary amount of differing workflows in the first place) which we do not need.

A traditional system can be capable of supporting different workflows, and this is what is usually done. Differing processes are grouped together, for better (or more commonly) worse and they each get their own tables, and their own code in Java. It should be obvious that this leads to duplicated code and is one of the primary
problems the current systems have with local flexibility. See state of the art at 2. In conclusion it can be said that it can work, but not that it works well.

**Meeting:** *This is one of the four basic blocks. The control group meets at a designated place and time and discusses something of importance. Then, one of the members uploads a report.*

This means that the workflowXML has to contain a step of type meeting, like this:

```xml
<step id="6">
  <work>
    <type>meeting</type>
    <place>Room 15</place>
    <time>15:30</time>
    <role>Control Group</role>
    <purpose>Discuss positions</purpose>
  </work>
  <mode>any</mode>
</step>
```

On completion (see further down for details on that) this results in a data block like so:

```xml
<completed stepid="6" finished="true">
  <work>
    <type>meeting</type>
    <report>We came to an agreement</report>
    <user>David Davidson</user>
    <role>Control Group</role>
  </work>
</completed>
```

This XML is then stored in the database in the projects collection for further use. Most nodes are self-explanatory, those that are not will be discussed elsewhere.

**Approval:** *This is one of the four basic blocks. In a meeting, the people decide whether to continue with a project or stopping it.*

An Approval statement has to be added. After the meeting happened, the users will be prompted by a form which allows them to vote YES or NO on behalf of the step in `<otherid/>`. This means that YES results in the work done in that step being accepted, and NO signifies rejection.
Note the <prerequisite> node which specifies the meeting of the former example. The system will understand that this step cannot be completed until the meeting has taken place. Policy tells the users (and the system) how the votes will be counted and tallied (majority, unanimity, and so on). <contingency> shows what the system does with this. It might for example terminate the project or just redo one step. Since this is highly dependent upon the project and company itself, it has to be implemented specifically. At the time of writing, this feature is not implemented, since it usually results in versioning, which is a feature that sausalito plans to support soon.

Upon completion, it will also leave a new data node behind to be able to see what has happened:

<completed stepid="6" finished="false">
  <data>
    <type>approval</type>
    <user>Hans Muster</user>
    <role>Control Group</role>
    <decision>yes</decision>
  </data>
</completed>

Documentation: This is one of the four basic blocks. Assume a reviewing system for scientific papers. Any paper coming in will have to have a review written on it by three different reviewers. They write such a text and upload it into the system. This can only be done after the paper is submitted in step 1 and the reviewer has been chosen in step 2.

Again, a step is needed for such a documentation and it looks like this:
4.1 Creating and configuring the system

When completing this step, a form is displayed to the user in which he or she can enter a document. Currently, only plain text is supported, because sausalito has a bug in its API with document upload. On the other hand, the system will not distinguish between different file types and just store those as binary objects.

Employment: This is one of the four basic blocks. Often, it is necessary to add additional people to a project. This use case covers the case where this is planned, for example, first a single expert reviews a business plan draft and if he accepts it as decent enough, two additional peers get asked to take a look at it. The expert may choose these people from anyone associated with the project.

Again, a step in the workflow which lists exactly what was just stated, but more mathematically of course.
The expert is then prompted with a form and all the names of people in the associate group to choose two of these. This is then put into effect by the database which not only stores a `<data>` node similar to the ones we have seen before (only for the sake of completeness), but also associates those users with the group they have been designated to, allowing them to use the software, see the project and complete any steps that are meant to be done by anyone in the role ‘peer’.

Any versus All: There are two variants for each step: Either all members of a group have to complete a step (for example, every reviewer has to upload his or her opinion on the subject in form of a document), or only one of them has to do so; five people meet but only one of them has to write the report of that meeting. The system must be able to distinguish between those two cases.

Every step has a `<mode>` node which tells the system whether the step is `_any` (only one user) or `_all` (every user). As soon as any user has completed a step marked with `_any`, the step is considered finished. A step marked with `_all` on the other hand must be completed by all users (designated in `<role>`). Note that as long as the step is not finished by anyone, it can be changed from one value to the other without implications. This constraint is checked every time a user commits data.

Prerequisites: As briefly mentioned in the use case on documentation, sometimes it is necessary to wait for another step to complete before the next step can be started. This is modelled with the `<prerequisites>` node. It contains a set of unique identifiers of other steps in the same project for steps that have to be completed first. Completion is checked on commit and then written down in the `<completed>` node in the `@finished` attribute.
4.2 Working with the system

These use cases explain how a user will interact with the system most of the time, how work is done and what kind of features can be expected of such a system.

**Executing a workflow:** After specifying a workflow, the software has to provide the means to work with it. It needs some graphical user interface which can display both completed and uncompleted steps and visibly mark those steps which are next in line to be completed. The system must also pass enough information about the steps to the user so that he or she can handle them.

There are a lot of things necessary to fulfil this very broad requirement, some of which are broken down into smaller chunks below. First, the user is presented with a screen to log in to the system with a user name (which is unique and extensively used as a key in many queries) and password. The XQuery-engine will then use the user name to query for all projects in which that user has an active role (or more than one). It then checks in all projects if there is an open step for that user and returns all of that information. This is shown as a list of project-role pairs and for convenience, if work is required, this will be displayed too.

The user then clicks on one of the project-role combinations and gets taken to the next screen. This screen displays a graphical representation of a workflow with boxes for steps and arrows for prerequisites (see detailed explanation). In addition, a table with all steps that need work is shown to give more complete information to the user. Both the table and the graph can be clicked to either finish a step or look at data of a completed or future step.

The user can click on any step to see the about panel if he or she is interested in the parameters, such as a time for a meeting or a title for a document that needs to be written.

**Finding work in progress:** After logging in, the user wants to see which step must be completed next, and what kind of work he or she is supposed to do.

The system can be asked to return so-called work in progress (WIP) nodes for any user. This query goes through a rather elaborate list of checks: It first fetches all projects where the user has any role in and keeps the role. It then finds all steps in all the workflows matching the project and joins these steps over role, resulting in all steps that a certain user could ever complete. Then, it removes all the completed steps from this list. In a last step, all steps which have the prerequisites unfinished are also cut. The result is a list of steps which the user can and should complete right now.

**Executing a single step:** As a user selects a step he or she wants to work at, the user is presented with the data that is required to do so, for example if the user is supposed to select five peers for an employment step from the group of all associates, then all these should be displayed to him.
4.2 Working with the system

Generally speaking, this functionality is dependent on the type of step that is selected. Therefore, the query to get this information is written down in the XQuery file that belongs to this type, in this case, employment.xq. The system does a type check and runs a query on employment:preData().

The query itself is straight-forward. The system can find all this required information by joining the step requirements over the already existing project data, for example a join of <from="associate"/> (in an employment-step) over the <role kind="associate"> <user> Hans Muster </user> </role> leads directly to the list of associates consisting of (only) Hans Muster. This list is returned to the Flex GUI and displayed as a list of check-boxes. The user then marks his or her choices and clicks the commit-button. The behaviour for the other three basic steps is analogous.

On commit, this happens: First, the database uses code in step.xq to write general information down, such as id, user or role. Then, the commit-function in the appropriate file for the committed type is called which might do things such as modifying the user database.

**Executing a workflow (more than once):** It is usually of interest to be able to do a workflow more than once (for two different projects, of course), either in succession or in parallel.

It is trivial to copy the workflow of the first project, give it a new name and store it. The second instance is completely independent of the first and changes to one will have no effect to the other. This can be done as often as wished for. Note that it is also possible to take an old workflow, copy and rename it but then change some details. The resulting workflow is then slightly different from the former one, but again, the old one is not connected to it anymore., since no reference what so ever exists between the two.

Compared to relational systems, this is a completely different behaviour. In a table, every line in a table has the identical headings and all related tables (where the foreign keys point) are also identical. That means they have the same meta data. Having different workflows leads to different columns used, depending on the row. In the most extreme case where the two workflows have nearly nothing in common, a relational system might use two different tables. This then leads to a big problem: Queries will not catch all workflows by default anymore, but one would have to write the same query twice for the two tables.

**Displaying the process; SVG:** Because of convenience, it is important that the users can work with the system without deep knowledge about the database system, XML or the technical aspects. Therefore, it is imperative to have a simple, graphical view of the workflow. It can be represented very well with a graph consisting of boxes and arrows. The boxes symbolise steps, while an arrow pointing from step A to step B means that step A has to be completed before step B can be started.
Further, these boxes should be click-able and context-sensitive. Clicking on a WIP-step should bring up the PreData-dialogue, whereas clicking on a finished step should show the committed data for this step, and clicking on a future step should only display the About. (See chapter 6 for more explanations)

This graph can be generated automatically with sausalito 0.9.1 and up. The layout of the graph corresponds to the steps and their prerequisite-nodes. This is transformed into an XML conforming to the GXL-standard [3] by the database. The GraphViz-library [1] can then use that GXL-document in a two-step process to return a scalable vector graphics file [4]. This SVG is also based on XML and can be easily post-processed on the XML-engine, adding colours (red for WIP, blue for finished, orange for partially finished and white for unfinished) and labels to the graph. The SVG file can be displayed in most current browsers without the need to install any plugins, which would make an upper layer of PHP or JavaScript possible. Flex cannot load .svgs during runtime, but the SVGViewer [2] project adds this functionality.

SVGs (usually) support hyper-links. In this project, that functionality was replaced with a simple event, that can be caught and understood by the Flex application, offering the required context-click functionality. Depending on what state the application is in and what kind of node is clicked, it will show either the About, PreData, Workflow or Data panels.

**Removing a member from a role:** *Peter is member of the Expert role in January. In February though, he retires and therefore loses access (and rights) to the system.*

He will be removed from the role Expert with an employment step which has an <action=remove/>, which works analogous to any other employment step. But now, if a query wants to evaluate a node from January, he cannot be found anymore in the Database. To solve this conflict, his data is not just deleted from the system, but sausalito will support to ‘go back in time’. It is possible to check what data looked like at a specific date, and not just in the now.

This is currently not implemented and only <action=add/> steps are meaningful, since sausalito does not support this functionality yet. But it is important to note that the big part of this feature is not developed in the application (the query will need an additional flag at that point, but nothing more), but in the underlying database. This is a direct result of the as-flexible-as-possible approach taken.

**Querying data of a completed step:** *Since people want to look at committed data too, it is imperative that this information can be easily displayed. A user can retrieve data of a single step or of all completed steps.*

There are multiple ways of finding committed data, the simplest of which is a ‘find all’ function to get all data ever written for a chosen project. This function collects all the <data> nodes that have been stored and returns these items. This list can be filtered against with an arbitrary filter function in the DBMS. Currently there
are few filters supported, such as for example a set of ids, but adding more is rather simple, as seen in the next few use cases.

**Querying data on a workflow:** Assume one is interested in all documents that have been written for a project. There should be a way to get to these easily.

The system can just return all data nodes in the `projects` collection which satisfy the constraint that they must be of `documentation` type. This is a simple addition to the function that returns data of completed steps, adding an additional

```sql
where step/@type = "documentation"
```

clause. Of course, querying for different types of steps or against some attributes can be done in the same fashion. Technically it would even be possible to pass a filter function from the GUI to the DB-tier, but this is discouraged, as it would violate the separation of business logic and GUI.

**Querying data over multiple workflows (simple):** It is very reasonable that queries might span multiple projects. For example, one wants to see all the documents written by a certain user.

This is as easy as it would be in any relational model too. The system just searches for all `<data>` nodes which satisfy the condition

```sql
where data/user = $name
```

possibly filtering against type as in the use case before. Also see the use case on sub-typing in the flexibility-chapter later on.

**Querying data over multiple workflows (advanced):** 'Which reviewer is the strictest?' Is a Query that can be of interest. Strict is defined as giving out the lowest scores in average over all projects.

To be able to answer such a query in the first place, it would be necessary to somehow define ‘score’ first, see the appropriate use case in the subchapter on flexibility. Assume we have a subtype of documentation called `review` which sports a `<score>` node. A query asking for all reviews will then be able to get all these scores and calculate an average value, or apply other math on it, such as median or deviation. Note that `documentation` nodes are not found by the query, because it is a super-type of `review`.

For a traditional system, this can be very difficult. Depending on what kind of changes have already been applied to the system, old data cannot be parsed well anymore. It might have an older format, or it might even be stored in different tables. Someone has to go through all these tables and figure out by reading up on legacy source code how this kind of data was stored at the time those tables were used.
4.3 Use cases on flexibility

Since some of the use cases change how data is treated, it is important to note that such a change should not require a lot of work for rewriting the UI or the DBMS, and of course, it must be possible in the first place without developing a new system from scratch. Nonetheless, writing code is not completely out of the question for these use cases. As we will see, the traditional approach cannot cope with the requirements without doing so considerably more often than the new system.

**Altering a step in the workflow after completing it:** A document title which was chosen when the workflow was set up is misleading and should be corrected. But the process has already finished and the document was committed with the wrong name.

This is partially possible. Some things are not evaluated anymore. After a step is finished, that status is stored and not checked again, even if circumstances change later. Other values cannot be altered without seriously endangering the integrity of the data (such as `step-id`), but changing human-readable content (such as descriptions or titles) is unproblematic, as this is never parsed or evaluated, only returned and displayed. It is also only stored once in the meta data collection and can be edited safely in this example.

At first glance, the relational system seems to be superior in this case, since changing the Java application after the completion of a step is as possible as it was before. But in truth, this is not beneficial at all. Doing this naively will result in broken retrieval-functions due to altered meta data and therefore altered table/column meanings. To prevent that, one has to keep the old code as legacy, just to be able to read old data.

**Altering a step in the workflow before completing it:** It is realised that an employment step should not be completed by the expert, but rather by the owner of a start-up.

The change required is simple. In a `<work = "employment">` node, there is a `<role>` value, which would have to change from "expert" to "owner". As long as that step is not executed in the exact same moment the change is made, this will have no after-effects. In the current interface, one can just use the ‘Change Workflow’ button, click on the node that should be altered and edit this meta-information. This uses two functions explained in section 3.2.1, namely `receiveWorkflow` and `commitWorkflow`.

For a relational system, this means one has to change, recompile and redeploy the Java part. Doing such a thing on the fly certainly falls out of the question, even if it is not completely unfeasible. But more annoyingly and completely different from the new system, the same limitations as in the former use case apply. If the queries are changed for all future workflows, this will also change the queries for
old data since those are the same lines of code. This might or might not break old
data, just like the common phenomenon of new software versions being unable to
read or write files used by the predecessor.

**Changing a form (syntax):** *Instead of only displaying room and time for a meeting, a
bigger company also requires the declaration of the building the meeting is scheduled in.*

The current form that the **meeting** step uses does not have a field to write the
building name into. Since this change does not change any logic behind how it
works, this is easy to accomplish. If one is absolutely sure that every meeting
that will ever be done should have this functionality, changing the meeting step
itself makes sense. Otherwise, see the real sub-typing use case (this is the more
common case as it allows both kinds of meetings to co-exist afterwards). Altering
**meeting** itself can make sense and would be simple to do: Just add a field for that
information on the GUI in the **meeting** class and store this information in the data
node as before.

In case this discrepancy between the current system and the company deploying it
is realised before any **employment** steps are used, this results in no further fallout.
If there are already such steps in the system, the best solution is to go over all that
data and add the needed nodes. If that is impractical, refer to real sub-typing.

The relational system will add a column to the table storing meetings and change
the Java client to display the values. In case there is data already stored there, it
has the same problems. In case this is realised early, it manages just as fine.

**Changing a form (semantics):** *Some company does approval differently. Instead of
having a boolean YES/NO choice, they require a third option ‘abstention’. But the
form for approval does not have that option, therefore the system must be changed
to accommodate this.*

This is a deeper variant of the case before. During the execution of a process,
people often discover drawbacks to an approach which results in wanting to do
things differently next time, or just optimisations. Either way, different code is
required. This often concerns forms, as in the example given before. What makes
this a lot less simple than one might assume is the fact that the old data must not
get compromised, even if the logic behind the step changes. Not having to read
through all old code in such a case is one of the major objectives of the design.

There are two ways to do this. The first one would be to create a new class **vote**
which inherits from **step**, just like **approval** does, see real sub-typing for that.
One ends up with a new kind of step for the new functionality, but leaving the old
one intact. Old data will still get treated as approval, therefore not corrupting it.

The second way to do this would be to just change **approval.as**, as is done in the
first use case on changing forms. If that is done, one has to be careful on the part
that displays committed data as to take into account that some of the new data might not be there for older instances. In this case, since the data displayed is a string, no change is needed at all. In the example given, this would be the better approach to take, but for other form changes using a new step-type often makes sense, see the uses cases on real sub-typing or new forms.

Contrary to the case before this one, a change of functionality often has some small impact on the database layer. Adding a new type of step requires a new commit-function and a new retrieve-function for that type. Adding new logic functionality such as abstention from voting means that the method on counting votes need to be reworked. None of these changes are very complex and they are all very confined.

Relational systems generally solve this problem with additional columns for the new data, which might or might not be NULL in case the data is older. It also means that any query on that table must be rewritten to take the new column into account. In XML, we can just return the parent node in many cases.

Adding a step during the process: Assume the user has just written and uploaded a document in step 5. Now the teams realises that for the next step (using that document) they must first have a meeting and discuss some details.

The workflow layout can be changed by adding following step to the workflow.

```xml
<step id="6">
  <prerequisite>
    <id>5</id>
  </prerequisite>
  <work>
    <type>meeting</type>
    <place>Room 15</place>
    <time>15:30</time>
    <roles>Peers</roles>
    <purpose>Discussion on billing</purpose>
  </work>
  <mode>any</mode>
</step>
```

The system should also check if this addition generates a cycle, since it is easy to mistakenly have three steps depend on each other in a cyclic fashion. This can be avoided by known graph-algorithms, but is currently not implemented.

It is important to note that the system can handle new steps or changing steps (before completion) more than just well. There is no need to write code, as all this information on when what step is supposed to be completed by whom is stored as data. In fact, the system even offers a GUI to add a step during execution, which uses the functions for altering workflows explained in section 3.2.1.

A system relying on Java and SQL on the other hand can often not do this without
major software overhaul, because adding a step usually means adding another state in the Java code with a new interface and logic. It also needs to know where to store this data which again will be different from before. Separating the workflow data from the real data and storing it like normal data is the decisive difference.

**New forms:** Instead of just writing a generic document, the users agree that a more strict layout would appropriate, such as a structured audit containing texts on multiple subjects. These subjects are selected and agreed upon. But when using the documentation-type, none of this can be conveniently found anymore.

This problem can be alleviated easily. Of course it does not make sense to use a documentation-step when this is not really one. The suggested course of action would be to create a new kind of step (called audit) which incorporates the changes. This new type needs its own Flex-class, its own database-code for committing and retrieving and will end up as a fifth step in the basic line-up of steps. Audit will then inherit from step which will supply the vast majority of methods needed, such as any database-access, committing and parsing functions. The only thing that has to be added is code that creates the interface of the new form, similar to documentation (which has about 30 lines of meaningful code).

The new type of step can then be used in any way together with the old types and in any workflow. Since it uses new code only for its new functionality, it will not impact old and finished projects in the slightest.

The RDBMS could handle this similarly, but with the trouble that this new type of step would require a new table and possibly new columns in the project-table with foreign keys in them. This means that someone would have to check every piece of code which accesses the fact table (which is nearly everything) to take the new column into account, which is a very tedious and error-prone task that also scales badly.

But see the next use case for a different take on this problem.

**Review; Real sub-typing:** As mentioned earlier, it is necessary to always inherit the step-class for any new kind of step. But assume a review type is requested. This type is exactly like documentation, except instead of a single text, an additional number should be stored, which is called score. How is such a case handled?

There are two ways in which to do this. The first and most obvious way would be to design a completely new type of step, called review. This new step would then incorporate most (if not all) of the code of documentation and have a <score> node for the new feature. That new type could be easily queried for and all the other basic functions would work. This is the same as the approach taken in the new forms use case.

But there is a much more elegant way. Instead of re-creating all the functionality of documentation, it would be very practical to just declare review as a subtype
of documentation (which of course is a subtype of step still), inheriting all that code too. That way, only code that directly interfaces with the new $<score>$ node would have to be written, which would be less than a dozen lines of code in Flex, and about again as much in XQuery.

Now, any query on documentation would return this data too, but when passed to the stepFactory, the switch statement there can easily be written in a way to return the lowest subtype instead of any of the super-types. It would not be completely wrong to return a documentation step, but not practical for most purposes, since the $<score>$ information could get lost in translation. This way, no data is ever lost or corrupted.

Either way needs changes to stepFactory (to implement the new type), with about the same amount of complexity and effort.

To prove the claim that this would be easy, the review class is implemented as specified here, sub-typing documentation. It works as predicted and required less than a dozen nontrivial lines of code in total.

A relational system not only has trouble with subtypes of subtypes, but it rather fails already for normal subtypes. Even if there is a table where generic step-data is stored, one cannot have multiple foreign keys in a single column which go to different tables, depending on what kind of subtype is used. This can be worked around by having a ‘type-table’ with a column for every subtype possible (and a row per step), declaring all those values NULL if a step does not have a property, and having a foreign key to a table for each type. This leads to complex queries for every step, requiring n joins (where n is the amount of types that exist in the system) for each step retrieved. Ignoring the performance issues this might create, this still results in very complex code.

**Multiple roles:** One might want to have a meeting where the owner talks to the expert. Since those are two different roles in the system, how does one go about this?

Currently, multiple roles in a step are not supported, because there is the question of what happens when a user is in both roles (which is fairly common). Instead, one has to use an employment-step first which adds all experts and all owners to a new, third role, which can then be used in the meeting step. The $<role>$ node in steps and data is not an attribute on purpose, so this could be improved in later versions without breaking old code.

**Single-valued field becomes multi-valued:** Assume we have a person who has the role of ‘editor’ in the system. He or she writes some documents during the workflow. At some point, the requirements change slightly: A second person gets appointed editor too and now both have to write one of these texts each.

For our XML system, this is trivial. We just add the second person to the editor group in the system, and possibly change any future steps the editor has to com-
4.3 Use cases on flexibility

Complete which were any to all (since there was no difference between the two modes, they might be ‘wrong’).

But for a relational system, this problem is far from trivial. Suddenly a foreign key is not a single foreign key anymore, but two of them. If we add a second column, someone would have to dig through the whole code-base and change all references to that column to incorporate multiple values now. If such a change happens again (going from two to three editors), this has to be done once more and with higher complexity. This is a very well known problem.

Authorisation: *A company wants to have a stricter hierarchical structure where not everyone can see all committed data, even if they work at the same project.*

In the current work, this is not supported. It is, however, rather easy to implement. There is a function in the XQuery code which returns finished steps (called getFinishedSteps) which takes project-name, user-name and role as parameters. It goes through all the finished steps and picks those which match. In case only certain roles are allowed to see certain steps, first, one would have to add a node to each `<step>` called `<authorised>`, which contained all roles that are allowed to see the data that is stored. Since the getFinishedSteps-function already joins `<data>` over `<step>`, a simple comparison would add this functionality, possibly with the special case that no authorisation means that anyone can look at the data.

Essentially, this is the same as views, and relational systems generally support them.

Teleconference: *Someone wants to have a teleconference, not a meeting in real life.*

There are (again) more than one approach possible. The first would be to use a normal meeting type, and just put in ‘Teleconference’ in the `<place>` field. Since this is displayed to the user as-is, he or she would be able to just start up the program and call whoever is supposed to meet.

One could also go one step further, and change the meeting-class. Since it is possible to incorporate REST-calls from anywhere in the Flex-code, one could easily add a button to the PreData panel of meeting which asks the database for the contact information of the joining people and then automatically calls all of them. This is not implemented but it is not hard to image to add this.

Media: *Instead of writing a text, one might want to upload a video.*

The beta release of sausalito which was used did not handle files well and for that reason no such feature is implemented yet. But using a different kind of file does not make much of a difference to a database and displaying it could be done with similar technologies like youtube or other flash-based video-rendering services. Flex can execute any Flash-code, making embedded videos an easy task.
5 Evaluation

This is a comparison between the Java/SQL-approach and the new XQuery-methodology. In the left column, the use cases are listed. Some use cases where both systems perform well and no meaningful difference is found are not listed. In the second column we show if the traditional, relational database can support the use case, and in the third column, how well the system presented in this paper works.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Flex / XQuery</th>
<th>Java / SQL</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>New workflow</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Multiple classes of workflows</td>
<td>yes</td>
<td>yes 3</td>
<td>-</td>
</tr>
<tr>
<td>Any versus all</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Executing a workflow more than once</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Removing a member from a role</td>
<td>possible 4</td>
<td>yes</td>
<td>global</td>
</tr>
<tr>
<td>Query over mult. workflows (simple)</td>
<td>yes</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Query over mult. workflows (adv.)</td>
<td>yes</td>
<td>no</td>
<td>-</td>
</tr>
<tr>
<td>Altering a step after completing it</td>
<td>partially</td>
<td>yes 1</td>
<td>local</td>
</tr>
<tr>
<td>Altering a step before completing it</td>
<td>yes</td>
<td>on compile 2</td>
<td>local</td>
</tr>
<tr>
<td>Changing a form (syntax)</td>
<td>yes</td>
<td>yes</td>
<td>global</td>
</tr>
<tr>
<td>Changing a form (semantics)</td>
<td>yes</td>
<td>yes 3</td>
<td>global</td>
</tr>
<tr>
<td>Adding a step during the process</td>
<td>yes</td>
<td>no</td>
<td>local</td>
</tr>
<tr>
<td>New forms</td>
<td>yes</td>
<td>yes 3</td>
<td>global</td>
</tr>
<tr>
<td>Review; Real sub-typing</td>
<td>yes</td>
<td>no</td>
<td>global</td>
</tr>
<tr>
<td>Multiple roles</td>
<td>possible 4</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Single-valued becomes multi-valued</td>
<td>yes</td>
<td>no</td>
<td>global</td>
</tr>
<tr>
<td>Authorisation</td>
<td>possible 4</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Teleconference</td>
<td>possible 4</td>
<td>yes</td>
<td>-</td>
</tr>
<tr>
<td>Media</td>
<td>possible 4</td>
<td>yes</td>
<td>-</td>
</tr>
</tbody>
</table>

1Annoyingly, changing the meta data of a step after it has completed is undesirable, see the explanations in the use case itself.
2Even if possible, it can have terrible side-effects like loss of data.
3It can be implemented with an inordinate amount of code rewriting and is therefore commonly not done.
4These things are definitely possible and not incredibly complicated since all the required technologies are already in place, but are not implemented in the current version. The use case describes how one would go about it.

As can be seen, in more than a few of the use cases, the relational system is troubled to fulfill the requirements. Nearly all of these are related to flexibility which is no surprise. It is also of no surprise that the relational system can get the most basic work done, as it is already employed. Even though one probably would have believed that without proof, a system based on XQuery can also cope with the basic requirements.
6 Conclusion

6.1 Overview

The software developed needs to stand up to two major criteria. The first one is easy to define: It has to work. As far as we know, there is no other system which has gone all the way from concept to implementation, even if some ideas float around, such as [20]. It can be safely said that the proposed system is fully functional, even if rather bare bones. The employed technologies, such as XML and XQuery are perfectly capable to keep up with traditional relational models, and the most impressive features, such as inheritance and separating met data would just not be possible in a relational world. The second criteria is flexibility:

6.2 Local flexibility

All of the system has been designed with the primary objective in mind of creating a process management system which not only runs a single instance of a single workflow (or multiple instances of the exact same workflow, which is what most systems do currently), but rather to construct an application which can run any workflow. The most obvious example of this is the addition of new steps or the alteration of steps that are already there. This works extremely satisfactory and covers local flexibility much better than any relational system possibly could. The ability to do this originates from the separation of meta data and data (see 3.3.2).

In the end, it was even possible to write a simple GUI which demonstrates how steps can be added or edited at runtime with a couple intuitive clicks.

6.3 Global flexibility

Global flexibility still is difficult. This is mostly due to how difficult it is to predict the changes that will happen in the distant future. Still, a lot of functionality is offered which should alleviate these issues by a significant margin. The ability to just write a new kind of step and add it to the system at very low maintenance cost (old code can be left untouched) allows to adapt the software to new environments or changing requirements. Using OO-techniques such as inheritance and dynamic binding makes the process practical and fast. Not only is such a system of interest to the business that was used to demonstrate it (scientific due diligence), but to many more. Thinking of companies or institutions which model their work as processes is easy, to give a few examples:

- In hospitals, patients get checked in and then examined and monitored with different techniques. They have their weekly or daily meetings with doctors who fill out
 papework. Writing step-classes for the different examinations or different treatment systems can be done, and connecting machines which create cardiograms or x-rays directly to a database is nothing unusual. These procedures can change a lot with increasing knowledge on sicknesses and medication.

- For scientific conferences, there are paper submission software systems such as CMT [9] or easychair [15], but both have a hard time adapting to change, since those are coded and finished. They work with a single, defined workflow and are very rigid, alterations during execution are all but impossible. More flexibility would be very useful.

- Any kind of lecture or teaching is also a process. There are lessons, exams, grades and homework, all of which could be modelled and centrally stored. Many universities already employ software to register online for courses, but often, it cannot do much more, nor is it cheap to maintain.

It seems reasonable that the system can be adapted to these environments without reworking the core code, but by just defining new steps and implementing them. In addition, it is possible to invent not only completely new steps, but also to have refined versions or differing variants with subtypes, such as the demonstrated review type which inherits all documentation functionality, but offers something on top of that. The prime advantage of this over traditional systems is significant: Full downward compatibility, no matter the alterations. The Java-solution would have to rewrite code to do this, and at that point, there is a choice: take compatibility into account or not. If it is decided in favour, additional code needs to be written, which adds complications for every database access and gets more and more difficult to maintain. This can be as simple as a switch-statement on a version number returning a slightly different query, but it can also be as complicated as accessing other fields in joins of different tables. It quickly results in legacy code which nobody ever wants to look at again. If it is decided against, code bloat is avoided, but at the cost of the old data model. Of course, this technically works and leads to decent programs, but in the end, the old data has to be sacrificed, something that is rarely an option.

### 6.4 Drawbacks and limitations

But even if the XQuery-solution does not have any of these drawbacks, it has its problems. The first one is certainly performance. Neither the XQuery engine itself is exceptionally fast, nor are the implemented queries very efficient. Some of this can be fixed by adding good caches, but still, it is not expected to achieve the high throughput relational systems offer. Luckily, this is not as important as it could be, since the load will generally be low: Assume a person completes a couple steps per day, each sending (at the worst) a dozen queries and even for a big company this would still result in less than a query per second.

It is also possible to create pitfalls when designing new steps. To give an example:
Having too much functionality in the base step and trying to reduce that in a subclass does not work anymore. To solve this problem, one would have to very carefully write a proper subclass-relationship and then convert the old data into the new subclass, while using the less complicated super-class in the future. Not impossible, but not quite simple either. Generally, writing bad step-types may result in inconveniences later on and it can be hard to tell whether a new idea for a step is bad or not. On the other hand, databases suffer from similar risks, such as a badly designed entity relationship diagram in the beginning.

Another limitation of this work is one of scope. Process management systems rarely are small, designing, testing and implementing a new kind of data-model takes a lot of time, and working with cutting-edge technologies (which still were being developed during conception of this thesis) slowed down progress even more. That means that the feature list is focused on the most important things first. It does not include features such as VoIP, instant-messaging between users or a schedule or calendar directly in the client and it generally feels a bit rough around the edges.

Since this is a thesis on flexibility and adaptability first and foremost, it is left to future work to judge. There are a great many ideas to continue on, and hopefully, this work laid a solid foundation for doing so.
7 References

1. Graphviz, a C-library to display any kind of graph, implemented in sausalito.  
   http://www.graphviz.org/

2. Zavoo SVG Viewer, a Flex-library to load svg-files during runtime.  
   http://labs.zavoo.com/

3. GXL, an XML standard for graphs.  
   http://www.gupro.de/GXL/

4. SVG, a scalable vector graphics file format.  
   http://www.w3.org/Graphics/SVG/

5. Sausalito, an XML-engine. 
   http://sausalito.28msec.com/

6. Adobe Flex, flash for business users.  
   http://www.adobe.com/products/flex/

7. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software.  


9. Microsoft’s Academic Conference Management Toolkit  
   http://cmt.research.microsoft.com/cmt/

10. The company behind sausalito: 28msec  
    http://www.28msec.com/

11. XQuery, a w3c standard  
    http://www.w3.org/TR/xquery/

12. Zorba, the XQuery processor.  
    http://www.zorba-xquery.com/

13. XQuery Update Facility, a w3c standard extending XQuery  
    http://www.w3.org/TR/xquery-update-10/

14. Subversion, a code-versioning system.  
    http://subversion.tigris.org/

15. Easychair, another system to review papers for a conference.  
    http://www.easychair.org/
16. Amazon Web Services. An environment where anyone can run code on a distributed network, paying by cpu-cycles and storage used.  
http://aws.amazon.com/

17. Swiss National Science Foundation  
http://www.nsf.ch/

18. eValueScience, an enterprise which does scientific due diligence.  
http://www.evaluescience.com/

19. ETH: Eidgenössische Technische Hochschule  
http://www.ethz.ch

20. Dr. Michael Kay, Building Workflow Applications with XML and XQuery  
http://www.stylusstudio.com/xml/workflow.html