Bachelor's Thesis

Scenarios, Usability and XForms in Ad-Hoc Business Processes

Alexander Koglin

Supervision
Prof. Dr. Mathias Weske, Hasso-Plattner-Institute, Potsdam, Germany
M.Sc. Gero Decker, Hasso-Plattner-Institute, Potsdam, Germany
M.Sc. Harald Schubert, SAP AG, Walldorf, Germany

June 30, 2008
Abstract

The Bachelor’s project “Ad-Hoc Business Processes”, at the HPI in Potsdam, has investigated solutions for extending systems to Ad-Hoc interactions in business processes and has implemented them in two different environments, the HPI Oryx Editor and the SAP Galaxy Business Process Event Management. This Bachelor’s thesis documents relevant use cases for taking design decisions for both prototyping systems. Three motivating scenarios are illustrated and make the usefulness and the necessity of much greater flexibility clear to the user. Based on these findings, this thesis gives an overview about the user interfaces which help make formerly rigid Business Process Management toolsets much better adapted to the circumstances of current individual user environments. It describes an intuitive work list design, clearly arranged forms for user interaction, and shows a way the user can recognize exceptions during process execution. It goes into greater detail when implementing the user interface design into the prototype with the recent XForms technology and SAP’s Web Dynpro environment. Finally, the results of this thesis underline the relevance of flexible user interfaces for both the future Business Process Management modeling and its execution environments.

Kurzinhalt

Contents

1 Introduction and Definitions 1

2 Principles and Requirements 3
   2.1 Principles of Adhocness 3
   2.2 User interaction in Human Resource Management 4
   2.3 Work list design for Purchase Order Management 11
   2.4 Re-evaluation in Supplier Relationship Management 17

3 Prototypes and Interface Technologies 21
   3.1 Prototype architectures 21
   3.2 User interface technologies in the prototypes 23
   3.3 Task life cycle 25
   3.4 Integrating XForms 29
   3.5 Generating forms and managing data objects 35
   3.6 Web Dynpro prototype 38

4 Summary 43

Bibliography 45
1. Introduction and Definitions

Business process management systems need to support the execution of business processes with IT to help automating the organisation’s business processes. Organisations have to adapt their solutions to the rigid solutions of the market to compete in businesses. Processes have to be analyzed to improve the throughput and, thus, customer satisfaction. Workflows are partly or completely automated processes, that map the process structure onto a hardly fixed flow of ordered activities. Therefore, a large amount of Workflow Management Systems have been installed over the last decades. A Workflow Management System is a “software system that defines, creates and manages the execution of workflows through the use of software...” [15]

The aim of the Bachelor project “Ad-Hoc Business Processes” at the HPI Potsdam was to relax the trend of automation in finding ways of Ad-Hoc interaction. In particular, problems have occurred in situations, which are not as strictly structured, e.g. processes not generating assembly line products, and these involving a high amount of responsible knowledge workers. Typically, users choose the indirect, but more comfortable way of avoiding the Process Management System if their real life activities are not supported. Such workers would rather run from desk to desk looking for support instead of getting help by the Business Process Management System. By finding mechanisms to make workflows more flexible, they shall be engaged using the system approach. In our context, “Ad-Hoc” is called the capability of a system to ease user interaction, to let the modeler easily build grades of flexibility into the process, and to make the system respond efficiently to changes in data. Two prototypes have been developed to show how to implement the mechanisms of Adhocness.

Our project was subdivided into two phases: During the first phase, lasting from early September 2007 to late January 2008, we found appropriate use cases waiting to be applied in business context. For that reason, we collaborated with SAP, Walldorf. Our collaboration consisted of an initial kickoff visit to Walldorf, a final presentation meeting there, monthly checkpoint meetings in Potsdam, and regular
telephone meetings to synchronize our current development status. SAP provided help in each phase with information retrieval and guidance. The worked-out use cases we decided to support are discussed in this thesis in detail and mechanisms to support them are derived (chapter 2). In this thesis, use cases and scenario descriptions we gathered are presented in the Business Process Modeling Notation [2]. With the help of the use cases we found a definition for Adhocness in our special context (section 2.1). The second phase yielded two prototypes developed out of the gathered mechanisms of Adhocness. The prototype architectures shall be discussed in section 3.1 and are to be described in greater detail in [5] and [6].

This Bachelor thesis was written against the background of five further Bachelor theses [6, 10, 7, 9, 5] for the Bachelor project at the HPI in the department of “Business Process Technologies” at the chair of Prof. Weske. As a basis (chapter 2), we shall extensively describe use cases, which motivate the usage and necessity of increased flexibility in Business Process Management. The use cases will deduce solution strategies, that are the basis for concepts and features an advanced system is to provide. Our key features will be developed with ease of use in mind to be able to offer them intuitively to the user within the context of Business Process Management features, well-known to him. This paper follows the idea of deriving requirements, possible solutions and connected problems in order to come to a conclusion about solving strategies. All the described problems will be covered as good as possible with our prototyping systems. It is the objective of my thesis to describe a feasible surface for handling the recent features of Adhocness. So, the first part shall describe the principles of Adhocness (section 2.1) and the fields of process management that can be made Ad-Hoc. Then, it suggests three relevant business scenarios (sections 2.2, 2.3, 2.4), we decided to support. Within the discussion of each scenario, we will consider the scenario’s situation, requirements, solutions, and difficulties. We will highlight special features which will make the chosen solution more desirable.

The second part of the investigation (chapter 3) deals with observing the user interface technologies we used in both the SAP Business Process Event Management (in the following called BPEM) and the HPI toolset based on the Oryx Editor (further on called Oryx). We shall examine the two prototypes built (section 3.1), and place the user interface technologies, we used, in the engine architecture (section 3.2). Based on this overview we will look more closely at the user interfaces we actually built into the execution engine of the Oryx Process Management toolset and point out the involved points of the engine architecture if that appears to be necessary. For the Oryx Editor, we shall visualize how the task lifecycle determines the procedure of task management (section 3.3). In that context, we shall illustrate what the user needs to do in order to manage his outbound tasks in the work list. Beneath, we will look especially at the forms inside the work list (section 3.4). A section (section 3.5) visualizes how we generate XForms [1]. Furthermore, a Web Dynpro prototype for delegation is introduced and Web Dynpro is compared to the Xforms technology (section 3.6). Finally, the contribution and relevance to Business Process Management is worked out (chapter 4).
2. Principles and Requirements

This chapter introduces leading use cases to deduce features we aim to support. For each of the three features an applicable interface design will be presented. We will look at the user interfaces during the execution of processes. For each user interface, design considerations are made, and advantages as well as disadvantages are identified.

For this purpose, we will first examine the principles of Adhocness, illustrate previous definitions of ad-hoc interactions, upon which we built our definition we will propose (section 2.1). Yet, we will keep in mind that our definition is merely an approach instead of an exhaustive systematic classification, because Adhocness, potentially, comprises an unlimited amount of aspects, whereas we concentrate on the limited mechanisms we wish to support. In the first leading use case, we will have a look at the flexibility of interaction mechanisms (section 2.2), the second one will describe how the user gets presented the mechanism of an unstructured amount of tasks inside the already existing BPMN modeling construct Ad-Hoc task (section 2.3), and the third leading use case will describe the need for re-evaluating data objects in order to catch unexpected changes during the runtime of business processes (section 2.4). The entirety of use cases will offer a good overview of what we suggest, where the chances of business processes lie to make them more ad-hoc.

2.1 Principles of Adhocness

Our Bachelor project deals with the term Adhocness in an adapted way. There are some scientific works which build the foundations of flexible and dynamic business process management. They are stated in greater detail in Stefan Krumnow’s thesis, which describes the recent research [7].
Initially, the linguistic term *ad-hoc* means that something is “set up for a special purpose, situation, or case” and has widely been used as a term that refers to individual cases. In the world of Business Process Management, it is applied to the sudden appearance of an individual model instance, for which the activity order differs from that of the other instances for one of the many varying reasons, which Stefan Krumnow describes.

Krumnow especially points out the three dimensions of change [13], as they can occur in workflow management, which are flexibility, dynamism, and adaptibility. Our term *Adhocness* deals with the definition of flexibility in particular, which determines the grade how loosely or partly specified a process model is. *Ad-Hoc Tasks, Pockets of Flexibility* and *Worklets* are constructs which inspired the way in which we have defined *Adhocness* and how we have determined the semantics we committed to our slightly modified readaption of the Ad-Hoc task (section 2.3). The Ad-Hoc task’s runtime behaviour is partly specified and it allows a loose ordering of its contained subtasks.

Adaptability plays another important role in our definition of *Adhocness*. Adaptability is the ability of a system to respond to changes and exceptions. So, all our mechanisms have to be supported by an adaptive process management runtime, since ad-hoc exceptions always require accurate handling. A special case we support (section 2.2) is the invitation of people to let them add relevant information to allocated tasks and, therefore, the accurate support of handling exceptions which would otherwise cause a holdup of the whole process. Workers tend to postpone tasks to their deadline if they are not able to fill in all information themselves.

This thesis illustrates the runtime based behavior of occurring ad-hoc events. The behavior can either result from the modeler modeling a foreseen runtime methodology, as it is the case in Ad-Hoc tasks, or from allowing the end user an unforeseen reaction to exceptions, as they occur during the delegation or before re-evaluation. The runtime based behavior is always a predefined and standardized way of reaction, that must be supported by and integrated into the workflow engine of a Business Process Management System. The following sections are to describe use cases and will propose user interfaces with a referral to the current prototype implementation we conducted.

### 2.2 User interaction in Human Resource Management

After we have illustrated the basics of Adhocness, a use case for the first Ad-Hoc mechanism *Interaction* is described.
2.2. User interaction in Human Resource Management

Use case description

For exemplifying the ad-hoc mechanism of user interaction and its interfaces, I decided for a use case in the field of Human Resource Management, which visualizes how the current state-of-the-art Business Process Management tools fail in supporting the user in the collaboration with involved process participants. We have a look at a scenario in a huge company, which wants to hire a new employee for sales. The process describes how employee information is collected in order to give the department manager information needed for an employment decision. Additionally, an employee contract is prepared to allow the manager a fast signing of the employment offer.

![Employment process diagram](image)

In the use case, critical information is required to meet the employment decision to be delivered by the secretary, let’s call her Ms. Russel. Ms. Russel has to prepare and fill out the following information:

- Employee’s name
- Employee’s address
- Insurance number
- Job description
- Job type
- Suggested salary
- Attaching his curriculum vitae

Assuming that Ms. Russel has no knowledge about common starting salaries in the requested job type, she asks to be helped by the Human Resource manager, Paul Robson. After being helped, the process can proceed as usual and the employment decision is met by the Department Manager.
The employment process contains a sequential list of tasks, which is in a rigid and predetermined order. The missing knowledge of Ms. Russel serves as an occurring runtime exception and cannot be handled accurately. There is no point in the process where an additional action of information gathering could be added. The only chance of exception handling is to cover the process outside of the Business Process Management System.

In figure 2.1 an illustration of the corresponding business process in BPMN notation can be seen.

To find a fix for handling an exception accurately, we have to find mechanisms of Adhocness to be built into the process. The exception is purely of an adaptive nature (see 2.1), that’s why the runtime engine has to support the workaround inherently itself. There are several possibilities:

- **Add an additional task**: A completely new task, handling the exception itself, can be introduced by consulting Paul Robson for the employee’s salary. But, there is no runtime information about what the additional task can look like nor which pieces of information it collects.

- **Redo the existent task**: The process of collecting relevant employee information can be done completely by letting another runtime user of a similar or superior role redo it. But, the already completed parts of the information would have to be filled out repetitive: The reiterating roles reinitialize the current data items respectively.

- **Delegate the existent task**: The activity can be reallocated to a different user by Ms. Russel herself. The already filled out parts of the information will be delegated as well and pieces of information can be hidden or declared as read only if they are critical.

We suggest the last solution strategy as the most effective one. It, actually, turns out to have a number of advantages:

- It enables the user to choose his delegates freely and make them responsible.
- It allows the user configure many delegation parameters, e.g. setting a deadline.
- It not only lets the user set data attribute values, but also visibility rights on them.

Mike Nagora’s Bachelor’s thesis [10] focusses on the roles and rights management of the delegation principles in greater detail. In the next paragraphs, we shall suggest a detailed solution strategy and will develop an appropriate user interface design for it.
Variations of the delegation pattern

The user interaction comprises several variations to the pattern, described the last paragraph. For a more detailed theoretical examination of them see [10]. In this place, we merely want to mention them before having a closer look into their user interfaces at a later point.

- **Delegation**: The standard delegation pattern lets the user choose a single delegate for getting his task done or for having a correction on the task. The user can either keep the responsibility or simply pass it to the delegate to be done with the task. Being responsible means to be accountable for decisions and to have a review after the delegate has finished the processing.

- **Additional approver**: An additional approver is a person, who has a second look at the decisions of another process participant. The additional approver use case made it necessary to add an additional step, the owner of which has the same data access and visibility rights to the task and the final responsibility for task decisions. This pattern can be mapped completely to the delegation pattern (see [10]).

- **Multiple delegation**: The multiple delegation pattern is the multiplication of delegates on multiple task items. Conceptually, the task can be divided into several task properties, for which the multiple delegates have own, mutually exclusive visibility rights. Yet, each delegate can have a read access on each item. The initial owner can determine several collaborational owners contiguously and reobtains the task when all delegates have finished.

- **Shared task**: This concept was introduced by SAP and means to have a shared and concurrent processing on tasks. Beneath the “potential owner role” appear several “collaborational owner roles” which can work on the task themselves. Conceptual problems occur in missing data locking strategies, because the collaborational owner’s access does not have to be urgently mutually exclusive.

User interface design

Therefore, following requirements for the mechanism’s user interface design arise:

1. The work list should offer an unallocated task to the user if he has the access rights to allocate it. If it is assigned to a user, he, first, needs to see superior information of interest like the original owner, then the message why it was delegated to him, and the deadline he has to keep in mind. Having recognized them he can change the task’s input and output data.

2. The data have to be persisted if the user suspends a task and recovered if he resumes it. The system must not confuse the user with unexpected behaviour and discard any data items. Any delegate is to have the chance to suspend a task.
3. The common way of conducting the process should not be affected. The typical way, in which the process management system behaves, is not a feature of change. The capability to delegate a task or to use any other ad-hoc routines are additional features offered to the user as an alternative. The mechanism shall not be propagated as obvious or regular feature of a running task, because it describes a routine of individual case treatment.

4. Only when the user has chosen the option to delegate, he is presented the differing visibility rights on data items. Visibility rights act on the item level of data objects and have to be presented in an intuitive manner. We have identified the three states “invisible”, “readable” and “writable”.

5. Directly after the user filled in assured data items, he is to be given the option to choose the delegates, a deadline for each delegate, and a message to pass to the delegate, so that he can submit the delegation routine.

6. The user will be offered a list of delegates. The delegate should be chosen from among the group of people, who initially were “potential owners” of a task, but not any other users. Otherwise, the user delegation would threaten the system security and authorization concepts.

7. A typical user delegation in business people’s understanding comprises a “reviewing” functionality. If “reviewing” is enabled two possibilities unfold: Either “reviewing” shall be permitted the initial owner for process performance purposes, or the “reviewing” is allowed to any preceding delegates to check what the subsequent person did. This strategy calls more for the responsibility and awareness on the part of each delegate.

8. Another issue lies in the progress indication of the delegation procedure. A responsive interface should give all the feedback possible. The user should be informed who of the delegates has already submitted their correction procedure or has, in turn, delegated to another user of trust.

9. The “Additional approver” use case should be considered in the interface design of the delegation. The “Multiple delegation” and the “Shared Task” concept make a slight modification of the interface necessary. It is to be considered how a list of users with their multiple and differing visibility rights can be managed. Therefore, different concepts with their pros and cons will be weighed against each other in the following.

Based on these requirements, we particularly aim to describe the realization in user interfaces, stress special features that can be supported, analyze the problems that inherently exist, and, then, refer to the concrete implementation in chapter 3.

Figure 2.2 describes the most basic behaviour of a delegatable task. It uses the notation of UML state machines as they are specified by the OMG [11]. It describes the simplified features of a task to illustrate the delegation principles at execution time and to concentrate on the interface parts in visual surfaces. We distinguish four
2.2. User interaction in Human Resource Management

![User Interaction Diagram]

Figure 2.2: The task behaviour (as a UML State Machine diagram) with cut outs of responsive user interfaces in states “running” and “delegated”

different states of a task which a user can differentiate: The task can be *allocated* if it has not yet been started, it can be *running* if it’s form has only now opened and is in progress, *delegated* if the user decides to refer the task to another user, and *submitted* if he has already submitted his decision (which can be the finishing or delegation of the task). If he chooses to remain responsible, the additional state review is added.

The first proposal of a user interface is the less intuitive one, but, therefore, it can be implemented rather easily. Figure 2.2 depicts the user interfaces which are shown during the *running* and *delegated* state. The running state offers the simple and in former Business Process Management Systems well-known execution form we wanted to keep (see requirement 3). The execution form, in which the user has the possibility of submitting and cancelling the fill-in process, is extended by an “interaction” button to give the user all the possibilities, the interaction should have as described in the previous paragraph.

By taking the decision to interact, the form is enriched by similar (otherwise invisible) interface elements. First, metadata information on the delegation feature is provided (as claimed in requirements 1 and 5). The user has the task to choose the delegate, leave him a responsive message in order not to *shock* him by leaving him with an unexpected task, and, optionally, set him a limited time or deadline to submit.
After the delegate has solved the delegation issues, he has to set the visibility items (as required in point 4). I do not aim to develop the idea of visibility rights in great detail, because another paper has already investigated the issue of rights and roles management in Ad-Hoc Business Processes [10]. The only point I need to mention is that an intuitive coloring metaphor could be set for making the decision of visibility more understandable for the unexperienced user. The “invisible” state can be set red (for a not regarded element value), the “readable” state yellow (for not having the choice to edit, but to see the value item), and the “writable” state green (for having all editing options). Thus, the “writable” state shall be the standard right which is set initially, because in most cases you want to pass the delegates the whole amount of items.

The biggest disadvantage of this approach, as we implemented it, is the not too intuitive navigation (see rounded number items in figure 2.2). Before the user has not investigated the features once, he can hardly understand what the “Plus” button means and why the same window reappears. He cannot have the contextual information about when he is in the delegation state and how he can return to the running state. We try to give him this information by using the headlines, which name the status at the top of other information.

A much better, but not implemented approach, is the suggested “popup window approach” in figure 2.3. The execution form is exactly the same, but there are differences that have to be regarded in more detail.
This approach proposes to open an additional popup window, that takes over the interaction issues. It is much more focussed on the “Multiple delegation” requirement (as can be seen in requirement 9). The popup window affords a much better overview, because the popup window is a modal dialog that grays out the background execution form. This makes the scope of “Delegation state” much clearer.

The delegates, who were nominated, are listed in an intuitive “list view” which also determines the precedence rules among the delegates (requirement 6). An advantage is that the approach offers a slim interface. The disadvantage is that it hides important information from the user. The visibility rights on data items are not presented in the list view as that would blow up the user interface enormously. This disadvantage is overcome with an additional “catch eye” button, left of the delegates’ names. If you clicked on it, you would be presented all the visibility rights on the data items for the regarded delegate.

The reviewing functionality (requirement 7) lets the task reappear in the user’s inbox when all the delegates have finished their work and have submitted their appropriate view on the task. The interface looks like the normal execution interface of a task. It, additionally, contains the hint at the top that the task is currently reviewed. The concerning user interface implementation is described detailed in the implementation part, where the prototype description is given (see section 3.4).

Additionally, the indication functionality (requirement 8) has to be visualized and such a concept is shown in the next paragraph, when the user interfaces of desired, ad-hoc enabled work list designs are proposed. This conceptual part is enriched by chapter 3, where we observe the actual user interface implementation of the delegation functionality in the Bachelor project’s XForms based execution environment.

### 2.3 Work list design for Purchase Order Management

New functionality in Adhocness asks for an adapted user interface to handle the amount of incoming Ad-Hoc subtasks, to achieve a rapid overview about the current delegation status and to see which data still have to be provided to proceed with the processing status of the business process. When we present the actual work list design, a motivating use case shall be described to show how the handling of data dependencies makes a process more flexible and how an Ad-Hoc subprocess, including varying independent and unordered tasks, can help manage the modeling and the execution of ad-hoc decisions.

With the Purchase Order scenario above and the Human Resource employment scenario from the last chapter in mind, requirements to the work list design are to be derived and changes to support Adhocness will be discussed. The requirements, again, lead to a proposal for a well-designed interface, for which screenshots from the actual implementation of the HPI execution engine based prototype\(^1\) shall provide an overall picture.

---

\(^1\)The HPI execution engine is described in the following chapter
Use case description

The purpose of the following use case is to exemplify the application of a comfortable modeling with Ad-Hoc tasks and to see the data dependencies in use. The section does not aim to show how the Ad-Hoc task modeling approach helps ease the modeling of a huge amount of decisions, since finding reasons is not a runtime purpose and since this topic has already been dealt with in Stefan Krumnow’s thesis [7].

![Diagram of Purchase order process with product approval of a SRM department](image)

Figure 2.4: Purchase order process with product approval of a SRM department

The use case is settled in the area of Supplier Relationship Management (SRM) and was partly proposed by our business partners SAP and partly by [3]. In the automated process management of profit oriented organisations, it is of great interest to perform many tasks in parallel in order to gain a high process throughput and outcome of the process. Also, many processing steps in reliable organisations have to be approved in order to secure process integrity and check outcomes twice or even multiple times.
The use case is split into two relevant parts. The first one describes how the shopping cart is produced by three parallel workflow activities *Define Maximum of Expense* evaluated by a department manager, *Define Shipment Address* processed by the secretary, and *Create Shopping Cart Items* operated by an arbitrary company employee. *Check Availability in Stock* can be performed, optionally, in order not to purchase items redundantly and can be done when the shopping cart items are created. During the check, the shopping cart items are recalculated and potentially dropped, so that the *Shopping Cart* should be re-evaluated (see section 2.4). The three outcomes are merged together by the *Package Order* activity. The second part of the workflow describes the three parallel approval activities that are skipped if certain conditions apply to the shopping cart items. The *IT Approval* is only performed for IT items, the *Product Approval* has always to be run through and the *Cost Approval* is undertaken by a cost centre if the shopping cart exceeds a defined cost value.

In our work list design proposal, I decided to deal only with the first part of the process, because it satisfies our requirements for introducing a proper work list design completely. The second part will be focussed on in the following section, where we propose a runtime design for re-evaluation.

In figure 2.4, you can see a proposed ad-hoc modeling approach in an extended BPMN modeling notation, as Matthias Kleine introduced it for Adhocness purposes after investigating the disadvantages of the former approaches to Ad-Hoc task modeling [6]. The first point to notice is the parallel processing of the shopping cart preparation tasks inside an Ad-Hoc task. They can be processed in parallel and produce data objects that act as preconditions for the *Package Order* task. Immediately after the required data objects are produced the activity state can switch to running. A later examination will focus on the mentioned features *Ad-Hoc sub-tasks, data dependencies and sequence flows inside of Ad-Hoc tasks*. How the data dependencies have been realized in our prototype is described in section 3.5. In the following chapter, we will focus on how this influences the UI design of a work list. For design decisions of the work list, made during our development stages, the thesis of Lutz Gericke shall be mentioned, as he has implemented the work list interface on the engine side [5].

**User interface design**

At this stage, a user interface design shall be proposed that supports a work list for letting the end user manage the so-far mentioned features of Adhocness. The focus lies in extending a work list with *Ad-Hoc* features. The user interface shall not be pictured with the implementation as this work has already been done in [5].

Initially, we want to examine a general work list design and what it offers to conform to the user’s needs. A general work list design (as it is dealt with in the *SAP Process Desk*), typically, provides the features of a usual Mail client: Each task wanders into the user’s inbox, exactly as mails do in the Mail client pendant. The tasks can be allocated, filled in and submitted, but there is no capability of the system to support
2. Principles and Requirements

Figure 2.5: The left hand side navigation of our work list with the capability to see cases, have analysis features for cases and tasks and to spawn instances of processes user interaction, task suspension, or task skipping. To free the user’s possibilities, we want to support such mechanisms of task handling as the next passages will reveal.

An ad-hoc work list design shall not drop the feature of behaving like a Mail client, but extend it. Our work list supports the concept of a Case View that shows all tasks of a started process instance (figure 2.5). The specific order of cases can be thought of as a flat folder hierarchy to keep an overview about the started process instances. Since the user does not care for cases he usually acts in the special Case View “Any cases” which corresponds to a Task View of common Business Process Management Systems. This feature has nothing to do with Adhocness, but is introduced here merely for a better understanding of the features of the prototype’s work list later on. Cases are spawned in a special left hand side navigation tab, where the deployed process models can be seen (figure 2.5). What could be supported additionally is to drop running process instances. But, this is not a point under discussion here. The requirements of our work list design, resulting from the above use case and the delegation features are the following:

1. The usual behavior of running a task and submitting it shall not be changed, because it is the expected behavior of a user familiar with Business Process Management Systems.

2. Activities have to be suspended, resumed, skipped, delegated and reviewed without having to allocate the activity nor without having to display the attached form. All the extended features have to be available in one click for the comfort of the user.

3. There has to be a difference between tasks that can be done (optional) and tasks that have to be done (required) in the work list. The user shall always have an overview about which tasks are necessary for the progress in case processing.

4. Statistics for running tasks are to be provided. A delegation progress indicator turn out to be of special interest to the original owner of a delegated task, since it gives important feedback on how many delegates there currently are and who of them submitted the task.
5. The extended Ad-Hoc task has to be supported by the work list. It is required not to give the user the impression that he is located in the processing of an Ad-Hoc task, since a business user is not interested in knowing about an Ad-Hoc task in the first place. Instead, the user shall get all the Ad-Hoc subtasks in his inbox that are ready for processing.

6. Sequence flows inside of Ad-Hoc tasks have to be supported. Tasks being reached by sequence flows appear inside the work list if and only if their preceding tasks have all been processed, i.e. are completed or skipped.

7. The work list has to be enabled for data dependencies: Tasks shall be such, that they could be processed only if their input data items were available. But they have to be indicated as not meeting all their preconditions.

8. A further progress indicator for a task has to show which input data items a task claims for being enabled and how many and which of them already have been delivered.

These requirements were either completely or partly implemented in our HPI prototyping work list. The integration shall be considered in the following.

![Figure 2.6: The actions of tasks are independent features and are derived from the Petri net mapping of a usual task](image)

![Figure 2.7: Optional task “Check Availability in Stock” decreases in inbox ordering](image)

As stated in requirements 1 and 2, the work list concept is capable to run, suspend, resume, skip, delegate and review tasks (see picture 2.6). However, a task in our work list can be suspended, resumed and skipped at any time, as visualized in figure 2.6. The capabilities of a task appear directly after the task is selected on the right hand side work list part. For an activity, you first have to run a task before you can
interact on it with other employees. This behavior stems from the design decision to implement the first proposed delegation UI, already described in 2.2. If we decided to support a popup window, which reimplements the interaction, the separation of running and delegating a task would have been made possible. Requirement 2 was not reached completely, because our work list needs an additional user submission inside forms after pressing the suspend, resume, skip, delegate, or review actions. That was necessary, because transitions inside the execution engine were still not enabled to be finished by an HTTP request.

**Figure 2.8:** A conceptual delegation progress indicator depicts the actual status of finishing the delegation

Requirement 3 states that optional tasks have to be separately visualized in the work list implementation. If you imagine having three tasks Define Maximum of Expense, Define Shipment Address and Check Availability in Stock in your inbox, the optional activity has to decrease in the inbox’ ordering and is to be marked with a question mark after it (figure 2.7).

**Figure 2.9:** “Package Order” does not fulfill all its preconditions: The input data object “Cost Max” is not initialized and therefore marked in red
A delegation progress indicator (requirement 4), as depicted in figure 2.8 states the current delegation status. It describes how many delegates of the task have already submitted their supplemented results. The progress means the quotient between the submitted and the overall delegates. It is not implemented in the current work list, but can be imagined as a valuable help.

An Ad-Hoc task does not have to be presented to the user, as point 5 requires. For all the subtasks that do not have missing input dependencies or foregoing subtasks, a direct appearance in the inbox exists. Input data objects of subtasks (requirement 7), which are not available, can be presented in red color. This is a feature which was not implemented in our current prototype. If implemented, it would give the user the chance to prove why a task does not meet its preconditions and would help him decide which data objects are to be delivered faster. The idea is conceptually depicted in figure 2.9. The Package Order task does not meet the precondition of a maximum of expense, so the input data object Cost Max is colored in red.

2.4 Re-evaluation in Supplier Relationship Management

This final conceptual section shall introduce a use case for re-evaluation in order to come to a solution for managing time variant data objects. Values of data objects could be modified as the requirements of internal process participants or of external circumstance changes, e.g. the optional availability check of the purchase order use case (figure 2.4) requires that the shopping cart has to change and therefore the approvals have to be redone if necessary.

The section before last introduced an extended user interface concept for user interaction that we implemented. In this section, we will propose our additional modification to interfaces for informing the end user about critical changes in data and present the historical data items in comparison to the modified data items in a usable manner. Philipp Maschke’s thesis [9] describes the implementation for re-evaluation in our prototype. How the design is integrated in XForms is to be presented in section 3.4.

Use case description

Let us now reconsider the second part of the SRM use case in greater detail. The organisation regularly orders goods in short supply with a special SRM system. It has an amount of approval processes that underlie various organisational rules. The rules specify the approval checks that have to be done. The ordered items have to be approved by approval checks which are processed in parallel for process performance issues.
Let us assume that the staff member Paul Simon orders two shopping cart items: A Mac computer and a print-out paper (see figure 2.10). The Mac computer fails the IT approval check since the organisation does not allow the ordering of Mac computers any longer. If the optional check of the SRM department now unfolds that there is already a Mac computer in stock, the data item *Shopping Cart* changes and it has to be re-evaluated for all the approval checks. The re-evaluation skips the IT approval since the shopping cart does not include any relevant item for an IT approval any more.

In current Business Process Management suites, there is no chance during workflow execution to have changes to data objects that influence the so-far executed workflow processing. Already known and common solutions have to cancel the process instance and restart it with the updated data items. Alternatively, changes to data items could be explicitly modeled out with an exception handling for each activity. This would add an unnecessary amount of graphical complexity to the models. A better process performance would be gained if a needless repetition of processing of already done activities was skipped automatically inside the workflow engine.

**User interface design**

Different requirements to the user interface design result if re-evaluation is implemented:

1. The modification of already initialized data items during the process execution must be supported by the engine architecture. Therefore, a low level support of data dependencies, as they are already mentioned in section 2.3, can be implemented. Many tasks of the process may have a right to write to the same data items and write operations have to be restricted as transactions.

2. The user has to be informed that and why a task has to be re-executed. The corresponding data object, that triggered the re-evaluation, has to be mentioned because the user wants to be aware if he already processed the same task once before and if a task is worth redoing or not.

3. The user is to be shown the old data item values right before the change. These values allow him to decide if he has to repeat or recall his previous decision. Additionally, he shall be presented exactly those elements of the output data objects affected by the change. The other data items have to be grayed out for easing the user’s decision.

In figure 2.11, a proposed interface design for re-evaluating data objects can be seen. Remarkably, the changes of data can be made clear by a visual feedback through the use of signal colors. Our current prototype supports both re-execution feedback and signal coloring of changes in data items. The implementation is based
2.4. Re-evaluation in Supplier Relationship Management

Figure 2.10: SRM scenario with substituted Mac computer item for which the Shopping cart has to be re-evaluated

on XForms and described in section 3.4. Changes in data are triggered by modifying the current token value 2.12, whose token structure is sketched in 3.3. It would be desirable to change the data object values inside tasks. In order to do so re-evaluation events on data objects would have to be implemented to the engine and triggered if data items change. Now, we have described the prototyping surfaces for user’s re-evaluation. For a more detailed description on what has to be changed to a complete Business Process Management suite if tasks are allowed to be re-executed, and how far a process instance has to be rolled back if data changes, refer to the paper on re-evaluation [3]. It examines the concrete phases of an engine’s re-evaluation algorithm. Philipp Maschke describes the concrete implementation of re-evaluation at the HPI Oryx Editor and the execution engine of our prototype [9].
Figure 2.11: The “Product Approval” is re-executed and gives a visual feedback about the changed data items.

Figure 2.12: The re-evaluation can be triggered by modifying token values of data objects running through the petri net.
3. Prototypes and Interface Technologies

After describing the requirements for a well-adapted ad-hoc user interface design in the last chapter, we have to take a look at the current prototype implementation of the features we developed at the Bachelor’s project “Ad-Hoc Business Processes”. First, we shall analyze the overall prototype architectures of the two ad-hoc enabled prototypes (section 3.1) and illustrate where the user interface technologies in the prototype reside (section 3.2). Then, we will examine the task life cycle which builds the foundation of the HPI execution engine (section 3.3) and how the interface technology XForms corresponds to it (section 3.4). Interfaces are created automatically out of various design time parameters, the modeler has to define (section 3.5). Finally, we will give an overview of the current interface technologies at the SAP AG and identify how they can be used to support the requirements of ad-hoc user interfaces, as they were already stated in chapter 2 (section 3.6).

3.1 Prototype architectures

Our Bachelor’s project “Ad-Hoc Business Processes” partly developed two prototypes in parallel. A team of two students worked on extending the Galaxy Process Composer for letting it model Ad-Hoc constructs (see the Bachelor’s theses of Stefan Krumnow [7] and Matthias Kleine [6]), whereas four others, including me, explored modeling and runtime concepts at the HPI Business Process toolset (see Bachelor theses [10, 5, 9]).

The granular architectures are quite similar when it comes to a comparison (see figure 3.1): The SAP Business Process Event Management, with its recent code name Galaxy, basically consists of three main components. A similar partitioning can be seen in the HPI toolset that was created by a number of participants, especially present and past students of the Hasso-Plattner Institute.
There is always a modeling component, responsible for creating business process models. SAP delivers the “Process Composer” and the HPI toolset provides the “Oryx Editor” [14]. Oryx consists of extensible stencil sets letting applicants model numerous notations. The Process Composer offers the Business Process Modeling Notation [2]. The Process Composer and the Oryx Editor required changes for allowing them to model Ad-Hoc tasks, data dependencies, re-evaluation, and Shared tasks [7, 5].

The process runtime is a complex architecture, including a process model compiler to deploy the model in an executable runtime format and an execution engine for letting the system manage deployed diagrams in an expected runtime behavior. SAP comes with a “Compiler” and a “Process Server” and the HPI toolset offers the “BPMN2PN Converter” and the “Petri Net Execution Engine” [8]. Extensions have been made to the Converter and the Petri Net Execution Engine at the HPI side and concern the various interaction mechanisms, the execution of an Ad-Hoc task, data dependencies and the re-evaluation mechanism. The complexity of the architecture made it necessary to leave out a correspondent development at the SAP runtime. The subsequently described implementations mainly concern the HPI runtime environment.

The user environment comprises all the components an end user can see when executing business processes. It contains a work list, for which an appropriate design is suggested in section 2.3, and displays the forms that are presented to the user for running tasks. The HPI toolset offers a single “Worklist” for this purpose and the SAP BPEM has a “Process Desk” that makes the administrator capable to manage business processes and their parameters as well as a task list for task management.

Process runtime and the user environment together build the execution environment. The HPI execution environment architecture is sketched in figure 3.2. It consists of a Petri Net Execution Engine [8] that uses the Web Browser technology XForms [1].
3.2 User interface technologies in the prototypes

The user interface technologies we used in both prototypes are to be described in the following section.

Figure 3.2: Execution environment architecture of the HPI Business Process toolset

3.2 User interface technologies in the prototypes

After we have introduced the granular prototype architectures, we shall focus on involving the user interface technologies. The HPI execution environment in figure 3.2 consists of a work list, that shows the enabled tasks a user may see, and can display forms attached to a task to the user. The work list implementation was done with the help of the graphical Javascript-framework ExtJS and accomplished by Lutz Gericke [5]. The graphical presentation of tasks is done by forms using the XForms technology, which is to be classified in the following.

XForms technology in the HPI execution engine

XForms is a web browser based technology of the W3C consortium for data recording in forms. It currently evolves into an important standard for the World Wide Web as it is platform independent and will be part of the XHTML 2.0 standard. We have developed the prototype with the current version 1.0 of XForms.
The XForms technology is firmly integrated into the execution engine. The Petri net engine \[8\] in it is written in Ruby and uses the browser based XML dialect XForms for displaying data. XForms has not been supported in recent browsers, yet there is the possibility to render XForms with an XForms Firefox plugin. To display it, you have to put an XForm behind transitions. Beneath TransformationTransitions and AutomaticTransitions lies FormTransitions in the Petri net engine, which the user can attach XForms for the purpose of data display. If a transition of the Petri net is no FormTransition, a standard form is attached or the transition is executed without the user’s impact. The user retrieves the form with an HTTP request to the transition URL and the form is delivered to him in the response in an XHTML 1.1 format.

The Oryx Editor must offer mechanisms to specify executable diagrams. Therefore we implemented a special stencil set to support an executable subset of BPMN called Executable BPMN. To support executing Executable BPMN diagrams XForms are used as the technology for data display of common tasks in the workflow. The XForms are automatically generated for any ExecTasks in executable diagrams.

During the transformation a Petri net structure is created for the task to let the task execution support several properties of an executable task such as suspending, resuming, delegating, and skipping it. The Petri net life cycle of a task can be seen in section 3.3. It supports the semantical separation of a process flow, its activity data, and metadata information on the Petri net level structure. For transformation algorithms used to translate BPMN into Petri nets see the Bachelor’s thesis of Philipp Maschke \[9\]. During the transformation process of the above mentioned Oryx Converter component, an XForm is generated and attached to special FormTransitions of the task’s Petri net structure as can be seen in 3.4. If the transition FormTransition fires, the XForm it is presented to the user. Therefore, the XForms can have a number of states, one for each FormTransition inside the task’s Petri net structure.

**Web Dynpro inside the Process Desk**

Web Dynpro is the framework for the development of user interfaces for SAP’s Netweaver environment \[4\] based on the MVC design pattern. It distinguishes the development of single Web Dynpro applications for use in Java, ABAP and .NET and the creation of single user interface components that can be attached to and reused in different applications. The Netweaver development environment delivers a design tool for the creation of Web Dynpro components. Web Dynpro components ease the development of user interfaces, because their data can be easily synchronized with the backend.

The main advantage Web Dynpro offers is that its UIs can be rendered for many non-proprietary front end types, e.g. Javascript enabled web-browsers, and can be used within different programming languages and standards. So, even the Business Process Event Management architecture uses it inside its Process Desk for displaying forms attached to tasks. If you model a task in the Process Composer, you can attach...
Web Dynpro components to it. When the process is deployed into the Process Server the Web Dynpro form is stored with it. The Web Dynpro components’ application is started when you double click a task and the data inside the browser based Web Dynpro application is synchronized with the component model and, thus, with the case data inside the Process Server.

As already categorized into the architecture above, the Process Desk is the web based execution environment for administrators and end users. Administrators can observe a list of processes, underlying databases, statistics of running instances, and execution logs inside the SAP Management Console, whereas end users manage their tasks within the UWL integration and work on them within the Web Dynpro forms. The Process Desk with all its mentioned technologies is completely made for being displayed in browsers.

Early, we built some XForms and Web Dynpro prototypes for delegation that had to demonstrate the delegation principle and its functionalities. In section 3.6 I aim to present the Web Dynpro prototype to show how a delegation principle would have been integrated to the surrounding of the SAP execution environment.

### 3.3 Task life cycle

#### Transformation inside the Oryx Converter

As mentioned in section 3.2 instantiated BPMN tasks from an Executable BPMN diagram shall have different capabilities like skipping, suspending, resuming, and delegating. We decided to support the execution of BPMN diagrams in structurally and formally easy Petri net semantics, so that the behaviour of such a task must be mapped into a Petri net structure. Thus, a task can be in an amount of states which can be seen in figure 3.3. Each task of the BPMN diagram gets initialized if a new modeling instance is created. So, initialized tasks are in the state initial when they come to life. If the task can be processed and all preconditions as data dependencies (see section 3.5) and the incoming process flow are fulfilled, the task gets in the ready state. The task can be finished at any time by skipping or completing it. After the task is allocated, it is in the running state and can, then, be completed or delegated. The delegation can correspond with an additional review if this is desired by the user.

The capabilities of a task are mapped into a Petri net life cycle, that has grown evolutionarily during the development process. So, the conceptual model of a task in the meantime consists of 13 transitions at the time of writing. An advantage of the standardized life cycle is that the task’s behavior is decomposed by its Petri net pendant: Each transition can present a different part of the behavior of a task. The Petri net structure of a task is depicted in figure 3.4 and described in detail by Philipp Maschke [9]. It uses Petri net semantics with places that can store an arbitrary amount of tokens, transitions that consume tokens of every incoming place, and arcs to allow the process flow amongst them.
Each task has context places $pl_{context\_id}$ with the task ID in its name in order to store execution data as can be seen in the code snippets below for context places. The state model of a task can be mapped directly to the Petri net structure. A task is first enabled by $tr\_enable$, when the Petri net is instantiated with its unique instance case id the Petri net engine manages. Afterwards it is allocated in $tr\_allocate$ by resolving the task’s defined role with a concrete user. There the task context is first set. A task can be suspended and resumed as $tr\_suspend$ and $tr\_resume$ unfolds it. Finally, it can be finished with $tr\_submit$, which completes the task automatically.

The possibility of delegation is also, inherently, built into the structure and is to be described in the following. The context places inside the Petri net structure are dotted, because the sequence flows, which connect them with transitions, are faded out for the purpose of intuition. Any of the transitions inside our task specification have connecting links to the task’s $pl_{context\_id}$.

$tr\_enable$, $tr\_consume$, $tr\_reset\_user$, $tr\_done$, $tr\_merge$, and $tr\_finish$ are Automatic Transitions as they do not have a user interface. $tr\_allocate$, $tr\_suspend$, and $tr\_resume$ are Transformation Transitions, because they have to be manually submitted and forms will be set up for them by the engine. $tr\_delegate$, $tr\_submit$, and $tr\_review$ are Form Transitions in order to put a manual user form, provided by the XForms technology, behind them.

The Petri net structure, we have just described, conforms to tasks of the translated BPMN diagram. The overall BPMN diagram is translated recursively with all its scope objects, subprocesses, and Ad-Hoc subprocesses making a Petri net model truly complex. BPMN diagrams also contain data objects which are directly mapped to data places in our Petri net structure. The precise specification and the advantages of data places and their transformation will be explained in section 3.5. Initially, tasks have single read arcs to synchronize their local data inside the sequence token with the global $pl_{data}$ places if the task is only connected to the corresponding BPMN data object. If and only if the task has a read/write relationship with the data object, it can also write its local data back to the global data object after finishing.
Executing the life cycle inside the Petri net engine

The engine understands different types of transitions than the Oryx Converter: AutomaticTransitions do not require any interaction with other systems or users outside of the system. They contain XSLTs to transform the input tokens into an output token. ReceiveTransitions are form attached transitions to interact with end users of the engine. SendTransitions shall not be dealt with here as they send an HTTP request to other systems.

Every transition has its own input and output token XML structure which it understands. The output token values, which the transitions put at context places, are shown in the code snippet below and are placed inside the metadata tags. At all the other places of the net, another structure of token is put: The structure of flow token is shown in the next section inside processdata tags. Context places contain values for the execution analysis such as the start time value, the current task status, the next delegate that will receive the task, if the task is set to be delegated or to be reviewed (determines which arc conditions apply), and if the firstOwner of the task wishes to review it later (checkbox for Stay responsible for the process in figure 3.5), what the message for the next delegate is, and which deadline the delegate’s processing underlies.

```
<startTime>2008-06-12T13:31:44+00:00</startTime>
<endTime />
<status>running</status>
<owner>Steve Jobs</owner>
<delegate /> <isDelegated>true</isDelegated>
<isReviewed>false</isReviewed>
```
There are two ways for transitions transforming the input tokens into one output token. XSLTs are XML documents defined by a recommendation of the W3C and transform the input XML structure into an output XML structure. They can be attached to Automatic Transitions. The XForms technology itself is capable of carrying out similar transformations by copying complete XML subtrees between model instances of the forms (see section 3.4). Arc transformations separate the one and only output token with its metadata and processdata format into the different token values relevant for the pl_context_id place of the kind type=CONTEXT and the places of the kind type=FLOW.

As you see in the Petri net life cycle structure, each transition can contain a guard condition prior to it. The engine evaluates the guard conditions to determine the sequence paths taken by the incoming tokens of transitions.

The simple delegation, as described in section 2.2, means to give a delegate a limited or scoped view on these data items, that the original owner holds. You can refer a task to another person, so that chosen data items can be edited by him. Each XForm, which the ExecConverter attaches to tasks, supports this behavior completely, as it is described in section 3.4. In tr_submit the user is presented the decision to submit or to delegate a task, therefore, a special flag isDelegated inside the process token is set. The tr_delegate transition contains a guard condition that, then, is evaluated and can fire if the isDelegated state inside the token is set. There, the user can decide who he wants to delegate the task to. After that, the user in the token is set by the tr_delegate transition and can delegate once again. The simple delegation feature supports the only original owner to review the task again. This is reached by deciding if the wantsToReview flag of the token is set, before the task finishes. It is set by the Stay responsible for process checkbox in the first delegation form.

In chapter 2.2, the multiple delegation has been described as a concept. As Mike Nagora makes it clear in his Bachelor thesis [10], multiple delegation is a certain way to see the Shared task principle implemented with a secure roles and rights management. If the user specifies in tr_submit that he wants to use a multiple delegation, tr_delegate produces tokens in pl_running and pl_consume. If the user adds more delegates with the Delegate option, tokens are repeatedly added in pl_running and pl_consume, that are returned to the reviewer in tr_reset_user by resetting the owner in the token data. The different versions of data items, created by the delegates, are merged in tr_consume with a special merge algorithm.
3.4 Integrating XForms

Integrating the XForms UIs into our Petri net life cycle turned out to be a full time job. It required much work in the Editor, in the Converter and in the Petri net engine as well. Displaying forms inside a Work list frame was done by Lutz Gericke [5]. My work in designing XForms has had the purpose to propose a usable interface design for end users of the HPI execution environment and to develop a strategy to suggest the modeler automatically generated forms. How I achieved the first goal is to be described in the following and what the last purpose required is explained in section 3.5.

Extended user interfaces for interaction in XForms

Typical Business Process Management Systems use task attached forms that have the only requirement to give the user an interface for storing data to the backend of the engine. For a system that is to support the two runtime mechanisms for interaction (section 2.2) and re-evaluation (section 2.4), the purpose and, therefore, the features of the forms had to be extended.

The interaction required the extension of many states inside the XForms implementation. From then on, the XForms had the purpose to support user navigation during the typical execution (execution state), the delegation (delegation state), the review (review state) and the multiple delegation (multiple delegation state). Therefore, forms had to be generated automatically with all their states, additional form controls, and bindings and had to be published to the engine. The states of the form are bound to input token values and are mutually exclusive. It is shown only own form at a time. The different states and when they are active (relevant attribute) can be seen in the following:

```xml
<x:bind id="delegationstate"
    nodeset="instance('ui_settings')/delegationstate"
    relevant=".../isDelegated = 'true'
    and .../multipleExecutionFired != 'true'/">
<x:bind id="executionstate"
    nodeset="instance('ui_settings')/executionstate"
    relevant=".../isDelegated != 'true' and .../isReviewed != 'true'
    and .../multipleExecutionFired != 'true'/">
<x:bind id="reviewstate"
    nodeset="instance('ui_settings')/reviewstate"
    relevant=".../isDelegated != 'true' and .../isReviewed = 'true'
    and .../multipleExecutionFired != 'true'/">
<x:bind id="reviewstate"
    nodeset="instance('ui_settings')/multipledelegationstate"
    relevant=".../multipleExecutionFired = 'true'/">
```

We developed another requirement, that a task had to contain just one and only one form we wanted to attach to all the ReceiveTransitions the task life cycle supports:
The layout of model instances can be seen in the following code snippet. The model instance is named inside its id attribute. Model instances exist for the inputtokens, one outputtoken, one ui_instance, and one historytoken for re-evaluation.

```xml
<x:model id="model1">
  <x:instance id="incoming-tokens">...</x:instance>
  <x:instance id="output-token">...</x:instance>
  <x:instance id="ui_settings">...</x:instance>
  <x:instance id="history-token">...</x:instance>
</x:model>
```
The *input-tokens* instance is infused by the engine, when creating the XForms, after the user calls the unique transition’s URL and choosing one of the token combinations of incoming tokens. The input tokens instance consists of all the token values taken from incoming places (a token combination) and has the following structure:

```xml
<x:instance id="incoming-tokens"><root xmlns=""
    <place id="pl_deciding_resource0">
      <token href="/places/925/tokens/392">
        <data>
          <processdata place_id="pl_deciding_resource0" name="resource7" />
          <processdata place_id="pl_ready_resource0" name="EmployerForm">
            <name visible="true" readonly="false" />
            <salary visible="true" readonly="false" />
          </processdata>
        </data>
      </token>
    </place>
    <place id="pl_context_resource0">
      <token href="/places/923/tokens/393" title="">
        <data>
          ... context data...
        </data>
      </token>
    </place>
  </root></x:instance>
```

The output token instance is filled during the initialization of the XForm by copying the input token values into the output token structure (snippet below) and it immediately contains the output token values, that are bound to the form controls. At the XForms initialization, the incoming token’s structure is added a place_id of the place it came from to any metadata and processdata tags, and its complete subtrees are copied to the output token. This mechanism became necessary, because the input tokens could have another structure for each of the XForms states (e.g. for the place attribute *id*) and, therefore, the output model would have had to change. That’s why we could not keep the automatic output model creation inside the *Converter*, as it was used at first. The *id* attribute is now copied at runtime, as you can see in the code below:
The model for the \textit{ui_instance} contains all the states an XForm can be in and all the user interface elements, which are steadily resident in each XForm. The user interface elements, like buttons for submitting, delegating, and canceling the task, as well as the mask for setting delegation details, have special bindings and they are displayed merely in certain form states and are made invisible otherwise.

The different states showed above have each its own particular bindings, which are used for setting the visual character of the current state in which the XForm is situated. A binding determines that a form control is made visible (\textit{relevant} property) or read only (\textit{readonly} property). The four states of an XForm are nodes of the \textit{ui_elements} instance inside the XForms model. The \textit{relevant} binding for the states determines with an XPath expression when a state is active, e.g. the delegation state is active only, when the input context token value \textit{isDelegated} is set to \textit{true}. The XPath expressions have to be chosen carefully. Only one state needs to be active at a time, so, they have to be mutually exclusive.

An example for an intelligent binding is the use of encapsulating groups, that include form controls you want to hide under special circumstances. If you create an \textit{invisibility} binding for a form control and the invisibility is applied, the belonging form control model element would disappear from the output token entirely. This would cause a broken output token and the delegation processing would be violated because of missing values. The solution to this problem was the intelligent adoption of groups, that surround the form control tags inside the form section of the XHTML document. If you want to hide a special form element, you can set the invisibility of its encapsulating group to \textit{true}.

With the firing of button interface elements, bounded output token values can be set, e.g. the firing of the \textit{Delegate} button sets the binding \textit{isDelegated} to \textit{true}. That effects the path taken by the token in the Petri net. In a second button action, the form is submitted by a POST to the bound \textit{form1} header URL.
Now, I would like to present the concrete visual surfaces of the different states a form can have. A form in the *execution state* includes the labels for the item names and the text input to fill in. In this state, you can submit and delegate as well as choose the multiple delegation. As a delegate, you have the ability to see the previous delegator. In the *delegation state* (figure 3.5), an additional drop down box to choose the delegate out of a list of all other potential owners, a task had before allocation, is shown. You have the convenience to send the delegate a message and set a deadline for him that is evaluated by the engine. Also, a review functionality is offered. For setting the visibility right to the delegate, it is desirable to have a button or a clickbox with three states to choose from (see section 2.2). This was an issue that kept us busy until we found a satisfying solution in XForms. Now, a button is implemented that uses three cases. In each button state, there are fired actions setting the actual status in the output model. The initial button state is

Figure 3.5: The XForm for letting the user choose a delegate and visibility rights
writable, so that the delegates can delegate the whole task items without clicking too much around. Our prototype supports the convenience function to set all button states to invisible. If you press the Delegate button, the delegation is submitted. By canceling your form, the decision is discarded (figure 3.5) and the task will be returned to the execution state.

By pressing the Plus button (it is hidden in the form above), the multiple delegation is kicked off and a second token for the next delegate is produced. The multipleDelegationExecution tag is set true and the delegate of the form sees the form interface in its MultipleDelegationState, in which he does not have the possibility to press Interact, but he can submit his decision. So, the only difference inside the multiple delegation state is that you do not have the possibility to delegate the task to a further delegate. The first token is returned to the original delegator by resetting the owner in the token to the original owner in tr_reset_user. So, the delegator can delegate again or submit his last delegator. The advantage of this approach is that the delegates can process their task in parallel, while the delegator still chooses his next delegates. By submitting his last delegate, the delegator token is consumed by tr_consume. All the delegate tokens are, then, merged. The corresponding XSLTs are described by [10]. When you reside in the reevaluation state, you can see exactly the same form as you saw during execution with the changed data during delegation. You have the possibility to submit the final changes or cancel them.

**Extended user interfaces for Re-evaluation in XForms**

We last implemented the re-evaluation mechanism in the HPI execution engine. The re-evaluation made it necessary to extend the user interfaces for the requirements named in section 2.4. We let a new model instance historytoken be infiltrated through an engine mechanism, Philipp Maschke implemented [9]. The history token instance is empty during normal task execution. When the re-evaluation is triggered, history token values are created directly before the execution state of a form. The historytoken instance contains tokens in a parallel structure to the input sequence tokens (code snippet below), that contain the local process data.

```xml
<x:instance id="history-token">
  <historymessage />
  <name /> <salary />
</x:instance>
<x:instance id="input-tokens">
  <processdata>
    <name /> <salary />
  </processdata>
</x:instance>
```

For displaying the changes in the execution state of the form in comparison to the current data values, we had to add a state reevaluation state, form controls for displaying the recent values, and bindings for the historytoken input values.
The *historytoken* also contains a re-evaluation message created by the engine. It is displayed in a red coloring at the top of the document to make the user aware of an exception in data change which just occurred (figure 2.10).

### 3.5 Generating forms and managing data objects

As we have explained in the last section tasks have attached forms for displaying data. The next section describes, how the data items are made known to the process model and how the forms to display data items are build automatically from a data object’s data model property.

#### The life cycle of data objects

The life cycle of the Petri net, known from section 3.3, had to be conceptually clear. Therefore, we decided to implement a rigid separation between persistent execution data (called *metadata*) and the task internal and volatile data (called *processdata*) and created special context places for that purpose, which can be accessed by the runtime for analysis purposes at any time. So, the context data places offer different advantages for the process execution: They build a central access point to the meta data information of a running task and are a more central access to monitor information of finished tasks. Other tasks could access the central context place if a Two Eyes Principle\(^1\) were implemented. Each transition of the task life cycle updates the task’s context with its runtime information.

Additionally, we took a consistent design decision for letting a task be *aware of global data*. This decision followed directly from the requirements of data dependency support, as can be seen in section 2.3. We introduced global data places (*pl\_data* in figure 3.4) in order to support a *global context*. The Petri net life cycle of tasks was changed by implementing data synchronization with the global context data places. Our approach requires that the task’s context has to be modeled explicitly with read and write associations to the data objects. Thus, a write association is equal to a read/write association. The approach of our prototype diminishes any locking mechanisms on data objects. Each task reads the data of incoming data objects inside its *allocate* transition and writes to the data place inside its *finish* transition if it has write access. Therefore, *Dirty Reads* can be possible any time.

Context places and data places are initialized inside the runtime when a case is fired. All the context and data places are made known to the runtime and an empty token is put to them, that gets its information during the enabling of a task. These globally defined places are accessible from the outside to get an overview about tasks and data in the process and can be used from the inside of a running task to see if the task is enabled, who the current owner and the original owner are or what data was submitted of already finished tasks.

\(^1\)Two Eyes Principle: Two tasks forbid the identical executing owners
For the purpose of data enabling, the Converter had to change. Data objects in BPMN had to be transformed into data places one to one. Synchronization arcs to data places had to be introduced as already illustrated in the previous section. Additionally, the data model of data items inside the objects had to be specified, because it is made known to the engine what data items can be interrogated at data places for analysis. The appropriate PNML structure for generated data places can be seen in the code snippet below:

<place id="pl_data_EmployerForm" type="data">
   <toolspecific tool="Petri Net Engine" version="1.0">
      <name>EmployerForm</name>
      <locator>
         <name>name</name>
         <type>xsd:string</type>
         <expr>/processdata/EmployerForm/name</expr>
      </locator>
      <locator>
         <name>salary</name>
         <type>xsd:integer</type>
         <expr>/processdata/EmployerForm/salary</expr>
      </locator>
   </toolspecific>
   <model>
      &lt;data&gt;
         &lt;processdata name="data1"&gt;
            &lt;name/&gt;
            &lt;salary/&gt;
         &lt;/processdata&gt;
      &lt;/data&gt;
   </model>
   ...
</place>

Locators are created by the Converter for each place and are used for accessing the token values inside the places. Places do not have an inherent token model and are simple containers for XML documents representing tokens. Locators are indicators for guard conditions that a token value is available. So the guard conditions and external accessors are guaranteed not to throw an exception by accessing non resident values.

Inside the Oryx Editor you can specify a data model for the properties of an Executable BPMN data object inside the property pane (see figure 3.6). This data model is interpreted in the XML Schema Definition [12] and has to be defined inside the XSD namespace. By creating a data object a schema definition with an empty complex data type is proposed and can be extended by the user. It has to look like this:
Generating XForms inside the Converter

Before conversion to the Petri net structure, a URL is attached to an ExecTask and it refers to its generated form, the belonging bindings, and to its output token model. The output token model always has an empty structure, because it is not used until runtime before it has to be initialized. The ExecConverter inside the Oryx Editor contains methods to create forms and bindings for a task. A form and a bindings URL is attached to all the task’s FormTransitions.

```java
public class FormTransitionImpl
    extends LabeledTransitionImpl
    implements FormTransition {

    protected String modelURL;
    protected String bindingsURL;
    protected String formURL;
}
```
The form generator needs to know the specific data objects that are attached to the task. The task’s attributes are stored inside a Java HashMap behind its data object ID to refer to the output token model, that is created at runtime. First, a form and bindings template is created that contains the invariable form components existent in each form. For each data attribute, its special form controls and bindings are added by the appropriate methods. The code snippet represents the methods for creating templates and individual parts of controls and bindings. The parser, then, converts it to a string representation of the XML document and posts it to the HPI execution engine, whose storing mechanism persists it and returns the URL to the persisted document.

```java
// build form Document template without attributes
Document formDoc = buildFormTemplate(parser);
// adds form fields of necessary attributes
formDoc = addFormFields(formDoc, processdataMap);
// build bindings Document for attributes
Document bindDoc = buildBindingsDocument(parser);
// adds binding attributes for tags
bindDoc = addBindings(bindDoc, processdataMap);

// persist form and bindings and save URL
model = this.postDataToURL(domToString(modelDoc),enginePostURL);
form = this.postDataToURL(domToString(formDoc),enginePostURL);
```

### 3.6 Web Dynpro prototype

We created two proof-of-concept prototypes for getting acquainted with the architectures, wherein we planned to implement the delegation principle. The Web Dynpro application and, later on, an early XForms prototype cleared the way for the subsequent surfaces of delegation and multiple delegation. Both of them avoided the engine’s token passing mechanisms and were used for a single user point-and-click through the single and multiple delegation interfaces. The Web Dynpro prototype was helpful for a proof of the feasibility inside the SAP’s Process Desk, even if we did not touch the engine side at any time.

First of all, a Web Dynpro application was used to test the Web Dynpro component inside a runtime container and outside the SAP Process Server. The component included three views with their respective view controllers, a component model, and a component interface including all the model elements exposed to the Process Server. The ProcessServer has its only central access to the component interface of the Web Dynpro component. Our prototype does not use any complex data types, but only strings for the context mapping. The component model describes the internal prototype context of data elements a task owner can see. The component model interface offers the environment some of these contextual elements.
The views let the user change the view internal data with controls inside the belonging view controllers. Therefore, the views behave exactly like the XForms body, where form controls and formatting tags lay out the XHTML document. The controls are also bound to controller internal data elements. A difference is that the Web Dynpro component binds the view controller data to the form controls itself, whereas the XForms bindings have to be set manually. The component controller interface of Web Dynpro components corresponds to the exposed Petri net data places of our engine. The aim of both of them is to split up a task external context and to offer the data to the outside. The internal token passing of the HPI execution engine equals the component controller data that exist task wide if the Web Dynpro component is understood as the task’s layouting and data storing component. Controllers of Web Dynpro views define the responds to changes in UI elements exactly as the bindings for XForms do. All in all, the two approaches are quite similar, but differ in their data passing mechanisms. Either way, the data is persisted internally and propagated to the outside. Therefore, the Web Dynpro views require a binding to their Web Dynpro component context, which is not the same as XForms bindings for model data to form controls.

In the following, we describe the functionality of the mentioned prototype. The user starts the prototype by an internal application starting point (see figure 3.8), which is set by the Web Dynpro DelegateComponent that we created. The starting point of the Web Dynpro application is the TaskEditView (figure 3.7). Each view has its belonging view context that contains the context variables for the InputFields, their visibility values, their visibility checkbox values and the visibility state of the checkbox values, e.g. the delegate name has the context variables cBNameEditClicked, cBNameEditClickedVisible, cBNameVisibleClicked, and cBNameVisibleClickedVisible. We use the navigation plugs to switch between views (see figure 3.8). The views contain Button elements that trigger certain actions defined in Java code. The Java method stubs are created automatically and we defined (like in the XForm) the transfer of Web Dynpro control values to the context inside the button’s actions. So, the view data is propagated into the view’s controller context. The view controller context elements themselves are bound to the component controller context.
elements with the same names. By pressing the button, an action triggers the next plugged view that is defined by the Web Dynpro navigation. After showing the prototyping work list, the TaskEditView is triggered. Then, the user has to choose if he wants to delegate. If not he gets back to the work list, otherwise he is presented the TaskEditDelegationView. When he chooses the delegate, the delegate work list is triggered and the delegate sees the TaskEditView, so he can delegate himself. After submitting and if a review was demanded, the ReviewView is shown and conducts back to the initial work list. The view navigation presents a complete task life cycle as it exists inside the HPI execution engine.

The Web Dynpro architecture 3.9 differs from XForms in similar points: The Web Dynpro application runs inside the Web Dynpro engine and the form is delivered to the Web Dynpro client to be presented at the client computer. So, the Web Dynpro engine is settled inside the presentation server, an additional layer between back end and client and can have its own context. It regularly transfers data to the back end, namely the Process Server if synchronization becomes necessary. Event dispatching and model management is controlled by the Web Dynpro engine. It completely contradicts the approach of the XForms prototype (without an own runtime), where the form internal model is managed by and stored inside the volatile web browser state. It is directly delivered to the server after submitting the form. Our HPI execution engine itself simulates an XForms runtime by its token passing mechanisms.
3.6. Web Dynpro prototype

The Web Dynpro component was created without regarding the user support of the engine to abstract from the concrete engine mechanisms. A delegate and the actual owner of the task has to be present as an attribute in the external component interface and has to be changed from the external processing engine if the delegation was submitted. This could be done in three steps and looks exactly as it has already been implemented in the HPI execution environment (see [10]):

- The first actual owner had to be set inside the component’s model as the original owner (as it is set by our engine as the output token value of the trAllocate transition). Alternatively, all the owners of a task had to be remembered in a list of delegates.
- The delegation form had to set the chosen delegate in the component’s model, so that the delegate would emerge in the component context of the Web Dynpro component.
- When the delegation form was submitted, the Process Server would have to interrogate the desired delegate, would check its rights for being the actual owner, and would, finally, set him as the actual owner. By this moment the task would have changed its actual owner ad-hoc at runtime.
- After the last delegate had submitted the task, the engine had to reset the actual owner to the preceding delegate or to the original owner, respectively.

For the reason of implementing the delegation inside the Process Server the following architectural interfaces have to change:
• The engine must be capable of changing the actual owner of running tasks at runtime and must have access to the context values of Web Dynpro components to read the chosen delegate and write the actual owner of a task.

• At the enabling of a task, the engine has to know a list of the potential owners of a task and must write it to the delegate list in the Web Dynpro component interface.

• The engine has to throw delegated tasks out of the Process Desk UWL of the delegator. It must put delegated tasks inside the UWL of its delegators.

The advantage of this approach is that the implemented task states of the Process Server would not have to change, because the order of Web Dynpro forms is forced by the Web Dynpro controller and not by token values as it is the case in the HPI execution engine. Therefore, we exploit the Web Dynpro component’s own internal states and do not have to extend too much the engine’s internals.
4. Summary

Building flexibility into Business Process Management systems offers a wide range of possibilities and mechanisms, you can hand over to the user. The guidance of user interfaces turns out to be equally important. The more flexible the processes are, the more guidance a user needs in order to not get lost in manuals as it was the case in the “Anchor approach” of SAP Guided Procedures, a former Business Process Management System. There, the user had the chance to insert additional activities at runtime, manipulating the process model structure. The user should not be forced to have any knowledge in the business modeling domain at all.

With the approach to expose an intuitive mail client interface as a well-known metaphor and a well analyzed guide line, we allow the user to act within a relaxed corset of task management. Within this corset of his inbox, he manages his personally owned tasks, is permitted to be informed about exceptions relevant to his work, and to delegate tasks he might otherwise delay until the task’s deadline. The only additional issue our mailbox approach has to deal with is to inform the user of any information about tasks he once owned or which he is still responsible for. The additional capability of interaction, our system provides, simply appears to the user as a well known forwarding function, nothing new at all for an amateur mail client user with his knowledge of previous Business Process Management systems.

In addition, the user is guided through the application of our mechanisms in an easy-to-understand way. All the features for user interaction were composed in one button click inside the allocated task’s form and give the user a good idea of the additional functionality. The delegation procedure is straight forward for the user when you look at the intuitive visibility buttons and the delegate selection. An additional feature, fitting well into the context of delegation and being informative as well, is the delegation progress bar, which we have not yet implemented.
The implementation of user interfaces we explored in the prototypes of our Bachelor’s project, has demonstrated the substantial advantages of more flexible Business Process Management Systems. The design decisions we made were an approach to implement the prototype’s mechanisms. The delegation could still have been made easier with more intuitive UIs, as can be seen in the first part of this thesis. But, the prototypes proved the feasibility of the mechanisms and made their ease of use visible.
Bibliography


