Enhancing Content Management in DPG

Ana G. Pino

Department of Informatics
University of Bergen
Norway

Long Master thesis
November 2013

Foreword

This document is the result of my master degree studies in the Department of Informatics at the University of Bergen.

I would like to thank my supervisor Khalid A. Murgal for his patience and guidance through this process. I would also like to thank my fellow master students, in particular those in the JAFU office for making this experience a good one and always being willing to provide help.

To my friends and family my warm gratitude for their continued support and understanding. Special thanks to Louise Iden and Line Eeg-Larsen for being my family away from home.

Ana G. Pino
Bergen, 19. November 2013

Contents

List of Figures

List of Tables

Listings

Definitions

ADO:	Abstract Data Object
ADV:	Abstract Data Views
AJAJ:	Asynchronous JavaScript and Jason
AJAX:	Asynchronous JavaScript and XML
API:	Application Programming Interface
CMS:	Content Management System
CSS:	Cascading Style Sheets
CRUD:	Create, Read, Update, Delete
DAO:	Data Access Object
DOM:	Document Object Model
DPG:	Dynamic Presentation Generator
HTML:	HyperText Markup Language
HTTP:	Hypertext Transfer Protocol
HP:	Hewlett-Packard
JAFU:	Java i Fjernundervisningen
JPA:	Java Persistence API
LSM:	Learning Management System
PCE:	Presentation Content Editor
POJO:	Plain Old Java Object
PM:	Presentation Manager
PV:	Presentation Viewer
RAT:	Repository Administration Tool
SCA:	Static Code Analyser
SPA:	Single Page Application)
TA:	Teacher Assistant
UI:	User Interface
UML:	Unified Modelling Language
URI:	Uniform Resource Identifier
URL:	Uniform Resource Locator
XSLT:	Extensible Stylesheet Language Transformations
XHTML:	Extensible HyperText Markup Language
WYSIWYG:	What you see is what you get
XML:	Extensible Markup Language

ZAP:	Zed Attack Proxy

1 Introduction and Background

Java i Fjernundervisningen (JAFU) a project within the Department of Informatics at the University of Bergen (UiB) with the objective of providing distant learning courses to students that cannot attend classes at the university. The project has been active since 1999 and currently offers two courses on Java programming each Spring semester, INF-100F and INF-101F. Since the beginning, several tools have been used and developed to fulfil this objective. The system currently in use is *Dynamic Presentation Generator* (DPG).

Its first version was developed by Yngve Espelid in 2004 but in 2008 after evaluating the system, Karianne Berg [7], Bjorn Ove Ingvaldsen [23] and Bjorn Christian Sebak [52] made the decision of reimplementing the system from the ground up; this led to version 2.0. DPG has continued to be extended and improved since then by several students through their master theses [53] [56], focusing mostly on making the system more robust, improving the plugin architecture and adding functionalities. However, since its development, little attention has been given to the user interface and ease of use.

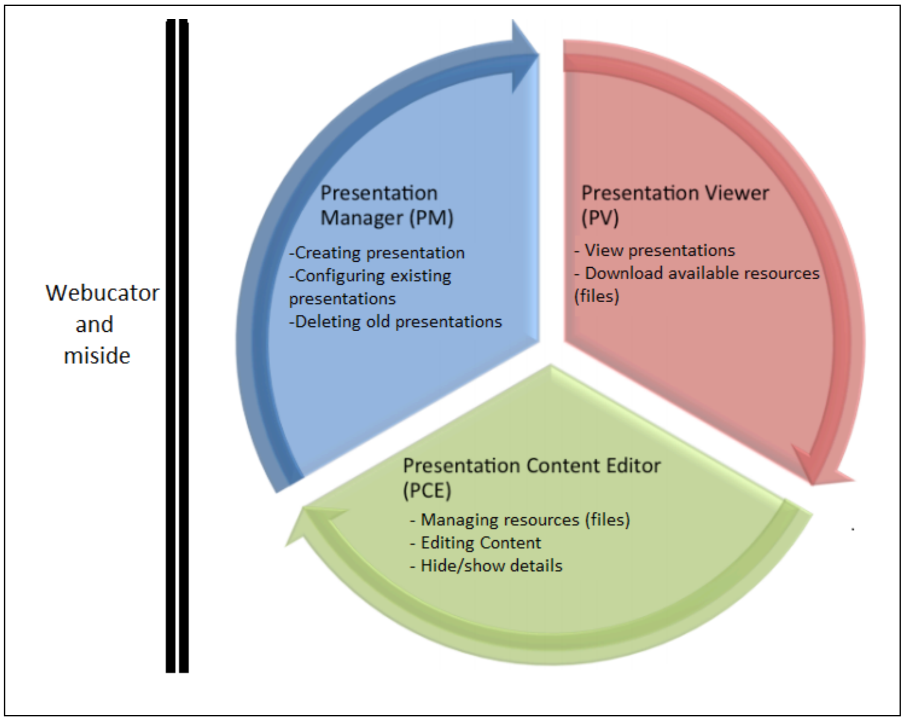


Figure 1.1: DPG subsystems based on illustration in [52]

1.1 Dynamic Presentation Generator (DPG)

DPG is a *Content Management System* (CMS) that allows the creation, modification, general maintenance and management of web content that can be presented to users that have access to the system. DPG consists of four subsystems:

- **Lobby** It is the entry point to DPG and handles authentication and authorization of the users, currently this can be done from Mi Side [38] or through Webucator [33]
- **Presentation Viewer (PV)** Renders the content of presentations and displays it to the user.
- **Presentation Content Editor (PCE)** It allows the user to create, edit or delete content from presentations.
- **Presentation Manager (PM)** Provides the functionality of creating, configuring and deleting presentations.

The system defines three different roles that have increasing access to the different subsystems of the DPG.

- **Reader** It can view the content of presentations it is authorized to see. Access to PV.
- **Publisher** It can view and modify content of the presentation that it publishes. Access to PV and PCE.
- **Administrator** It can view and modify the content of all presentation and create, configure and delete presentations. Access to PV, PCE and PM.

1.2 Presentation Pattern and Presentations

DPG was developed as an implementation of the concept of *Presentation Patterns* [37] as first defined by Khalid A. Mughal in 2003. The main idea of presentation patterns is to allow the separation of content and presentation so that they can be reused without depending on the other.

1.2.1 Presentation Pattern

A presentation pattern (from now on *pattern*) describes the data structures that will be used in the *presentations* based on it, and how the data will be organized. These elements are defined using the components outlined by the *Presentation Pattern Specification*. In figure 1.2 on the following page is possible to see the different components and how they relate to each other.

Type, *field* and *entity* are used to define data structures. For example in listing 1.1 it is possible to see the definition of `messageEntity` in `pattern.xml`. The `messageEntity` is composed of two fields, `title` and `content`, and each field has a type, `string` and `xhtml` respectively, that defines the values the field can take when the entity is instantiated.

Listing 1.1: messageEntity in pattern.xml.

```xml
<entity id="messageEntity">
    <field type="string" required="true">title</field>
    <field type="xhtml">content</field>
</entity>
```

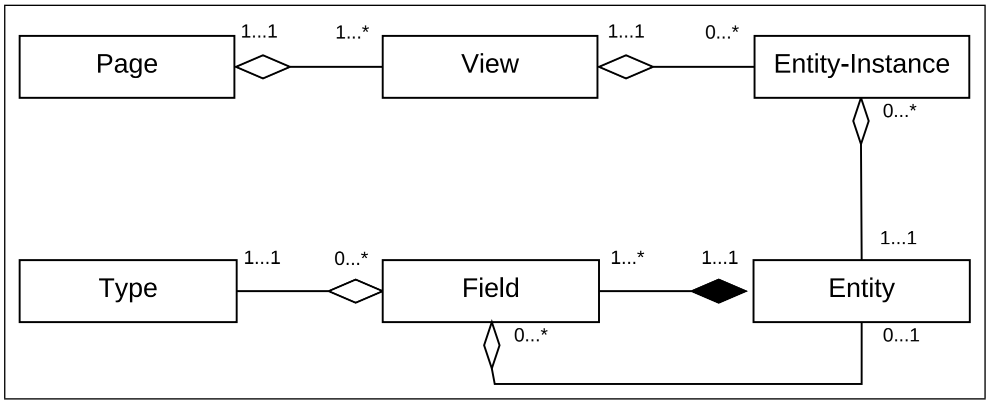


Figure 1.2: Components defined by the Presentation Pattern Specification and their relationships

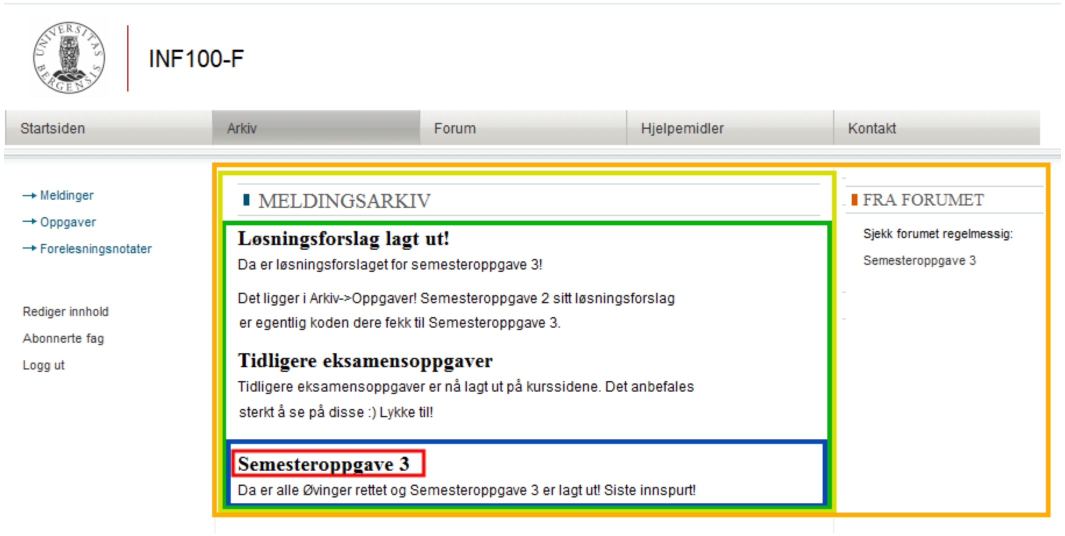
There are two special *types* called *subentity* and *list*, that when instantiated can take the value of one or more entity instances. In these cases the field also declares the entity that will be used as the data structure for the instances. An example of this case can be seen in listing 1.2 on line 7.

Listing 1.2: contactEntity and contactListEntity in pattern.xml.

```
1 <entity id="contactEntity">
2     <field type="string" required="true">name</field>
3     <field type="string">email</field>
4 </entity>
5
6 <entity id="contactListEntity">
7     <field type="list" entity-id="contactEntity">contactList</
          field>
8 </entity>
```

The componentsentity-instance, *view* and *page* describe how to organize the data to present it to the user. *Entity-instance*, as the name implies, declares an instance of an entity and gives it an id. It is these instances that will be used to map the data structure given by the entity to the content. *Views* connect an entity-instance with a transformation file that is in charge of transforming the content of the instance into a format that can be presented to the user.

In the case of DPG the content is persisted in XML files and the transformations are XSLT files that transform the XML into XHTML to be presented as web content. Views give the system the possibility of presenting the same information using different formats since the same entity-instance can be mapped to several views, allowing for example to only display the title of a `messageEntity` in one view while



Løsningsforslag lagt ut!
Da er løsningsforslaget for semesteroppgave 3!

Det ligger i Arkiv->Oppgaver! Semesteroppgave 2 sitt løsningsforslag
er egentlig koden dere fekk til Semesteroppgave 3.

Tidligere eksamensoppgaver
Tidligere eksamensoppgaver er nå lagt ut på kurssidene. Det anbefales
sterkt å se på disse :) Lykke til!

Semesteroppgave 3
Da er alle Øvinger rettet og Semesteroppgave 3 er lagt ut! Siste innspurt!

Sjekk forumet regelmessig:
Semesteroppgave 3

Figure 1.3: A page component seen in PV. Red marks a title field. Blue marks a messageEntity. Green marks the listMessageEntity. Yellow marks a view. Orange marks the page.

presenting all the data in another.

Views are not presented directly to the user, they are organized in pages. *Pages* can contain one or more views and the views can be in one or more pages. This allows for the same views to be shown together with different data in different pages.

1.2.2 Presentation

A *presentation* is an instantiation of a pattern and uses the pattern to structure and organize its content. Several presentations can be created from the same pattern. For example the *course pattern* is currently used to provide the structure for INF100F and INF101F presentations each semester.

Figure 1.3 shows how all the components come together in DPG. It is a screen shot of the page `allMessages` from the presentation INF100F based on the course pattern.

1.3 Motivation

Since its first version, DPG has gone through several modifications that have made it a more dynamic tool for presenting and reusing content and structures increasing in complexity and functionality. However little attention has been given to the methods used to create presentation content and how to make the job of adding content more intuitive and less time consuming for the publisher user since the development of version 2.0.

This aspect is one of the success factors of any CMS since their main functionality is to facilitate content management, and the first step in the management is allowing the user to enter the content into the system without adding complexity that takes away attention from the content itself.

1.4 Objectives

The main purpose of this thesis is to improve content management capabilities in DPG and in particular that of the PCE so that it becomes a more reliable tool. For this reason the following goals were established:

- Analyse current use of PCE
- Propose solutions to the issues found in the analysis
- Implement proposed solutions
- Evaluate the implemented solutions

The aim is to make the DPG a flexible system for content management, that is user-friendly for non-technical users.

2 Presentation Content Editor (PCE)

2.1 Introduction

This Chapter will explain the current functionality, structure and user workflows of the Presentation Content Editor (PCE), paying special attention to usability and security aspects.

PCE is in charge of maintenance and management of presentations in DPG. It is used mostly by publisher users, and when necessary, by administrator users.

The current version of PCE was first developed by Bjorn Christian Sebak [52] in 2008 as an improvement of the existing system at the time called Repository Administration Tool (RAT) developed in 2004 by Yngve Espelid [11]. The main goals were to create an editing tool that could be used without having knowledge of the underlying technologies and improve usability.

Since 2008 several other students have worked with DPG for their master thesis, but none of them worked on PCE directly, therefore, the changes this subsystem has sustained since its creation are minimal.

2.2 PCE use context

ISO 9241-11 [24] defines the use context of a work system as the description of the conditions where the system is to be used. The conditions include the set of users, tasks, equipment and environment where the systems are to be used. For the purpose of this analysis the rest of DPG will be considered as static; meaning that the structure of the patterns and the functionalities of the system will remain the same, only how PCE deals with these elements and how they are presented to the user will be analysed.

2.2.1 Tasks

Task describe the actions that can be taken within the system to reach an objective. The objective will be used as the name of the task. A list of all the tasks in PCE can be seen in Table 2.1.

The tasks in PCE can be divided into two categories. *Pattern Management* and *Content Management*.

2.2.1.1 Content Management

Entity-instance selection As it was explained in section 1.2, DPG divides the content in different levels (Presentation, Page, View, Entity-Instance, Field), the content can not be modified until the user gets to the Entity-Instance level that actually contains the data. The other levels contain information about how that data is organized for the viewer user. This task allows the user to navigate to the Entity-Instance level so that it can proceed with other tasks.

Content CRUD - Allows the user to create, read, update and delete content by managing the data in the different existing entity-instances, adding entities-instances to the existing ones and deleting the ones that are not needed.

2.2.1.2 Pattern management

This category of actions relate to changing the default settings of the pattern.

Disable Page/View: Hides the element from the viewer user in PV.

Enable Page/View: Shows the element to the viewer user in PV.

Name	Frequency of Use	Dependencies
Content selection	Twice a week	None
Content creation	Twice a week	Content Selection
Content modification	Twice a week	Content Selection
Content deletion	Indeterminate	Content Selection
Enable Page/View	Once per semester	Content Selection
Disable Page/View	Once per semester	Content Selection
Change Page/View name	Once per semester	Content Selection

Table 2.1: PCE Tasks

Change name of Page/View: Changes the label of the element to the user.

2.2.2 Users

In this section the characteristics of the different users of PCE will be described so that they can be taken into consideration during the rest of the analysis.

2.2.2.1 Primary User

Teacher Assistants: This PCE user has knowledge of the content to be introduced to the system but not necessary of the system itself. They tend to use the system during one semester, the time that their assistantship lasts. The majority has had previous contact with at least one LMS or CMS before. Example of these are Mi side [38] and It's learning [25] since they are used at the University of Bergen and at College of Bergen, respectively. In DPG they are assigned the publisher role.

2.2.2.2 Secondary Users

System Administrator: They have knowledge of how the system works and its workflows but no direct knowledge of the content. They tend to do the tasks regarding pattern adjustments at the start of the semester to adapt the new presentation to the desires of the publisher running the course. In DPG they are assigned the administrator role and as such can modify any presentation.

Lecturer: Occasional user with knowledge of the content but not necessarily of the system. They use the system for as long as they have courses that use the platform. In practice the task of maintaining the DPG is delegated to the TAs so the lecturer rarely has contact with the system. In DPG they are assigned the publisher role.

2.2.3 Equipment

As part of a web application, PCE requires as minimum a computer that has an Internet connection and the capabilities to support a compatible browser. The client application of PCE, and DPG in general, are currently based on JQuery [28]. Browser compatibility can be checked in the jQuery Browser Support webpage [27].

2.3 Workflows in PCE

The work of the publisher as such is mostly limited to PCE where the necessary tools to create, improve and delete content are located.

Most of the work flows for the different scenarios will be described as starting from the `listContent.html` webpage after the user has archived a successful login and entered in PCE. The details of the use case describing the procedure the user needs to follow to get to the `listContent.html` webpage can be seen in Table 2.2.

The UML activity diagrams [14] used to represent user interface navigation workflows are based on the ones described in [31] with the addition of some stereotypes that make it easier to identify at first glance exactly what type of user interface they represent. An example of an added stereotype is `<<js:alert>>` that represents a Javascript alert window.

2.3.1 Content management

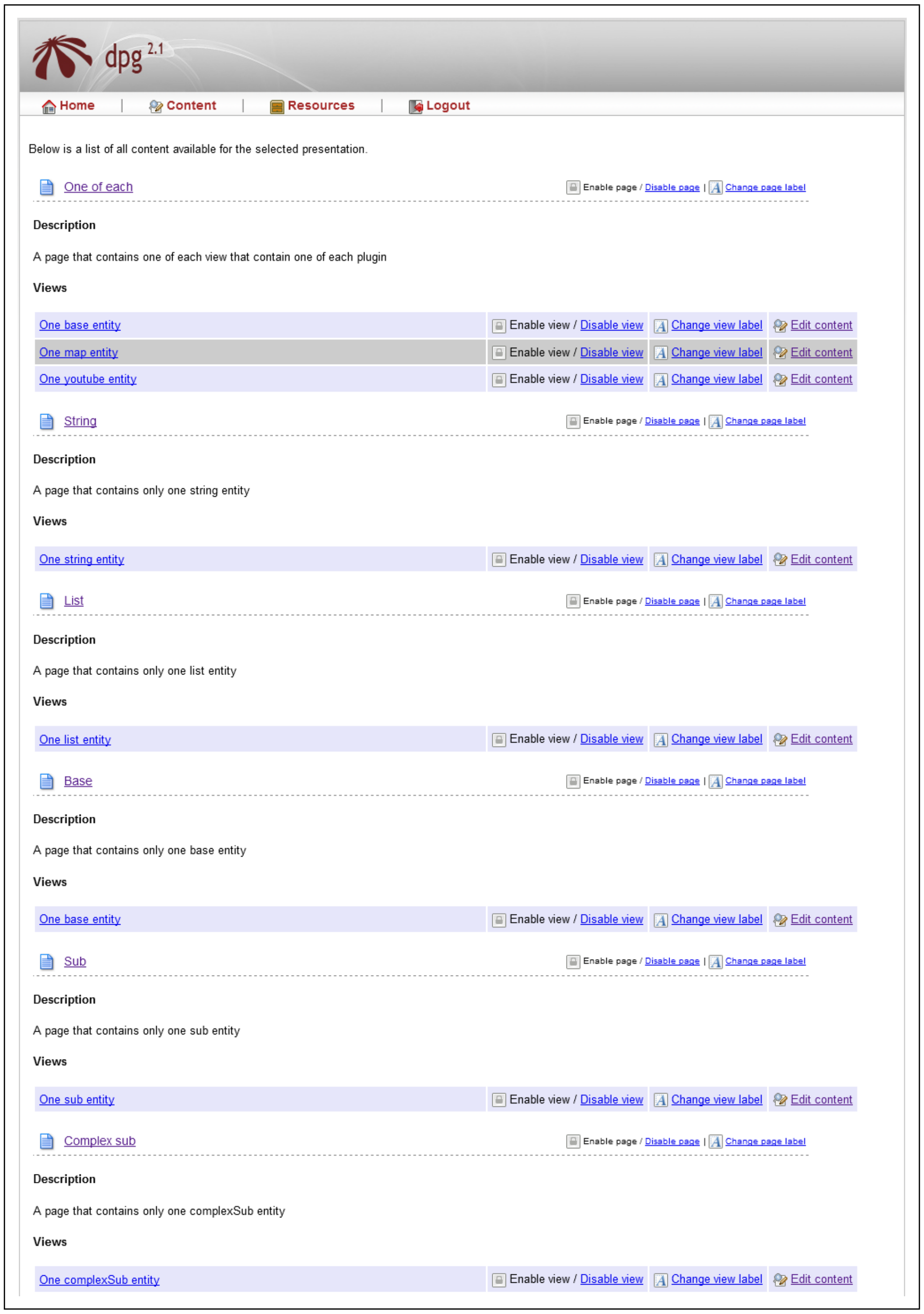
The content of each presentation is updated by modifying the entity-instances in the presentation. The scenarios in this category include the tasks *Content Selection* and *CRUD of Content* as described in 2.2.1. The activity diagram in Figure 2.5 shows the workflows for these tasks.

2.3.1.1 Content Selection

1. System: Displays the webpage `pce/listContent.html` (Figure 2.1) that contains a list of all the *pages* with all the *views* contained in the presentation.

Use Case	Access PCE
Main Scenario	
User	**System**
1. User directs browser to navigate to DPG	
	2. System displays */lobby/login.html*
3. User enters correct credentials	
	4. System displays *lobby/presentations.html* with a list of all presentations of the user
5. User selects the presentation they want to modify and clicks link "Edit Content"	
	6. System displays *pce/listContent.html* with a list of all pages and views of the selected presentation.
Alternative Scenarios	
3a. The credentials were invalid. 3a1. System displays an error message in */lobby/login.html.* 3b2. User enters right credential.	

Table 2.2: Use Case - User login and entering PCE



Below is a list of all content available for the selected presentation.

One of each — Enable page / Disable page | Change page label

Description

A page that contains one of each view that contain one of each plugin

Views

One base entity	Enable view / Disable view	Change view label	Edit content
One map entity	Enable view / Disable view	Change view label	Edit content
One youtube entity	Enable view / Disable view	Change view label	Edit content

String — Enable page / Disable page | Change page label

Description

A page that contains only one string entity

Views

One string entity	Enable view / Disable view	Change view label	Edit content

List — Enable page / Disable page | Change page label

Description

A page that contains only one list entity

Views

One list entity	Enable view / Disable view	Change view label	Edit content

Base — Enable page / Disable page | Change page label

Description

A page that contains only one base entity

Views

One base entity	Enable view / Disable view	Change view label	Edit content

Sub — Enable page / Disable page | Change page label

Description

A page that contains only one sub entity

Views

One sub entity	Enable view / Disable view	Change view label	Edit content

Complex sub — Enable page / Disable page | Change page label

Description

A page that contains only one complexSub entity

Views

One complexSub entity	Enable view / Disable view	Change view label	Edit content

Figure 2.1: Screenshot of the listContent.html webpage.

2. User: Locates a *page* that contains a *view* that uses the *entity-instance* that it is interested in modifying and click on the `Edit Content` link next to the name of the *view*.

3. System: Displays the `pce/content/viewDetails.html` (Figure 2.2)webpage which contains a list of all the *entities instances* in the view, its fields and its sub-entities.

4. User: Scrolls to find the section containing the type of instance it wants to work with and then click on one of the three possibilities that correspond to the desired task (add new, edit or delete).

2.3.1.2 Create and Update

1. System: Displays `pce/content/editContentForm.html` (Figure 2.3) which shows a form with a form element for each field in the entity-instance. In update use case they will appear already containing the information currently in the system.

2. User: Enters the new data and clicks on the `Submit` button.

3. System: Validates data

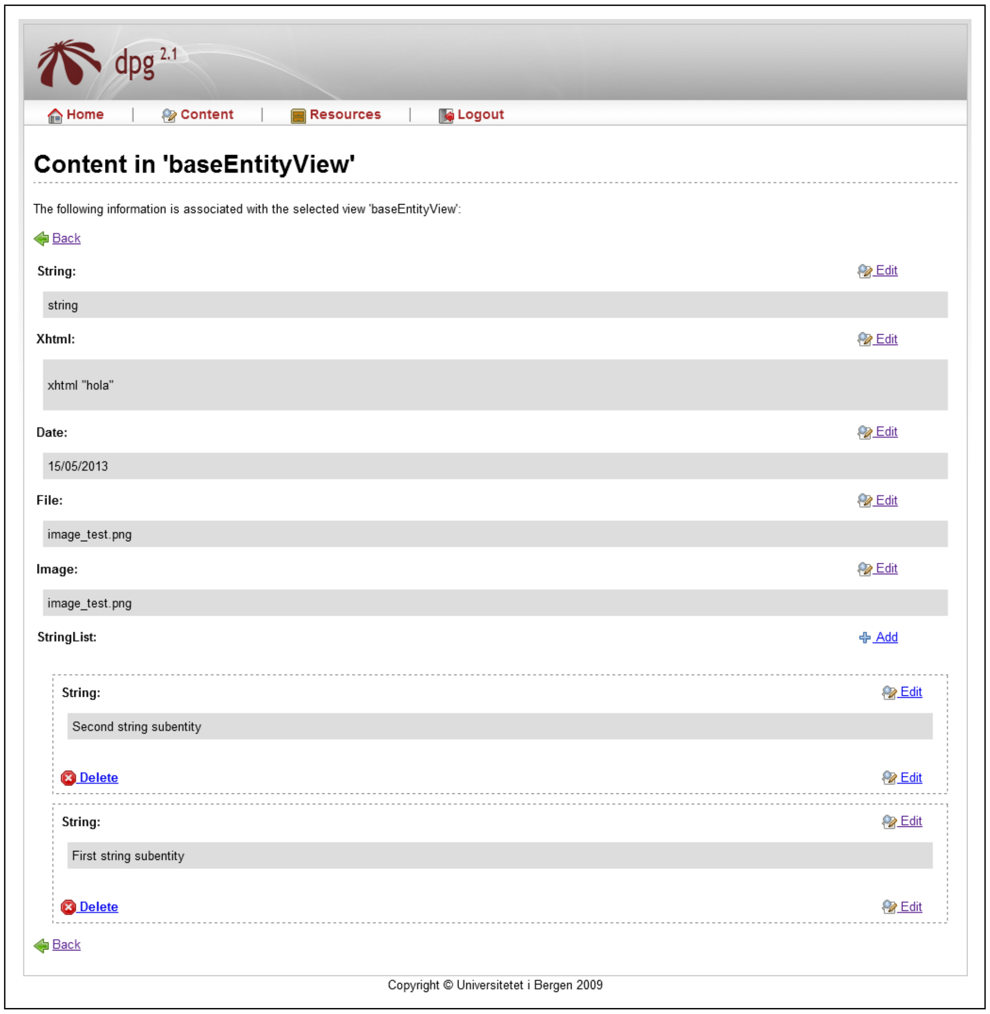
   (a) If the data is correct system: Displays the `pce/content/editContentSubmit .html` webpage.

2.3.1.3 Delete

1. System: Prompts the user with a Javascript alert asking whether it is sure that it wants to complete the deletion.

2. User: Answers the prompt.

   (a) If answer is affirmative: System displays `pce/content/deleteContent .html`. (`pce/content/deleteContent.html` and `pce/content/ editContentSubmit.html` currently have the same format). They provide links for the user to go back to PCE or to go to PV as seen in Figure 2.4)



Content in 'baseEntityView'

The following information is associated with the selected view 'baseEntityView':

Back

String: Edit

string

Xhtml: Edit

xhtml "hola"

Date: Edit

15/05/2013

File: Edit

image_test.png

Image: Edit

image_test.png

StringList: Add

String: Edit

Second string subentity

Delete Edit

String: Edit

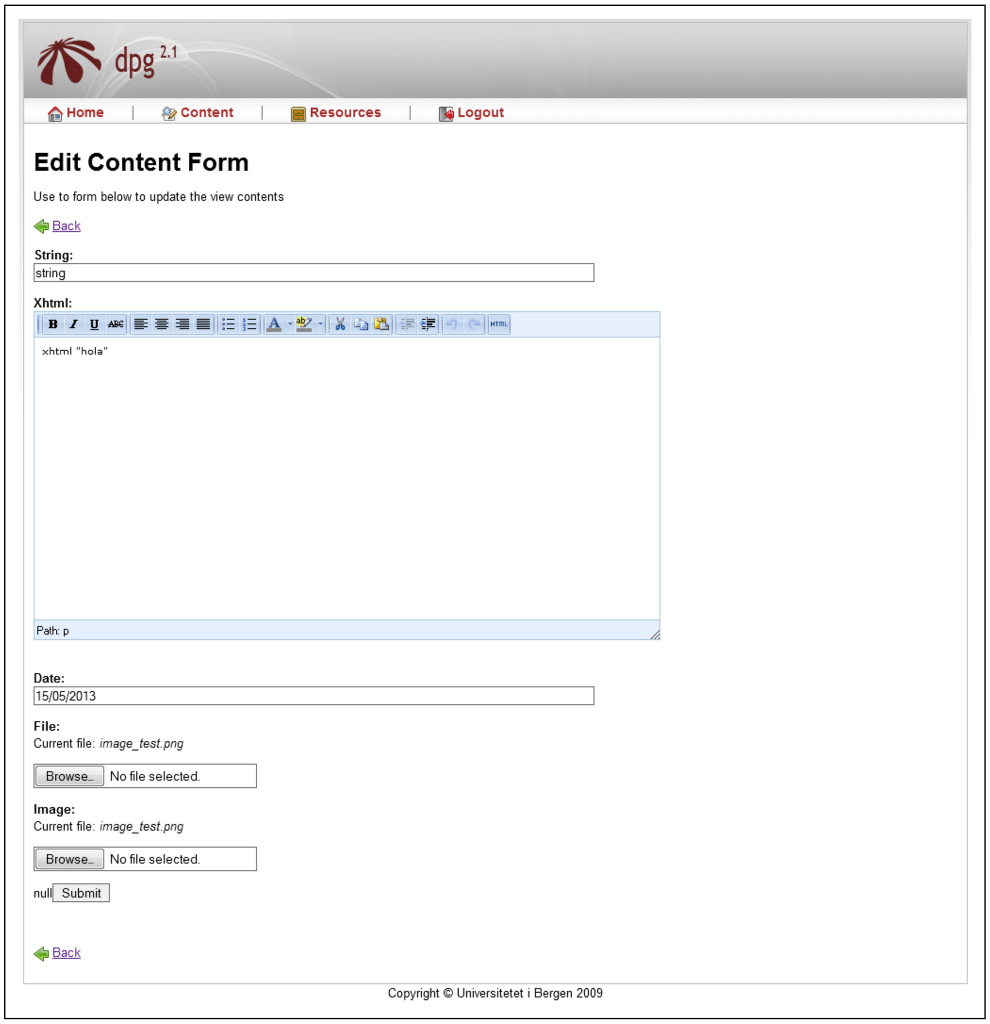
First string subentity

Delete Edit

Back

Copyright © Universitetet i Bergen 2009

Figure 2.2: Screenshot of the viewDetails.html webpage.



Edit Content Form

Use to form below to update the view contents

Back

String:
string

Xhtml:
xhtml "hola"

Path: p

Date:
15/05/2013

File:
Current file: *image_test.png*

Browse... No file selected.

Image:
Current file: *image_test.png*

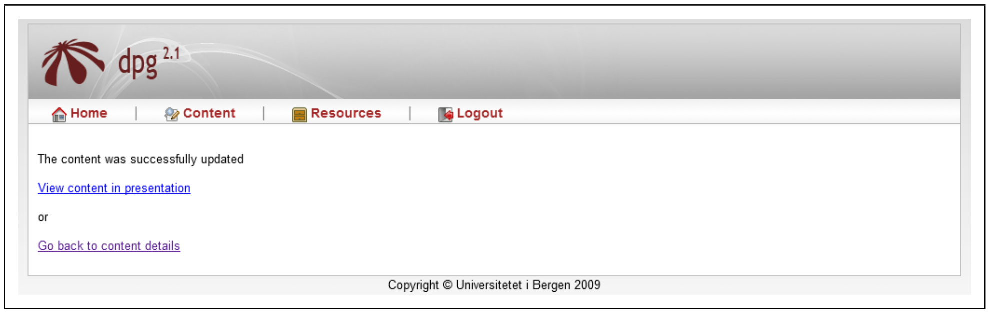
Browse... No file selected.

null Submit

Back

Copyright © Universitetet i Bergen 2009

Figure 2.3: Screenshot of the editContentForm.html webpage.



The content was successfully updated

View content in presentation

or

Go back to content details

Copyright © Universitetet i Bergen 2009

Figure 2.4: Screenshot of the editContentSubmit.html webpage.

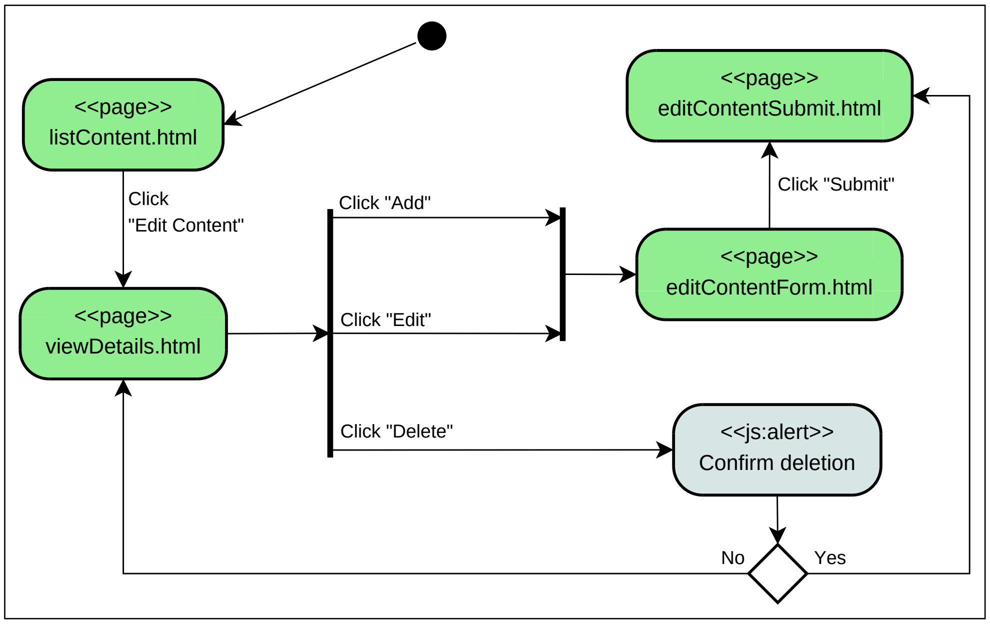


Figure 2.5: Activity diagram describing the workflow of managing content in DPG

2.3.2 Pattern management

Since the workflows for the actions are the same whether they are applied to a *page* or a *view* to avoid duplication the word *element* will be used to represent the mentioned pattern components. this means that for the next two subsections the word element can be replaced by either view or page. The activity diagram in figure 2.6 on the next page shows the workflows for these tasks.

2.3.2.1 Disable and enable Views and Pages

1. User: Clicks on the link for the respective action in `listConent.html`.
2. System: Presents a javascript alert window asking for confirmation of the action.
3. User: Replies to the promt.
4. System: Returns to `listContent.html`.
   (a) If the confirmation was given: System executes the action and a message is displayed for a couple of seconds, stating the new status of the element and in the list the element appears as disabled or enabled.

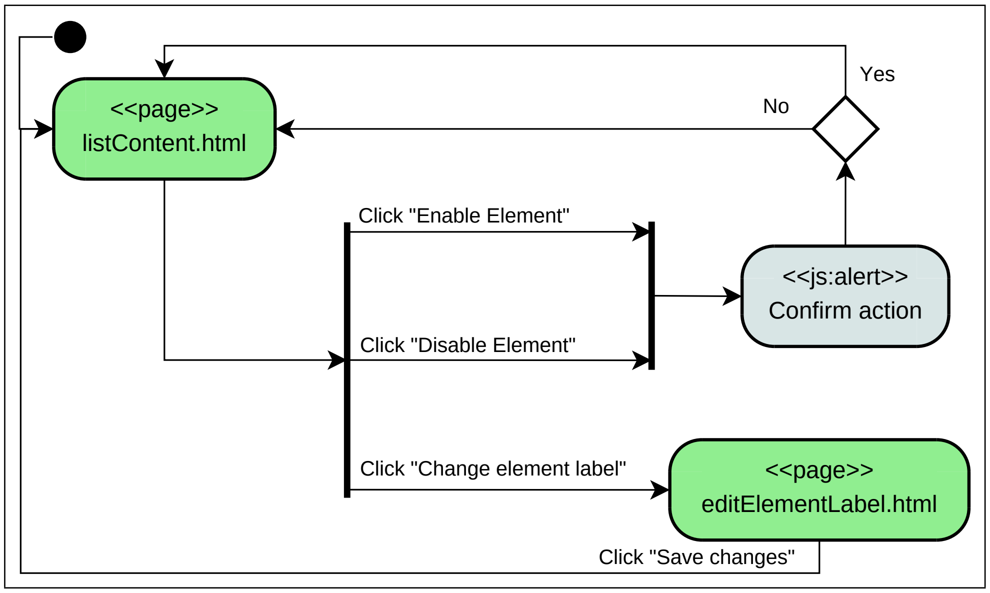


Figure 2.6: Activity diagram describing the workflow of pattern management in DPG

2.3.2.2 Editing names of Views and Pages

The label of elements can be changed in the presentation without affecting the name given by the pattern. This can be done from `listContent.html`.

1. User: Clicks on the link with the text ”Change element label”.
2. System: Displays `editElementLabel.html` with an HTML form with an input box with the current label of the element.
3. User: Enters the new label and click on the ”Save changes” button.
4. System: Displays `listContent.html` where a message will be displayed stating that the label has been changed and the new label will be shown in the list of pages and views.

2.4 Current Design and implementation

The components of the design and implementation of PCE can be subdivided in three levels of abstraction from raw data to the HTML elements that are presented

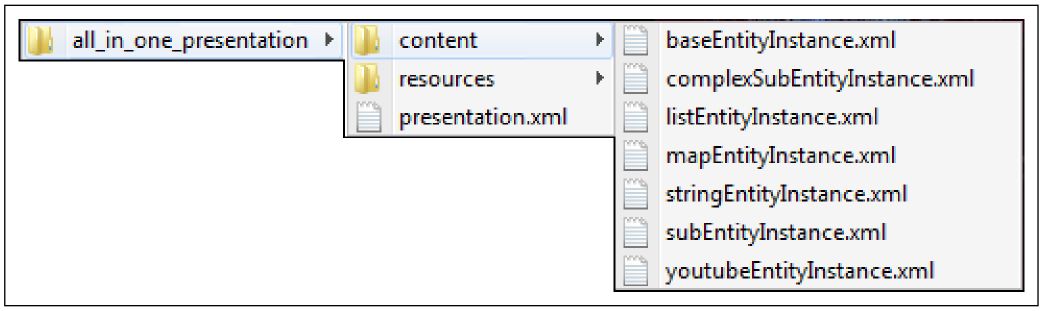


Figure 2.7: Files generated by DPG for the all-in-one presentation

to the user. The first level deals with how the data is stored by the system. The second level represents how the system interprets the data and the last level handles how the data will be presented in the user interface so that the user can interact with it.

2.4.1 Persistence

The majority of raw content data in DPG is stored in XML files, the exception are resource files defined in the pattern and the plugins. For example, files that can be made available to the user for direct download.

Figure 2.7 shows the file structure created by DPG to store the content data. The `presentation.xml` file contains the information regarding pattern managements, a list of the views and pages with their label and enabled attributes.

The files in the `content` folder hold the content data. Each file represents the data of one *entity-instance*. In Figure 2.8 shows the structure of a content file. While the components in presentations share the names with the pattern specification ones presented in section 1.2, they are not the same and they do not relate to each other in the same manner.

Each *entity-instance* can contain *fields* which have *type*, *value* and an attribute that express whether the field is *required*. Fields can also contain entity-instances when the type in the pattern was defined as *subentity* or *list*. There is also another case, when the plugin that is used by DPG to interpret the type returns its own structure, this last case will be farther discussed in section 2.4.2.

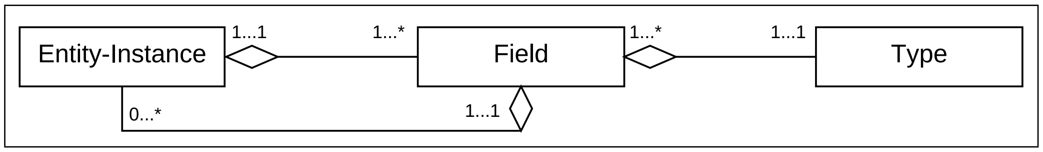


Figure 2.8: Diagram of component structure in a view persistence file

2.4.2 Plugins

In DPG all field *types* are handled by plugins. The plugin architecture has been the focus of several theses, most recently [56] in 2011. This architecture is based on the plugin pattern by David Rice and Matt Foemmel and described in [13] by Martin Fowler. It was first implemented by Bjorn Ove Ingvaldsen [23] and it was selected due to its flexibility. It allows to update existing plugins or add new ones without the necessity of recompiling the main application.

As dictated by the plugin pattern, an interface defines the behaviours that will be different in each implementation of the plugin. In the case of DPG this interface is called `FieldPlugin` and it defines seven methods that need to be implemented.

- `generateElement`: Returns a JDOM [26] element containing the DOM elements that will be presented in PV.
- `getParameters`: Returns a list of the parameters that can be used to configure the plugin in the file `pluginConfig.xml` of the pattern.
- `getFormElement`: Returns the `FormElement` that will be presented in PCE.
- `getXmlContent`: Returns the content of the plugin in XML based on the input `FormElement`.
- `setPluginresourceDao`: Used by `FieldPluginManager` to give the plugin access to resources DAO.
- `setPluginResourceJpaDao`: Used by `FieldPluginManager` to give the plugin access to resource JPA DAO [39].
- `generatePatternStructure`: In the case that the plugin requires more than one field this method returns a list of JDOM elements defining the structure of fields.

DPG also provides a standard implementation of this interface `AbstractFieldPlugin` that most plugins extend.

2.4.3 FormElements

As it was explained in the previous section, each plugin returns a JDOM element for their presentation in PV. This element is then transformed to HTML using XSLT [63].

In PCE, the two webpages that present content employ very different methods. The `viewDetails.html` page uses the XML content files and the `PceTransformationUtil -view.txt` XSLT file to display the content. The plugin system is not involved in the process and as a result the content is presented as raw data, without interpretation.

The `editContent.html` webpage on the other hand uses the `FormBuilder` class to create a form according to the structure provided by each plugin. The sequence diagram in figure 2.9 shows the interaction between the `EditContentFormController` class, controller of the `editContent.html` webpage, the `FormBuilder` class, in charge of creating the form, and the `FieldPlugin` interface that is implemented by all plugins. The main steps are as follows.

1. `EditContentFormController` request the new form from the `FormBuilder` class by calling the `createForm` method with several parameters amongst them the XPath [61] of the *entity instance*. `FormBuilder` then calls the `resolveEntity` method of the `EntityPathResolver` class which returns the *entity instance*

2. `FormBuilder` calls the `generatePatternStructure` method in the corresponding `FieldPlugin` implementing class according to the *entity instance*. This method can return `null`, in which case then the system infers that the plugin uses the default pattern structure, the current default structures is one text field. It can also return a list of JDOM elements each representing one field of the structure needed by the *entity instance*.

3. For each of the fields required by the *entity instance*, the `FormBuilder` class executes the `handlePluginEntity` method. This methods requests the plugin for the field to return the `FormElement` needed to represent it on a form.

4. `FormBuilder` adds all the form elements into an instance of the `Form` class and returns it to `EditContentFormController` that transforms it into an XHTML form ready to be displayed to the user.

The `FormElement` class extends the abstract class `XmlConvertible` that as the name indicates, provides it with a method that converts the element into an XML node. The node is later used in combination with an XSLT to display the field as an HTML form element.

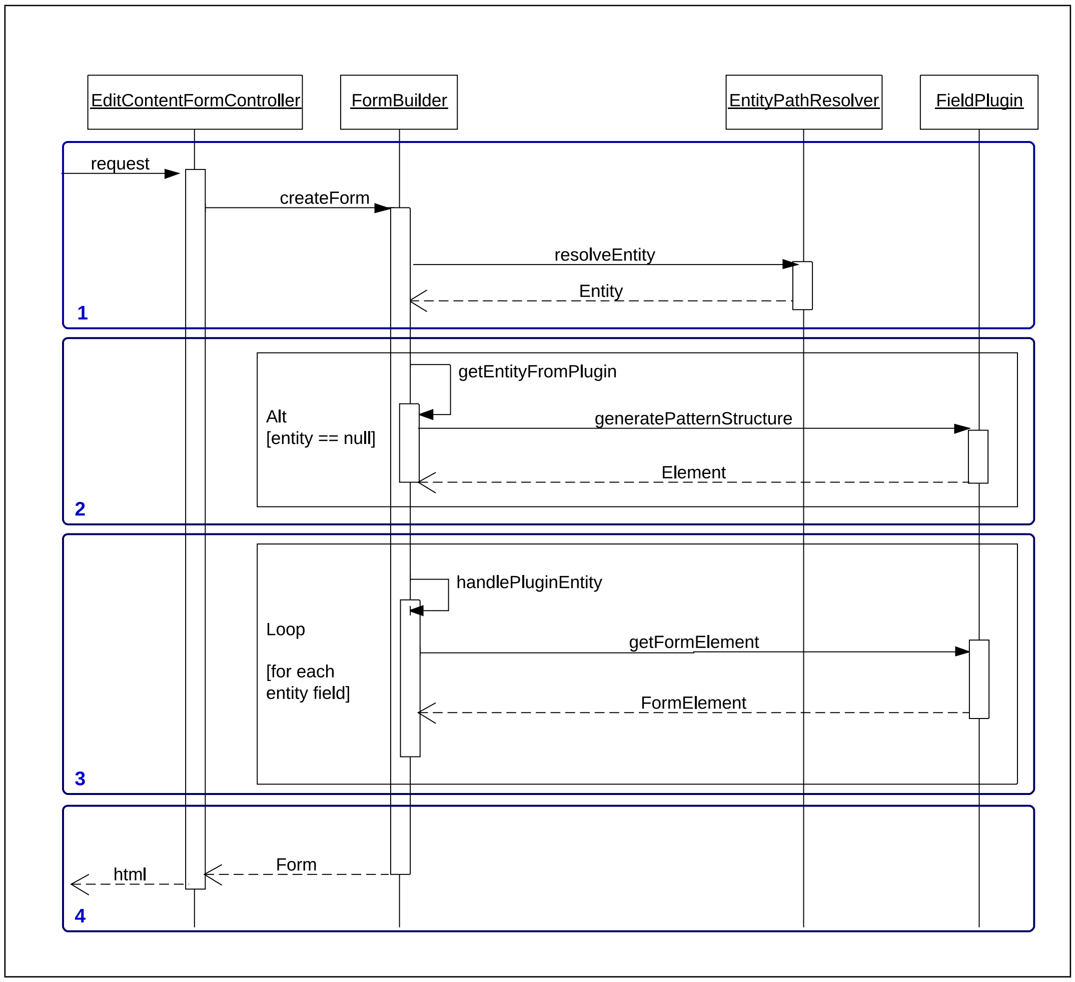


Figure 2.9: Sequence diagram describing the creation of forms in PCE for editContent.html. Image based in the sequence diagram found in [56]

3
Analysis of PCE

The previous chapters have established the structure and purpose of DPG and more specifically of PCE. This chapters analyses the current use of PCE and its usability from the point of view of the publisher.

3.1 Data Gathering and Analysis

The usability of the system was evaluated in two different ways. The first one was feedback from current publisher users through unstructured interviews with open questions. There are four users that have worked with PCE and they were consulted regarding their experiences. The second method was testing by the candidate using two patterns. First the pattern that is currently in use, the Course pattern. This pattern contains all the basic plug-ins and has been in use in production for several semesters. The second pattern was created for the purpose of testing some of the plug-ins that are not contained in the Course pattern and is called All-in-one Pattern.

Usability Aspect	**Importance in PCE**
Navigability	High Importance
Effectiveness	High Importance
Credibility	High Importance
Understandability	High Importance
Learnability	Moderate Importance
Accessibility	Low Importance
Customization	Low Importance

Table 3.1: Usability aspects as mentioned by [16] and their perceived relevance in PCE.

3.2 Usability

The focus of usability is the ease of use of a system according to its target audience. It is an abstract concept that can not be measured directly and it is therefore usually divided in different components that can be more easily observed. The components differ depending on the source and there are several studies and recommendations including ISO 9241-11 [24] that describe possible frameworks to analyse the usability of user-system interfaces. For the purpose of analysing PCE we will use the aspects mentioned in [16] taking into consideration that according to the ISO recommendation each system has to be evaluated according to the context in which it is meant to be used. A summary of the importance of the different usability aspects for PCE can be found in Table 3.1.

3.2.1 Usability aspects

According to [16] there are seven usability aspects that need to be taken into consideration when refactoring web applications for usability. They were analysed taking into consideration the use context described in the previous chapter. What follows is a brief descriptions of these aspects and their importance for PCE and the scope of this thesis.

3.2.1.1 High Importance

- **Effectiveness**: is the degree to which the system provides the necessary tools for experienced users to accelerate the process towards archiving the desired effect. In the case of PCE steps that the publisher needs to follow are always the same and once learned should not present any impediments for the user to repeat them in an expedite fashion. In PCE it is the content that is important, not the process itself, for this reason effectiveness is one of the most important usability factors.

- **Navigability:** describes the organization of the web application and the extent to which the user is presented with, and understands the use of, links and navigation tools and how they can be used to follow the application process towards its conclusion, or simply towards finding the content it is looking for. This aspect is particularly important in PCE because the user is trying to achieve a goal: improve the content that is presented to the viewer. If at some point during the process the publisher is not able to understand what the next step is then this goal can not be reached.

- **Credibility:** relates to the ability of the web application to gain the trust of the user in that it will perform the tasks in a reliable manner. During the creation of content this aspect is of particular interest in two ways. First, the publisher needs to trust that the content it is creating is going to be saved and used in the manner it is intended and second, from the point of view of the viewer, the content needs to be that which the publisher intended for them.

- **Understandability:** describes the degree to which the layout and other visual elements of the system allow the user to comprehend how the application works, what it can do and what is its current state. This is of especial importance in PCE since if the user does not understand how to manage the content then it will not create it.

3.2.1.2 Medium Importance

- **Learnability:** is the extend towards which the system makes it easy for a new user to learn its use and to guide it towards the desired action or content. Regarding PCE this is particularly important when the publisher is confronted with a new plug-in. Details like how the information should be entered and what kind of information the system is expecting are necessary to allow the publisher to do its work without delays.

3.2.1.3 Low Importance

- **Accessibility:** examines the ease to access and work with a system, in the case of web applications, it relates mostly to users with different disabilities and the use of standards that facilitate the interaction with assistant software. Even though on the open web this is an important factor, it does not affect

DPG as it is currently used, for this reason, while desirable, it is not a primary goal due to time limitations.

- **Customization:** implies the capacity of the system to deliver especial content targeted towards the user based on previous use of the system or other data gathered regarding the wishes and objectives of the user. In the case of PCE this aspect of usability does not have priority since the actions to be taken are straight forward, although there could be small details like short cuts towards the content most recently modified, they would not add much value since once an *entity-instance* is created it is usually not modified.

  The next section describes the issues found in PCE. The aspects of usability that they influence will be analysed, the current situation will be explained and sources and possible solutions will be discussed.

3.3 Lack of feedback

This issue relates mainly to the following usability aspects:

- *Credibility*: If the system feedback to the actions of the user does not communicate in clear and consistent way about the current status, the user will not be able to trust that the system is working in the desired manner.
- *Understandability*: Feedback is one of the main tools that any system has to communicate with the user so that it understands what the current situation is and the steps that it needs to take to achieve the desired goal. If the feedback fails to be understandable then the user can be lost as to what to do next.
- *Learnability*: The inexperienced user needs feedback to learn how the system works and what is the expected input. If the communications from the system is not clear then it is not possible to learn how it functions.

3.3.1 Current Situation

On submitting a new *entity-instance* or changes in an already existing one there are two possibilities: the submission is a success, the changes are accepted and persisted; or it fails, be it because of failing validation or other reasons. In the case of success the user is redirected to `editContentSubmit.html` (Figure 2.4 on page 27). In the case of failure, `editContentForm.html` (Figure 2.3 on page 27) is reloaded without the changes made by the user.

During the different stages of the process there are several deficiencies in the feedback information provided to the user.

- Before the data is submitted in the `editContentForm.html` web page the required fields are not marked as such. This is true when an existing *entity-instance* is edited but not when a new *entity-instance* is created, adding inconsistency of feedback to the problem. This is shown in Figure 3.1
- If a submission of `editContentForm.html` fails, the web page is reloaded but there is no error information or any indication that there was an error, the data entered by the user is lost and the old data is reloaded.
- If a submission is a success the `editContentSubmit.html` web page is loaded but the new data is not displayed. It is only available by following the links to go back to the `viewDetails.html` web page or to PV subsystem.

3.3.2 Discussion

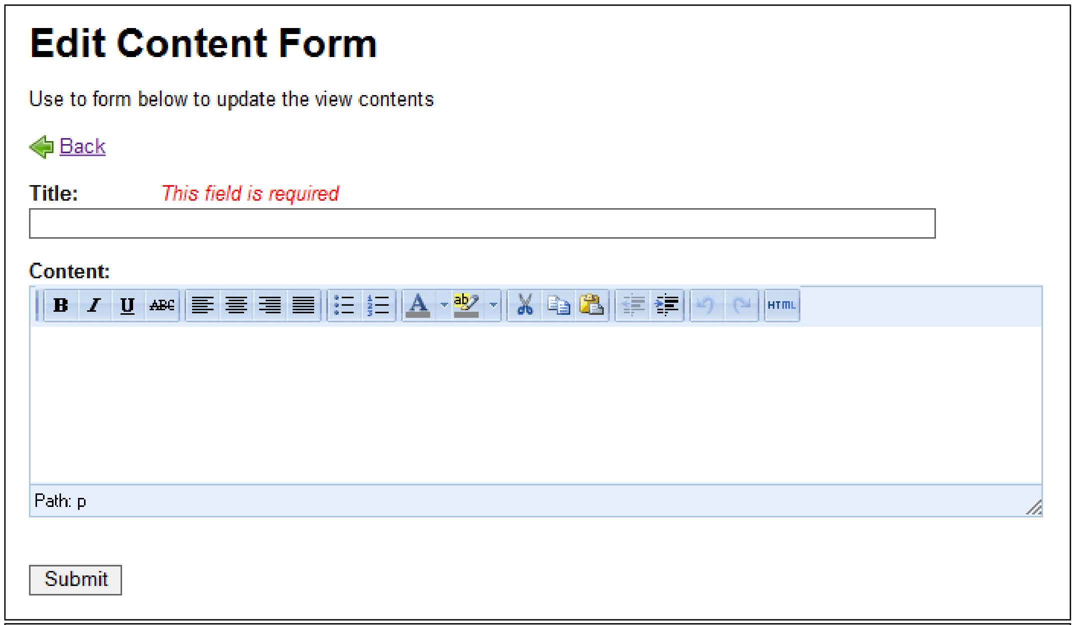
Feedback relates to giving information to the user about the result of an action it has taken and allows it to continue with other activities. To explain the importance of feedback in a system [48] compares it with real life situations. For example if it took a couple of seconds between the moment that a person starts writing and the words appear in the paper it would be difficult to continue writing without knowing where the words are located and if they were correctly written.

In the case of PCE there is some feedback but it is insufficient and does not provide the user with all the information that it needs. There have been several studies on the best practices of form feedback, two examples are [66] [20]. They examine several methods of form validation and explain the different moments where feedback can be given and the reaction of users.

In particular [66] did user testing to evaluate different methods of validation and feedback. Six different forms where used for the test with the control version using *after submit validation* and the others using different *inline* validation methods. The results show that inline validation was better in every aspect.

In respect to the moment when the feedback is given to the user, the method used currently by PCE falls into the category of *after submit validation* and not all the data is provided to the user as explained in section 3.3.1

Selected Solution



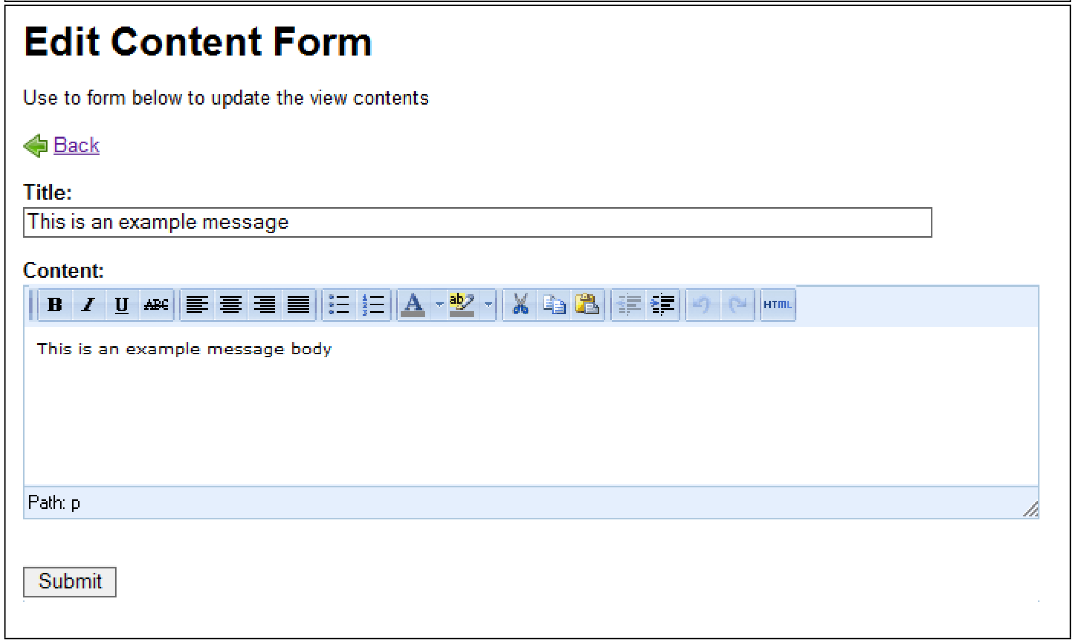
Edit Content Form

Use to form below to update the view contents

Title: This field is required

Content:

Path: p



Edit Content Form

Use to form below to update the view contents

Title: This is an example message

Content:

This is an example message body

Path: p

Figure 3.1: Two screen shot of the editContentForm.html web page. On the top in Add state with the feedback to the user explaining which fields are required. On the bottom in Edit state without any feedback.

Add inline validation and feedback in the different forms of PCE and the data missing as explained in section 3.3.1.

3.4 Unnecessary page loads

This issue relates to the *effectiveness* usability aspect. Unnecessary page loads delay the completion of a procedure by incrementing the amount of times that the client needs to communicate with the server without adding to the functionality.

3.4.1 Current Situation

In section 2.3.1 on page 22 it is explained that it is necessary to pass through four different web page loads to add or edit content in a presentation. The first three pages have the same functionality, help the user find the atomic unit of content (the *entity-instance*) that it wants to modify. They are part of the task *Content Selection*. This is accomplished by narrowing the scope of the elements from the presentation pattern shown in each web page. The `listContent.html` web page displays pages and views, the `viewDetails.html` web page displays entity-instances in a view and the `editContent.html` web page displays only one *entity-instance* with all the fields that it contains.

After the user has completed editing and submits the changes, there is one last web page in the work-flow, `editContentSubmit.html`, its functionality is to provide the user with feedback regarding the result of the changes that the user executed and links to see the content in the PV.

3.4.2 Discussion

A family of design patterns called In-Page Editing is described in [51] based on the design principle “Allow input wherever you have output” presented by Alan Cooper in [9]. As the principle advocates that if it is possible to see the content then it should be possible to edit it in the same place. This would imply that the best place to allow the edition of content in DPG is PV; however this is not a trivial task due to the implementation of the Presentation Pattern and the separation of roles in the system.

At least two roles are involved when creating a new presentation pattern. The first role is the *Pattern Designer*, it is in charge of creating the data structure for the pattern. The knowledge required of this role regards how the content should be organized and the XML syntax used to describe such structure.

The second role is the *Graphic Designer* whose task is to desing and create the XSLT and CSS files necessary to present the data in PV. The knowledge required in this role are the data structure, and the languages used by the system, HTML, XSLT and CSS.

To display content in PV the system uses a method similar to that employed by `viewDetails.html` explained in 2.4.3. As consequence, in order to allow edition in PV, the XSLT would have to include some kind of link or anchor to inform the system that it should allow edition. For this to be possible the *Graphic Designer* would have to know how the system works which would nullify the separation of concerns in the creation of a new pattern.

As a matter of fact, the system is already capable of doing this, however, because the designers of the pattern currently in use were not aware of this fact, this functionality is not used. As a result an alternative solution would be preferable.

In PCE the content is presented in the third loaded web page, `viewDetails.html`, nevertheless editing currently is not possible until loading the fourth web page of the workflow. Using the Single-Field pattern presented in [51] the fourth page could be removed from the work flow.

Selected Solution Integrate the functionalities of `viewDetails.html` with those of `editContentEdit.html` into only one web page using the In-Page family of patterns [51].

3.5 Difficult to find entity-instance

Several usability aspects are affected by the ability to locate entity-instances, mainly *effectiveness* due to the amount of effort that is necessary for the user to find the desired *entity-instance*.

Other aspects affected are *learnability* and *understandability* especially by how the *entity-instances* are ordered. This will be explained with more details in the discussion in section 3.5.2.

3.5.1 Current Situation

The responsibility of the `viewDetails.html` web page (figure 2.2) in the *Content selection* task is to provide the user with the necessary elements to find the desired *entity-instance*. For this purpose the page displays a list of all the *entity-instances* in a *view*. Currently the web page shows all the *field* values and *sub-entities* contained in the *view*.

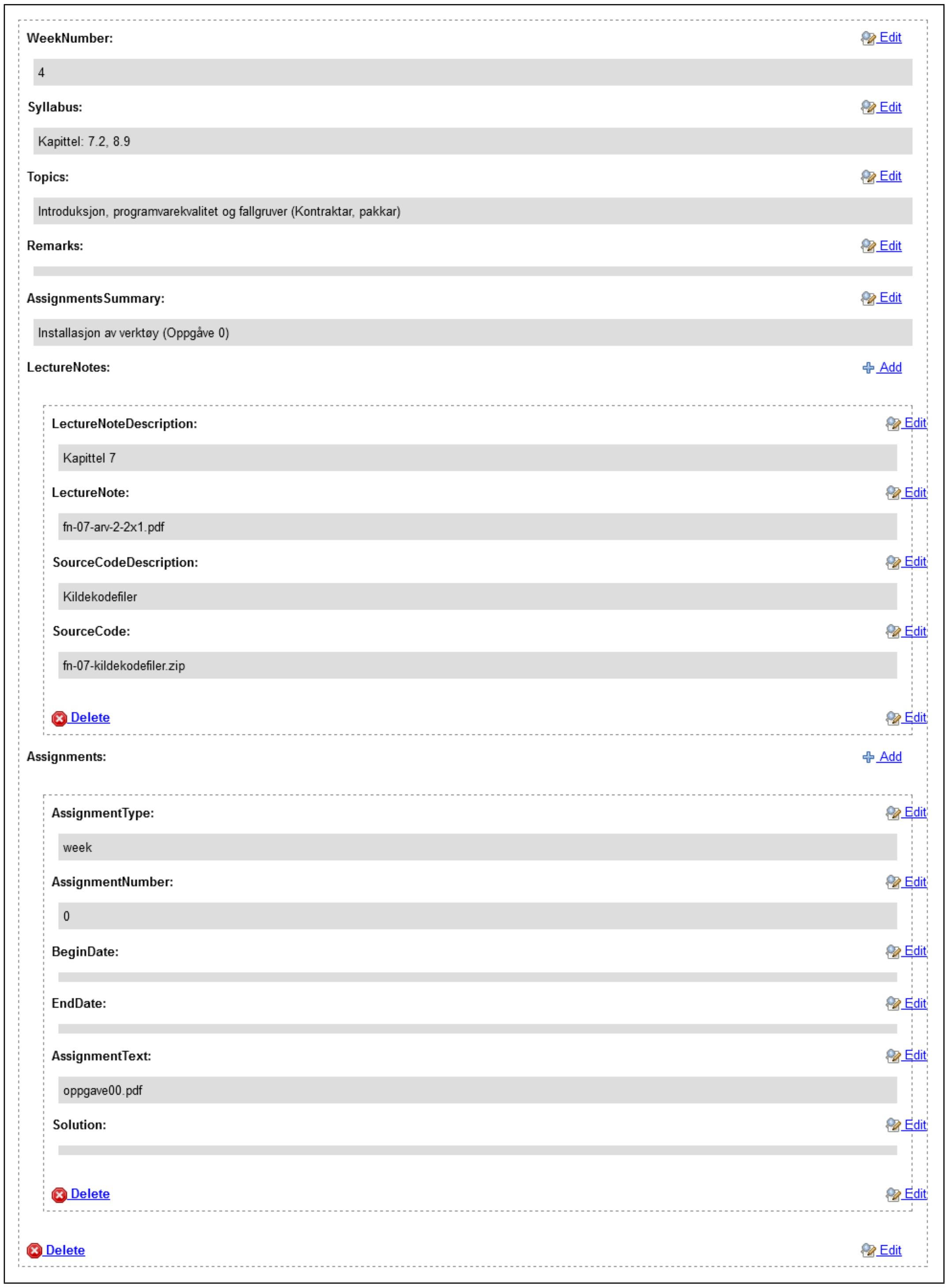
3.5.2 Discussion

The `viewDetails.html` web page accomplishes its objective, however, it fails in making this task effective due to the volume of information presented and the size of the web page itself. For example the `viewDetails.html` that displays the content in the `listWeekView` of the presentation *INF101-F, våren 2013* based on the course pattern contains 14 *entity-instances*, each of which has two *sub-entities* and holds 178 fields in total. In a monitor at 1366x768 resolution it takes 24 screen scrolls to see all the content in the page. A `listWeekView` *entity-instance* can be seen in figure 3.2

Even though the myth of the page fold that stated that users tend to avoid scrolling has been proven false. It has also been proven that the real state before the fold is important and should give as much important information as possible without overloading the user and should invite the user to continue scrolling [8] [55]. In the case of the `viewDetails.html` the most important information is an overview of the *entity-intances* in the *view* not all the details of each one.

Another factor that diminishes effectiveness is that after a user has selected an *entity-instance* to work with and modifies it, on returning to `viewDetails.html` the *entities-instances* have been reorganized. This is because they are presented in the same order as they are in the underlining XML content file. This behaviour increases the difficulty of finding the next desired *entity-instance* since they are no longer in the expected position. This is aggravated in cases where the *entity-instances* have an apparent order. A clear example of this can be found in the previously mentioned `listWeekView` which is organized in `week` *entity-instances*. The first field of the `week` entity is `WeekNumber` after updating information in the system PCE shows these weeks ordered by default and not by week number that would be the logical choice.

In summary there are three issues that make the task for the user difficult: *Information Overload*, *Size of the web page* and *Entity reorganization*. The second is a consequence of the first, and possible mitigation measures are:



WeekNumber: Edit

4

Syllabus: Edit

Kapittel: 7.2, 8.9

Topics: Edit

Introduksjon, programvarekvalitet og fallgruver (Kontraktar, pakkar)

Remarks: Edit

AssignmentsSummary: Edit

Installasjon av verktøy (Oppgåve 0)

LectureNotes: Add

LectureNoteDescription: Edit

Kapittel 7

LectureNote: Edit

fn-07-arv-2-2x1.pdf

SourceCodeDescription: Edit

Kildekodefiler

SourceCode: Edit

fn-07-kildekodefiler.zip

Delete Edit

Assignments: Add

AssignmentType: Edit

week

AssignmentNumber: Edit

0

BeginDate: Edit

EndDate: Edit

AssignmentText: Edit

oppgave00.pdf

Solution: Edit

Delete Edit

Delete Edit

Figure 3.2: Screen shot of an *entity-instance* in the listWeekView view of the course pattern.

- *Show only fields of first level entity-instances*: While this reduces the size of the web page to some degree, a quick test shows that it is not enough, for example in the listWeekView the size is reduced from 24 screens to 11 and the number of fields displayed from 178 to 70 but it still requires for a visual scan of 11 screens of content.

- *Add collapse/expand functionality*: This would allow the user to collapse all the data of each *entity-instance* into a limited amount of fields that are identified as the most important for that type of *entity-instance*. However, due to the characteristics of the presentation pattern specification it is impossible to know a priori which fields are important to recognize an *entity-instance*. There are several possible solutions for this, the optimal one includes changing the specification to allow the pattern creator to add a property to each field that describes it as "important" and update the course pattern, the only one currently in use, to reflect this change.

- *Better use of white spaces*: The importance of the correct use of white space for comprehension in web pages with important amount of content has been discussed by several authors [65]. In `viewDetails.html` there is no lack of white space, but it is homogeneous in the distance between fields of the same *entity-instance* and between the *entity-instances* themselves. This gives the possibility of reducing the space between fields to allow for the user to easily group the fields of one *entity-instance* as the same cognitive unit while at the same time reduce the size of the web page.

Possible solutions for *Entity reorganization* are:

- *Create a default order*: Either by adding the possibility in the presentation pattern specification for the pattern creator to select a criteria by which the *entity-instances* should be ordered. This approach would take into consideration how the *entity-instances* were designed to be used but not how they are actually used and the two are not necessarily the same.

- *Allow the user to select the order*: Add the functionality of ordering the first level of *entity-instances* according to a field of their choosing. This option allows for the user preferences to be taken into consideration and it does not require deep changes to the system.

The measures mentioned so far target existing problems in the existing solution and add new functionalities to mitigate those issues. It is possible nevertheless, to go a step farther and add a search functionality. By allowing the user to type what they are looking for and filtrating the content to only show the search results.

Selected Solution

Only display first level *entity-instances* by default and implement collapse/expand, order and search functionalities.

3.6 Cross site Scripting

This issue influences the *credibility* of the system. The user needs to be able to trust as much as possible that the website will not allow third parties to inject attacks into the process. Vulnerabilities to XSS leave the system open to attacks that may ultimately diminish the credibility of the site.

3.6.1 Current Situation

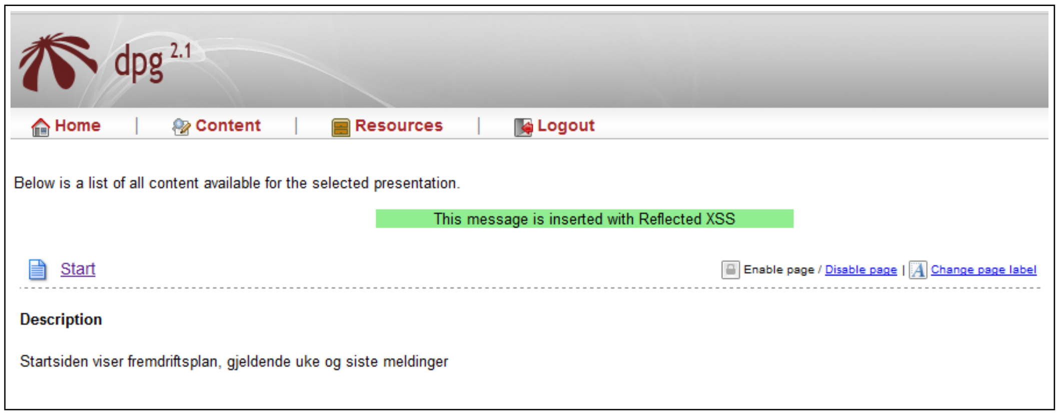
The security of PCE has been the object of several reports, most recently [47], and one master thesis [57]. One issue that is discussed in both but has yet to be solved is the vulnerability of the system to Cross Site Scripting (XSS) attacks.

3.6.2 Discussion

A successful XSS attack is the result of an attacker being able to inject code into a client application of a trusted website. Once this is achieved, the client browser can not differentiate whether the code comes from the original, trusted site or from an attacker and will treat and execute both in the same manner; allowing them access to ambient authentication, cookies and other data related to the original site. Any application that uses data entered by the user as part of its output could be vulnerable to Cross-site Scripting attacks.

In [47], done in collaboration with Anne Elise Weiss, PCE was analysed using HP Fortify Static Code Analyser (SCA) [45] and OWASP Zed Attack Proxy (ZAP) [44] for penetration testing. One example of *reflected* XSS [43] was discovered in the `listContent.html` web page since it receives a get parameter that is inserted into the body of the page without any sanitation. This problem is easy solvable by changing the method which is employed to display the status message. An example of an attack can be seen in 3.3. The url requested in that attack is `localhost:8080/dpg2/pce/content/listContent.html?pid=219_definitive&statusMessage=This+message+is+inserted+with+Reflected +XSS`.

A more serious threat to the DPG users however is the vulnerability that PCE presents to *persistent* XSS attacks [43]. This type of attack is successful when the malicious code



Below is a list of all content available for the selected presentation.

This message is inserted with Reflected XSS

Description

Startsiden viser fremdriftsplan, gjeldende uke og siste meldinger

Figure 3.3: Example of XSS attack in the listContent.html web page. Screen shot from [47].

can be persisted into the system. As a consequence all users that access the content of the website after the attack are affected by it. The reason for this vulnerability is usually lack of sanitation, validation and canonization of data entered by an untrusted source before being stored. This situations happens in all fields of PCE except those that provide a reduced list of possible values and are displayed in combo boxes. These values are corroborated against the list of possible choices and if they do not match, they return an error. This is the only case in PCE where there is data validation.

Selected Solutions

Remove the `statusMessage` from the `listContent.html` web page and find already existing frameworks that could sanitize the user input on the server side.

4

Solution Design

In the previous chapter the problems with the usability of the current implementation of PCE were discussed, together with possible solutions for each of these problems and the best options were selected for implementation. This chapter takes the selected solutions and analyses the changes needed to the current system in order to implement them.

4.1 Proposed Solutions for enhancing PCE

The proposed solutions selected in the previous chapter are:

1. Integrate the functionalities of the `viewDetails.html`, `editContentEdit.html` and `editContentSubmit.html` web pages using the In-Page Editing family of patterns [51].
2. Add inline validation and feedback in the `editContentForm.html` web page.
3. Only display first level *entity-instances* by default
4. Implement collapse/expand.
5. Implement ordering of *entity-instances*
6. Implement search for *entity-instances.*
7. Remove the `statusMessage` from the `listContent.html` web page.
8. Research already existing frameworks for sanitizing user input on the server side.

While solution 7 can be easily implemented by changing the expected data type of the `statusMessage` parameter from a string to an integer and change the code used to display the message from the one shown in Listing 4.1 to the one shown in Listing 4.2; implementations of the others solutions are not so straight forward. As a consequence of solution 1, functionalities that before were distributed amongst three web pages, and their corresponding controllers, are now in one, and due to solutions 2 to 6 new functionalities will also be implemented into `viewDetails.html`.

Listing 4.1: Current code that interprets the statusMessage parameter.

```
<% if (request.getParameter("statusMessage") != null) { %>
  <center>
    <span class="statusMessage">${param.statusMessage}</span>
  </center>
<% } %>
```

Listing 4.2: Proposed code to interpret the statusMessage parameter.

```
<% if (request.getParameter("statusMessage") == 1) { %>
  <center>
    <span class="statusMessage">The element has been enabled</
       span>
  </center>
<% else if (request.getParameter("statusMessage") == 2) { %>
  <center>
    <span class="statusMessage">The element has been disabled</
       span>
  </center>
<% } %>
```

4.2 Definitions

The current implementation of PCE is founded on the traditional HTTP [60]/HTML web page architecture, based on a thin client that is only responsible for presenting HTML to the user and dispatching requests to the server with synchronous interaction. This leaves the business logic, data state and data storage on the server side.

This approach is not optimal for implementing the proposed solutions since it would imply that for each action from the user, an HTTP request to the server would have to be made, the client would have to wait for the HTTP response and the response would include data that the client had in a previous page load and presentation HTML together with the new information.

4.2.1 Asynchronous JavaScript and XML (AJAX)

Asynchronous JavaScript and XML (AJAX) [46] [40] as the name insinuates is a combination of several pre-existing technologies that has evolved into an Internet standard in the last 15 years. AJAX utilizes XMLHttpRequest [62] to request data asynchronously from a server. This data is transported in XML format and a combination of Javascript, XHTML and Cascading Style Sheets (CSS) [58] are used to manipulate the Document Object Model (DOM) [59] [64] to display the data.

AJAX provides the opportunity of reducing the size of the data being transmitted between the server and the client, since everything that has to do with presentation and functionality can be sent to the client only once; on the first page load. The rest of the requests would be for state changes of the application, and the data will only contain relevant information that describes the modification to the state since the last request.

Another advantage of this approach is that from the point of view of the user the page is not reloaded, enhancing the experience. An example of the improvements perceived in the usability is the scrolling of pages. As explained in Section 3.5 on page 42 one of the factors that make finding *entity-instances* more difficult is the amount of information presented in the `viewDetails.html` web page, and moving through that page. Even though steps are going to be taken to improve the situation and reduce the size of the web page, using AJAX and eliminating page loads will add the benefit that the user will never loose the place in the page being displayed.

4.2.2 Single page applications (SPA)

A consequence of the introduction of AJAX into web development is the apparition of Single Page Applications (SPA) [54]. These applications are composed of a single web page that in itself contains several independent components with their own functionality that support intermediary states.

The increasing complexity of the web page has several disadvantages, mainly that it makes it difficult to develop and maintain the applications if it is not designed from the beginning taking into consideration a clear knowledge of the different components and their states. As it happens in the server, it is necessary to separate the concerns of data.

Since what before was a simple static HTML page has become an application it is necessary to apply applications solutions to the problems that this new conception brings. One of the solutions to this problem has been to adapt the concepts presented by the MVC pattern. This will be discussed in more depth in Chapter 5.

4.3 Refactoring to a SPA

There are two types of tasks to accomplish during the implementation of the client side application for this project: the first one is to refactor the already existing functionalities into a SPA unifying the `viewDetails.html`, `editContentEdit.html` and `editContentSubmit.html` web pages into one and adjusting some of its behaviour (Solutions 1, 2, 3 in Section 4.1 on page 48); the second requires the implementation of new functionalities (Solutions 4 to 6 listed Section 4.1 on page 48).

There has been several studies and papers describing different approaches that can be used to create SPA and refactor multi-page web applications into SPA [49] [34] [32], also called Rich Internet Applications (RIA) [15].

A 5 step migration process is delineated in [34].

1. *Retrieving Pages*: Examine the application from the point of view of the user and use static analysis to follow links to other pages and dynamic analysis to retrieve pages that require parameters.
2. *Navigational Path Extraction*: Understanding how the user navigates from one page to the other and what are the different paths that it follows.
3. *UI Component Model Identification*: Identify the portion of the web pages that changes when navigating to the next page in the navigational Path.
4. *Single-page UI Model Definition*: Select an AJAX component that will represent the components identified in the previous step. This component should be defined in an abstract model so that it is not dependent of the tools that will be used for the implementation and should describe state changes and navigational paths inside the component.
5. *Target UI Model Transformation*: Transform the abstract model described in the previous step to the platform-specific elements. This step will be explained in Chapter 5.

No abstract model is specified for step 4 of the process in the approach presented by [34]. The approach described in [49] however suggests *Abstract Data Views (ADVs)* as described by [10]. ADVs are objects that specify the presentation and presentation behaviour of a RIA interface component, however they do not deal with the data or business logic of the element, this is the responsibility of the Abstract Data Object (ADO). Each ADV declares an ADO *owner*, defines how it will be presented in the UI and interacts with it, being able to change its state or trigger its behaviours.

The UI transformations that ADVs go through can be described using ADV charts. For each possible interaction with the ADV the chart defines an event that triggers the transformation, a precondition that has to be met before the transformation can take place and a postcondition that describes the new state of the interface. An example of an ADV chart can be seen in Figure 4.2 on page 56.

Since ADVs are related to ADOs, it is necessary to also model the data of the application. As a result of following the steps defined by [34] and using ADV for modelling the interface components and ADOs to model the content four diagrams are created:

- Navigational Path Diagram for the current application
- User Interface Components in the form of ADVs
- Data Object Diagram in the form of ADOs
- Navigational Path Diagram for the new implementation

The last three elements coincide with the three models described in [16]. Since these names are easier to recognize and understand at first glance they will be used instead:

1. Presentation Model
2. Content Model
3. Navigational Model

It is important to mention that the Content Model will be created based on the data needed by the client application and not as it is modelled in the Server.

4.3.1 Current Navigational Model

To do step 1 of the process [34] recommends using automatic tools to map the possible *Navigational Paths* in the system. However PCE has a very straight forward navigational model. The URLs within *Content Managment* tasks in PCE can be seen in table 4.1 and the activity diagram for the user workflow shown in Figure 2.5 on page 28, also serves as the Navigational Model for the current system.

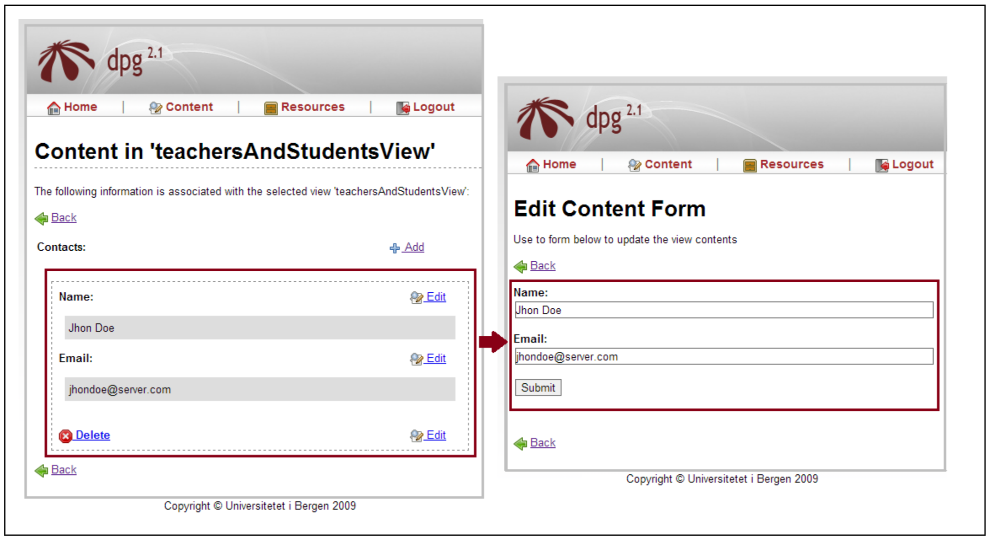
4.3.2 Presentation Model

In the workflow in Figure 2.5 on page 28 it is possible to see that there are currently five transitions between web pages. From `listContent.html` to `viewDetails.html` , from `viewDetails.html` to `editContentForm.html`, from `editContentForm.html` to `editContentSubmit.html` (on add or on edit), from `viewDetails.html` to `editContentSubmit.html` (on delete) and lastly from `editContentForm.html` to `editContentForm.html` (on error).

In analysing these transitions more closely, it can be observed that only the header and the footer of the page do not change from `listContent.html` to `viewDetails.html`. The level of content presented changes from displaying *Pages* and *Views* to displaying *entity-instances*. It is still possible to model this in a single page with the client requesting the data from the server and keeping the presentation aspects from changing. However, the benefits

URL	**Param**	**Param Description**
/pce/content/listContent.html	pid	Id of the presentation
/pce/content/viewDetails.html	pid	Id of the Presentation
	pageId	Id of the page
	viewId	Id of the View
/pce/content/editContentForm.html	presentationId	Id of the presentation
	pageId	Id of the page
	viewId	View Id
	entityInstanceId	Id of the entityInstance
	entityPath	Indicates the location of the entity instance in the view
	entityAction	Indicates whether the user is adding a new entity-instance (ADD) or editing an existing one (EDIT)
/pce/content/editContentSubmit.html	presentationId	Id of the presentation
	pageId	Id of the page
	viewId	View Id
	entityInstanceId	Id of the entityInstance
	entityPath	Indicates the location of the entity instance in the view
	entityAction	Indicates whether the user added a new entity-instance (ADD) or edited an existing one (EDIT)

Table 4.1: List of PCE Content Management URLs and their parameters.


dpg 2.1
Home | Content | Resources | Logout
Back
Add
Edit
Delete
Back
dpg 2.1
Home | Content | Resources | Logout
Back
Submit
Back

Content in 'teachersAndStudentsView'

The following information is associated with the selected view 'teachersAndStudentsView':

Contacts:

Name:

Jhon Doe

Email:

jhondoe@server.com

Copyright © Universitetet i Bergen 2009

Edit Content Form

Use to form below to update the view contents

Name:

Jhon Doe

Email:

jhondoe@server.com

Copyright © Universitetet i Bergen 2009

Figure 4.1: Differences between the presentation of content in the viewDetails.html and editContentForm.html web pages.

to the usability would not be substantial and due to the time constrains this transition will be left out of the solution design of the new PCE.

The second transition however has more promise. There are two paths to transition from `viewDetails.html` to `editContentForm.html`. *Update entity-instance* and *Add entity-instance*. We will take a look at each of them separately.

Update entity-instance
As can be seen in figure 4.1 the content displayed in the `editContentForm.html` web page is already presented in the `viewDetails.html` web page, only the presentation itself changes. In `viewDetails.html` the value of the *fields* are displayed as simple text and can not be edited, while in the `editContentForm.html` web page they are presented in form HTML elements or WYSIWYG and it is possible to edit the values. As explained in 4.3 this identifies the representation of the fields as UI components.

Add entity-instance
This transition is different in that the content does not exist in the `viewDetails.html` web page since the user has not yet created it. To add the new content it is necessary to know the data structure of the *entity-instance* to be created. This means that the `viewDetails .html` web page will need to have information not only of the content of already existing *entity-instances* but also of their structure and the structure of the possible *entity-instances* that can be added to the *view*.

The transitions from `editContentForm.html` to `editContentSubmit.html` and from `viewDetails.html` to `editContentSubmit.html`, happen on Update or on Add, in the first case and on Delete on the last. The transitions are currently alike in that the new content is not displayed on the `editContentSubmit.html` web page. Showing the changes made to the content is something that is desired because of the reasons explained in Section 3.3.1. However this is something that falls into the category of new functionality and will be analysed in 4.4. The on add case is different since space is needed to add the new *fields* for the new *entity-instance*, this can be solved using an overlay window as recommended and detailed by [51] in cases where there is not enough space for the edition of content to take place.

The last transition, from `editContentForm.html` to `editContentForm.html` happens when there is an error on the form submit. In this case there is no change in the current system, except for the loss of the content entered by the user.

The results following this analysis confirms that solution 1 described in 4.1 coincide with the necessary steps of transforming the `viewDetails.html` web page into a SPA in charge of modifying the content within a *view*. They also provide the necessary information to create the ADV charts that will be the basis for the implementation of the different UI components. Figure 4.2 shows some of the ADVs and ADV charts of the Presentation Model.

For example the first diagram on the top left represents the `BasicPluginFieldADV`. This ADV acts as the parent of all the FieldPlugins in the new system and shows their interaction with the user. The diagram shows that the component has two internal states and can transform from one to the other in any direction. The transformations are described in the ADV chart under the diagram: Transformation 1 happens on the MounseClick event if the parent of the FieldPlugin has Focus. If the precondition is met and the event is triggered then the postcondition will be applied; the DisplayElement used to show the content will be taken from the web page and a WISIWYG that allows editing will be added to the web page.

4.3.3 Content Model

The Content Model needed for PCE has one particularity, the structure of an *entity-instance* or even a *field* is not known since they depend on the definition of the *entity* in the corresponding *pattern* and the *plugin* respectively. It is important to remember that on the Presentation layer of DPG when talking about the *field* level of content it refers to the *plugins* so each different type of *field* has its own *plugin* implementation, and plugins have the ability of defining their own structure. As a consequence it will be necessary for the client application to be informed of the structure of *entity-instances* and *fields* that the *view* contains.

A similar model exists in the server as was explained in Section 2.4 on page 29. However, the current implementation has several issues, related mainly to the dependency that currently exists between the storage technology (XML) and the presentation technology (XSLT). Creating a new PCE presents the opportunity to separate both concerns completely and allow the content model implementation to be independent. As a consequence the presentation content model and the previously existing one are different.

The proposed content model is diagrammed in the form of a UML [17] class diagram [18]

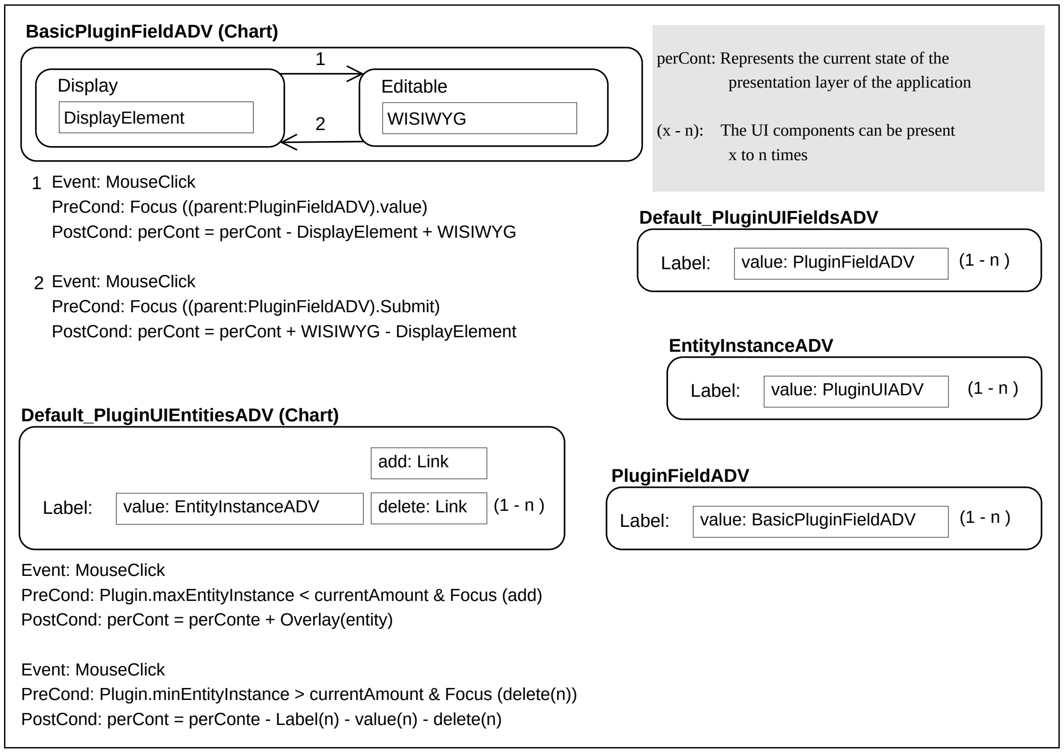

BasicPluginFieldADV (Chart)
Display
DisplayElement
Editable
WISIWYG
1
2
perCont: Represents the current state of the presentation layer of the application
(x - n): The UI components can be present x to n times
1 Event: MouseClick
PreCond: Focus ((parent:PluginFieldADV).value)
PostCond: perCont = perCont - DisplayElement + WISIWYG
2 Event: MouseClick
PreCond: Focus ((parent:PluginFieldADV).Submit)
PostCond: perCont = perCont + WISIWYG - DisplayElement
Default_PluginUIFieldsADV
Label:
value: PluginFieldADV
(1 - n)
EntityInstanceADV
Label:
value: PluginUIADV
(1 - n)
Default_PluginUIEntitiesADV (Chart)
add: Link
Label:
value: EntityInstanceADV
delete: Link
(1 - n)
PluginFieldADV
Label:
value: BasicPluginFieldADV
(1 - n)
Event: MouseClick
PreCond: Plugin.maxEntityInstance < currentAmount & Focus (add)
PostCond: perCont = perConte + Overlay(entity)
Event: MouseClick
PreCond: Plugin.minEntityInstance > currentAmount & Focus (delete(n))
PostCond: perCont = perConte - Label(n) - value(n) - delete(n)

Figure 4.2: ADV and ADV charts of the main UI components of the Presentation Model for the client side application.

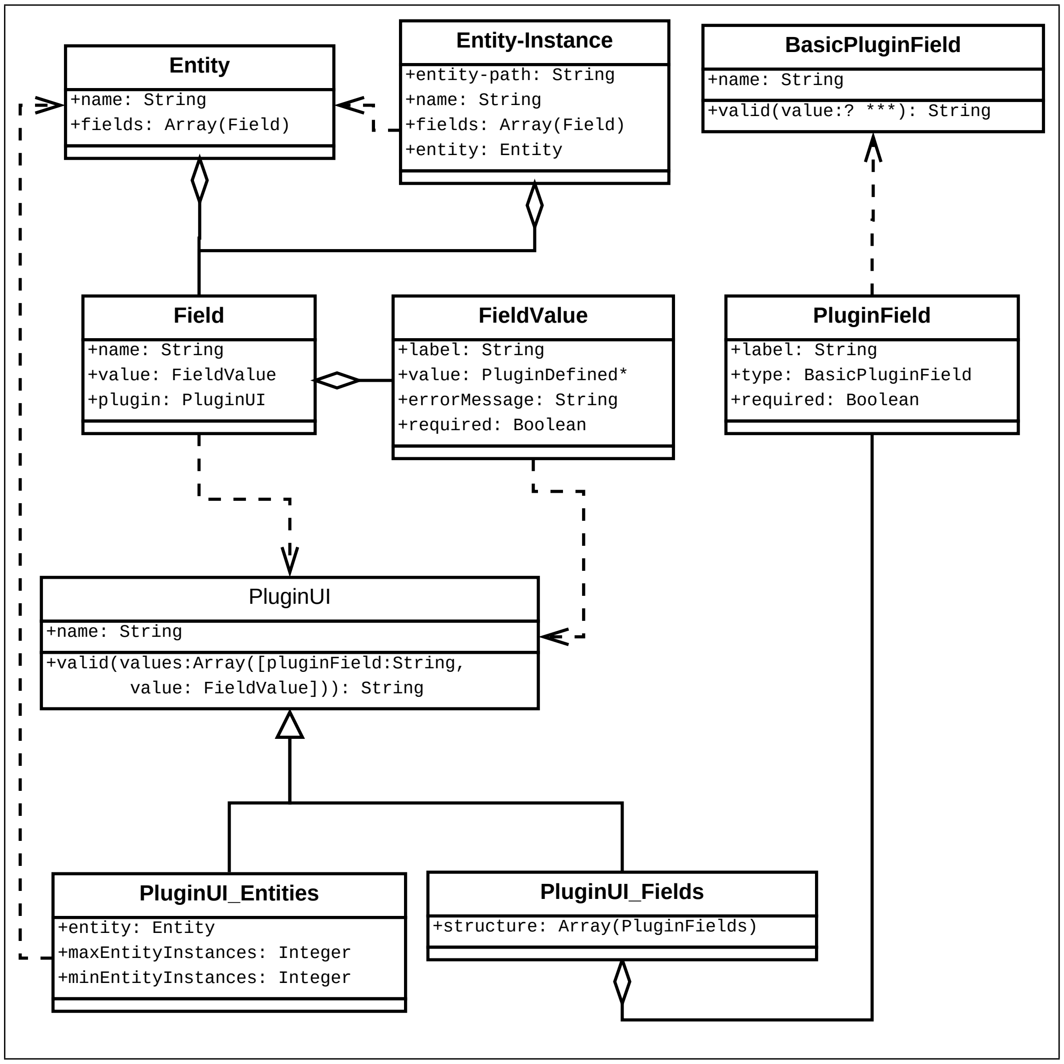


Figure 4.3: Class diagram of proposed Content Model for the client side application.

and can be seen Figure 4.3. It defines the main structure and value classes that will be used to represent the different elements in PCE.

- *BasicPluginField*: Represents a basic or primitive type of data, for example String or Date. It contains its own validation logic that restrict the values that data of a BasicPluginField type can take.
- *PluginField*: Provides the necessary structure for the fields that are specified in a Plugin.
- *PluginUI*: Contains the logic and data that all specifications of plugins have in common.
- *PluginUI_Fields*: Represents the types of plugins that are composed of PluginField. The homologue of the presentation pattern type of field that are not *list* or *entity.*
- *PluginUI_Entities*: Represents the types of plugins that contain *entity-instances* as part of their structure for example the current *list* or *entity* types.
- *Field*: Provides the structure for the field that is part of an entity. And connects the field with the plugin that is used to represent it.
- *FieldValue*: Represents the content data that correspond to a field in the entity.
- *Entity*: Provides the structure for the entity-instance.
- *Entity-Instance*: Main content unit. It is an instance that takes the structure provided by entity and takes the values entered by the user.

4.3.4 New Navigational Model

The new Navigational Model shown in Figure 4.4 presents itself after the creation of the ADVs and simplifies the previously existing one by unifying `viewDetails.html`, `editContentForm.html` and `editContentForm.html` and adds the overlay UI for adding new *entity-instance.*

4.4 New Functionalities

4.4.1 *Entity-instances* collapse/expand

In Section 3.5.2 on page 43 one possible implementation of this functionality was discussed. It implied changing the *Specification Pattern* to allow the Patter Designer to specify which *fields* in each *entity-instance* should be considered important and would be displayed when the *entity-instance* was in collapsed state. The implementation of this option would imply changing several parts of the system and the presentations already created for DPG. Because of this issues another option was conceived.

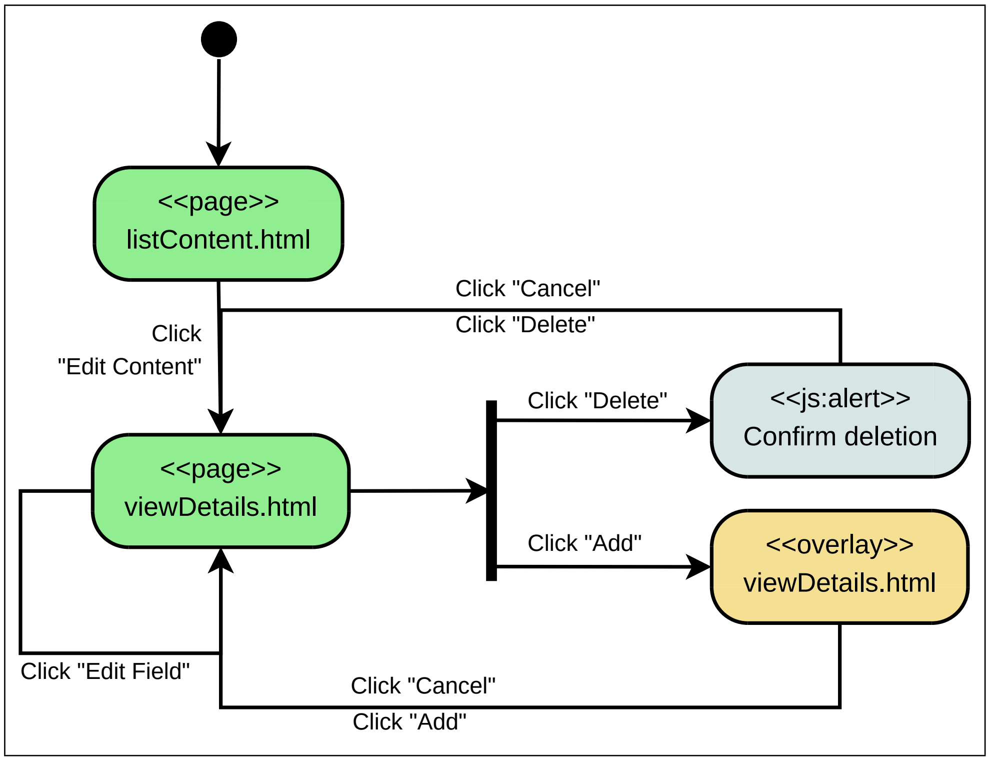


Figure 4.4: Activity diagram describing proposed workflow of Content Management tasks in DPG

It is safe to assume, and it has been the case so far, that the pattern designers will put the most important *fields* of each *entity-instance* first in its definition. And it could also be set as a rule from this point on for future creations of patterns. Taking this into consideration when *entity-instances* are collapsed the first field defined by the pattern will be the one displayed.

The changes necessary to implement this functionality are then only to the presentation model, where each *entity-instance* will have a new *interactive element* that will allow the user to change the state from collapsed to expanded and vice versa.

4.4.2 Adding Feedback

In Section 3.3.1 several instances where feedback to the user could be improved where mentioned. Here is a summary of the ones that where selected to be implemented:

- Mark required fields in all form instances.
- Add inline validation messages for submission errors.
- After successful modification of content, the new content should be shown.

The changes required to implement the mentioned functionalities involve the *Content Model* and the *Presentation Model*. In the *Content Model*, two attributes need to be added to the *field* ADO; one that will mark the *field* as required and another that will contain messages from the server in case of error. The modifications to the *Presentation Model* include adding UI elements that can be used to show both messages.

4.4.3 *Entity-instances* ordering

Several factors need to be taken into consideration when deciding on the implementation of this functionality, here is a brief list:

- What is going to be ordered?
- What UI component will be responsible for ordering?
- What criteria will be used for ordering?

To reply to the first two questions, lets take a look at how *entity-instances* are organized. Each *view* can only have one *entity-instance* and each *entity-instance* can have 1 to n *fields*, the fields can be simple, have one attribute, composite, have their own structure, or complex, have children *entity-instances*. There are two types of complex fields in the system so far, the *entity* type, that can have one *sub-entity* or the *list* type that can have 0 to n entities. It is this last case that is of interest to this functionality, as the *list* type is the only case where *entity-instances* of the same *entity* type are together and they are the ones that need to be ordered.

The responsibility for ordering the *entity-instance* as a consequence should be allocated to the *PluginUI* in charge of the *list* field type. This opens the door for the necessity of making the different *PluginUI* have their own implementation in the new PCE system.

Regarding what criteria to use to order the *entity-instances* it is impossible to know a priori their structure and what the user is going to be looking for. One possibility as in the previous case would be to order them according to the first field; another possibility would be to let the user choose what field they would rather use. The second possibility offers more flexibility without adding much complexity to the implementation and as such is the best option to be implemented.

For the purpose of creating this functionality the following changes need to be done: different plugins/fields need to have different ADVs in the client and they need to have control over their behaviour. Not all plugins/fields will need especial treatment so a default ADV can be created for those cases. An ADV will be created for the *list* plugin with the corresponding ordering UI component.

4.4.4 ***Entity-instances*** search

This functionality has several coinciding points with the previous one and the same questions need to be answered.

- What is going to be searched?
- What UI component will be responsible for searching?
- What criteria will be used for searching?

The PCE user that will be employing this functionality will most likely be searching for an element of the content that they want to edit or delete and will have by this point in the process found the view that the content is in. It can be inferred that what should be searchable then is all the content within the view and the main responsibility would befall the view ADV and ADO. There is one problem with this, the plgins ADVs will define how the content is to be displayed and are the only ones that have the information about what information within their content is relevant to be searched. The best solution would be for the view ADO to ask the different plugin ADOs to look within themselves and the ADV to show when the searched query has been match.

4.5 Persistent XSS Solution

Aleksander Vines did a thorough analysis of the security of DPG in his thesis [57]. Amongst the possible solutions that he proposed of the Persistent XSS attacks were the use of *OWASP*

AntiSamy Project [42] to sanitize the HTML/CSS code of content entered by the user in the system. *OWASP AntiSamy* is API for cleaning suspect HTML code, the definition of what is considered dangerous code is provided by a policy files. *OWASP AntiSamy* provides predefined policy files that range from strict, that only allows for a subset of HTML tags, to more liberal approaches, that allow everything except javascript.

OWASP AntiSamy presents a good solution for the HTML and string plugin, however other plugins can have a more strict approach through their validation logic. As an example lets take the youtube plugin, all that it needs is a URL that points towards the desired video. Said URL should start with `http://www.youtube.com/` and should not have any characters that are not alpha-numeric in the name or value of its parameters. It is important nevertheless that the core system is protected and to do that the plugin creators need to be thought of as a possible attack vector. It is for this reason that in the server a three layer protection is proposed.

1. *Plugin validation*: It is the first line of defence, the responsibility relies on the plugin creator.
2. *AntiSamy sanitation*: Clears the code of dangerous HTML and CSS tags and code.
3. *Storage sanitation*: Even though the proposed solutions until now protect the user, the system also needs to be protected and not all content is safe to save in XML code. For example the CDATA tag can not be used inside content without corrupting the data inside.

The approaches mentioned here protect DPG and the user from presentation content in the form off different type of strings. There are other plugins however that are not taken into the scope of the project due to time constrains, for example the file plugin.

5 Implementation

The result of the previous chapter was the design of a new single page application that will replace the current PCE implementation. This chapter concentrates on the development of the new PCE, the tools used during this process, the problems faced and necessary modifications to the current server-side implementation.

5.1 MV* and Backbone

When using plain Javascript and HTML or even JQuery and HTML to create sophisticated applications the result in inevitably *spaghetti code* mixing both Javascript and HTML in ways that decreases the scalability and maintainability of the application. In order to solve this issue, as client side applications increased in complexity, Javascript framworks were created. Some of these frameworks offer variations of the MVC pattern, providing the developer with means to separate the concerns of the application. However due to the nature of Javascript the role of the controller is not as independent as in the server and Javascript framworks offer what is referred to as MV* where the * changes according to the flavor of MVC implemented by the framework.

There are several Javascript frameworks, choosing the correct one for the implementation is important. In order to make this decision three actions were taken. The criteria given in the *Journey Through The Javascript MVC Jungle* [41] article were followed. The article proposes a set of variables that can be used to choose the correct framework for an implementation. The variables can be divided in two groups, general and development dependent. Examples of the general variables are: quality of documentation, maturity of the framework, size, good community. Examples of development dependent factors are: flexibility vs opinionated, developers experience, capabilities. The second action was to revise the examples in the *TODO MVC* [5] web site. *TODO MVC* provides the examples of the same application developed with the different frameworks. The third step was to read opinions on the web, for example [50]. The conclusion was the choice of Backbone [2]. Backbone is a mature framework with an active community, good documentation and minimal size. The developer did not have experience with any framework so this factor was not taken in consideration, however Backbone has a small learning curve. Backbone is also a flexible framework, this means that it does not impose a file structure on the developer and can be applied to only part of a web site, which is not the case with other frameworks.

The core components of Backbone are:

- *Model*: Contains the data of the applications and the logic necessary to interpret its behaviours and manage it, for example validation, access control, etc.
- *Collections*: They are lists used to order models. Each collection can define the type of model that it will accept and can listen to the different events triggered by the models.
- *View*: The views represent the different UI components in the application and are the intermediary between the DOM and the data. Backbone allows the use of templates for views, which permits the separation of concerns between logic and presentation. It is agnostic regarding the template technology used.
- *Events*: The event module is embedded in the other modules and allows for the communication between components. Events can be triggered by the user on a view, for example a mouse click, or by other modules, for example on change, when a model is modified.

Backbone by default utilizes RESTful web services to communicate with the server. RESTful Web services are based on the *Representational State Transfer* (REST) architecture [12] and are designed to work with stateless transfer protocols. The service exposes system resources that can be accessed by their Universal Resource Identifier (URI) and can be manipulated using the CRUD operations, each of them mapped by Backbone to an HTTP method: Create to POST, read to GET, update to PUT and delete to DELETE.

5.1.1 JSON vs XML

Backbone works natively with the Javascript Object Notation (JSON) [3] format to parse data into their objects and communicate with the server. On the other hand DPG and the different classes that are used for presentation and logic of *fields* and *plugins* are designed to work with XML; and as it was explained in Section 2.4.3 the `viewDetails.html` web page uses XSLT to show the content data. This leaves the following possibilities to make Backbone and the DPG server application compatible:

Backbone with XML Backbone is an extremely flexible library and it is possible to configure it to work with XML by redefining the parse and fetch implantations in the Collection module to work with XML. This approach offers the advantage of not having to modify how the server processes the XML data today, as the XML could be sent to the client. However this would leave the responsibility of converting the XML data into logic units to the client application. It would also imply that Backbone could not be used out of the box since extra configuration would be required making it more difficult in the future to update to a new version of the library.

XML to JSON In Java there are libraries that allow the transformation of XML to JSON and vice-versa. Two examples are Xml2Json [22] and Json In Java [21]. One of this libraries would fulfil the function that is currently served by the `PceTransformationUtil-view.txt` XSLT file; to prepare the XML content to be presented. In the current implementation the content output is XHMTL and in the new system it would be JSON. The advantage of this implementation would be that the code in the server would not have to be significantly changed since it would use the same logic as it is currently using. The disadvantages are that the XML to JSON transformation would have to be adapted to the current specification of the *Presentation Pattern* and this specification is something that has changed through time making the realization of this transformation dependent on future changes. Also, the problem with the *list* and *entity* plugin would not be solved since special codes to deal with those plugins would have to be created.

XML to POJO to JSON This option would include an intermediary step between the XML data and the JSON object in the form of a Plain Old Java Object (POJO) and it would require an implementation closer to the one used currently for the `editContentForm.html` web page explained in Section 2.4.3. Currently each plugin defines its XHTML code by using their own XSLT file or returning their own structure made out of XSLT of elements that are already defined. The advantage of this approach would be the separation of concerns between the presentation layer using JSON and the storage technology using XML. The disadvantage would the necessity of creating a new `FormBuilder` class from scratch, creating the POJO classes and modifying the *plugins* to return their editable structure with the POJO structure.

Even though the last option requires more development it has been selected because it keeps and even improves the separation of concerns in the system. Providing the possibility of completely decoupling DPG from XML in the future if PV is changed to using POJOs.

5.1.1.1 System Dependencies

Dynamic Presentation Generator, HP Fortify and ZAP [19], is a report based on an analysis of the security in DPG. One of the weaknesses mentioned regarding the implementation of DPG was that the lack of update of the dependencies since they were first declared. For this reason it was decided to add as little dependencies as possible for this project.

The only Java dependency used is a consequence of the strategy selected to communicate between the server and the client applications. Jackson [4] is a Java JSON processor with the capability of data binding a POJO through annotations or property accessors that is intuitive to use and easy to configure and set up thanks to the *Spring Framework* [1] annotation `@ResponseBody` that informs the application that the return value of the function has to be written directly to the HTTP response body.

To configure and work with Jackson, the Maven dependency has to be added in the `pom.xml` file, seen in Listing 5.1, and the URLs that will be used for RESTful services need to be mapped in the web application `web.xml` file, seen in Listing 5.2.

Listing 5.1: Code in pom.xml needed to set up Jackson.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.7.1</version>
</dependency>
```

Listing 5.2: Code in web.xml needed to set up RESTful services in DPG.

```
<servlet-mapping>
    <servlet-name>applicationContext</servlet-name>
    <url-pattern>rest/*</url-pattern>
</servlet-mapping>
```

5.1.2 Underscore Templates

Backbone has one dependency, *Underscore.js*, in its web site [6] it is defined as a Javascript utility-belt library that provides functional programming support. Amongst them Javascript templating capabilities that compile the jacascript template as a function for evaluation and rendering. The underscore templates render HTML and JSON data sources.

5.2 Methodologies and Tools Used

5.2.1 Prototyping

The first step in the development was the creation of a prototype of the solution based on the `listWeekView` mentioned in Section 3.5.2 that includes the presentation aspects of the new PCE. It was decided to make a *Detailed Prototype* in the form of an HTML static webpage with Javascript interactions that would follow the visual aesthetics of the previous version of PCE. This would allow to compare the new and old PCE functionalities and use without the style of the presentation having an impact on it. However the opportunity was taken to clean the HTML code and the CSS. For example `<table>` tags that were used to create the layout of the web page were changed for `<div>` tags since tables are meant to show data and not to present layouts.

5.2.2 Object Oriented CSS

CSS has a very important roll when developing Single Page Web Applications, since it is through changes in style that it is possible for the user to see that the system is reacting to the actions made and where the actions can be made. A simple examples of this are for example a change in color and pointer when hovering over an HTML element that presents the possibility of interaction. For these reason it is important to keep the CSS code clean, maintainable and scalable. It was with this in mind that the principles of *Object Oriented CSS (OOCSS)* were followed.

As it is the case with *Object Oriented Programming*, *OOCSS* is based on the main concept of High Cohesion and Low Coupling, advocating for the creation of the different styles in a manner that emphasizes that each style has only one function. According to [30] the two main principles of *OOCSS* are *Separate the Structure from the Skin*, *Separate the Content from the Container*.

5.2.3 Testing Tools

QUnit [29] is a Javascript framework developed by the jQuery Fundation with the purpose of doing unit testing on their products. It was selected for this project due to its easy set up and the resemblance of its approach to unit testing with JUnit, the Java unit testing tool used by previous developers in DPG.

To debug the application Firefox [36] with Firebug [35] where used. Firebug provides the possibility of making changes on the fly to the code of the Javascript application and the CSS code making it possible to see the results right away.

5.3 Client Application

The client application follows the Content Model presented in Figure 4.3 in the previous chapter, however the implementation of it simplifies the diagram by unifying the structure and value classes. For example `FieldValue` is unified with `Field` and adds the attributes `plugin`, that specifies the plugin used to display the data and `name` that is used in conjunction to the `names` of the parent elements to form the `entity-path` value that is used by the server system as the data identifier. The final implmentation contains the following Backbone models:

- *ViewModel:* It represents the *View* element in Presentation Pattern and it is the root node for the application. As in the Presentation Pattern it contains one child *EntityInstanceModel* that is the starting point for the data tree.

- *EntityInstanceModel*: Contains a representation of its structure and the corresponding values.
- *FieldValueModel*: Represents the unit of information that will be interpreted by a plugin

In the case of *EntityInstanceModel* and *FieldValueModel* when used to provide only the structure of their respective data element, the value fields are empty. In addition to the mentioned models, for each plugin there exists an Underscore template that is used to interpret and present the data.

Listing 5.3 shows the render algorithm of the *FieldValueModel* and exemplifies the flexibility of the system.

Lines 1 through 5 check if a *plugin template* exists for the *plugin* type declared by `this.model`. If it does then said plugin is used, if it does not then the `app.templates.defaultBaseFieldTemplate` is used. Currently the default tamplate is the `StringTemplate`. This segregates the model form the presentation since the `value` of the `FieldValueModel` will be presented using the corresponding template.

Lines 11 through 16 insert the template and the data in the wrapper that contains the label of the field and line 18 adds the HTML code to the web page.

Listing 5.4 and 5.5 respectively display the templates for the String Plugin and the FieldValueView.

Listing 5.3: Render implementation of the FieldValueModel.

```
1  render: function(){
2     var templateName = this.model.get("plugin") + "Template";
3     if(typeof app.templates[templateName] === 'undefined'){
4        templateName = app.templates.defaultBaseFieldTempalte;
5     }
6
7     var template = _.template(app.templates.StringTemplate, this
          .model.attributes );
8
9     this.model.set('pluginTemplate', template);
10
11    var outerTemplate;
12    if (this.model.get("firstField")){
13       outerTemplate = _.template(app.templates.FirstField, this
             .model.attributes);
14    }else{
15       outerTemplate = _.template(app.templates.OtherField, this
             .model.attributes);
```

```
    }

    this.$el.html(outerTemplate);
}
```

Listing 5.4: Underscore template for the String Plugin

```
<div class="contentField stringBaseField"><%- value %></div>
```

Listing 5.5: Underscore template for the FieldValueView

```
<div class="firstField">
    <div class="field">
        <div id="extend_collapse" class="expanded"> </div>
        <div class="nameField"><%- label %>:</div>
        <%= pluginTemplate %>
    </div>
</div>
```

5.4 Server Application

The server application required three main changes to be implemented. First, POJOs representing the data needed by the client application were created. The most relevant POJO is `ClientViewModel`, this class represents the root object used by the client and contains the attributes GET attributes used to identify and locate the *view* in the system, such as `presentationId`, `viewId` and other attributes of the *view* needed for the presentation such as `name` and `description`. This class also contains a `ClientEntityInstanceModel` with its children `ClientEntityInstanceModel` and `ClientFieldValueModel`. In addition `ClientViewModel` has a list of the structure of all the *entity-instances* within the *view*

The second change in the server was to modify the `FormBuilder` class so that it would return all the *entity-instances* and *fields* of a *view*, plus the structure of the view. This step however was completely finalized in time and some work remains to be done. One of the disadvantages to this solution is that all the XML files for the different *entity-instances* have to be accessed while in the previous implementation of `FormBuilder` only one needed to be opened. This increments the process duration and difficulty in the server, however it is the same process followed by PV when displaying a *view*.

The third change was the creation of a RESTful controller, called `EditContentRestController` that handles the requests of the PCE SPA an example of one of the methods can be seen in Listing 5.6. This method is used to request the view.

Listing 5.6: getView method in the EditContentRestController

```
@RequestMapping(value="/getViewForm", method=RequestMethod.GET)
public @ResponseBody Form getView(HttpServletRequest request,
              @RequestParam String entityPath){
    ClientViewModel view = null;
    EditContentFormParameterGetter paramGetter = new
        EditContentFormParameterGetter();
    FormParameters formParameters = paramGetter.
        getFormParameters(request);
    formParameters.setEntityPath(entityPath);
    view = fullFormBuilder.createView(formParameters);
    return view;
}
```

6 Experience, Future Work and Conclusion

In this chapter the experience with the different tools and techniques is discusses and the possibilities for continuing improvement of the DPG system are presented.

6.1 Experience

Javascript, AJAX and Single Page Web Applications have been around for around 15 years and they have evolved during that time. Now a days we there are several options when it comes to choosing framewoks and technologies to work with. Backbone even though small in size and in the amount of modules that it offers, still presents the developer with enough tools to structure the code and it also allows the developer the flexibility of Javascript. It does not confine the possibilities of development to a strict structure which made it a good choice for this project. However this comes with the disadvantage that any small implementation can be done in one of several manners and sometimes it is time consuming to figure out which ones are the best.

The candidate did not have much previous experience with Javascript and its flexibility is a disadvantage to try to access the knowledge without guidance. There is no doubt though that it has open the doors to several opportunities of improving usability.

Coming from a programming background where things are usually defined in terms of 1 and 0 it is difficult and interesting to be confronted with the concept of usability. More and more the user is demanding things to be "easy" and understandable but what the user regards as "easy" changes with time and is not so straight forward to define. This proved to be an interesting challenge and a good experience.

6.2 Future work

6.2.1 Update client on server change

The new system, as does the original PCE, is not updated when changes are done on the server. Since the new client side part of the system is now an application on its own right it could be improved by adding the capability of refreshing the information in the client when the data is updated by another user to the server.

One way of implementing this would be to have a timer in the client that would trigger a check for new information in the server. The disadvantage would be the increase of the communication between server and the client.

6.2.2 Implement Cascading Search and Security Solution

The search functionality even though it was designed was not implemented and needs to be introduced in the system. This would improve the time needed to find the correct entity to work with. The three layer security solution should also be implemented in each of the plugins.

6.2.3 Use Modules to load Plugins

Currently the new PCE loads all the plugin templates and javascript component files as did the old implementation. These files are only needed if one of the *entity-instance* in the *view* has a *field* that uses said plugin.

There are two ways of implementing this, both require separating the templates into different files that could be loaded dynamically. The first solution would be to add the plugins code and templates used in the *view* within the structure sent to the client on page load. Once on the server the plugins are added to the `app.templates` variable.

The second solution would be implemented on the client side, with the SPA requestion the plguins from the server when they are needed. This approach would have a smaller payload on the first request but would require subsequent requests to the server each time the user expands an *entity-instance* that contains a *field* using a plguin that the SPA has not yet loaded.

6.2.4 Implement Plugins and PV in the client

The templates and Javascript code necessary for the plugins should also be implemented in PV. This could be done while maintaining the freedom provided by the XSLT transformation. The only difference would be that when transforming the XML data into HTML in place of the data itself, a reference to the plugin used and the id of the *entity-instance* would be added. The client application would load the data in the correct place with the corresponding template being used.

6.2.5 Integration of PCE and PV

PV is based on XSLT preventing code that is not in the transformation file to be added to the final HTML. It is for that reason that in the current implementation the XSLT creator needs to be aware that the system allows for the possibility of adding a link to PCE in order to connect both, but this is not always the case.

However, if PV was changed to the new implementation this would mean that a list of the *entity-instances* in the current *view* would be at the disposal of the Javascript application and if the current user has edition credentials it could provide this list for it to choose which *entity-instance* it wants to edit or create. The system could then present an overlay window with the necessary editable fields and the changes could be displayed on PV dynamically without a page reload.

6.3 Conclusion

The objective of this thesis was to enhance PCE, the path chosen to achieve this was to improve usability. This would make the system more attractive to the *publisher* user and in doing so encourage the creation of content. The improvements have been done taking into consideration recommendations and interaction design patterns in the literature. However, by creating the new SPA not only is the usability of PCE improved but it opens the door to a new approach for the plugin system, allowing a more flexible implementation since it now depends on Javascript code instead of the strict structures required by Java and static HTML. The plugin editing and viewing modes can behave independent of the server application and data structure.

Bibliography

[1]

[2] backbonejs.
`http://backbonejs.org/`. Accessed 2013.11.07.

[3] Introducing JSON.
`http://www.json.org`. Accessed 2013.11.13.

[4] Jackson json processor wiki.
`http://wiki.fasterxml.com/JacksonHome`. Accessed 2013.11.13.

[5] Todomvc.
`http://todomvc.com/`. Accessed 2013.11.07.

[6] underscorejs.
`http://underscorejs.org/`. Accessed 2013.11.07.

[7] Karianne Berg. Persistensproblematikk i dynamic presentation generator. Master's thesis, Department of Informatics, University of Bergen, 2008.

[8] ClickTale. Clicktale scrolling research report v2.0 part 1: Visibility and scroll reach.
`http://blog.clicktale.com/2007/10/05/clicktale-scrolling-research-report-v20-part-1-visibility-and-scroll-reach/`, 2007. Accessed 2013.10.02.

[9] Alan Cooper, Robert Reimann, and David Cronin. *About face 3: the essentials of interaction design*. John Wiley & Sons, 2007.

[10] Donald D Cowan and Carlos Jose Pereira de Lucena. Abstract data views: An interface specification concept to enhance design for reuse. *Software Engineering, IEEE Transactions on*, 21(3):229–243, 1995.

[11] Yngve Espelid. Dynamic presentation generator. Master's thesis, Department of Informatics, University of Bergen, 2004.

[12] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[13] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[14] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[15] Piero Fraternali, Gustavo Rossi, and Fernando Sánchez-Figueroa. Rich internet

applications. *Internet Computing, IEEE*, 14(3):9–12, 2010.

[16] A Garrido, G Rossi, and D Distante. Refactoring for usability in web applications. *Software, IEEE*, 28(3):60–67, 2011.

[17] Object Management Group. UML.
`http://www.uml.org`. Accessed 2013.11.08.

[18] Object Management Group. UML 2.0 Infrastructure.
`http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/`. Accessed 2013.11.08.

[19] Andreas Hjortland and Jø rgen Telles. Dynamic Presentation Generator , HP Fortify and ZAP. *INF226 project report*, Department of Informatics, University of Bergen, 2012.

[20] Christian Holst. Form-Field Validation: The Errors-Only Approach, 2012. Accessed 2013.08.14.

[21] infoScoop OpenSource. Json in java.
`http://www.json.org/java/`. Accessed 2013.11.13.

[22] infoScoop OpenSource. Xml2json.java.
`https://code.google.com/p/infoscoop/source/browse/branches/3.0/src/main/java/org/infoscoop/util/Xml2Json.java?r=629`. Accessed 2013.11.13.

[23] Bjørn Ove Ingvaldsen. Multimedia i dynamisk presentasjons generator 2.0. Master's thesis, Department of Informatics, University of Bergen, 2008.

[24] ISO. Ergonomic requirements for office work with visual display terminals (vdts) – part 11: Guidance on usability. ISO 9241-11, International Organization for Standardization, Geneva, Switzerland, 1998.

[25] itslearning AS. itslearning.
`https://www.itslearning.com/`. Accessed 2013.06.25.

[26] JDOM.
`http://www.jdom.org/`. Accessed 2013.06.26.

[27] The jQuery Foundation. Browser Support.
`http://jquery.com/browser-support/`. Accessed 2013.06.25.

[28] The jQuery Foundation. jQuery.
`http://jquery.com/`. Accessed 2013.06.27.

[29] The jQuery Foundation. QUnit.
`http://qunitjs.com/`. Accessed 2013.07.27.

[30] Louis Lazaris. An introduction to object oriented css (oocss).
`http://coding.smashingmagazine.com/2011/12/12/an-introduction-to-object-oriented-css-oocss/`, 2011. Accessed 2013.10.04.

[31] Ben Lieberman. UML activity diagrams: detailing user interface navigation. *The Rational Edge*, 2001.

[32] Marino Linaje, Juan Carlos Preciado, and Fernando Sánchez-Figueroa. Engineering

rich internet application user interfaces over legacy web models. *Internet Computing, IEEE*, 11(6):53–59, 2007.

[33] Kristian Sknberg Lø vik. Webucator 3.0 - Brukerhåndtering og aksesskontroll for DPG 2.0. Master's thesis, Universitet i Bergen, 2008.

[34] Ali Mesbah and Arie van Deursen. Migrating multi-page web applications to single-page ajax interfaces. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 181–190. IEEE, 2007.

[35] Mozilla. Firebug.
`https://getfirebug.com/`. Accessed 2013.07.10.

[36] Mozilla. Mozilla Firefox Web Browser.
`http://www.mozilla.org/en-US/firefox/central/`. Accessed 2013.07.15.

[37] Khalid A. Mughal et al. Presentation Patterns: Composing Web-based Presentations. Technical report, Department of Informatics, University of Bergen, 2003.

[38] University of Bergen. Mi Side.
`https://miside.uib.no/`. Accessed 2013.06.25.

[39] Oracle. Java Persisence API.
`http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`. Accessed 2013.06.26.

[40] Tim O'reilly. What is web 2.0: Design patterns and business models for the next generation of software. *Communications and strategies*, (1):17, 2007.

[41] Addy Osamani. Journey through the javascript mvc jungle.
`http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/`, 2012. Accessed 2013.11.07.

[42] OWASP. Category:OWASP AntiSamy Project.
`https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project`. Accessed 2013.11.10.

[43] OWASP. Cross-site Scripting (XSS).
`https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29`. Accessed 2013.10.17.

[44] OWASP. OWASP Zed attack Proxy Project.
`https://www.owasp.org/index.php/OWASP/Zed/Attack/Proxy/Project/`. Accessed 2013.06.25.

[45] Hewlett Packard. HP Fortify Static Code Analyzer.
`http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812`. Accessed 2013.06.25.

[46] Linda Dailey Paulson. Building rich web applications with ajax. *Computer*, 38(10):14–17, 2005.

[47] Ana G Pino and Anne Elise Weiss. Security Analysis of Dynamic Presentation Generator. *INF226 project report*, Department of Informatics, University of Bergen, 2012.

[48] Y. Rogers, H. Sharp, and J. Preece. *Interaction Design: Beyond Human - Computer Interaction.* Wiley, 2011.

[49] Gustavo Rossi, Matias Urbieta, Jeronimo Ginzburg, Damiano Distante, and Alejandra Garrido. Refactoring to rich internet applications. a model-driven approach. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 1–12. IEEE, 2008.

[50] Steven Sanderson. Rich javascript applications the seven frameworks (throne of js, 2012).
`http://blog.stevensanderson.com/2012/08/01/rich-javascript-applications-the-seven-frameworks-throne-of-js-2012/`, 2012. Accessed 2013.11.08.

[51] Bill Scott and Theresa Neil. *Designing Web Interfaces: Principles and Patterns for Rich Interactions.* O'Reilly Media, Incorporated, 2009.

[52] Bjørn Christian Sebak. Dynamic Presentation Generator 2.0 – Utvikling av ny dynamisk presentasjonsgenerator og presentasjonsmønsterspesifikasjon. Master's thesis, Department of Informatics, University of Bergen, 2008.

[53] Peder Lå ng Skeidsvoll. Støtte for rike klienter i DPG. Master's thesis, Department of Informatics, University of Bergen, 2010.

[54] Mikito Takada. Single page apps in depth.
`http://singlepageappbook.com/`. Accessed 2013.10.30.

[55] Milissa Tarquini. Blasting the myth of the fold.
`http://boxesandarrows.com/blasting-the-myth-of-the-fold/`, 2007. Accessed 2013.10.02.

[56] Kelly A Teigland Whiteley. Resource management for plugins in the dynamic presentation generator, 2011.

[57] Aleksander Vines. Sikker Arkitektur for Innholdshåndteringssystemer. Master's thesis, Department of Informatics, University of Bergen, 2013.

[58] W3C. Cascading Style Sheets home page.
`http://www.w3.org/Style/CSS/`. Accessed 2013.10.29.

[59] W3C. Document Object Model (DOM).
`http://www.w3.org/DOM/`. Accessed 2013.10.29.

[60] W3C. HTTP - Hypertext Transfer Protocol.
`http://www.w3.org/Protocols/`. Accessed 2013.10.29.

[61] W3C. Xml path language (xpath) version 1.0.
`http://www.w3.org/TR/xpath`. Accessed 2013.10.02.

[62] W3C. XMLHttpRequest.
`http://www.w3.org/TR/XMLHttpRequest/`. Accessed 2013.10.29.

[63] W3C. XSL Transformations (XSLT) Version 1.0.
`http://www.w3.org/TR/xslt`. Accessed 2013.06.26.

[64] W3C. What is the Document Object Model?
`http://www.w3.org/TR/DOM-Level-2-Core/introduction.html`, 2000.

Accessed 2013.10.29.

[65] Luke Wroblewski. Developing the invisible. `http://www.uxmatters.com/mt/archives/2006/05/developing-the-invisible.php`, 2006. Accessed 2013.10.02.

[66] Luke Wroblewski. Inline validation in web forms. `http://alistapart.com/article/inline-validation-in-web-forms`, 2009. Accessed 2013.08.14.