Entwurfsbeschreibung zum SmartPortalWiki

Inhaltsverzeichnis

3. Struktur- und Entwurfsprinzipien des Gesamtsystems

Bei der Anforderungsanalyse konnten wir 4 relativ unabhängige Funktionseinheiten bestimmen. Daraus haben wir folglich auch die 4 Grundpakete deriviert. Zum Einen sind dies die verschiedenen Parserklassen, die aus verschiedenen Content-Eingaben String-Ausgaben in dem von uns gewünschten Format generieren.
Um Lese- und Schreiboperationen auf das Dateisystem durchzuführen um geänderte oder erstellte Inhalte anzulegen, wird ein entsprechendes Paket benötigt, das gewünschte Funktionalitäten implementiert.
Innerhalb des Paketes core befindet sich die wichtigste Verbindung zwischen diesen bisher relativ unabhängigen Paketen. Sie alle werden nämlich alle benötigt, um die im core enthaltene Page als Repräsentation unserer Programmlogik zu generieren. Ausserdem ist hier auch die Search-Engine eingefügt.
Das Paket portlet schließlich enthält unsere eigentliche Portletimplementierung, die die Funktionalität unseres Produktes in die standardisierte Portlet-Struktur mit ihren vorgegebenen Methoden einbindet.
Für die Darstellung unserer Inhalte haben wir uns gegen die Verwendung von Velocity und stattdessen für JavaServerpages entschieden. Unsere Vorbehalte gegenüber Velocity liegen v. A. darin begründet, dass Velocity standardmässig nicht in Jetspeed implementiert ist. Innerhalb des elate Portals existiert zwar eine entsprechende Anbindung, im Hinblick auf die Portierbarkeit wäre es jedoch nicht ratsam, diese in unserem Projekt zu verwenden.

Struktur- und Entwurfsprinzipien einzelner Pakete:

spwiki.parser

XMLParser

Der Parser liest den Content eines bestimmten Tags einer XML-Datei aus und liefert diesen als String zurück
Begründung:

Der XMLParser ist unabdingbar, da die Speicherung in XML explizit gefordert ist und somit auch das Lesen und Parsen entsprechender Datenbestände eine Grundfunktionalität des Produktes darstellt. Aufgrund des Umfangs und der mangelnden Beziehungen zu anderen Funktionen empfiehlt es sich, diesen Parser als Klasse zu realisieren.

WMLParser

Der WikiMarkupLanguage Parser wird auf eingegebene Strings angewandt und übersetzt diese in HTML Text mit entsprechenden Formatierungen, welche dann später in der entsprechenden .vm Datei enthalten sind. Insbesondere HTML Tags sowie die Zeichen "<" und ">" sollen durch entsprechende Symbole ohne syntaktische Doppeldeutigkeit ersetzt werden. Effektiv sollen dann nur die entsprechenden Elemente der WikiMarkupLanguage in die entsprechenden HTML Tags übersetzt werden.
Begründung:

Der WMLParser ist eine eigene Klasse, weil er auf verschiedene Klassen

angewendet werden soll. Es dient der besseren Strukturierung und Lesbarkeit des Codes wenn solche eindeutig von der restlichen Programmlogik abgekapselten Funktionen ausgelagert werden.

URLParser

Durch die Übergabe eines Entitätsnamens als Parameter soll der URLParser einen Pfadnamen für die mit der Entität korrespondierende XML-Datei liefern.

Begründung:

Der URLParser ist unabhängig von anderen Klassen (außer der generischen Parser Klasse) und stellt Funktionalitäten bereit, die immer gebraucht werden, jedoch unabhängig von Pages oder der Pagerepräsentation zu sehen sind. Deshalb empfiehlt sich die Implementierung als eigene Klasse.

Parser (abstract)

Alle Parser befinden sich aus bereits beschriebenen Gründen in diesem Paket

Begründung:

In unserem Projekt werden verschiedene Parser verwendet.Alle Parser erben von der abstrakten Parser Klasse, da alle Parser ähnliche Funktionalitäten besitzen (aus Content-Eingaben Strings mit ausgelesenen Informationen erstellen), diese jedoch unterschiedlich realisieren.

spwiki.io

FileManager

FileManager realisiert alle Funktionen zur Bearbeitung aller Dateien. Folglich stellt dieser Methoden zum Lesen- und Schreiben in XML Dateien bereit.

Begründung:

Der FileManager implementiert Methoden, die von verschiedenen Komponenten unseres Wiki-Systems genutzt werden. Um einfache Zugriff auf diese Funktionalitäten zu gewähren und um die Unabhängigkeit vom gesamtsystem zu unterstreichen haben wir uns für die Implementation als Klasse entschieden.

spwiki.core

Page

Zur Präsentation einer Seite oder auch zur Darstellung von Fehlermeldungen wird ein Page Objekt erstellt. Alle Lese- und Schreiboperationen auf eine Entität werden auf diesem Objekt durchgeführt. Fehlermeldungen (wenn beispielsweise ein angefragter Link nicht existiert) werden auch als Page Objekt realisiert, deren Inhalt jedoch nicht aus einer XML Datei ausgelesen wird.

Begründung:

Die Page ist das zentrale Objekt unseres Produktes. Da alle Operationen auf sie angewendet werden und sie einen eindeutig beschreibbaren Inhalt und Aufbau besitzt, empfiehlt sich die Implementation als Klasse.

SearchEngine

SearchEngine implementiert die Funktionen Titelsuche und Volltextsuche. Die Titelsuche greift nur auf die struktur.xml Datei über unseren XMLParser zu und versucht, innerhalb der Titelstrings Entsprechungen zu finden. Volltextsuche hingegen überprüft alle XML Dateien mit Pageinhalten. Als Ergebnis werden in beiden Fällen Arrays zurückgeliefert, die die Pagetitel sowie die korrespondierende URL enthalten. Später werden diese in einer .jsp Datei als Ergebnisse zur Ausgabe in einem Page Objekt formatiert.

Begründung:

Auch die SearchEngine ist elementar und wird in verschiedenen Zusammenhängen benötigt. Sie bildet in sich einen eindeutig abgegrenzten Funktionsumfang und kann daher als Klasse implementiert werden.

spwiki.portlet

SPWikiPortlet

SPWikiPortlet enthält als Attribut die Page über welche alle Darstellungen nach Außen hin realisiert werden. Dies geschieht mittels der portletspezifischen doView() Methode.
Eine geänderte Darstellung setzt geänderte Inhalte voraus. Diese Änderungen innerhalb der Daten/des Modells werden durch die processAction() Methode verarbeitet.
Begründung:

Da wir ein Portlet entwickeln sollen muss selbstverständlich auch eine Portlet-Klasse existieren, die entsprechende Methoden nach Portlet Standard bereitstellt (doView() etc)

Dynamisches Modell