LifePin

Liferay Pinboard System
Architectural Report

Strumpflohner Juri (3195)

Advanced Web Programming
Free University of Bozen-Bolzano

June 25, 2009

Contents

1 Introduction

LifePin is a simple pinboard system designed to be deployed as a portlet on the Liferay portal server. It allows users to communicate in an easy fashion between each other by writing posts and comments that are shown on LifePin.
The name stands for “Life” for Liferay and “Pin” for Pinboard system.

2 Use Case View

This section first outlines the main actors of the LifePin application and then shows the major functionalities with a more detailled description.

2.1 Actors

The following are the main actors that use the LifePin application.

- Guest
  The guest user can be any arbitrary user that visits the portal website containing the LifePin portlet. The user cannot be identified since it either isn't logged in or it doesn't yet have an account.
  Guest users are limited to reading messages. They have no possibility to add, remove or leave comments on LifePin entries.

- LifePin User
  The LifePin user is a user that has a valid account on the portal server and is registered and authenticated. It has rights to create new entries, edit entries written by himself, add comments to LifePin messages and delete his comments and written messages.

- Administrator/Power User
  The administrator / power user is a specific user of the portal that has the role of an "Administrator" and/or "Power user" associated. He can act just like a normal LifePin user but in addition he has augmented rights to edit messages of other users as well as delete their comments or messages.

- LifePin System
  This is a special actor representing the LifePin system. This actor executes operations behind the scene such as cleaning expired LifePin messages.

2.2 Use Case Diagram

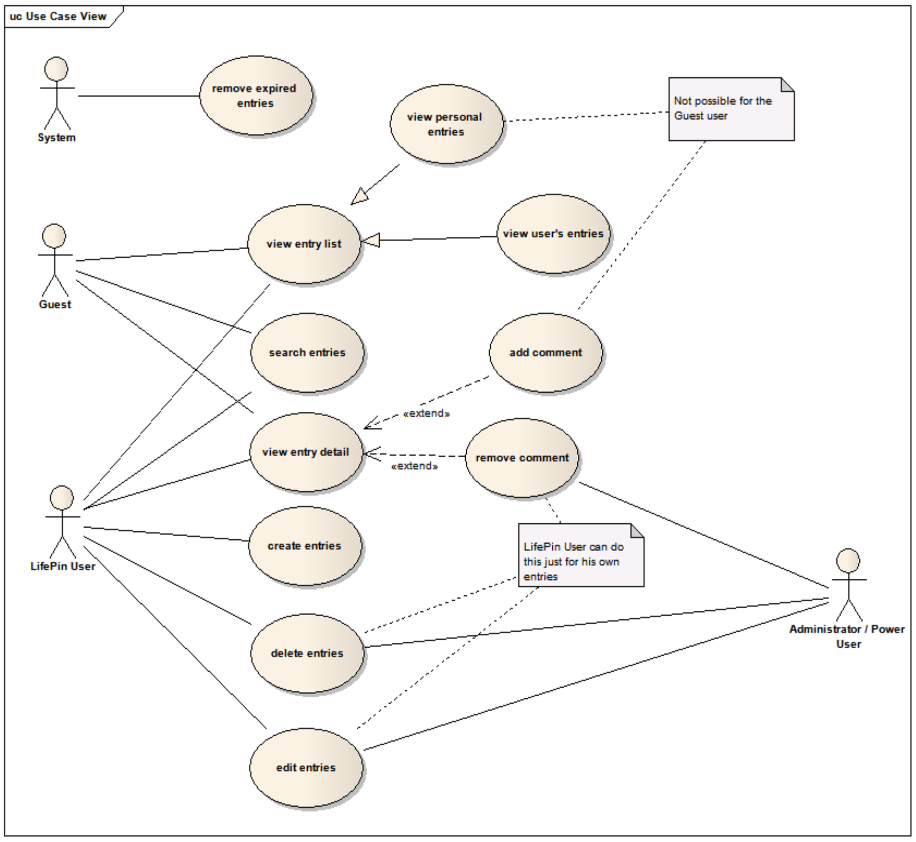

Figure 1: Use case diagram

2.3 Use Case Descriptions

It follows a detailled description of the single use cases.

Name	**Create entries**
Description	The user can create new LifePin entries. These entries will then be displayed on LifePin's list of entries.
Actors	• The LifePin user.
Special requirements	*none*
Preconditions	• The LifePin user must be registered to the portal. • The LifePin user must be authenticated.
Postconditions	• The entry created by the user appears on the top position of LifePin's entry list. • The user receives a notification message about the successful creation.
Basic flow	The use case has the following flow: • The user authenticates himself on the portal with his username and password • On the main interface showing the list of all entries, the user clicks on the link for creating a new entry. • The user fills out the form with the desired information. • If the entered information is valid, the user can submit the request for creating the entry.
Alternative flow	The user fills the form with invalid data: • A red error message appears just below the field containing the invalid data, notifying the user about the problem.

Table 1: Use case: Create entries

Name	**View entry list**
Description	The user can view the list of all available LifePin messages
Actors	• The Guest user • The LifePin user
Special requirements	*none*
Preconditions	*none*
Postconditions	• The user sees the list of all available messages in descending order according to their creation date (newest on top position).
Basic flow	The use case has the following flow: • The user views the LifePin portlet and sees the list of all available messages.
Alternative flow	*none*

Table 2: Use case: View entry list

Name	**View personal entries**
Description	The user can view all the entries that have been written by himself.
Actors	• The LifePin user
Special requirements	*none*
Preconditions	• The LifePin user must be registered to the portal. • The LifePin user must be authenticated.
Postconditions	• The user sees a list of all the messages that have been written by himself.
Basic flow	The use case has the following flow: • The user authenticates himself on the portal with his username and password • On the main interface showing the list of all entries, the user clicks on the link for retrieving his personal entries.
Alternative flow	The user has not written any entries yet: • A message will be shown notifying the user that no entries have been found

Table 3: Use case: View personal entries

Name	**View user's entries**
Description	The user can view entries written by a specific LifePin user.
Actors	• The Guest user • The LifePin user
Special requirements	*none*
Preconditions	*none*
Postconditions	• The list of entries written by the selected user are being displayed
Basic flow	The use case has the following flow: • The LifePin user or Guest user visit the LifePin application. • The user selects an entry of his interest • On the detail view he can click on the appropriate link just below the user's portal avatar for viewing the user's written messages.
Alternative flow	The selected user has not written any entries yet: • A message will be shown notifying the user that no entries have been found

Table 4: Use case: View user's entries

Name	**View entry detail**
Description	The user can view the details of a LifePin message such as its title, content, the associated comments, the user which wrote it etc...
Actors	• The Guest user • The LifePin user
Special requirements	*none*
Preconditions	*none*
Postconditions	• The user sees the details of the LifePin's message he's viewing
Basic flow	The use case has the following flow: • The user selects the message of his interest on the main LifePin's interface showing the list of all available entries. • A new view opens showing the details of the selected entry.
Alternative flow	*none*

Table 5: Use case: View entry detail

Name	**Add comment**
Description	When viewing the details of a LifePin message, the user has the possibility to interact with the author of the message by leaving a comment.
Actors	• The LifePin user
Special requirements	*none*
Preconditions	• The LifePin user must be registered to the portal. • The LifePin user must be authenticated. • The user must be on the detail view of a given LifePin message.
Postconditions	• The comment written by the user appears at the end of possibly already existing comments. • The user receives a notification message about the successful creation of his comment.
Basic flow	The use case has the following flow: • The user authenticates himself on the portal with his username and password • On the main interface showing the list of all entries, the user selects the entry of his interest. • In the detail view, the user clicks on the appropriate link for adding a new comment. • On the appropriate form, the user enters the content of his comment. • If the entered information is valid, the user can submit the request for adding the new comment.
Alternative flow	The user fills the comment form with invalid data: • A red error message appears just below the field containing the invalid data, notifying the user about the problem.

Table 6: Use case: Add comment

Name	**Remove comment**
Description	The user can remove a comment he has previously added to a LifePin message.
Actors	• The LifePin user • The Administrator/Power user
Special requirements	*none*
Preconditions	• The LifePin/Administrator/Power user must be registered to the portal. • The LifePin/Administrator/Power user must be authenticated. • The LifePin/Administrator/Power must be on the detail view of a given LifePin message.
Postconditions	• The deleted comment is no more visible on the list of comments of the current LifePin entry. • The user receives a notification message about the successful deletion of the comment.
Basic flow	The user case has the following flow: • The LifePin/Administrator/Power user authenticates himself on the portal with his username and password • On the main interface showing the list of all entries, the LifePin/Administrator/Power user selects the entry of his interest. • In the detail view, the LifePin/Administrator/Power user selects the comment he wants to delete by clicking on the appropriate link.
Alternative flow	The current LifePin message has no comments: • The links for deleting comments will not be shown.

Table 7: Use case: Remove comment

Name	**Delete entries**
Description	The user can delete a LifePin message he has previously written.
Actors	• The LifePin user • The Administrator/Power user
Special requirements	*none*
Preconditions	• The LifePin/Administrator/Power user must be registered to the portal. • The LifePin/Administrator/Power user must be authenticated.
Postconditions	• The deleted LifePin message is no more visible on the list of entries on the main LifePin interface. • The user receives a notification message about the successful deletion of the entry.
Basic flow	• The LifePin/Administrator/Power user authenticates himself on the portal with his username and password From this point there are two different flows: • The user starts from the main list of all LifePin entries – The user identifies the entry he wants to delete and clicks on the appropriate link. • The user starts from the detail view of a specific message – The user clicks on the appropriate link for deleting the entry
Alternative flow	*none*

Table 8: Use case: Delete entries

Name	**Edit entries**
Description	The user can modify the details of his message at any time.
Actors	• The LifePin user • The Administrator/Power user
Special requirements	*none*
Preconditions	• The LifePin/Administrator/Power user must be registered to the portal. • The LifePin/Administrator/Power user must be authenticated.
Postconditions	• The changes are applied on the existing LifePin message. • The existing LifePin message is still visible on the main page, reflecting the applied modifications.
Basic flow	• The LifePin/Administrator/Power user authenticates himself on the portal with his username and password From this point there are two different flows: • The user starts from the main list of all LifePin entries – The user identifies the entry he wants to edit and clicks on the appropriate link. • The user starts from the detail view of a specific message – The user clicks on the appropriate link for editing the entry
Alternative flow	*none*

Table 9: Use case: Edit entries

Name	**Search entries**
Description	The user can search for the title and content of existing LifePin messages. The search may allow simple OR and AND concatenated queries.
Actors	• The Guest user • The LifePin user
Special requirements	*none*
Preconditions	*none*
Postconditions	• The user sees a list of matching LifePin messages.
Basic flow	The use case has the following flow: • The LifePin/Guest user types the desired search query on the appropriate text field on the main LifePin interface. • After entering the search query he submits it.
Alternative flow	No entries match the given query: • A notification will be shown, indicating that no entries have been found with the given query

Table 10: Use case: Search entries

Name	**Remove expired entries**
Description	The system automatically deletes all entries that are no more valid according to the expiry date specified by the user.
Actors	• The System
Special requirements	*none*
Preconditions	*none*
Postconditions	• Expired LifePin messages have been removed and are no more visible to the users on the LifePin user interface.
Basic flow	The user case has the following flow: • The LifePin system checks in regular intervals for expired posts • All expired messages are removed permanently.
Alternative flow	No expired messages have been found: • Nothing will be done; the system waits for the next scheduled check.

Table 11: Use case: Remove expired entries

3 Infrastructural View

3.1 Development tools

The following table gives an overview of the major libraries and tools that have been used for the development.

Name	**Description**	**URL**
Netbeans 6.5	Main development ID	`http://www.netbeans.org/downloads/`
Liferay portal server	Portal- / Application server	`http://www.liferay.com/web/guest/home`
Spring	Web application framework	`http://www.springsource.org/`
Hibernate	ORM, persistency framework	`https://www.hibernate.org/`
PostgreSQL	Database server	`http://www.postgresql.org/`
DisplayTag	Used for displaying data with paging functionality	`http://displaytag.sourceforge.net/1.2/`
jQuery	JavaScript library used for enhancing the user experience	`http://jquery.com`
JUnit	Unit testing framework	`http://www.junit.org`

Table 12: Libraries and tools used for development

3.2 Application components

Figure 2 shows the main application components.

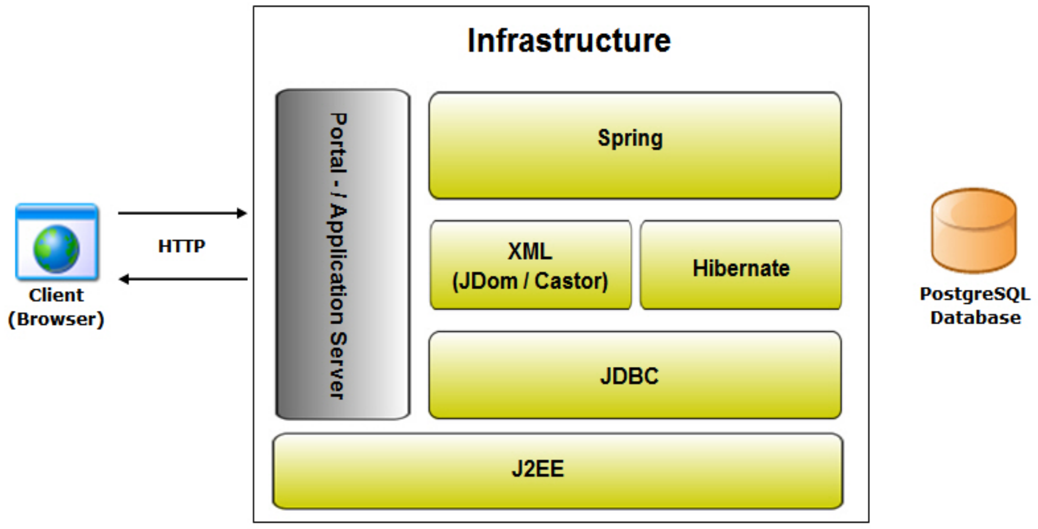

Figure 2: Infastructure diagram

3.2.1 Liferay Portal- / Application server

The Liferay portal is the hosting system of the LifePin application. It handles the basics of the portlet's lifecycle. Moreover it provides useful functionalities such as user authentication and user information as well as UI utilities.

Details	
Version	5.2.1

3.2.2 Spring Web MVC framework

LifePin uses the Spring Web MVC framework as main framework for development. Spring supports the development through all the different application layers by using the IoC container and the dependency injection pattern for facilitating loosely coupling among the different objects. In specific it provides for the..

- ..**presentation layer** the MVC construct that allows to define HTTP handlers that map HTTP requests to specific controllers. In addition validators can be attached for user input validation.
- ..**data access layer** an integration of the Hibernate ORM tool and declarative transactions management.

Details	
Spring version	2.5.5
Spring Web MVC version	2.5.5
Spring Web MVC portlet version	2.5.5
Presentation layer configuration	lifepin-portlet.xml
Service layer configuration	lifepin-service.xml
Data access layer configuration	lifepin-data.xml

3.2.3 Hibernate ORM framework

Hibernate is the object relational mapping framework that has been used for mapping the application entities to the according relational database tables. Spring's HibernateDaoSupport has been used to integrate Hibernate into the application.

Details	
Hibernate Version	3.2.0.cr4
Entity mapping files location	/docroot/WEB-INF/classes/com/lifepin/entities
Entity mapping files	• PinboardEntry.hbm.xml • Comment.hbm.xml

3.2.4 PostgreSQL database

PostgreSQL has been chosen as the underlying database system for storing data persistently.

Details	
DBMS Version	8.3
Database schema	awpdb
Username	unibz
Password	unibz

4 Logical View

4.1 Application layering components

The application is designed in three major layers:

- Presentation layer
  This layer contains the UI related components such as the different controllers that are resonsible for handling the request and response. Objects in this layer do not contain any business logic but they call the appropriate service objects for that purpose.

- Service layer
  The service layer exposes the applications' services containing the main business logic.

- Data Access layer
  The data access layer contains the according DAOs (Data Access Objects) for persisting data to the database. Hibernate is used as the object relational mapping tool for mapping the application's business entities to the according database tables.

- Transversal (layer)
  The "transversal layer" is not a real layer, but a component that is shared among all the other layers. It mainly contains the business entities that are used for transporting the data among the layers.

In order to further decouple the different layers, the access to each of them is clearly defined with interfaces (layers). Service classes as well as data access classes are never referenced directly but always through the according interfaces which define the contract for the communication. Figure 3 illustrates this further.

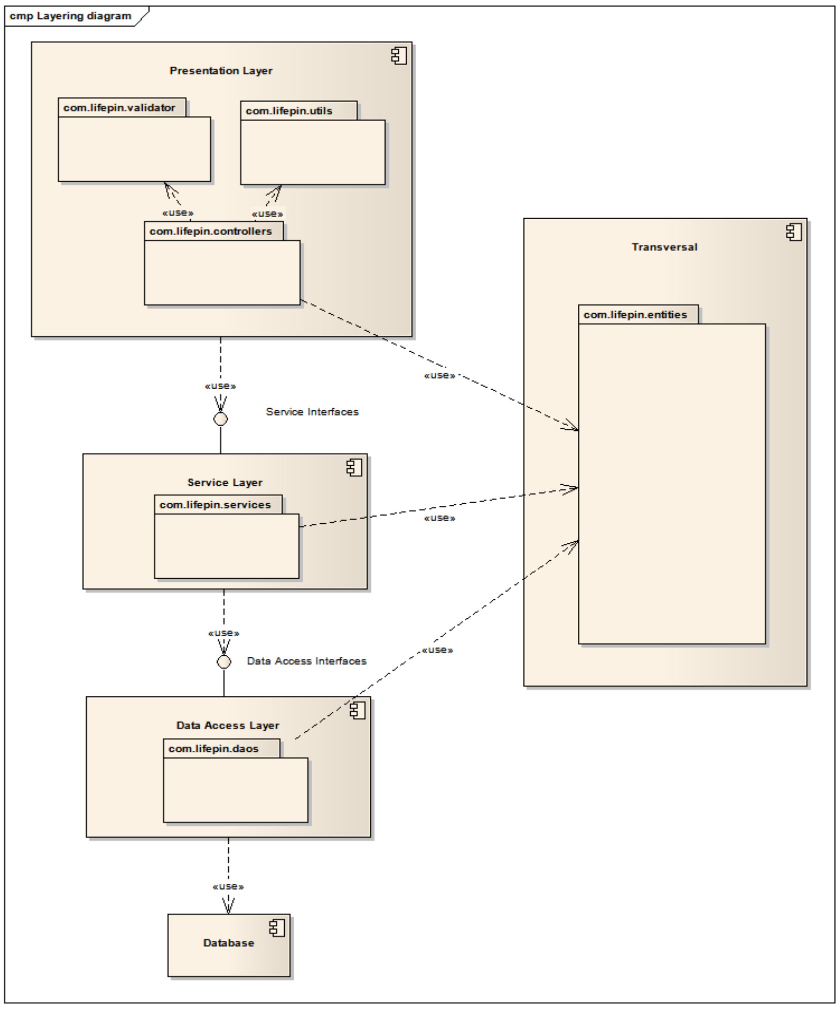

Figure 3: Application layers

4.2 Packages and class diagrams

The following diagram gives an overview of the packages and their relationships. A more detailed descripton of the single packages with the according classes grouped by layers follows in the subsequent sections.

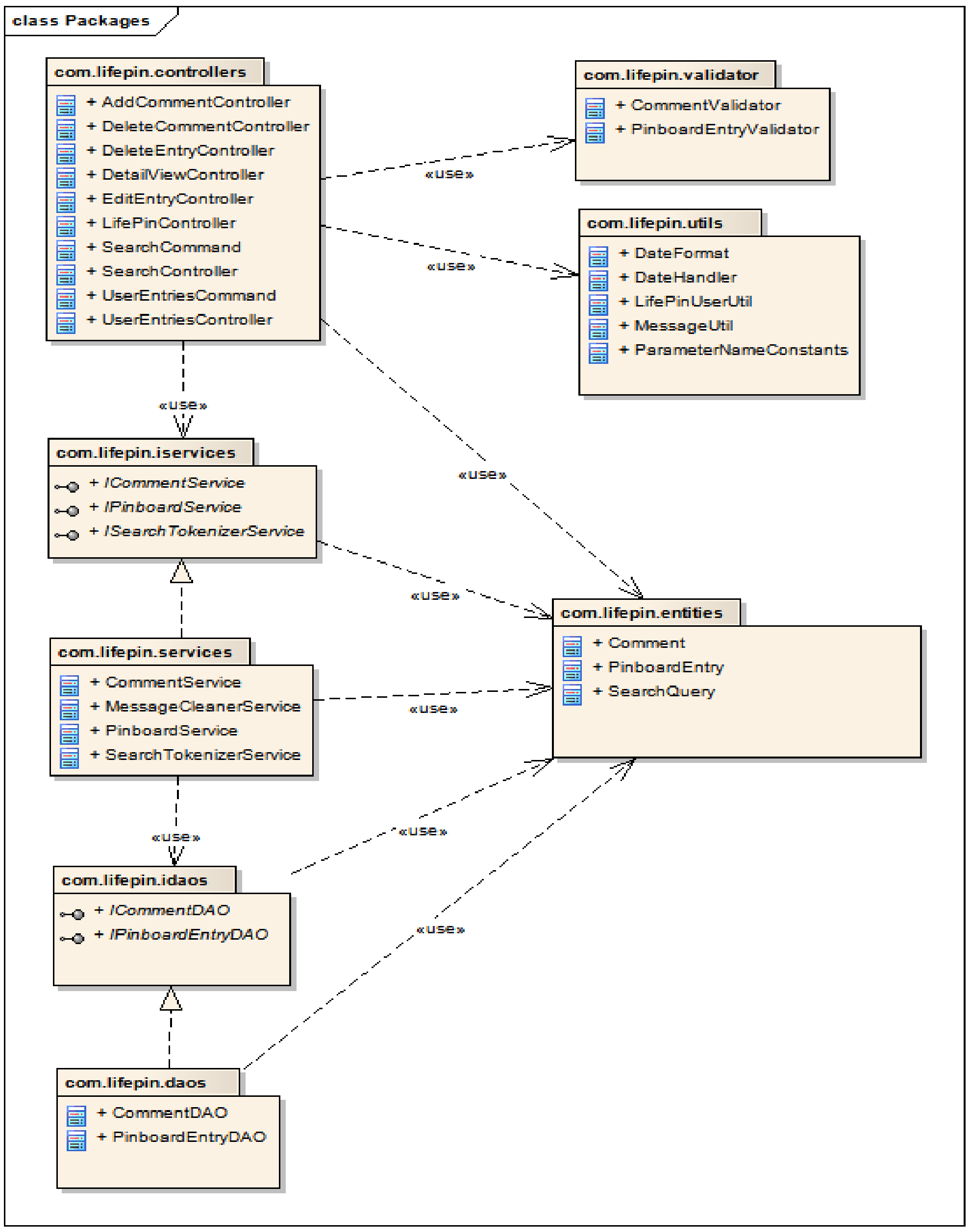

Figure 4: Packages diagram

4.3 Presentation Layer

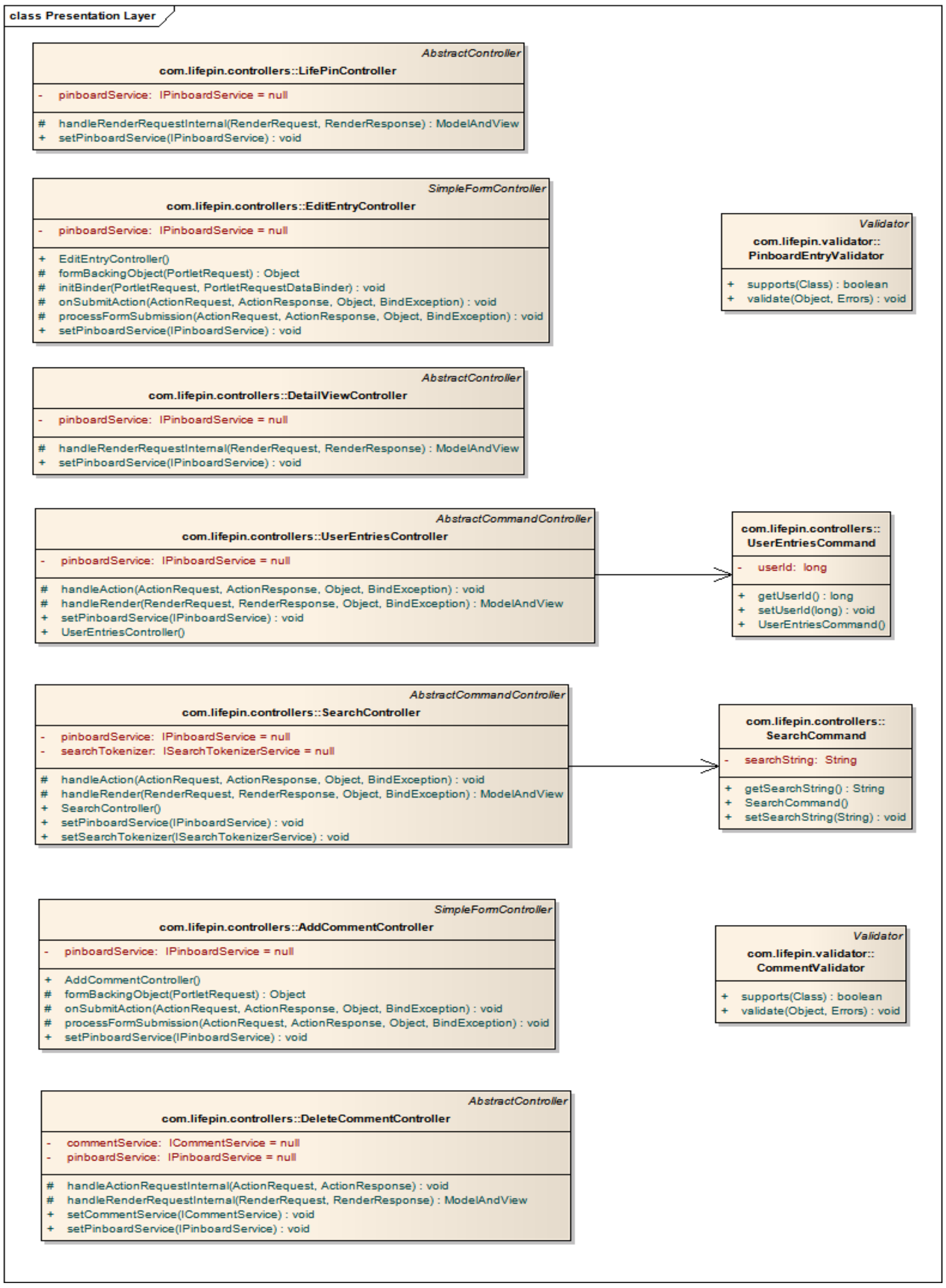
class Presentation Layer
AbstractController
com.lifepin.controllers::LifePinController
- pinboardService: IPinboardService = null
# handleRenderRequestInternal(RenderRequest, RenderResponse) : ModelAndView
+ setPinboardService(IPinboardService) : void
SimpleFormController
com.lifepin.controllers::EditEntryController
- pinboardService: IPinboardService = null
+ EditEntryController()
# formBackingObject(PortletRequest) : Object
# initBinder(PortletRequest, PortletRequestDataBinder) : void
# onSubmitAction(ActionRequest, ActionResponse, Object, BindException) : void
# processFormSubmission(ActionRequest, ActionResponse, Object, BindException) : void
+ setPinboardService(IPinboardService) : void
Validator
com.lifepin.validator::PinboardEntryValidator
+ supports(Class) : boolean
+ validate(Object, Errors) : void
AbstractController
com.lifepin.controllers::DetailViewController
- pinboardService: IPinboardService = null
# handleRenderRequestInternal(RenderRequest, RenderResponse) : ModelAndView
+ setPinboardService(IPinboardService) : void
AbstractCommandController
com.lifepin.controllers::UserEntriesController
- pinboardService: IPinboardService = null
# handleAction(ActionRequest, ActionResponse, Object, BindException) : void
# handleRender(RenderRequest, RenderResponse, Object, BindException) : ModelAndView
+ setPinboardService(IPinboardService) : void
+ UserEntriesController()
com.lifepin.controllers::UserEntriesCommand
- userId: long
+ getUserId() : long
+ setUserId(long) : void
+ UserEntriesCommand()
AbstractCommandController
com.lifepin.controllers::SearchController
- pinboardService: IPinboardService = null
- searchTokenizer: ISearchTokenizerService = null
# handleAction(ActionRequest, ActionResponse, Object, BindException) : void
# handleRender(RenderRequest, RenderResponse, Object, BindException) : ModelAndView
+ SearchController()
+ setPinboardService(IPinboardService) : void
+ setSearchTokenizer(ISearchTokenizerService) : void
com.lifepin.controllers::SearchCommand
- searchString: String
+ getSearchString() : String
+ SearchCommand()
+ setSearchString(String) : void
SimpleFormController
com.lifepin.controllers::AddCommentController
- pinboardService: IPinboardService = null
+ AddCommentController()
# formBackingObject(PortletRequest) : Object
# onSubmitAction(ActionRequest, ActionResponse, Object, BindException) : void
# processFormSubmission(ActionRequest, ActionResponse, Object, BindException) : void
+ setPinboardService(IPinboardService) : void
Validator
com.lifepin.validator::CommentValidator
+ supports(Class) : boolean
+ validate(Object, Errors) : void
AbstractController
com.lifepin.controllers::DeleteCommentController
- commentService: ICommentService = null
- pinboardService: IPinboardService = null
# handleActionRequestInternal(ActionRequest, ActionResponse) : void
# handleRenderRequestInternal(RenderRequest, RenderResponse) : ModelAndView
+ setCommentService(ICommentService) : void
+ setPinboardService(IPinboardService) : void

Figure 5: Presentation Layer classes (util classes have been left away to keep it simple)

4.3.1 Package "com.lifepin.controller"

Class	**Description**
LifePinController	This controller is responsible for showing the list of all available LifePin messages on the main LifePin user interface.
AddCommentController	This is the controller that handles the adding of new comments by LifePin users. It has a form behind with an appropriate text area for entering the comment and then redirects back to the detail view of LifePin messages.
DeleteCommentController	This controller handles the deletion of existing comments by a LifePin user or system administrator.
DeleteEntryController	This controller handles the deletion of exisiting LifePin entries by a LifePin user or system administrator.
DetailViewController	This controller is responsible for showing the details of a LifePin message to the user. For this purpose it invokes the according service class for retrieving the data and forwards it to the right detail JSP view.
EditEntryController	This controller handles requests for editing existing LifePin messages by a LifePin user or Administrator.
SearchCommand	This command class is used for binding the data entered by the user on the search text field. The bound data will then be consumed by the appropriate controller.

Class	**Description**
SearchController	The search controller is used for retrieving the data bound by the SearchCommand object and for then calling the appropriate service classes for performing a search for LifePin messages.
UserEntriesCommand	This command class is used for binding the user id.
UserEntriesController	This controller retrieves all LifePin messages written by a specific user that is identified by the id stored in the UserEntriesCommand class. Instead of using the UserEntriesCommand object, the according userid could also have been just passed over the URL.

4.3.2 Package “com.lifepin.validator”

Class	**Description**
CommentValidator	This is the validator used for validating the user input for a new comment.
PinboardEntryValidator	This validator validates the correctness of the data entered by the user such as validating the given expiry date or entered email address.

4.3.3 Package "com.lifepin.utils"

Class	**Description**
DateFormat	This enumeration is used for keeping the different date formats used throughout the application.
DateHandler	This is a utility class for creating date instances or formatting dates according to the formats specified in the DateFormat enumeration.
LifePinUserUtil	This utility class is a kind of wrapper that bundles all operations releated to retrieving the currently authenticated portal user. In this way this logic is not spread throughout the applicaton and can be changed more easily.
MessageUtil	This class is used for retrieving LifePin notification messages from the request object and for passing them to the according model or response to get them rendered on the according view. It is basically a kind of forwarding mechanism.
ParameterNameConstants	This class contains a set of string constants that are used for identifying request or response parameters. This avoids hardcoded strings throughout the whole application and makes it more easy to change parameter names if needed.

4.4 Service Layer

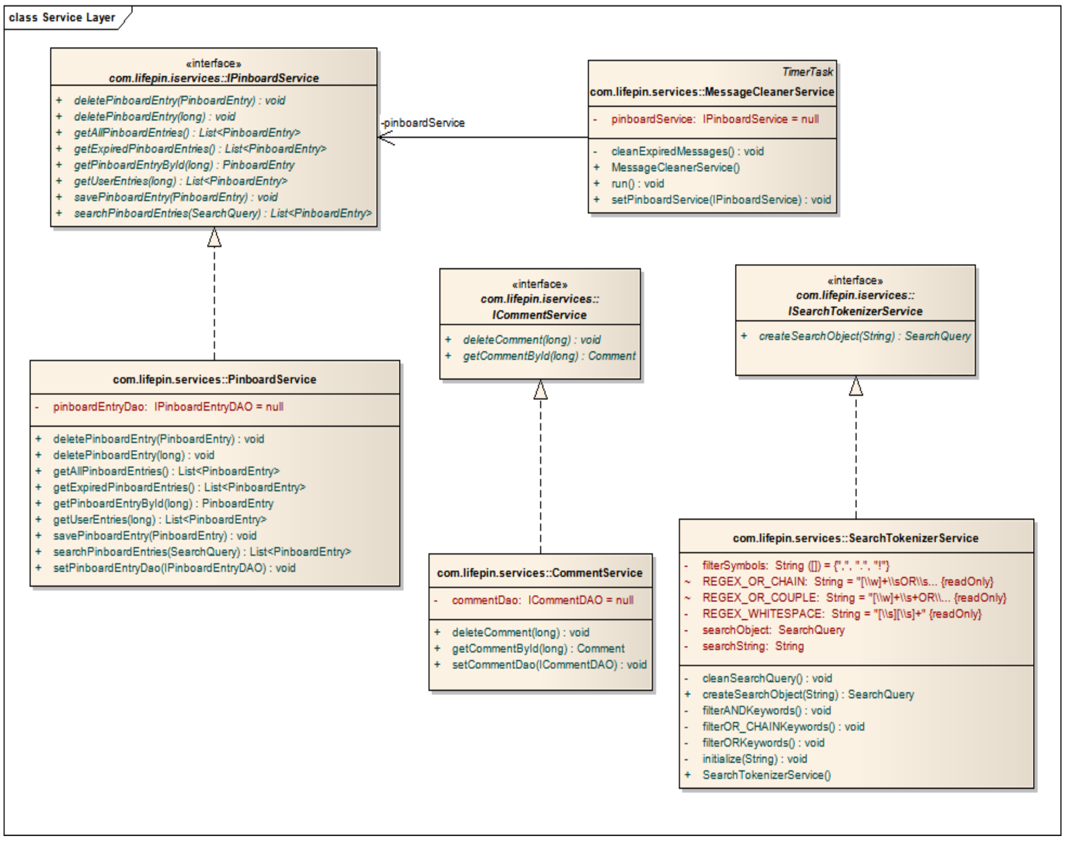
class Service Layer
«interface»
com.lifepin.iservices::IPinboardService
+ deletePinboardEntry(PinboardEntry) : void
+ deletePinboardEntry(long) : void
+ getAllPinboardEntries() : List<PinboardEntry>
+ getExpiredPinboardEntries() : List<PinboardEntry>
+ getPinboardEntryById(long) : PinboardEntry
+ getUserEntries(long) : List<PinboardEntry>
+ savePinboardEntry(PinboardEntry) : void
+ searchPinboardEntries(SearchQuery) : List<PinboardEntry>
-pinboardService
TimerTask
com.lifepin.services::MessageCleanerService
- pinboardService: IPinboardService = null
- cleanExpiredMessages() : void
+ MessageCleanerService()
+ run() : void
+ setPinboardService(IPinboardService) : void
«interface»
com.lifepin.iservices::ICommentService
+ deleteComment(long) : void
+ getCommentById(long) : Comment
«interface»
com.lifepin.iservices::ISearchTokenizerService
+ createSearchObject(String) : SearchQuery
com.lifepin.services::PinboardService
- pinboardEntryDao: IPinboardEntryDAO = null
+ deletePinboardEntry(PinboardEntry) : void
+ deletePinboardEntry(long) : void
+ getAllPinboardEntries() : List<PinboardEntry>
+ getExpiredPinboardEntries() : List<PinboardEntry>
+ getPinboardEntryById(long) : PinboardEntry
+ getUserEntries(long) : List<PinboardEntry>
+ savePinboardEntry(PinboardEntry) : void
+ searchPinboardEntries(SearchQuery) : List<PinboardEntry>
+ setPinboardEntryDao(IPinboardEntryDAO) : void
com.lifepin.services::CommentService
- commentDao: ICommentDAO = null
+ deleteComment(long) : void
+ getCommentById(long) : Comment
+ setCommentDao(ICommentDAO) : void
com.lifepin.services::SearchTokenizerService
- filterSymbols: String ([]) = {",", ".", "!"}
~ REGEX_OR_CHAIN: String = "[\\w]+\\sOR\\s... {readOnly}
~ REGEX_OR_COUPLE: String = "[\\w]+\\s+OR\\... {readOnly}
- REGEX_WHITESPACE: String = "[\\s][\\s]+" {readOnly}
- searchObject: SearchQuery
- searchString: String
- cleanSearchQuery() : void
+ createSearchObject(String) : SearchQuery
- filterANDKeywords() : void
- filterOR_CHAINKeywords() : void
- filterORKeywords() : void
- initialize(String) : void
+ SearchTokenizerService()

Figure 6: Service Layer classes

4.4.1 Package “com.lifepin.iservices”

Class	Description
ICommentService	This is the interface used for accessing the comment service that provides the according methods for retrieving comments by their id and for deleting existing comments. For the database related operations, the according DAO class is used.
IPinboardService	This is the interface to the PinboardService which provides functionalities for handling LifePin entries, i.e. retrieving all entries, storing/updating new/modified entries, deleting or searching for them.
ISearchTokenizerService	This is the interface for the SearchTokenizerService that is used for splitting concatenated “OR” or “AND” queries that are used for performing searches.
IMessageCleanerService	This interface is used for accessing the MessageCleanerService that gets invoked by a Spring timer task and cleans the expired messages from the DB.

4.4.2 Package "com.lifepin.services"

Class	Description
CommentService	This service class implements the operations specified by the ICommentService interface.
PinboardService	This service class implements the operations defined by the IPinboardService interface.
SearchTokenizerService	This service class implements the operations defined by the ISearchTokenizerService interface.
MessageCleanerService	This service class implements the operations defined by the IMessageCleanerService interface.

4.5 Data Access Layer

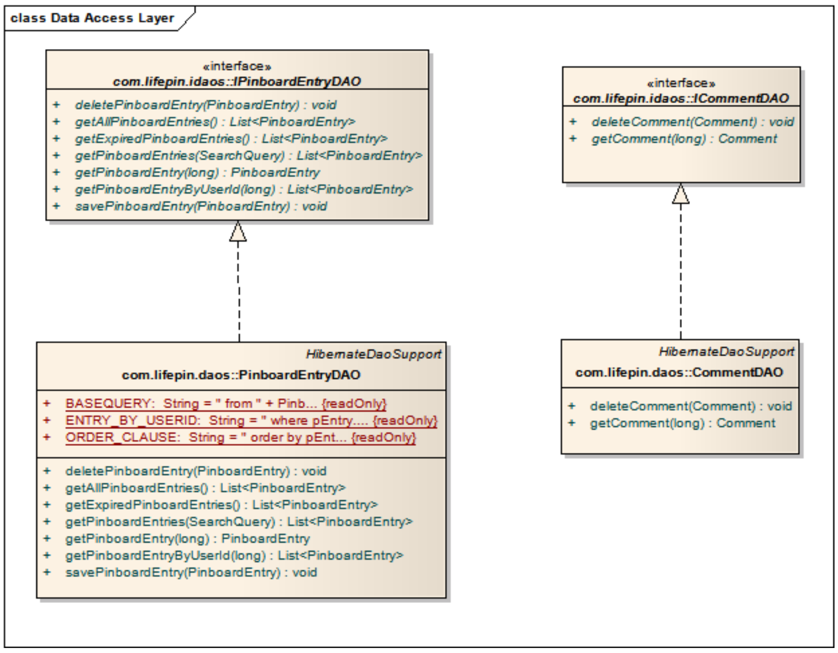
class Data Access Layer
«interface»
com.lifepin.idaos::IPinboardEntryDAO
+ deletePinboardEntry(PinboardEntry) : void
+ getAllPinboardEntries() : List<PinboardEntry>
+ getExpiredPinboardEntries() : List<PinboardEntry>
+ getPinboardEntries(SearchQuery) : List<PinboardEntry>
+ getPinboardEntry(long) : PinboardEntry
+ getPinboardEntryByUserId(long) : List<PinboardEntry>
+ savePinboardEntry(PinboardEntry) : void
«interface»
com.lifepin.idaos::ICommentDAO
+ deleteComment(Comment) : void
+ getComment(long) : Comment
HibernateDaoSupport
com.lifepin.daos::PinboardEntryDAO
+ BASEQUERY: String = " from " + Pinb... {readOnly}
+ ENTRY_BY_USERID: String = " where pEntry.... {readOnly}
+ ORDER_CLAUSE: String = " order by pEnt... {readOnly}
+ deletePinboardEntry(PinboardEntry) : void
+ getAllPinboardEntries() : List<PinboardEntry>
+ getExpiredPinboardEntries() : List<PinboardEntry>
+ getPinboardEntries(SearchQuery) : List<PinboardEntry>
+ getPinboardEntry(long) : PinboardEntry
+ getPinboardEntryByUserId(long) : List<PinboardEntry>
+ savePinboardEntry(PinboardEntry) : void
HibernateDaoSupport
com.lifepin.daos::CommentDAO
+ deleteComment(Comment) : void
+ getComment(long) : Comment

Figure 7: Data Access Layer classes

4.5.1 Package “com.lifepin.idaos”

Class	**Description**
ICommentDAO	This interface defines the operations exposed by the according CommentDAO class that is used for retrieving comments by their id or deleting existing comments.
IPinboardDAO	This is the interface for the PinboardDAO class that is used for all persistency-related operations for PinboardEntry objects such as retrieving/saving/deleting/searching PinboardEntry objects from the underlying DB.

4.5.2 Package “com.lifepin.daos”

Class	**Description**
CommentDAO	This data access object implements the operations defined by the ICommentDAO interface.
PinboardDAO	This data access object implements the operations defined by the IPinboardEntryDAO interface.

4.6 Transversal layer

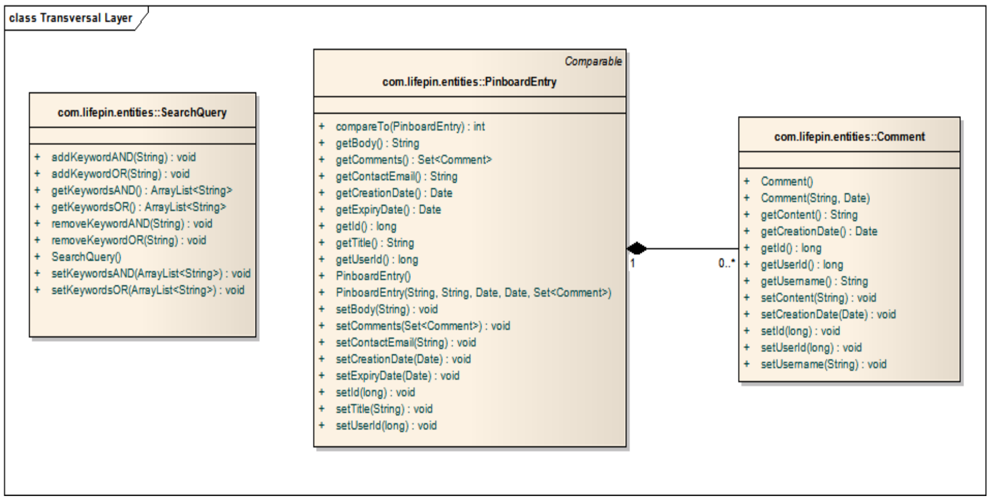
class Transversal Layer
com.lifepin.entities::SearchQuery
+ addKeywordAND(String) : void
+ addKeywordOR(String) : void
+ getKeywordsAND() : ArrayList<String>
+ getKeywordsOR() : ArrayList<String>
+ removeKeywordAND(String) : void
+ removeKeywordOR(String) : void
+ SearchQuery()
+ setKeywordsAND(ArrayList<String>) : void
+ setKeywordsOR(ArrayList<String>) : void
Comparable
com.lifepin.entities::PinboardEntry
+ compareTo(PinboardEntry) : int
+ getBody() : String
+ getComments() : Set<Comment>
+ getContactEmail() : String
+ getCreationDate() : Date
+ getExpiryDate() : Date
+ getId() : long
+ getTitle() : String
+ getUserId() : long
+ PinboardEntry()
+ PinboardEntry(String, String, Date, Date, Set<Comment>)
+ setBody(String) : void
+ setComments(Set<Comment>) : void
+ setContactEmail(String) : void
+ setCreationDate(Date) : void
+ setExpiryDate(Date) : void
+ setId(long) : void
+ setTitle(String) : void
+ setUserId(long) : void
1
0..*
com.lifepin.entities::Comment
+ Comment()
+ Comment(String, Date)
+ getContent() : String
+ getCreationDate() : Date
+ getId() : long
+ getUserId() : long
+ getUsername() : String
+ setContent(String) : void
+ setCreationDate(Date) : void
+ setId(long) : void
+ setUserId(long) : void
+ setUsername(String) : void

Figure 8: Transversal Layer classes

4.6.1 Package "com.lifepin.entities"

Class	Description
Comment	This is the business entity representing user comments.
PinboardEntry	This entity represents LifePin messages such as its owner, the message, creation date, expiry date etc...
SearchQuery	This entity represents a search query entered by the user and processed by the SearchTokenizerService class. It contains already the processed AND and OR concatenated keywords.

4.7 Database schema

The following figure shows the database schema that is used for storing the LifePin entries with its associated comments.

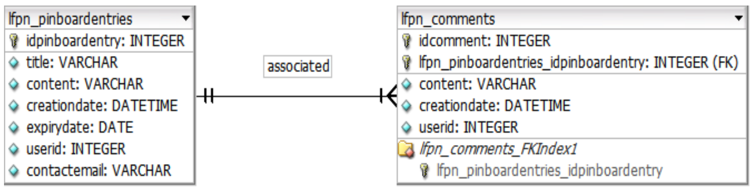

Figure 9: LifePin DB schema

4.8 Sequence diagrams / Scenarios

4.8.1 Creating a new LifePin message

The following sequence diagram shows the major parts that are involved when the user creates a new LifePin pinboard entry. The Dispatcher Portlet has been taken as a starting point of the application. The calls between the DispatcherPortlet and the EditEntryController don't have to be taken strictly. There are a couple of Spring objects and handlers in between that inject the service classes, forward the Http request etc.

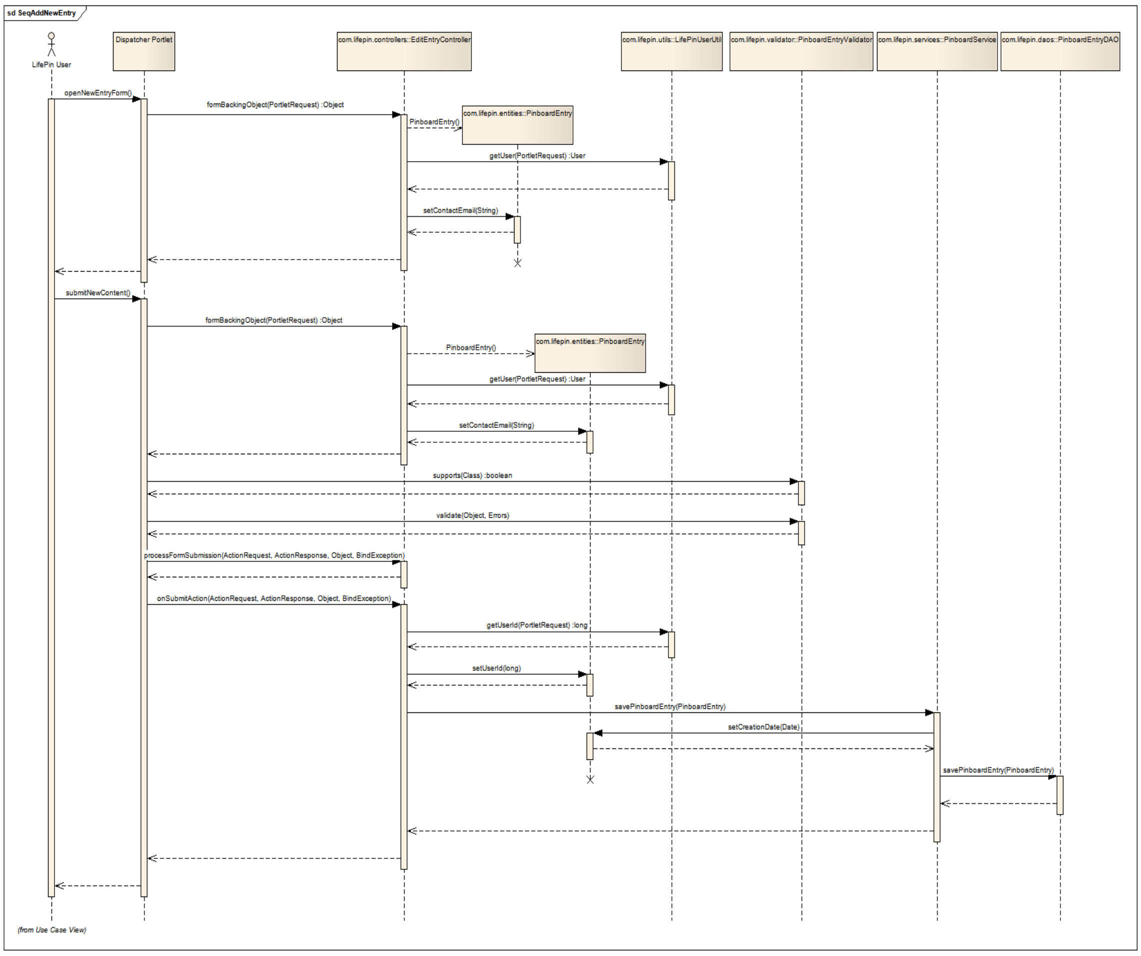
sd SeqAddNewEntry
LifePin User
Dispatcher Portlet
com.lifepin.controllers::EditEntryController
com.lifepin.utils::LifePinUserUtil
com.lifepin.validator::PinboardEntryValidator
com.lifepin.services::PinboardService
com.lifepin.daos::PinboardEntryDAO
openNewEntryForm()
formBackingObject(PortletRequest) :Object
PinboardEntry()
com.lifepin.entities::PinboardEntry
getUser(PortletRequest) :User
setContactEmail(String)
submitNewContent()
formBackingObject(PortletRequest) :Object
PinboardEntry()
com.lifepin.entities::PinboardEntry
getUser(PortletRequest) :User
setContactEmail(String)
supports(Class) :boolean
validate(Object, Errors)
processFormSubmission(ActionRequest, ActionResponse, Object, BindException)
onSubmitAction(ActionRequest, ActionResponse, Object, BindException)
getUserId(PortletRequest) :long
setUserId(long)
savePinboardEntry(PinboardEntry)
setCreationDate(Date)
savePinboardEntry(PinboardEntry)
(from Use Case View)

Figure 10: Adding a new LifePin entry

The actions for editing an existing LifePin entry are similar with the difference that not a new PinboardEntry object is created, but the existing one is loaded from the DB with its id (taken from the request parameters) by passing through the PinboardService and PinboardEntryDAO. In addition when saving the edited entry, the userid and the creationDate are not updated again, since they should remain the same as initially.

5 Development

5.1 Testing

A test-driven development approach has been used for developing the application by first defining the interfaces, then by writing the according unit tests before finally implementing the actual logic. The layered architecture as well as Springs dependency injection pattern greatly facilitate this approach by drammatically reducing the object coupling and so as a consequence increasing the testability of classes. So it was quite easy to create a separate (test) data source provider in the according lifepin-data.xml for testing the persisting and reading of objects without polluting the main database.
Also due to the dependency injection approach, injecting mock objects for testing would be quite easy.

5.2 Technical problems

This section points out some technical problems that have been encountered during the development and their according solutions.

5.2.1 Loosing parameter values on validation errors

A problem that has been encountered is the loosing of parameter values on a form when a validation error occurs. This has been the situation during the implementation of the “Add comment” use case. When the user clicks the “Add comment” link, the request starts from a detail view page where a specific LifePin message is being shown. In order to be able to return to that view, the id of the corresponding message has to be passed to the view containing the comment form. In the comment form the parameter is read directly in the JSP code and passed again to the actionURL of the submit and cancel action. This is necessary for then being able to read the id from the URL in the AddCommentController.

```
<portlet:actionURL var="actionUrl">
    <portlet:param name="action" value="addComment"/>
    <portlet:param name="pinboardEntryId"
     value="${param.pinboardEntryId}" />
</portlet:actionURL>
```

Note, the ”${param.pinboardEntryId}” reads the parameter value from the request URL.
In the onSubmitAction method of the AddComment controller, this specific

parameter is read from the request and again passed to the response, s.t. the detail view can again read it and display the correct LifePin message:

```
@Override
protected void onSubmitAction(ActionRequest request,
      ActionResponse response,
      Object command, BindException bindException)
      throws Exception {
  long pinboardEntryId =
    PortletRequestUtils.getLongParameter(
      request,
      ParameterNameConstants.PINBOARDENTRY_ID,
      −1
    );
  ..
}
```

The problem is that in case of a validation error, the onSubmitAction doesn't get called and so the parameter value is lost since it cannot be passed any more to the response. The solution to this problem was to override the processFormSubmission(. . .) of the AbstractFormController in the following way:

```
@Override
protected void processFormSubmission(
  ActionRequest request,
  ActionResponse response,
  Object command,
  BindException errors)
  throws Exception {

  if (errors.hasErrors()) {
    long pinboardEntryId =
      PortletRequestUtils.getLongParameter(
        request,
        ParameterNameConstants.PINBOARDENTRY_ID,
        −1
      );
    response.setRenderParameter(
      ParameterNameConstants.PINBOARDENTRY_ID,
      String.valueOf(pinboardEntryId)
    );
  } else {
    super.processFormSubmission(request,
        response, command, errors);
  }
}
```

This allows to forward the request parameter to the response in case of an error which can be retrieved from the BindingException object. If there are no validation errors, the standard processFormSubmission(. . .) will be called which in turn calls the onSubmitAction(. . .) of the AddComment controller and the normal lifecycle continues.

5.2.2 Localization problems

The LifePin portlet has been fully localized into the English (default), German and Italian language. This has been done by using the Spring *ResourceBundleMessageSource*. Moreover in order to increase the efficiency, the liferay-ui:icon tags have been used for adding command links on the user interface. This has the advantage that images are rendered in an optimized way and moreover the portlet uses standard icons as used on other parts of the Liferay portal. In addition command texts are automatically localized into the different languages. Here however a problem has been found with the localization of the "add-article" liferay-ui:icon message. Although the according localized message is present in the Liferay message bundles it isn't rendered correctly on the portlets UI, neither in the default English language nor in others. It isn't clear whether this is a bug in the liferay-ui:icon tag since no according documentation has been found.

5.2.3 Hibernate "Set" vs. "List" mapping

A strange behavior that has been found during development is the difference in using the "List" or "Set" type in the Hibernate mapping files for handling 1:n relations. By using the "List" element also its subelement "list-index" has to be specified which must be mapped to a separate DB column. When deleting elements however, one has to take care of updating the according list indices manually, otherwise Hibernate will load "null" entries on the missing positions of the list when reading it from the DB. This problem doesn't appear when using the List as annotation in Hibernate annotations as well as when using the "Set" element.