Buzz Discussion Board
Requirements and Design Specifications
(version 0.1)

Fritz Solms, Vreda Pieterse, Stacey Omeleze & Lecton Ramasila
Dept Computer Science, University of Pretoria

March 3, 2015

Contents

1 Vision and scope **2**
- 1.1 Project background . . . 2
- 1.2 Project vision . . . 2
- 1.3 Project scope . . . 3

2 Application requirements and design **3**
- 2.1 Modular System . . . 3
- 2.2 The Buzz-Authorization module . . . 4
  - 2.2.1 Scope . . . 4
  - 2.2.2 Use cases . . . 5
    - 2.2.2.1 addAuthorizationRestriction — priority: important . . . 5
      - 2.2.2.1.1 Service contract . . . 5
    - 2.2.2.2 isAuthorized — priority: important . . . 6
      - 2.2.2.2.1 Service contract . . . 6
    - 2.2.2.3 removeAuthorizationRestriction — priority: important . . . 7
      - 2.2.2.3.1 Service contract . . . 7
    - 2.2.2.4 getAuthorizationRestrictions — priority: important . . . 8
      - 2.2.2.4.1 Service contract . . . 9
    - 2.2.2.5 updateAuthorizationRestriction — priority: important . . . 9
      - 2.2.2.5.1 Service contract . . . 9
  - 2.2.3 Domain model . . . 10
- 2.3 The Buzz-Spaces module . . . 11
  - 2.3.1 Scope . . . 11
  - 2.3.2 Use-cases . . . 11
    - 2.3.2.1 Login and administrative user . . . 11
      - 2.3.2.1.1 Service contract . . . 11
      - 2.3.2.1.2 Functional requirements . . . 12
      - 2.3.2.1.3 Process specification . . . 13
    - 2.3.2.2 BuzzSpace.createBuzzSpace — priority:critical . . . 13
      - 2.3.2.2.1 Services contract . . . 13
      - 2.3.2.2.2 Functional requirements . . . 14
      - 2.3.2.2.3 Process design . . . 15
    - 2.3.2.3 Close buzz space — priority:important . . . 16
      - 2.3.2.3.1 Service contract . . . 16
    - 2.3.2.4 buzzSpace.registerOnBuzzSpace — priority:critical . . . 17
      - 2.3.2.4.1 Service contract . . . 17
    - 2.3.2.5 getProfileForUser — priority:critical . . . 18
  - 2.3.3 Domain model . . . 18
- 2.4 The Buzz-Data-Sources module . . . 19
  - 2.4.1 Scope . . . 19
  - 2.4.2 Use cases . . . 19
    - 2.4.2.1 Login and administrative user . . . 19
      - 2.4.2.1.1 Service contract . . . 19
      - 2.4.2.1.2 Functional requirements . . . 20

2.4.2.1.3 Process specification . . . 21
2.4.2.2 getUsersRolesForModule . . . 21
2.4.2.2.1 Service contract . . . 21
2.4.2.3 getUsersWithRole . . . 22
2.4.2.3.1 Service contract . . . 22
2.4.3 External database structures . . . 22
2.4.4 Domain model . . . 24
2.5 The buzzResourcesModule . . . 24
2.5.1 Scope . . . 24
2.5.2 Use cases . . . 24
2.5.2.1 uploadResource . . . 24
2.5.2.1.1 Services contract . . . 25
2.5.2.1.2 Functional requirements . . . 25
2.5.2.1.3 Process design . . . 25
2.5.3 Domain model . . . 27
2.6 Threads . . . 27
2.6.1 Scope . . . 27
2.6.2 Use cases . . . 28
2.6.2.1 submitPost — priority:critical . . . 28
2.6.2.1.1 Services contract . . . 28
2.6.2.2 markPostAsRead – priority:niceToHave . . . 29
2.6.2.2.1 Services contract . . . 29
2.6.2.3 closeThread — priority:important . . . 30
2.6.2.3.1 Services contract . . . 30
2.6.2.3.2 Thread summarizers . . . 31
2.6.2.3.3 Functional requirements . . . 32
2.6.2.3.4 Process design . . . 33
2.6.2.4 moveThread — priority:important . . . 34
2.6.2.5 hideThread — priority:important . . . 34
2.6.2.6 Threads.queryThread — priority:medium . . . 34
2.6.2.6.1 Services contract . . . 34
2.6.2.6.2 Functional requirements . . . 35
2.6.3 Domain model . . . 35
2.7 The Buzz-Status module . . . 36
2.7.1 Scope . . . 36
2.7.2 Use-cases . . . 37
2.7.2.1 assessProfile . . . 37
2.7.2.2 setStatusCalculator . . . 38
2.7.2.2.1 Services contract . . . 38
2.7.2.3 getStatusForProfile . . . 39
2.7.2.3.1 Services contract . . . 39
2.7.2.4 createAppraisalType . . . 40
2.7.2.5 activateAppraisalType . . . 41
2.7.2.6 assignAppraisalToPost . . . 42
2.7.2.6.1 Service contract . . . 43
2.7.3 Domain model . . . 44

2.8 buzzNotification . . . 44
2.8.1 Scope . . . 44
2.8.2 Use-cases . . . 45
2.8.2.1 registerForNotification — priority:niceToHave . . . 45
2.8.2.1.1 Services contract . . . 45
2.8.3 The domain model for buzzNotification . . . 46
2.9 The web module . . . 46
2.9.1 Scope . . . 46
2.9.2 UI requirements . . . 47
2.9.2.1 Login/out . . . 47
2.9.2.2 Spaces . . . 47
2.9.2.3 UI considerations around threads . . . 47
2.9.2.4 Posts . . . 47
2.9.2.5 Profile information . . . 48
2.9.2.6 Personalization and configurability . . . 48
2.9.2.7 Accessibility of functionality . . . 48
2.10 The Buzz Reporting module . . . 48
2.10.1 Use-cases . . . 49
2.10.1.1 Treads.getThreadStats — priority:medium . . . 49
2.10.1.1.1 Functional requirements . . . 49
2.10.1.2 Get Thread Appraisal — priority:medium . . . 49
2.10.1.2.1 Functional requirements . . . 49
2.10.1.3 Export Thread Appraisal — priority:medium . . . 50
2.10.1.3.1 Functional requirements . . . 50
2.10.1.4 Import Thread Appraisal — priority:medium . . . 50
2.10.1.4.1 Functional requirements . . . 50
2.10.1.5 Export Thread — priority:low . . . 50
2.10.1.5.1 Functional requirements . . . 50
2.10.1.6 Import Thread — priority:low . . . 50
2.10.1.6.1 Functional requirements . . . 51
2.11 The Buzz-Android-Client module . . . 51
2.12 The Buzz-Hamster-Integration module . . . 51

1 Vision and scope

1.1 Project background

Students can provide a lot of knowledge to other students and can learn a lot from another. However, at the University where there are naturally large student numbers, it is often difficult to establish contact and get to know fellow students and to remain in contact with them. Often one student has a problem where other's could and would help, but the communication channels for this are simply not there.

The University would like an application which can enrich the student experience, by providing an environment where students of a module can get to know one another, request assistance from the class, contribute to the class and appreciate each other's contributions. It should also enable teaching staff to pose challenges and provide feedback to the students.

1.2 Project vision

Buzz space is aimed to provide an observable communication space for University modules where students can raise questions, share knowledge add comments and mark up each other's contributions. It should also enable students to follow threads of conversation and be notified of any events occurring in *Buzz* space.

The proposed system is a discussion board which is to be integrated into the department's web site and ultimately also with the *Hamster* marking system. It should enable lecturers to set up a discussion board for a module which provides an observable communication infrastructure for a course can be accessed by the students and teaching staff for that module.

1.3 Project scope

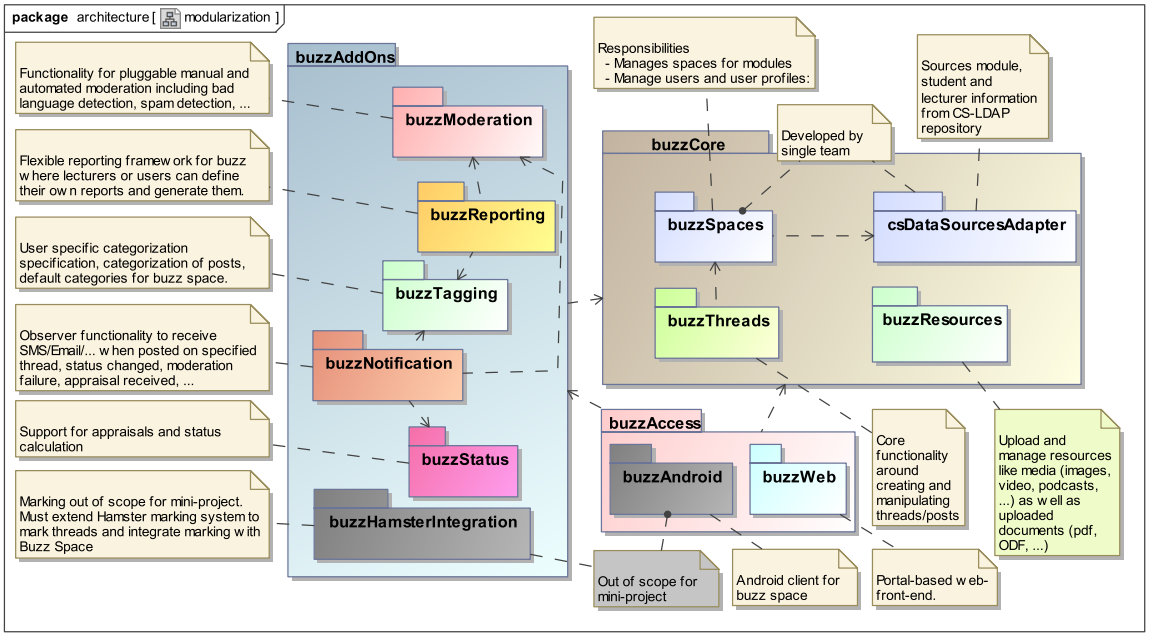


Figure 1: High level system modules and their responsibilities.

The core of the system is a discussion board which is enriched with functionality for tagging, appraisals and status building, notifications, post moderation and reporting. The high level modules and their responsibilities are shown in Figure 1.

2 Application requirements and design

This section discusses for each module of the *Buzz Discussion Board* the functional requirements as well as the process designs for the use cases and the domain models which must be maintained within persistent storage (databases) by that module.

2.1 Modular System

The system is to be a modular system which allows for

- only a subset of modules to be deployed – minimally the system will require the core modules to be deployed.
- further modules to be added at a later stage.

To this end there should be

- minimal dependencies between modules, and
- no dependencies of core modules on any add-on modules.

For example, there is no service on `BuzzSpace` which sets a `ProfileAssessor` as `statusCalculator` for that `BuzzSpace` as that would directly introduce a dependency between the core *buzzSpaces* module and the add-on *buzzStatus* module. Instead there is a *statusCalculatorAssignment* which maintains a reference to both the `ProfileAssessor` and the `BuzzSpace` for which the `ProfileAssessor` is assigned to be the `statusCalculator`.

2.2 The Buzz-Authorization module

All service requests in *Buzz*, irrespective of the channel through which they are requested, are to be intercepted by an `AuthorizationInterceptor` which checks whether the user is authorized to use the service. The interceptor raises a `NotAuthorized` exception if the user is not authorized to use a particular service.

Nearly all use case specifications have thus no separate authorization specifications and hence also no actor (user role). The only two exceptions are

- the `createBuzzSpace` `addAdministrator`, and `removeAdministrator` use cases which can be accessed only by a user of the CS systems who is assigned to be the lecturer for the module for the year for which a buzz space is created or for which an administrator is assigned or removed, and
- the `assignAuthorizationInterceptor` and `removeAuthorizationInterceptor` use cases which can be accessed only by users who have been assigned to be administrators for that space.

The access rules for all other services are fully configurable (by space administrators) from this authorization module. The access to any of the services except the ones discussed above is solely determined by authorization interceptors. If no authorization interceptor covers a particular service, then access is open, i.e. anyone can access the service.

An authorization interceptor can restrict access to one or more system services based on

- the role a user has within a particular module, and
- the status the user has earned on the buzz space for that module.

In addition to being able to configure access rules and to enforce these through interception, the module also enables access channels to query access rules in order to configure user interfaces such that they only show those services to a user to which a user has access.

2.2.1 Scope

The scope of the `buzzAuthorization` is shown in Figure 2.

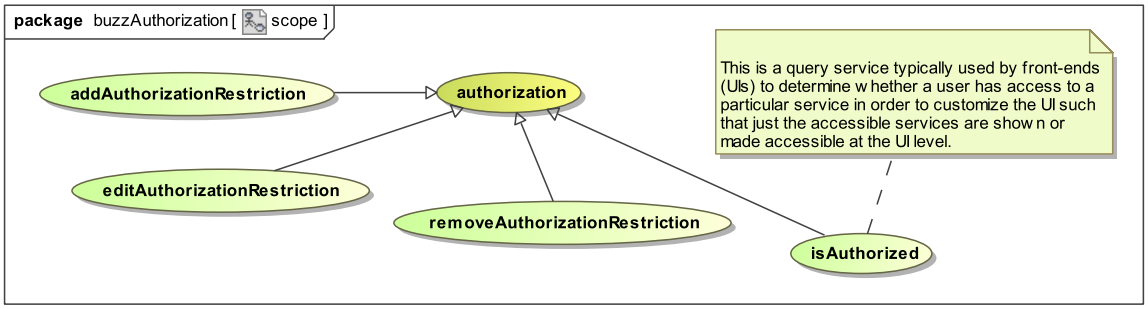


Figure 2: The scope of the buzzAuthorization module.

Administrators are able to assign authorization interceptor to a buzz space which restrict access to one or more services to users who have a particular role and/or a minimum status.

If no authorization interceptor has been added to a particular service, the access to that service is not restricted.

2.2.2 Use cases

This section provides the detailes use case requirements for the use cases offered by the `buzzAuthorization` module.

2.2.2.1 addAuthorizationRestriction — priority: important This use case adds an authorization restriction for a user role and a particular buzz space.

2.2.2.1.1 Service contract The service contract for the `addAuthorizationRestriction` service is shown in Figure 3. The pre-conditions are enforced (raising the appropriate exception should they not be met) and a relevant authorization restriction entity is created and persisted through to database.

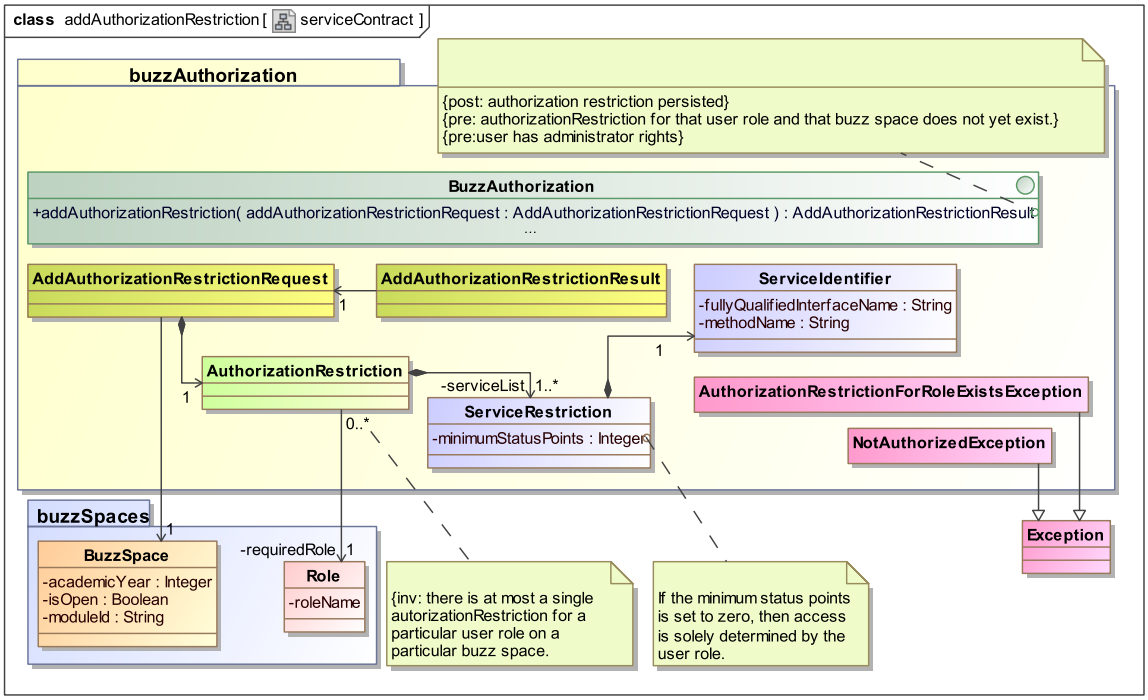


Figure 3: The service contract for adding an authorization restriction to a buzz space.

2.2.2.2 isAuthorized — priority: important This use case is meant to be used by front-ends (web interface, android client, ...) to query the services a user may access in order to customize the user interface for the user. This could be used to, for example, to show to the user only those services which the current user may access.

2.2.2.2.1 Service contract The service contract for the `isAuthorizated` service is shown in Figure 4. This is a simple query service.

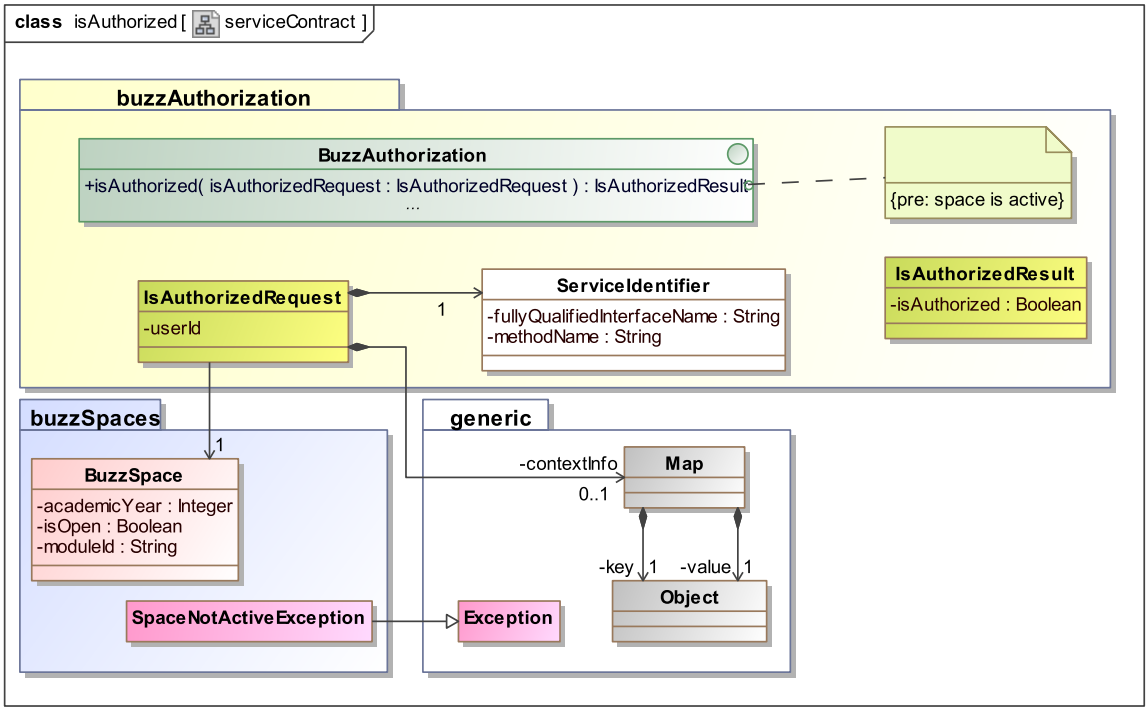


Figure 4: The service contract for querying whether a particular user may access a particular service on a specific buzz space.

2.2.2.3 removeAuthorizationRestriction — priority: important This is a simple use case which removes an authorization restriction for a user role from a buzz space.

2.2.2.3.1 Service contract The service contract for the `removeAuthorizationRestriction` is shown in Figure 5.

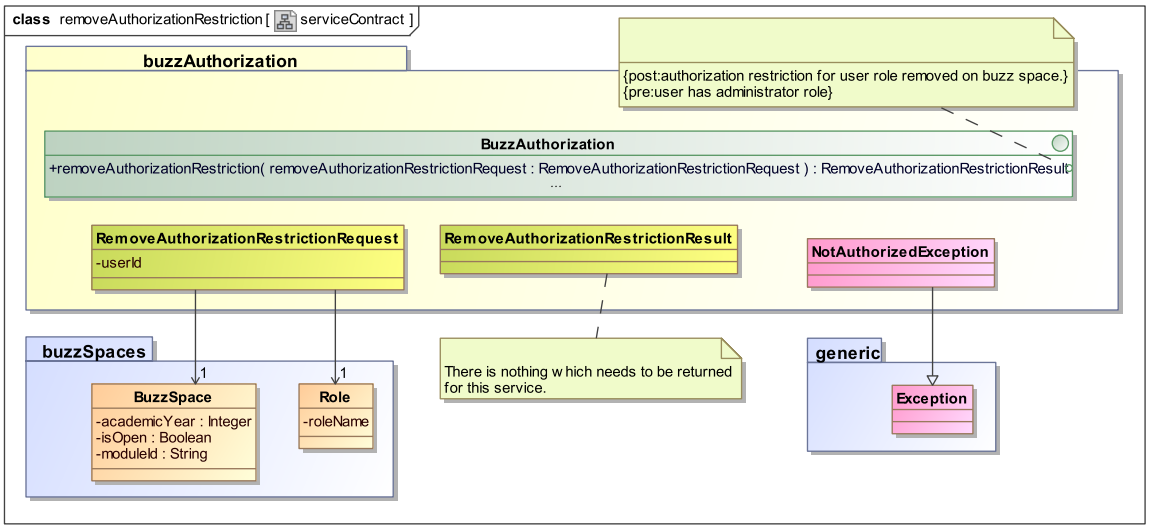


Figure 5: The service contract for removing an authorization restriction for a user role on a buzz space.

2.2.2.4 getAuthorizationRestrictions — priority: important This use case us used typically by front-ends (web interfaces, Android clients and other UIs) to retrieve the autorization restrictions to enable users to select an authorization restricxtion they would like to update.

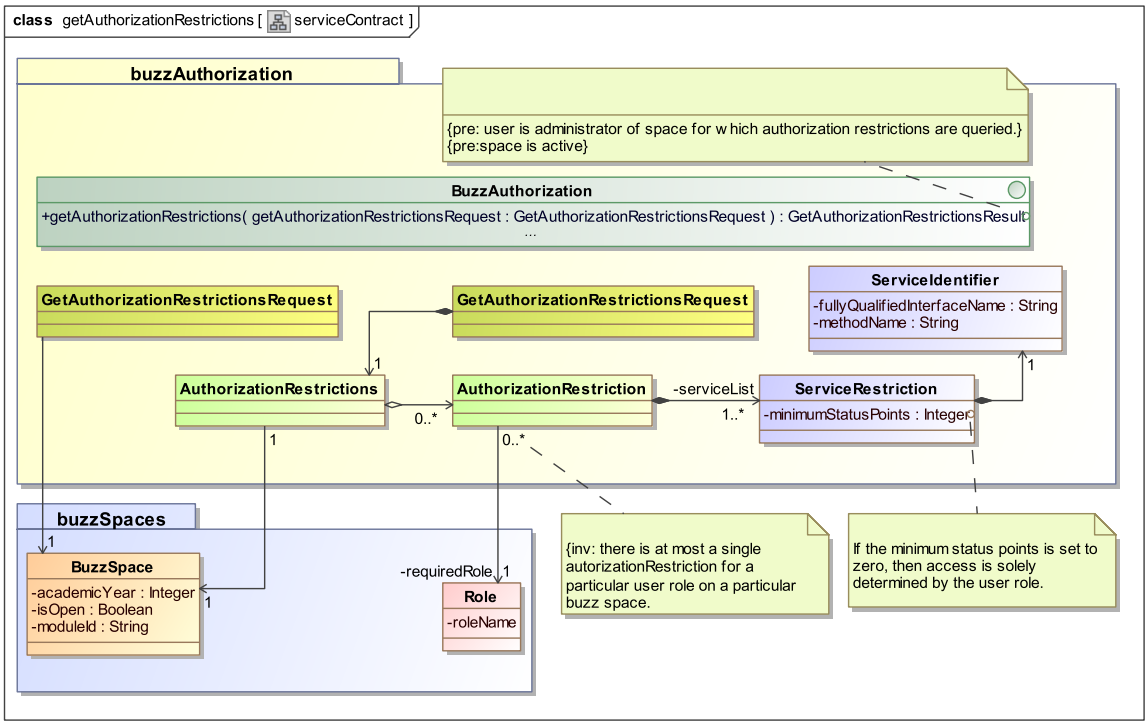


Figure 6: The service contract for retrieving authorization restriction for a user and a buzz space.

2.2.2.4.1 Service contract

2.2.2.5 updateAuthorizationRestriction — priority: important This use case facilitates editing of authorization restrictions.

2.2.2.5.1 Service contract The service contract for the `updateAuthorizationRestriction` is shown in Figure 7.

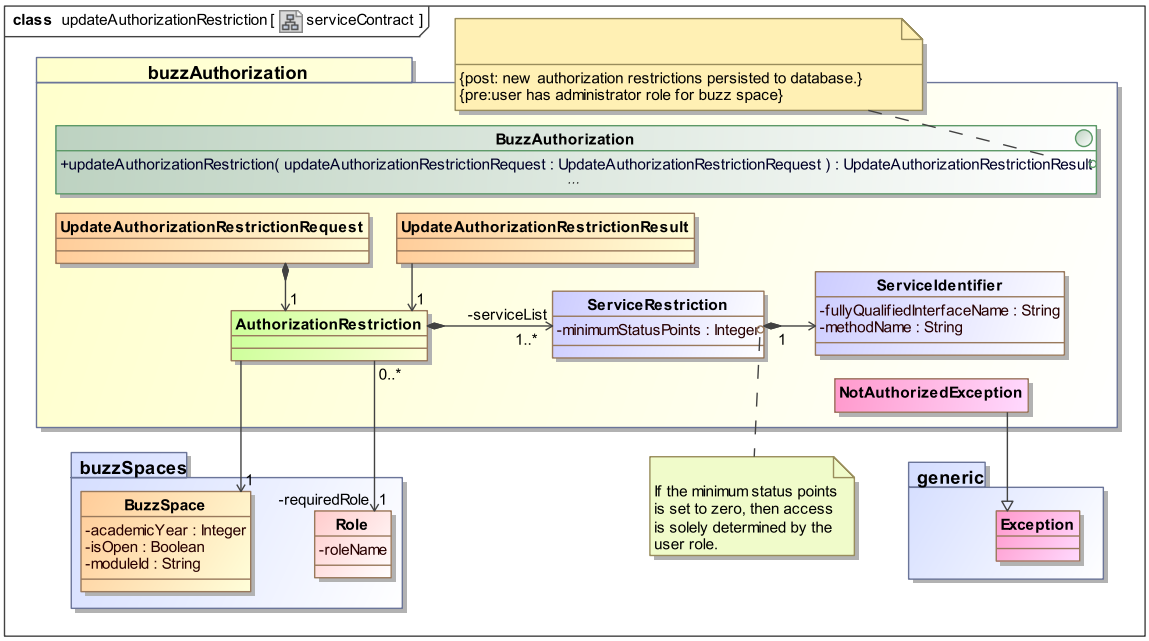


Figure 7: The service contract for updating an authorization restriction for a user role on a buzz space.

2.2.3 Domain model

The `buzzAuthorization` module maintains the access restriction information in the form of authorization restrictions. Each authorization restriction applies to one or more services identified by the fully qualified name of the interface representing the service contract and the method name.

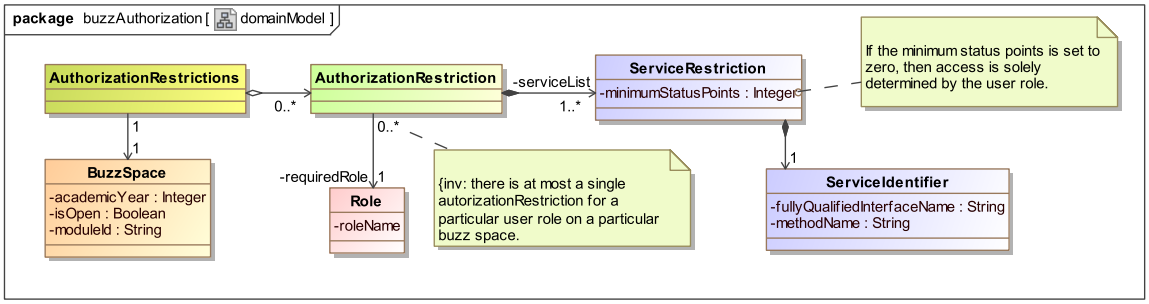


Figure 8: The domain model of the buzzAuthorization module.

If there is no authorization restriction for a service, then access to that service is not restricted. If there are multiple atuhorization restrictions, then at least one of the restrictions need to grant

access to a user for a user to be able to make use of the restricted service.

2.3 The Buzz-Spaces module

BuzzSpaces is the core module which is responsible for managing the buzz spaces for the different modules.

2.3.1 Scope

The scope of the *buzzSpaces* module is shown in Figure 9. The scope is really restricted to space and user management.

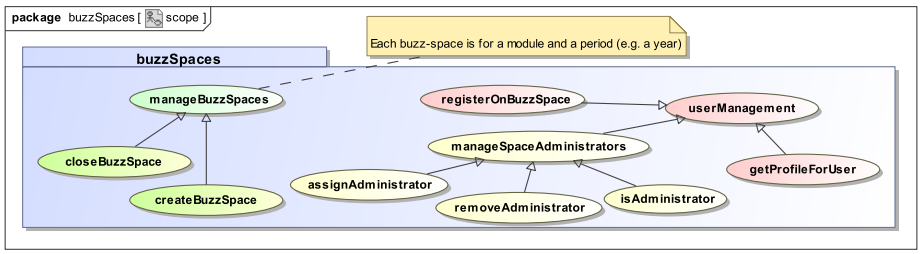


Figure 9: The scope of the buzzSpaces module.

2.3.2 Use-cases

The *buzzSpaces* module provides services to create and close spaces.

2.3.2.1 Login and administrative user The system will authenticate against the CS data sources within which authentication credentials are currently stored. Currently this is an LDAP repository. This ensures that the same authentication credentials are used across the different system used within the department.

2.3.2.1.1 Service contract The service request for the `login` use case contains the authentication credentials which are currently `UsernamePasswordCredentials` but which could change in future. The service will perform the authentication against the *Computer Science LDAP Repository*.

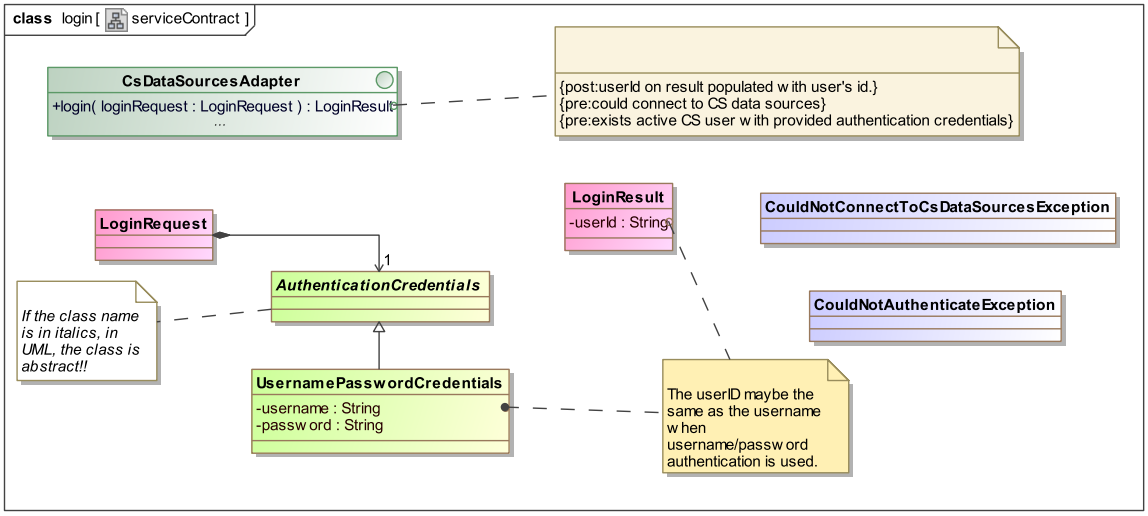


Figure 10: The service contract for the login use case.

If the `login` is successful, the `userId` is returned – this may be the same as the username used for authentication (if the username and password are used as authentication credentials).

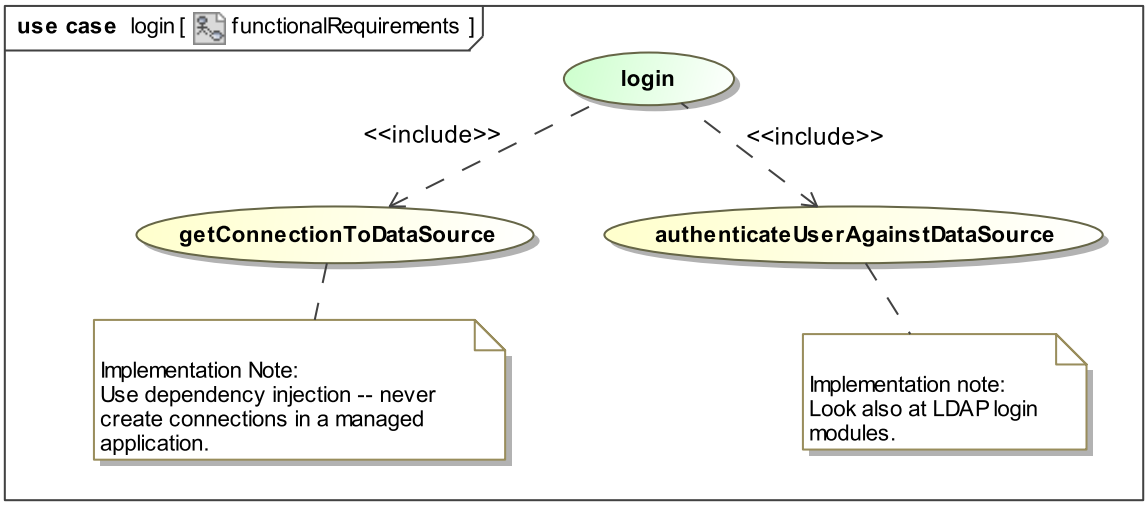


Figure 11: The functional for the login use case.

2.3.2.1.2 Functional requirements

2.3.2.1.3 Process specification The process for this is trivial. Upon receiving the login-request, the service tries to authenticate using the CS data sources adapter. Any exception raised by this lower level service is also raised by this service. In the success scenario the result object is populated and returned.

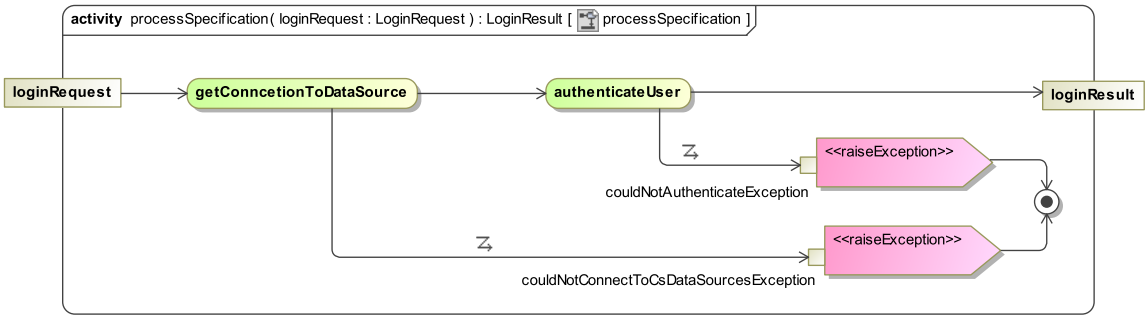


Figure 12: The process specification for the login use case.

2.3.2.2 BuzzSpace.createBuzzSpace — priority:critical The service enables lecturers to create a Buzz space for a particular module they present during a particular year. This creates a root thread for the buzz space with an associated welcome post and assigns the lecturer as administrator to the space.

2.3.2.2.1 Services contract The `CreateBuzzSpaceRequest` identifies the user who is requesting the creation and the module for which the buzz space is to be created. The system will assume that a buzz space for the current academic year is to be created.

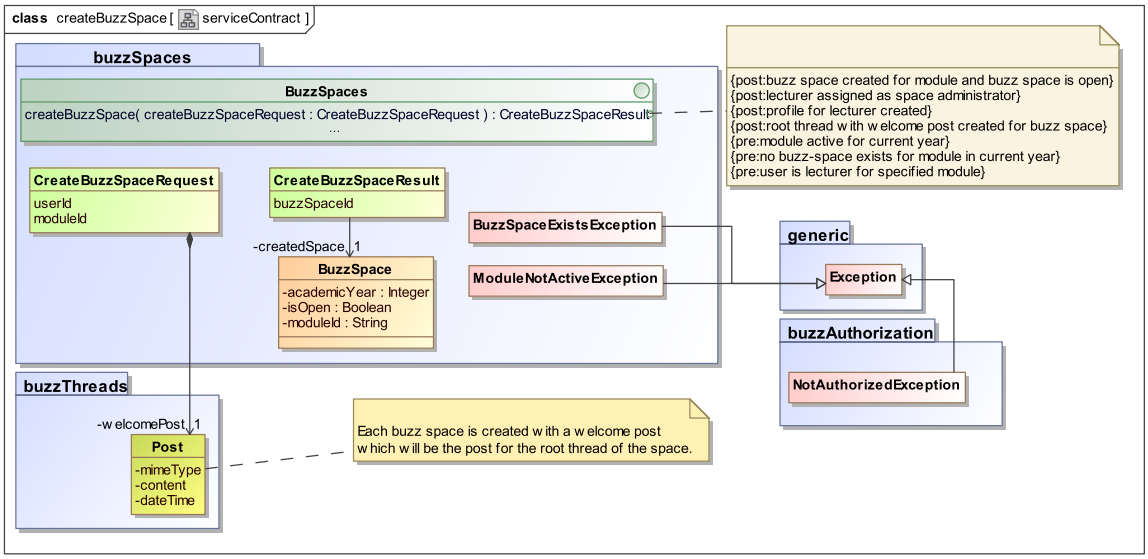


Figure 13: The service contract for the createBuzzSpace use case.

The service has 3 pre-conditions, i.e. 3 conditions under which the buzz space will not be created. For each of those pre-conditions an exception is introduced which is raised by the service to notify the caller that the service is not being provided because the pre-condition associated with that exception has not been met.

The post-conditions specify the conditions which must hold true when the service has been provided. The post-conditions for this service are that the requested buzz space has been created and that a root thread has been created for the buzz space.

2.3.2.2.2 Functional requirements The use case realizes the service specified in the services contract for `BuzzSpaces`. The functional requirements for the use case specify the lower level functions required by the use case to realize the service as specified in the service contract. Each functional requirement is either checking for a pre-conditions, realizing a post-condition or both.

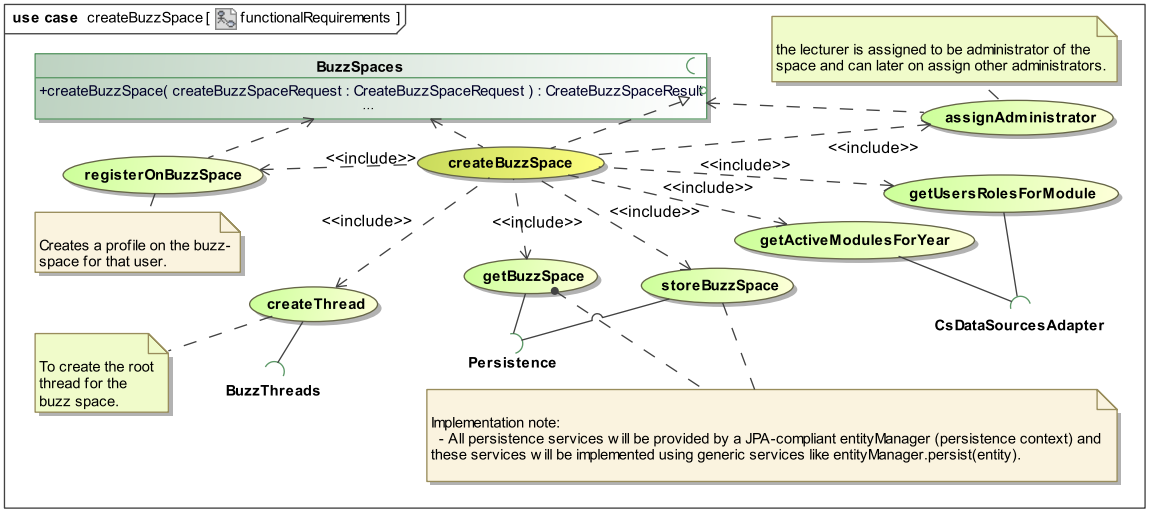


Figure 14: The functional requirements for the createBuzzSpace use case.

Each functional requirement is associated with a services contract for the responsibility domain within which that lower level service falls. Whichever object realizes that services contract needs to provide the service.

2.3.2.2.3 Process design The process is now assembled by "orchestrating" the service from the lower level services. The outer (context) activity is the activity of realizing the `createBuzzSpace` service. It receives the service request and requests in its process the lower level services from whichever object is deployed to realize the services contract[1].

[1]In the implementation dependency injection is used to fully decouple the implementation classes — i.e. the dependencies are purely on contracts (interfaces) and not on classes.

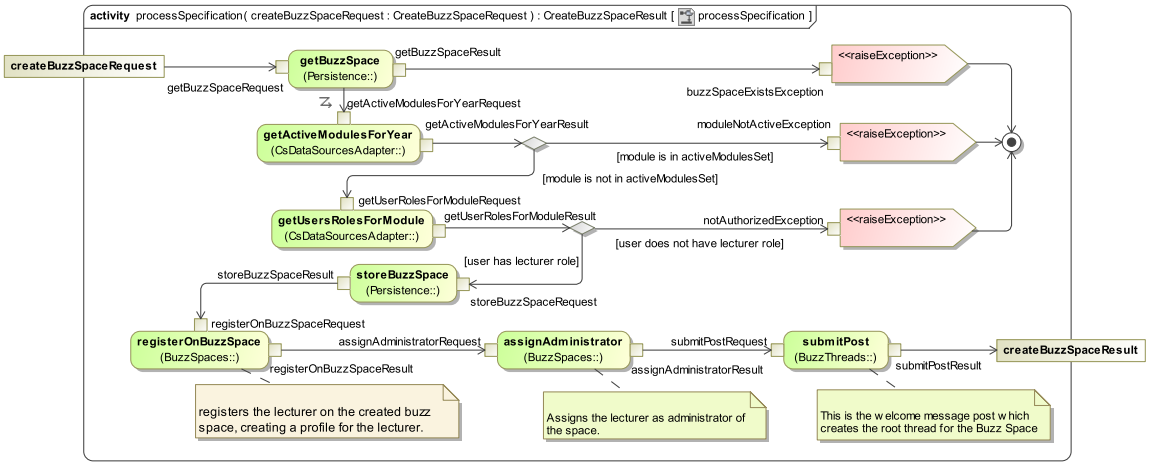


Figure 15: The process design for the createBuzzSpace use case.

Note that the first activity is to retrieve the buzz space for the module. If a buzz space is provided, our service raises an exception because a buzz space for that service and the current academic year exists already, i.e. we only continue with our service if the lower level `getBuzzSpace` service raises an exception signalling that no such buzz space exists.

Note also that if any of the three pre-conditions for our service is not met, that an exception is raised and the service is not provided. Only if all preconditions are met is the buzz space created and the result returned.

2.3.2.3 Close buzz space — priority:important The `closeBuzzSpace` use case is very simple. It is simply sets the buzz space as inactive.

2.3.2.3.1 Service contract The service contract for the `closeBuzzSpace` use case is shown in Figure 16.

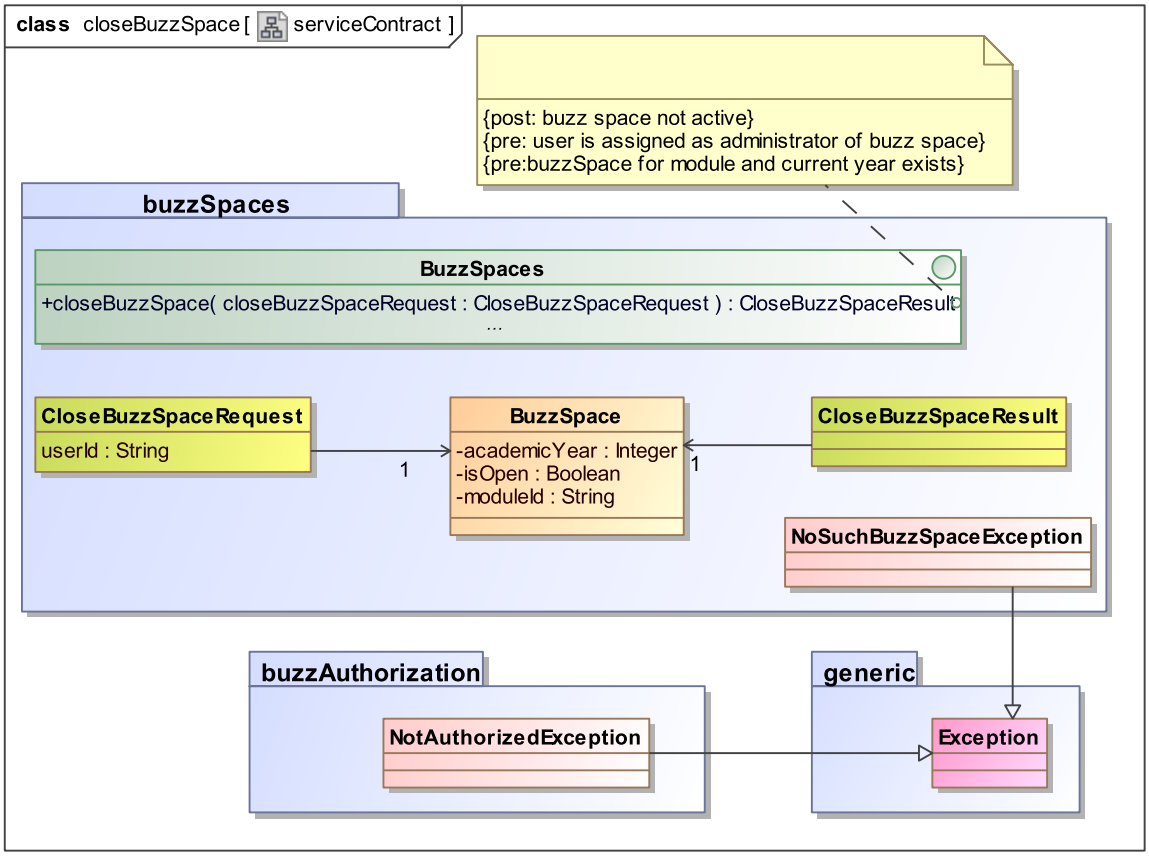


Figure 16: The service contract for the closeBuzzSpace use case.

2.3.2.4 buzzSpace.registerOnBuzzSpace — priority:critical Registering a user with a buzz space effectively creates a profile on that buzz space for that user. The profile will be populated, for example, by various modules with activities done and status earned by the user, as well as any notifications or setup information persisted for that user.

2.3.2.4.1 Service contract The service contract for the `registerOnBuzzSpace` use case is shown in Figure 17.

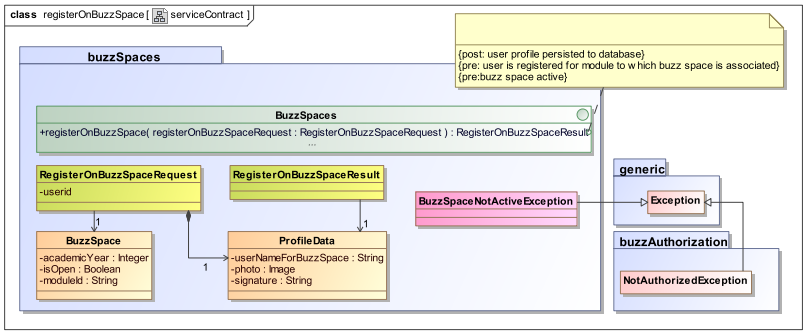


Figure 17: The service contract for the registerOnBuzzSpace use case.

2.3.2.5 getProfileForUser — priority:critical This is a simple query service which returns the profile the user has on the buzz-space.

2.3.3 Domain model

The domain model for this module is very simple. It only requires that buzz spaces are persisted and that each buzz space has a thread which plays the role of the root thread for that buzz space.

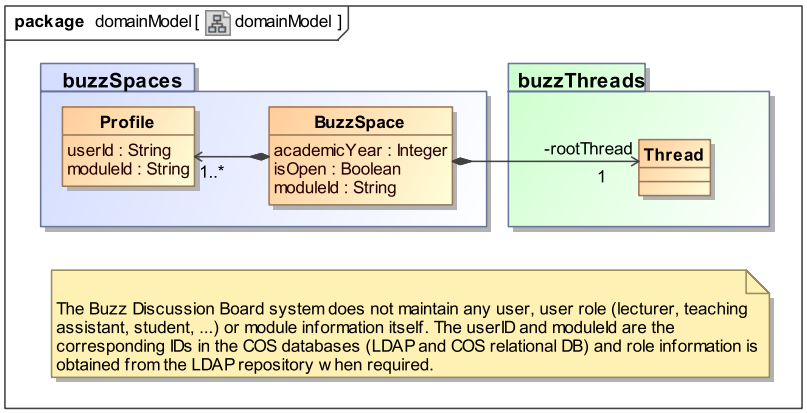


Figure 18: The domain model for the buzzSpaces module.

Note that the relationship between a `BuzzSpace` and that `Thread` which is the root thread for that space is a *composition* relationship as the root thread

- is only accessible from that space, and
- if the space is deleted, so is its root thread.

2.4 The Buzz-Data-Sources module

This module is responsible for sourcing data from external CS databases, i.e. databases maintained by Computer Science but which are not part of this system. The access is purely read-access, i.e. no data will be modified in the external databases.

2.4.1 Scope

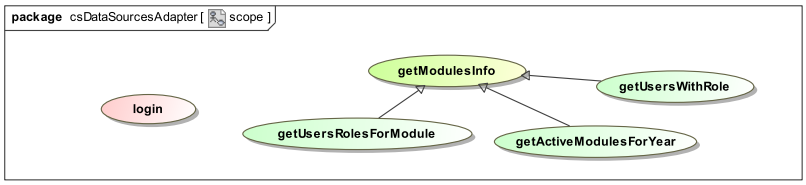


Figure 19: The scope of the CsDataSources module.

2.4.2 Use cases

This module provides a number of services to *Buzz* which require information from data stored in external databases maintained by the *Department of Computer Science*.

2.4.2.1 Login and administrative user The system will authenticate against the CS data sources within which authentication credentials are currently stored. Currently this is an LDAP repository. This ensures that the same authentication credentials are used across the different system used within the department.

2.4.2.1.1 Service contract The service request for the `login` use case contains the authentication credentials which are currently `UsernamePasswordCredentials` but which could change in future. The service will perform the authentication against the *Computer Science LDAP Repository*.

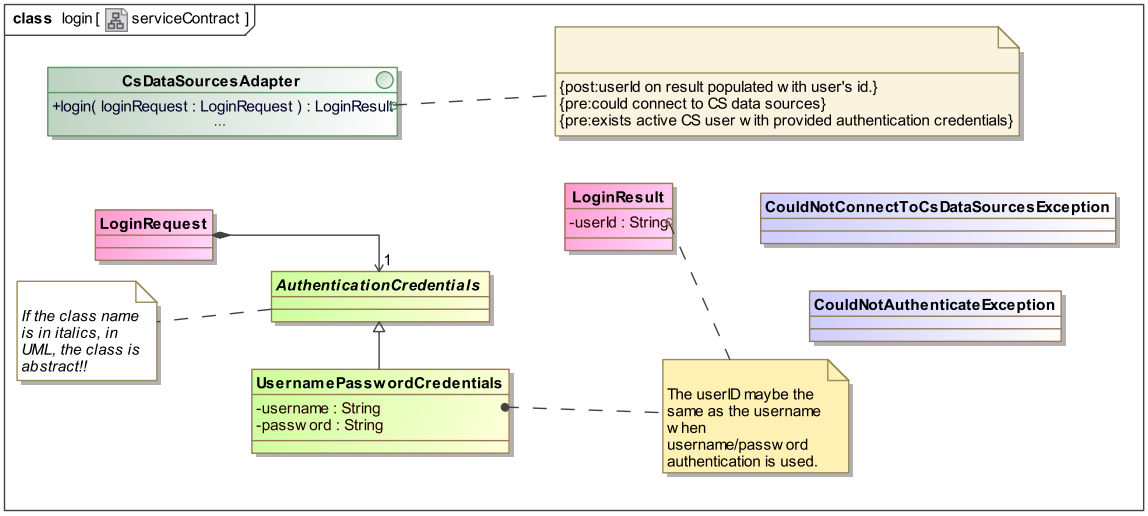


Figure 20: The service contract for the login use case.

If the `login` is successful, the `userId` is returned – this may be the same as the username used for authentication (if the username and password are used as authentication credentials).

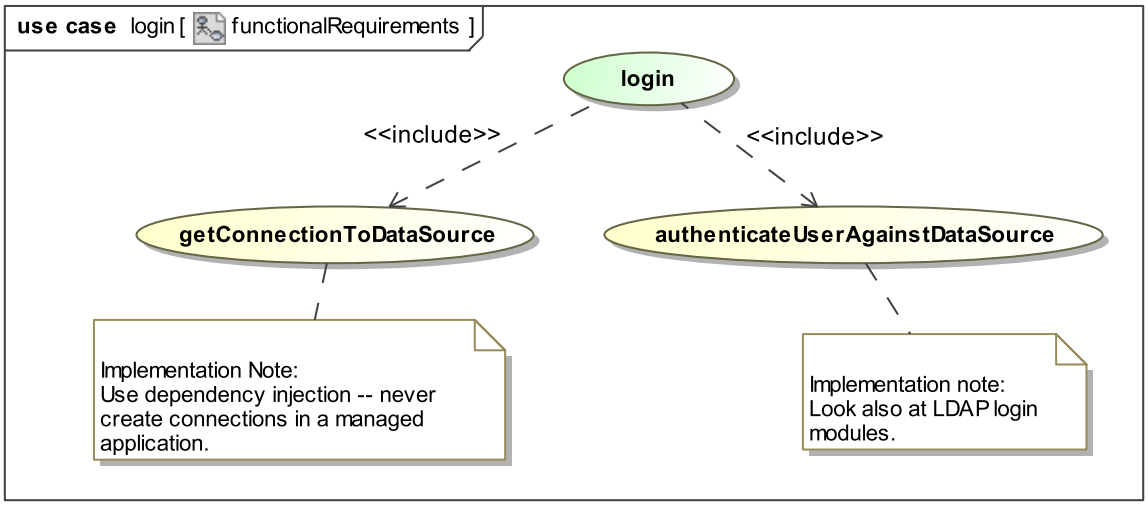


Figure 21: The functional for the login use case.

2.4.2.1.2 Functional requirements

2.4.2.1.3 Process specification The process for this is trivial. Upon receiving the login-request, the service tries to authenticate using the CS data sources adapter. Any exception raised by this lower level service is also raised by this service. In the success scenario the result object is populated and returned.

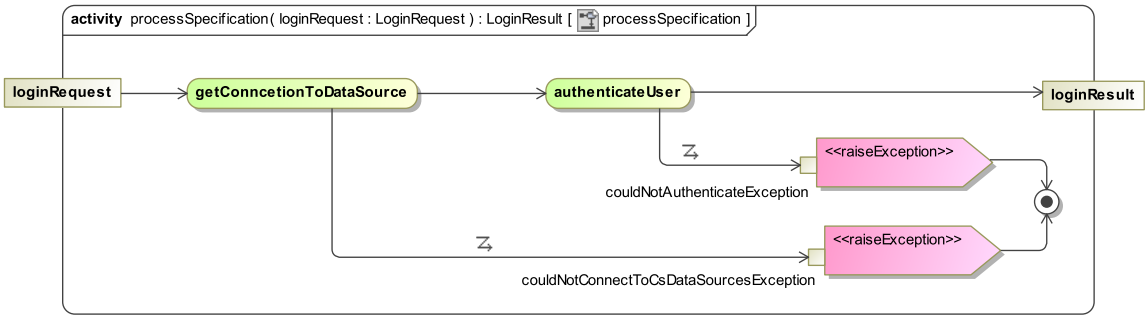


Figure 22: The process specification for the login use case.

2.4.2.2 getUsersRolesForModule This use case queries the user roles for a particular user. This is basically a database lookup and hence only the service contract is specified. The design (functional requirements and process specification) are so simple that they are left to the for the implementation phase.

2.4.2.2.1 Service contract Figure 23 depicts the services contract for the use case.

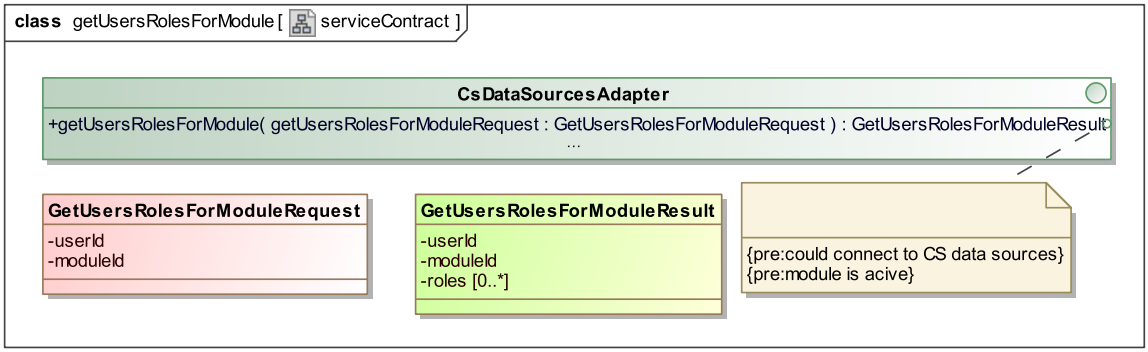


Figure 23: The service contract for the getUsersRolesForModule use case.

Note that there are two pre-conditions and hence that the service will possibly throw two notifiable exceptions. Not also the associations to `User`, `Module` and `Role`. These are not composition relationships and hence the request and result object do not contain the user, module and role information themselves, just references or identifiers for users, roles and modules.

2.4.2.3 getUsersWithRole

This use case retrieves all users which have a particular role (e.g. a `teachingAssistant` role) for a particular module. Since the use case is effectively just a database lookup with the appropriate result object creation, we only specify the services contract and leave the design to development.

2.4.2.3.1 Service contract

Figure 24 depicts the services contract for the use case.

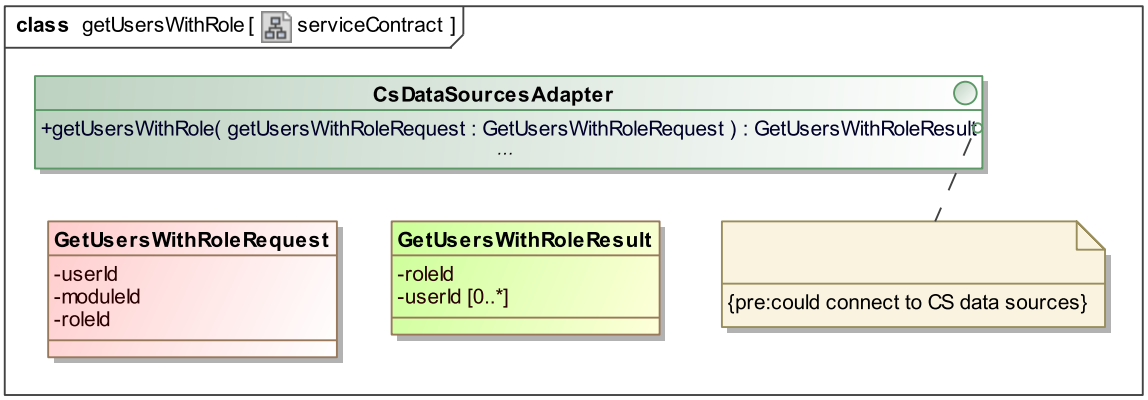


Figure 24: The service contract for the getUsersForRole use case.

Add missing use cases

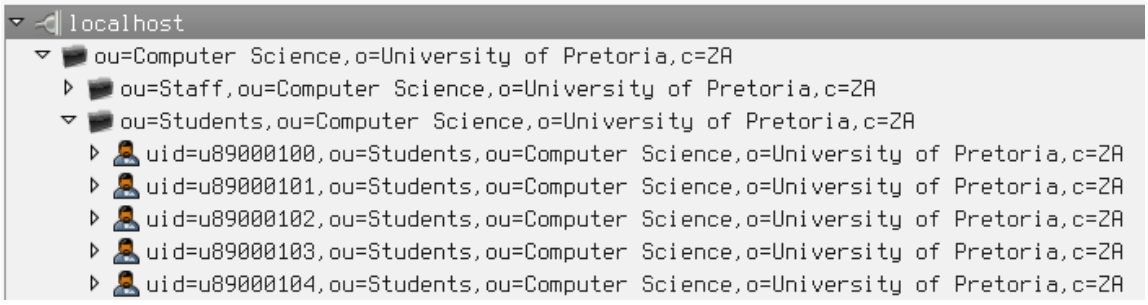
2.4.3 External database structures

The details for the external database structures can be obtained from Neels van Rooyen who is working at the tech team. Neels may also assist you with other technicalities around integrating with the CS databases.

The relational database structure containing the courses/module information has the following structure:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
code	varchar(20)	YES		NULL	
name	varchar(255)	NO			
lecturer	varchar(255)	NO		0	
description	text	YES		NULL	
semester	smallint(6)	NO		0	
has_webct	tinyint(4)	YES		NULL	
year_group	int(11)	YES		NULL	
hidden	tinyint(3) unsigned	NO		0	
last_updated	datetime	NO		0000-00-00 00:00:00	
discussion_board	tinyint(4)	YES		NULL	
tutors_allowed	tinyint(2)	YES		NULL	

The structure of the LDAP repository is shown in the following figure



For example, an LDAP search for stud_COS301 returns:

dn:	cn=stud_COS301,ou=Modules,ou=Groups,ou=Computer Science,o=University of Pretoria,c=ZA
objectClass:	top
objectClass:	posixGroup
cn:	stud_COS301
gidNumber:	10093
memberUid:	u11061015
memberUid:	u12214834
memberUid:	u11063612
memberUid:	u11002566
memberUid:	u12019837
memberUid:	u11371910
memberUid:	u12148858
memberUid:	u29557373
memberUid:	u11247143

and an LDAP search for u29052735 returns:

dn:	uid=u29052735,ou=students,ou=Computer Science,o=University of Pretoria,c=ZA
objectClass:	inetOrgPerson
objectClass:	posixAccount
objectClass:	top
uidNumber:	26526
gidNumber:	10001
loginShell:	/bin/bash
title:	Mr
initials:	CJ
st:	8911085042083
sn:	van Rooyen
homeDirectory:	/home/cs/students/u29052735
employeeNumber:	29052735
uid:	u29052735
mail:	nexusdk@gmail.com
cn:	Neels

and an LDAP search for memberuid=u29052735 returns:

cn: stud_COS333
cn: stud_COS301
cn: stud_COS326
cn: stud_COS216
cn: stud_COS330

2.4.4 Domain model

There is no domain model for this module as this is purely an adapter providing services to *Buzz*. This module does not persist any information into a database.

2.5 The buzzResourcesModule

The buzzResources module is used to upload asnd manage resources like media (e.g. images, video, ...) and documents (e.g. PDF documents, Open Document Format documents, ...). These resources can be either embedded or linked to in posts.

2.5.1 Scope

The scope of the *buzzResources* module is shown in Figure 25.

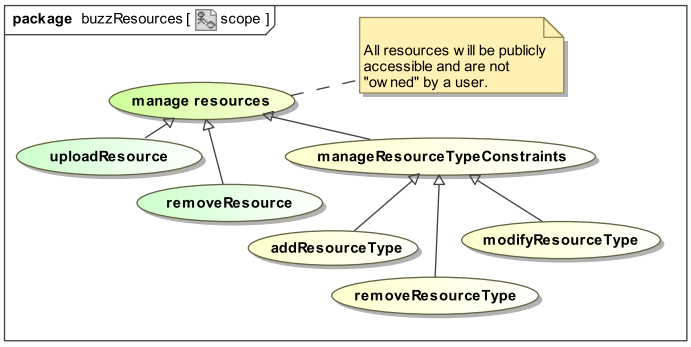


Figure 25: The scope of the buzzResources module.

2.5.2 Use cases

This section specifies the use case requirements for the use cases offered by the buzzResources module.

2.5.2.1 uploadResource Users can upload resources lik media files or documents. Any uploaded resource is accessible by other users whoc can also specify links to that resource.

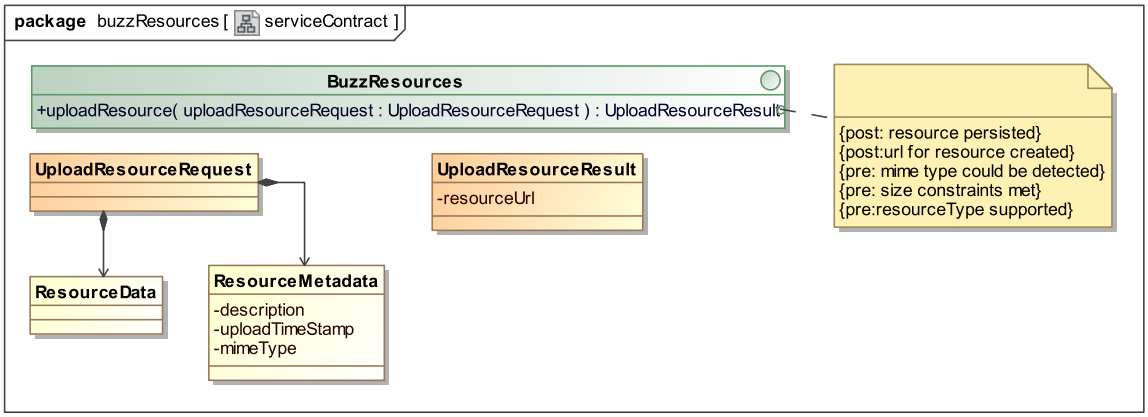


Figure 26: The service contract for the uploadResource use case.

2.5.2.1.1 Services contract

2.5.2.1.2 Functional requirements The functional requirements for the `uploadResource` are shown in Figure 27.

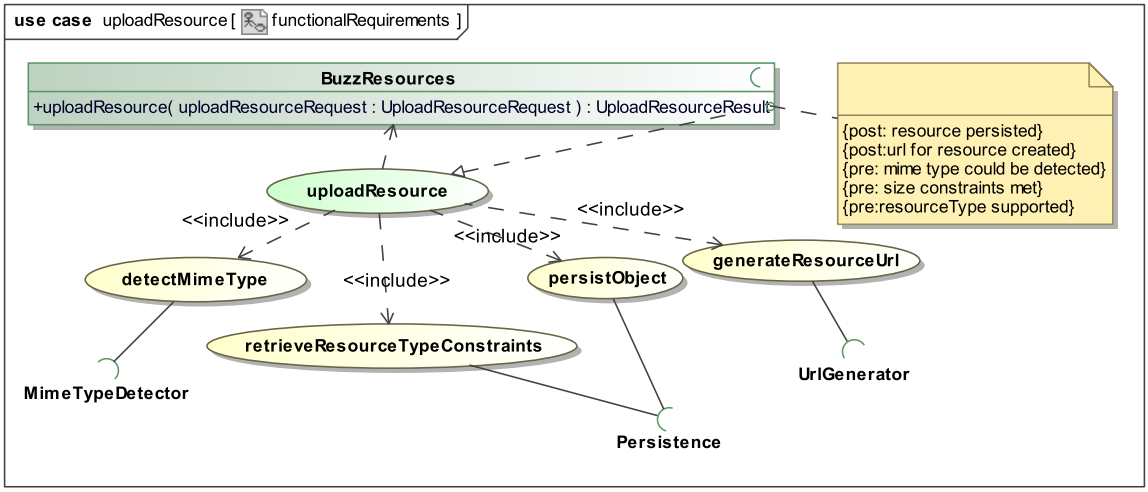


Figure 27: The functional requirements for the uploadResource use case.

2.5.2.1.3 Process design The process is now assembled by "orchestrating" the service from the lower level services. The outer (context) activity is the activity of realizing the `createBuzzSpace` service. It receives the service request and requests in its process the lower level services from

whichever object is deployed to realize the services contract[2].

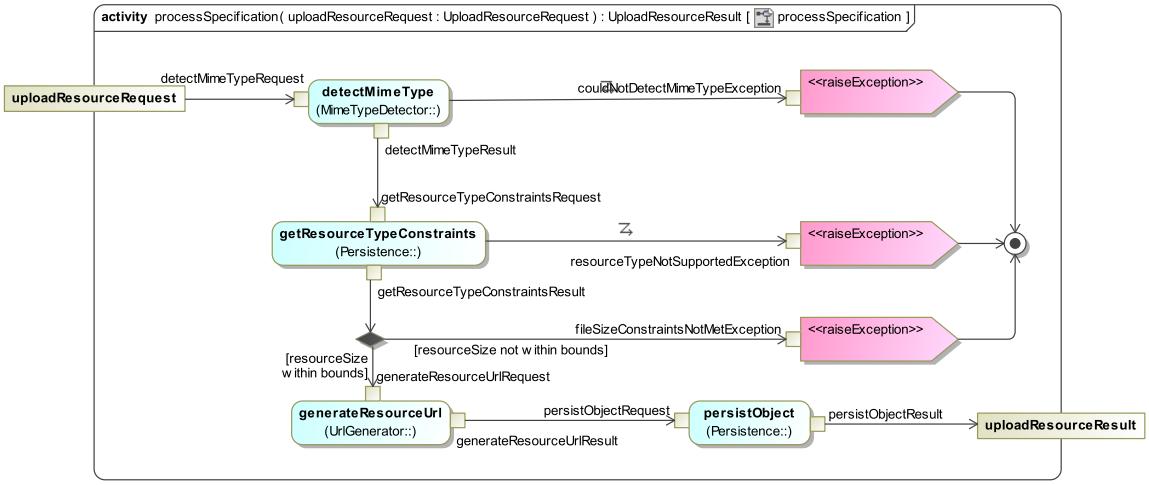


Figure 28: The process design for the createBuzzSpace use case.

Note that the first activity is to retrieve the buzz space for the module. If a buzz space is provided, our service raises an exception because a buzz space for that service and the current academic year exists already, i.e. we only continue with our service if the lower level `getBuzzSpace` service raises an exception signalling that no such buzz space exists.

Note also that if any of the three pre-conditions for our service is not met, that an exception is raised and the service is not provided. Only if all preconditions are met is the buzz space created and the result returned.

[2]In the implementation dependency injection is used to fully decouple the implementation classes — i.e. the dependencies are purely on contracts (interfaces) and not on classes.

2.5.3 Domain model

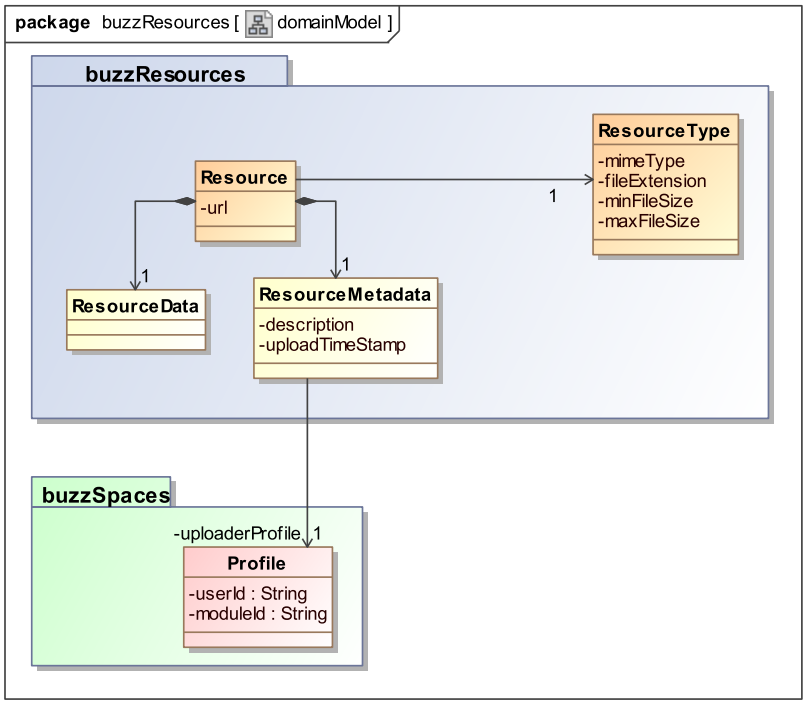


Figure 29: The domain model for the buzzResources module.

2.6 Threads

The *BuzzThreads* module is responsible for the functionality around threads and posts.

2.6.1 Scope

The scope of the *Treads* module is depicted in Figure 30. Users can CRUD posts. Depending on their privileges there may be restrictions on where users will be allowed to create posts.

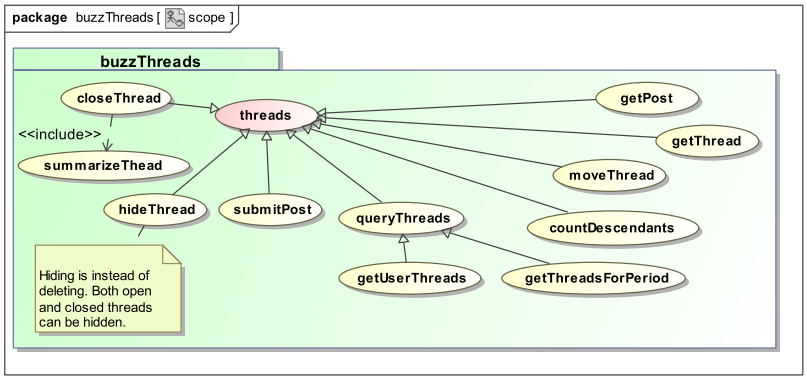


Figure 30: The scope of the CsDataSources module.

2.6.2 Use cases

The *Threads* module provides services to CRUD posts. In addition it also provides functions to change properties of a thread that influences how a thread is displayed, like hiding and closing threads.

An important feature is to allow users with authority to move a branch to a new location. In essence it boils down to change the parent of a selected post.

Furthermore information regarding the posts in a thread can be gathered using the versatile functions called **queryTread** and **getTreadStats**.

2.6.2.1 submitPost — priority:critical The service enables a user to submit a post to thread, creating a child thread. Top level threads for a buzz space are created by submitting a post against the `rootThread` of the buzzSpace.

2.6.2.1.1 Services contract The `SubmitPostRequest` contains the post itself and identifies the thread the user wants to post on and through this the buzz space the user is posting on. The post itself has a content and a mime type. Initially only `txt` and `html` mime types will be supported. Within HTML posts one can embed uploaded images and provide links to uploaded documents.

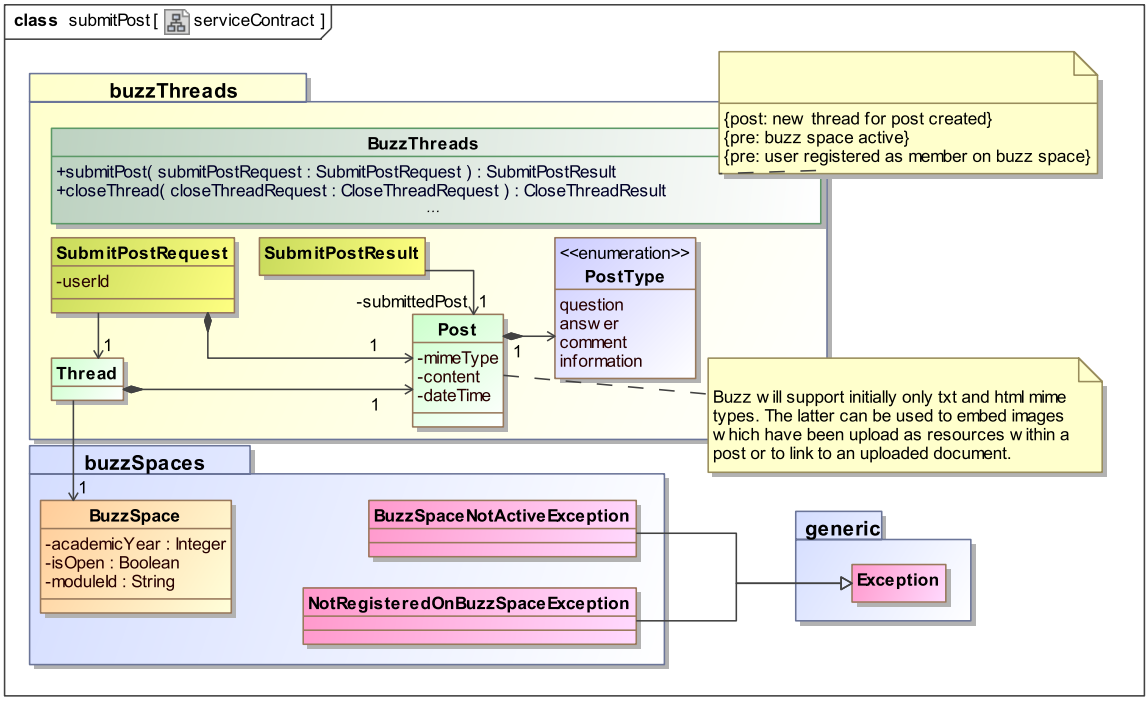


Figure 31: The service contract for the submitPost use case.

The service has a number following pre-conditions. For each pre-conditions an exception is introduced which is raised by the service to notify the caller that the service is not being provided because the pre-condition associated with that exception has not been met.

2.6.2.2 markPostAsRead – priority:niceToHave This use case simply stores a reading event which contains the information that a user has read a particular post at a particular time.

2.6.2.2.1 Services contract The `CloseRequest` identifies the thread the user wants to close and may optionally specify a `ThreadSumamrizer` which should be used to summarize the thread. The thread summary can be text-based or hypertext (HTML) with embedded images and links to uploaded documents.

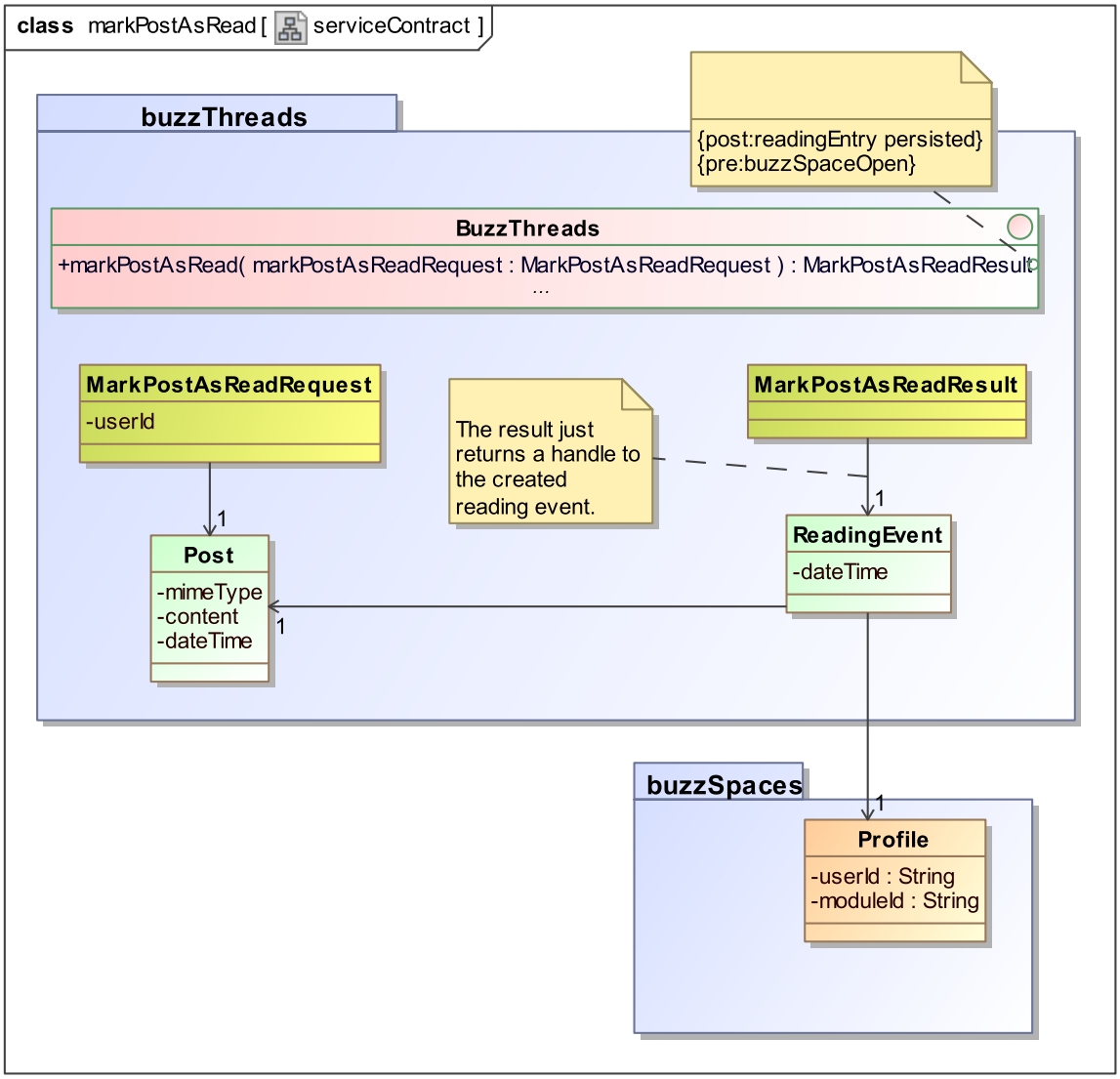


Figure 32: The service contract for the closeThread use case.

2.6.2.3 closeThread — priority:important This service closes a thread preventing any further changes to a thread. This includes preventing further posts, appraisals and all other activities on a thread by any normal users. Optionally a thread can be summarized either manually or via an automated thread summarizer.

2.6.2.3.1 Services contract The `CloseRequest` identifies the thread the user wants to close and may optionally specify a `ThreadSumamrizer` which should be used to summarize the

thread. The thread summary can be text-based or hypertext (HTML) with embedded images and links to uploaded documents.

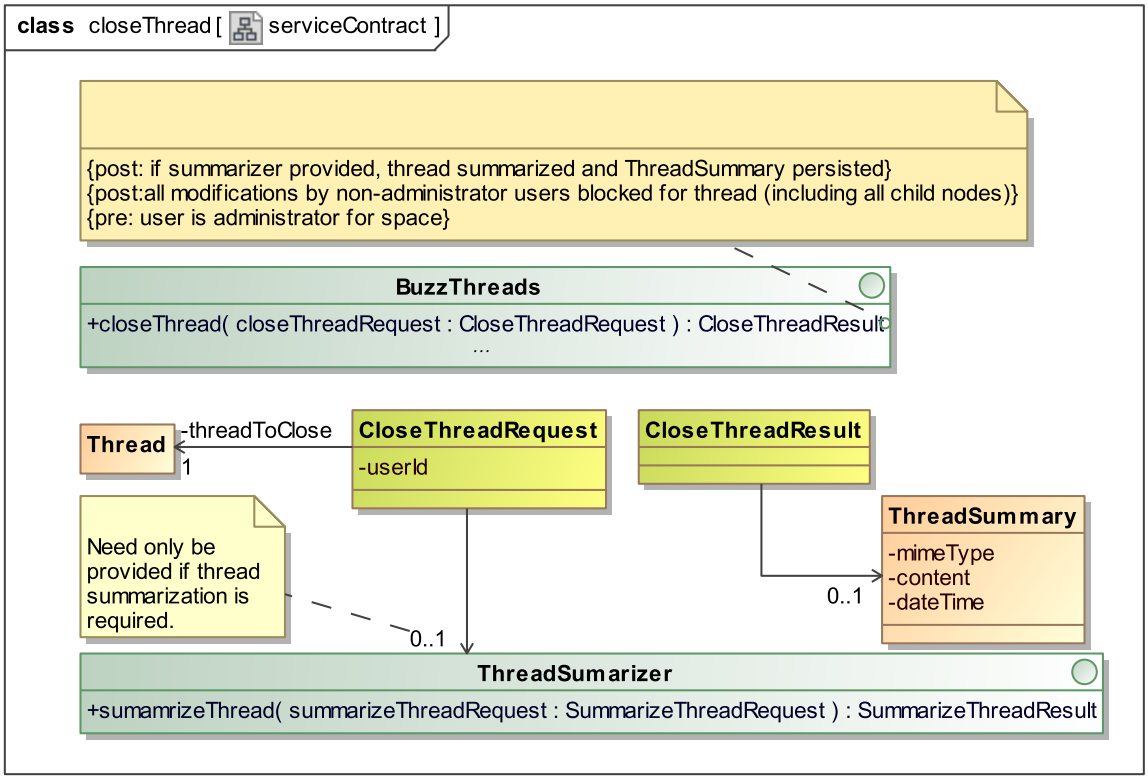


Figure 33: The service contract for the closeThread use case.

2.6.2.3.2 Thread summarizers The design for automated and manual thread summarizers is shown in Figure **??**

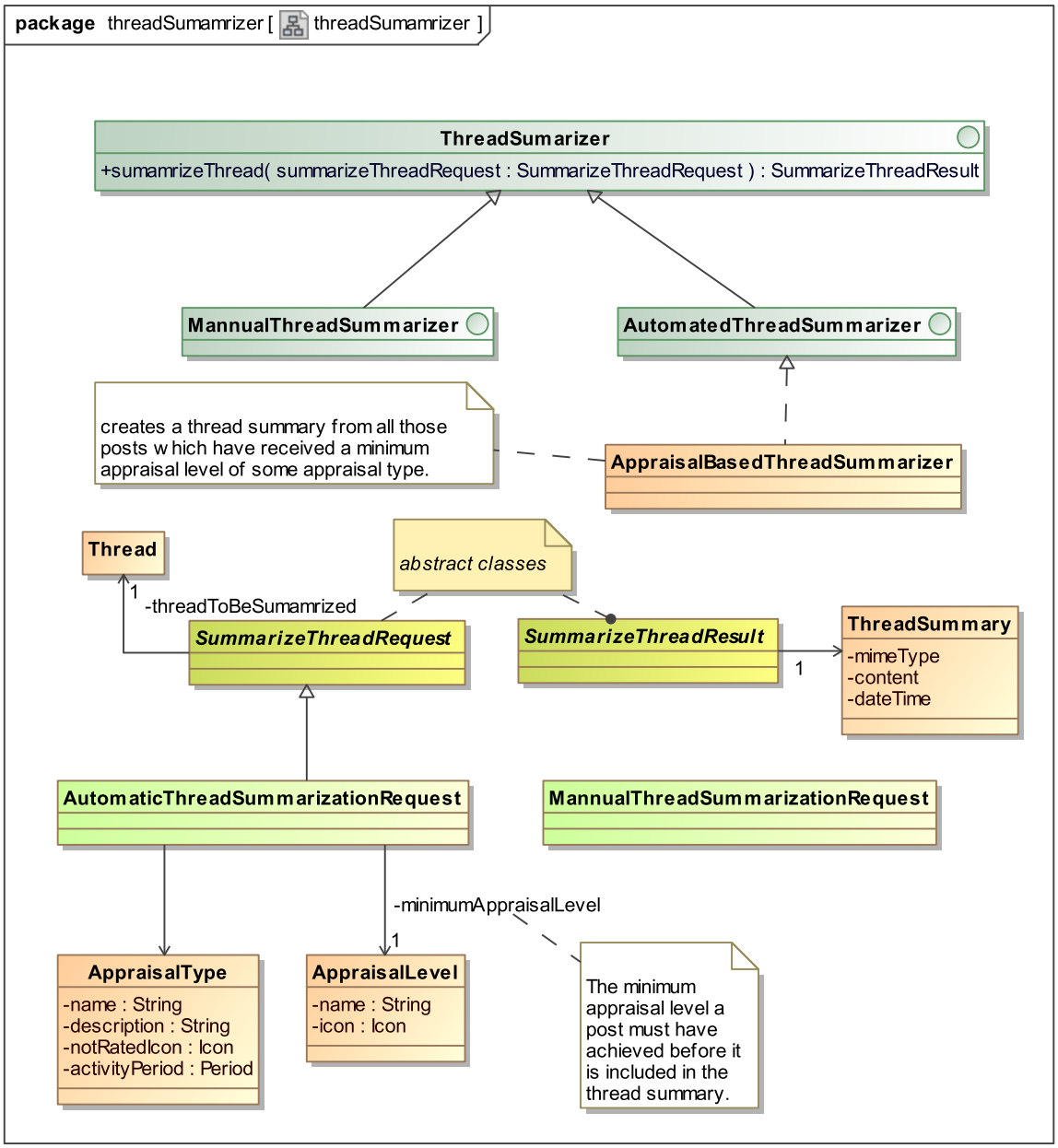


Figure 34: The design for thread summarizers.

2.6.2.3.3 Functional requirements When a thread is closed one can optionally request thread summarization. If thread sumarization is not requested, the thread continues to be readable, but no modifications can be made by any users other than administrators. Administrators will be

able to add appraisal types to closed thread for assessment purposes.

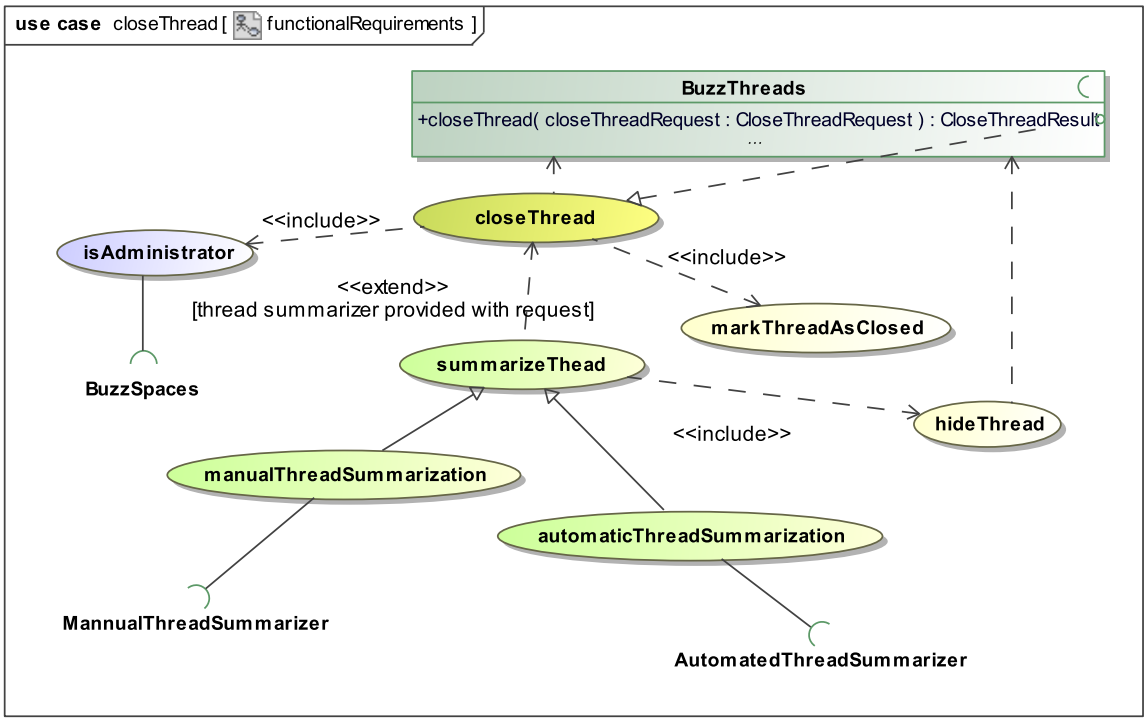


Figure 35: The functional requirements for the closeThread use case.

Threads can be summarized manually or using an automatic thread summarizer.

2.6.2.3.4 Process design Figure 36 is an activity diagram which specifies the process for the `closeThread` use case.

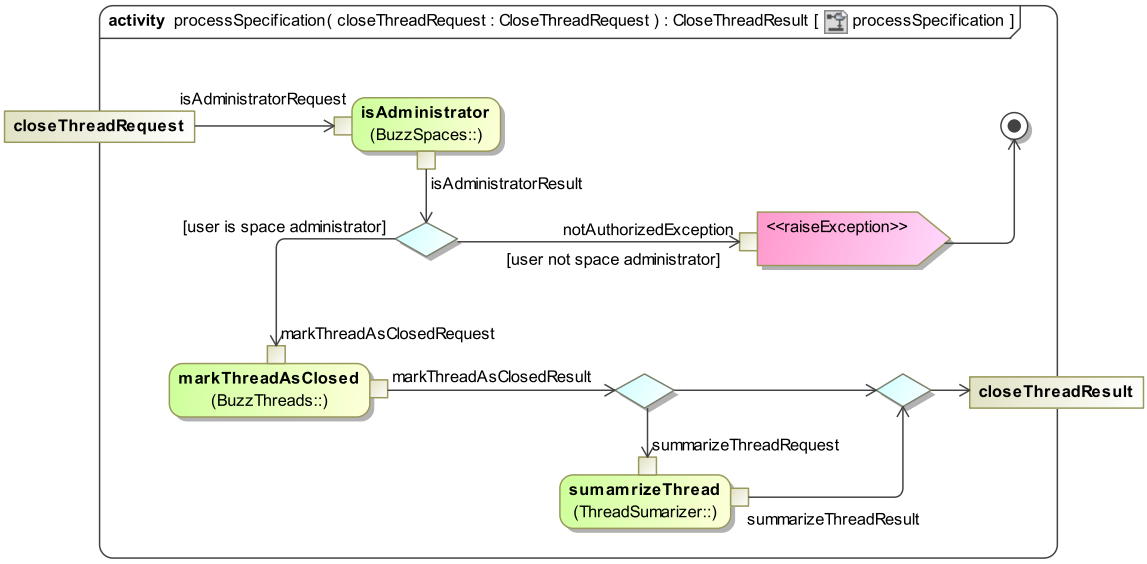


Figure 36: The process specification for the closeThread use case.

2.6.2.4 moveThread — priority:important This use case is used to detach a sub-tree of thread nodes from one thread and add it to another thread. The accessibility of this use case is determined by the `AuthorizationRestrictions` set for the buzz space.

2.6.2.5 hideThread — priority:important This use case is not meant to be used directly by users, but is a functionality required by the moderation module. The selected thread nodes and all its descendant nodes will be marked as hidden and user interfaces are meant not to render them.

2.6.2.6 Threads.queryThread — priority:medium The functionality provided by the *queryThread* function is to provide a versatile way to get all the information of subsets of posts complying with specified restrictions.

2.6.2.6.1 Services contract The `queryThreadRequest` requires no pre-conditions and does not change anything.

The data set returned by this function should contain the following fields for each post in the set:

ParentID Post ID of parent post

Author the ID of the user who created the post

TimeStamp Date and time when the item was posted

Content the text in the post

Status hidden / deleted / closed / etc.

Level depth in the Buzz space tree.

2.6.2.6.2 Functional requirements The use case realizes the service specified in the services contract for `Threads`.

The `getThread` service returns a handle to the thread node and provides access to the thread tree. The default that is returned by `queryThread` is a data set containing all posts in a thread, i.e. all posts in the tree of which the postID is specified. It should, however, be possible to specify each of the following as well as any combination of these:

startDateTime Restrict returned posts to be after this time stamp. Default is the time stamp of the root post in the Buzz space.

endDateTime Restrict returned posts to be before this time stamp. If unspecified all posts are returned.

maxLevel Restrict returned posts to be at most at the specified depth relative to the post. If this value is 0, minLevel will also be 0 only the specified post is returned.

minLevel Restrict return posts to be at least at the specified depth relative to the post. Obviously it has to be less or equal to maxLevel. If both minLevel and maxLevel is 1, only the immediate children is retrieved.

userGroup Restrict returned posts to be limited to a specified user group. Default is all users. It should be easy to limit it to a single user or a set of users who have a specified status or reputation.

phraseSet Restrict returned posts to be only posts that contains all the strings specified in the phrase set. The default is an empty set. If the set is empty all posts are returned.

2.6.3 Domain model

The domain model for this module requires that posts are persistent. Each post has a parent post and may have any number of children posts. All posts within a BuzzSpace are descendants of the root thread of the BuzzSpace.

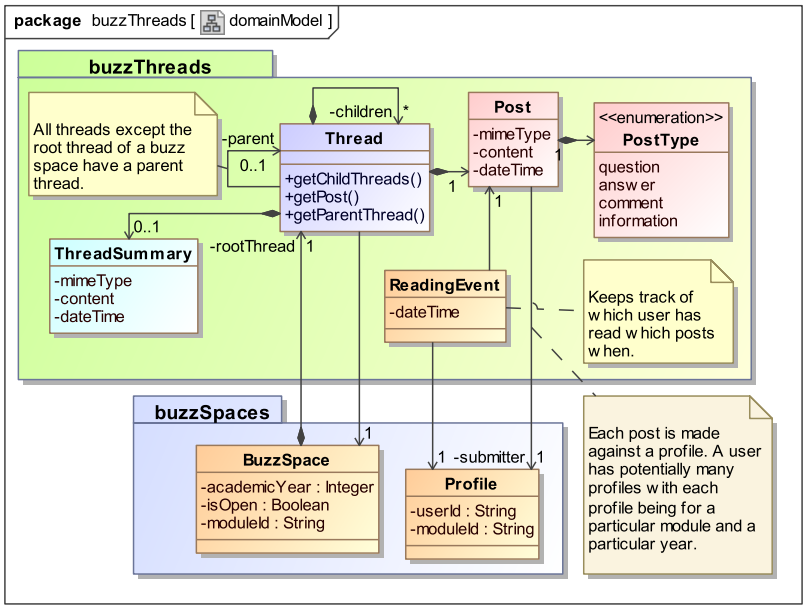


Figure 37: The domain model for the buzzThreads module.

2.7 The Buzz-Status module

Buzz-Status is the module which provides the functionality to assess a number of measures around individual buzz-space contributions, to use these to calculate a *BuzzSpace status* for a profile (i.e. for a user and a specific module), and to restrict access to system functionality based on user status and user role.

2.7.1 Scope

The scope of the *buzzStatus* module is shown in Figure 38.

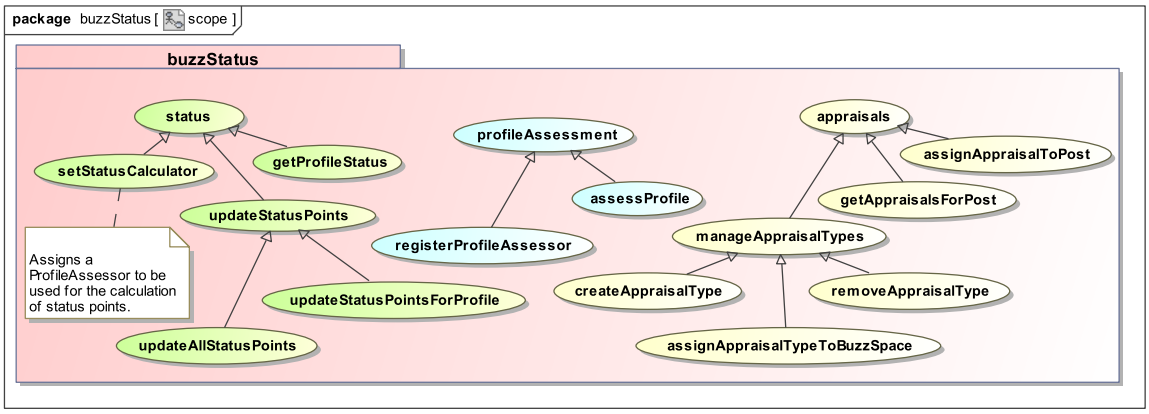


Figure 38: The scope of the buzzStatus module.

2.7.2 Use-cases

The use cases of the status module provide functionality around appraisals of posts, profile assessment, and status calculation.

2.7.2.1 assessProfile The system will enable lecturers to specify different ways in which the profiles of users can be assessed. This is done through a number of pluggable `ProfileAssessors`, each implementing the `ProfileAssessor` interface (see Figure **??**).

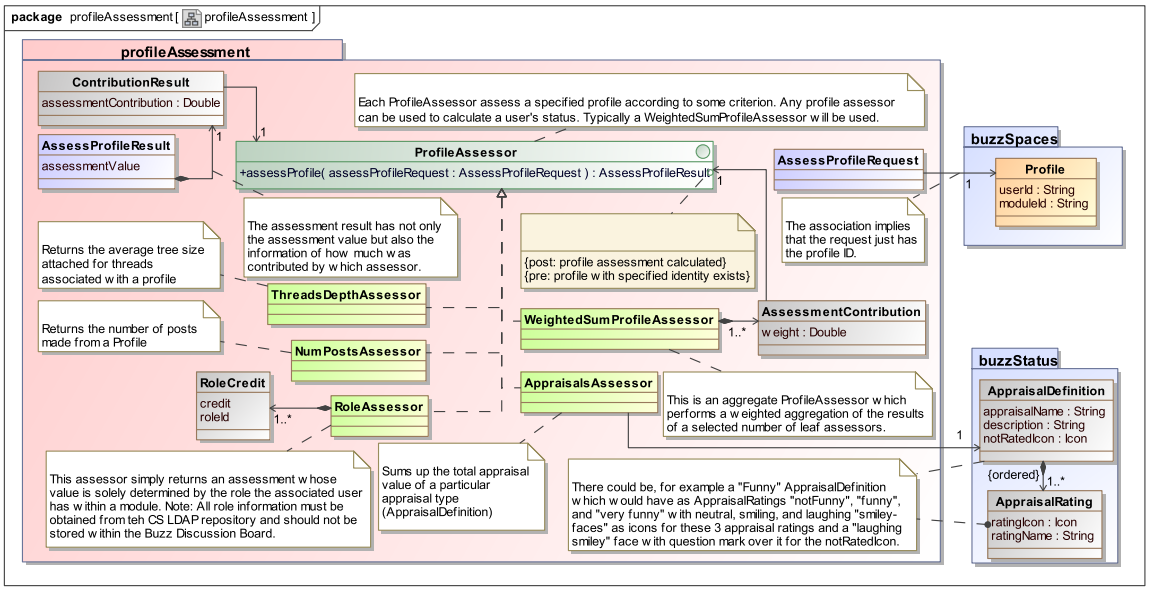


Figure 39: Profile assessors.

2.7.2.2 setStatusCalculator By default the `NumPostsAssessor` is assigned as `statusCalculator` for a *buzz space*, i.e. the status earned by a space member is directly proportional to the number of posts the member has made. This use case is used to assign any other `ProfileAssessor` to be used as the `statusCalculator` for a particular Buzz Space.

2.7.2.2.1 Services contract The status calculator can only be set for a buzz space which is open. The `SetStatusCalulatorRequest` identifies the buzz space for which the status calculator is to be set and the profile assessor which should be used as status calculator.

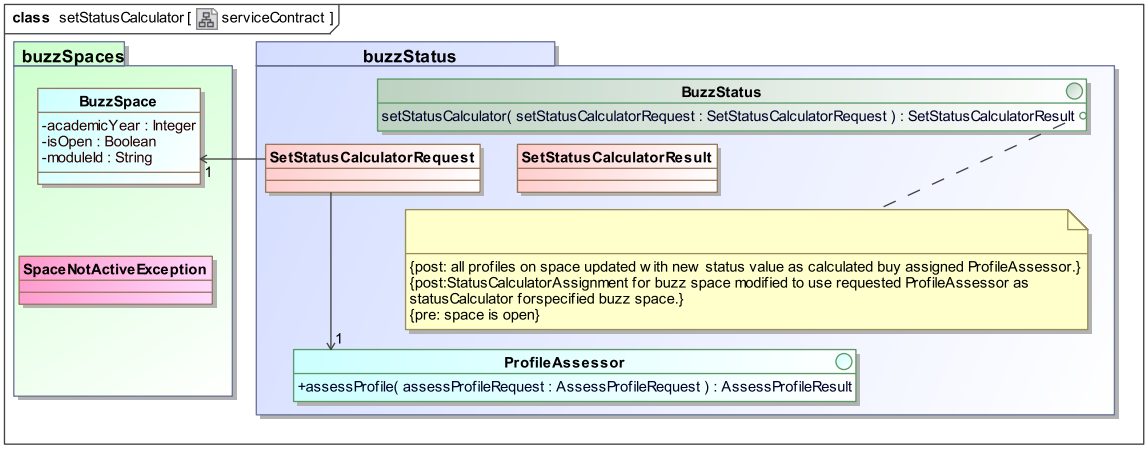


Figure 40: The service contract for the setStatusCalculator use case.

Once the status calculator has been set for a buzz space, the status of all profiles of that space will have been recalculated.

2.7.2.3 getStatusForProfile This is a simple query service which returns the status for a specific profile, i.e. the status a user has on a particular buzz space.

2.7.2.3.1 Services contract The service contract for the `getProfileStatus` use case is shown in Figure 41.

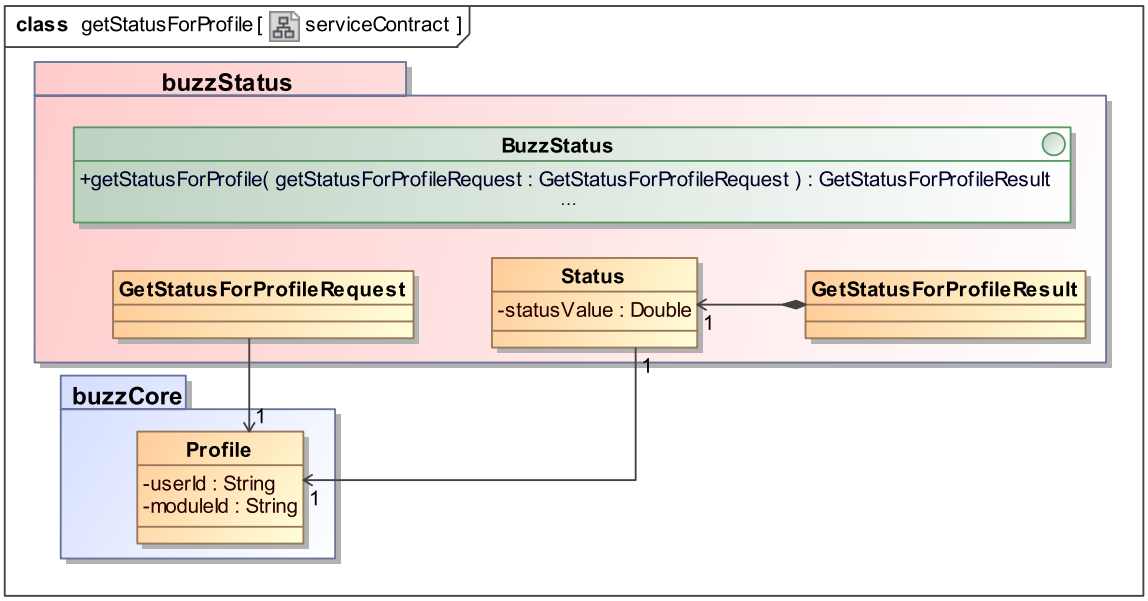


Figure 41: The service returns the status of a particular profile, i.e. the status a user has on a particular buzzSpace.

2.7.2.4 createAppraisalType This use case creates a new appraisal type which can be reused across buzz spaces. s

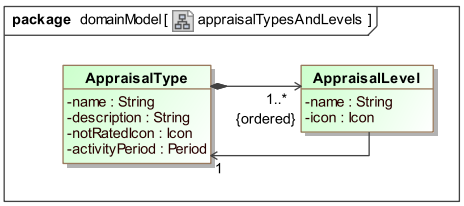


Figure 42: The data structure of appraisal types and levels.

An appraisal type (e.g. `FunnyAppraisalType`) has different appraisal levels (`Hilarious`, `Funny`, `Boring`). These are in an ordered list (0, 1, 2) which correspond to the level values. Each appraisal level has optionally an icon (e.g. laughing, smiling and bored smiley type face) which can be used in the web interface to provide clickable icons to assign appraisals to posts. One can also specify an `notRatedIcon` which can be displayed in the UI when the current user has not yet rated the current post with this particular appraisal type.

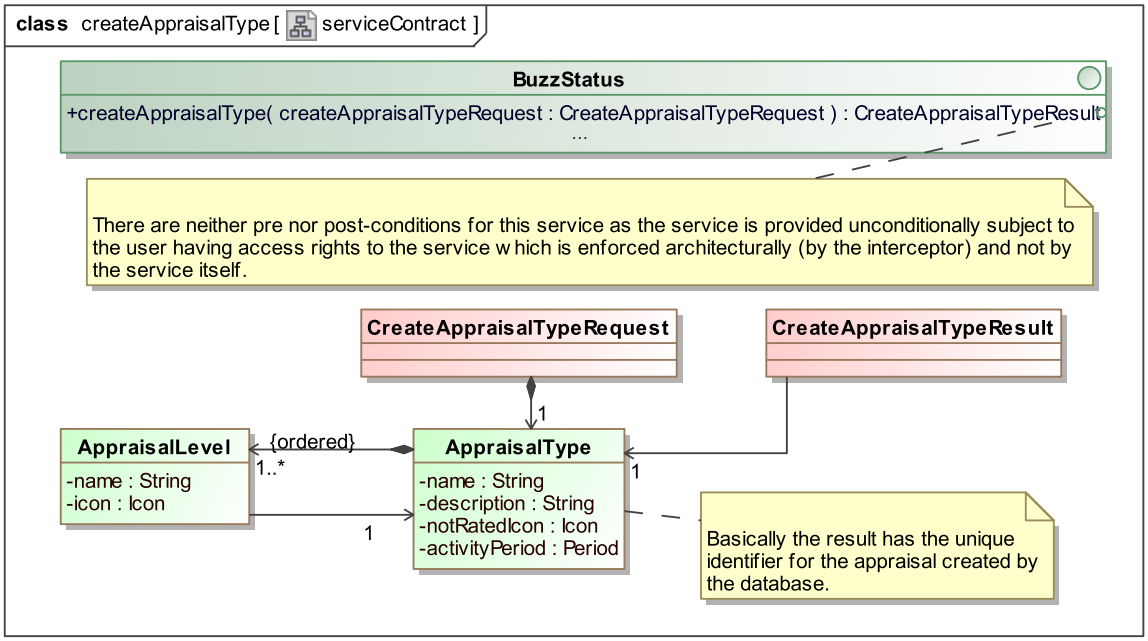


Figure 43: The service contract for creting an appraisal type.

2.7.2.5 activateAppraisalType This use case activates an appraisal type for a specified period on a specified buzz space.

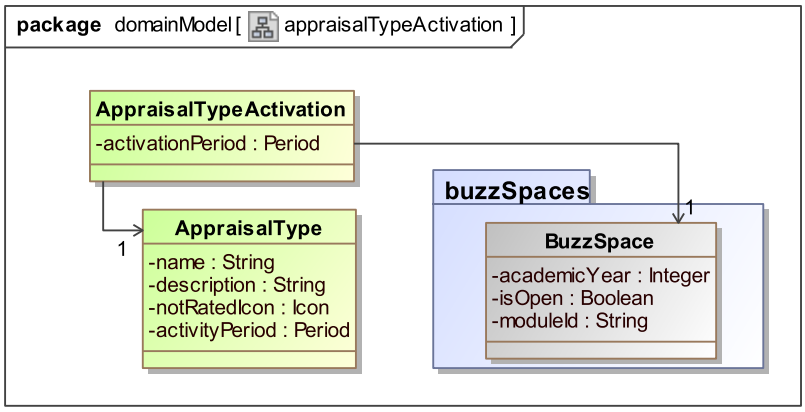


Figure 44: The data structure of appraisal type activation.

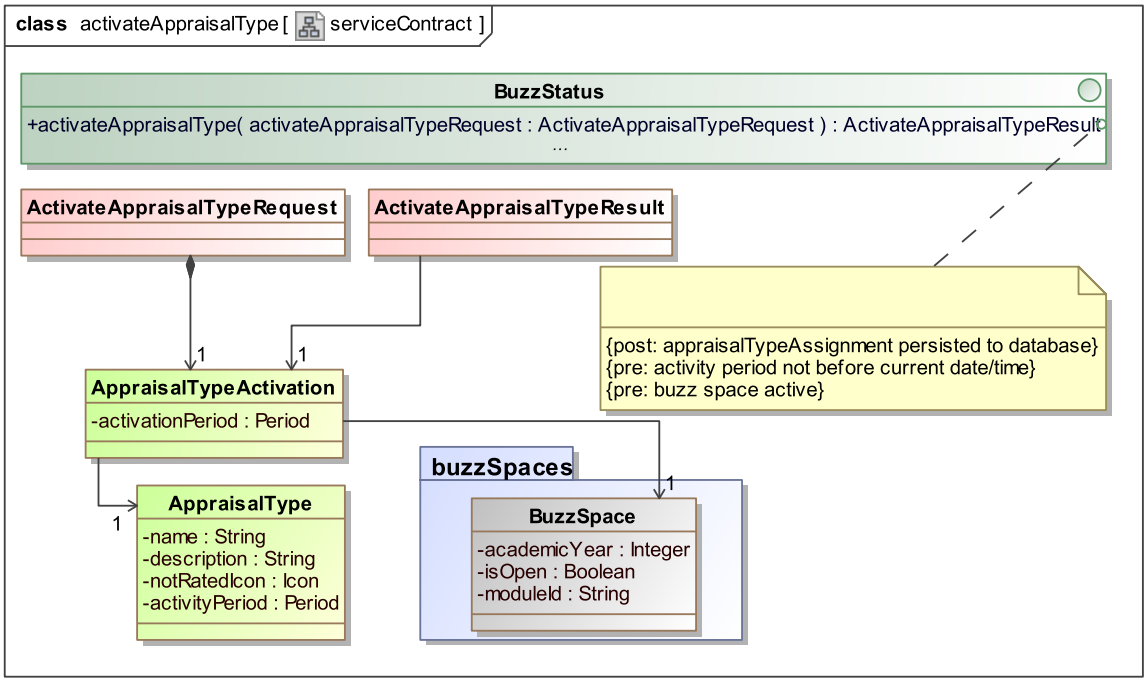


Figure 45: The service contract for creting an appraisal type.

2.7.2.6 assignAppraisalToPost This use case enables a member to assign an appraisal to a post.

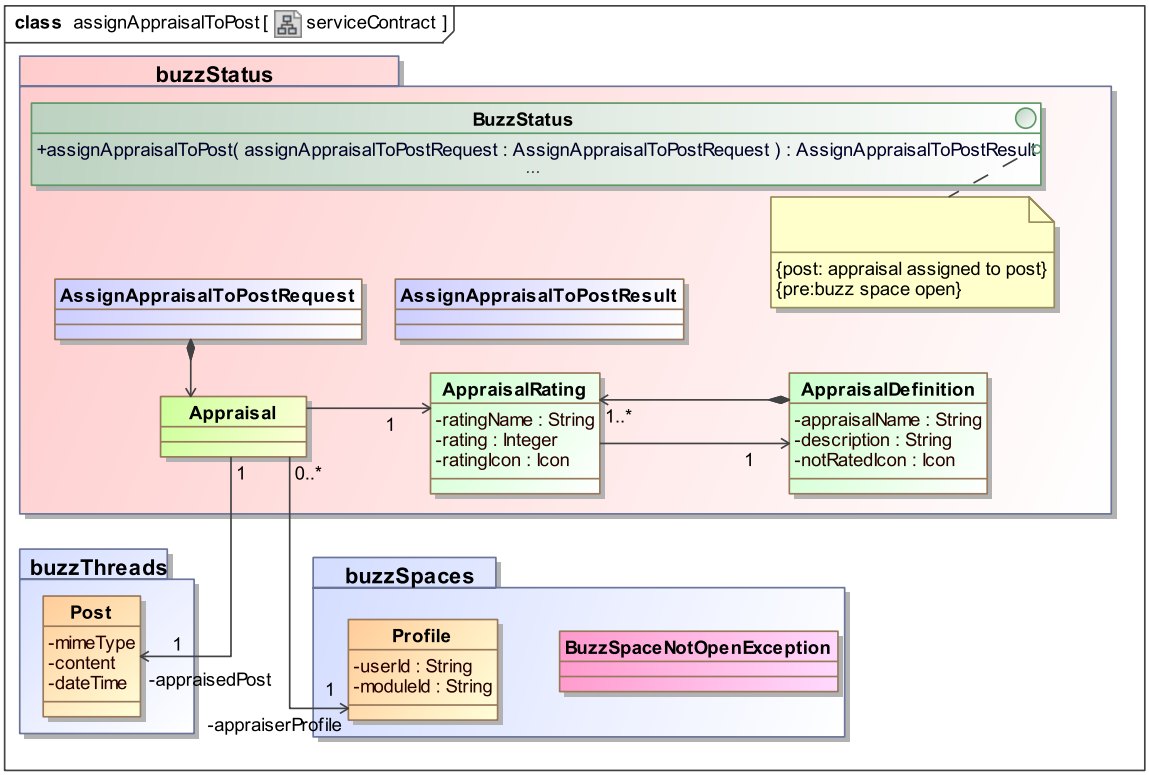


Figure 46: The service contract for assigning an appraisal to a post

2.7.2.6.1 Service contract

2.7.3 Domain model

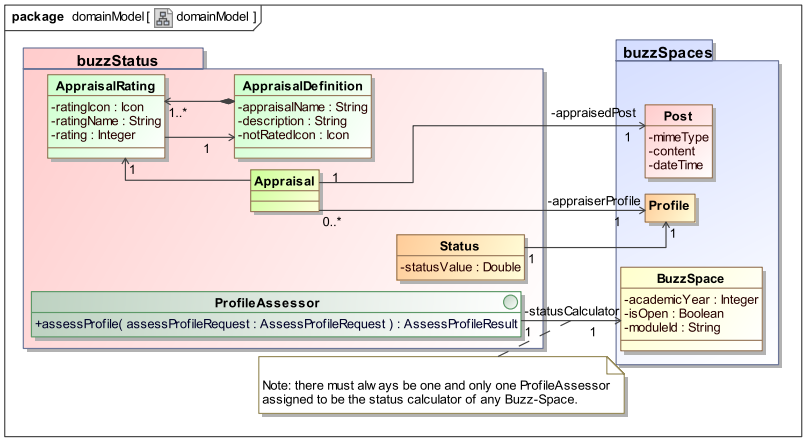


Figure 47: The domain model for the buzzStatus module.

2.8 buzzNotification

The functionality provided by the *buzzNotification* module is focused on registering to receive notification messages for, for example, new posts submitted on a thread, posts of a particular user (tracking a user) and notification of any appraisals one receives for any of ones posts. Notifications are at this stage via email and the email address maintained in the CS database will be used.

2.8.1 Scope

The scope of the *buzzNotification* module is shown in Figure 48.

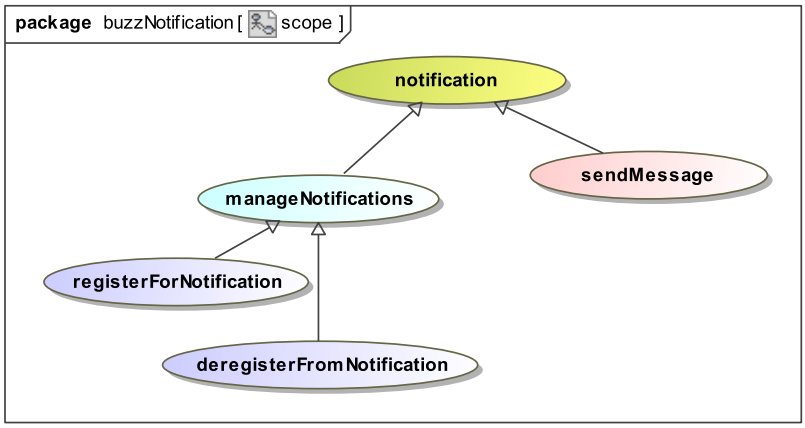


Figure 48: The scope of the buzzNotification module.

2.8.2 Use-cases

The buzz notification module track threads from other modules and users of the buzz system. Its main functional requirements is to track activity threads via email. Below is the overview of the notification module.

2.8.2.1 registerForNotification — priority:niceToHave This use case enables users to register to receive notifications on posts and appraisals. For example, a user may want to receive notification of any post made on a particular thread. If this is requested for the root thread of a buzz space, then the user will receive notification of any post submitted on that entire buzz space.

Users can restrict the notifications to posts made by certain users. If no users (profiles) are specified, then notification for posts made by all users on a thread is requested.

In addition to post notifications, user can also request notifications on any appraisals they receive on a specified thread. Once again, if the root thread for the space is selected, users will receive notification of all appraisals they receive on a buzz space.

2.8.2.1.1 Services contract The service contract for the `registerForNotification` use case is shown in Figure 49.

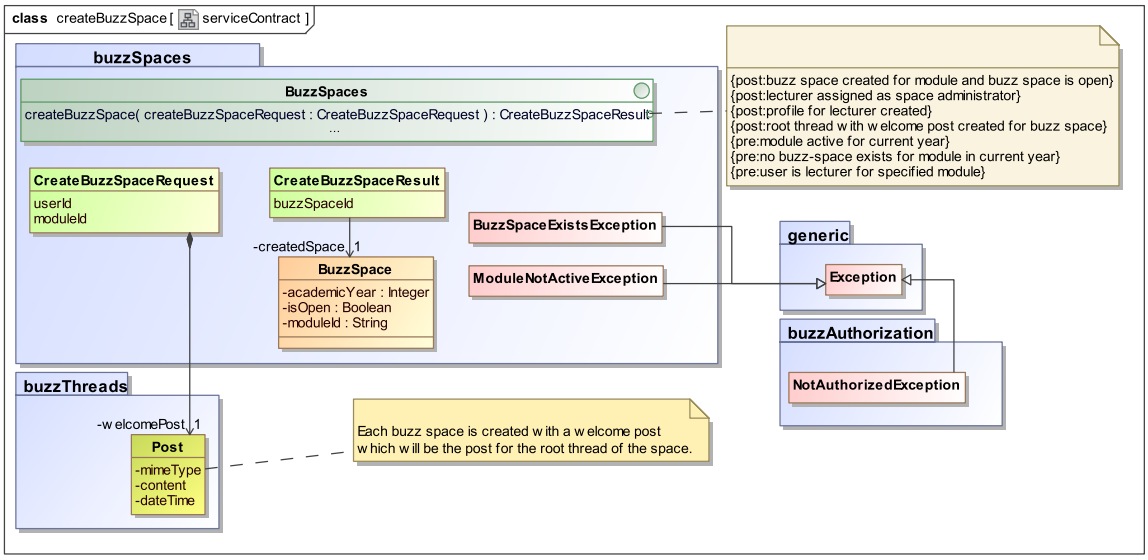


Figure 49: The service contract for theregisterForNotification use case.

2.8.3 The domain model for buzzNotification

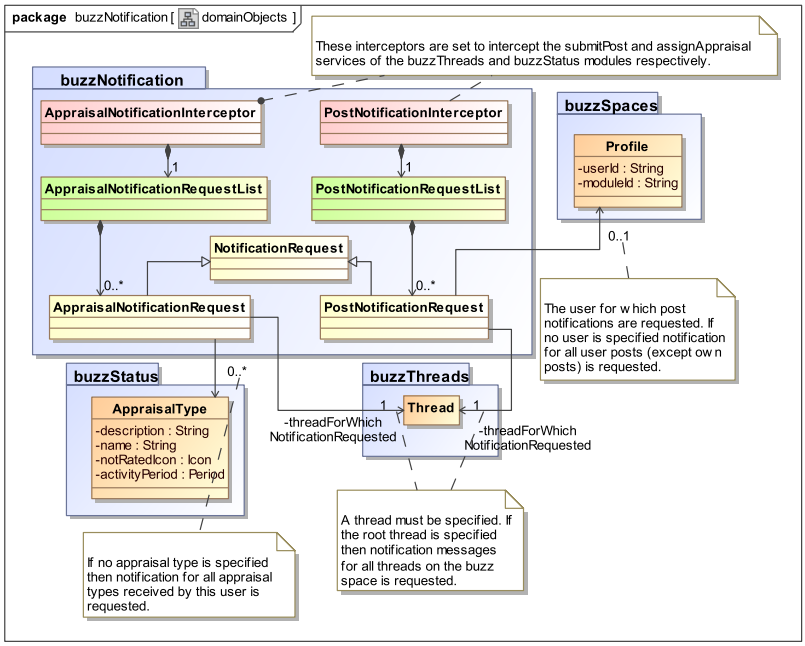


Figure 50: The domainModel for the buzzNotification module.

2.9 The web module

The *Web* module provides a portal based, web responsive front end that allows a user to get access to the application and its facets. The web module provides a web-based human integration channel. The purpose of this module is to make all user services from the business logic layer accessible through web browsers.

As such the purpose of the web module is to capture the information for the request objects in the business logic layer, submit the service requests to that layer and render the responses in a useful way back to the user.

To this end one needs to differentiate between business processes and user work-flows. user-work flows are managed in the presentation layer whilst business processes are encoded within the services of the business logic layer.

2.9.1 Scope

The scope of the web module is the that of making all the system's user services accessible to humans through the web. Hence it needs to cover the user services from the various modules. Lecturers must be able to create buzz spaces, members must be able to submit posts and appraise posts of other users, moderators must be ale to moderate posts and so on.

It is thus important to check that all user services from the different modules are includd within the scope of the web module.

2.9.2 UI requirements

The UI development should be guided throughout by standards and usability guidelines. The following UI requirements need to be balanced

- intuitiveness and usability,
- making the UI interesting and dynamic, and
- making the user interface configurable and personalizable.

2.9.2.1 Login/out The system will allow users to log into or out of the CS systems. This means a user is either logged into the entire buzz space system, being able to see multiple buzz spaces potentially simultaneously). The system will also have a logout functionality where a user would like to reduce the risk that another person uses their computer or device to perform unauthorized actions (e.g. posts) on their behalf.

2.9.2.2 Spaces Users should be able to see all spaces which they could currently be a member of, i.e. the spaces of all those modules which they are currently registered for. There needs to be a differentiation between the spaces for which they have registered and the spaces they have not yet registered for.

2.9.2.3 UI considerations around threads Threads should be viewable in two different ways

1. as an expandable/collapsable tree structure rendering nodes containing posts which have been read in a faint style.
2. as a flat list of posts showing the contents of each post inline.

The user interface needs to facilitate that a user can post on any thread node from either view. If a user posts on the root thread node of a buzz space, a new first-level thread is created on that space.

2.9.2.4 Posts Each post should be rendered with a header area, a content area and a footer area.

The header area should contain

- the title of the post,
- the username of the member who submitted the post,
- the date/time stamp for the post,
- and the current appraisals received by the post.

The content of a post can be encoded as either text or as hypertext markup (HTML). The latter may have embedded images (where images are sourced from uploaded content) and links to uploaded documents and should be rendered in a way which is analogous to a browser rendering the same HTML.

The footer area has the links or buttons to respond to the post and to appraise a post.

Each space can have different and multiple appraisal types. An appraisal type should typically be rendered as a group of icons with each icon representing an appraisal level for that appraisal type. For example, you might have a laughing, smiling and bored phase to represent three levels of funny in a funny appraisal type or thumb symbols for a usefulness appraisal type.

Similarly, in order to submit a new post, the post content must be captured. For that a rich editor supporting the capturing of marked up text (HTML) needs to be used.

2.9.2.5 Profile information A user should be able to view his or her profile information for a space including

- the user name on that space,
- the profile picture the user has on that space,
- the role the user has on that space,
- the status the user has on that space, and
- the number of posts and appraisals the user has made/received on that space.

2.9.2.6 Personalization and configurability A user should be able to configure a dashboard type of view for buzzSpaces where the user should be to assemble a page by positioning different view windows on that page. For example, a user might have on the dashboard a window with his/her profile information, and for each buzz space for which she/he is registered a window with a tree view of the threads.

2.9.2.7 Accessibility of functionality Authorization[3] is enforced at the business logic (services) layer and not at the presentation layer. Access to a service may depend on a user's role on a buzz space as well as on a user's status on that space.

Nevertheless, the user interface should only enable the user to access functionality which he/she is authorized to use. This can be, for example, done by greying out buttons or menu items which are used to access functionality which the user is currently not authorized to use. The `buzzAuthorization` module provide a `isAuthorized` service which can be used y the web module to determine whether a user may currently use some functionality.

2.10 The Buzz Reporting module

The functionality provided by the *buzzReporting* module includes the following:

1. It provides statistical information that can be used in the interface to display facts about the average user and how the logged-in user compares with the average.
2. It provides ways to gather data that is in the persistent store and present it in a format that is usable by other modules that are external to Buzz.
3. It provides functionality to alter record sets in a Buzz space by uploading the relevant information that is stored in a csv file.

[3]Please read the section on the buzzAuthorization module to understand this section

2.10.1 Use-cases

The *reporting* module provides services to gather, import and export information about posts in specified subsets.

2.10.1.1 Treads.getThreadStats — priority:medium The functionality provided by the *getThreadStats* function is to provide a versatile way to get statistical information of subsets of posts complying with specified restrictions.

2.10.1.1.1 Functional requirements The parameters passed to the function is

- a set of posts returned by the *Threads.queryThread* function.
- action keyword

The set of posts returned by the *Threads.queryThread* function is analysed according to the specified action keyword. The following describes the result returned upon each of the action keywords:

Num A count of the entries in the dataset that was created.

MemCount A count of the number of members who are the creators of posts in the dataset.

MaxDepth The maximum depth of a post in the queried thread tree.

AvgDepth The average depth of a post in the queried thread tree.

2.10.1.2 Get Thread Appraisal — priority:medium The functionality provided by the *getThreadAppraisal* function is to provide a versatile way to get detailed or statistical information of subsets of posts complying with specified restrictions and their associated appraisals assigned by specified members.

2.10.1.2.1 Functional requirements The parameters passed to the function is

- a set of posts returned by the *Threads.queryThread* function.
- a set of members
- a set of appraisals
- action keyword

A data set is created containing entry for each valid member-appraisal-post combination. i.e. for each member in the set that may assign an appraisal in the set to a post in the set a separate entry is created.

Each entry has a field that should contain an ordinal number that represent a level of the specified appraisal. This field is empty for posts for which a member has not assigned an appraisal.

The following describes the result returned upon each of the action keywords:

All A detailed dataset containing all post detail, memberID, appraisalID and appraisal value (ordinal number).

Sum The sum of all appraisal values for the entries in the dataset that was created.

Avg The average of all appraisal values for the entries in the dataset that was created.

Max The maximum of all appraisal values for the entries in the dataset that was created.

Min The minimum of all non-empty appraisal values for the entries in the dataset that was created.

Num A count of all non-empty appraisal values for the entries in the dataset that was created.

2.10.1.3 Export Thread Appraisal — priority:medium The functionality provided by the *exportThreadAppraisal* function is to realise an off-line facility to apply a manual appraisal. It creates the dataset to be used that can be edited off-line and allow updates to be inserted through the *importThreadAppraisal* function.

2.10.1.3.1 Functional requirements An external file is created containing data generated by the *getThreadAppraisal* function only if the function executed with the **All** action keyword.

2.10.1.4 Import Thread Appraisal — priority:medium The functionality provided by the *importThreadAppraisal* function is to realise an offline-facility to apply a manual appraisal. It is dependant on the *exportThreadAppraisal* function.

2.10.1.4.1 Functional requirements Data in an external file that was created using the *exportTreadAppraisal* function is used. It is required that the data set is associated with only one member and only one specified appraisal to be eligible for import.

A record contains all detail about the post along with a field that should contain an ordinal number that represents the levels of the specified appraisal.

It is assumed that the detail of the posts are not edited. Edits to this data is ignored when importing a thread appraisal.

For each record the **assignAppraisalToPost** function is applied. The appraisal level as stored in the file for each post is updated as an appraisal assigned by the member associated with the data set. Although it can be assumed that the member and the appraisal is valid for each post in the set, their validity should be checked. If invalid an exception is raised and the service not delivered. If the appraisal level is outside the range specified for the appraisal an exception is raised and the service not delivered.

2.10.1.5 Export Thread — priority:low The functionality provided by the *exportThread* function is to provide means to backup the content of a thread or subset of a thread in a serialised text file.

2.10.1.5.1 Functional requirements An external file is created containing data generated by the *Treads.queryThread* function.

2.10.1.6 Import Thread — priority:low The functionality provided by the *importThread* function is to provide means to restore the content of a thread or subset of a thread that was stored using the *exportThread* function.

2.10.1.6.1 Functional requirements An external file that was created by the *exportThread* function is restored in the Buzz space. A post is added to the Buzz space only if it is not in the Buzz space i.e. it should be ensured that duplicates of posts are not created when using this function.

2.11 The Buzz-Android-Client module

This module is out of scope for the mini-project.

2.12 The Buzz-Hamster-Integration module

This module is out of scope for the mini-project.