Meta Resource Management System

Design Model

René Freude (707168) <rene.freude@web.de>
Henry Hinze (681566) <henry@lilit.net>
Daniel Sadilek (707297) <sadilek@tfh-berlin.de>
Stephan Weiß (706830) <steph.weiss@web.de>

2003-11-03

Revision History

Revision 0.1	2003-05-19
Initial public release.	
Revision 0.2	2003-06-09
Corrected some cardinalities, extended descriptions, added operations.	
Revision 0.3	2003-06-15
Added "user logs in" sequence diagram.	
Revision 0.4	2003-06-23
Extended from static model to analysis model.	
Revision 0.5	2003-10-06
Incorporated optional feature "Resource Reservation" (see appendix of this document for the use cases derived from that feature); refined package structure; introduced distinction between physical resource containment hierarchy and resource usage.	
Revision 0.6	2003-10-20
Extended from analysis model to design model.	
Revision 0.7	2003-11-03
Refined model for server and client; added sequence diagrams for verification.	

This document contains the class diagrams and class descriptions that resulted from the static analysis and the design analysis as well as sequence diagrams and state chart diagrams that we used to verify the class model.

Table of Contents

1. Overview

So far, the design model only covers a subset of all use cases - completely defined in the document “Use Cases”. The remaining use cases will be considered during the next revisions of this document. The use cases covered so far are:

- User logs in
- Create filtered collection of resource entries
- Edit resources

This document is organized along the package structure of the MRMS. Every package describes one aspect of the system:

- *model.entity*: The MRMS can handle resources and the employees; the attributes that are to be saved for each resource type and employee type can be configured by an administrator. This common functionality is pulled up to the super type Entity. The package model.entity contains the classes to handle entities (resources, employees) and their attributes (number, text, boolean).
- *model.linkage*: Resources and employees do not exist detached. Resources can be organized in a physical containment structure (e.g. a room contains workplaces, workplaces contain a computer, and so on) and resources can be used by employees. The package model.linkage contains the classes that are necessary to represent these links.
- *model.user*: The users of the system need different access rights according to the role they play in the business. The package model.user contains the classes that represent the rights users have to create and delete entities, edit their attributes and create links.
- *model.filter*: Creating a filtered collection of resources is a complex function that is required in different use cases. The package model.filter contains the classes needed to configure a filter with constraints and execute it.
- *client*: Classes needed to realize an interaction between the user and the MRMS.
- *server*: Classes for the MRMS server.

(The classes imported from others packages are colored yellow.)

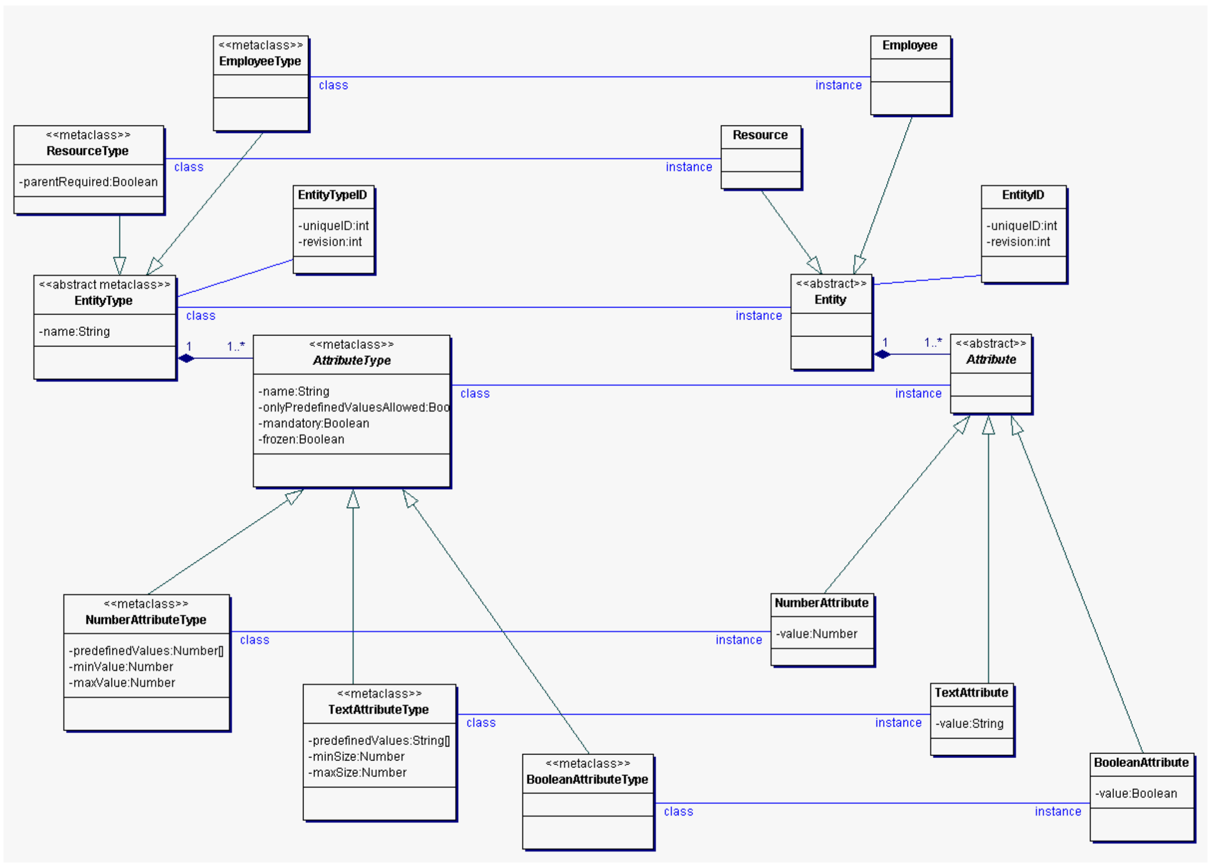
2. Package: model

This package does not contain any classes but only the subpackages entity, linkage, user and filter.

2.1. Package: model.entity

The following diagram depicts the classes to handle entities (resources, employees) and their attributes (number, text, boolean).

Figure 1. Entity Classes



2.1.1. Class: EntityType

Description	An *EntityType* has a name and specifies (by composition) the *Attributes* that an *Entity* of this type has, it references a unique *EntityTypeID*.
Attributes	*name* (String): the name of the *EntityType*
Operations	---

2.1.2. Class: EntityTypeID

Description	An *EntityTypeID* is a unique identifier for an *EntityType*.
Attributes	*uniqueID* (int): an integer which is unique within the set of all *EntityTypes*
	revision (int): an integer which is incremented by the server with every change; this field is used by the server to verify that the *EntityType* a client refers to has not changed since the client received the *EntityType*'s data
Operations	---

2.1.3. Class: ResourceType

Description	A *ResourceType* is a specialised *EntityType* for defining *Resources*.
Attributes	*parentRequired* (Boolean): specifies whether instances of this *ResourceType* must have a parent Resource

Operations	---

2.1.4. Class: EmployeeType

Description	An *EmployeeType* is a specialised *EntityType* for defining *Employees*.
Attributes	---
Operations	---

2.1.5. Class: Entity

Description	An *Entity* is composed of its *Attributes* and is an instance of an *EntityType* which specifies which *Attributes* the *Entity* may have, it references a unique *EntityID*.
Attributes	---
Operations	---

2.1.6. Class: EntityID

Description	An *EntityID* is a unique identifier for an *Entity*.
Attributes	*uniqueID* (int): an integer which is unique within the set of all *Entitys*
	revision (int): an integer which is incremented by the server with every change; this field is used by the server to verify that the *Entity* a client refers to has not changed since the client received the *Entity*'s data
Operations	---

2.1.7. Class: Resource

Description	A *Resource* is a specialised *Entity* for representing real-life-resources and is an instance of a *ResourceType* which specifies if this *Resource* must have a parent *Resource* within the Resources-Containment-Hierarchy.
Attributes	---
Operations	---

2.1.8. Class: Employee

Description	An *Employee* is a specialised *Entity* for representing users of real-life-resources and is an instance of an *EmployeeType.*
Attributes	---
Operations	---

2.1.9. Class: AttributeType

Description	Abstract base class for attribute types that an *EntityType* is composed of.
Attributes	*name* (String): the name of the *AttributeType*
	onlyPredefinedValuesAllowed (Boolean): if true, the user may only select the predefined values for an *Attribute* that has this type; if false, he may enter another value as well
	mandatory (Boolean): if true, the user must enter a value for *Attributes* of this type
	frozen (Boolean): if true, the user may not change the value of *Attributes* of this type
Operations	---

2.1.10. Class: BooleanAttributeType

Description	Concrete *AttributeType* for logical property characterisation of an *Entity*.
Attributes	*value* (Boolean): logical property characterisation of an *Entity*
Operations	---

2.1.11. Class: NumberAttributeType

Description	Concrete *AttributeType* for *NumericalAttributes*.
Attributes	*predefinedValues* (Number[]): an array specifying predefined values for *Attributes* of this type
	minValue (Number): the minimum value *Attributes* of this type may have
	maxValue (Number): the maximum value *Attributes* of this type may have
Operations	---

2.1.12. Class: TextAttributeType

Description	Concrete *AttributeType* for *TextAttributes*.
Attributes	*predefinedValues* (String[]): an array specifying predefined values for *Attributes* of this type
	minSize (Number): the minimum number of characters *Attributes* of this type may have
	maxSize (Number): the maximum number of characters *Attributes* of this type may have
Operations	---

2.1.13. Class: Attribute

Description	Abstract base class for *Attributes* that an *Entity* is composed of.
Attributes	---
Operations	---

2.1.14. Class: BooleanAttribute

Description — Concrete *Attribute* for a boolean property characterisation of an *Entity*.

Attributes — *value* (Boolean): numerical property characterisation of an *Entity*

Operations — ---

2.1.15. Class: NumberAttribute

Description — Concrete *Attribute* for a numerical property characterisation of an *Entity*.

Attributes — *value* (Number): numerical property characterisation of an *Entity*

Operations — ---

2.1.16. Class: TextAttribute

Description — Concrete *Attribute* for a textual property characterisation of an *Entity*.

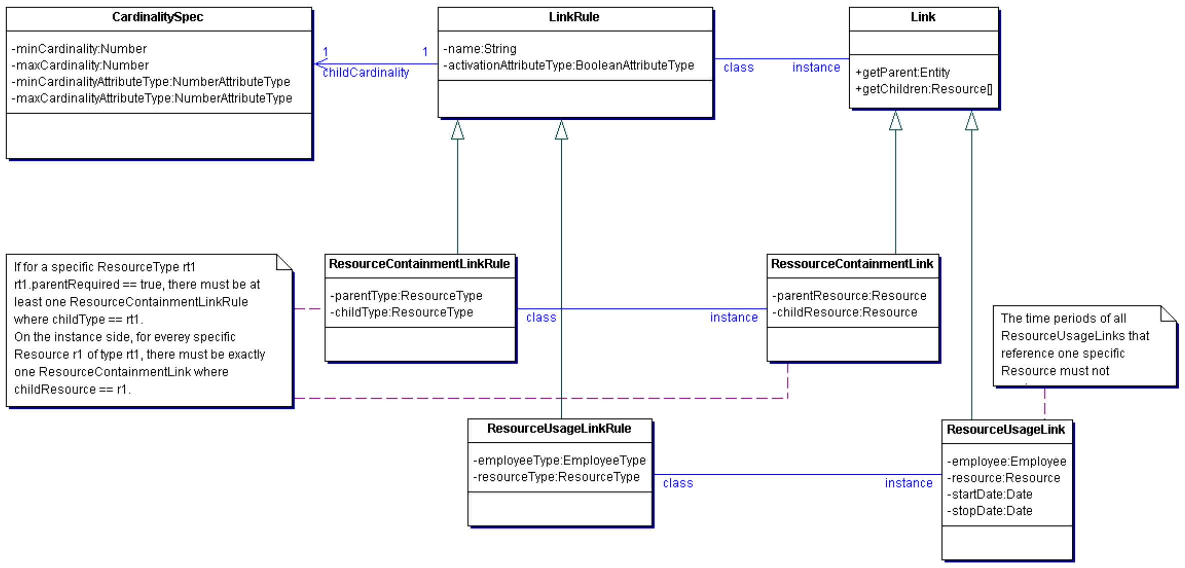
Attributes — *value* (String): textual property characterisation of an *Entity*

Operations — ---

2.2. Package: model.linkage

The following diagram depicts the classes that are necessary to represent the physical containment links between resources and resources and the usage links between resources and employees.

Figure 2. Linkage Classes



If for a specific ResourceType rt1 rt1.parentRequired == true, there must be at least one ResourceContainmentLinkRule where childType == rt1.
On the instance side, for everey specific Resource r1 of type rt1, there must be exactly one ResourceContainmentLink where childResource == r1.

The time periods of all ResourceUsageLinks that reference one specific Resource must not

2.2.1. Class: LinkRule

Description — A *Link Rule* defines the characteristics of a consistent *Link*. Both the physical containment structure of the resources as well as the usages of the resource by the users can be modelled as

	links. In both cases the corresponding link rules have the characteristic that the cardinality of one side is 1; for the physical containment links this side is the parent resource type and for the usage links this side is the employee type. The other side of the link rule can have an arbitrary cardinality (i.e. the number of children a parent has in the physical containment structure as well as the number of resources an employee may use is not constrained by the system but can be customized by the administrator); this cardinality is contained in the *CardinalitySpec* referenced by the *LinkRule*. A *LinkRule* can reference an *BooleanAttributeType* of the parent resource / using employee; in this case *Link*s of this *LinkRule* can only be created for those *Resource*s / *Employee*s where the corresponding *BooleanAttribute* is true.
Attributes	*name* (String): name of the *LinkRule*
Operations	---

2.2.2. Class: Link

Description	Base class for *ResourceContainmentLink* and *ResourceUsageLink*.
Attributes	---
Operations	---

2.2.3. Class: ResourceContainmentLinkRule

Description	A *ResourceContainmentLinkRule* defines the characteristics of a consistent *ResourceContainmentLink*. It references two *ResourceTypes* which may be linked together by a *ResourceContainmentLink.*
Attributes	---
Operations	---

2.2.4. Class: ResourceContainmentLink

Description	A *ResourceContainmentLink* references two *Resources* that are linked together by it; one resource takes the parent role, the other is its child in the pysical containment. Its consistency is checked against the *ResourceContainment LinkRule* references.
Attributes	---
Operations	---

2.2.5. Class: ResourceUsageLinkRule

Description	A *ResourceUsageLinkRule* defines the characteristics of a consistent *ResourceUsageLink*. It references one *ResourceType* and one *EmployeeType* whose instances may be linked together by a *ResourceUsageLink.*
Attributes	---
Operations	---

2.2.6. Class: ResourceUsageLink

Description	A *ResourceUsageLink* references one *Resource* and one *Employee* that are linked together by it. Its consistency is checked against the *ResourceUsageLinkRule* it references. There may be more than one *ResourceUsageLink* at a *Resource*; but only one of can be active at a certain time.
Attributes	*startDate* (Date): Time when usage starts.
	stopDate (Date): Time when usage expires.
Operations	---

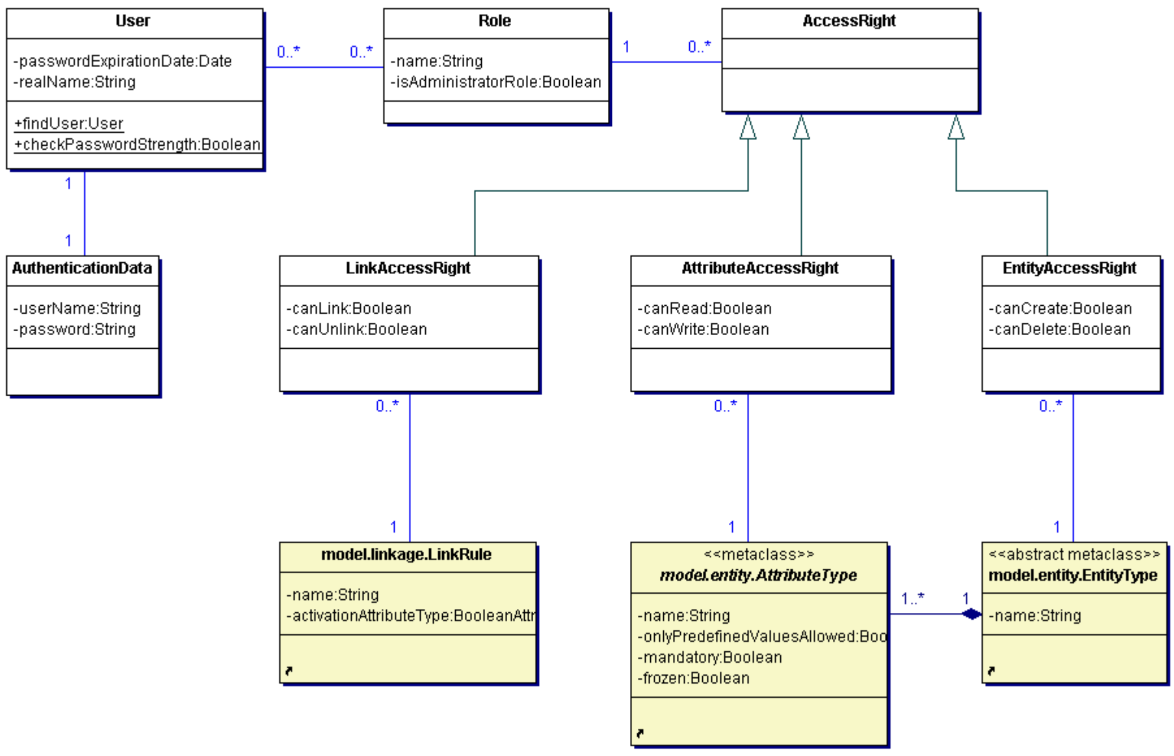
2.2.7. Class: CardinalitySpec

Description	Specifies the minimum and maximum cardinality for a certain *ResourceType*, referenced by a *ResourceUsageLinkRule* or a *ResourceContainmentLinkRule*. Example: A *ResourceContain-mentLinkRule* has two ends *ResourceType*1 (parent) and *ResourceType*2 (child). The *Resource-Type*1 always has the cardinality 1 while *ResourceType*2 has the cardinality min=1 and max=4, this means that one specific *Resource* of *ResourceType*1 must have at least 1 and may have up to 4 Links to *Resources* of *ResourceType*2. The *CardinalitySpec* may also reference a *Number-AttributeType* of the *ResourceType*1.
Attributes	*minCardinality* (Number): value for the minimum cardinality; will be ignored when there is a “min”-reference to a *NumberAttributeType*, in this case the *NumberAttribute*'s value will be used instead
	maxCardinality (Number): value for the maximum cardinality; will be ignored when there is a “max”-reference to a *NumberAttributeType*, in this case the *NumberAttribute*'s value will be used instead
Operations	---

2.3. Package: model.user

The following diagram depicts the classes for user and access rights management of the MRMS.

Figure 3. User and Access Rights Management Classes



2.3.1. Class: AuthenticationData

Description	Value class, encapsulating the authentication data of a user.
Attributes	*userName* (String): the user's name
	password (String): the user's password
Operations	---

2.3.2. Class: User

Description	Class for user accounts of the MRMS. Its instances may play *Roles* in the system.
Attributes	*passwordExpirationDate* (Date): date after which the user has to enter a new password
	realName (String): real name of the user
Operations	

- static checkPasswordStrength(password: String): Boolean

Effect	Checks, if the given password String is strong enough (minimum length, mixed letters and numbers, ...) to be accepted by the system.
Parameters	*password*: the password to be checked
Return	The boolean value *true*, iff the password is strong enough.
Exceptions	---

Actor	Control class of the use case “User changes password”.

- static findUser(authData: AuthenticationData): User

Effect	Searches the system for a *User* matching the given *AuthenticationData*.
Parameters	*authData*: the *AuthenticationData* to search for
Return	If a matching *User* object could be found it is returned, otherwise the operation returns the *null* pointer.
Exceptions	---
Actor	Control class of the use case “User logs in”.

2.3.3. Class: Role

Description	A *Role* defines which *AccessRights* its players (*Users*) have.
Attributes	*name* (String): name of the *Role*
	isAdministratorRole (Boolean): defines if *Users* of the *Role* have administration rights
Operations	---

2.3.4. Class: AccessRight

Description	Abstract base class for access rights. If a *Role* references an *AccessRight* it has this *AccessRight*. *Users* have the *AccessRights* which the *Roles* they play have.
Attributes	---
Operations	---

2.3.5. Class: ResourceAccessRight

Description	Concrete *AccessRight* that defines owner's authority of working with *Resources* that are of a specific *ResourceType*.
Attributes	*canCreate* (Boolean): defines if *Resource*s of the referenced *ResourceType* may be created
	canDelete (Boolena): defines if *Resource*s of the referenced *ResourceType* may be deleted
Operations	---

2.3.6. Class: AttributeAccessRight

Description	Concrete *AccessRight* that defines owner's authority of working with *Atrributes* of a specific *At-*

	tributeType that belongs to a specific *ResourceType*.
Attributes	*canRead* (Boolean): defines if *Attributes* of the referenced *AttributeType* may be read
	canWrite (Boolean): defines if *Attributes* of the referenced *AttributeType* may be written
Operations	---

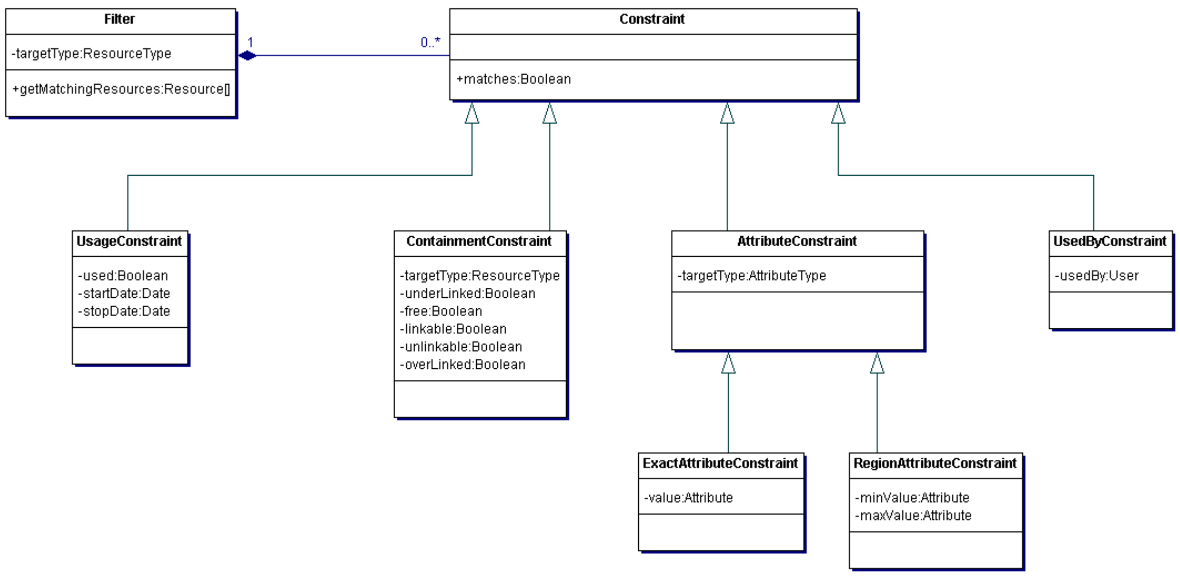
2.3.7. Class: LinkAccessRight

Description	Concrete *AccessRight* that defines owner's authority of creating and deleting *Links* according to a specific *LinkRule*.
Attributes	*canLink* (Boolean): defines if *Links* according to the referenced *LinkRule* may be created
	canUnlink (Boolean): defines if *Links* according to the referenced *LinkRule* may be deleted
Operations	---

2.4. Package: model.filter

The following diagram depicts the classes needed to configure a filter and get a collection of *Resource*s out of it.

Figure 4. Filter Classes



2.4.1. Class: Filter

Description	A *Filter* is used to get a subset of all *Resources* of the referenced *ResourceType*. The *Filter* is defined by the *Constraints* it is composed of.
Attributes	---
Operations	

- getMatchingResources(): Resource[]

Effect	Searches the system for *Resources* matching the referenced *Constraints*.
Parameters	---
Return	An array of the matching *Resources*.
Exceptions	---
Actor	Control class of the use case “Create filtered collection of resource entries”.

2.4.2. Class: Constraint

Description	Abstract base class for constraints. *Constraints* are used by a *Filter* to describe a specific state that *Resource* must fulfill to pass.
Attributes	---
Operations	

- matches(resource: Resource): Boolean

Effect	Tests, if the given *Resource* matches this *Constraint*.
Parameters	*resource*: the *Resource* to be tested
Return	The boolean value *true*, iff the given *Resource* matches this *Constraint*.
Exceptions	---
Actor	Class *Filter*.

2.4.3. Class: AttributeConstraint

Description	An *AttributeConstraint* is a concrete *Constraint* that checks whether an *Attribute* of the referenced *AttributeType* is either equal to the referenced *Attribute* or lays between the two referenced min- and max-*Attributes*.
Attributes	---
Operations	---

2.4.4. Class: ContainmentConstraint

Description	A *ContainmentConstraint* is a concrete *Constraint* that checks whether a *Resource* matches the physical containment state that is described by the following attributes. A *ContainmentConstraint* references the *LinkRule* it refers to. If in this *LinkRule* the *ResourceType* that is to be filtered has (1) the parent role minimum and maximum cardinality are taken from *LinkRule*'s *CardinalitySpec* and refer to the number of children; if it has (2) the client role then min = max = 1 iff the field requiresParent of the *ResourceType* is *true*, min = max = 0 otherwise.
Attributes	*underLinked* (Boolean): cur < min

	free (Boolean): cur = 0
	linkable (Boolean): cur < max
	unlinkable (Boolean): cur >= max
	overLinked (Boolean): cur > max
Operations	---

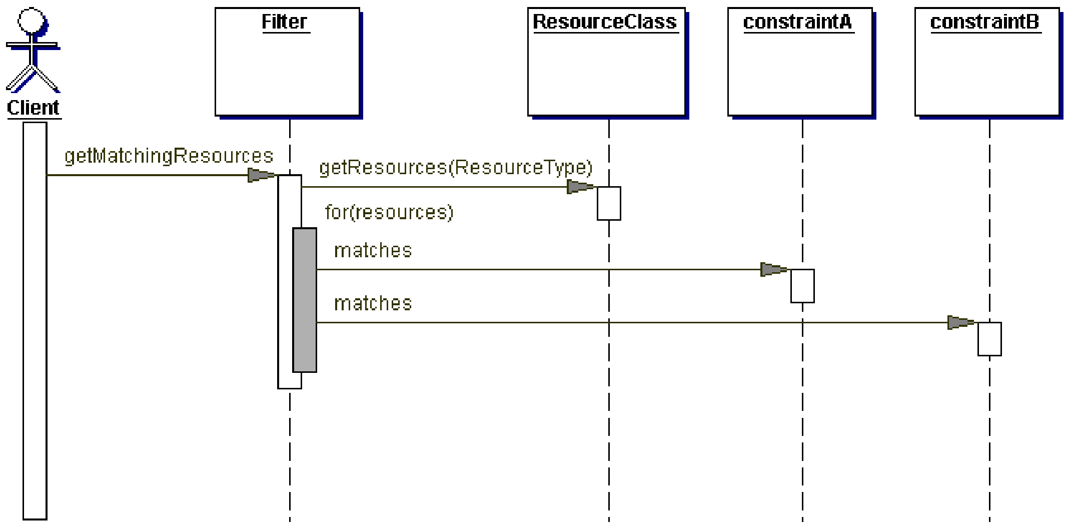
2.4.5. Class: UsageConstraint

Description	A *UsageConstraint* is a concrete *Constraint* that checks whether a *Resource* is used or unused in a given time period.
Attributes	*used* (Boolean): Defines whether the filtered *Resource*s have to be used or unused in the given time period.
	startDate (Date): Start time of the time time period.
	stopDate (Date): End time of the time period.
Operations	---

2.4.6. Sequence diagram: Filter.getMatchingResources

The following diagram shows how a filter determines the matching resources.

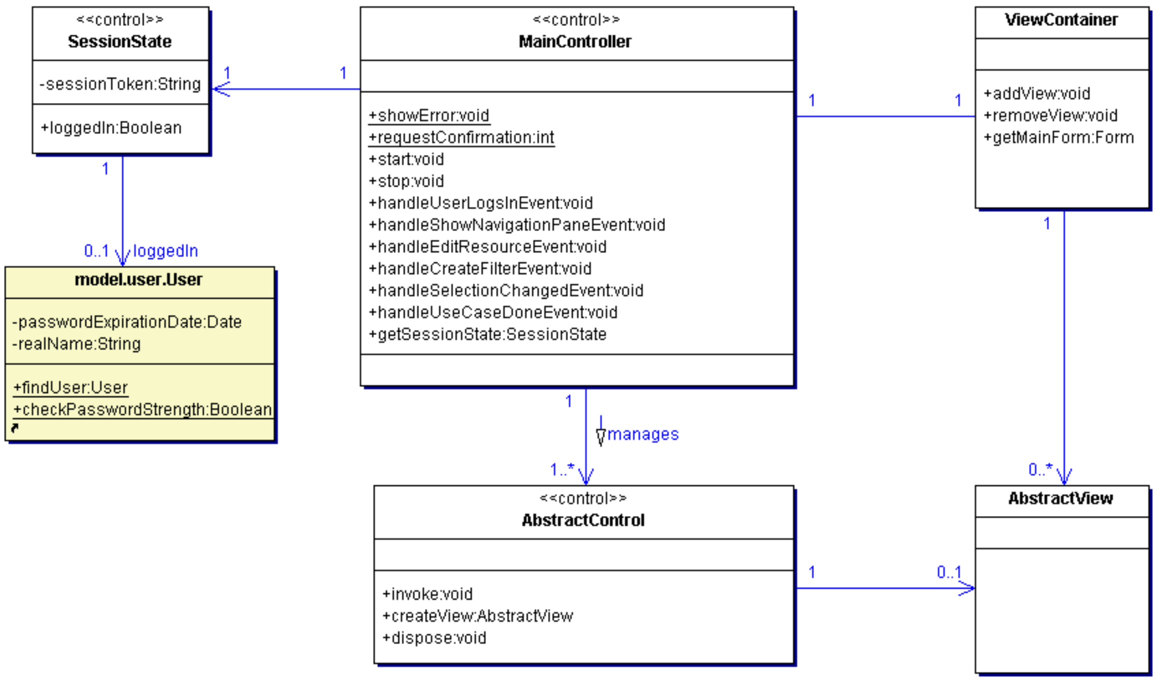
Figure 5. Sequence: Filter.getMatchingResources



3. Package: client

The following diagram depicts the main classes needed to realize an interaction between the user and the MRMS.

Figure 6. Control and Boundary Classes



3.1. Class: SessionState

Description	A *SessionState* describes a session of interaction between the MRMS and a user. A *User* is logged in in a *SessionState* if it references that *User*. If logged in it has a remote reference to an instance of *MrmsFacade* on the server which can be used by the control to communicate with the server.
Attributes	---
Operations	

- loggedIn(user: User): Boolean

Effect	Checks whether the given *User* is logged in in this *SessionState*.
Parameters	---
Return	The boolean value *true*, if the given *User* is logged in in this *SessionState*.
Exceptions	---
Actor	*MainController*

3.2. Class: MainController

Description	The *MainController* manages concrete *AbstractControls*. It provides a *ViewContainer* were *AbstractViews* of *AbstractControls* may be plugged in. It is associated with a *SessionState* that

provides a reference to the suitable MRMS server facade. Managed *AbstractControl*s may interact with the *MainController* by using Events. For this the *MainController* provides delegate operations that may be registered at the *AbstractControl*s. Moreover it contains static helper operations for showing dialogs to the user (used by *AbstractControls*). The *MainController* implements the *Mediator* pattern as described by the GoF. See also Section 3.7, “Sequence diagrams for package model.client” [].

Attributes	---

Operations

- static showError(text: String): void

Effect	An error pop up is shown to the user.
Parameters	*text*: Error message
Return	---
Exceptions	---
Actor	*AbstractControls*

- static requestConfirmation(text: String): int

Effect	A confirmation dialog is shown to the user.
Parameters	*text*: Confirmation message
Return	An int value that is representing the decision of the user
Exceptions	---
Actor	*AbstractControls*

- start(): void

Effect	Activates default controls, *MenuBar*- and *ToolBarControl*, shows application window and starts event handling.
Parameters	---
Return	---
Exceptions	---
Actor	User

- stop(): void

Effect	Disposes the application and all its controllers.
Parameters	---
Return	---
Exceptions	---

Actor	*MainController*

- handleUserLogsInEvent(sender: AbstractControl, args: System.Args): void

Effect	A *UserLogsInControl* is created and invoked. Its view is plugged into the *ViewContainer*.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*AbstractControls* using an *EventHandle*

- showNavigationPaneEvent(sender: AbstractControl, args: System.Args): void

Effect	An *NavigationControl* for the selected *Resource* is created and invoked. Its view is plugged into the *ViewContainer*.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*AbstractControls* using an *EventHandle*

- handleEditResourceEvent(sender: AbstractControl, args: System.Args): void

Effect	An *EditResourceControl* for the selected *Resource* is created and invoked (uses *static createEditResourceControl()*) .
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*AbstractControls* using an *EventHandle*

- handleCreateFilterEvent(sender: AbstractControl, args: System.Args): void

Effect	A *CreateFilterControl* is created and invoked. Its view is plugged into the *ViewContainer*.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation

Return	---
Exceptions	---
Actor	*AbstractControls* using an *EventHandle*

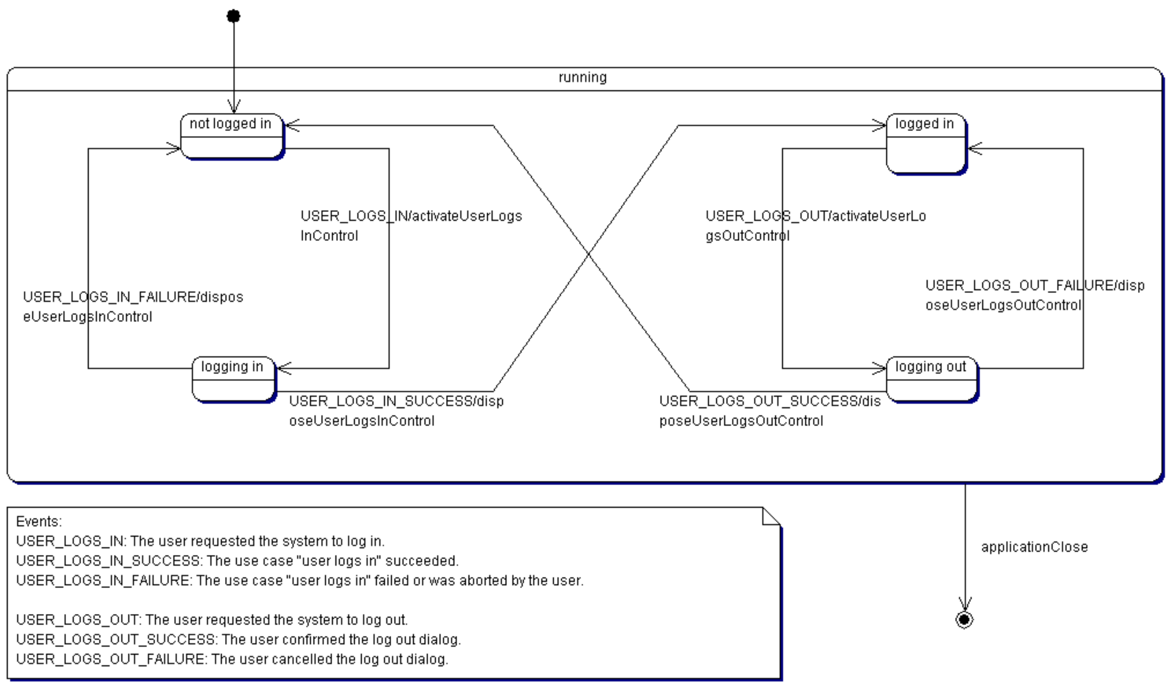
- handleUseCaseDoneEvent(sender: AbstractControl, args: System.Args): void

Effect	The sender (concrete *AbstractControl*) is disposed and its view is removed from the *ViewContainer*.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*AbstractControls* using an *EventHandle*

- handleSelectionChangedEvent(sender: AbstractControl, args: System.Args): void

Effect	All controls are informed about the new selection by an event.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*AbstractControls* using an *EventHandle*

Figure 7. State chart: MainController


running
not logged in
logged in
logging in
logging out
USER_LOGS_IN/activateUserLogsInControl
USER_LOGS_IN_FAILURE/disposeUserLogsInControl
USER_LOGS_IN_SUCCESS/disposeUserLogsInControl
USER_LOGS_OUT/activateUserLogsOutControl
USER_LOGS_OUT_FAILURE/disposeUserLogsOutControl
USER_LOGS_OUT_SUCCESS/disposeUserLogsOutControl
applicationClose

Events:
USER_LOGS_IN: The user requested the system to log in.
USER_LOGS_IN_SUCCESS: The use case "user logs in" succeeded.
USER_LOGS_IN_FAILURE: The use case "user logs in" failed or was aborted by the user.

USER_LOGS_OUT: The user requested the system to log out.
USER_LOGS_OUT_SUCCESS: The user confirmed the log out dialog.
USER_LOGS_OUT_FAILURE: The user cancelled the log out dialog.

3.3. Class: AbstractControl

DescriptionUser logs in	Base class for all concrete use case controllers. Encapsulates the common control flow. See also Section 3.7, “Sequence diagrams for package model.client” [].
Attributes	---
Operations	

- invoke(): void

Effect	Constructor operation that activates this AbstractControl instance.
Parameters	---
Exceptions	---
Actor	MainController

- createView(): AbstractView

Effect	The AbstractControl is told to create its view component.
Parameters	---
Return	The created view component
Exceptions	---
Actor	*MainController*

- dispose(): void

Effect	Destroys the AbstractControl and its view component.
Parameters	---
Return	---
Exceptions	---
Actor	*MainController*

3.4. Class: ViewContainer

Description	A *ViewContainer* is a component were the *MainController* may plug in *AbstractViews* of *AbstractControls*. See also Section 3.7, “Sequence diagrams for package model.client” [].
Attributes	---
Operations	

- addView(view: AbstractView,location: int): void

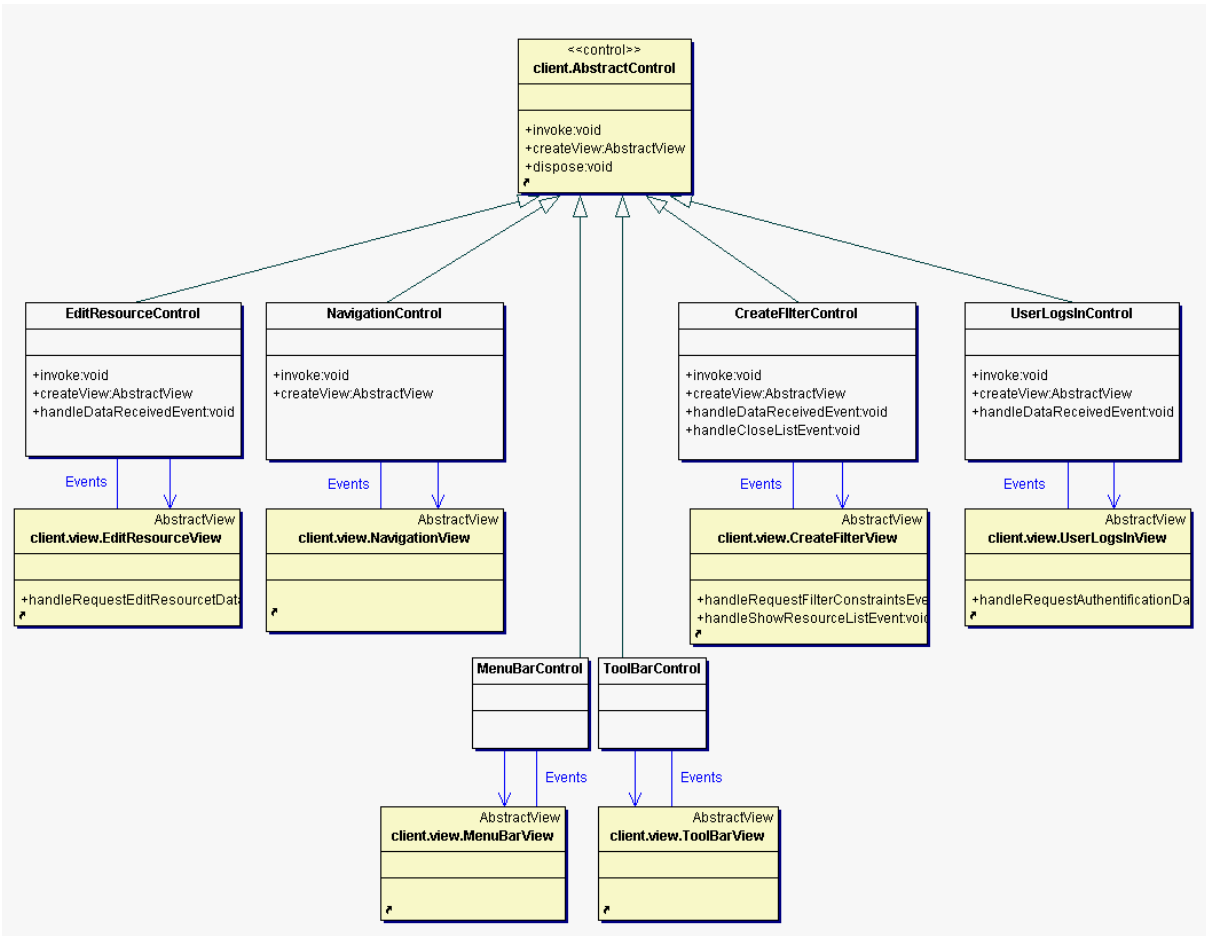
Effect	Adds the given *AbstractView*.
Parameters	*view*: *AbstractView* to be added
	location: An identifier determining the location to place the *AbstractView*
Return	---
Exceptions	---
Actor	*MainController*

- removeView(view: AbstractView): void

Effect	Removes the given *AbstractView*.
Parameters	*view*: *AbstractView* to be removed
Return	---
Exceptions	---
Actor	*MainController*

3.5. Package: client.control

Figure 8. Package: client.control



3.5.1. Class: UserLogsInControl

Description	A concrete *AbstractControl* for logging a user in. See also Section 3.7, “Sequence diagrams for package model.client” [].
Attributes	---
Operations	

- invoke(sender: AbstractControl, args: System.Args): void

Effect	The user is requested to enter his/her *AuthentificationData*.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*MainController*

- createView(): AbstractView

Effect	An *UserLogsInView* is created and returned.

Parameters	---
Return	The created *UserLogsInView*
Exceptions	---
Actor	*MainController*

- handleDataReceivedEvent(sender: AbstractControl, args: System.Args): void

Effect	Either the user gets logged in or an error dialog is shown to him.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*UserLogsInView*

3.5.2. Class: NavigationControl

Description	A concrete *AbstractControl* for navigating entities managed by the MRMS system. See also Section 3.7, “Sequence diagrams for package model.client” [].
Attributes	---
Operations	

- invoke(sender: AbstractControl, args: System.Args): void

Effect	The *NavigationControl* waits for user's interaction.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*MainController*

- createView(): AbstractView

Effect	An *NavigationView* is created and returned.
Parameters	---
Return	The created *NavigationView*
Exceptions	---

Actor	*MainController*

3.5.3. Class: EditResourceControl

Description: A concrete *AbstractControl* for editing *Resources* managed by the MRMS system. See also Section 3.7, “Sequence diagrams for package model.client” [].

Attributes: ---

Operations

- invoke(sender: AbstractControl, args: System.Args): void

Effect	Locks the *Resource* that shall be edited using the *MrmsFacade*. The user is requested to enter the changes to be performed.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*MainController*

- createView(): AbstractView

Effect	An *EditResourceView* is created and returned.
Parameters	---
Return	The created *EditResourceView*
Exceptions	---
Actor	*MainController*

- handleDataReceivedEvent(sender: AbstractControl, args: System.Args): void

Effect	Updates the edited *Resource* using the *MrmsFacade* or shows an error dialog.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*EditResourceView*

3.5.4. Class: CreateFilterControl

Description	A concrete *AbstractControl* editing *Resources* managed by the MRMS system. See also Section 3.7, “Sequence diagrams for package model.client” [].
Attributes	---
Operations	

- invoke(sender: AbstractControl, args: System.Args): void

Effect	The user is requested to enter the filter constraints.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*MainController*

- createView(): AbstractView

Effect	An *CreateFilterView* is created and returned.
Parameters	---
Return	The created *CreateFilterView*
Exceptions	---
Actor	*MainController*

- handleDataReceivedEvent(sender: AbstractControl, args: System.Args): void

Effect	Gets a list of matching resources and presents it to the user.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*CreateFilterView*

- handleCloseListEvent(sender: AbstractControl, args: System.Args): void

Effect	Terminates the *CreateFilterControl*.
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---

Exceptions	---
Actor	*CreateFilterView*

3.5.5. Class: MenuBarControl

Description	A concrete *AbstractControl* that is managing an client's application window menu bar.
Attributes	---
Operations	

- invoke(sender: AbstractControl, args: System.Args): void

Effect	---
Parameters	*sender*: The Sender of the event that invoked this operation
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*MainController*

- createView(): AbstractView

Effect	An *MenuBarView* is created and returned.
Parameters	---
Return	The created *MenuBarView*
Exceptions	---
Actor	*MainController*

3.5.6. Class: ToolBarControl

Description	A concrete *AbstractControl* that is managing an client's application window tool bar.
Attributes	---
Operations	

- invoke(sender: AbstractControl, args: System.Args): void

Effect	---
Parameters	*sender*: The Sender of the event that invoked this operation

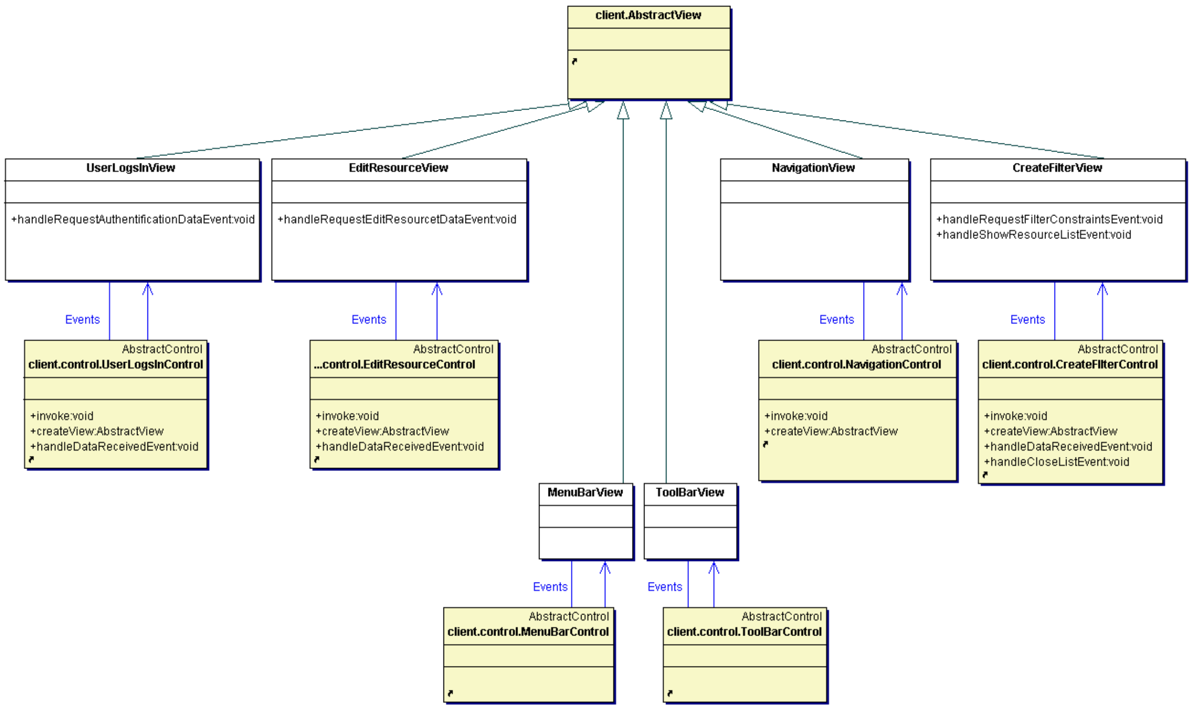
	args: Arguments of the event that invoked this operation
Return	---
Exceptions	---
Actor	*MainController*

• createView(): AbstractView

Effect	An *ToolBarView* is created and returned.
Parameters	---
Return	The created *ToolBarView*
Exceptions	---
Actor	*MainController*

3.6. Package: client.view

Figure 9. Package: model.view


client.AbstractView
UserLogsInView
+handleRequestAuthentificationDataEvent:void
EditResourceView
+handleRequestEditResourcetDataEvent:void
NavigationView
CreateFilterView
+handleRequestFilterConstraintsEvent:void
+handleShowResourceListEvent:void
Events
AbstractControl
client.control.UserLogsInControl
+invoke:void
+createView:AbstractView
+handleDataReceivedEvent:void
...control.EditResourceControl
client.control.NavigationControl
client.control.CreateFilterControl
+handleCloseListEvent:void
MenuBarView
ToolBarView
client.control.MenuBarControl
client.control.ToolBarControl

3.6.1. Class: UserLogsInView

Description	A concrete *AbstractView* representing a log-in-dialog.
Attributes	---
Operations	handleRequestAuthentificationDataEvent(sender: AbstractControl, args: System.Args): void

3.6.2. Class: NavigationView

Description	A concrete *AbstractView* for navigating entities managed by the MRMS system.
Attributes	---
Operations	---

3.6.3. Class: EditResourceView

Description	A concrete *AbstractView* for editing *Resources* managed by the MRMS system.
Attributes	---
Operations	handleRequestEditResourcetDataEvent(sender: AbstractControl, args: System.Args): void

3.6.4. Class: CreateFilterView

Description	A concrete *AbstractView* for creating a *Filter*.
Attributes	---
Operations	handleRequestFilterConstraintsEvent(sender: AbstractControl, args: System.Args): void
	handleShowResourceListEvent(sender: AbstractControl, args: System.Args): void

3.6.5. Class: MenuBarView

Description	A concrete *AbstractView* for that shows a menu bar.
Attributes	---
Operations	---

3.6.6. Class: ToolBarView

Description	A concrete *AbstractView* that shows a tool bar.
Attributes	---
Operations	---

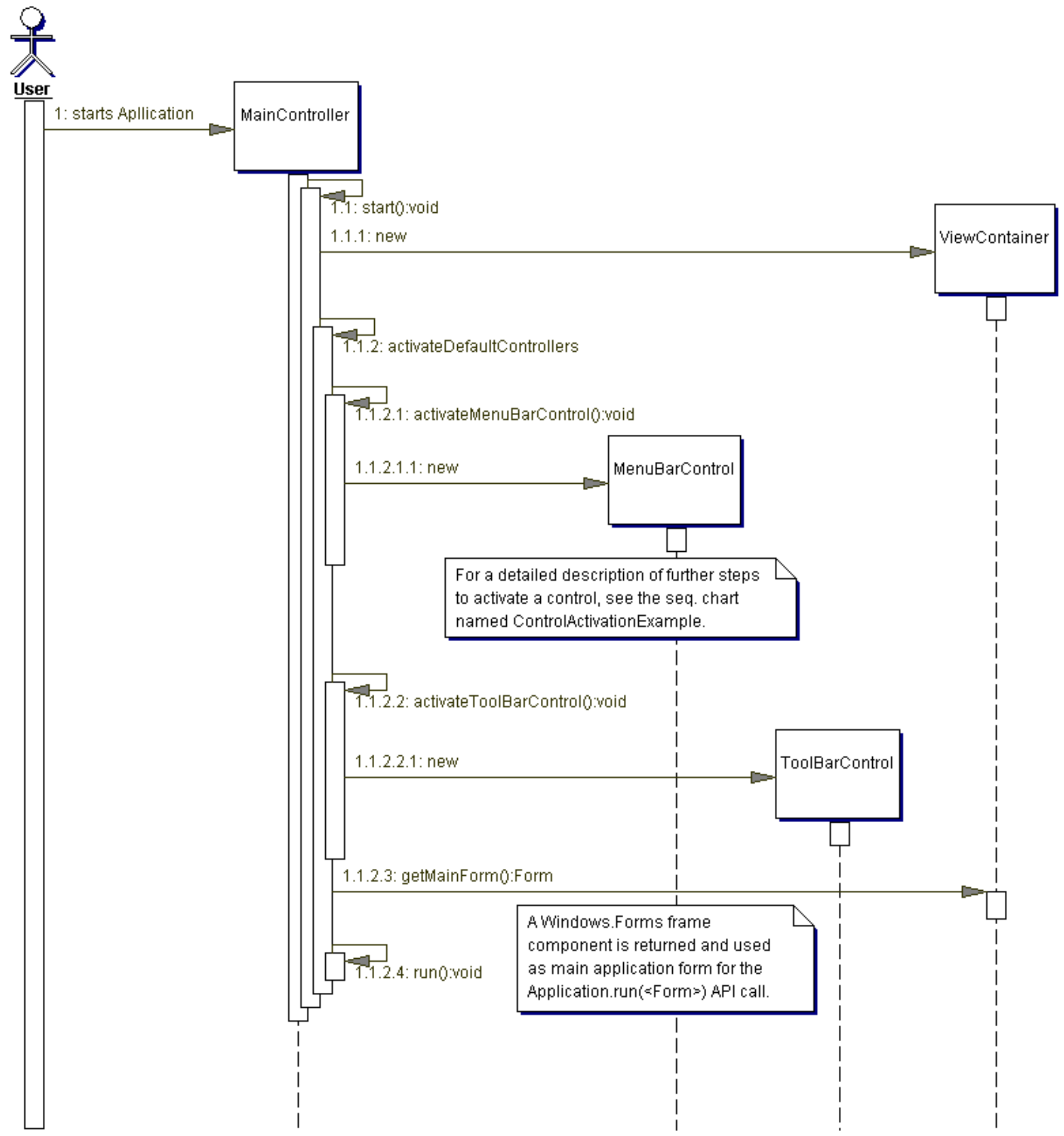
3.7. Sequence diagrams for package model.client

These diagrams verify the model.client package.

3.7.1. Sequence diagram: Application start

The following diagram illustrates the starting process of the application.

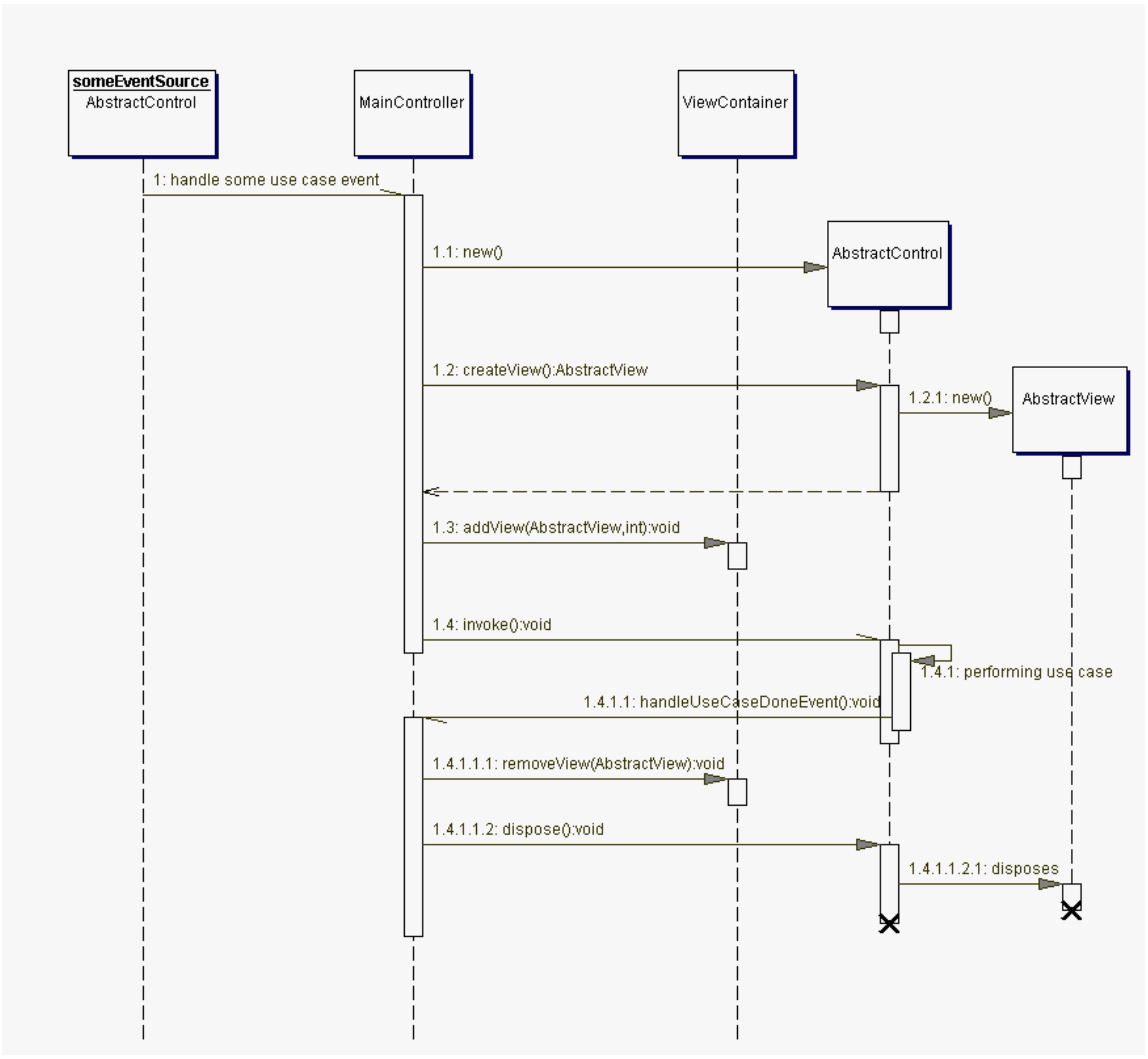
Figure 10. Application start


User
1: starts Apllication
MainController
1.1: start():void
1.1.1: new
ViewContainer
1.1.2: activateDefaultControllers
1.1.2.1: activateMenuBarControl():void
1.1.2.1.1: new
MenuBarControl
For a detailed description of further steps to activate a control, see the seq. chart named ControlActivationExample.
1.1.2.2: activateToolBarControl():void
1.1.2.2.1: new
ToolBarControl
1.1.2.3: getMainForm():Form
A Windows.Forms frame component is returned and used as main application form for the Application.run(<Form>) API call.
1.1.2.4: run():void

3.7.2. Sequence diagram: AbstractControl's life cycle

The following diagram illustrates the life cycle of an AbstractControl and its view.

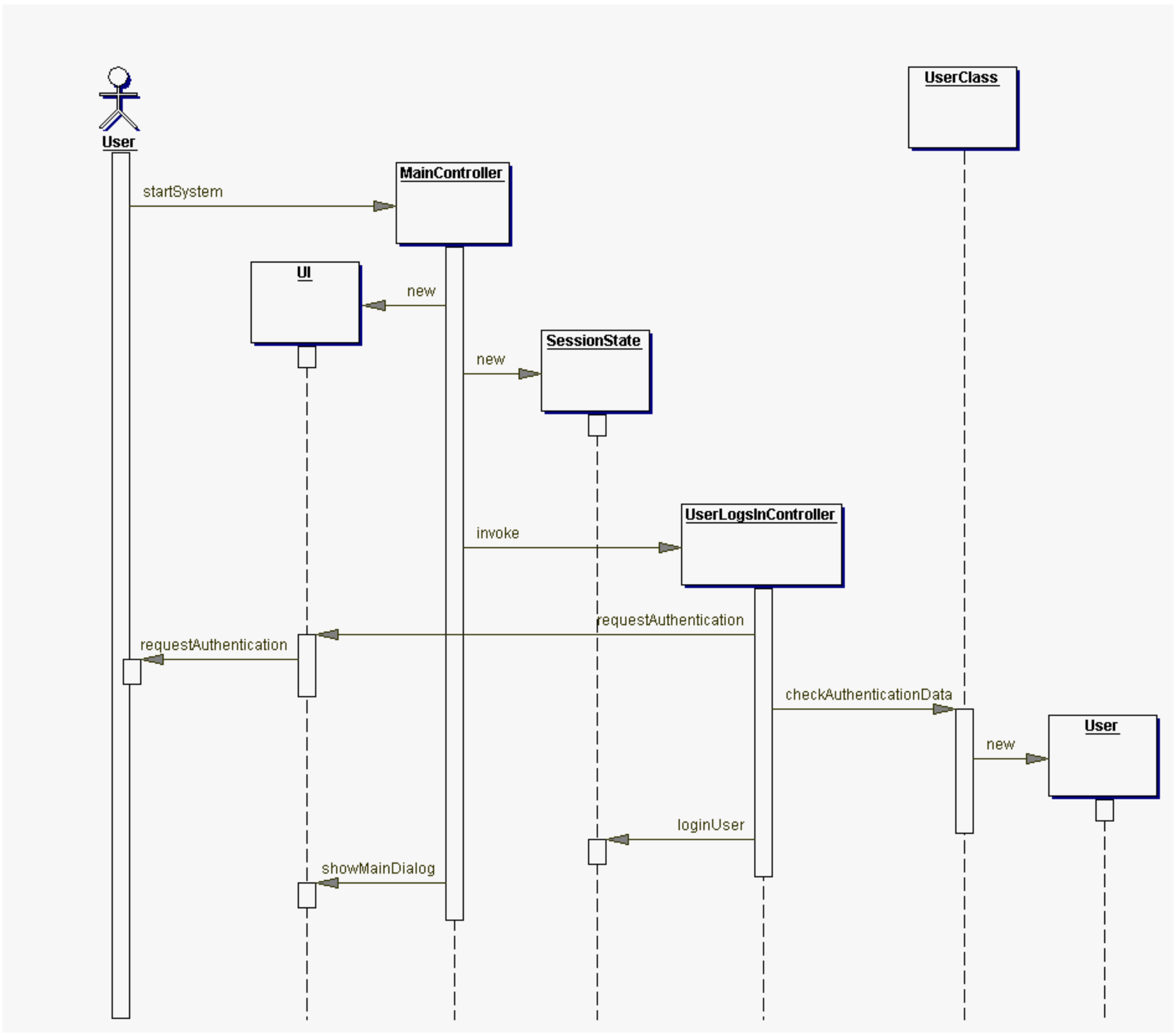
Figure 11. AbstractControl's life cycle


someEventSource
AbstractControl
MainController
ViewContainer
1: handle some use case event
1.1: new()
AbstractControl
1.2: createView():AbstractView
1.2.1: new()
AbstractView
1.3: addView(AbstractView,int):void
1.4: invoke():void
1.4.1: performing use case
1.4.1.1: handleUseCaseDoneEvent():void
1.4.1.1.1: removeView(AbstractView):void
1.4.1.1.2: dispose():void
1.4.1.1.2.1: disposes

3.7.3. Sequence diagram: User logs in

The following diagram illustrates the object interaction while performing the use case "User logs in"

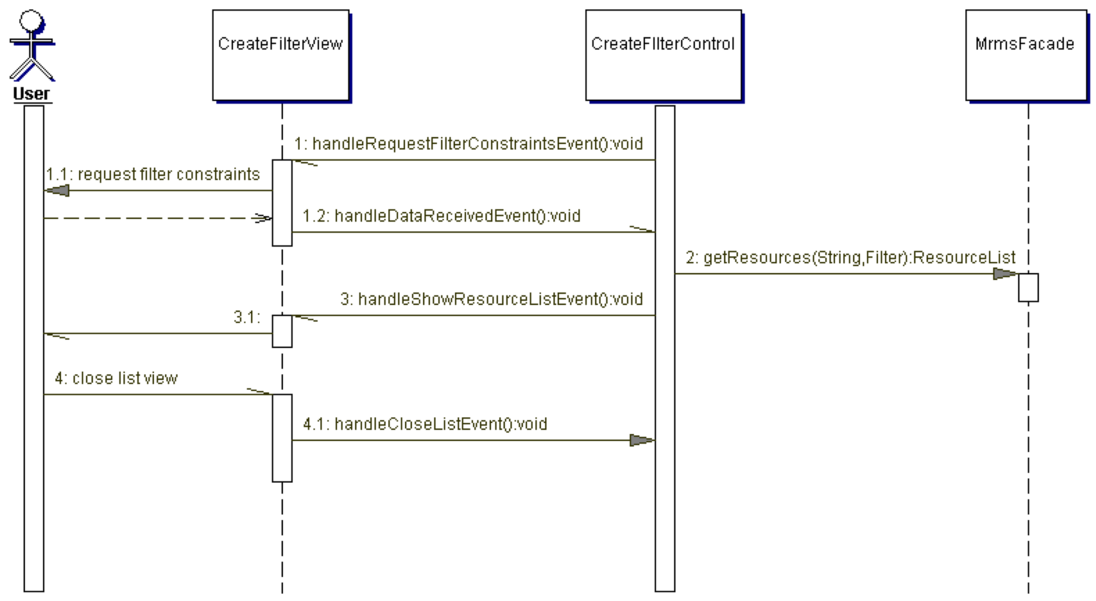
Figure 12. User logs in



3.7.4. Sequence diagram: Create filtered collection of resource entries

The following diagram illustrates the object interaction while performing the use case "Create filtered collection of resource entries"

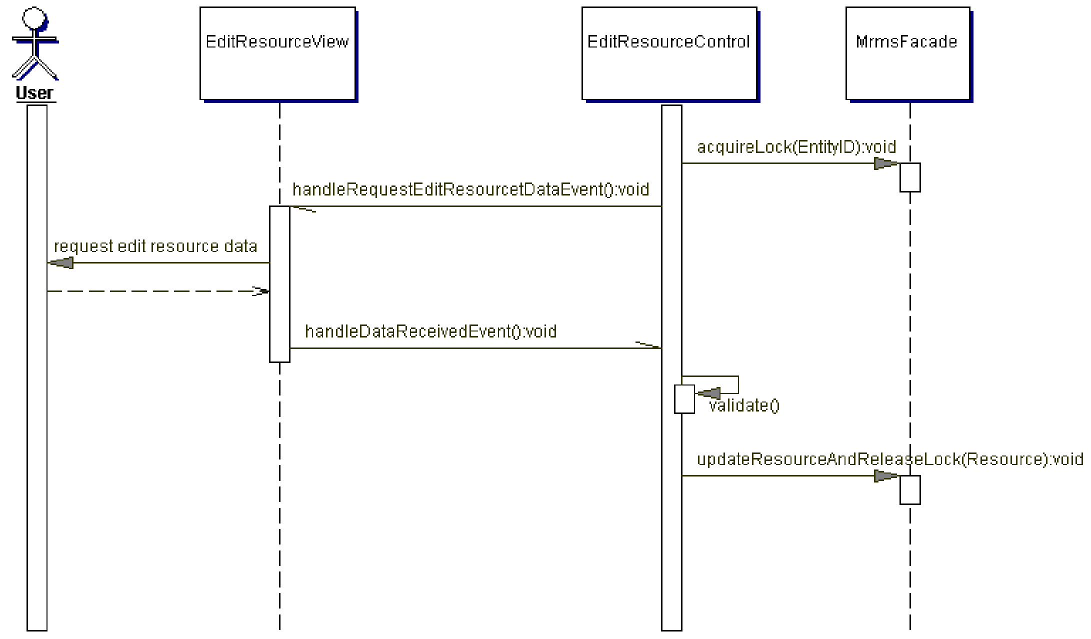
Figure 13. Create filtered collection of resource entries


User
CreateFilterView
CreateFIlterControl
MrmsFacade
1: handleRequestFilterConstraintsEvent():void
1.1: request filter constraints
1.2: handleDataReceivedEvent():void
2: getResources(String,Filter):ResourceList
3: handleShowResourceListEvent():void
3.1:
4: close list view
4.1: handleCloseListEvent():void

3.7.5. Sequence diagram: Edit resources

The following diagram illustrates the object interaction while performing the use case "Edit Resources"

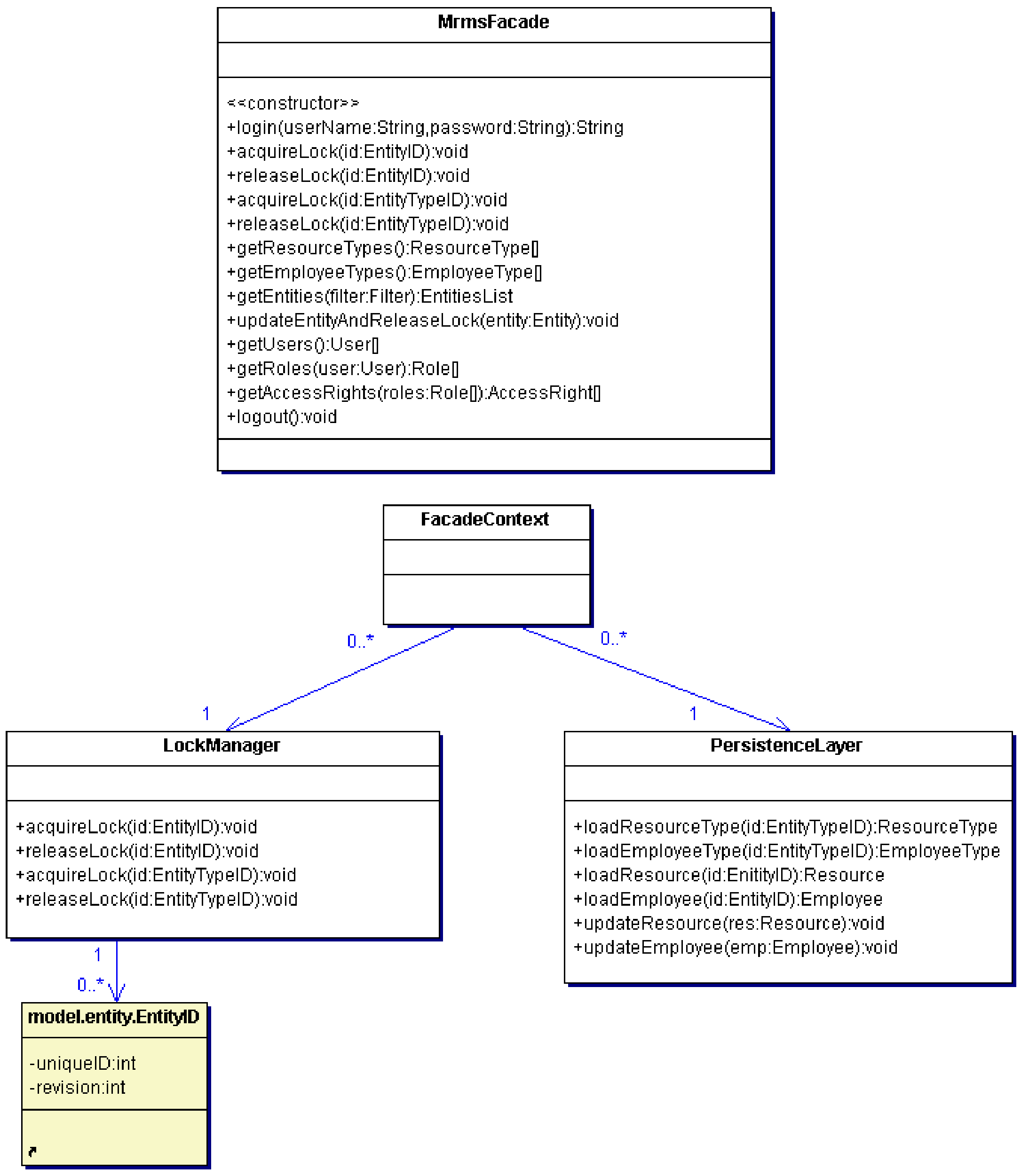
Figure 14. Edit resources


User
EditResourceView
EditResourceControl
MrmsFacade
acquireLock(EntityID):void
handleRequestEditResourcetDataEvent():void
request edit resource data
handleDataReceivedEvent():void
validate()
updateResourceAndReleaseLock(Resource):void

4. Package: server

The client's interface to the server application is the *MrmsFacade*. Every instance of an *MrmsFacade* has a *FacadeContext* that provides access to a central *LockManager* as well as to the *PersistenceLayer*.

Figure 15. The MRMS Server Core Classes


MrmsFacade
<<constructor>>
+login(userName:String,password:String):String
+acquireLock(id:EntityID):void
+releaseLock(id:EntityID):void
+acquireLock(id:EntityTypeID):void
+releaseLock(id:EntityTypeID):void
+getResourceTypes():ResourceType[]
+getEmployeeTypes():EmployeeType[]
+getEntities(filter:Filter):EntitiesList
+updateEntityAndReleaseLock(entity:Entity):void
+getUsers():User[]
+getRoles(user:User):Role[]
+getAccessRights(roles:Role[]):AccessRight[]
+logout():void
FacadeContext
0..*
0..*
1
1
LockManager
+acquireLock(id:EntityID):void
+releaseLock(id:EntityID):void
+acquireLock(id:EntityTypeID):void
+releaseLock(id:EntityTypeID):void
PersistenceLayer
+loadResourceType(id:EntityTypeID):ResourceType
+loadEmployeeType(id:EntityTypeID):EmployeeType
+loadResource(id:EnitityID):Resource
+loadEmployee(id:EntityID):Employee
+updateResource(res:Resource):void
+updateEmployee(emp:Employee):void
1
0..*
model.entity.EntityID
-uniqueID:int
-revision:int

4.1. Class: MrmsFacade

Description — The *MrmsFacade* provides the server's functionality to connected clients. Every instance references one *FacadeContext* which provides access to a central *LockManager* as well as to the *PersistenceLayer*.

Attributes — ---

Operations

- login(user: User): MrmsFacade

Effect	Constructor operation. Verifies and authorizes the given *User* and initializes a new MrmsFacade on success. If the *User* could not be authorized a *Login-FailedException* is thrown.
Parameters	*user* (User): the *User* object identifying and authorizing the user to log in.
Return	An instance of *MrmsFacade*
Exceptions	*LoginFailedException*
Actor	*UserLogsInControl*

- acquireLock(id:EntityID): void

Effect	Requests to acquire a lock for the given *EntityID* at the central *LockManager* instance.
Parameters	*id* (EntityID): the *EntityID* of the *Entity* that should be locked.
Return	---
Exceptions	*LockNotAvailableException* - if the `id` is already locked
	RevisionChangedException - if the `id`'s revision is not current
Actor	*EditResourceControl*

- releaseLock(id: EntityID): void

Effect	Requests to release a lock for the given *EntityID* at the central *LockManager* instance.
Parameters	id (EntityID): the *EntityID* of the *Entity* to unlock
Return	---
Exceptions	---
Actor	*EditResourceControl*

- acquireLock(id:EntityTypeID): void

Effect	Requests to acquire a lock for the given *EntityTypeID* at the central *Lock-Manager* instance.
Parameters	*id* (EntityTypeID): the *EntityTypeID* of the *EntityType* that should be locked.
Return	---
Exceptions	*LockNotAvailableException* - if the `id` is already locked

	RevisionChangedException - if the `id`'s revision is not current
Actor	*EditResourceControl*

- releaseLock(id: EntityTypeID): void

Effect	Requests to release a lock for the given *EntityTypeID* at the central *LockManager* instance.
Parameters	id (EntityTypeID): the *EntityTypeID* of the *EntityType* to unlock
Return	---
Exceptions	---
Actor	*EditResourceControl*

- getResourceTypes(): ResourceType[]

Effect	Getter without side effects.
Parameters	---
Return	An array with all *ResourceType*s that the administrator has configured in the system.
Exceptions	---
Actor	*NavigationControl*

- getEmployeeTypes(): EmployeeType[]

Effect	Getter without side effects.
Parameters	---
Return	An array with all *EmployeeType*s that the administrator has configured in the system.
Exceptions	---
Actor	*NavigationControl*

- getEntities(filter: Filter): EntitiesList

Effect	Requests a list of *Entity*s matching the given *Filter*; has no side effects.

Parameters	filter (Filter): the *Filter* that all returned entities must match.
Return	an *EntitiesList* containing all matching *Entities*
Exceptions	---
Actor	All controls that need to access entities.

- updateEntityAndReleaseLock(entity: Entity): void

Effect	Updates the given *Entity* object and releases the associated lock.
Parameters	entity (*Entity*): the *Entity* to update
Return	---
Exceptions	---
Actor	*EditEntityControl*

- getUsers(): User[]

Effect	Getter without side effects.
Parameters	---
Return	An array with all *User*s that the administrator has configured in the system.
Exceptions	---
Actor	*NavigationControl*

- getRoles(user: User): Role[]

Effect	Getter without side effects.
Parameters	user (*User*): the user whose roles should be returned
Return	An array with all *Roles*s that the administrator has configured in the system for a specifc user.
Exceptions	---
Actor	*NavigationControl* and *EditUserControl*

- getAccessRights(roles: Role[]): AccessRight[]

Effect	Getter without side effects.

Parameters	roles (*Role[]*): the roles whose access rights should be returned
Return	An array with merged *AccessRight*s that all given *Role*s have.
Exceptions	---
Actor	*NavigationControl* and *EditUserControl*

- logout(): void

Effect	Informs the *MrmsFacade* that the client does not need its services anymore; any open locks will be released. This method is automatically called if the client did not do any request for a specific amount of time (15 Min).
Parameters	---
Return	---
Exceptions	---
Actor	*MainController*

4.2. Class: FacadeContext

Description	The *FacadeContext* provides a context for a *MrmsFacade* which consists of references to the central instances of *LockManager* and *PersistenceLayer*.
Attributes	---
Operations	---

4.3. Class: LockManager

Description	The *LockManager* holds information about locked *Entity*s and *EntityType*s. Client classes may acquire and release locks with instances of this class.
Attributes	---
Operations	

- acquireLock(id:EntityID): void

Effect	Requests to acquire a lock for the given *EntityID* at the central *LockManager* instance.
Parameters	*id* (EntityID): the *EntityID* of the *Entity* that should be locked.
Return	---

Exceptions	*LockNotAvailableException* - if the `id` is already locked
	RevisionChangedException - if the `id`'s revision is not current
Actor	*EditResourceControl*

- releaseLock(id: EntityID): void

Effect	Requests to release a lock for the given *EntityID* at the central *LockManager* instance.
Parameters	id (EntityID): the *EntityID* of the *Entity* to unlock
Return	---
Exceptions	---
Actor	*EditResourceControl*

- acquireLock(id:EntityTypeID): void

Effect	Requests to acquire a lock for the given *EntityTypeID* at the central *LockManager* instance. A lock on an *EntityType* also locks all *Entity*s of this type and no *Entity*s of this type may be created.
Parameters	*id* (EntityTypeID): the *EntityTypeID* of the *EntityType* that should be locked.
Return	---
Exceptions	*LockNotAvailableException* - if the `id` is already locked
	RevisionChangedException - if the `id`'s revision is not current
Actor	*EditResourceControl*

- releaseLock(id: EntityTypeID): void

Effect	Requests to release a lock for the given *EntityTypeID* at the central *LockManager* instance.
Parameters	id (EntityTypeID): the *EntityTypeID* of the *EntityType* to unlock
Return	---
Exceptions	---
Actor	*EditResourceControl*

4.3.1. Sequence diagrams: Locking

Figure 16. Acquire lock without problems

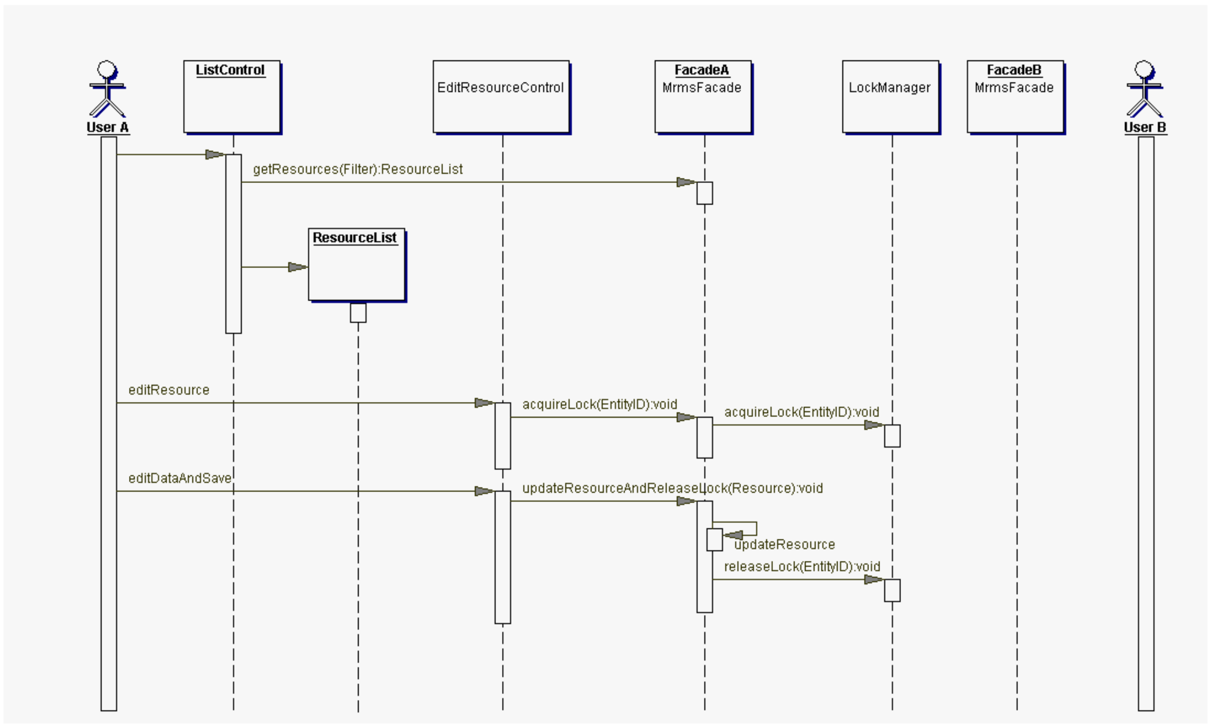

User A
ListControl
EditResourceControl
FacadeA
MrmsFacade
LockManager
FacadeB
MrmsFacade
User B
getResources(Filter):ResourceList
ResourceList
editResource
acquireLock(EntityID):void
acquireLock(EntityID):void
editDataAndSave
updateResourceAndReleaseLock(Resource):void
updateResource
releaseLock(EntityID):void

Figure 17. Acquire lock of already locked entity

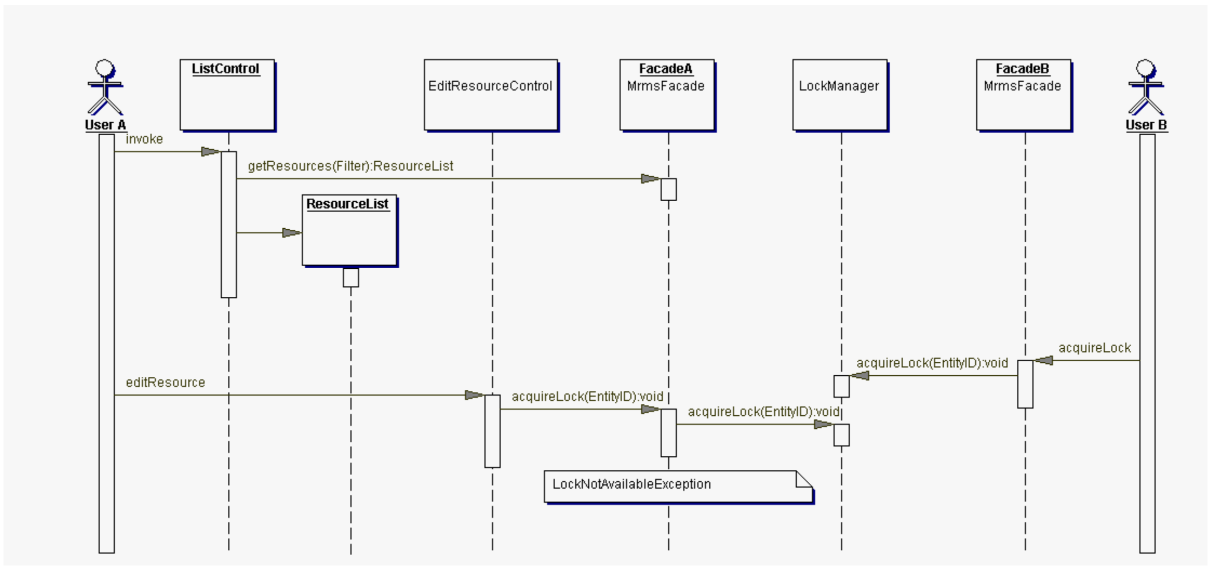
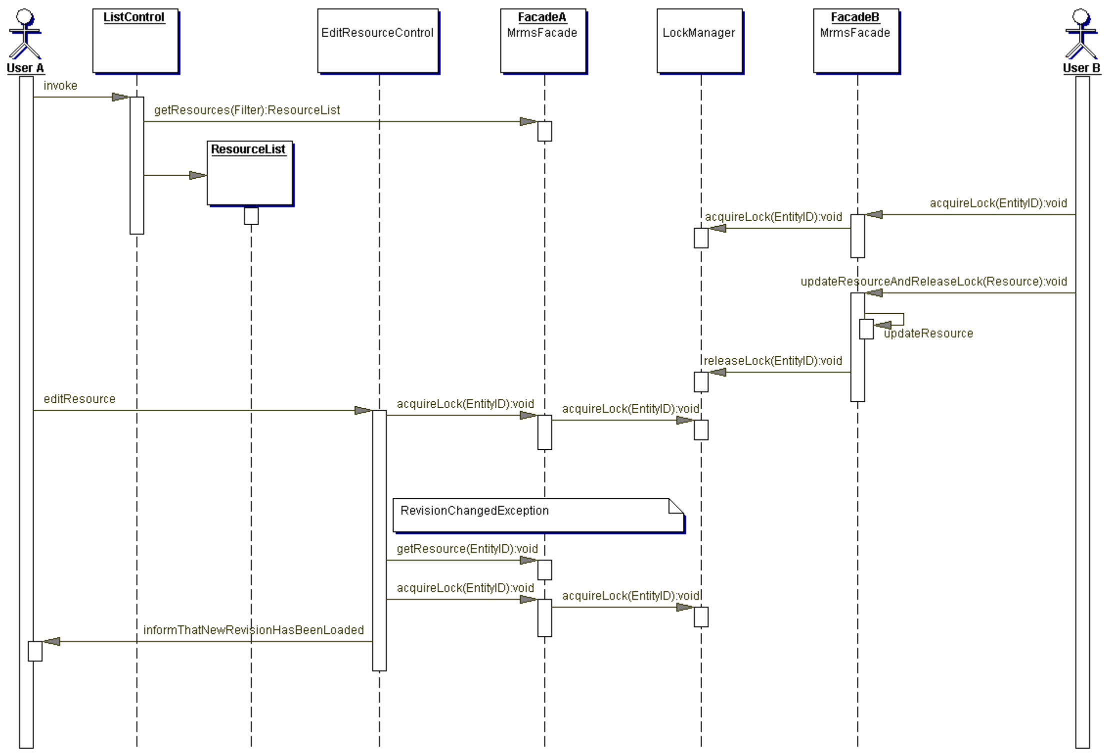

User A
ListControl
EditResourceControl
FacadeA
MrmsFacade
LockManager
FacadeB
MrmsFacade
User B
invoke
getResources(Filter):ResourceList
ResourceList
acquireLock
acquireLock(EntityID):void
editResource
acquireLock(EntityID):void
acquireLock(EntityID):void
LockNotAvailableException

Figure 18. Acquire lock of a outdated entity


User A
ListControl
EditResourceControl
FacadeA
MrmsFacade
LockManager
FacadeB
MrmsFacade
User B
invoke
getResources(Filter):ResourceList
ResourceList
acquireLock(EntityID):void
acquireLock(EntityID):void
updateResourceAndReleaseLock(Resource):void
updateResource
releaseLock(EntityID):void
editResource
acquireLock(EntityID):void
acquireLock(EntityID):void
RevisionChangedException
getResource(EntityID):void
acquireLock(EntityID):void
acquireLock(EntityID):void
informThatNewRevisionHasBeenLoaded

4.4. Class: PersistenceLayer

Description	Implements a persistence layer for objects of MRMS model classes. It provides atomic load, update and delete methods for all important model classes as well as query functionality.
Attributes	---
Operations	---

5. Appendix: .NET Event Handling

The MRMS is implemented for the .NET platform and therefore partly builds up on the .NET model for event handling. The following two figures are an overview on how that mechanism works.

Figure 19. Classes within the .NET event model

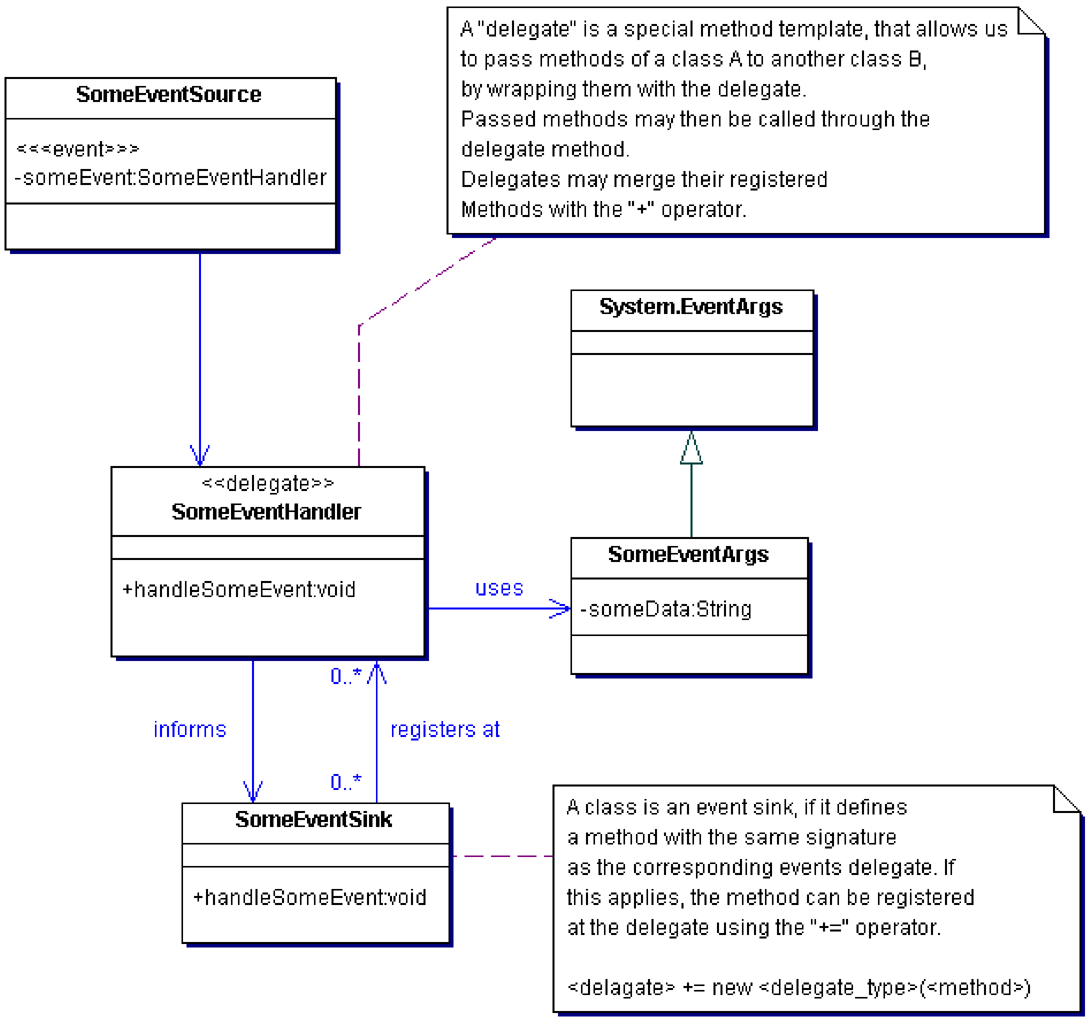
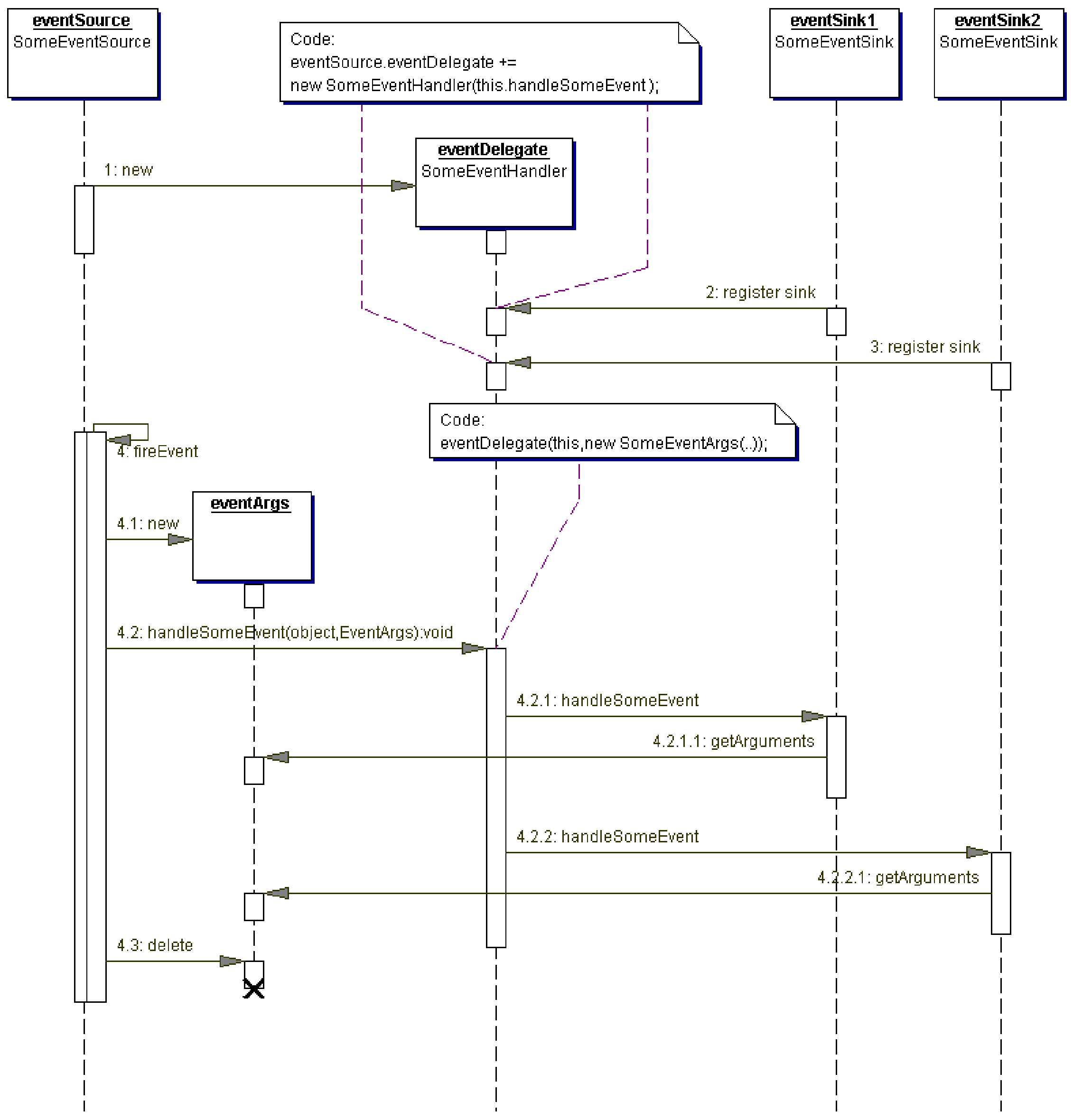

SomeEventSource
<<<event>>>
-someEvent:SomeEventHandler
A "delegate" is a special method template, that allows us to pass methods of a class A to another class B, by wrapping them with the delegate. Passed methods may then be called through the delegate method. Delegates may merge their registered Methods with the "+" operator.
System.EventArgs
<<delegate>>
SomeEventHandler
+handleSomeEvent:void
uses
SomeEventArgs
-someData:String
0..*
informs
registers at
0..*
SomeEventSink
+handleSomeEvent:void
A class is an event sink, if it defines a method with the same signature as the corresponding events delegate. If this applies, the method can be registered at the delegate using the "+=" operator.
<delagate> += new <delegate_type>(<method>)

Figure 20. Sequence: Sample setup and action



6. Appendix: Use Cases

These are the use cases derived from the additional functionality “Resource Reservation”.

6.1. Create resource reservation

Goal	A new reservation for a resource is created.
Category	Primary
External Actors	User
Precondition	A user is logged in who has proper access rights to create the new resource reservation.
Triggering Event	The user requests the system to create a new resource reservation.

Postcondition Success	A new resource reservation has been created according to the users input.
Postcondition Failure	No new resource reservation has been created.
Description	1. The system requests the user to choose the resource, start time, end time of the reservation and the customer. 2. The user determines resource reservation and submits his input. 3. The system creates a new resource reservation.
Extensions	---
Alternatives	---
Additional Requirements	---
Annotation	---

6.2. Delete resource reservation

Goal	The reservation for a resource is deleted.
Category	Primary
External Actors	User
Precondition	A user is logged in who has proper access rights to delete the resource reservation.
Triggering Event	The user requests the system to delete a resource reservation.
Postcondition Success	The resource reservation has been deleted.
Postcondition Failure	The resource reservation has not been deleted.
Description	1. The system requests the user to choose a resource reservation. 2. The user determines the resource reservation to delete and submits his input. 3. The system deletes the resource reservation.
Extensions	---
Alternatives	---
Additional Requirements	---
Annotation	---

6.3. Change resource reservation

Goal	The reservation for a resource is changed.

Category	Secondary
External Actors	User
Precondition	A user is logged in.
Triggering Event	The user requests the system to create a filtered collection of resource reservations.
Postcondition Success	The user is shown a collection of resource reservations that passed the filter he created.
Postcondition Failure	---
Description	
	1. The system requests the user to choose a resource reservation for changing.
	2. The user determines the resource reservation to change and submits his input.
	3. The system changes the resource reservation.
Extensions	---
Alternatives	---
Additional Requirements	---
Annotation	---

6.4. Create filtered collection of resource reservation entries

Goal	Collect a set of resources reservations meeting a specific criterion and offer it to the user for further processing.
Category	Secondary
External Actors	User
Precondition	A user is logged in who has proper access rights to change the resource reservation.
Triggering Event	The user requests the system to change a resource reservation.
Postcondition Success	The resource reservation has been changed.
Postcondition Failure	The resource reservation has not been changed.
Description	
	1. The system requests the user to configure a filter listed resource reservations will have to pass
	2. The system collects all resource reservations passing the specified filter and offers them to the user for further processing.
Extensions	---

Alternatives	---
Additional Requirements	---
Annotation	---