QLectives – Socially Intelligent Systems for Quality
Project no. 231200

Instrument: Large-scale integrating project (IP)
Programme: FP7-ICT

Deliverable D4.4.1
Generic Metadata Model

Submission date: 2010-02-12

Start date of project: 2009-03-01 **Duration: 48 months**

Organisation name of lead contractor for this deliverable: IRT

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**x**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Document information

1.1 Authors

Author	Organisation	E-mail
J. Groh	IRT	Groh@irt.de
M. Guelbahar	IRT	Guelbahar@irt.de
A. Zistler	IRT	Zistler@irt.de

1.2 Other contributors

Name	Organisation	E-mail

1.3 Document history

Version#	Date	Change
V0.1		Starting version, template
V0.2	08-10-2009	Definition of ToC
V0.3	18-11-2009	First draft version
V0.4	11-12-2009	Draft version on QL Wiki
V0.5	26-01-2010	Version sent to deliverables committee for approval
V1.0	11-02-2010	Approved version to be submitted to EU

1.4 Document data

Keywords	QLectives, Metadata, Quality
Editor address data	Name: Alois Zistler Partner: IRT Address: Floriansmuehlstrasse 60, D-80939 Muenchen, Germany Phone: +49 89 323 99 234 Fax: +49 89 323 99 200 E-mail: Zistler@irt.de
Delivery date	February 2010

1.5 Distribution list

Date	Issue	E-mail
	Consortium members	QLECTIVES@LIST.SURREY.AC.UK
	Project officer Jose Fernandez-Villacanas	Jose.FERNANDEZ-VILLACANAS@ec.europa.eu
	EC archive	INFSO-ICT-231200@ec.europa.eu

QLectives Consortium

This document is part of a research project funded by the ICT Programme of the Commission of the European Communities as grant number ICT-2009-231200.

University of Surrey (Coordinator)
Department of Sociology / Centre for Research in Social Simulation
Guildford GU2 7XH
Surrey
United Kingdom
Contact person: Prof. Nigel Gilbert
E-mail: n.gilbert@surrey.ac.uk

Technical University of Delft
Department of Software Technology
Delft, 2628 CN
Netherlands
Contact Person: Dr Johan Pouwelse
E-mail: j.a.pouwelse@tudelft.nl

ETH Zurich
Chair of Sociology, in particular Modelling and Simulation,
Zurich, CH-8092
Switzerland
Contact person: Prof. Dirk Helbing
E-mail: dhelbing@ethz.ch

University of Szeged
MTA-SZTE Research Group on Artificial Intelligence
Szeged 6720, Hungary
Contact person: Dr Mark Jelasity
E-mail: jelasity@inf.u-szeged.hu

University of Fribourg
Department of Physics
Fribourg 1700
Switzerland
Contact person: Prof. Yi-Cheng Zhang
E-mail: yi-cheng.zhang@unifr.ch

University of Warsaw
Faculty of Psychology
Warsaw 00927, Poland
Contact Person: Prof. Andrzej Nowak
E-mail: nowak@fau.edu

Centre National de la Recherche Scientifique, CNRS
Paris 75006,
France
Contact person: Dr. Camille ROTH
E-mail: camille.roth@polytechnique.edu

Institut für Rundfunktechnik GmbH
Munich 80939
Germany
Contact person: Christoph Dosch
E-mail: dosch@irt.de

QLectives introduction

QLectives is a project bringing together top social modelers, peer-to-peer engineers and physicists to design and deploy next generation self-organising socially intelligent information systems. The project aims to combine three recent trends within information systems:

- **Social networks** - in which people link to others over the Internet to gain value and facilitate collaboration
- **Peer production** - in which people collectively produce informational products and experiences without traditional hierarchies or market incentives
- **Peer-to-Peer systems** - in which software clients running on user machines distribute media and other information without a central server or administrative control

QLectives aims to bring these together to form Quality Collectives, i.e. functional decentralised communities that self-organise and self-maintain for the benefit of the people who comprise them. We aim to generate theory at the social level, design algorithms and deploy prototypes targeted towards two application domains:

- **QMedia** - an interactive peer-to-peer media distribution system (including live streaming), providing fully distributed social filtering and recommendation for quality
- **QScience** - a distributed platform for scientists allowing them to locate or form new communities and quality reviewing mechanisms, which are transparent and promote

The approach of the QLectives project is unique in that it brings together a highly inter-disciplinary team applied to specific real world problems. The project applies a scientific approach to research by formulating theories, applying them to real systems and then performing detailed measurements of system and user behaviour to validate or modify our theories if necessary. The two applications will be based on two existing user communities comprising several thousand people - so-called "Living labs", media sharing community tribler.org; and the scientific collaboration forum EconoPhysics.

Executive Summary

This document describes a generic metadata model being capable of representing terms of quality for any kind of (digital) object. The model presented is a model in terms of a software design or software architecture expressed as a class scheme. Consequently, we model a software structure.

First, chapter 1 gives an introduction about metadata in general. It is outlined that different types of metadata exist, and that their presence is crucial for the search for quality. Further, it concludes that handling, processing and collection of metadata in QLectives may result in a complex scheme.

In section 2.1 we provide the answers to why a software framework is considered to be helpful with respect to the project goal and why the approach leads towards a generic metadata architecture.

Subsequently, section 2.2 presents the metadata framework design. The generic metadata model is the basis for a toolkit that forms the QLectives internal framework extension, which should enable the design of applications on an abstract level. It is represented as an UML (Unified Modelling Language, [3]) class model and therefore is not bound to any programming language.

In another section we describe external framework extensions which encapsulate bindings to external frameworks, illustrated on an example given in the TV-Anytime [1] metadata format.

Finally, several case studies with respect to QMedia are presented in section 2.5 as class- and sequence diagrams. An example scenario related to QScience “reviewer rates reviewer that rated a scientific paper” will demonstrate our framework extensions, specifically regarding the recursive rating mechanisms.

The conclusions and references are presented in chapters 3 and 4 respectively.

Contents

1 Introduction

Quality Collectives or QLectives will be based – as the naming already states - on the concepts of “quality” and “collective”. Due to the fact that the process of evaluating quality is a rather complex and difficult one, and since it intensively depends on the expectations of various individuals, our primary focus will not be on the *definition* of quality – we will just provide *tools to represent* them. The key aspect we want to raise here is the fact that Quality consists of various different notions, most of them being even subjectively biased. Using the example of audio/video content, it might be available in high resolution (HD) at high bitrates, but on the other hand less informative or entertaining. Therefore, the definition of the quality of this content item depends on the individual use case, as well as the context of it. Someone’s expectations may also differ significantly when consuming content from broadcasters or when viewing user-generated content (UGC). Due to the fact that there is a huge amount of UGC growing rapidly on social networks like YouTube [4], it is becoming easier and easier to search for, find and consume interesting clips matching for example a specific type of content genre. But still, the *quality* of a search result depends on two aspects – the presence of adequate and correct metadata, and search algorithms which efficiently and precisely interpret this metadata, thereby taking into account the information provided by the seeker (such as search keywords, genres, user profile information, context, etc.).

The definition of “metadata” is very generic, meaning “data about other data”. In fact, metadata can be manifold. For instance, metadata might be purely technical, e.g. providing information about technical parameters of audiovisual content like resolution, bit rate, encoding et cetera, or providing statistics, for instance how often media content has been viewed completely, or partially. Moreover, metadata may contain information about the date of issue, actors, producers, genres, similarity to other content, etc. Finally, metadata may represent also rating information given by the viewers - and of course ratings about reviewers.

Hence, metadata plays a substantial role when it comes to definition, search and retrieval of Quality for digital items. Generally, the more precise and the more detailed the metadata is, the better approaches for filtering can be applied.

Consequently, “collecting metadata” is a key element within QLectives. This actively involves users in the process of completing metadata - which might also imply providing motivation to the users to participate, by possibly making use of a “crediting system”. In respect to QMedia this could result in an increased bandwidth for downloading content, or access to HDTV offers. A capable reward system in conjunction with a reasonable input mask for the user would be necessary in this case. In QLectives we will need to be able to deal with a set of different types of metadata formats, ranging from high-quality feeds provided by broadcasters (TV-Anytime format, PrestoSpace [5] or others) up to - in some cases rather limited and proprietary - metadata inputs coming from the user generated content world (made available via YouTube or Peer-to-Peer systems like Tribler [6]). Due to this fact, generation, handling and processing of metadata in general will only be manageable with the support of a unified, underlying metadata scheme. Therefore, this scheme will be the first step towards a solution for this complex topic. In order to be flexible, adaptable towards various existing as well as future metadata formats, and enhance-able, the scheme demands a generic approach: A generic metadata model. This generic model will provide a *toolkit*, which enables us to design our applications inside the project. Its implementation is primarily targeted to QMedia, but it will be generic enough to be adapted by QScience as well.

As a next step in this workpackage, we will focus on the design of filtering algorithms, which will be part of *D4.4.2 “Algorithms for detecting and handling HQ-MD (High Quality Meta Data) and Spam metadata”*.

Deliverable D4.4.3 “Implementation / integration guidelines for HQ-MD in QMedia v2” will afterwards deal with a set of implementation guidelines for QMedia and QScience.

Finally, *D4.4.4 Test report on HQ-MD implementation*, due at the end of the project period, will report the test results from the living lab implementation.

2 Generic Metadata Model

2.1 Motivation for a software framework

QLectives will make use of already existing metadata (e.g. provided by broadcasters), and of course will acquire, collect and combine metadata in its different types for further processing and manipulation. As a result, search for high quality should be facilitated. With respect to system design and implementation, we consider a generic metadata scheme helpful for achieving this. A generic metadata scheme should form the basis of developing appropriate algorithms. The software framework is the abstraction of the Generic Metadata Model/Scheme. Hence, it provides the scheme for a scheme.

There are predefined schemes for various purposes. In principle, QLectives could try to simply re-use one that already provides, for example, user-rating elements, and determine quality by processing these ratings. Yet the present approach will introduce an abstraction to keep specific metadata systems and the QLectives mechanisms as loosely coupled as possible. Abstraction layers, together with the definition of their programming interfaces, are a well-established means to avoid unintentional dependencies. Although the methodologies of software design cannot be explained in the considerations presented here, we will use the most common techniques [8] of developing an object-oriented software framework for our task.

Internally, the framework uses the “data source perspective”, known as “federated data orchestration” [9], which allows services from different technology domains and sources to maintain their internal formats and bridges the differences between them. The advantage of this approach is that it eases the integration of multimedia content from “auxiliary” technology domains into the QLectives framework. An alternative approach is to use a single metadata format for all technology domains and sources, and force them to migrate to that format, but this requires major modifications in multiple technology domains, which we believe is unrealistic.

The approach chosen leads to a generic metadata architecture that will enable QLectives to use existing metadata types, like the media-optimised TV-Anytime

standard, as well as possibly entirely new, QLectives-specific or other types of metadata yet to be defined, without the need for defining another complete scheme.

In general, the existing metadata schemes are data definitions, often in a so-called XML (Extensible Markup Language) format. The problem is - they are "just data". They will not bring along any behaviour or knowledge. All kinds of behaviour would have to be implemented within the program that handles them. Certainly, we might programme so-called objects that encapsulate those data and also develop software that can handle them. However, those objects would simply be "dumb", not "smart". They would not be generic, and tailored just to specific application areas. In case a generic metadata system was to be made only of data definitions, it would have to be an empty "envelope" for anything, without any properties. However, what we intend to be able to do with this system can be abstracted, thus be made universally usable. This is more a matter of relationships than of data, and by adding behaviour to the abstractions, the objects of our framework can be made "smart". Concretisations will happen in the application layer by defining subclasses of our framework classes, or, if applicable, in adapter packages, where the bindings to external frameworks, like MPEG7, take place.

2.2 Metadata Framework Design

Naming scheme

Before defining any classes, it is advisable to create a naming scheme. When classes are defined, their names shall get some QLectives-specific prefix. For now, the prefix "QL" will be used for class definitions. Additionally, we may also decide to use Java-style naming, for example "eu.qlectives.", which would open the possibility to organise the framework into sub-packages.

Knowledge

The first central idea, forming the basis of the metadata model, is an abstraction of a central data type: The fundamental entity of our model, which performs this abstraction, shall be *"knowledge"*. In particular, that is "knowledge about something".

One could argue that this is too indirect, and that the fundamental entity could simply be that *“something”* itself. However, as the things we deal with and therefore want to describe the *quality of* are not all of the same type, and even are not absolute in many cases, this results in their metadata descriptions not being guaranteed to be invariable, unambiguous, complete, trusted, available, or true. An imaginable generic *"something"* base class would be property-less anyway, and would have to be sub-classed to become meaningful. Instead, we intend to subclass the “*knowledge”* class. The *knowledge-about-something*-class thus we name ***"QLSubject"***. Besides, there might be completely different types of knowledge about very different types of things. In example, we could think of knowledge about a metadata item, a user of some Web2.0 Service, a scientific citation, a BitTorrent .torrent file, or even some mathematical object deep inside a machine learning system. *How* each type of knowledge represents the connection to its "something" will be encapsulated in subclasses, which are outside the scope of the metadata model. Anyhow, wherever we have some information that is actually fixed, we can encapsulate it directly in a subclass. The only idea that remains in the base class is the "about-ness".

The vantage point of this type of abstraction is not only to get rid of the above mentioned "just data" problem, but also the inferring that we can make the knowledge objects "smart" – which will be discussed in the following sections.

Our central theme – quality - helps us to pose a productive question: How do we apply the notion of *"quality"* to *"knowledge"*? Whatever quality may be - regardless of how it might be evaluated and described - we will definitely want to increase it. That means to modify it, or more general, to *update* its current, outdated representation. At this stage, we even do not yet need to know the strategies and algorithms to perform and process that. We also do not need to take care of whether those algorithms will be implemented in a central, or in a distributed manner. What we will do, is to design our software system in such a way that those algorithms can be plugged in on demand, leaving their internal behaviour open, for later definition. Appropriate Algorithms for the creation of quality-generating collectives are currently under development and documented in the subsequent deliverable D4.4.2. Nevertheless, here are a few of the most important subclasses, assuming that quality will be represented by discrete entities:

- Quality has a "target" subject. For example, a media content metadata set, or a scientific paper.
- Quality can be updated. For example, due to knowledge about something having changed or evolved.

Quality

Thinking further, we notice that not only *"quality"* has these properties. For example, an *"origin"* would refer to a target, too, and it would be subject to updates, too, when we want to edit it, say, by changing its state from *"unknown"* to a *reference to some author*. Another valid example could be the *"rating"* of some QLSubject, which additionally encapsulates a value or a set of values.

What is common to these cases is that they relativise, or even to some degree qualify, some knowledge. So, the second central idea is to abstract this by introducing a new entity, called ***"QLQualifier"***. We will give the QLQualifier class a target QLSubject reference, and some updating mechanism.

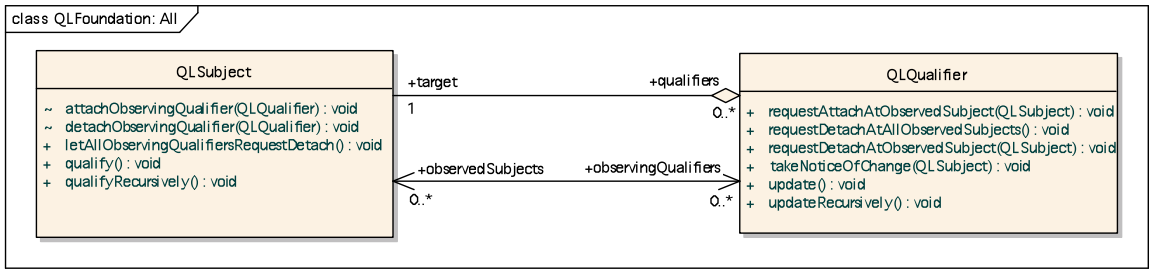


Figure 1: Class Diagram QLFoundation: QLSubject and QLQualifier

We only have what we know – our knowledge may change, or someone else might have some different knowledge about the same subject, and our system should still work in this situation. This infers that: “All knowledge can be relativised”. Thus, our QLSubject class will own a collection of QLQualifiers that are relativising the QLSubject. By doing so, the QLQualifier's "target" is always its "owner", too, since it is the back relationship of the mentioned collection. Additionally, it might make sense that a QLQualifier can also be relativised. For example, a metadata item can have a rating, where this rating can

have an origin and so forth. Creating (calculate-able) support for these meta-levels could easily be realised by making QLQualifier a subclass of QLSubject. Nevertheless, although this approach may appear to suggest itself, it can lead to inconsistencies in inheritance, if a meta-qualifier qualifies a super-class of itself, which additionally has subclasses that do not qualify anything. Therefore, it is advised to use the *"bridge"* design pattern instead [8], defining two companion classes associated to each other, one being a subclass of QLQualifier and one being a subclass of QLSubject. This scheme is realised by classes residing within the *"toolbox"*, which will be explained in a later section.

Knowledge updating

As knowledge may change over time, representations of knowledge need to be changeable, or better – updateable. Within the QLectives framework, updating knowledge representations can be an active, or a passive process. The former follows a "push" - principle, the latter follows a "pull" - principle. Our approach shall support both of them.

When there is some knowledge *B* dependent on some knowledge *A*, we can say that *B* is a conclusion of *A*. An active mechanism is useful if *A* "knows" how the conclusion B must change when *A* is updated. For example, one knowledge object might represent the statement "the earth is a disk" and hold a reference to the conclusion "beware of sailing too far". When we update the first knowledge object to "the earth is a globe", it can directly update its conclusion to withdraw the warning, replace the conclusion, or maybe remove it altogether. We need a passive mechanism in cases when nothing about the necessary changes in *B* is known by *A*. Instead, *B* retrieves the updated information from *A* that it needs to change itself.

In our example, the conclusion would only get notified that its precondition "the earth is a disk" is no more valid. It would then be the responsibility of the conclusion to re-check "how the earth is" and to decide which consequences to draw. The ***active updating mechanism*** API only requires a simple entry point, which we realise by an abstract method called ***"update()"***. What happens in a concrete case will be decided by the appropriate concrete subclass, but in general, we can argue that the class will invoke internal and/or external methods, which results in either an update, or a re-calculation of

the current value expressing the related quality aspect. The
only distinction that makes sense in the QLQualifier class already is whether updating shall be performed recursively, or not. So we define an additional abstract method ***"updateRecursively()"***. This method, like *update()*, shall be concretised in subclasses appropriately.

To invoke *update()* or *updateRecursively()* for all qualifiers of a subject, the following convenience methods are being added to the QLSubject

"qualify()", and
"qualifyRecursively()"

respectively. The default behaviour still may be overridden in subclasses.

The collection management API consists of the QLSubject methods

"addQualifier(QLQualifier qualifier)"
"removeQualifier(QLQualifier qualifier)"
"removeAllQualifiers()".

With respect to the ***passive updating mechanism***, we use the well-known *"observer"* design pattern [8] . The central point in it is a communication between two participants:
One QLQualifier, and one QLSubject that is not the QLQualifier's owner. The QLSubject can notify the QLQualifier of some state change in the QLSubject that can be evaluated by the QLQualifier to effect something appropriate. We create a dependency-behaviour that way. Therefore, we could call the participants "precondition" and "conclusion", too. "Concluding" is performed by invoking the QLQualifier method ***"takeNoticeOfChange(QLSubject observedSubject)"***, where "observedSubject" is the invoking "precondition" object.
In real life situations, a conclusion could depend on many preconditions, and a precondition might have many conclusions – causal connections are just one example. Hence, with respect to universal applicability, we define it as an N-to-M observer mechanism. Both participants have collections of their respective partner class, both for the communication purpose itself, and for easy cleanup of the connections on object removal.

The observer collections management API consists of the QLQualifier methods

"requestAttachAtObservedSubject(QLSubject observedSubject)"

"requestDetachAtObservedSubject(QLSubject observedSubject)"

"requestDetachAtAllObservedSubjects()"

and the QLSubject methods ***"letAllObservingQualifiersRequestDetach()"***.

Connecting and disconnecting is done indirectly, so consistency can be maintained automatically. Therefore, the auxiliary QLSubject methods

"attachObservingQualifier(QLQualifier observingQualifier)"

"detachObservingQualifier(QLQualifier observingQualifier)"

are non-public and only to be called by the request...() methods.

To recapitulate the design in table form:

QLSubject	knowledge about something
Relationships:	
qualifiers	relativise the knowledge
observingQualifiers	support of 'passive' update - registered here, but owned by other subjects
Methods:	
qualify	requests update for qualifiers then notifies *observingQualifiers*
qualifyRecursively	requests *updateRecursively* for qualifiers, then notifies *observingQualifiers*
addQualifier	collection support
removeQualifier	collection support
removeAllQualifiers	collection support
attachObservingQualifier	non-public
detachObservingQualifier	non-public
letAllObservingQualifiersRequestDetach	Observer collection support
QLQualifier	relativises a knowledge, observes changes on other subjects
Relationships:	

target	what it relativises, and object's owner
observedSubjects	where qualifier has been registered as observer; useful for resolving references on removal
Methods:	
update	re-evaluate what may have changed (may use cached information if sources are not accessible)
updateRecursively	re-evaluate all accessible information
takeNoticeOfChange	change of preconditions have occurred, conclusions may follow
requestAttachAtObservedSubject	observer collection support
requestDetachAtObservedSubject	observer collection support
requestDetachAtAllObservedSubjects	observer collection support

Table 1: Basic Design Listing

With these powerful and still flexible fundamental abstractions, our framework may become a compact foundation for a lightweight metadata toolbox, or it might even evolve to a full-blown multi-purpose expert system, depending on how it will be extended in derived classes and combined with other frameworks that are being developed in the other work packages, or with external ones. First steps of how some extensions for metadata handling can be designed will be shown in the following section.

2.3 Framework Extensions

The framework is extended with a package *"QLToolbox"*, which provides a "toolbox", a set of useful classes that can be used or sub-classed by applications. They abstract common tasks that are assumed to be shared by many applications. Though it is not mandatory to use them and all of their functionality can be realised in application classes, too, they may save the programmer some work.

Currently, the toolbox set is not considered complete or fixed. It is more a collection of ideas, which will continue to evolve in the future, especially within the working area covered by D4.4.2.

Origins

As already mentioned within the framework introduction section, the origin of a QLSubject instance would be a candidate for a common type of relativisation of knowledge. Therefore, we will subclass QLQualifier with a class ***"QLOrigin"***, as depicted in Figure 2.

What is common is that an origin consists of the combination of the subject's source, or "creator", and the circumstances of creation ("creationContext"), which can be, for example, the time or place of creation. An example for this could be a reviewer of a scientific paper (being the “creator”), and the context of the review itself (i.e. a specific conference). We let both of these attributes be unspecific QLSubjects, leaving specialisation to subclasses, if necessary.

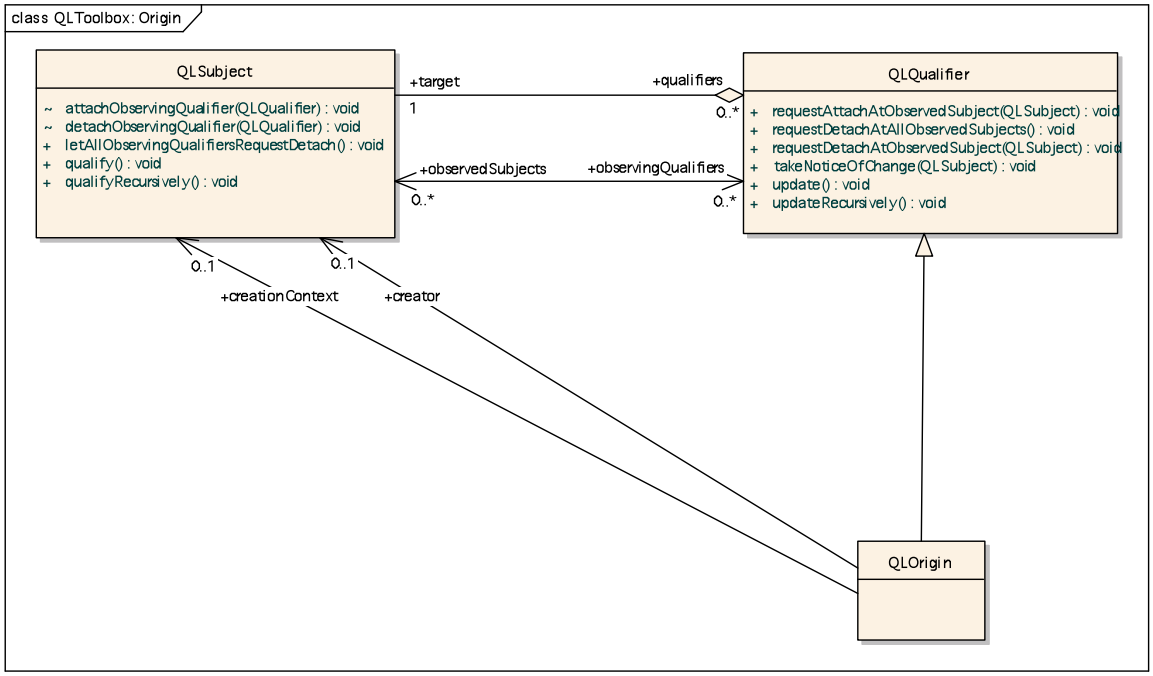


Figure 2: Class Diagram QLToolbox: QLOrigin

Meta Qualifiers

The recursive use of qualifiers (so-called meta-qualifiers) has already been mentioned. It is expected to be necessary in many cases, so we propose to provide a generic *“QLMetaQualifier”* class for that purpose. As being a QLQualifier, it is initially intended to be sub-classed. It has a companion class QLQualifierRepresentation, which is a

subclass of QLSubject that serves as the type for the meta-qualifier's target. Each QLMetaQualifier instance shall be a qualifier of a *"QLQualifierRepresentation"* object. Currently, there is no mechanism guarantying this type - pairing, but this may be added in a future revision. The QLQualifierRepresentation object, as shown in Figure 3, has a back relationship to the QLMetaQualifier ("metaQualifier"), and an association with the to-be-qualified qualifier ("representedQualifier"). If generic behaviour of "being meta" will be identified, the appropriate methods will be added at a later stage.

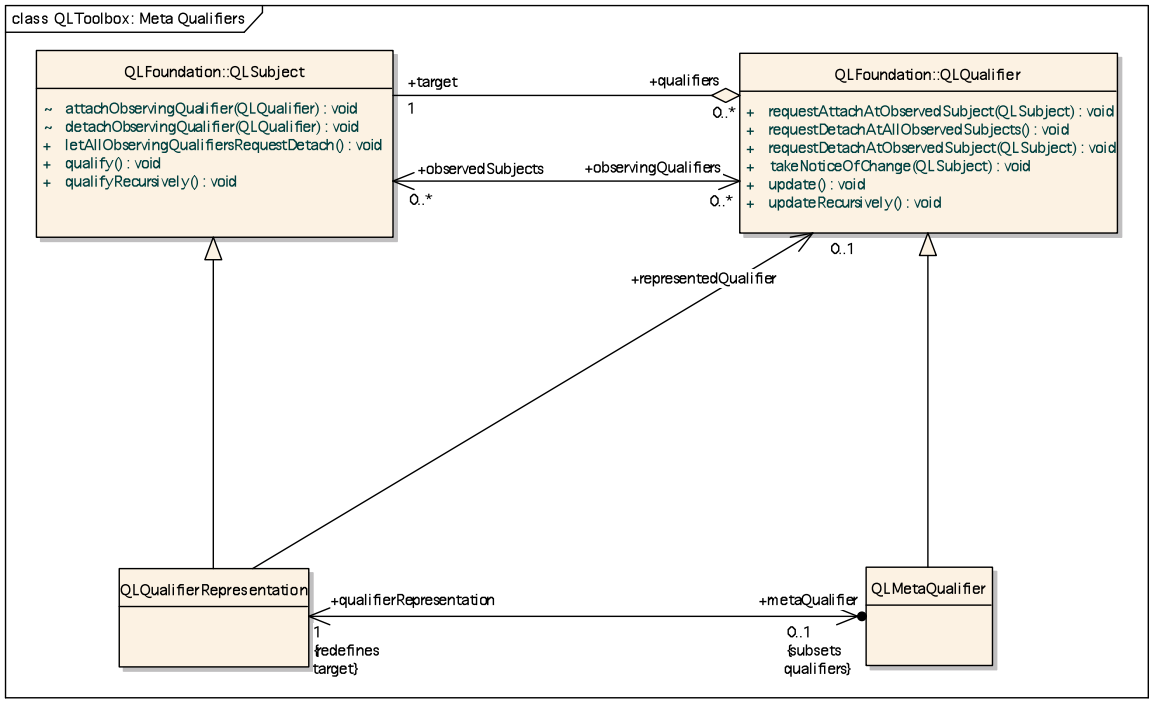


Figure 3: Class Diagram QLToolbox: Meta Qualifiers

Similarity

Similarity is another notion that will probably occur at many places in the considerations regarding quality. We could think of similar genres of multimedia content, several episodes of a TV series, or even several conferences on Peer-to-Peer networks. The appropriate toolbox class, *"QLSimilarity"*, is simply a qualifier with an additional QLSubject refererence ("comparisonTarget") for comparison with the qualifier's normal "target" and a query method to execute the comparison ("getExtent()"), which returns a numeric value for the extent of similarity. Figure 4 outlines this:

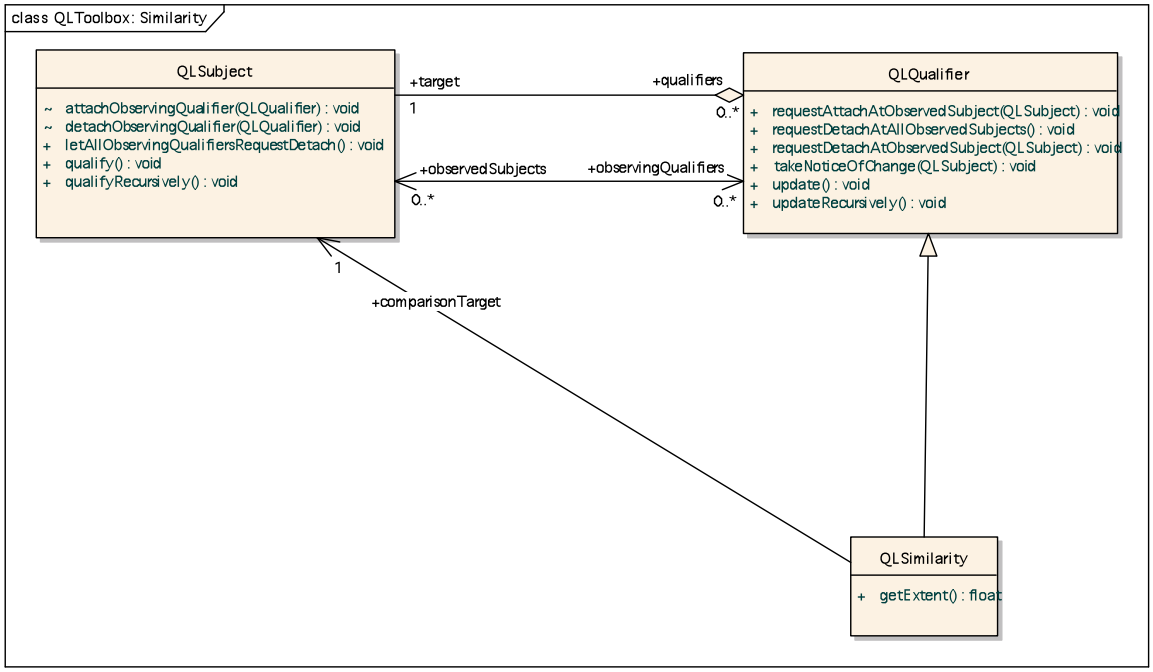


Figure 4: class diagram QLToolbox: Similarity

Binding to an external framework – an example

A QLectives application that utilises functionality of an external framework is dependent on both the QLectives framework and the external one, and therefore requires compile-time bindings to both of them.

TV-Anytime [1] is such an external framework. TV-Anytime is a standardised metadata format, based on MPEG7 [2]. It consists of definitions of data structures and their relationships for purposes within the TV and Multimedia fields, being intended to be used to create metadata services.

In order to avoid having to make QLectives dependent on TV-Anytime, an *adapter* is created that encapsulates the binding to the TV-Anytime schema. It is a package that forms an abstraction layer on top of both frameworks and is made to be re-used by multiple applications. Its member classes are designed as "wrappers" around TV-Anytime entities with a QLectives-conformant API.

The TV-Anytime adapter package contains a number of QLSubject and QLQualifier subclasses. An excerpt of the package shows an example in the following class diagram:

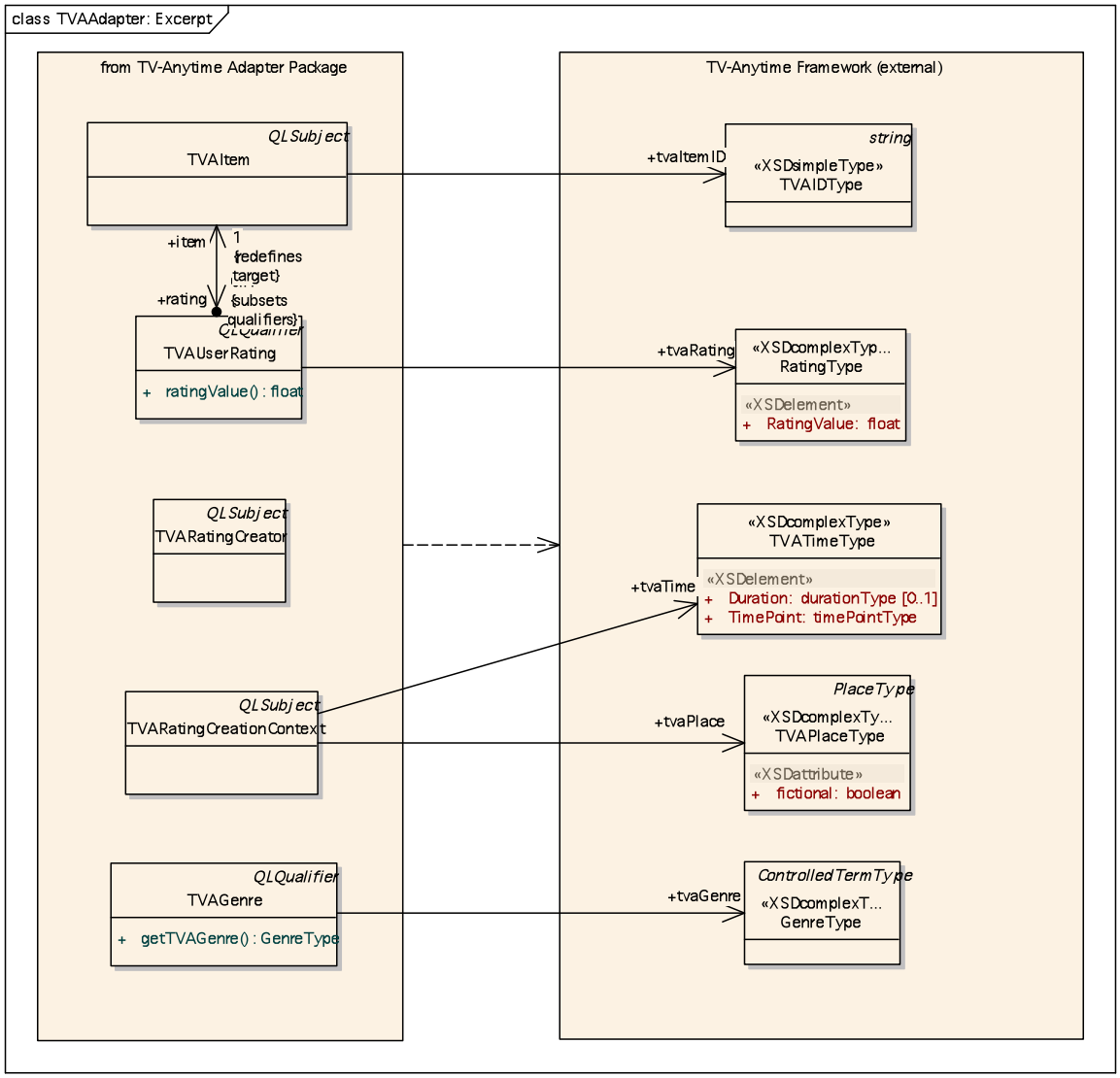


Figure 5: Class Diagram TVAAdapter: an excerpt

Which of the adapter classes should be QLSubject subclasses and which are QLQualifier subclasses depends on decisions that should be made in a more concrete stage of the development process. Here, for example, *TVAItem* is a QLSubject subclass, and *TVAGenre* and *TVAUserRating* are QLQualifier subclasses, intended as qualifiers to relativise TVAItem subjects.

An adapter class may be a "wrapper" for one TV-Anytime entity, like TVAItem and TVAGenre, or multiple ones, like TVARatingCreationContext. For example, a TVAItem object wraps a Tva_metadata_3-1_v141::TVAIDType object. *TVARatingCreator* and *TVARatingCreationContext* are specifically prepared to be actors of creator and creationContext in QLOrigin objects. It would also be possible to define a TVAnytime-specific rating origin, but we left this out of the example, in order to show within the following section that a possible alternative is to make the specialisation in the application layer instead.

2.4 Case Studies

In this section we depict, based on examples from the QMedia, as well as the QScience world, how the generic design of the Metadata Model works.

2.4.1 QMedia Use Cases

The first example is a scenario of an implicit update of incomplete "Genre" information set within a multimedia application. It demonstrates the use of adapters for external framework bindings, as well as of the updating mechanisms, both active and passive. Additionally, usage of the toolbox is shown for QLOrigin. We assume there are media items of some type, which is a subclass of QLSubject. As the application uses TV-Anytime - based metadata, it is assumed that these classes or their super-classes have been defined in a QLectives-TVAnytime adaptation package, and provided with "intelligence" (implementation of algorithms, as will be defined in D4.4.2) and the necessary bindings towards the external TV-Anytime framework. We will re-use the "TVAAdapter" that was introduced as an external binding example in the previous section, and therefore, the contained TVAItem class is just appropriate as an item type.

The adapter package classes can be used either by defining application-specific "wrapper" classes, or by sub-classing them. The former approach will be followed to show an example support for user ratings, the latter will be applied for an example support for automatic updating of genres.

Use Case: User Ratings

User ratings are represented by the event that a user gives or assigns onto the rating value, encapsulated in a class "*RatingIncidence*". There is an associated qualifier class "*RatingOrigin*" (a subclass of QLOrigin), which uses the TVAAdapter classes TVARatingCreator and TVARatingCreationContext as its creator and creationContext references, respectively.

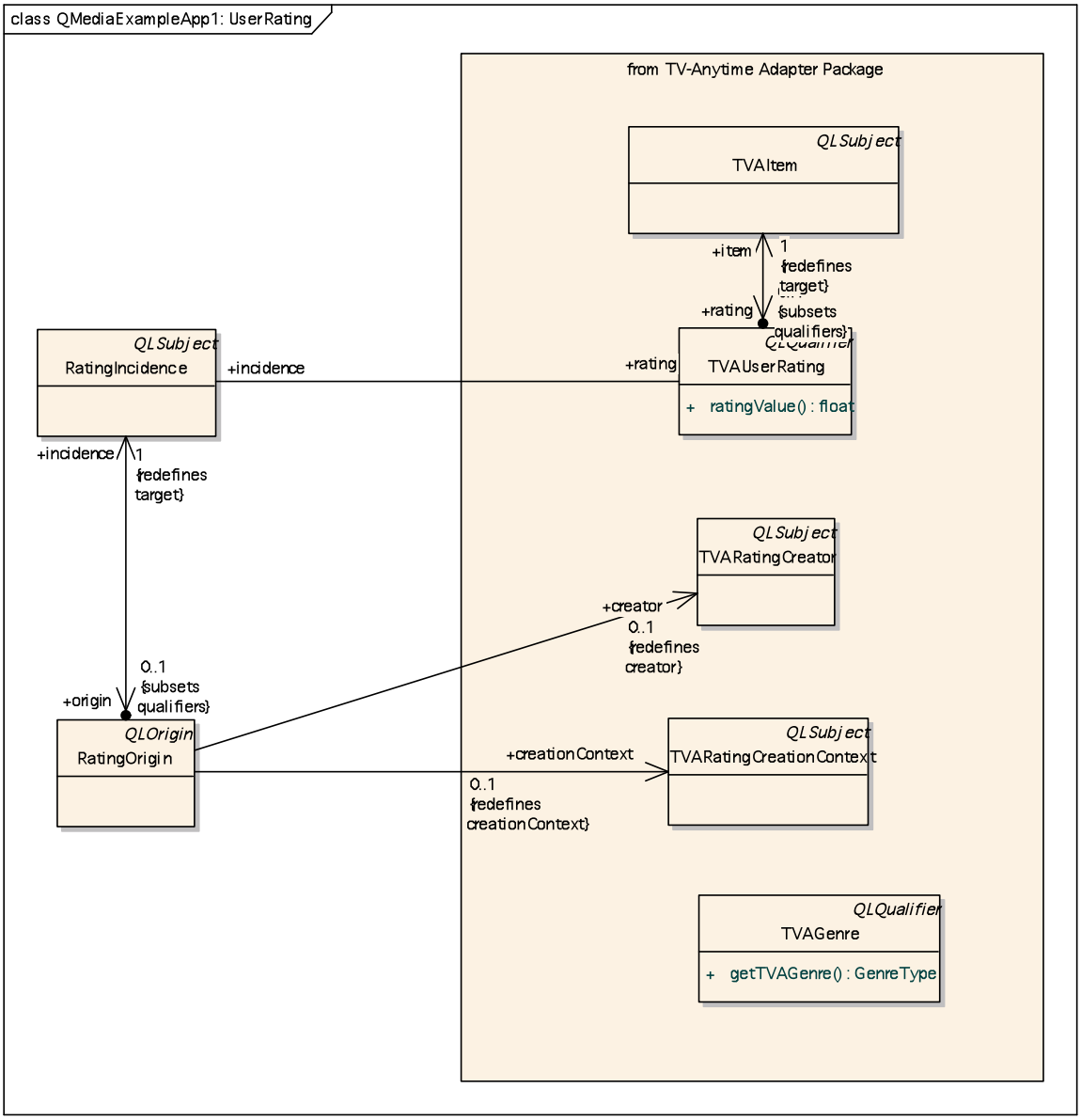


Figure 6: Class Diagram QMediaExampleApp1: UserRating

Furthermore, our application uses an external, hypothetic expert system for genres, which may be a "crowdsourcing"-based association mechanism. It may do its work by

finding similarities of titles by clustering the occurrences of their mention and their associations with mentioned genre names, but it may also incorporate other strategies.

We will use an adapter package "XYZAdapter", equivalent to the external binding example introduced in the previous section. This expert system package provides an API in the form of an object "theGenresKnowledge" of type XYZGenresKnowledge. It represents each genre to be resolved by an object of type XYZGenreResolver (a subclass of QLSubject), which provides access to the XYZ-specific genre representation "XYZGenre".

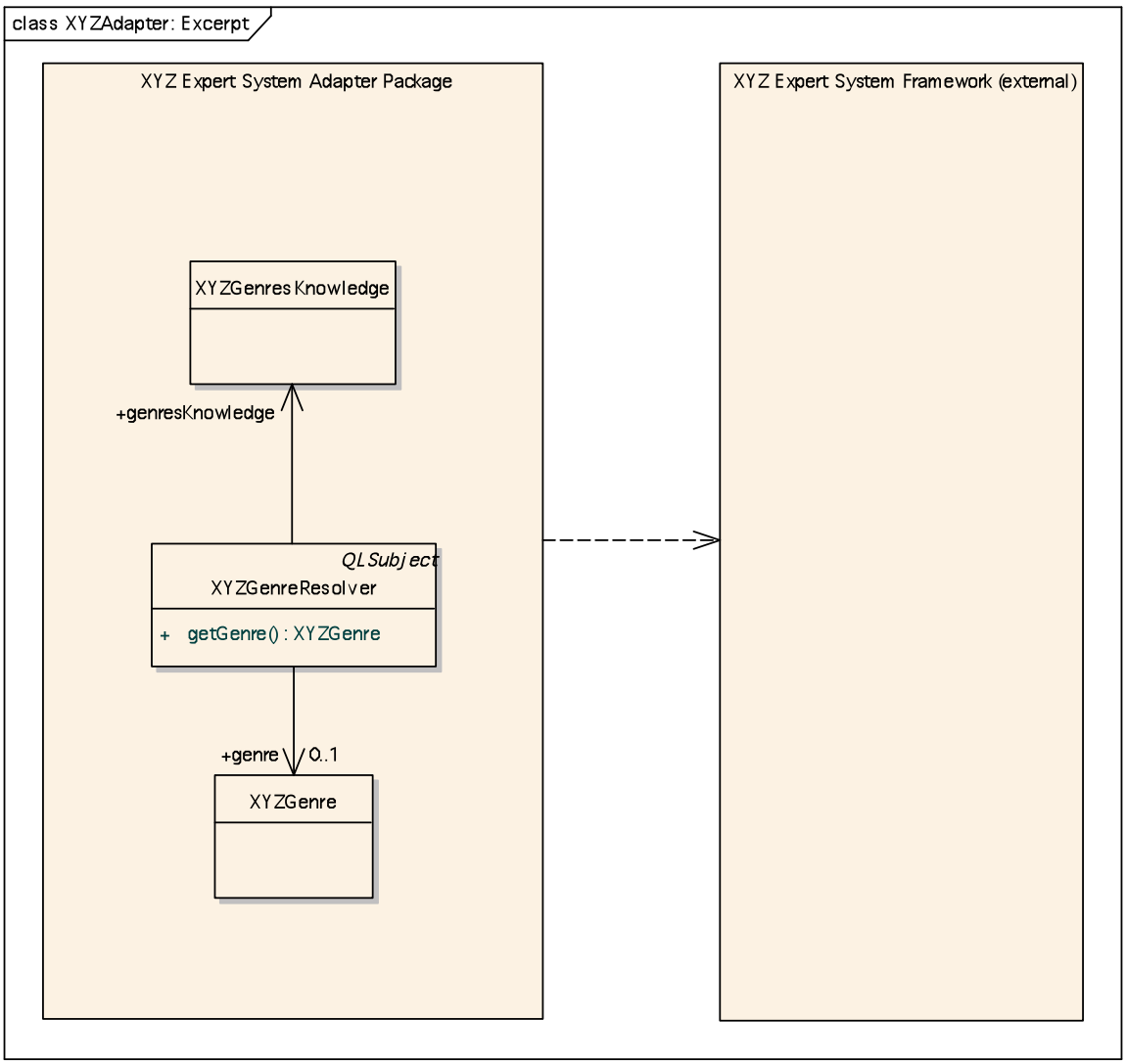


Figure 7: Class Diagram XYZAdapter1: an excerpt

The expert system specific, as well as the media specific classes reside in distinct adapter packages, containing bindings to the QLectives foundation framework only, but not between each other (modular approach). Only the application interconnects them, by establishing bindings to both of them. That way, the encapsulation of packages is maintained, although their objects can still communicate. The layered software architecture is preserved, and external bindings are kept as local as possible. The connecting element between the TV-Anytime domain and the XYZ domain are genre qualifiers of the type GenreQualifier. Since TVAGenre is already a QLQualifier, we have chosen to make GenreQualifier a direct subclass of it, in order to demonstrate that inheritance is a viable option for our purpose in this case. GenreQualifier may use a reference to its associated genre resolver in the expert system (the instance variable "resolver"), but for our example communication process, this is not necessary.

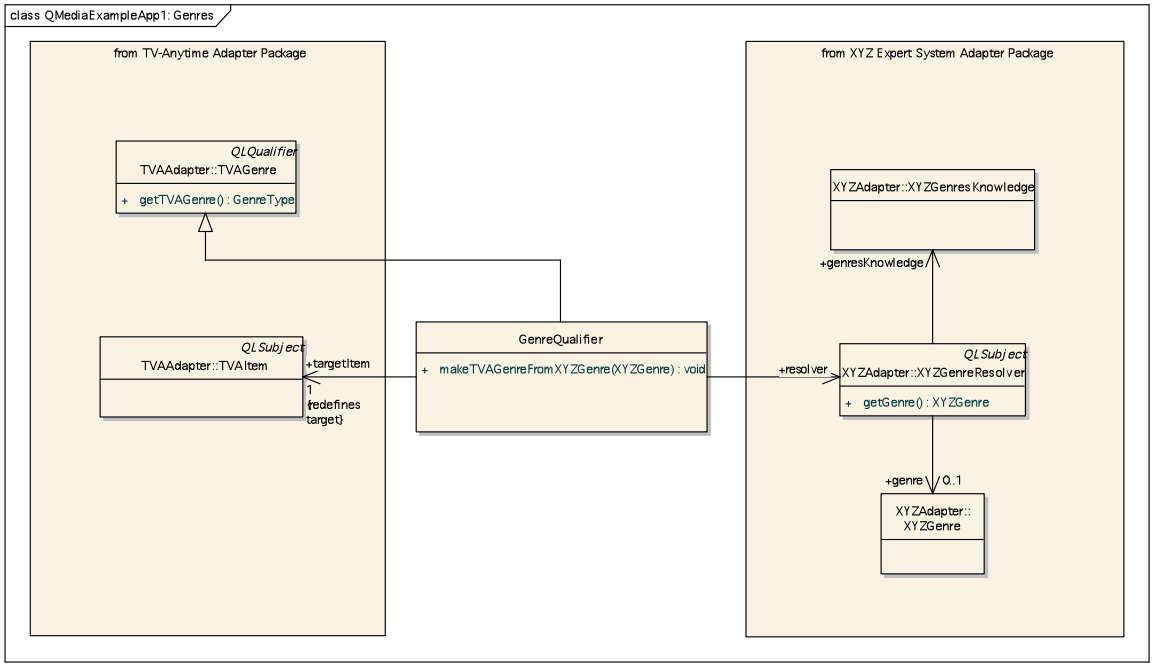


Figure 8: Class Diagram QMediaExampleApp1: Genres

Use Case: Genre Update

The following use case will be used to explain one possible communication process:

We have one media item object 'anItem' of a music track, which has a genre qualifier 'qualifiers[i]' that denotes that the genre is unknown. Additionally, we have another

reference 'aGenreQualifier' *onto* that qualifier. The genre qualifier was produced by some retrieval mechanism from a metadata table, but the table had an empty genre field, since its author did not know the genre yet. The reason for that was that the musical style was somehow new, and would not fit into any of the already known genres, at best a very broad category, and there was no agreed new genre name yet. Hence, the retrieval mechanism generated a qualifier encapsulating an "unknown genre". When asked for the text representation, the qualifier returns the text "(unknown)". If our program tries calling 'aGenreQualifier.update();', there will be no change, since the qualifier will get no new information, wherever it may want to derive it from. We connect to the genres expert system by registering our unknown-genre qualifier object as a conclusion at a GenreResolver object 'aGenreResolver' owned by the expert system object 'theGenresSubject' **'aGenreResolver.registerConclusion(aGenreQualifier);'**. This 'aGenreResolver' has been created specifically for our missing genre.

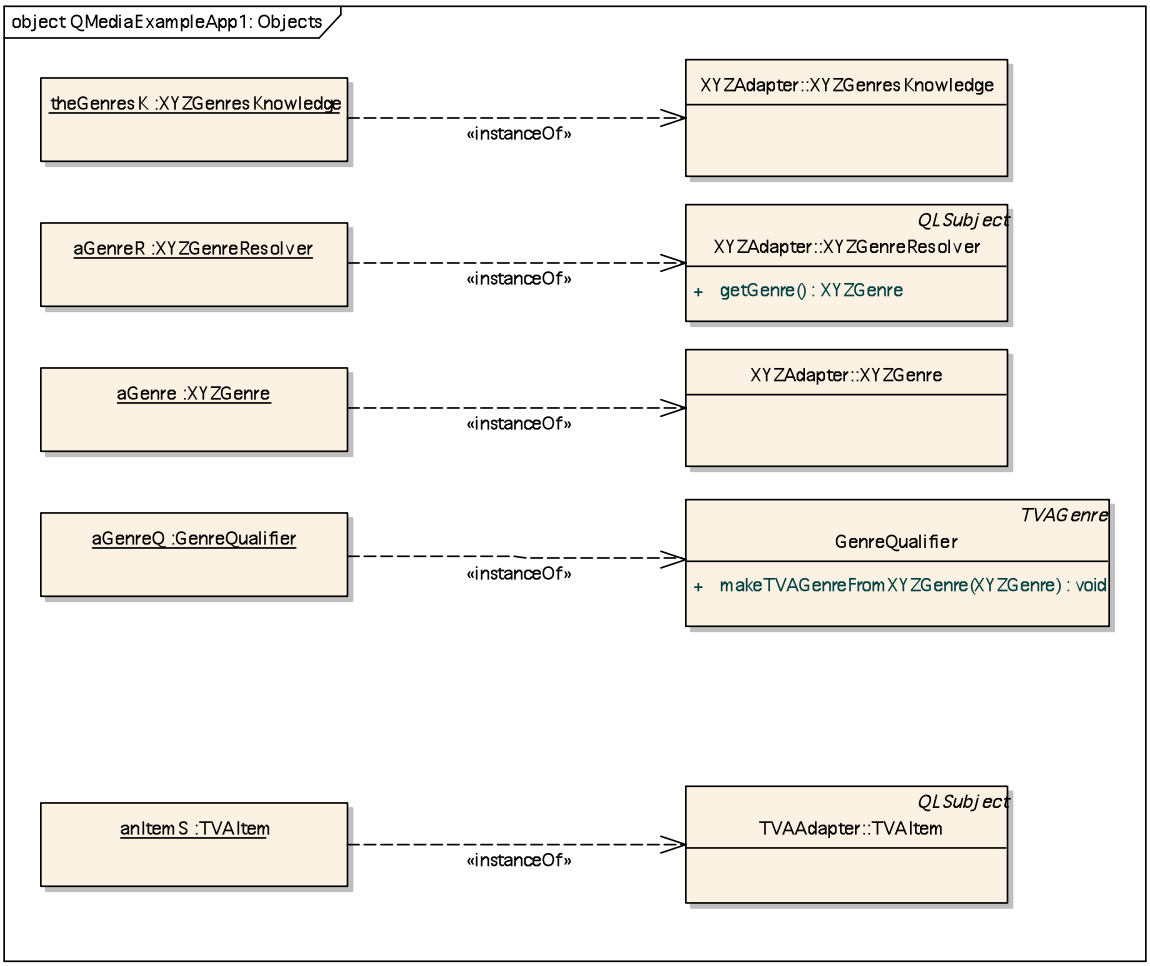


Figure 9: Class Diagram QMediaExampleApp1: Objects

Now, as musical styles evolve, people communicate about them, and at some point in time invent a new genre name when they feel that there should be one. This evolutionary process will be observable in digital social networks, and it makes sense to "tap" this information generation process. Our expert system performs such observations, and reacts on appearance of notions, meanings and associations. At some point in time, it decides that there is a new genre definition, and the already mentioned music item assigned to it. From now on, the genre information exists in the knowledge base of the expert system, represented by 'theGenresKnowledge', and is made accessible through our genre resolver object by invoking the 'qualify()' method. This method is implemented in a way that it propagates the "news" by calling 'takeNoticeOfChange()' on all its *registered observers*, one of which is the unknown-genre qualifier 'aGenreResolver' of our music item. By that, our genre qualifier gets the opportunity to change its state from 'unknown' to 'resolvable' and notifies its target subject, the media item by calling 'target.qualify()' or 'target.qualifyRecursively()'. This in turn re-executes 'aGenreQualifier.update()', which this time yields a valid genre name.

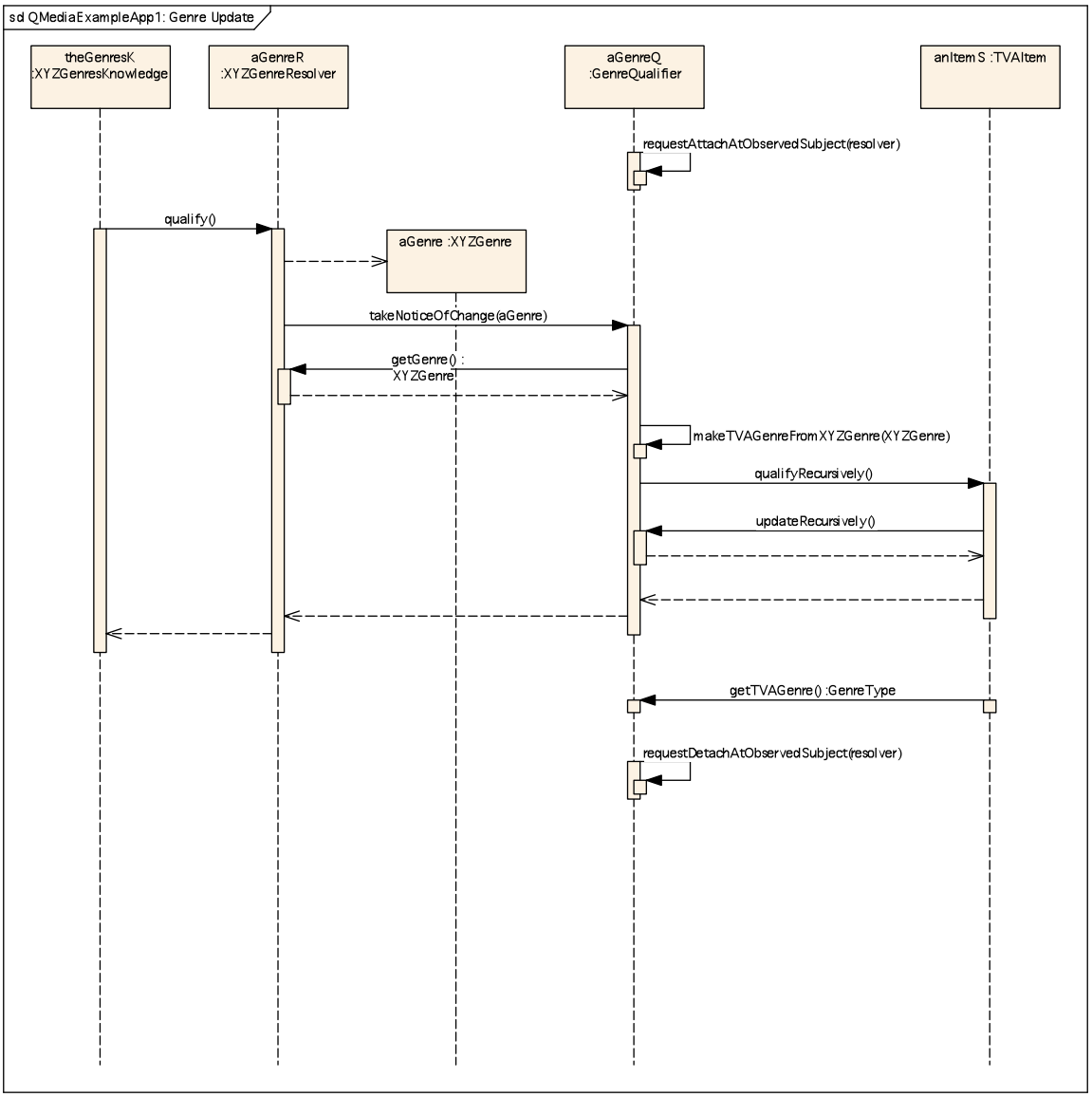


Figure 10: Sequence Diagram QMediaExampleApp1: Genre Update

We have shown how some new piece of information propagates through the system and effectuates changes on only a single other object. Nevertheless, all other required effects towards other objects are still carried out the same way – automatically, enabled by our updating mechanisms.

2.4.2 A QScience Use Case

The second example shows a construct that supports a publication reviewing process within a scientific information management application.

The framework classes QLSubject and QLQualifier are consequently and extensively used, that is, specialised by defining subclasses. Only structural aspects of the design are shown; behavioural aspects are assumed to be realised with the mentioned updating mechanisms of the framework. Additionally, the usage of the toolbox is shown for QLOrigin. For simplification issues, neither external frameworks, nor their corresponding adapter packages are depicted here.

The central parts are entities that represent scientists ("*Scientist*") and their written products ("*Product*"). A product in this context can be either a scientific paper ("*Paper*") or a review ("*Review*"). A scientist in this context can be a paper author or a review author or can play the roles of both, a paper author and a review author. To represent the latter, there is a relation class "AuthorRole", of which "PaperAuthor" and "Reviewer" are subclasses.

QLQualifier subclasses are used to assign a relevance ("Relevance") to a Product and a reputation ("Reputation") to a Scientist, or to be more precise, to the appropriate AuthorRole. Ratings ("Ratings") form the basis of both, Reputation and Relevance.

QLOrigin subclasses designate the authorship of Papers and Reviews, and the origin of Ratings, which is always a Review.

A Review is about a Paper or, recursively, about another Review, thus generally about a Product. But since Review is already derived from QLSubject, it cannot be a QLQualifier. Therefore, the qualifying property is *split off* and assigned to a *separate companion* class ("*ReviewResult*"), which is a QLQualifier. Review and ReviewResult instances are meant to occur pair-wise. Here, we again applied the same "bridge" design pattern - like in the toolbox - but this time for a "meta-subject", instead of a meta-qualifier.

The design is depicted in the following class diagram:

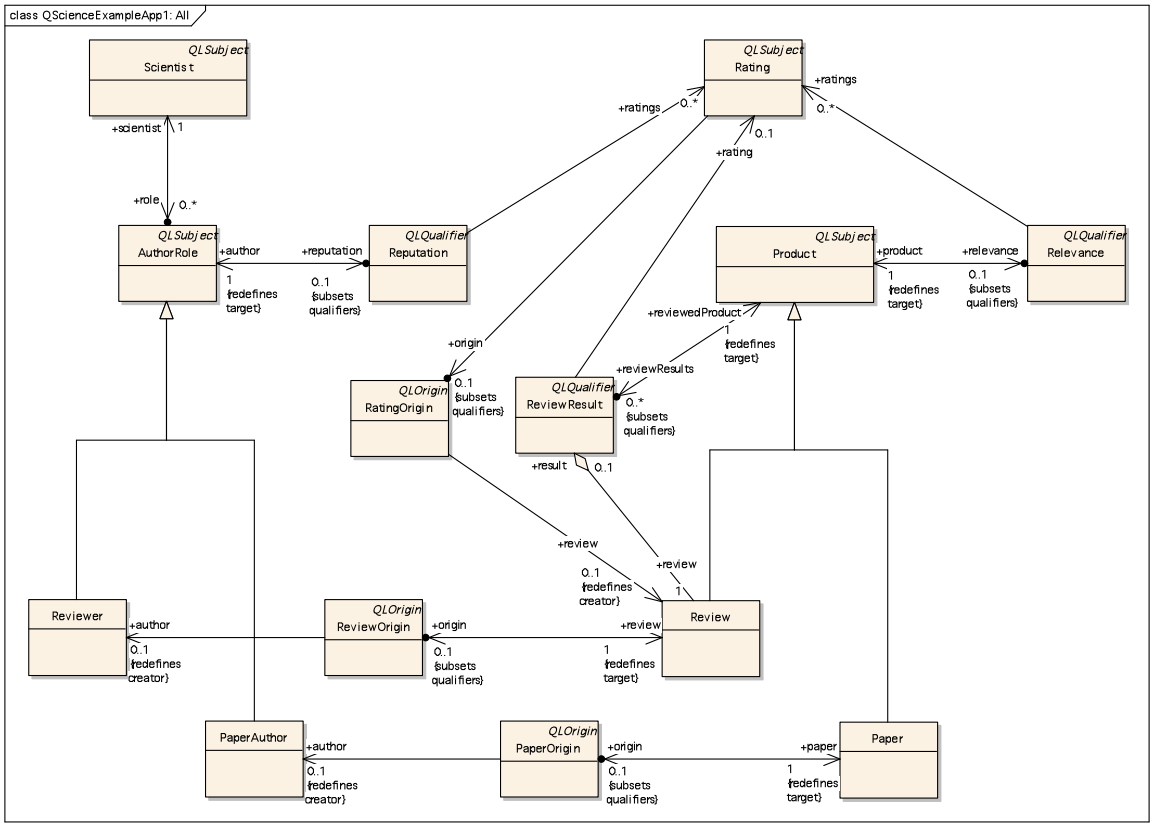


Figure 11: Class Diagram QScienceExampleApp1: All

Of course, a complete application design would define many more classes that make up "scientific quality", like, for instance, citations. Moreover, it would have to carefully define the formal behaviour of each implementation of the methods update(), updateRecursively(), qualify(), qualifyRecursively() and takeNoticeOfChange() in each QLSubject and QLQualifier subclass.

3 Conclusions

In this document, we introduced a generic metadata framework to support the representation of quality-related aspects within the media – related domains, as well as the "world of science". It realises this by being designed flexibly enough to allow implementations via several metadata formats, thus maintaining the possibility to realise various representations of (meta)data that describe, rate or enhance user choice and expertise of available content – independent of the *type* of content (i.e., multimedia content, scientific papers etc.). It builds the basis for the design of a peer-based metadata subsystem, which will - in combination with advanced filtering evaluation algorithms, designed in D4.4.2 – promote high quality metadata and limit the distribution of low-quality and incorrect metadata.

Our metadata model can be adapted towards existing metadata formats such as TV-Anytime, MPEG7 or Dublin Core [7], as shown in section 2.3; additionally, various up-to-date metadata requirements can be adapted to the metadata subsystem, such as segmentation information metadata, personalisation and recommendation, tagging, emotion-descriptive data, community-oriented rating and others. To realise this, we have decoupled our model from existing metadata formats, and excluded the actual data parts from the fundamental abstraction layer. Concepts we expect to be necessary can be realised by extending this foundation in the next abstraction layer of our framework, where further sub-classing allows for building specific applications with possible bindings to third-party frameworks.

We outlined that knowledge can be relativised and introduced a generic way of representations for M-to-N relations, descriptions, and knowledge about objects. Within D4.4.2, we will define the *algorithms* for a processing system for such knowledge objects, and a refined set of the QLectives Toolbox.

Finally, we have verified applicability to exemplary use cases, for the multimedia world (QMedia) in section 2.4.1, as well as for the world of science (QScience) in section 2.4.2.

4 References

[1] TVAnytime Forum: http://www.tv-anytime.org

[2] MPEG, MPEG-7 Overview, Oct 2004: http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm

[3] Unified Modeling Language: http://www.uml.org

[4] YouTube website: http://www.youtube.com

[5] PrestoSpace metadata format: http://prestospace.org

[6] Tribler Peer-to-Peer network: http://tribler.org

[7] Dublin Core Metadata Initiative: http://dublincore.org

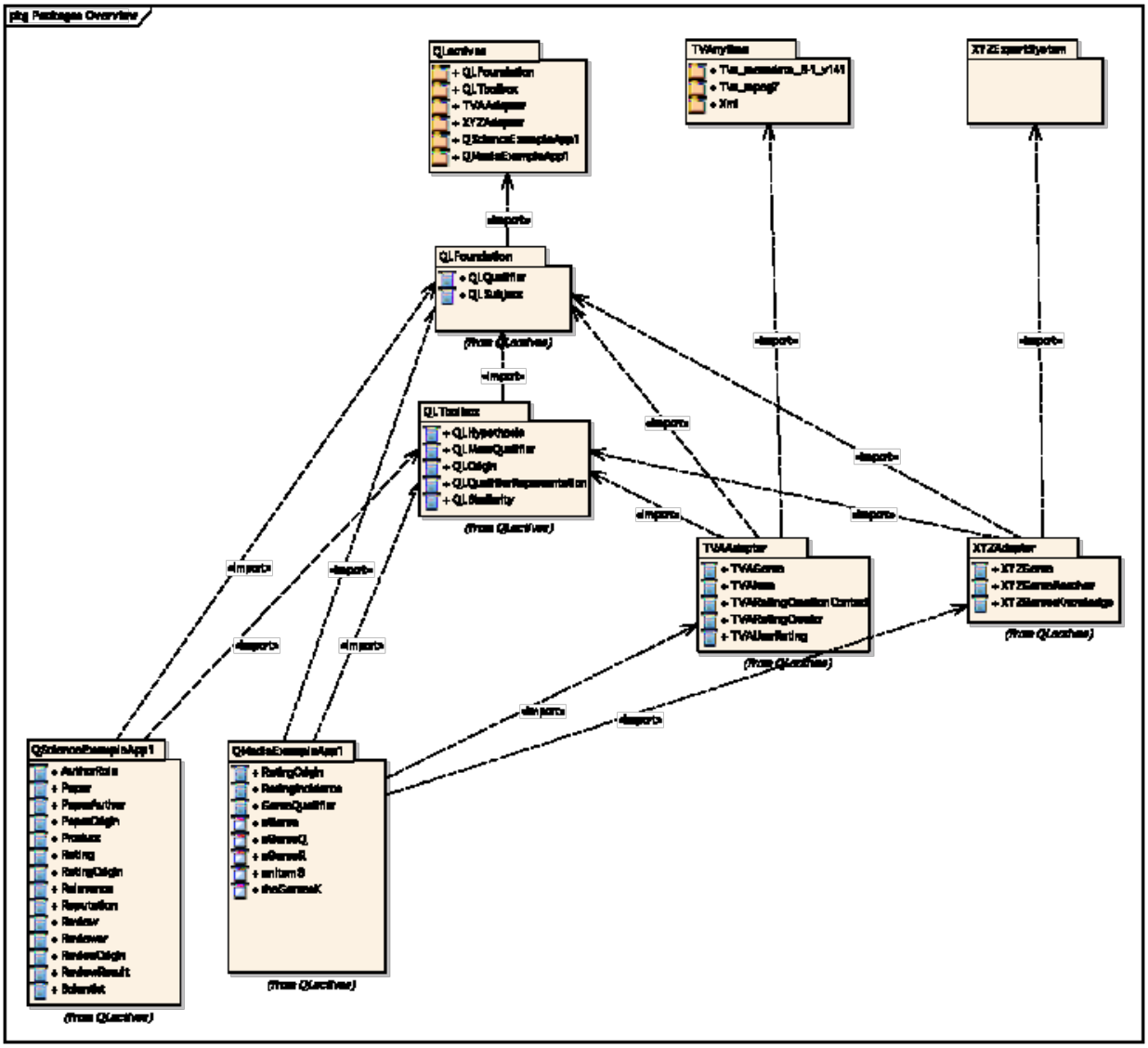
[8] E. Gamma, R. Helm, R. Johnson – "Design Patterns, Elements of Reusable Object-Oriented Software" [ISBN: 978-0-2016-3361-0] http://www.pearsoned.co.uk/student/detail.asp?item=171742

[9] S. Lahanas, "Enable SOA Transformation and Cross-Domain Data Fusion", DM Review Magazine, January 2008.

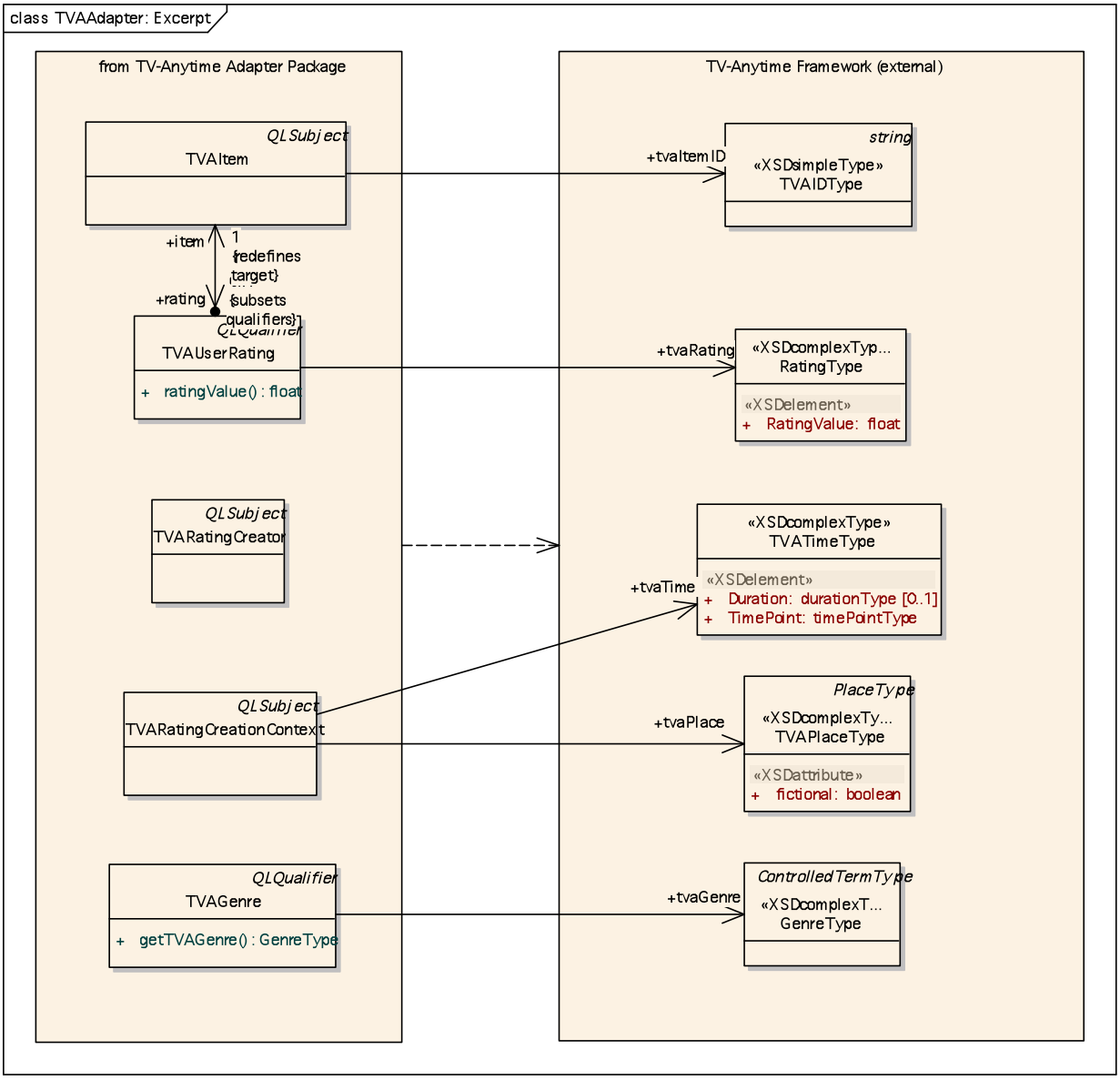
5 Annex A: UML Diagrams

In this section, we present additional UML diagrams, which provide further insight on the QLectives Metadata Model.

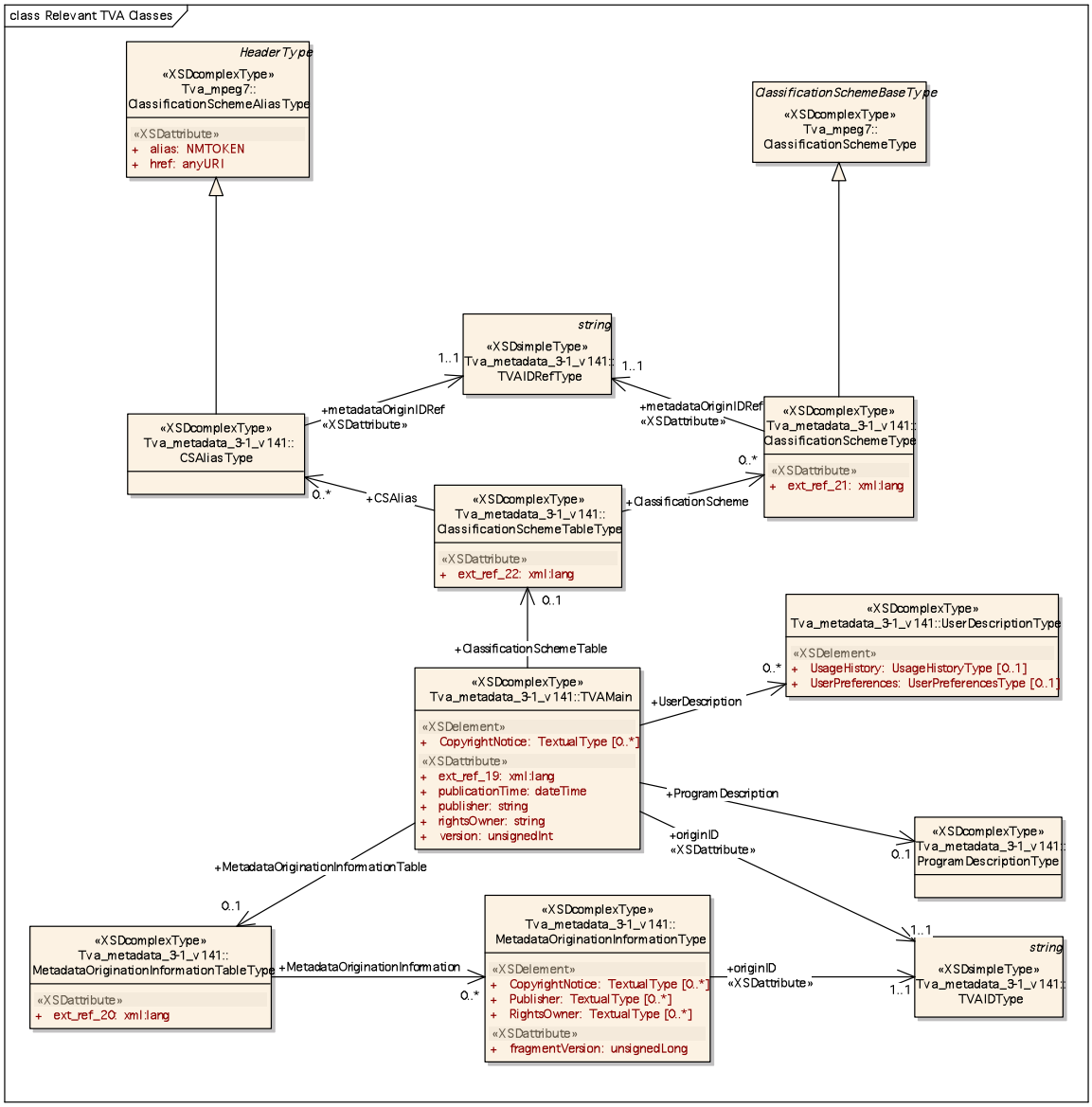
The complete Metadata Framework is attached to this Deliverable, given in the UML – typical XMI format.



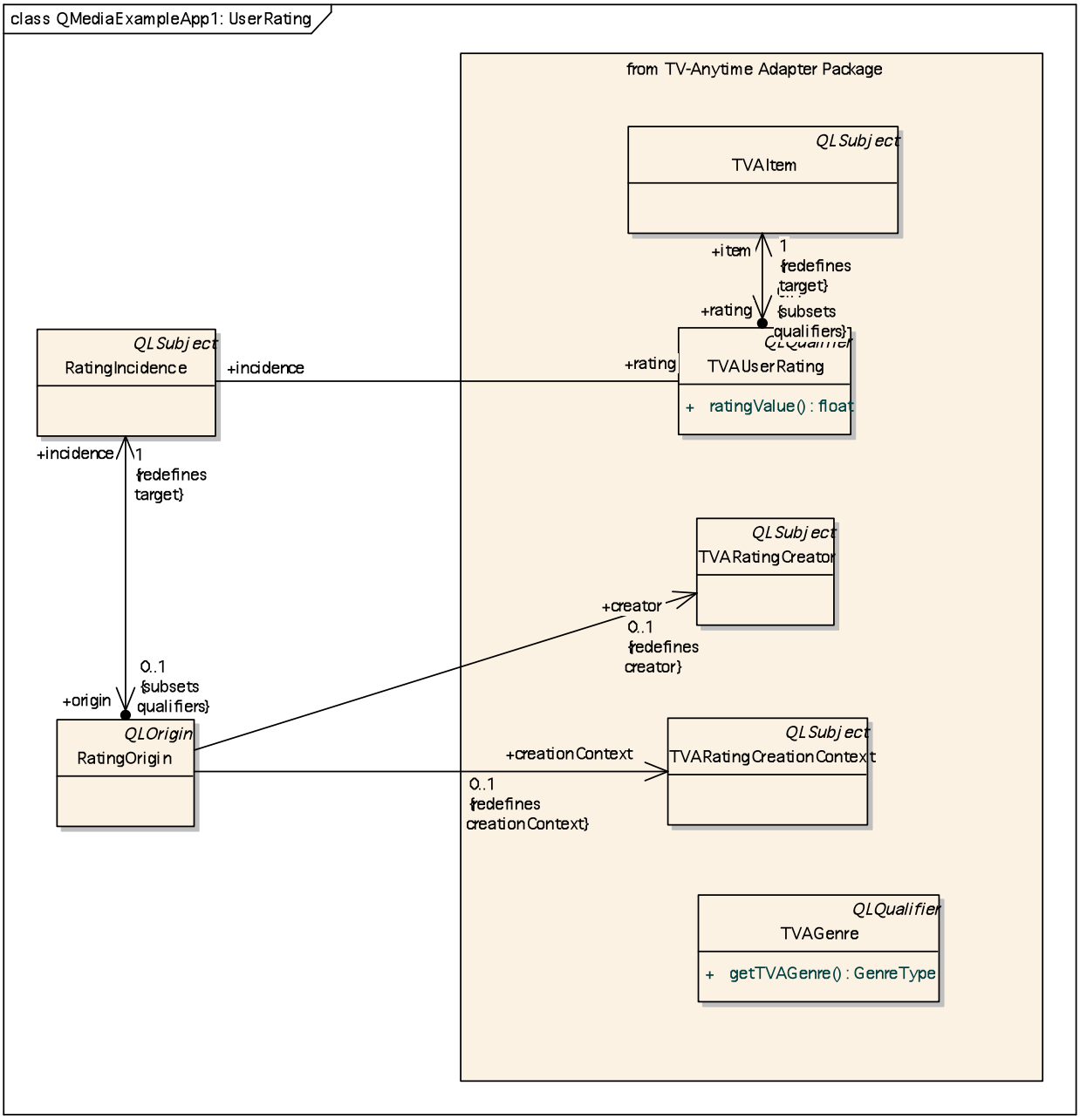
UML Diagram: Packages Overview



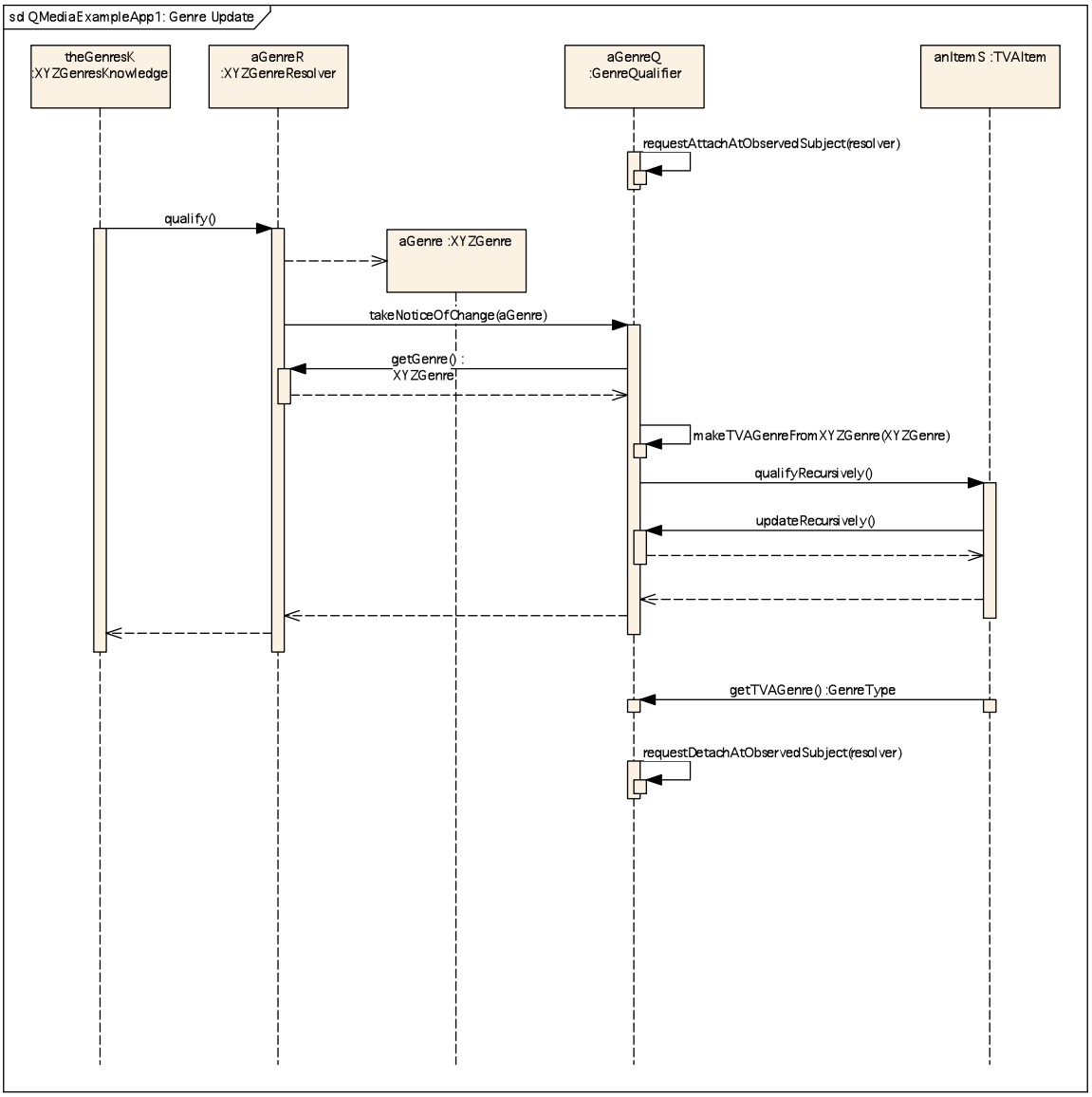
UML Diagram: TV-Anytime Adapter



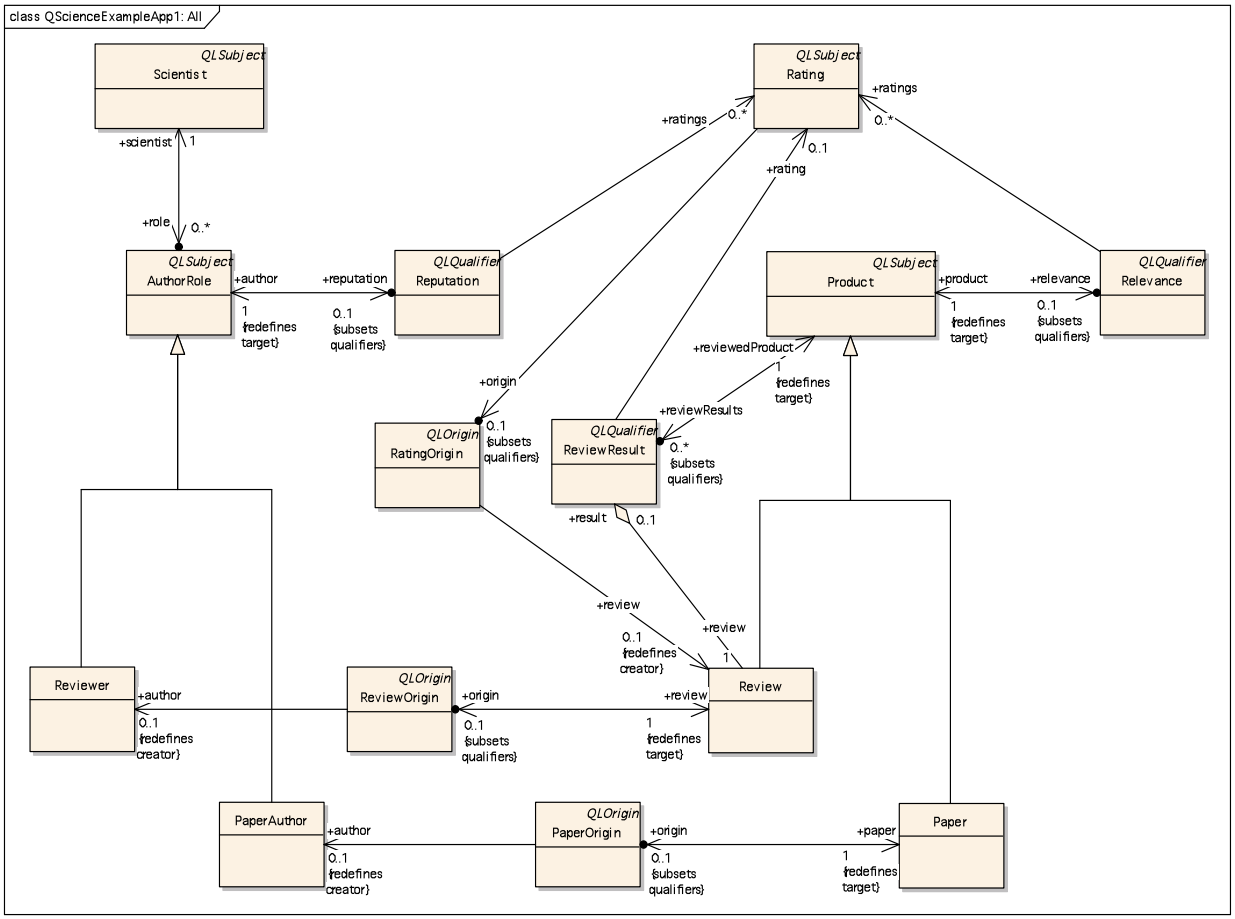
UML Diagram: Relevant TV-Anytime Classes



UML Diagram: QMedia Example User Rating



UML Diagram: QMedia Example Genre Updating Process



UML Diagram: QScience Example