

g-Eclipse final Architecture

g-Eclipse Deliverable D1.8

WP1.1

Document Filename:	D1.8.pdf
Workpackage:	WP1.1
Partner(s):	FZK, INO, JKU, PSNC, RUR, UCY
Lead Partner:	FZK
Config ID:	D1.8
Document classification:	PUBLIC

Abstract: This document provides an overview of the g-Eclipse architecture at the second year of the project, after the implementation work for the second middleware was started. The g-Eclipse project intends to provide a uniform and extensible platform for potentially all existing and even emerging grid middlewares. One central problem which has to be addressed by the architecture is therefore the abstraction of the middleware access policies. As a first exemplary implementation, support for the gLite-middleware and the related VOMS security service was implemented during the first project year. Starting with the second project year the implementation work for the GRIA middleware was undertaken. This second implementation didn't require changes to the already available architecture, helping to validate it.



This document is intended to be gender neutral. Nevertheless, we use gender-specific language where expedient to ensure simplicity.

Delivery Slip

	Name	Partner	Date	Signature
From	Mathias Stümpert	FZK		
Verified By				
Approved By				

Document Log

Version	Date	Summary of changes	Author
0.1	10/13/2006	First Draft	Mathias Stümpert
0.2	05/07/2007	Update of architecture	Mathias Stümpert
0.3	25/01/2008	Update of architecture	Ariel García

Contents

1	Introduction	7
1.1	Grids and user roles	7
1.2	The g-Eclipse framework	8
1.3	Benefitting from the Eclipse framework	8
2	The Grid Model	9
2.1	Grid element hierarchy	9
2.1.1	Abstraction layer	9
2.1.2	Public implementation layer	14
2.1.3	Internal implementation layer	15
2.2	The Grid project	17
2.3	Grid element creators	19
2.3.1	Creator hierarchy	20
2.3.2	Model creation process	21
2.4	Grid element managers	22
3	Authentication and Authorisation	24
3.1	Server-side authentication	24
3.1.1	Authentication tokens	24
3.1.2	Token management	25
3.1.3	Token providers	26
3.2	Client-side authentication	27
3.2.1	CA-certificates	27
3.2.2	The CA-certificate manager	27
3.3	Managing passwords	28
4	Error reporting	29
4.1	The Eclipse way of reporting runtime errors	29
4.2	Extension of the error reporting mechanism within g-Eclipse	29
4.2.1	Problems	30
4.2.2	Solutions	31

5	User interface	32
5.1	Grid model views	32
5.1.1	Abstraction layer	32
5.1.2	Grid project view	33
5.1.3	Grid job view	34
5.1.4	Grid connection view	34
5.2	Perspectives	34
5.2.1	User perspective	34
5.2.2	Developer perspective	35
5.2.3	Operator perspective	35
5.2.4	Grid exploring perspective	36
5.3	Wizards	36
5.4	UI part of the Error Reporting mechanism	37
5.4.1	Active solutions	37
5.4.2	The problem dialog	38
A	Screenshots of the most important UI elements	39
A.1	Views	39
A.2	Perspectives	41
A.3	Dialogs and wizards	45

Changelog

- Smaller updates in the whole document to reflect the changes in the APIs, perspectives, etc.
- Chapter 4: updated to reflect the new error reporting plug-in
- Chapter 5.4: updated to reflect the new error reporting plug-in
- Chapter A: new set of screenshots

1 Introduction

This document is intended to give an overview of the g-Eclipse architecture. It may be seen as an update of the second architectural deliverable that was produced in month 9 of the project's lifetime [1]. Therefore, many concepts included here were already described in this last architectural document. However, and as a result of the refinement of the g-Eclipse architecture in the last 9 months of development, some concepts are new and described here for the first time. The general architectural patterns such as the Grid model (see chapter 2), the security aspects (see chapter 3) or the Eclipse Filesystem based connection mechanisms have reached production quality with month 18. They are already used throughout the whole framework and several middleware-dependent implementations are already based on these patterns. Furthermore, the UI parts of the architecture (see chapter 5) provide an easy and common access to these abstraction mechanisms. These UI contributions are in current use although several still undergo some minor development to improve the user-interaction. On the other hand, the error reporting mechanism (see chapter 4) was rewritten as a separate plug-in with the aim of allowing its inclusion in the Eclipse codebase.

As the links to the Eclipse side did not change since the first version of this document, we will focus more on g-Eclipse architecture than on the links to the Eclipse architectural patterns that are used within our architecture. Nevertheless, where appropriate, we will, at the very least, mention the Eclipse aspects that are important for a specific pattern within g-Eclipse. For further details we refer to the first architectural deliverable of month 3.

1.1 Grids and user roles

A Grid user can have diverse roles within the Grid. As g-Eclipse aims to become a standard tool for all Grid users, we have to carefully consider each conceivable role users can play in the Grid. The main roles are listed below.

Grid application users: Grid application users want to run their applications on the Grid. They have to be able to submit jobs, to monitor the progress of the submitted jobs and to cancel jobs. They have to have access to files on the Grid which could be input files for the jobs or resulting output files from the jobs. Therefore standard operations for files (i.e. copying, moving, renaming, ...) should be made available to application users. After a job has finished, application users should be able to visualize the resulting data either locally or remotely.

Grid operators: Grid operators have to have access to their managed Grid resources. They have to manage the configuration of their infrastructure and monitor the status of their managed resources on the Grid. The operators should also be able to benchmark and test a set of sites on the Grid. VO managers also belong to this category.

Grid application developers: Grid application developers should be able to develop their applications using the existing application development features of the Eclipse framework. Therefore, they create a standard Eclipse development project for their preferred programming language. Then, they create a g-Eclipse Grid project and declare this g-Eclipse project to depend on their development project. The g-Eclipse plug-ins then offer a variety of possibilities to deliver the application to the Grid and to debug the application, both on the local machine as well as directly on the Grid. Therefore, g-Eclipse acts as an interface between the local IDE (Eclipse) and the remote Grid infrastructure.

The g-Eclipse plug-ins will take these roles into account by offering different project natures and different perspectives which offer the user specific functionality for their role. Both concepts come from and are widely used in the Eclipse framework.

1.2 The g-Eclipse framework

Besides the Grid-related roles described in the last section, the users of the g-Eclipse framework fall into two groups according to the way they use g-Eclipse. It seems probable that the majority of our end-users will use g-Eclipse as a graphical tool within the Eclipse workbench. These users will only get in direct contact with the UI contributions of g-Eclipse that hide the complexity of subjacent core features. Therefore, g-Eclipse has to provide a reliable and intuitive graphical interface that is conformant to the Eclipse user interface guidelines. The second group of end-users may use g-Eclipse as a framework in the sense of an API. They want to create their own applications based on the core functionalities rather than on the UI components. For g-Eclipse, this means that we have to design a sophisticated, but at the same time managable, API.

1.3 Benefitting from the Eclipse framework

Eclipse itself is known to run on several platforms, like Windows, different types of Linux, Sun Solaris, Mac OS X and others. Most of the parts that will be developed in the g-Eclipse project are platform independent and should run on any platform which is supported by Eclipse. The target platforms that will be tested by g-Eclipse will be Windows, Linux, and Mac OS X.

Thanks to the graphical libraries, SWT and JFace, the Eclipse user interface is platform independent and always looks like a native application. This, and the consistent modus operandi, gave Eclipse a good reputation among the users. Most users who will work with g-Eclipse in the future, have some experience in using Eclipse and, therefore, will be able to use the new Eclipse Grid functionality without having to learn a new development environment.

The central point of the Eclipse architecture and framework is its plug-in architecture, a component based software architecture that leads to a clear and modular design. Today most software products are designed to be extensible by mechanisms that are sometimes called plug-ins, but within the Eclipse framework *everything* is a plug-in that can be dynamically added to and removed from the running Eclipse instance.

This is a different approach compared to other well-known software products such as the Firefox web-browser. The Firefox browser consists of a monolithic block that can be enhanced by extensions. The disadvantages of such a monolithic design are apparent when there is a need for a change in the software or the need for more flexibility.

In the Eclipse world, every plug-in amends the functionality of other plug-ins. This is achieved by the underlying OSGi [3] framework that defines the dependencies between the different plug-ins, and how and when additional plug-ins are activated. Also, Eclipse defines the mechanisms of extension-points and extensions. An extension-point is a definition of how to enhance existing functionality. For instance, plug-in A defines an extension-point and the appropriate interfaces that need to be implemented by a third party plug-in. Other plug-ins B and C follow this invitation and extend this extension point with new functionality. This way of building software components leads to an extensible architecture with well-defined interfaces.

This architectural pattern is used throughout the g-Eclipse project. For instance, the framework itself and the functionality on top of it are independent from the underlying Grid middleware. The functionality that is specific to a single type of Grid middleware is added via plug-in mechanisms at runtime. This is reflected in the description of the architecture.

2 The Grid Model

In order to provide a common way of accessing local and remote resources the g-Eclipse framework offers a mechanism for accessing and managing both local elements, i.e. local files and folders, and Grid elements, i.e. either local or remote Grid resources. This abstraction layer is called the Grid Model. It is responsible for providing basic interfaces and classes that may be extended by specific middleware implementations. Moreover it provides standard and abstract implementations for common types of resources, for instance local files or standard containers that may contain other resources. This makes it easy for developers to build their specific implementations upon the Grid model.

The model itself is part of the core plugin and consists of two layers, the abstraction layer and the implementation layer (see fig. 2.1). The abstraction layer is a collection of Java interfaces that define the basic functionalities of the various model elements. The implementation layer itself is subdivided into two subgroups, public implementations that are visible to the outside world and internal implementations that are used by the core plugin to set up the basic model structure. The public implementation layer contains all abstract classes that are available for developers in order to extend the model with middleware-specific functionality. In principle, it would also be possible to extend the functionality by directly implementing the interfaces of the abstraction layer. In practice, developers should always use the classes of the public implementation layer as these classes implement basic mechanisms in order to manage all known types within the Grid model in a common way. Furthermore, the public implementation layer only provides implementations for types that are intended to be externally extended. The abstraction layer itself presents types that are thought to be only used internally by the model. Nevertheless, the public definition of these types within the abstraction layer is necessary in order to allow user interface interaction with these types.

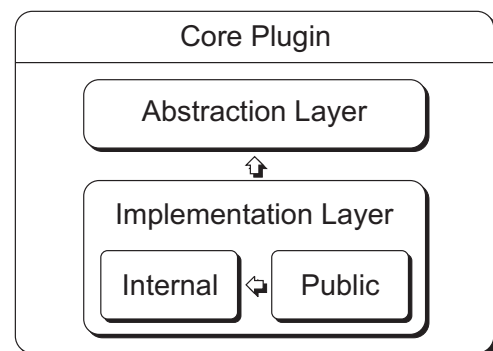


Figure 2.1: Basic layout of the Grid Model.

2.1 Grid element hierarchy

In this section we describe the element hierarchy of the three model layers – the abstraction layer, the public implementation layer and the internal implementation layer.

2.1.1 Abstraction layer

The abstraction layer of the Grid model contains a multitude of Java interfaces that define the basic functionalities of the various grid model elements. Fig. 2.2 shows a simplified outline of the inheritance tree of these interfaces. As this layer only contains interfaces, multiple inheritance is allowed. This is used to map the structure of local and grid elements to the model and to also map the relations between the different elements. Basically, the leaves of this inheritance tree are thought to be implemented either internally by the model or externally by middleware specific implementations. The public implementations will be described in section 2.1.2 whereas the internal implementations will be described in section 2.1.3. This section covers the model itself, i.e. the abstracted elements in the form of the presented Java interfaces. Therefore, a brief description of each element is given in the following list.

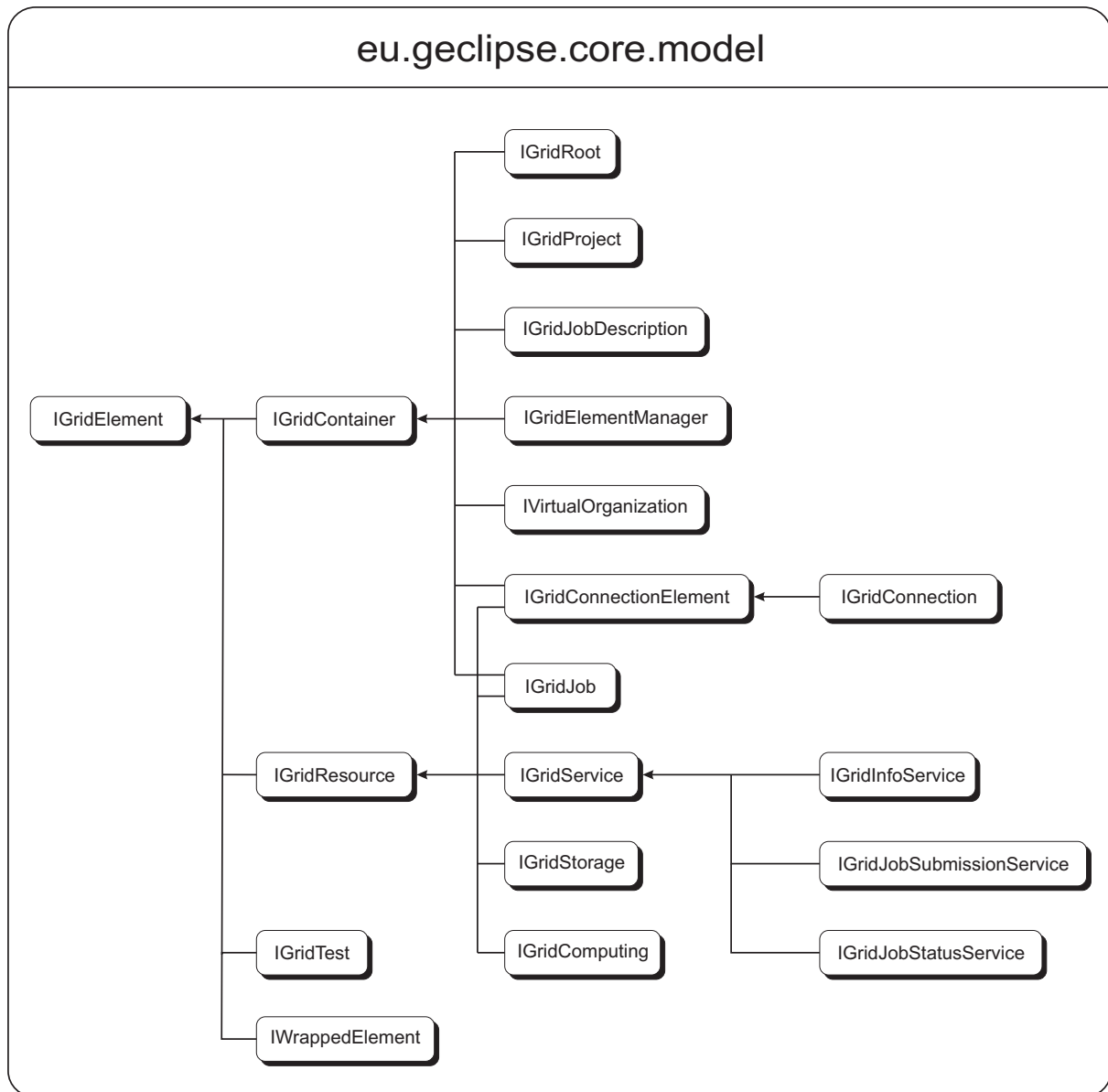


Figure 2.2: Inheritance tree of the Grid model’s abstraction layer for the Grid model elements. The base of all elements is the IGridElement-interface.

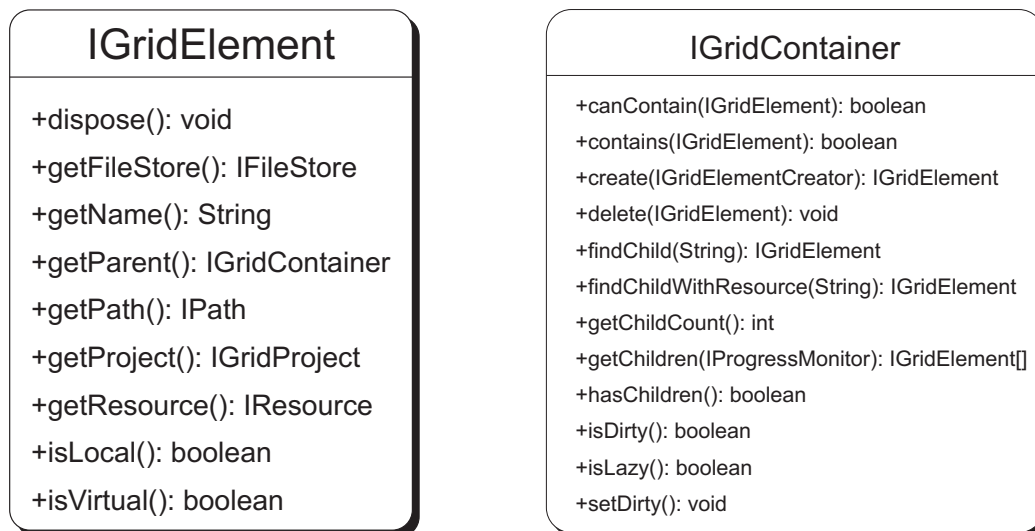


Figure 2.3: The `IGridElement`- and `IGridContainer`-interfaces and their most important methods.

IGridElement: The `IGridElement` interface (see fig. 2.3, left) is the base interface for all elements within the Grid model. It provides methods to access related `IResources` in the Eclipse workspace (`getResource()`) and to determine the location of the element within the model tree (`getPath()`). It also provides methods to access the parent of the element in the tree model (`getParent()`) and to retrieve the project to which this element may belong (`getProject()`). Besides this, every element has a name that can be accessed with the `getName()`-method. This name has to be unique within the `IGridContainer` that is the parent of this element. If an element is destroyed within the model its `dispose()`-method is called in order to clean up that element and potentially existing system resources.

A Grid element may either be local or remote. Remote elements have associations to resources that are not located on the local machine. Whether an element is local or not can be determined with the `isLocal()`-method. Furthermore, a grid element may be either virtual or non-virtual. Non-virtual elements must have an associated `IResource` object that has to be accessible with the `getResource()`-method. This means that the `getResource()`-method of virtual elements may return `null`, while that of non-virtual elements may not. For virtual elements the `getStore()`-method may provide either local or remote resources that are associated with the element. Whether an element is virtual or not can be determined with the `isVirtual()`-method. Both the `isLocal()`- and the `isVirtual()`-methods have to be defined on a type-base. This means that all instances of one type of element have to return the same value for these methods.

IGridContainer: The `IGridContainer` interface is the base interface for all elements that may contain other elements. In that sense grid containers are parents that may have children or may also be empty. These children can be accessed with the `getChildren(IPProgressMonitor)`-method. The number of children, if a container has children, can also be accessed with the `getChildCount()`- and the `hasChildren()`-methods. Whether a container contains a specific element can be tested with the `contains(IGridElement)`-method. Furthermore, the `findChildWithResource(String)`- and `findChild(String)`- methods can be used to determine if a container holds a child with the specified name or holds a child that has an associated `IResource` object with the specified name. New children cannot be added directly. Instead they have to be created with the `create(IGridElementCreator)`-method (see section 2.3). Children may also not be removed directly. Instead they are removed automatically when they are deleted (`delete(IGridElement)`). Deleting an element involves calling its `dispose()`-method.

Containers may either be lazy or non-lazy. Non-lazy containers will automatically load their children when they are created. The list of children of non-lazy containers is guaranteed to be immutable as long as neither `create(IGridElementCreator)` or `delete(IGridElement)` are called. Lazy containers load their children when `getChildren(IProgressMonitor)` is called and when they are marked as dirty, i.e. when `isDirty()` returns true. Lazy containers are intrinsically dirty when they are created. When a lazy and dirty container has loaded its children, it marks itself as not dirty. To explicitly mark it as dirty again, the `setDirty()`-method is used.

Containers may also be specialised to hold a specific type of element. These containers are called specific containers. A specific container has to implement the `canContain(IGridElement)`-method to decide which type of elements it may contain and which type of elements it may not contain. Non-specific containers, i.e. general containers should always return true when their `canContain(IGridElement)`-method is called.

IGridResource: This interface is just a declarative interface that provides no further functionality. Its purpose is to declare an element to be a resource on the Grid in order to allow filtering and decorations of these elements within the user interface.

IGridTest: This interface is the base for all grid tests, both single and structural tests, these last ones being composed of collections of simple tests. It allows querying for all test results as historical data, or for a given tested resource.

IWrappedElement: A wrapped element wraps up another `IGridElement` and delegates all methods to the wrapped element. This is needed in the model tree to present elements in multiple paths since an element itself can intrinsically only be present in a unique path.

IGridRoot: This is the root of the Grid tree within the Eclipse workspace. Every element in the Grid tree is a direct or an indirect descendent of the Grid root. Therefore `IGridRoot` is the only element that has no parent, i.e. its `getParent()`-method returns `null`. Furthermore, the implementation of this interface has to be a singleton.

IGridProject: Grid model element for representing projects in the Grid model tree. These projects do not necessarily have to be Grid projects but may also wrap up other projects in the Eclipse workspace like Java projects or C/C++ projects. `IGridProjects` have to be direct descendents of `IGridRoot`.

IGridJobDescription: Base interface for any type of job description. Job descriptions are usually text files in the workspace from which `IGridJobs` can be created. An example of a job description is a JSDL file.

IGridElementManager: Grid element managers are responsible for managing a certain type of manageable resource. In fact, Grid element managers are nothing other than specific containers. See section 2.4 for a further description.

IVirtualOrganization: Base interface for any kind of virtual organization (VO). These virtual organizations are the key component for each grid project and hold project-specific settings. An example of such a setting is the info service (`IGridInfoService`) that is used to retrieve information about the Grid. Furthermore, VO's contain data that is necessary to authenticate the user on the Grid.

IGridConnectionElement: An `IGridConnectionElement` is an element that represents a file or a folder in a remote location. These elements are managed in a common way with the help of Eclipse's EFS mechanism. Therefore, the `IGridConnectionElement` provides common access to remote elements no matter which protocol is used to access these elements. Examples of underlying protocols would be GridFTP or SRM.

IGridConnection: This is the root element of a connection to the Grid. It represents a folder on the Grid that is accessed with the help of Eclipse's EFS mechanism. The implementation of this interface is a specific container that may only contain `IGridConnectionElements`.

IGridJob: Base interface for any job that is submitted to the Grid. Jobs are created with the help of job descriptions. Once created, a middleware-specific implementation of the `IGridJob` interface is generated from the job description. This `IGridJob` is then used to control and monitor the lifecycle of the job.

IGridService: Each service in the Grid is represented by an implementation of this interface. Examples of such services may be info services (BDII, RGMA...) or job submission services (WMS, WMSPoxy...).

IGridInfoService: Specialisation of the `IGridService` interface for services that provide information about the Grid.

IGridJobSubmissionService: Specialisation of the `IGridService` interface for services that allow submitting jobs to the Grid.

IGridJobStatusService: Specialisation of the `IGridService` interface for services providing job status information.

IGridStorage: Implementations of this interface represent storage elements on the Grid.

IGridComputing: Implementations of this interface represent computing elements on the Grid. Currently an empty marker interface.

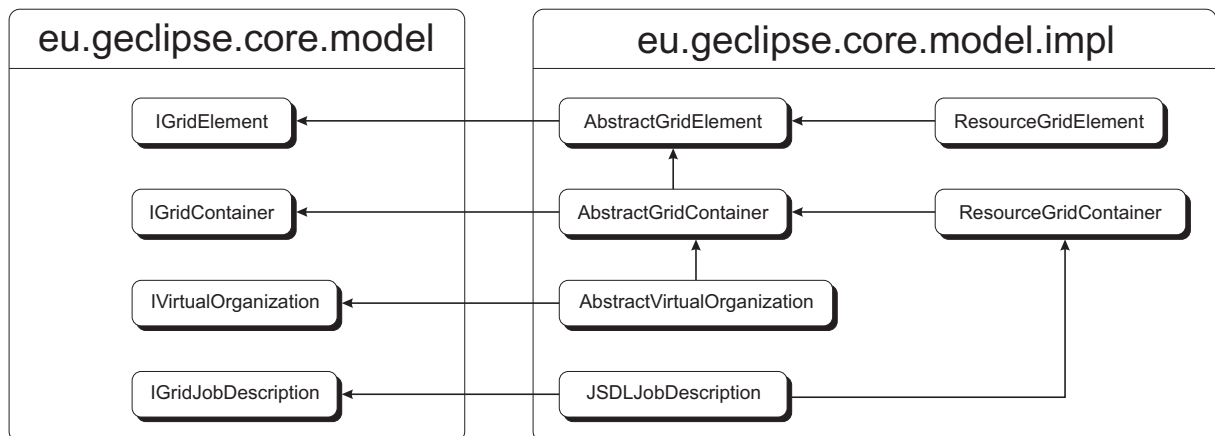


Figure 2.4: Inheritance tree of the Grid model’s public implementation layer for the Grid Model elements. Also shown are the links to the abstraction layer (fig. 2.2).

2.1.2 Public implementation layer

The public implementation layer of the Grid model consists of various classes that may be abstract (see fig. 2.4). All names of abstract classes start with the word “Abstract” in order to distinguish them from the fully implemented classes. The purpose of the classes contained in this layer is to implement basic and common functionality of a particular Grid model element. This way, developers do not have to fully implement their extensions of the model but can make use of these pre-defined functionalities by not directly implementing the interface of the abstraction layer but by extending the appropriate class of the public implementation layer. In fact, developers should make use of these classes rather than implementing their own since the integrity of the Grid model is based on these classes. In the following section a short overview of the classes contained in the public implementation layer will be given.



Figure 2.5: The public implementation layer’s AbstractGridContainer and its most important methods.

AbstractGridElement: Abstract implementation of the IGridElement-interface that provides basic support for the dispose(), getAdapter(Class), getProject() and isVirtual() methods. The dispose()-method is actually an empty implementation. The getAdapter(Class)-method checks whether the specified class is an IResource and if this element is not virtual. In that case, the internal resource of the element as retrieved with getResource() is returned. Otherwise, the request is delegated to PlatformObject#getAdapter(Class).

ResourceGridElement: Full implementation of the `IGridElement`-interface that makes use of the `AbstractGridElement`-class. This implementation provides a local, non-virtual grid element that is associated with an `IResource` in the local workspace. Therefore, it has just one constructor that takes a non-null `IResource`-object as argument.

AbstractGridContainer: Abstract implementation of the `IGridContainer`-interface that supports both lazy and non-lazy containers. All methods are implemented except the `isLazy()`-method which determines whether this is a lazy container or not. When `getChildren(IProgressMonitor)` is called, it is checked whether this container is lazy. If it is lazy and `isDirty()` returns also `true`, the `fetchChildren()`-method is used to lazily load the children of this container. If this method returns successfully, the container is set to be not dirty any more. Furthermore, the `AbstractGridContainer` provides protected methods to safely add and remove children to or from the container. Actually, the `create(IGridElementCreator)`- and `delete(IGridElement)`-methods delegate the work for adding and removing elements to these methods. Internally, a `java.util.List<IGridElement>` is used to manage the children of this container.

ResourceGridContainer: Full implementation of the `IGridContainer`-interface that is based on the `AbstractGridContainer`-class. This implementation provides a local, non-virtual and non-lazy grid container that is associated with an `IResource` in the local workspace. Therefore, it has just one constructor that takes a non-null `IResource`-object as argument.

AbstractVirtualOrganization: This abstract class implements the `IVirtualOrganization`-interface by extending `AbstractGridContainer`. It provides support for the `getInfoService()`- and the `getServices()`-methods. In these methods the children of this VO are queried and searched for instances of the appropriate services.

JSDLJobDescription: As g-Eclipse has decided to implement core support for JSDL and to provide adapters for other job descriptions languages later, the core itself contains this full implementation of the `IGridJobDescription`-interface for JSDL. It is implemented as a subclass of `ResourceGridContainer` as a JSDL is described by a file in the local workspace.

2.1.3 Internal implementation layer

The internal implementation layer contains classes that are to be used internally by the g-Eclipse core components to build up the basic structure of the Grid model (see fig. 2.6). These classes are not exported and can therefore be used only in the `eu.geclipse.core` plug-in. It consists of some fully implemented classes for model elements like `IGridRoot` or `IGridProject` and some other classes that provide support for specialised elements like `LocalFile` or `LocalFolder`. Basically these classes are used to create the model tree from the underlying Eclipse workspace. The only first level element – i.e. the root of this model tree – is the `GridRoot`-class. Second level elements are either the projects that are located in the workspace or `IGridElementManagers` that will be described in a separate section (2.4). Subsequent levels may contain elements of any type except `GridRoot`, `IGridProject`, and `IGridElementManagers`. The following list gives a short overview of the classes that are contained in the internal implementation layer.

GridRoot: This is a singleton implementation of the `IGridRoot`-interface. It is implemented as a `ResourceGridContainer`, i.e. a local and non-virtual container. The associated resource is the `IWorkspaceRoot` of the current Eclipse runtime. The `GridRoot` provides listening mechanisms for clients to track changes in the model, i.e. creation of new elements or deletion of existing elements. In that sense `GridRoot` is the *single-point-of-information* element where listeners can register themselves to keep informed about changes in the model. This is very important for user interface components that depict the model or at least parts of it.

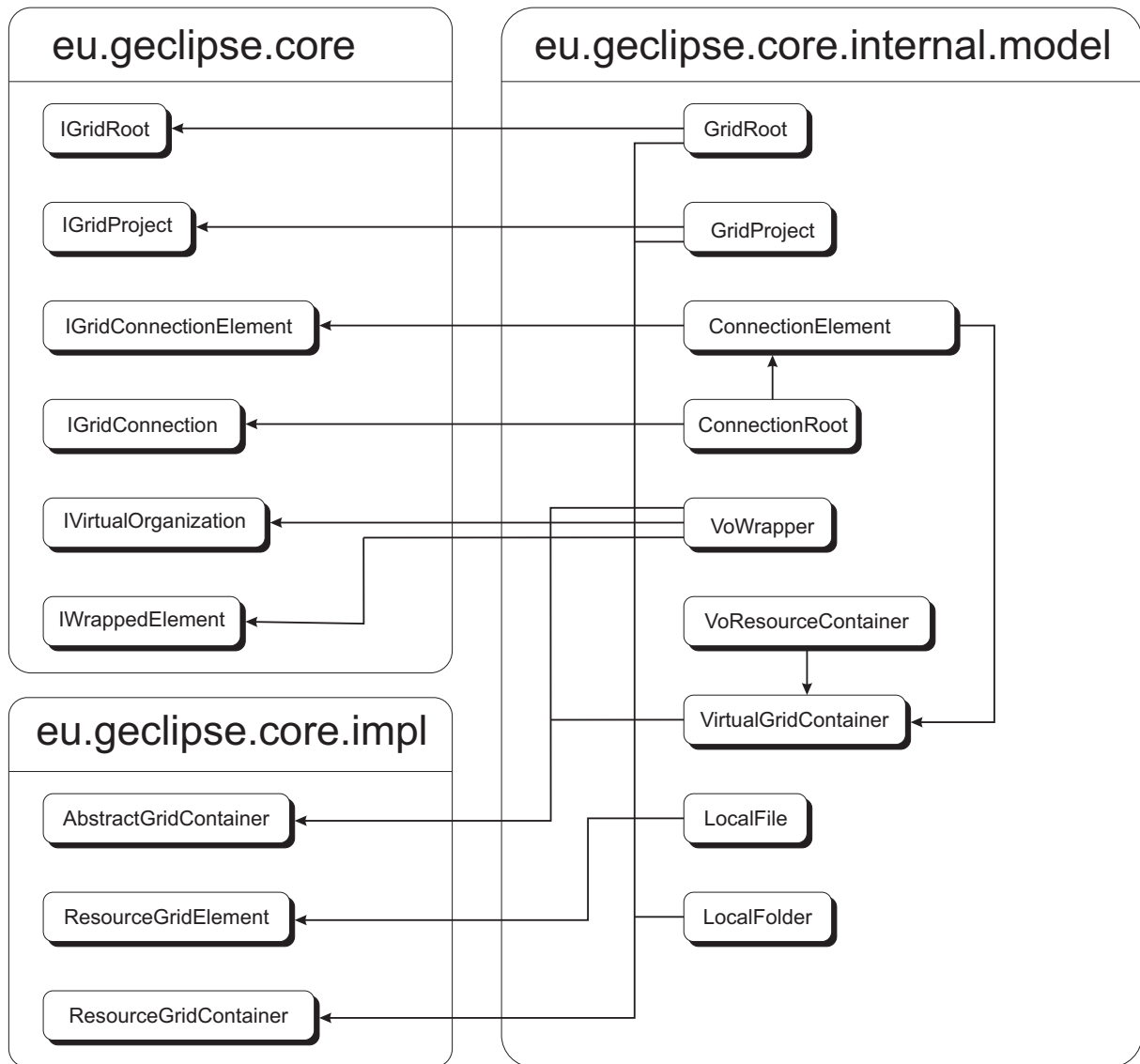


Figure 2.6: Inheritance tree of the Grid model’s internal implementation layer for the Grid model elements. Also shown are the links to the abstraction layer (fig. 2.2) and to the public implementation layer (fig. 2.4).

GridProject: Full implementation of the `IGridProject`-interface that provides support for `IProject`-objects located in the workspace. Every such `IGridProject` represents exactly one `IProject`. The projects do not necessarily have to be Grid projects but may also be of other project types like Java or C/C++ projects. The basic functionality of this class builds up the basic structure of the project if this is a real Grid project. For more details on the Grid project see chapter 2.2.

ConnectionRoot: A connection root is a live connection to a remote destination folder. The connection itself is managed by Eclipse's EFS-mechanism. This implementation of `IGridConnection` builds on this mechanism and uses it to fetch its children and to create new or delete exiting elements, i.e. to do any possible file operation on the remote destinations. With this class, it is possible to plug in remote directories into the local file structure. The user does not see any difference between these remote locations and local ones. Nearly all operations that are available for local resources are made available for remote resources through this class, i.e. resource management, file editing, drag'n'drop operations or just browsing.

ConnectionElement: A `ConnectionElement` is an element that wraps a remote element contained in a `ConnectionRoot`. This may be an element at any level of this connection. In fact the `ConnectionRoot` – i.e. the top level element – itself is a `ConnectionElement`. These connection elements may be files or folders. The operations that are available for an element therefore depend on the type of that element.

VirtualGridContainer: `VirtualGridContainer` is a virtual implementation of the `IGridContainer`-interface. It builds on the lazy-loading mechanism of the `AbstractGridContainer` class.

VoWrapper: Normally `IVirtualOrganizations` are managed by an appropriate Grid element manager, i.e. the `VoManager` (see chapter 2.4). This means that all virtual organizations are children of that particular manager. Hence, in order to include the VO of the current project in the project structure, an `IWrappedElement`-implementation is needed, i.e. the `VoWrapper`.

VoResourceContainer: Virtual organizations provide different Grid resources to the user. These resources are children of the corresponding VO. In order to group these resources in the model tree according to their type, the `VoResourceContainer` is used.

LocalFile: The `LocalFile` class wraps every file contained in the current workspace that is not handled by any other Grid model element. These may be any type of files – for instance, text files or binary data files. An example of a file that is handled by another Grid model element is a JSDL-file.

LocalFolder: Like the `LocalFile` class this class wraps every folder contained in the current workspace that is not handled by any other Grid model element.

2.2 The Grid project

The base of the g-Eclipse workflow within the Eclipse environment is the Grid project. Every time a user wants to work on the Grid, a Grid project has to be created. This project is a direct child of the root element of the Grid model (`IGridRoot`). In the Eclipse workspace Grid projects are marked with the `GridProjectNature`. A Grid project holds all information that is necessary to access the Grid and to manage jobs and resources. It also provides a basic set-up of the user's resources within the local workspace in order to provide an intuitive resource structure that is common for all middleware extensions. An example of such a Grid project can be seen in fig. 2.7. The left hand side of this figure is a screenshot of such a project within the Eclipse workspace. On the right hand side this structure is explained in a more detailed scheme. The names of the resources and also the underlying model classes or at least – for the cases where non-core classes are included – the core-interfaces on which these classes are based on are shown. This complete structure is based on the Grid model as explained in section 2.1.

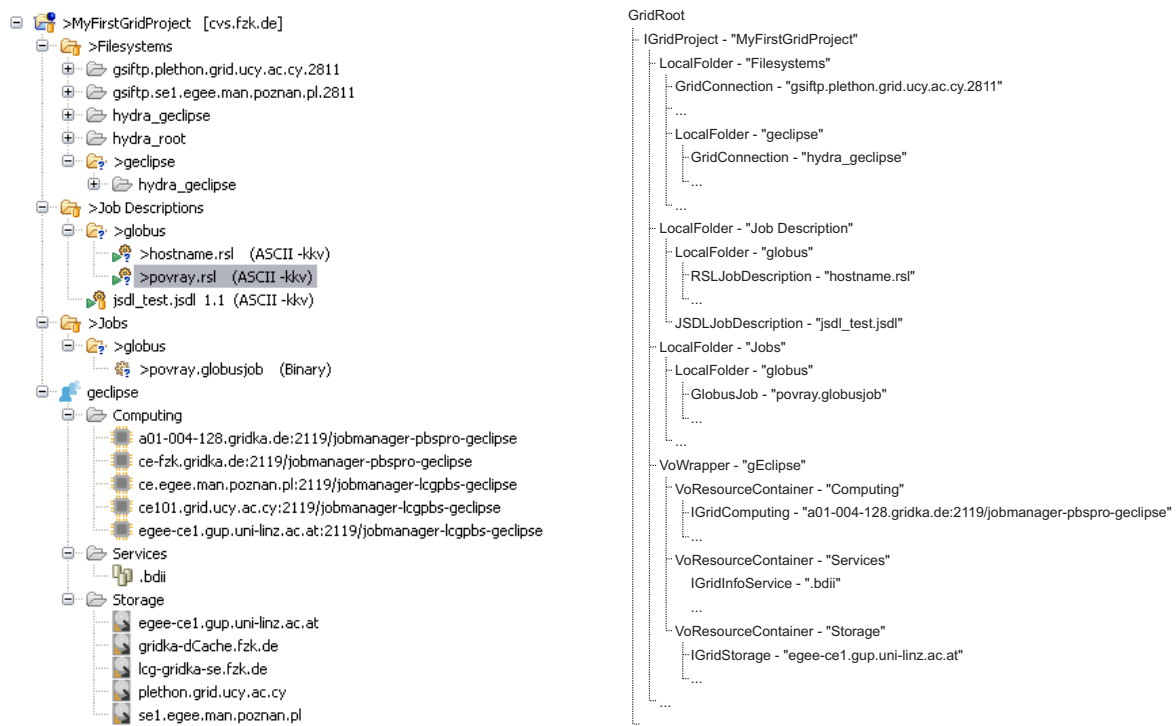


Figure 2.7: Exemplary Grid project structure as screenshot (on the left) and as scheme (on the right). The scheme also shows the `GridRoot`-object that is invisible in the UI-representation.

In g-Eclipse, Grid projects always have to be attached to a virtual organisation, i.e. an instance of the `IVirtualOrganization`-interface. This VO may or may not have the meaning of a virtual organization as is defined within the different Grid middlewares. In fact, this VO defines which middleware is used to manage the resources of a project. It carries all middleware-specific information that is needed in order to retrieve information about the Grid, for instance, a central information service, or to submit jobs, e.g. computing and storage elements. It may also carry authentication and authorization related tokens that grant the user access to the Grid.

Basically, a newly created Grid project presents four standard folders to the user – Connections, Job Descriptions, Jobs and a virtual folder that represents the VO the user is working with. Optionally two other folders may be created via the “New project wizard”, namely Workflows and SiteConfig for storing batch system configurations. Within these folders and also within the project itself, the user is able to create his own structure with local files and folders just like in an ordinary local filesystem. Exceptions from this rule are virtual folders like the VO-folder where no substructure can be created by hand. The following is a short overview of the standard folders of each Grid project.

Connections: This folder is expected to hold `GridConnection`-objects. It is a folder that presents the user with his remotely mounted filesystems and allows for easy access to these filesystems. These filesystems are presented as ordinary folders and may also be browsed as such. The user may, of course, create links to filesystems anywhere in his workspace so the Connections-folder may only be seen as a location where such mounts may be collected. In this folder, user-defined substructures up to an arbitrary level are allowed, in order to give him the ability to group his connections in the way he finds appropriate.

Job Descriptions: This folder is used as a container for all implementations of the `IGridJobDescription`-interface, so for all objects from which one can create running Grid jobs. Just as for the Connections-folder, the user may store his job descriptions anywhere in the project and not only in this folder.

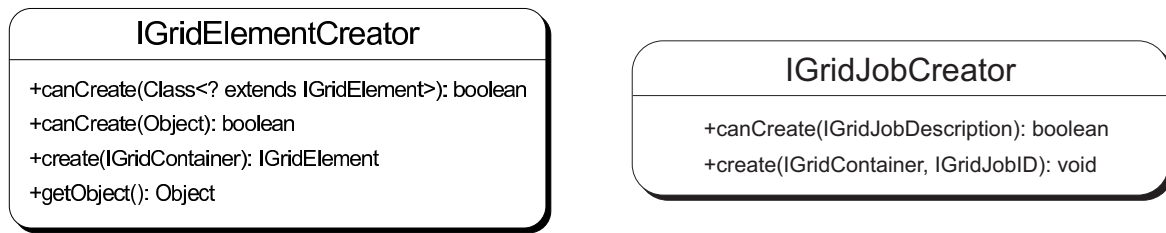


Figure 2.8: Grid element and job creator classes and their most important methods.

Nevertheless, the Grid project offers this folder as a standard location to store these resources. Again, the user is able to create a substructure up to an arbitrary level.

Jobs: All Grid jobs that are created by submitting a job description will show up in this folder or in one of its subfolders. In fact, if a user submits a job description, the resulting job will end up in a subfolder of the Jobs-folder that corresponds to the location of the job description in the Job Description-folder. If a job was created from a job description that is not located in the Job Description-folder, the resulting job will show up in the Job-folder itself. After the job is created, the user is free to change the location of that job (with Drag'n'Drop or with Copy/Paste operations).

Workflows: This folder is defined by the workflow.ui plug-in and can be used to collect workflow definitions, exactly in the same way as the Job Descriptions-folder is used for plain jobs. Workflows submitted for execution end up in the Jobs-folder together with normal jobs.

SiteConfig: Defined by the batch.ui plug-in, this directory is the place to collect batch system description files.

VoWrapper: This is purely a virtual folder that represents the VO with which the project was created. It has an intrinsic substructure that consists of three folders – Computing, Services and Storage. The Computing subfolder holds all computing elements that are available to the user w.r.t. his access rights. Similarly to the Computing folder, the Storage folder holds all storage elements that are accessible by the user. Finally the Services subfolder presents all currently available central services to the user. It is subdivided in subfolders which contain the available information services, the job submission and job status end-points, and all other remaining services.

As the VoWrapper folder is purely virtual, the user will not be able to create a substructure in it. However, this folder not only presents information about the Grid and the user's available resources, but it also allows him to interact with the elements contained within. The user may, for instance, choose to directly mount a storage element in his Connections folder by using the available context menu actions. Foreseen is also the possibility of submitting a job to a specific computing element by just dragging and dropping a job description onto that resource.

2.3 Grid element creators

Grid model elements may not be instantiated directly. Instead, they are created by implementations of a special interface called `IGridElementCreator` (see fig. 2.8, left side). This interface defines four methods:

`canCreate(Class<? extends IGridElement>):` Determines if this element creator is able to create elements of the specified type.

`canCreate(Object):` Determines if this element is able to create a Grid element from the specified object. The argument of this method is stored internally and may afterwards be retrieved with the `getObject()`-method.

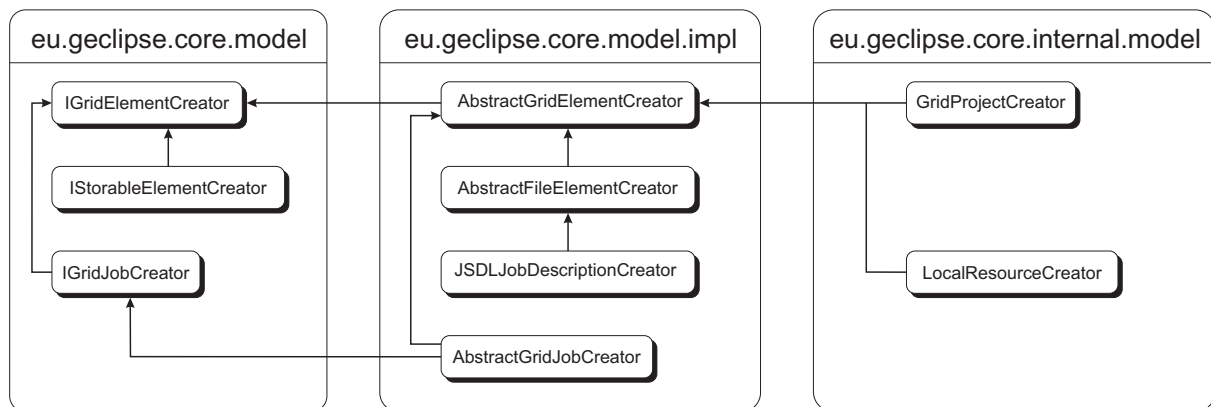


Figure 2.9: Inheritance tree for the element creators. The base interface for all job creators is the `IGridElementCreator`-interface (see fig. 2.8).

`create(IGridContainer)`: Creates a new Grid element as a child of the specified container and returns it. If necessary and possible, the element is created from the object that was formerly passed to the `canCreate(Object)`-method.

`getObject()`: Get the object that was formerly passed to the `canCreate(Object)`-method.

There are two reasons for separating the element creation process from the elements themselves:

1. The Grid model is closed to the outside world with this approach. No element can be created outside the model. This ensures the integrity of the model. Furthermore, it enables validation checks of the elements with respect to their location in the model/project tree.
2. The `IGridElementCreator`-interface is part of the g-Eclipse extension point with the unique id `eu.geclipse.core.gridElementCreator`. This enables extensions, i.e. middleware implementations, to plug-in their implementations of Grid elements with the associated element creators into the framework by using this extension point. To understand this point, one has to follow how the Grid model parses the workspace and its resources and builds the Grid elements from these resources (see section 2.3.2 for a detailed explanation).

2.3.1 Creator hierarchy

Fig. 2.9 shows a simplified outline of the element creator hierarchy as can be found in the g-Eclipse core. Again, the classes and interfaces are split into three groups, the abstraction layer, the public implementation layer and the internal implementation layer. Basically, the fully implemented element creators are split into two groups – internal creators, as can be found in the `eu.geclipse.core.internal.model` package and public creators that may be located anywhere. The internal creators are not registered as extensions of the formerly mentioned extension point. Instead they are directly referenced since they are known to the core. All other creators have to be registered as extensions of that extension point in order to let the core know about their existence.

The abstraction layer of the creator hierarchy contains two further interfaces besides the `IGridElementCreator`-interface itself. The `IStorableElementCreator` is used to create elements from an `IFileStore` while the `IGridJobCreator` is used to create Grid jobs from a job description element (see fig. 2.8, right side). Middleware implementations may implement this interface in order to plug-in their job-submission logic into the g-Eclipse framework. It is remarkable that a special implementation of an `IGridJobCreator` is not limited to a fixed type of job description. This means that a job creator for the `gLite` middleware

may not only take JSDL-descriptions but also JDL- or RSL-descriptions (or any other) as long as either, there is native support for that description, or a translator is available among the descriptions.

The public implementation layer provides abstract implementations of two of the interfaces contained in the abstraction layer. The most interesting implementation is `AbstractFileElementCreator`. This class implements the `canCreate(Object)`-method and tests if the specified object is an `IFile` instance. If this is the case it delegates the decision if an element can be created to an abstract method by passing the file extension to that method. So subclasses of this implementation may decide from the file extension if they are able to create an element from the specified file. An example for that is the `JSDLJobDescriptionCreator` that tests if the extension is that of a JSDL-file (“`.jsdl`”).

The internal layer contains two classes that are used internally for creating the basic elements of a Grid model, i.e. a project (`IGridProject`) and any local resource (`LocalFile` or `LocalFolder`). The `GridProjectCreator` is able to create a `GridProject` from a specified `IProject` instance. The class `LocalResourceCreator` creates appropriate Grid elements from all resources that are not handled by any other creator as long as their names do not begin with a period since these files are defined as hidden or special within the Grid model.

2.3.2 Model creation process

Generally the Grid model monitors the state of the Eclipse workspace. This is done by registering the model as an `IResourceChangeListener` to the Eclipse `ResourcesPlugin`'s workspace. To be more precise, the `GridRoot` element registers itself as such a listener. After that, all changes in the workspace are tracked, i.e. new files appear, existing files moved or deleted, directories created and so forth. If such a change appears, the corresponding element in the model will also be removed or deleted. If a new resource appears in the workspace, the `GridRoot` searches for an `IGridElementCreator` that is able to create a Grid element from this resource. Therefore, it fetches a list of all registered element creators with the help of the corresponding extension point and asks these creators if they are able to create an element from the new resource. Therefore, the `canCreate(Object)`-method is called with this resource. If a creator declares itself as responsible for creating an element from the resource by returning `true`; this creator is used for element creation by passing it to the `create(IGridElementCreator)`-method of the `IGridContainer`, that is the parent of the new element. The container then takes the responsibility of the element creation process. If no creator declares itself as responsible for creating the element, the Grid root searches the internal creators for the creation of the element. This means that public creators take precedence over the internal ones.

Here are two examples of such a creation process:

- The user creates a new file called “MyJob.jsdl” in the “Job Description” folder.
 1. The model gets informed that there is a new `IFile` object in the workspace.
 2. The `GridRoot` fetches the list of all registered public element creators and asks every creator if it is able to create a related Grid element.
 3. The `JSDLJobDescriptionCreator` returns `true` when asked if it is able to create an element.
 4. `GridRoot` passes the `JSDLJobDescriptionCreator` to the `create(IGridElementCreator)`-method of the “Job Description” grid container.
 5. The container creates the new element by calling the `create(IGridContainer)`-method of the `JSDLJobDescriptionCreator` and adds the newly created element to its list of children.
 6. The container sends an event that a new child was added to it in order to inform Grid model listeners about this change.
- The user creates a new folder named “MyJobs” in the “Jobs” folder.
 1. The model gets informed that there is a new `IFolder` object in the workspace.

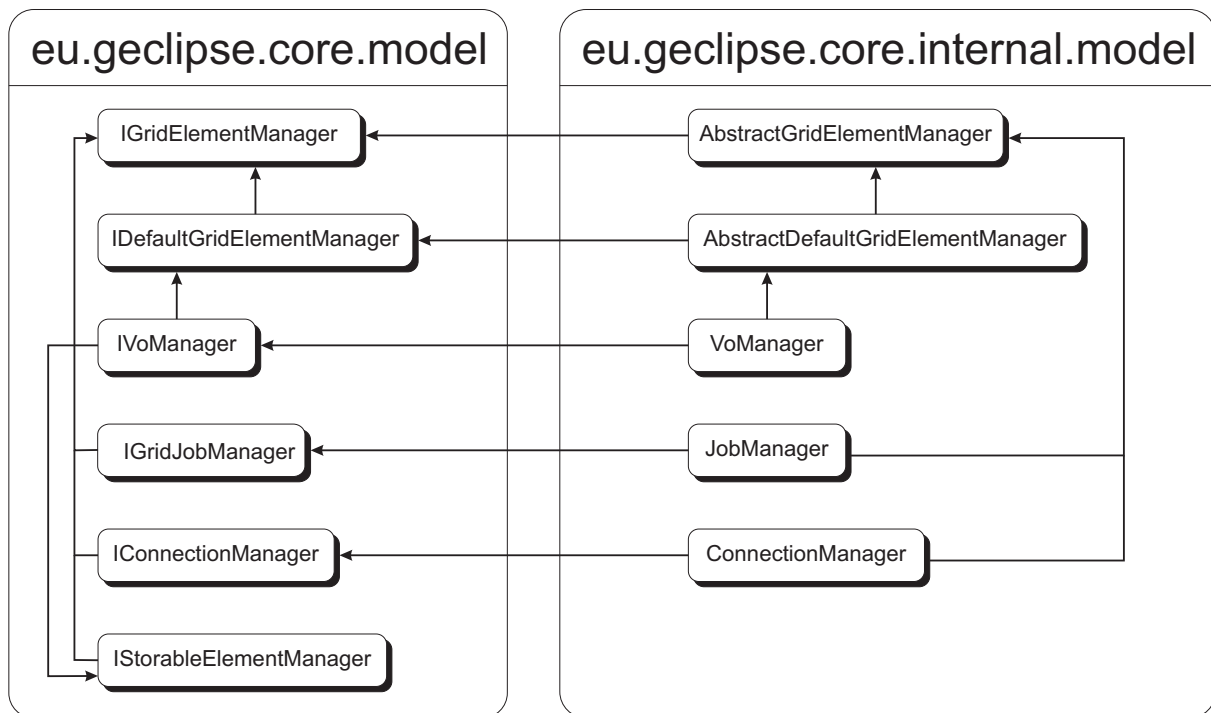


Figure 2.10: Inheritance tree of the Grid element managers. The base interface for all element managers is the `IGridElementManager`-interface (see fig. 2.11).

2. The `GridRoot` fetches the list of all registered public element creators and asks every creator if it is able to create a related Grid element.
3. None of the public element creators return true when asked if it is able to create an element from the specified `IFolder`.
4. The `GridRoot` asks the internal element creators if they are able to create a related Grid element.
5. The `LocalResourceCreator` returns `true` when asked if it is able to create an element.
6. The rest of the process is the same as in the previous example starting at step 4.

2.4 Grid element managers

The Grid model contains several elements that need to be managed in a central place in order to allow easy access to these elements. In particular, these elements are Grid jobs, Grid connections and virtual organisations. The corresponding model interfaces thereby inherit from the `IManegeable`-interface. This interface defines just one method – `getManager()` – that is used to refer to the central Grid element manager that is responsible for the management of elements of a particular type. According to this one-to-one relation there are three Grid element managers for the three element types that need to be managed – the `JobManager`, the `ConnectionManager` and the `VoManager` that are all implemented as singletons. Fig. 2.10 shows an outline of the inheritance tree of these element managers.

The base interface of the Grid element managers is the `IGridElementManager`-interface (see fig. 2.11, left) that defines three methods – one to add an element to that manager, one to remove an element from that manager and one to ask the manager if it is able to manage a certain element. Since these element managers contain the managed elements as children they inherit from `IGridContainer`. This



Figure 2.11: The Grid element manager and the default Grid element manager classes and their most important methods in detail.

indicates that a managed element may be both a child of an ordinary container in the project tree and at the same time a child of its dedicated manager. The difference is that the ordinary container will be the parent of that element and will therefore determine the path of that element within the project tree. The manager therefore is only a central repository that holds references to all elements of a special type that are spread all over the project tree. Whenever a new element is created the model tests if this element implements `IManageable` and if so, registers that element with the `addElement(IGridElement)`-method of the `IGridElementManager` that is returned by the managed element.

In addition to an ordinary element manager the `IGridDefaultElementManager` (see fig.2.11, right) supports the definition of a default element, i.e. an element that should be taken as default for a certain operation. This is of interest for virtual organisations where the user may want to specify numerous VO's with different access properties in order to have the possibility to access the Grid in different ways. The default VO is taken whenever no other VO is chosen explicitly.

The core itself defines the dedicated managers for all three managed elements, i.e. the `IGridJobManager`, `IConnectionManager` and `IVoManager`. The implementation of these interfaces is internal and is therefore located in the `eu.geclipse.core.internal.model` package. The corresponding classes are implemented as singletons in order to have exactly one manager for each managed element type. These managers are used from the UI-plugin in order to implement views and property pages to manage the elements (see chapter 5). An example is the job view that presents all currently available jobs in a list and gives the user the possibility to see the status of these jobs and to operate on them.

3 Authentication and Authorisation

Authentication and Authorisation (AA) are major issues when dealing with Grids. As soon as one interacts with Grids one has to authenticate himself to Grid services in order to get access to any resource. The specific resources that may be accessible by a Grid user are determined by authorisation mechanisms. On the other hand, the user has to be able to trust the servers where he stores his data or process his jobs. This implies that the servers in the Grid have to authenticate themselves to the user, to avoid fake server which could try to gather information from the users. Fig. 3.1 shows a very rough sketch of this mechanism.

The various AA mechanisms are most often based on X.509-certificates. Nevertheless, there exist several different authentication mechanisms and X.509-extensions among the different middlewares. Therefore g-Eclipse has to provide an abstraction mechanism for all these AA implementations. This abstraction is divided into server-side AA (ie, authenticating to the server the user is connecting to) and client-side AA (ie, trusting the service). The server-side AA mechanism deals with so called authentication tokens and will be explained in detail in section 3.1. The base of the client-side AA within our framework are the so called CA¹-certificates that will be discussed in section 3.2. The main difference between the two is that authentication tokens have to be generated at the client side, so g-Eclipse has to provide methods for the creation and validation of these tokens, whereas CA-certificates are used to authenticy remote servers on the client side.

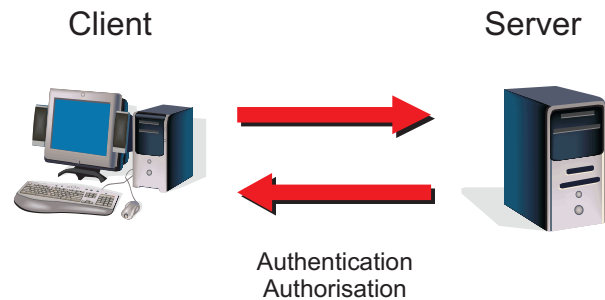


Figure 3.1: Authentication and Authorisation in a Grid.

3.1 Server-side authentication

Authentication tokens are the base for every operation on the Grid. They ensure the authentication of a Grid user and they are also the basis used by the services to manage the authorisation of this user, i.e. which resources the user may have access rights to. The most common method of authentication in the Grid uses X.509 certificates – issued by a Certificate Authority – but also other methods are possible (e.g. the ssh protocol with username and password). Since authentication tokens can have a limited lifetime, the user would want to activate these tokens at regular intervals. Therefore, g-Eclipse has to provide an easy way to generate and manage these tokens in a semi-automated way. This means that on the one hand, the user has to be able to have full control over the lifecycle of his authentication tokens, while on the other, g-Eclipse has to be aware of situations where a valid authentication token is needed – such as when submitting a job – but no (or only expired) tokens were found.

3.1.1 Authentication tokens

The base of g-Eclipse’s server-side authentication are the formerly mentioned authentication tokens that are based on the `IAuthenticationToken`-interface (see fig. 3.2, left). This interface defines methods for validating and activating such tokens. The validation process of such tokens normally includes local checks to see if all necessary information is available and is valid to activate a token. In the case of an X.509-token, this includes tests to check if the specified certificate- and key-files exist and if the

¹Certificate Authority

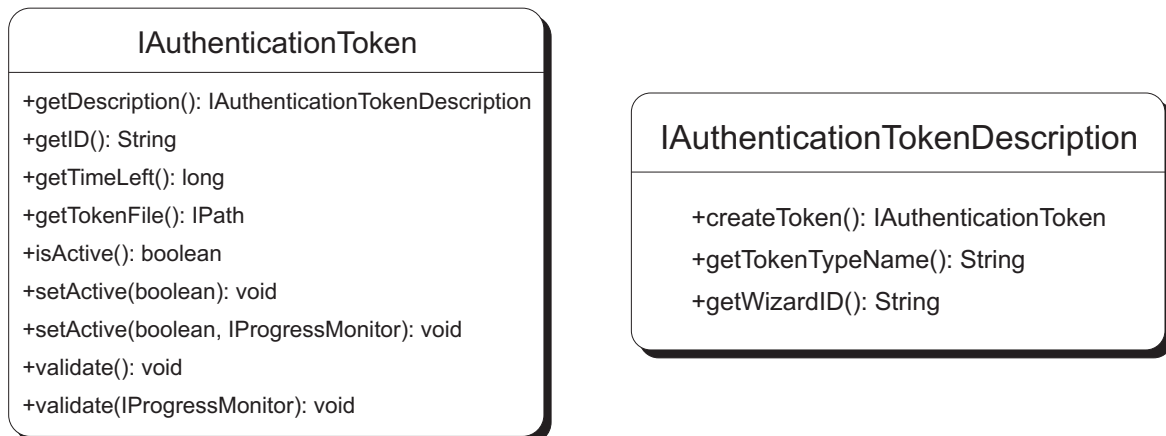


Figure 3.2: The `IAuthenticationToken`- and `IAuthenticationTokenDescription`-interfaces that are the base of g-Eclipse’s authentication and authorisation mechanism.

passphrase specified for that key-file is valid. The activation process creates a so-called “credential” – which is a middleware-specific AA-token wrapped up by the g-Eclipse authentication tokens – from the specified parameters and also provides a local copy of this credential in a file on disk. This file is not kept between different sessions and the access rights are set in order to prevent this file from being read by third parties.

Authentication tokens may not be created directly. Instead they are created by using an instance of an `IAuthenticationTokenDescription` (see fig. 3.2, right). These descriptions are registered in the `eu.geclipse.core.authTokens` extension point. This gives middleware specific implementations the possibility to plug-in their AA-mechanisms into the g-Eclipse framework. A token description provides a method to create tokens. Therefore the description holds all necessary information to create a token, such as certificate and key files, and the desired lifetime of the token. It also provides two other methods – `getTokenTypeName()` and `getWizardID()` – to present the token in the UI and to allow the creation of authentication tokens with the help of a dedicated wizard. The core itself offers an abstract implementation of the `IAuthenticationToken`-interface called `AbstractAuthenticationToken`.

3.1.2 Token management

Similarly to managed elements, authentication tokens are also managed by a dedicated manager class called the `AuthenticationTokenManager` (see fig. 3.3). Like the element managers this class is implemented as a singleton. The manager itself holds an internal list of all currently available authentication tokens. There are methods to create new tokens, to destroy existing tokens and to query the list of currently available tokens. Similar to a default grid element manager the token manager allows the definition of a default token that is used to authenticate the user in the Grid whenever no other token is chosen explicitly. Furthermore, there exists a listener mechanism. This means that the manager itself acts as a notifier and informs all registered listeners about changes

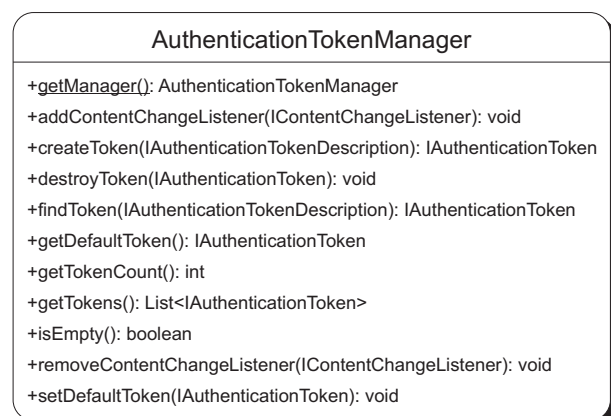


Figure 3.3: The authentication token manager and its most important methods.

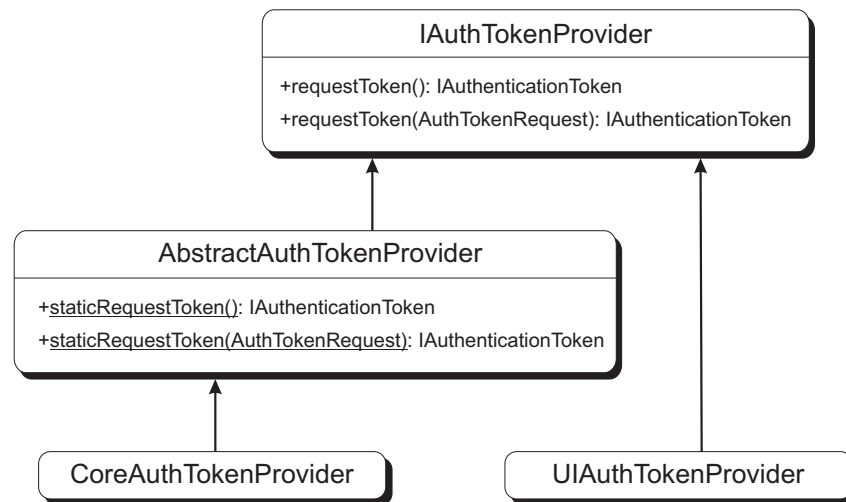


Figure 3.4: Inheritance tree for authentication token providers.

in the list of available authentication tokens, i.e. each time a new token is created or an existing token destroyed.

The reason for such an authentication token manager is to have a central repository of all authentication tokens. This ensures that every time a token is needed to access the Grid, the user need not create a new token. It also facilitates the management of tokens of different types for different middlewares (see chapter 3.1.3 for details).

3.1.3 Token providers

In order to get an authentication token for accessing the Grid, neither is the token created directly, nor is the token manager asked to create a token. Instead, g-Eclipse provides a mechanism using token providers to retrieve either any type of token or a token of a specified type. An outline of this mechanism is shown in fig. 3.4. The base of this mechanism is the `IAuthDataProvider`-interface that is part of the `eu.geclipse.core.authDataProvider` extension point. There are actually two implementations that make use of this extension point, `CoreAuthDataProvider` that is used if the UI is not available and `UIAuthDataProvider` that is used in any other case.

The `IAuthDataProvider`-interface defines two methods for retrieving a token. The first method takes no argument and returns a token of an unspecified type. The second method takes an instance of `AuthTokenRequest` as argument and returns a token that matches the specified request. In the first case the provider asks the token manager for a default token. If no such token is defined it means that the token manager is empty and a new token is created, set as default and returned. In the second case the token provider first looks up the default token, checks if this matches the token description (`IAuthenticationTokenDescription` instance) specified in the request and returns it if so. If it does not match the description a new token will be created, if necessary by opening a dialog where the user is prompted for the necessary data. This dialog makes use of the additional parameters provided by the `AuthTokenRequest`, namely the requester and the purpose of the request, to inform the user which component needs the token and why it is required. The newly created token is then set as default and returned.

All this functionality is implemented in the `AbstractAuthDataProvider`-class. Furthermore this class specifies two static methods that match the non-static methods in order to provide static access to authentication tokens. These static methods query the list of registered token providers and ask the provider with the highest priority for a token. The priority of a token provider is thereby part of the definition of

the corresponding extension point. The priority of the `UIAuthTokenProvider` is thereby higher than the priority of the `CoreAuthTokenProvider`. This means that whenever the UI is available the corresponding token provider will request the token. This has the advantage that the `UIAuthTokenProvider` may make use of UI-components to ask the user about additional parameters for its token. In that sense it is guaranteed that the token provider will return a token (if the user does not cancel the operation). If the UI is not available, the request-methods may also return false since there is no way to interact with the user and to ask him about the properties of his token.

3.2 Client-side authentication

When a user connects to a service in the Grid the corresponding server sends its authentication token to the user. This authentication token has then to be validated by the user. This is done by so called CA-certificates that have to be available on the client-side. The g-Eclipse framework supports this mechanism by providing these certificates and methods to install these certificates either from a local directory or from a remote location.

3.2.1 CA-certificates

CA-certificates are represented by implementations of g-Eclipse's `ICaCertificate`-interface. This interface is very simple and defines just two methods, `getCaHash()` that returns a unique hash code for a CA-certificate and `getID()` that returns a string that can be used to uniquely identify the certificate and to present this certificate in the UI. These certificates normally do not contain any functionality but are stored in a central repository in order to allow middleware specific implementations to read them. The implementations themselves have to handle authentication-issues concerning remote services.

3.2.2 The CA-certificate manager

The CA-certificate manager is the central repository for all currently loaded CA-certificates. It is implemented as a singleton in `eu.geclipse.core.auth` and is called `CaCertManager`. Fig. 3.5 shows an outline of this class and its most important methods. Besides the methods for querying the list of managed certificates and for deleting a certificate the manager basically defines two methods for importing certificates. The first is the `importFromDirectory(File, IProgressMonitor)` method that imports CA-certificates from a local folder into the g-Eclipse framework. The second and more interesting way to import certificates is `importFromRepository(. . .)`. With this method the user is able to download a list of certificates from a remote repository and store the certificates contained in this list in the g-Eclipse framework. After importing certificates in any of these ways, the certificates are available within the workspace. They can either be accessed by the access methods in the manager or by the path to the local directory where they are stored. This path can be obtained with the `getCaCertLocation()`-method. Several middleware implementations need this path to be set as an environment variable or as a Java property.

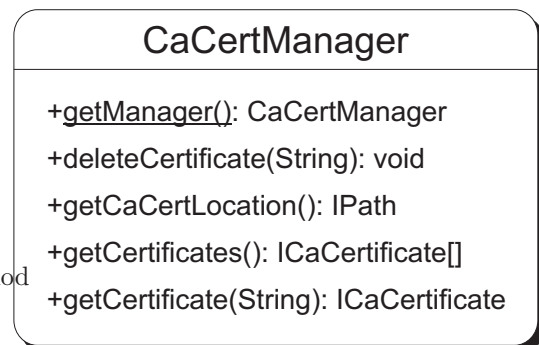


Figure 3.5: The CA-certificate manager and its most important methods.

3.3 Managing passwords

The g-Eclipse framework provides a way to securely store passwords during runtime. These passwords are not stored on disk but are only stored in memory during execution time. The central class for managing user passwords is the so called `PasswordManager` (see fig. 3.6). It provides three methods for managing passwords. One is used to register a specific password with a unique ID, another is used to retrieve a stored password using its unique ID, and the last one is used to erase a formerly stored password from the manager. In this way, it is possible to store context based passwords by giving them a corresponding unique ID. This may, for instance, be the full path of a key file.

PasswordManager

```
+erasePassword(String): void  
+getPassword(String): String  
+registerPassword(String, String): void
```

Figure 3.6: The password manager and its most important methods.

4 Error reporting

As g-Eclipse aims to be an easy to use framework for the Grid community and since error reporting in different middlewares is implemented in different ways and qualities, it is essential to provide the user with useful error message and to present the user with various solutions for the different errors. Parts of these error reporting techniques are already integrated in the Eclipse framework. Therefore, we will give a short introduction to those techniques in section 4.1. The extensions to this API within the g-Eclipse framework will afterwards be presented in section 4.2. Basically these extensions allow not only to specify error messages and (like is common in Java) stacktraces of the portion of code that causes the error, but also to provide hints and solutions to the user how these errors can be avoided in the future.

This error reporting system was rewritten during months 17-18 focusing in generality and simplicity, by making use of extension point mechanisms, with the aim of having it included in the upstream codebase. Furthermore these classes live now in its own separate plugin, to allow its distribution to other Eclipse projects.

4.1 The Eclipse way of reporting runtime errors

The base of Eclipse's error reporting mechanism consists of two elements, the `CoreException`-class and the `IStatus`-interface. The `CoreException` is implemented as an ordinary Java-exception and provides only one Constructor that takes an `IStatus`-object as argument. Therefore all the information about the error is contained in this `IStatus`-object (see fig. 4.1). If the error was caused by an exception, this exception is also contained in this object like a message that describes the error. `IStatus` also defines two error values, one is called the severity and the other is just called the error code. The severity has one of the values `INFO`, `WARNING`, `ERROR`, `CANCEL` or `OK`. The error code may be a plugin-specific value that classifies the error or may also be one of the formerly mentioned values. Furthermore the `getPlugin()`-method of `IStatus` gives the name of the plugin that reported the error.

If a `CoreException` is caught, the related status object may be used to directly generate a log message. This log message is written to a log file. Furthermore, the PDE¹ runtime-plugin offers a view called "Error Log" that watches this logfile and shows a graphical representation of this file. With this view, the user is able to browse all messages and errors and to show more detailed information about these items.

In addition to the error view Eclipse also provides an error dialog that takes an `IStatus`-object as argument. This dialog shows all relevant information contained in the status object, such as the error message. Furthermore, it contains an expandable area that shows the stacktrace of an exception that may be contained in the status object.

4.2 Extension of the error reporting mechanism within g-Eclipse

The g-Eclipse error reporting mechanism is based on the Eclipse error reporting mechanism. Therefore, g-Eclipse defines an exception class called `ProblemException` that is a direct subclass of `CoreException`.

¹Plugin Development Environment

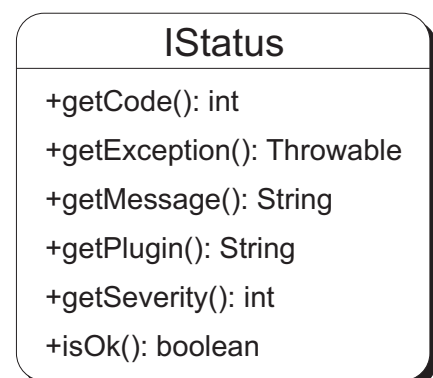


Figure 4.1: Eclipse's `IStatus`-interface and its most important methods.

`ProblemExceptions` may not only be instantiated as `CoreException` –ie, with a `IStatus` object as argument–, but also with a reference to a problem –either a `IProblem` object or simply a predefined problem ID. Each such problem may hold an arbitrary number of so-called ‘solutions’.

In that sense, a problem describes an issue that occurred during runtime and may also hold solutions that should support the user in finding ways to avoid that problem. An example would be a connection problem when trying to connect to the Grid. Probable solutions for that problem would range from “Check your internet connection” to “Check your access rights/authentication tokens”. In the following chapters we will present a brief description of the problems and the solutions.

4.2.1 Problems

Runtime errors in g-Eclipse are no longer called errors, but problems. Within the framework they are represented by the `IProblem`-interface. This interface defines several methods to retrieve further information such as a short description of the problem, an email address for reporting the problem, or a list of reasons which might have caused it. To be compatible with the Eclipse framework it also provides a method for generating an `IStatus`-object from a problem. Furthermore, a problem may be caused by an exception that may be retrieved by the corresponding getter-method. The most important improvement of the problem mechanism is the ability to link solutions to a problem. With this the developer is able to give the user hints on how to deal with a specific problem. These solutions may also be active in the sense that they may trigger some action to directly solve the problem. More details about solutions will be given in section 4.2.2.

An important feature of the error reporting mechanism is its use of Eclipse’s extension points to provide a shared “repository” of problems and solutions which can be reused and extended by different plug-ins. Each plug-in should declaratively define its own problems and solutions by registering them in the `eu.geclipse.core.reporting.problemReporting` extension point. This extension point also allows to bind solutions to problems, even if these come from different plug-ins. The corresponding `IProblem` objects can then be retrieved with the `ReportingPlugin.getReportingService().getProblem(problemID, ...)` method by providing the problem’s ID. Eventually, also the problem’s default description may be overwritten to better inform the user of the causes.

The outlined approach has several advantages:

- Problems are reusable since they are globally accessible by their problem ID.
- Plugins may specify problems of a general character that may afterwards be used from all other dependent plugins. For instance the core plugin specifies general problems and their related solutions for connection timeouts or job submission failures. To simplify the access to these problems, their IDs are defined in a public interface named `ICoreProblems`.
- Each time a certain problem occurs –even if it occurs in different plugins– it is ensured that it always looks the same, i.e. the error message is always the same and the related solutions are always the same.
- Internationalisation issues are not spread all over the framework but are concentrated in the plugin’s manifest file `plugin.xml`.

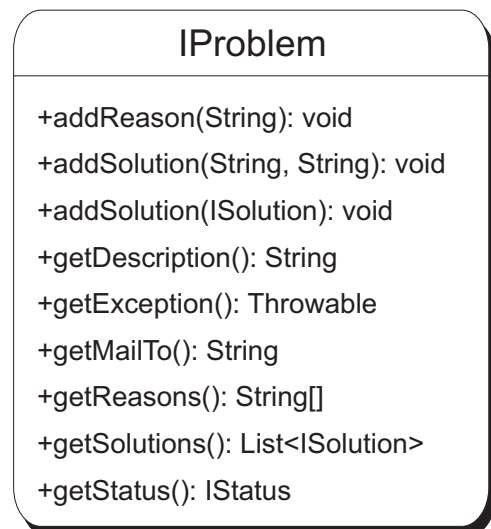


Figure 4.2: The `IProblem`-interface and its most important methods.

`IProblem`-objects can also be instantiated programmatically “on the fly” via the `ReportingPlugin.getReportingService()` call, although this should be reserved for rare situations because it negates the advantages of the declaratively defined problems.

4.2.2 Solutions

Solutions are defined as implementations of the `ISolution`-interface (see fig. 4.3). Each solution again has a its own solution ID in order to be reusable among different problems. This approach takes into account the fact that several problems may share the same solution. Furthermore each solution contains a text that describes it. A solution may be active in the sense that it may trigger an action in the UI that leads the user to the solutions of his problem. If a solution is passive it is represented as plain text to the user. If a solution is active it has to implement the `solve()`-method in order to trigger the action that solves the problem or in order to trigger an action that helps the user find the point in the UI where the solution may be located (for instance, a preference in the preference pages or a setting in the project).

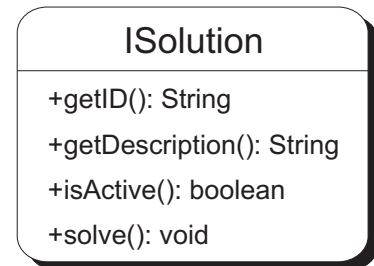


Figure 4.3: The `ISolution`-interface and its most important methods.

Similarly to the problems, solutions can be instantiated programmatically (discouraged) or defined in the central register via the same extension point `eu.geclipse.core.reporting.problemReporting`. If defined in the registry, `ISolution` objects may be retrieved with `ReportingPlugin.getReportingService().getSolution(solutionID, ...)`.

At runtime it is possible to add new solutions to a problem by calling one of the `addSolution(...)` methods. This allows for example to extend to list of suggestions presented to the user if more details of the issue are know at the current execution point. More details about the UI parts of g-Eclipse’s error reporting mechanism can be found in section 5.4.

5 User interface

The user interface (UI) plays a major role within the g-Eclipse framework. Its main task is to provide a set of common control elements for all types of middleware. Therefore, the user interface may only depend on the abstraction layers of the g-Eclipse core plugin but not on any other plugin or any concrete implementations. In this way, it is guaranteed that if a middleware implements the abstraction layer in a proper way, it will automatically be integrated in the g-Eclipse user interface.

The UI itself consists of several control elements. The most important are the so-called Grid model views that show a certain representation of the Grid model. Since these model views share a multitude of functionalities, it is convenient to have an abstraction layer for them. These views and the associated abstraction layer will be discussed in section 5.1. The g-Eclipse framework also makes use of another concept in Eclipse called ‘perspectives’. The g-Eclipse related perspectives will be discussed in section 5.2. Another important class of UI elements are the wizards that help the user perform complicated tasks in an easy way (see section 5.4). In addition to the g-Eclipse error reporting system that was presented in section 4.2, we will discuss the UI part of this mechanism in section 5.4.

5.1 Grid model views

Grid model views are the central UI elements for accessing the g-Eclipse core functionalities. A view itself is a rectangular area within the Eclipse workbench that provides views to present data and information to the user and control elements such as buttons and context menus to allow the user to manipulate that data (like copying and deleting). All Grid model views represent certain portions of the Grid model (see chapter 2) and provide functionality to manipulate the model. This implies that these views share a lot of functionalities. Therefore, Grid model views are based on a common abstraction layer that will be described in this section. Later, an overview of the individual views will be given.

5.1.1 Abstraction layer

Fig. 5.1 shows an outline of the inheritance tree of the Grid model views. This inheritance tree contains three abstract classes and three concrete implementations. The concrete implementations will be discussed in the subsequent sections. The abstraction layer itself is based on the `GridModelViewPart`-class. This class extends Eclipse’s `org.eclipse.ui.part.ViewPart`-class that is the base of all views in the Eclipse framework. The `GridModelViewPart` itself consists of a so-called “structured viewer” as a central UI element. Such a structured viewer may be a list, a table or a tree. At this level of abstraction, it is not specified what kind of viewer may be used to represent the model. Furthermore, such a structured viewer needs to have an input element from which it can create its content. As we are talking about Grid model views, this input element has to be an instance of `IGridElement`. Therefore, `GridModelViewPart` defines two abstract methods, `createViewer(Composite)` and `getRootElement()`. Subclasses have to implement these methods. The first method therefore has to create a specialised structured viewer that is used to present the model. This may be any of list, table or tree. The second method has to specify the root element within the g-Eclipse model tree that will be set as input data for the viewer.

The way the input data will be parsed and rendered in a view is defined by implementations of two Eclipse interfaces, `IContentProvider` that is responsible for parsing the data and `IBaseLabelProvider` that is responsible for rendering the items in the viewer. The g-Eclipse UI plugin therefore specifies its own implementations of these interfaces in order to interact with the structured viewer to present the Grid model to the user. These providers are adapted to the Grid model and provide functionality for decorating items with reference to their current state (such as a Grid job that may be in state scheduled,

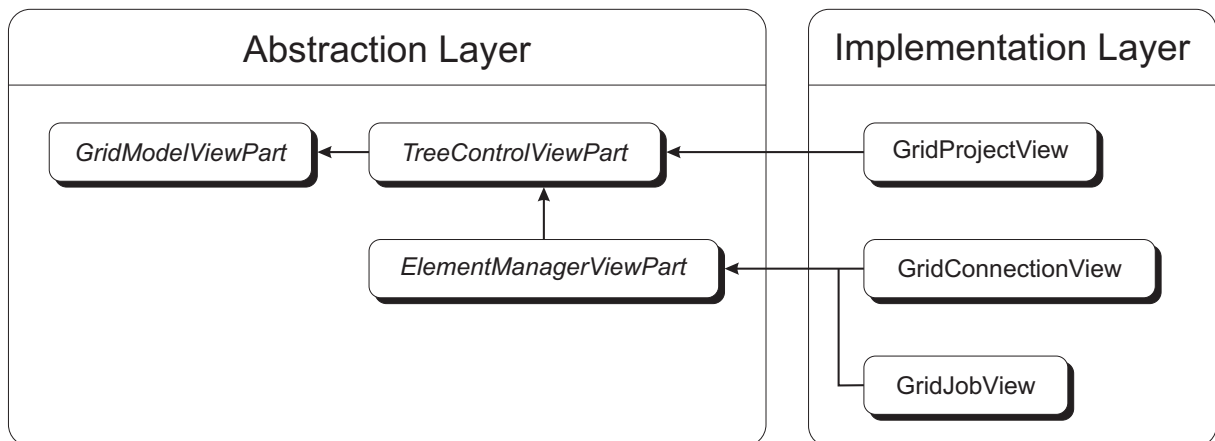


Figure 5.1: The inheritance tree of the Grid model views.

running, ...) or to support lazy loading of grid containers by presenting progress bars showing the progress of the loading operation.

After the viewer is initialised, the `GridModelViewPart` also specifies some global operations that are common for all Grid model views. These operations contain copy/paste support, deletion of files and drag'n'drop. If needed, the `GridModelViewPart` also creates UI elements for these operations like context menus or buttons.

All currently available implementations of the `GridModelViewPart` make use of a tree as a structured viewer. Therefore, the abstraction layer holds another class that relies on a tree viewer as structured viewer and builds up its functionality on this particular viewer. This class is called `TreeControlViewPart`. It basically implements the `createView(Composite)`-method of `GridModelViewPart` and defines another abstract method where subclasses may specify columns for that tree.

Last but not least, the `ElementManagerViewPart` accounts for these Grid model views that have an instance of `IGridElementManager` as their root element. It is a subclass of `TreeControlViewPart` and therefore uses a tree as its viewer. Element manager views are intended to show instances of a particular type of grid element such as jobs or connections. In this way, the user has access to the central repository of the elements. The views itself may present their data in two different ways that are already implemented in this abstract subclass. One is to show all elements in a flat structure. The other way shows the elements in the structure that is given by the Grid model, i.e. the project, subsequent folders and finally the elements themselves. An example can be seen in fig. A.7 for the Grid connection view.

5.1.2 Grid project view

The Grid project view is possibly the most important UI element within the g-Eclipse framework. It is based on the g-Eclipse view abstraction mechanism and has the Grid model root (`GridRoot`, see section 2.1.3) itself as its root element. This means it directly shows the model tree with the projects as top level elements and the project structure beneath these elements.

Fig. A.1 shows a screenshot of the Grid project view. Here, the view contains one project named "My-FirstGridProject" with its structure. The view defines further actions for the contained elements. Some of these actions can be seen in the context menu that shows up in the screenshot. Since this menu is context sensitive, only a subset of the full functionality is shown. The currently selected element (`MyJob.jsdl`) is of type `IGridJobDescription`. Therefore the context menu shows the action for submitting a job. All other actions are common actions that are already defined in the `GridModelViewPart`-class.

5.1.3 Grid job view

The Grid job view is the central managing facility for all submitted Grid jobs. Whenever the user submits a job to the Grid, this job is added to the `IJobManager` and the job view gets informed that there is a newly submitted job. Later, this job and some of its detailed information will be shown in the job view. Within this view, the user has the ability to monitor the status of the job or to cancel the job. As the job view is a subclass of `ElementManagerViewPart`, it is able to show the jobs in a flat or hierarchical structure.

Fig. A.2 shows a screenshot of the Grid job view displaying jobs in a flat structure. The jobs are presented in a tree with six columns. The first column shows the name of the job, the second shows the project this job belongs to, and the third column shows the ID of the job that is used to reference this job in the Grid. The fourth, fifth and last columns shows the information about the job's status, namely its current status, and additional "reason" message (which contains the reason for a job failure for instance), and the time of last update.

5.1.4 Grid connection view

Like the job view, the connection view is implemented as a subclass of `ElementManagerViewPart`. Therefore, its root element is the `IConnectionManager`. Hence, the connection view shows all currently defined connections in any project. Additionally, the Connection View allows the creation of connections that do not belong to any project, the so called "global connections". This is a functionality that is different from the job view since it does not make sense to create a Grid job without a corresponding project. In that sense, the connection view can be used independently from the Grid model to browse the Grid. Therefore this view is also called *gExplorer*.

Fig. A.3 shows a screenshot of the Grid connection view. The connections are shown in a tree as flat representation. The tree itself has four columns. The first shows the name of the connection, the second shows the project the connection belongs to, the third shows the size of the files and the fourth shows the date of last modification. Since the information that can be gathered about the elements of a connection depend on the protocol that is used to access these elements, it may happen that not all information is available for an element. This can be seen in the fourth column where no information about the date of last modification is shown for GridFTP folders. This is a result of the missing functionality for retrieving this information in the gridFTP protocol, in the case of this particular connection.

5.2 Perspectives

A perspective in the Eclipse sense is a collection of views and other UI elements that serve a dedicated purpose. For example, different views are required when developing a Java application, compared to when debugging it. The g-Eclipse framework defines four perspectives for accessing the Grid that will be described below. Screenshots for all these views are shown in section A.2.

5.2.1 User perspective

The Grid user perspective comes with the Grid Projects View and the Glue Information Viewer on the left, the Authentication Tokens view, the Connections view, the Jobs View and the Properties view in the bottom panel.

This perspective is intended to provide the necessary information and tools for a Grid user to perform typical grid tasks. Users are able to define their job descriptions using the JSDL wizard and edit them later with the multi-page JSDL editor to set some advanced parameters. Then users can submit the job description to the grid using the Job Submission Wizard.

The Jobs view shows all currently submitted jobs. Some jobs details are shown as well as the job description file used to create the job. Also the current job status is refreshed automatically every 30 seconds, but the user can turn this update off, or manually update the status of one or more selected jobs. More detailed information is shown by the Properties view or the Job Details view.

The Connection view is used to show all currently defined connections to remote storage elements. Connections can be defined by mounting storage elements from the Storage branch of the Grid Project view or by defining them using the Connection wizard. The Authentication Tokens view shows all currently existing tokens. These tokens can be created by the Authentication Token Wizard. This wizard can be started manually or by specific plugins if an authentication token is required to conduct user's actions on the grid.

JSDL Editor

The JSDL Editor is the default editor for JSDL (*.jsdl) files and is part of the User perspective. It is launched automatically after a JSDL file is created with the aid of the JSDL wizard, or when a user chooses to edit an existing JSDL file. This editor is a multi-page form editor that extends the `FormEditor` class. It is comprised of five pages (tabs) and one page for the raw source of the JSDL input file. Each of the five pages extends the `FormPage` class and allows viewing and editing of different JSDL elements. Each page is further divided into two or more sections. Each subsection is distinct and is responsible for maintaining the information of a different JSDL element.

The first page is actually an Overview page, introducing the user to the other pages and allowing for editing a few of the most important parameters. On the second page, Job Definition, Job Description and Job Identification elements along with their respective attributes can be viewed and edited. The third page is responsible for the generic Application and POSIX compliant Application elements. The fourth page holds information for the Data Staging elements. Finally, the fifth page holds information for the Resources elements. On each page, single-valued attributes are represented by text widgets and multi-valued attributes are represented by list widgets. Attributes with boolean data values are represented by checkbox widgets. Furthermore, checkbox widgets are also used to add or remove specific elements from the JSDL file as in the case of the Additional POSIX Application Elements section on the Application page.

5.2.2 Developer perspective

The Grid developer perspective comes with the Grid Project view in the left pane, the Authentication Token view, the Terminal view and the Monitoring view in the bottom pane. The toolbar area contains items for selecting different launch configurations. The initial perspective is kept simple on purpose, as the developers' view depends very much on the underlying programming language used. The idea behind its usage is that the user creates his CDT/JDT project which will switch to the corresponding perspective where the user can still use the launch configurations to debug his application on the grid. Wanting to change again to the meta level of grids, the user can switch back to the initial developer perspective, giving him the views to explore his running jobs and their status or manually interfere with their deployment or submission.

5.2.3 Operator perspective

The Operator perspective consists of seven views – Grid project, Information viewer, Connections, Authentication Tokens, Batch System Jobs, Properties and Terminal – plus the Batch System editor. This perspective is intended to provide the necessary information and tools for a Grid site administrator to perform his daily tasks. The administrator will be able to connect to computing elements that he controls. He can then connect and view the details of the corresponding batch server in the editor. The editor displays the batch system server with its properties and queues along with all the worker nodes

(including their properties) that are controlled by the computing element. Using the batch system editor, the administrator will be able to alter the properties of batch queues and its worker nodes. Examples of these alterations include setting the queue manager to “drain” its queue, allocating more resources to a specific VO, setting worker nodes to “down”, etc. The administrator can choose to save the properties that he has set for the batch system, allowing him to quickly switch to a previously saved configuration.

An example use of the Operator Perspective could be as follows (see fig. A.5). The administrator first transfers some files to the computing element using the Connections view. Then he drains the queues of the computing element using the batch system editor. After the queues have been drained, the administrator logs in to the computing element using the terminal. Maintenance work is then performed on that node, such as package updates, reconfiguration of the middleware, etc. When he has completed the maintenance work, the batch editor is used to open the queues again. In order to test that the site is operational, the administrator will then execute some test jobs on the site. This is done by first activating a certificate using the Authentication Tokens view and afterwards opening a project in the Grid Project view. The administrator uses this view to execute the test jobs, while, maybe at the same time, using the terminal to track the job at a lower level.

5.2.4 Grid exploring perspective

The Grid exploring perspective comes with three views – an editor and two identical Grid Connection views that are arranged side by side. This perspective is well suited to browse the Grid storage resources and to manipulate data on the Grid. With the help of copy/paste or even drag’n’drop, the user is able to copy files from one location to the other using both connection views. When a file is double-clicked in any of these connection views or the Open-action is selected from the context menu, the corresponding file is opened in the editor and can be edited on the fly. After the user has edited a file and chooses to save it, the changes are uploaded again and are incorporated to the remote file.

Fig. A.7 shows a screenshot of the Grid exploring perspective. The top half of the vwindow shows an editor with an open file. On the lower half the two connection views are displayed while a drag and drop file operation is being performed. These connection views can show their content in either a flat- or a hierarchical-representation. This representations may be changed on the fly by the user. In the hierarchical representation all connections are shown with their underlying structure within the Grid model, Grid Projects (and global connections) being at the top level of tree structure. All project-specific connections are shown within their projects. In the flat representation, on the other hand, all connections are shown together at the same level. However, here an additional column is visible in the table to show the project ownership of each connection.

5.3 Wizards

Wizards are commonly used in the Eclipse user interface. In addition to wizards which are provided within plugins, it is often necessary to be able to provide wizards that can be extended by other plugins. It is, for example, possible to provide new entries to the “New Project” Wizard by implementing the `INewWizard` interface of Eclipse and registering that implementation to an extension point.

To provide a similar functionality for g-Eclipse plugins, a wizard page can be used to create an extendable wizard. To add the possibility to select a wizard from within another wizard, a `WizardSelectionListPage` (see Fig. 5.2) has to be added to the base wizard. This page provides a list of all `IWizardSelectionNodes` that were passed to the `WizardSelectionListPage`. `IWizardSelectionNodes` provide the text and the icon for the list of wizards on the `WizardSelectionListPage`. They are also used to instantiate the wizard if the user chooses the wizard it represents.

Since, in most cases, the wizards to display in the list should be provided using an extension point, the class `ExtPointWizardSelectionListPage`, which is based on `WizardSelectionListPage`, provides

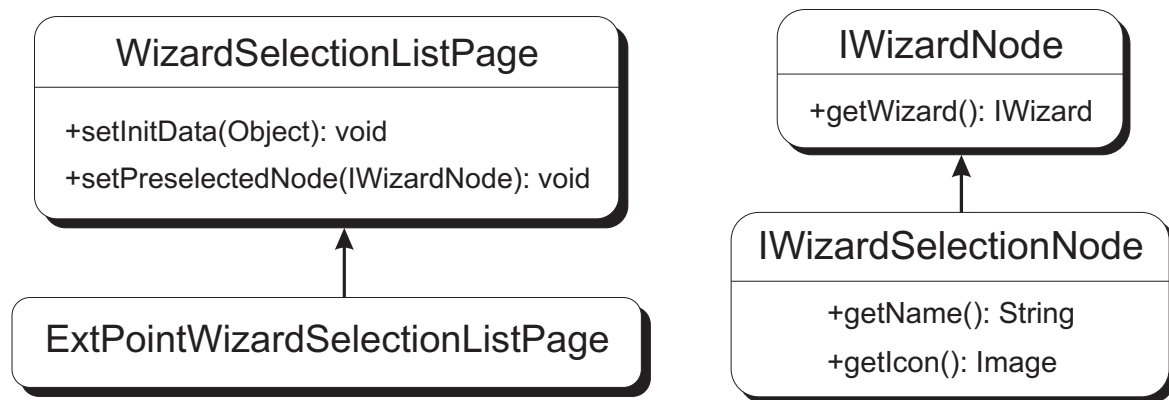


Figure 5.2: Wizard pages for selecting other wizards (left hand side) and the interfaces representing the wizards both including their most important methods.

this functionality. Instead of a list of `IWizardSelectionNodes` the `ExtPointWizardSelectionListPage` requires a `String` containing the extension point id to query the wizard list from.

As the wizards in the list are accessed using interfaces, it is not possible to pass any initialisation data through the constructor. If they require initialisation data (which might later be used, for example, in the `finish()` method) they have to implement the `IInitializableWizard` (see Fig. 5.3) interface.

If the `WizardSelectionListPage` should be skipped because it is already known which wizard should be used, the wizard can be set by using the `setPreselectedNode()` method.

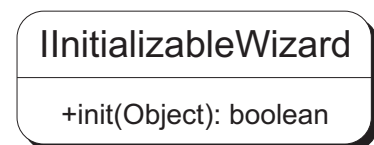


Figure 5.3: Interface for initialising wizards.

5.4 UI part of the Error Reporting mechanism

In section 4.2 we explained the g-Eclipse error reporting mechanism. In this section, we will give an overview of the UI resources that are built upon this mechanism. This includes the possibility of having active solutions and a dedicated error dialog that is called a problem dialog within g-Eclipse.

5.4.1 Active solutions

An active solution has to implement the `solve()`-method of the `ISolver` interface. This method is called when the user “activates” the solution by clicking it on the `ProblemDialog`. Currently, g-Eclipse contains some implementations of active solutions. These are:

LogExceptionSolution: This solution’s `solve()`-method reports the related problem’s exception to the Eclipse logging mechanism. Furthermore, if present, the error view will be opened in order to present the detailed log message to the user.

MailToSolution: This solution is automatically displayed by the `ProblemDialog` if the `IProblem.mailto` field is not empty. Activating it the user will open the system mail client with the provided email address to report the problem.

PreferenceSolution: The preference solution is used whenever the solution of a problem could be related to a preference setting. The `solve()`-method of this solution therefore opens the preference

dialog and points the user to that page of the dialog that may be responsible for an incorrect preference setting.

ViewSolution: This kind of solution is used whenever opening a view for displaying some data could help the user to understand the problem. The `solve()`-method of this solution therefore opens the view defined for the solution in the `problemReporting` extension point.

Furthermore, developers may specify their own active solutions that are adapted to a specific problem. Examples of every type of solution can be seen in fig. A.8 and will be explained in the next section.

5.4.2 The problem dialog

The problem dialog is an extension of Eclipse's error dialog. Besides the functionality to display the error message and more detailed information about that error in an optional area, the problem dialog also shows the solutions that are appended to a specific problem. If a solution is active, its representation in the dialog will be a link that executes the solution's `solve()`-method. A passive solution is simply represented as a declarative string.

Fig. A.8 shows a screenshot of this dialog. This dialog appeared when g-Eclipse tried to import CA-certificates from a remote location without success (see section 3.2). The problem obviously was that the connection could not be established. The details area at the bottom shows that the connection timed out. Beneath the reason for the problem, the dialog shows a list of possible solutions that were provided by the import mechanism:

Check your internet connection: This is a passive solution as this has nothing to do with the software and g-Eclipse is definitely not able to check if the user has a working connection to the Internet. Therefore, this solution is just presented as informative text.

Check your server URL: This is a custom solution that, when activated, will reopen the dialog where the user specified the URL from which the certificate should be imported.

Check your proxy settings: This solution refers to the g-Eclipse preference page where the user can specify his proxy settings. When this solution is activated, it will open the preference dialog and will bring the corresponding preference page to the top.

Log exception: This solution is always present in the Problem Dialogs and allows the user to log the underlying exception of the problem to the Eclipse logging mechanism. Furthermore, it will try to open the error log view.

Send error report: This active solution is displayed if the problem's `mailto` field is not empty. When clicking it the default mail client will open with the provided email address, allowing the user to report the issue.

A Screenshots of the most important UI elements

A.1 Views

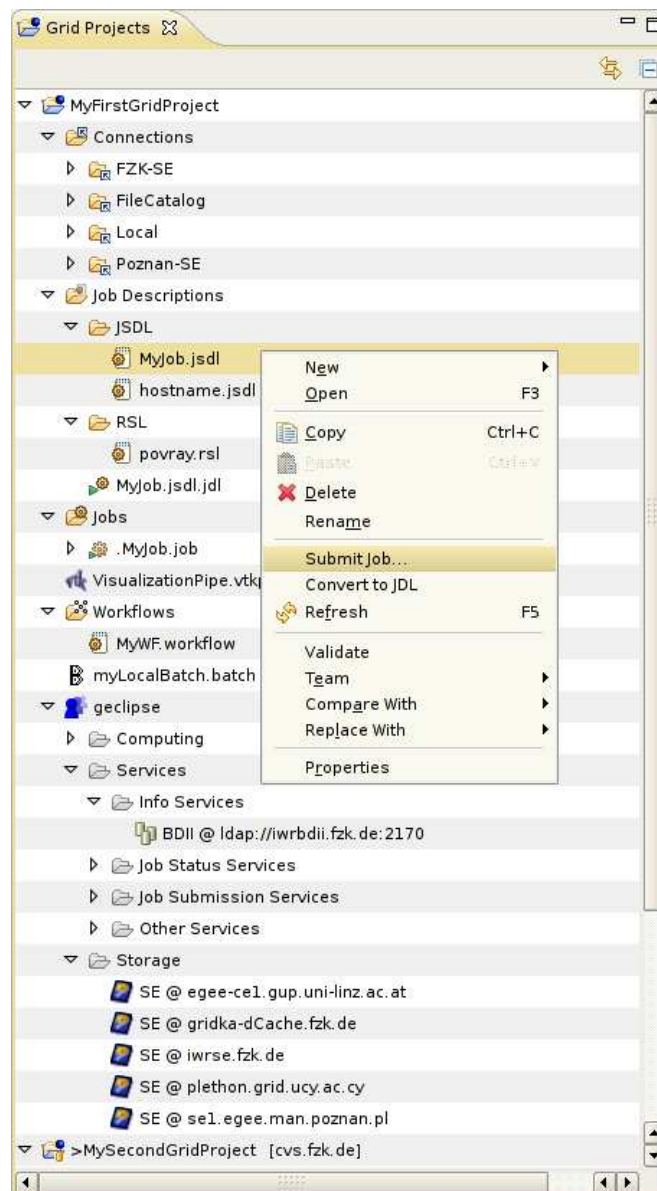
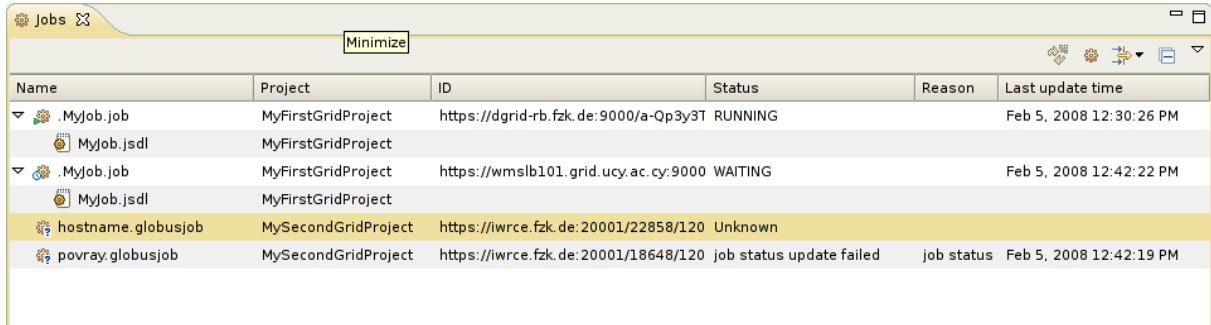
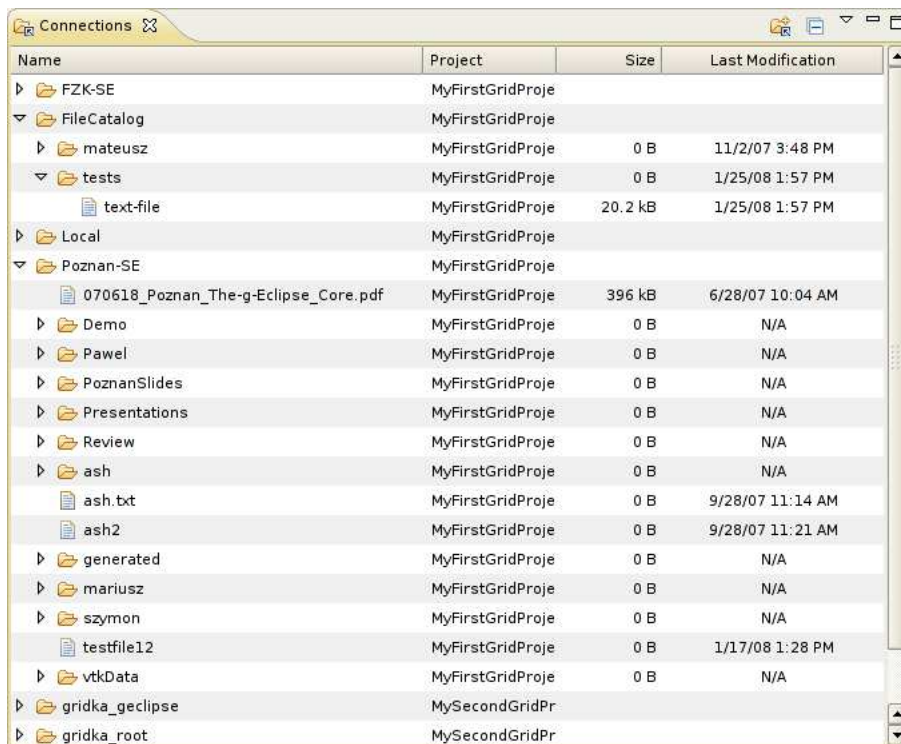


Figure A.1: The Grid Project view. See section 5.1.2 for details.



Name	Project	ID	Status	Reason	Last update time
MyJob.job	MyFirstGridProject	https://dgrid-rb.fzk.de:9000/a-Qp3y3T	RUNNING		Feb 5, 2008 12:30:26 PM
MyJob.jsdl	MyFirstGridProject				
MyJob.job	MyFirstGridProject	https://wmslb101.grid.ucy.ac.cy:9000	WAITING		Feb 5, 2008 12:42:22 PM
MyJob.jsdl	MyFirstGridProject				
hostname.globusjob	MySecondGridProject	https://iwrce.fzk.de:20001/22858/120	Unknown		
povray.globusjob	MySecondGridProject	https://iwrce.fzk.de:20001/18648/120	job status update failed	job status	Feb 5, 2008 12:42:19 PM

Figure A.2: The Grid Job view. See section 5.1.3 for details.



Name	Project	Size	Last Modification
FZK-SE	MyFirstGridProje		
FileCatalog	MyFirstGridProje		
mateusz	MyFirstGridProje	0 B	11/2/07 3:48 PM
tests	MyFirstGridProje	0 B	1/25/08 1:57 PM
text-file	MyFirstGridProje	20.2 kB	1/25/08 1:57 PM
Local	MyFirstGridProje		
Poznan-SE	MyFirstGridProje		
070618_Poznan_The-g-Eclipse_Core.pdf	MyFirstGridProje	396 kB	6/28/07 10:04 AM
Demo	MyFirstGridProje	0 B	N/A
Pawel	MyFirstGridProje	0 B	N/A
PoznanSlides	MyFirstGridProje	0 B	N/A
Presentations	MyFirstGridProje	0 B	N/A
Review	MyFirstGridProje	0 B	N/A
ash	MyFirstGridProje	0 B	N/A
ash.txt	MyFirstGridProje	0 B	9/28/07 11:14 AM
ash2	MyFirstGridProje	0 B	9/28/07 11:21 AM
generated	MyFirstGridProje	0 B	N/A
mariusz	MyFirstGridProje	0 B	N/A
szymon	MyFirstGridProje	0 B	N/A
testfile12	MyFirstGridProje	0 B	1/17/08 1:28 PM
vtkData	MyFirstGridProje	0 B	N/A
gridka_geclipse	MySecondGridPr		
gridka_root	MySecondGridPr		

Figure A.3: The Grid Connection view. See section 5.1.4 for details.

A.2 Perspectives

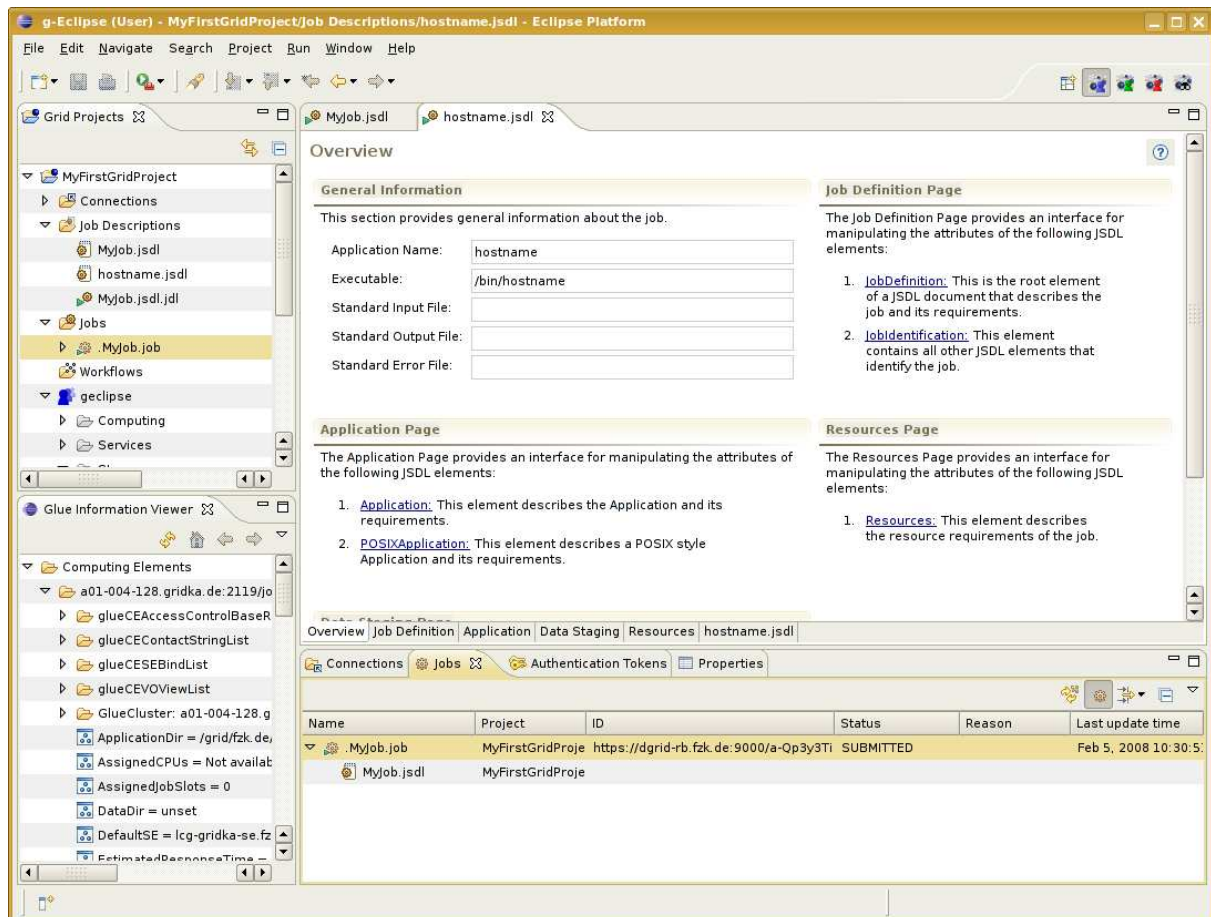


Figure A.4: The User Perspective with the JSDL multipage editor. See section 5.2.1 for details.

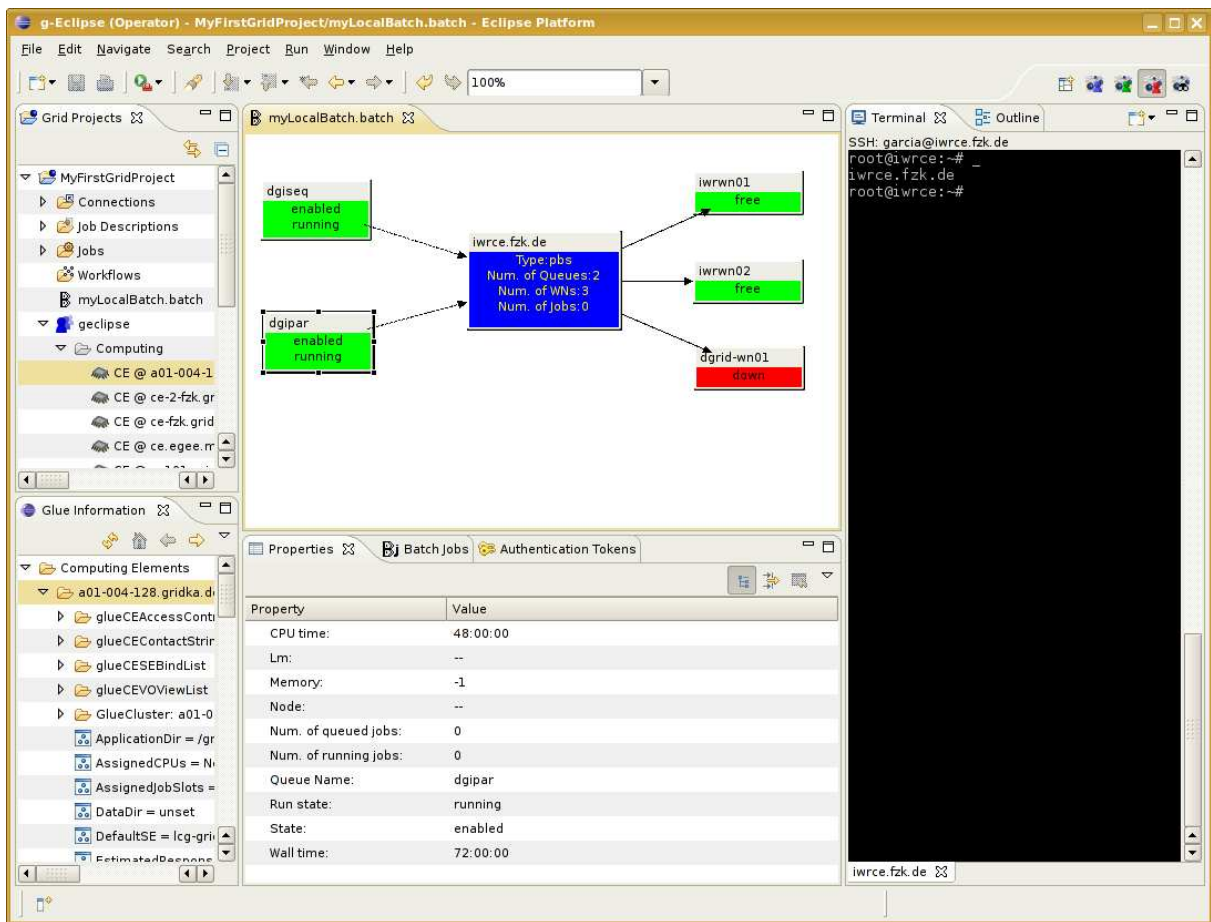


Figure A.5: The Operator Perspective with the batch system viewer. See section 5.2.3 for details.

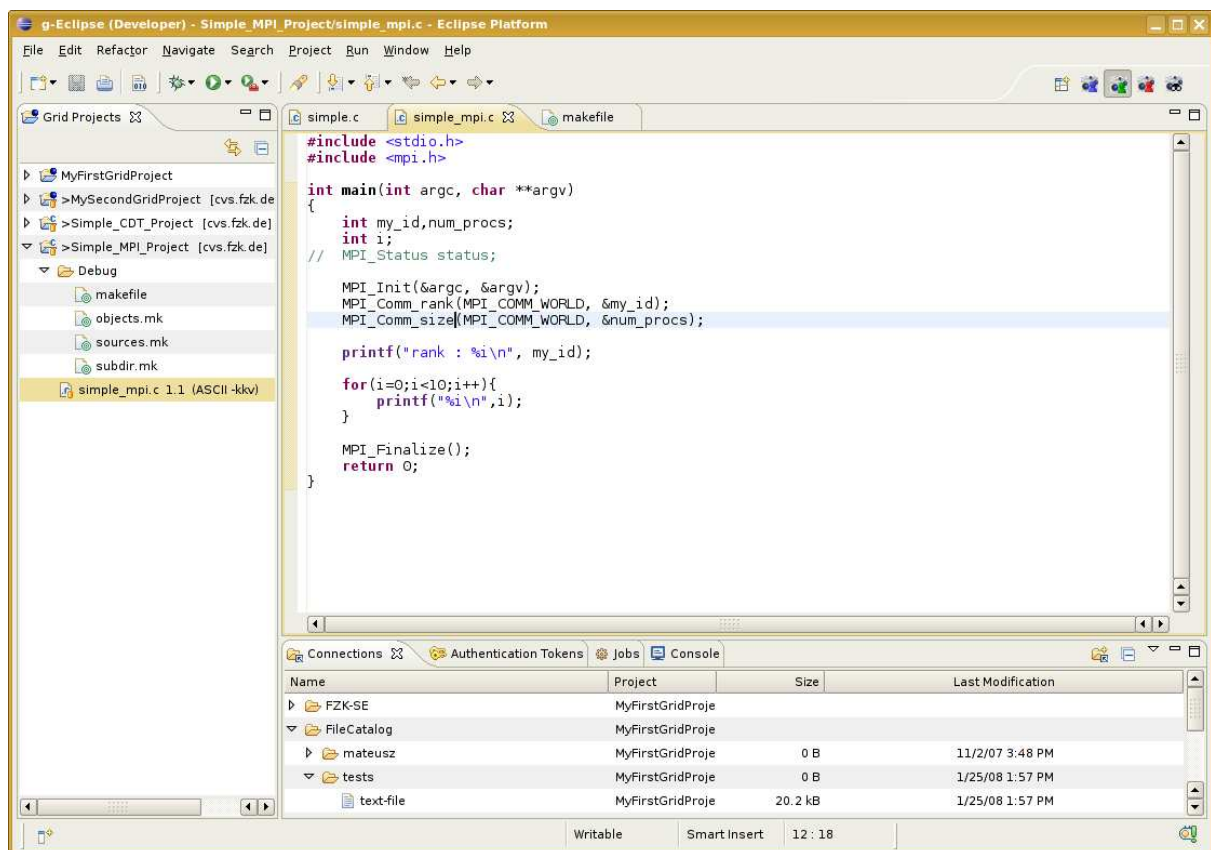


Figure A.6: The Developer Perspective. See section 5.2.2 for details.

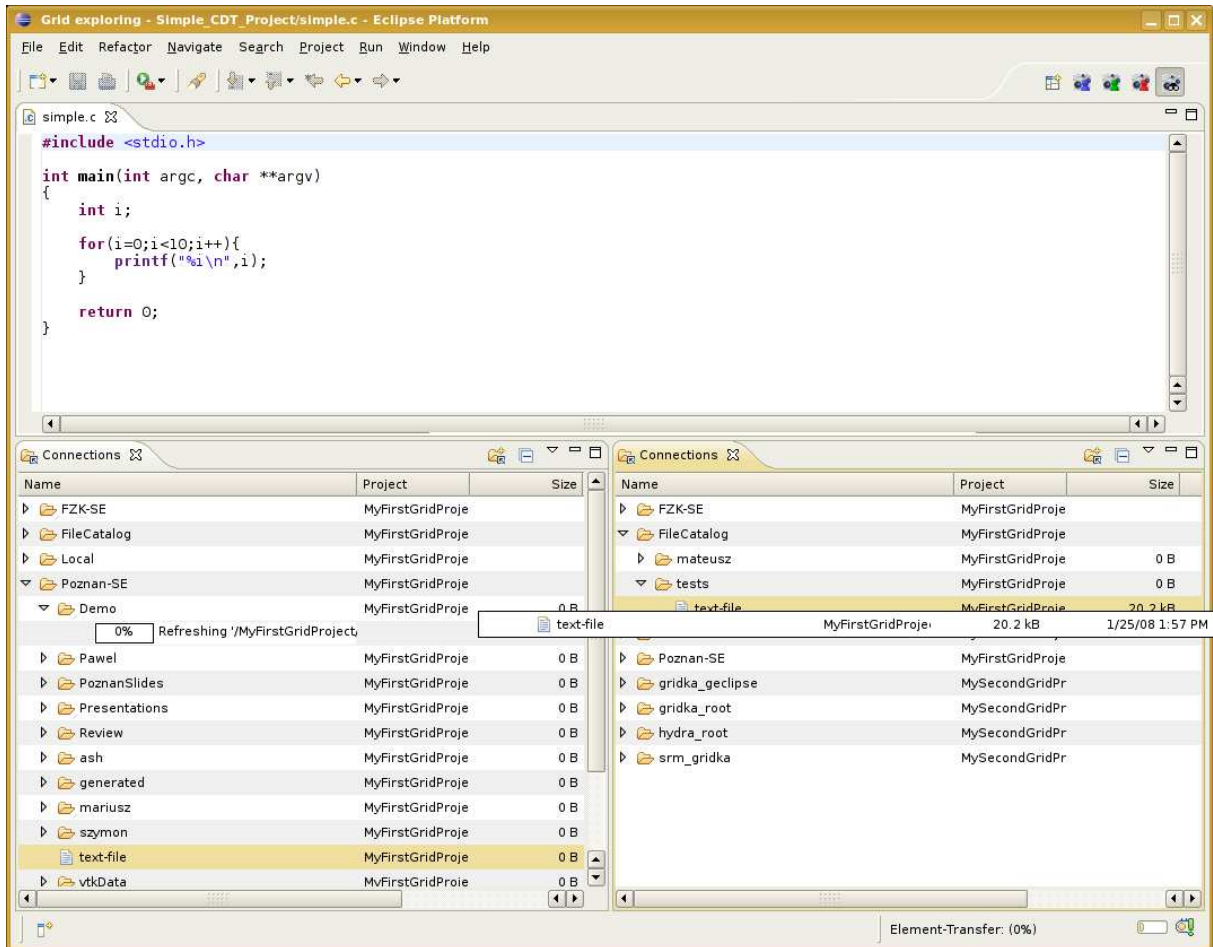


Figure A.7: The Grid Exploring perspective, copying a file by 'Drag and Drop'. See section 5.2.4 for details.

A.3 Dialogs and wizards

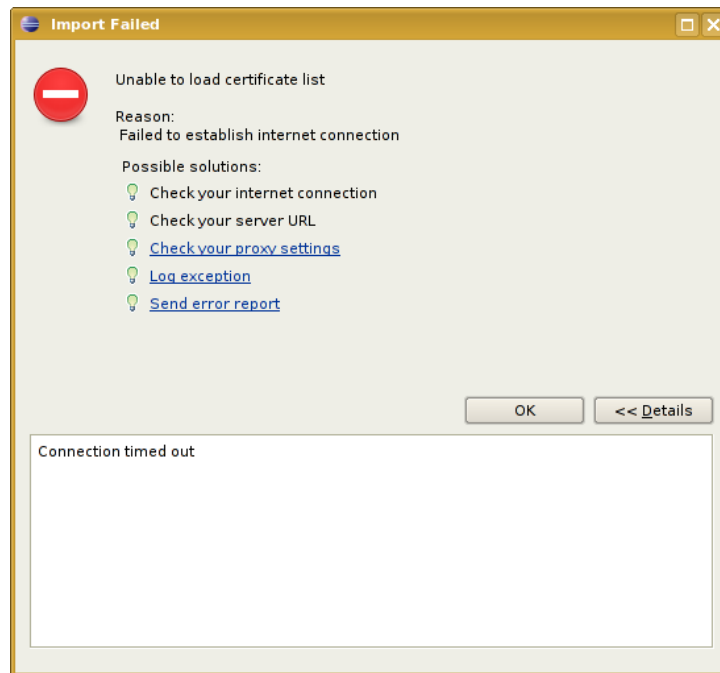


Figure A.8: The Problem Dialog. See section 5.4.2 for details.

Bibliography

- [1] *g-Eclipse Architecture II*, g-Eclipse Deliverable D1.5, month 9
- [2] *g-Eclipse Architecture I*, g-Eclipse Deliverable D1.2, month 3
- [3] *OSGi Alliance*, <http://www.osgi.org>