The Wizards

Design Overview

By MHGameWork  
aka Michiel Huygen

Table of Contents

[Introduction 4](#_Toc255337814)

[Some notes on this document 4](#_Toc255337815)

[Naming style 4](#_Toc255337816)

[Part I: Implementation 5](#_Toc255337817)

[Planning 6](#_Toc255337818)

[Small things to do 7](#_Toc255337819)

[Things to do 7](#_Toc255337820)

[Things to do in long term 8](#_Toc255337821)

[Cool things to do in long term development 8](#_Toc255337822)

[Bugs 9](#_Toc255337823)

[Design Flaws 9](#_Toc255337824)

[Part II: Design overview 10](#_Toc255337825)

[Application overview 11](#_Toc255337826)

[Data System 13](#_Toc255337827)

[DataObject 13](#_Toc255337828)

[Conclusion 13](#_Toc255337829)

[Some data examples 14](#_Toc255337830)

[Rendering Architecture 15](#_Toc255337831)

[Modules overview 16](#_Toc255337832)

[Part III: Detailed Design 18](#_Toc255337833)

[Client 19](#_Toc255337834)

[Physics 19](#_Toc255337835)

[Core 21](#_Toc255337836)

[EmbeddedFile 21](#_Toc255337837)

[QuadTree 21](#_Toc255337838)

[Graphics 22](#_Toc255337839)

[Core 22](#_Toc255337840)

[Helper classes 22](#_Toc255337841)

[Physics 24](#_Toc255337842)

[Editor 25](#_Toc255337843)

[Terrain 26](#_Toc255337844)

[Client 26](#_Toc255337845)

[Editor 27](#_Toc255337846)

[Todo: 28](#_Toc255337847)

[Entity 29](#_Toc255337848)

[Client 29](#_Toc255337849)

[Player 30](#_Toc255337850)

[Core 30](#_Toc255337851)

[Client 30](#_Toc255337852)

[ClientMain 31](#_Toc255337853)

[EditorMain 32](#_Toc255337854)

[WorldDatabase 33](#_Toc255337855)

[TODO 33](#_Toc255337856)

[Required functionality 34](#_Toc255337857)

[Concept 34](#_Toc255337858)

[Design 35](#_Toc255337859)

[Current Implementation!!! 39](#_Toc255337860)

[Notes 42](#_Toc255337861)

[Older 43](#_Toc255337862)

[Tree Engine 45](#_Toc255337863)

[Tile Engine 46](#_Toc255337864)

# Part I: Implementation

# Part II: Design overview

\* This is currently not implemented, or designed

## Data System

This explains how data is structured and stored in The Wizards.

Every DataElement belongs to a single DataObject. A DataElement can reference other DataObjects

### DataElement

A DataElement represents one piece of cohesieve data which is loaded in one piece into RAM.

This is a DATA class. It DOES NOT contain ANY LOGIC.

It can contain references to DataObjects by holding a pointer representing that DataObject.

### DataObject

The idea is that the main objects like ‘Entity, Terrain, Texture, Object … ‘ are DataObjects.

A DataObject contains DataElements. It does not hold any data of its own, only the elements.

The DataObject is implemented as a pointer by using an Interface as a type. (IObject, IEntity, ITexture)  
Factories can use these pointers to identify the different instances of the DataObjects and provide DataElements accordingly

### OLD

**It can’t contain DataItem’s (breaks encapsulation)**

To solve this DataItem problem. Interfaces are created (IEntity, IMesh,...). These are used in the DataElements. The objects behind those interfaces are totally free atm.

**Now it seems that there will be a Editor Data System en possibly a Client Data System and a Server Data System**

**Idea: maybe the Data System can work with some sort of identifier that is stored in the DataElement, and then the Data System provides a way to retrieve DataObjects form it using that ID.**

### DataObject

A DataObject is defined as an object that provides access to the data of the ‘thing/item/object’ the DataObject represents.

To ensure an easy to use and fast system data system, DataObject’s should have the following properties:

* All DataObject has to be unique. No 2 instances can be created referring to the same object.
* All DataObject’s have to be loaded into RAM at startup. This is to allow references from one DataObject to another.
* They should hold a minimum in data. This is a consequence from the fact that all objects have to be loaded in RAM.
* A DataObject is an object that provides access to the actual data from the object it represents. The way this data is acquired is encapsulated. (Users of a DataObject don’t need to know where the data comes from)

**IMPORTANT!** Because of the fact that the way data is retrieved, highly depends on use of network synchronization, disk caching, multi-threading … This makes it necessary to put the Data System on the **Application Logic** layer. **DataObject’s should be completely hidden from the non-Application Logic!**

**IMPORTANT EDIT!!!!**

**The Data System used to be application logic. Now i came to realise that it is possible to keep the same definition and not make the Data System AL. One could see the DataSystem as an Abstract interface to those AL concepts, but the implementation is hidden.**

### Conclusion

The logic for acquiring data resides in the Application Logic. In Application Logic, DataObject’s are used to retrieve and store necessary data. This requires the use of multi-threading, network sync, disk cache. Disk cache and maybe some other responsibilities (such as versioning) are given to the WorldDatabase.

The usage of the data is done by small ‘Factory’ objects. The Functionalities provide an interface that gives them the Data in the format they need, and that interface is then implemented at the Application Logic level and passed in the constructor to the Functionality classes.

Example

**DataObject**: Entity, Object  
**Data classes**: EntityData, ObjectRenderData, ObjectPreprocessedRenderData  
**Factory classes**: IEntityPhysicsFactory, EntityPhysicsFactory, EntityPreprocessedPhysicsFactory  
**Functionality classes**: EntityClientPhysics(this class currently exist, but is not implement yet as in this example)

The Application Logic creates the EntityClientPhysics objects. For each EntityClientPhysics object, a Factory class is provided. These factory classes are given a DataObject (in this case Entity).

The EntityClientPhysics then uses the IEntityPhysicsFactory it has been given to create a PhysX actor. Then the EntitPhysicsFactory, or the EntityPreprocessedPhysicsFactory can create the PhysX actor the way they want it. They acquire their data by using the DataObject. How they do this is encapsulated. (This is of course the point of the whole design). Currently the data is meant to retrieve the data from.

Note that, in this design, the EntityPhysicsFactory or EntityPreprocessedPhysicsFactory could be seen as the DataObject for the Functionality classes.

Design summary  
Functionality classes use a **semi-DataObject**. They define it as an **interface**, so that the Application Logic can implement them, using the real DataObjects. In unit-tests, such a semi-DataObject will be implemented without use of the actual DataObject, since this is application layer. This allows for very flexible design. (and transparent unit tests)

### Some data examples

Following are **examples of data** that should be stored. (This is illustrative only, for creating the design. It does not refer to any implementations at this time)

* Terrain
  + Heightmap (2 dimensional array of heights)
  + List of all weightmaps for the textures used in the terrain.
  + List of Materials used in the terrain (which are linked to textures). (Maybe this data should be in the same DataElement as the weightmaps???
  + Preprocessed terrain data, used for fast rendering
* Texture
  + Simply the dimensions and pixel colors of the texture
  + Dimensions, hash of texture, number of mipmaps, other user specified info about the texture (for example, whether it is tileable)
  + Full texture with mipmaps in for example DXT format.
  + Creation info: info about when, from which file the texture was created from

## Modules overview

Todo: explain all the different modules/submodules and their dependencies

Note that every module in this diagram can only use lower modules. No dependencies can go upwards. Dependencies between modules in the same level are currently allowed.

# Part III: Detailed Design

## Editor

Note: WizardsEditor is currently using the FormNew property as the GUI, the Form property is a remainder of the old DevComponents interface.

Maybe switch to a in-DirectX editor?

### General Design

There should be a strict abstraction from the user interface logic to other logic. All logic in the editor should be written in user interface independent classes. Then user interface classes should call switches and set parameters on the logic so that it responds to user input. These user interface classes are therefore also part of the Application Logic.

The WizardsEditor is the main class at the moment, and is full Application Logic.

There are interfaces available that can be added to the WizardsEditor to extend its functionality. Interfaces currently available are:

IWizardsEditorDatabaseExtension and IWorldEditorExtension

### TODO

Implement a system that can show a progress bar and notify the user that the app is working.

### Data System

Maybe a class EditorDataSystem should be created, holding the entire editor data system. (Note this is application logic, can only be used by other application logic)  
Currently the EntityWizardsEditorExtension has this responsibility.

There are no more FullData-classes. Data should be split in coherent pieces. The EditorObject performs the function of the FullData, by holding all the pieces (for editor use).

The Editor Objects implement the DataObject pointers (eg EditorEntity implements IEntity)

### Classes

#### Application Logic

* **WizardsEditor**  
  This is the main class for the editor.   
  The WizardsEditor exposes his responsibilities to other objects using interfaces  
  Responsibilities are:
  + GUI
  + WorldDatabase: Note that database functionality could be moved to the World Editor
  + WorldEditor creation
  + Undo/Redo pipeline
* **IWizardsEditorDatabaseExtension**  
  This interface exposes methods to extend the database functionality of the WizardsEditor.
* **IWorldEditorExtension**  
  This interface exposes methods to be able to extend the World Editor.

### Scene

**WARNING!! INCONSISTENCY!   
To conform with other DataObjects, Scene should be a namespace with Editor as a sub namespace, not the other way around!!**

**The Scene editor is what previously was the World Editor!**

The scene editor allows the user to create a scene, placing together different elements from different modules. It will save the spatial information to the scene.

The scene allows placement of predefined elements, but should also allow in place editing of the elements (eg editing an object internally, creating terrain).

A snapping system should be implemented

A Undo system should be implemented

The real World Editor will be a scene editor with a infinit scene. The details of this infinity implementation are not yet created (Using Sectors etc). This will be implemented when the world starts getting to big.

For the moment, there is a **World Scene**, specifiying the current world. The other scenes are meant as helper and construction scenes.

**TODO: WorldEditor should be adapted and renamed to SceneEditor**

#### Classes

* EditorScene  
  This is the Editor DataSystem object for a Scene. It provides functions to manage a scene.

## Player

### Core

* **PlayerController**Dit bevat the PhysX controller voor de player, en functies om hem te verplaatsen en te draaien.  
  **This should probably be moved to the client submodule**
* **PlayerInput**Dit gebruikt een PlayerController en verplaatst daarmee de player volgens standard input controls.  
  TODO: Dit moet later gekoppeld worden aan het systeem om zelf de keybindings in te stellen.  
  **This should probably be moved to the client submodule**
* **PlayerData**Dit bevat momenteel de positie en orientatie van de player, maar dit moet nog bekeken worden

### Client

* **PlayerClientPhysics**  
  This class implements the IClientPhysicsObject interface, thus represents the client physics object for a player.

## ClientMain

**Change: originally called** **ClientGame**

This module is the highest level module in the Client of The Wizards, and references all other modules. It is not referenced by ANY module.

THIS MODULE CANNOT BE REFERENCED.

This module contains the pure application logic: when to load, when to connect, when to render, what to render. It stitches all the client logic in the modules together to form the wizards.

### Classes

* **TheWizardsClient**  
  This forms the starting point for the client for the wizards. Start this to start the full client.  
  WARNING: there is some fishy server-client stuff going on in this class EDIT: less fishy now
* **LoginForm**This form is displayed at startup when the user needs to login. It connects to the server and logs in. This simply acquires user input, doesn’t do any connecting itself.

## ServerMain

This module is the highest level module in the Server of The Wizards, and (can) reference all other modules. It is not referenced by ANY module.

THIS MODULE CANNOT BE REFERENCED.

This module contains the pure application logic: when to load, what do load, what to process, what/when to send. It stitches all the Server logic in the modules together to form the wizards Server.

TODO: implement the ConnectClientNonNetworked functionality.

### Classes

* **TheWizardsServer**This is the main class, the starting point of the server, and delegates all functionality to the functionality classes, but decides which functionalities are used and when they are used.  
  This class will call other Application Logic classes that implemented some more complicated but specific functionality.

## ServerClientMain

The server and the client have constructors that allow them to run inside an already existing PhysX Core and XNAGame. This way, a single game can be created and multiple clients/servers can be added.

Probably next alternative method:

**TODO:** TheWizardsServerClient creates an xnagame and hooks the events. It then calls update, initialize etc on the server and client. Maybe a extra class should be added: a Client that is simply some kind of pluggable thing, and the real TheWizardsClient that creates xnagame en physx and uses this client to run the game.

### Classes

* **TheWizardsServerClient**Manages ServerClient stuff

## EditorMain

This class goes parallel with the ClientMain, as it is the highest level of the editor and calls all others.

THIS MODULE CANNOT BE REFERENCED.

This module contains the pure application logic: when to load/save, what to display, …. It stitches all the editor logic in the modules together to form the wizards editor.

* **TheWizardsEditor**  
  The starting point for The Wizards Editor. This starts and runs the editor.

## WorldDatabase

(Note: original name was just Database, but since that namespace already existed it was renamed to avoid confusion)

The WorldDatabase is closely linked to the Data System (see above). The point of the Data system design is to allow a full separation of the serialization and the data usage.

The WorldDatabase was originally designed to hold all data (it still is). But now its use is more aimed at the more final use in the Client/Server/Editor. For all objects, an alternative method should exist to export them to files. This is extremely useful for testing purposes. (See also the Data System)

### TODO

* Implement the ‘static’ DataElement’s. (assigned to a revision instead of a DataItem)
* Support DataElements that do not need versioning (e.g. preprocessed data).
* The IDataElementFactory has methods that require a DataItem and a revision. Maybe this should be removed, since a DataItem always contains its revision? This should be revised.
* Implement a safety hook: When a WorldDatabase is created, some kind of flag should be set in the directory it is operating on, that prevents creation of other WorldDatabase’s in that directory.
* Create helper factories:
  + A factory that uses .Net’s XMLSerializer to save
  + A factory that uses a common IStreamSerializer<T> interface (not defined yet). Idea is that this factory saves anything that given streamserializer can serialize.

### Required functionality

These are things that should be possible with the database system. Note that these things do **not have to be responsibilities** of the database!!

**Note**: by unique instance I mean one and only one instance of a class (NOT an instance that is different from all other)

* Store whatever data needs storing. Can hold anything.
* Store and retrieve a unique instance of a class that stores data.
* Allow storing multiple instances of a data class
* Some kind of linking between the data (There will be data that references something that is described by other pieces of data)
* Allow for some kind of object, which can hold unique instances of data classes. Which data classes it holds should be ‘not predetermined’, and can be different for different objects of that same kind.
* Allow for version storage, by choice
* Allow for network sync, by choice

### Concept

Next items simply describe the basic concept of the database.

#### Responsibilities

(The WorldDatabase is responsible for serializing the data to the hard drive.) It also takes care of the version control of the data.

#### Design constraints

A separation between this database system and the other functionalities is Extremely Important. (Since the design of this can and probably will change)

This means that the database can only be used at the application logic level.

### Design

What follows describes the current design of the Database. This is however very likely to change in the future, which is why the usage of the DB is encapsulated as much as possible from the rest of the engine. (Only access in the Application Logic)

Note: in the current design, all effort has been made to keep the database design as it is, but the things that will change are moved outside of the db.

#### Data Structure

Currently, there are 2 kinds of objects in the database. We will define them as follows.

* **DataItem**You have objects that are simply unique instances. They can be uniquely identified by their **type and a unique ID**.
* **DataElement**   
  You have objects that can be identified by **their type and a DataItem**.

For every type of DataItem, there is a list containing all unique instances of that type.

A DataItem can contain several DataElement’s. There can only be one DataElement of a certain type in a DataItem.

One exception to this rule is that there can be some DataElement’s that are uniquely defined by simply their type, and No DataItem.

This scheme resumes the layout. The squares represent instances and the names the type of the instance.

**DataItem**

A DataItem is identified by an ID and a revision, which is unique for the entire lifespan of the Database. All DataItem’s of one revision are loaded together in RAM.

A DataItem also has a type.

2 ways of using the DataItems:

Internal Use: Each revision contains a list of DataItems. This means that at one specific DataItem (eg with ID 4567) can be instantiated multiple times, one for each revision.

External Use: There is only 1 instance of DataItem with ID (4567). This is the way users of the database should look at the db, but currently there is no real support for this in the db itself. This will probably be the responsibility of the Data System.

A DataElement can reference a DataItem, but always refers to the DataItem of the revision it was saved in. There shouldn’t be any references from a DataElement to a DataItem of another revision (DataElement’s should be revision unaware!)

**DataElement**A DataElement is a part of the database system, meaning that the DataElement is **aware of the existence of the database**. (No encapsulation)

* Contains a **piece of cohesive data**.
* **Can reference DataItem’s**. (Can contain DataItem’s as part of their data)  
  **EDIT: Idea is to not have DataItem’s in the DataElement’s, but rather an interface that represents a reference to another object.**
* **Cannot reference another DataElement**!!! A DataElement can’t be referenced. (Duh!)
* **No persistence logic.** Some kind of DataElementManager (maybe DataElementFactory?) will take care of the specific logic for the persistence of the DataElement.
* **No restrictions on the actual data**. The database doesn’t care about redundant data (not one of its responsibilities).
* **Stored on disk**. DataElement’s are always loaded from and stored to disk.
* **No automatic caching**! There can be multiple instances of a DataElement, all referring to the same DataElement from a DataItem. The user is responsible for the caching. (This is because the logic by the user involves network syncing, multithreading and others, and this is of such complexity that it will be more simple to manage the caching at the same time as the multithreading,…)
* **Can contain change logic**. A DataElement can keep track of which data has been changed since it has been last saved.

**Important note!** The DataElement’s used by the database **do not have to match** (and probably won’t)the data classes used by the functionality classes. See the Data System for more info and an example.

**Database in short**: RAM skeleton of DataItem’s. DataElement’s can be acquired and stored by the Database.

#### Initialization

After the database instance is created and given the folder to store and retrieve data. The Database OverallData file is loaded, containing the NextUniqueID and the LastRevisionID.

The database itself holds a list of all the DataItemType’s that occur in any revision it has currently loaded.

You have to register all the types used as DataElements, and give them an UniqueID used to identify them on disk.

You also have to register all possible factories that can be used to load some given DataElement Type. You specify the default factories, which are used in saving the DataElement’s.

The database is now ready for use!

#### Serialization

DataItem’s have a type, a DataItemType attached to it. Currently, there is no real use for the DataItemType, other than debugging and serializing clarity for identifying the types of DataItem’s.

**DataItem’s can only be added/removed to the working copy!**

**DataElement’s can be saved to any revision!** This is to allow late retrieval of the data.

##### Loading

Loading is done by requesting the DataElement to the database. It will look in the supplied DataItem for the factory used to serialize that DataElement and tries to load it. If not found, ‘null’ is returned.

References in the data to DataItem’s are represented simply by pointers to the DataItem’s in the created DataElement.

##### Saving

The user can save a DataElement at any time by supplying a DataItemIdentifier and a RevisionIdentifier. This allows late retrieval of the DataElement, and does not require all revisions to be loaded.

Next only applies to the working copy.

When the user has changed a DataElement and wants to save it, it must pass it to the working copy, providing the correct DataItem. The Database will then cache the given DataElement in RAM. When the user then later requests a working copy save, all these cached DataElement’s are saved to disk(flushed), and removed from the cache.

~~Idea: maybe all changes to the db can remain in RAM until commited? This would then mean that Only One DataElement can correspond to a DataElement on disk!~~ (Nope, slow for saving, see above, and other more complicated issues; issues that arise when thinking about the final, very complex implementation with network sync and multithreading.)

#### Version control

Not all DataElement’s need version control (like preprocessed data).

Currently, version control is done in a SVN like fashion.

* On the Client the latest version of the world is stored, together with a working copy that holds the current world state as the Client last recorded it. The client can’t commit state.
* On the Server all versions are stored. Server also has a working copy?
* In the editor the last version is stored and the working copy. Other versions can be present too, in case reverting or comparing with previous version should be necessary. When the editor wants to commit, the working copy is sent to the server and is commited there. Same should be done in the editor (beware of the high coupling !!)

Commiting the database means that the working copy is stored to disk and a new version number is created and assigned to it. After commiting, there are no more DataElement’s assigned to the working copy revision itself.

**TODO:** DataItem’s have a version number. This number is the highest version number of all their elements, or otherwise state, the number of the last version where a DataElement of that DataItem has changed.

A DataItem also holds the list of all DataElement it contains, together with the factory used to save it and the version it was saved in. ~~This list doesn’t have to be loaded in RAM; it can be stored on disk.~~ Preferably loaded in RAM.

(There is an interface defined to make an abstraction of how the versions are saved.)

#### Disk Storage

For every version (and the working copy) a file is saved containing:

* All DataItemType’s used.
* Alle DataElementFactories used to store any DataElement that is owned by any DataItem in that revision. **Important:** The IDataElementFactories also have some kind of identifier that is unique everywhere (unique for always). This is to allow backwards compatibility: when a new DataElementFactory implementation is made for a DataElement, the old implementation can still be used to load the old data.
* Lijst met alle DataItem’s. Voor elk DataItem
  + UniqueID van het DataItem. Dit is uniek voor elk DataItem
  + DataItemType
  + List of all DataElement’s linked to this DataItem. For each DataElement, the factory used is stored, the revision it was saved in and the type of DataElementNotes

### Current Implementation

The db is not responsible anymore for caching, only caches for the working copy! Versioning is enough of a responsibility.

Current architecture consists of 2 parts:

* The DataItem’s and revisions structure. This stores which DataItem existed at which revision, and which DataElements it contained at that revision and by which Factory they were built.  
  All Data for a revision must be saved in one piece. Not all revisions have to exist on disk, but if a revision exists, it should contain all DataItem’s and info about them.
* The Data itself. All DataElement’s are stored to the disk independently of the structure of the DataItem’s. This allows late retrieval of the data or the structure in the other part.  
  There are no restrictions on which DataElement should exist.

#### Usages

You can save and load any DataElement at any time.

You add/change/remove dataelements, from the working copy only! These changes will be cached, and applied when the user calls SaveWorkingCopy.

You can commit the WorkingCopy. This will Save the working copy, and rename the working copy to the correct revision number. (Rename the folder, and the saved revision file)

You can get the list of all DataItem’s and the factories used for each DataItem for any revision. You can also save this data.

#### Classes

**TODO: implement support for DataElements that are simply attached to the database, not to a revision.**

**WARNING TODO: Maybe rethink the design for DataElement and DataElementFactory, it seems like there is something wrong there. (double identifier: identify a dataelement by its type and its factory, maybe only the factory should be enough)**

WARNING: If you put DataElements in the working copy, and try to load them before saving the working copy, you will not be able to load them. Functionality may be added to load dataelements using the workingcopy class

NOTE: Currently all revisions store a NextUniqueID value. This is some kind of backup measure to ensure that it is most unlikely that even by a malfunction the nextUniqueID gets corrupted. However it could be changed later on that only the WorkingCopy holds this value (and maybe the db for extra security?)

* **WorldDatabase**  
  This is the main class, representing the database  
  + Save/Load DataElement (simply stores the data, not the actual DataItem<->DataElement reference). Currently only saving with the default factory is supported. Other factories are only meant for backwards compatibility.
  + Save/Loawd RevisionData (= list of DataItems and factories used for each DataElement for each DataItem) (this can be cached)
  + Acquire Working copy
  + Manage DataElement Factories. Store the default ones used for saving, and old ones for loading old data.
  + Hold count of the next UniqueID for a DataItem
  + Manage a list of all registered DataElement’s. Only registered DataElement’s can be loaded.
* **WorkingCopy**This class provides methods to alter the working copy.  
  Holds cached DataElements and a RevsionDataElement
  + Add/Remove DataItem
  + Add/Edit/Remove DataElement (Sets a DataElement to a DataItem in the WorkingCopy and caches the DataItem)
  + Save Working Copy (Save All the Cached DataElements and save the RevisionData)
  + Commit Working Copy (Save Working Copy, and make it a revision)
* **DataItem**Each DataItem has a unique ID and is linked to a revision. It stores which DataElements exist for this DataItem in this revision, what revision they were saved in and with which factory.
* **DataRevision**Contains a list of DataItems. Also contains a RevisionIdentifier. This also saves the last unique ID in this revision
* **DataItemType**  
  DataItemType’s are unique in the entire database (not revision specific). So no versioning on the DataItemType’s
* **IDataElementFactory**This class provides functionality for retrieving a DataElement that is defined by a revision and a DataItem.
* **DefaultRevisionFactory**  
  ~~Default implementation of the IDataRevisionFactory~~  
  Is responsible for serializing the revisions.  
  This class exists because of a possible other file format later on, and it will be easier to have backwards compatibility that way. It was also introduced for clarity, (to remove some responsibilities from the WorldDatabase)
* **IDataElement**Interface for a DataElement. This is a empty interface, simply introduced for clarity. It probably shouldn’t ever have any members, since we don’t want any restrictions on the DataElement’s. It was introduced for clarity, instead of simply using ‘object’.
* **DataItemIdentifier**This struct uniquely defines a DataItem. The difference between this struct and the original, is that the original can have only one instance per DataItem, while there can be many identifier referring to the same DataItem
* **DataElementIdentifier**This class uniquely identifies a DataElement. It is used in serialization. The user should Register all Types used as DataElement’s with the database on initialization, so that these identifiers can be resolved.
* **DataElementFactoryIdentifier**This class uniquely identifies a IDataElementFactory. It is used in serialization. The user should Register all IDataElementFactory’s used on initialization, so that these identifiers can be resolved.
* **DataRevisionIdentifier**This struct uniquely defines a DataRevision. The difference between this struct and the original, is that the original can have only one instance per DataRevision, while there can be many identifier referring to the same DataRevision.
* **(IDataRevisionFactory)**This class is not yet implemented; it can be added when needed in the future. It may exist when there is a need to support different kinds of versioning (like GIT). But this will probably need more important design changes
* **~~DataElementXMLFactory<T>~~ Not possible due to technical issues with the xmlserializer**This is a helper factory, that uses the XMLSerializer in .Net to store the DataElement T. It also provides support for auto-resolving of DataItems. This works by some fishy code in DataItem, since XmlSerializer is quite non-versatile.  
  **EDIT: DataElement are not allowed anymore to use DataItem anymore. It will store an interface that abstracts the reference to an external object.**

### Notes

What follows are some random things that should be considered when implementing/reviewing the design.

* If for some DataElement only a small portion of the data changes, it may be desired to only save the changed part of the data. The DataElementFactory should take care of this. It should also save the previous version of the DataElement (the version number). The Database does not provide a fast way to access this info. The DataElementFactory is also responsible for assembly the pieces of the previous versions together.
* The DataElement can contain info about which data has been changed; the factory can make use of this data to only save changed sections.
* Not **All DataElement’s need previous versions stored!**
* There should also be a way to save data to a revision that is not the working copy. This is to allow late synchronization by the server.
* IDEA: Old DataElementFactories could be auto-detected by reflecting loaded assemblies. This could be done when a DataElement is requested and the factory was not loaded.

### Older

Not entirely the most recent definition, concepts still apply:

#### Overview

* ~~Full RAM storage (fake, of course disk optimized due to RAM limits)~~ (now Data System)
* All Data pieces are represented by a Single Instance
* ~~All Data is unique and independent, no data can be deducted from other data~~
* SVN or GIT –like system. Versions and commiting users stored.
* XML file for each data part. Compression should be implemented

Referenced by/Usage:

* Full access by Editor and Preprocesser
* Probably Full access by server. This will be a tight relationship
* NO ACCESS BY ClientMain!!!! (See notes below)

Note that full access means that the WorldDatabase will be used throughout the ENTIRE accessing module, so scattered along the classes. (Tight relationship)  
EDIT: This can only occur at the Application Logic level. The Database can’t be used anywhere in the functionality classes. They should directly use the data pieces, which they get from the Application Logic.

#### Detailed description

The World Database represents all the data that is necessary to reconstruct the **entire world** of The Wizards. The Database is presented as a **memory structure** and ~~secretly uses the disk~~ to persist its data (=this is the Data System). (In theory, when enough RAM, the design should allow the entire world to be

All the data in the database has a version number attached to it. It is for now unsure whether this will be a **SVN like or a GIT like system** (that is, with repositories everywhere, or just one repository and working copies everywhere). Also, for each piece of data, the user saving the data will be stored.   
The currently required things to have for the items are: a version per item, and a changed flag indicating that that items base was the set version, and it has been modified. A commit will then change the version number. (in case of svn-like this increase will be on the server, in case of git this will be on the client, with hash like revisions)

As for the Disk storage part, currently using files for data pieces and XML. This allows for max user readability and very low data loss risk. Compression should be used, certainly for storing older versions.

For the moment, the idea is that the WorldDatabase can be access by the Editor and Preprocessor on client side. The ClientMain itself is (For Now!!!) NOT ALLOWED TO ACCESS THE WORLDDATABASE.  
The server will also access the DB, most likely in a direct fashion (working on the database directly instead of copies like the ClientMain).

Note: Since the ClientMain is not allowed to access the db, it cannot access the frequently changing data of dynamic objects. (Note that in The Wizards, by definition, all objects can become dynamic at any moment) However, preprocessing is a procedure that is by idea something that works on static data. Since the only currently existing mechanism to access the WorldDatabase for the ClientMain is through the preprocesser, it is Most likely a parallel system must be devised for the dynamic data in the system.  
  
Edit: this part is kind of fishy. ClientMain will probably use the database, since it simply represents the world, which is exactly what the client is supposed to display.

## Networking

### Raw Design

#### Network layers

**This is a sketch of the networking layers:**

Actual packet sending layer:

This is responsible for transferring datapackets between pc’s

The Wizards comm layer:

This is responsible for the wizards specific communication. This will probably be application logic. It is responsible for linking remote connections to In game players/game clients, managing connected players and server state, connecting to server.

Communication layer. This layer provides a façade to communication functions. This allows implementing it using real Networking, or simply faking networking, which will come in handy during testing. This layer will provide interfaces that allow retrieval and sending of data between server/client

Idea for implementation of this layer:

There is an object that holds all types of packets and all packetbuilders on the client and the server. The server sends a list of packetbuilders and their packet ids on connection from the client. Packets can be send using this object, and there are functions that wait for a specific packet to arrive. (maybe multiple waits are possible)

#### Types of communication possible

Game data files (models, terrains, textures, scripts)

Game core files (exe updates etc)

Static entity locations, static object properties (or is this through files ?)

Dyanmic entity update packets, other dynamic object packets

Ping

Time synchronization

Player input: movement, spells

Player stats requests, items

Guild, of vereniging communicatie: taxes, economy, trade routes

Player chat, server notifications

Editor data: commits, custom dynamic updates.

#### Basic network flow

Implement this flow entirely on a different thread, so that this entire process can be written linearly, which will make everything extremely simple.

##### Main flow

Server starts running

Server listens for connections on TCP

Server receives TCP connection. Stores it as a client, with limited privileges

Server responds to limited privileges commands:

* Core files/executables update requests.
* Ping request
* Status/privileges request?

Server can receive a UDP authentication request. Server will send some random data to the client, which it has to send via UDP to the server within a limited time. The server will then store that UDPEndpoint as having the same privileges as the corresponding TCP connection

Server receives a login request/authentication. The server checks authentication data and if correct, changes privileges accordingly: client/editor/admin/…

Server processes **In-Game or editor flow.**

Server receives a logout request, privileges are changed, game state may be updated

TCP connection closes

##### In-Game or editor flow

In-Game:

Server sends Delta Packets via UDP at regular intervals

Server sends Time Packets at regular intervals

Server can send hit and contact info. (This can be delta packet)

Server can send bot/world state info. (This can be delta packet, but there is also data that hasn’t such high priority as in delta packets. Example: a bot walks from A to B. First the path needs to be send to the clients. Then the current position in the pad must be send. Since this is a very predictable movement, packets can be sent once per second instead of the high cost 20/second for delta packets.

Can receive player input.

Can receive **data requests**. This will have to be implemented per request (so no general system)(so a entity data request, a tree request etc.

#### Needed Classes

GameClient: is responsible for managing client privileges and connection state.

TODO: the NetworkPacketManager should have functionality to return an object that allows hard-typed sending and receiving of a type of packet.

### Systems

**Todo: merge this into the classes heading below**

#### NetworkPacketFactory Generater

TODO

#### NetworkPacketManager

Single packets: these are stored in a queue when received and can be retrieved one by one. Implemented by the IClientPacketTransporter<T>

Request packets: these are always in a one to one reply to a previous packet send.  
Implemented by the IClientPacketRequester<TSend,TReceive>

Multi-packets: when these packets arrive, all listeners to this type of packet are notified. EDIT: currently these packets are not allowed, because they are overly complex, and probably useless. Not implemented.

**Design:**

Sending/receiving a packet is ‘owned’ by an object. An object must request the manager for a transporter that sends/receives a type of packet. Different types of transporters are allowed (for starters: UDP/TCP, send-request packets). The send/receive logic is currently shared between the manager and the transporters.

#### TCP/UDP

TODO

### Classes

#### Basic Networking

* **TCPConnectionListener**  
  This class is responsible for accepting incoming TCP connections, and managing the listener socket.
* **TCPConnection**This class is responsible for managing the tcp socket, and for reconstructing the data packets from the tcp stream
* **TCPPacketBuilder**This class is responsible for getting individual packets from a stream or putting packets into a stream.

#### Abstract Packet Layer

These interface provide an abstract way for functionality classes to use remote-communication capabilities.

* **INetworkPacket**Defines a network packet which can be used by INetworkpacketFactory. This interface will most likely be empty.
* **INetworkPacketFactory**This interface defines a factory that is responsible for serializing an INetworkPacket. This will be for only one type of packet.  
  **NOTE: THIS INTERFACE IS FOR INTERNAL USE ONLY!**
* **INetworkPacketFactory<T>**This interface defines a factory that is responsible for serializing an INetworkPacket of type T for network transfer.  
  **TODO:** These network factories can be auto-generated, using attributes in the INetworkPackets. These classes should be created automatically and compiled into an assembly for optimal performance
* **IClientPacketTransporter<T>  
  TODO: Rename IPacketTransporter?; needs a ‘Queued’ in its name?**This interface provides functions to send/receive packets to/from a single remote. It can only send/receive packets of type T. This class **queues** all incoming packets, releasing one at a time when receive is called.  
  This class works in a synchronous way. Asynchronous methods are currently not provided, since it is easier to make the actual game logic asynchronous than the packet sending.  
  **IMPORTANT: THIS CLASS SHOULD BE THREAD-SAFE**
* **IClientPacketRequester<T>  
  Provides functions to send a request to a remote, and handle a request from a remote.**

#### Networked Packet Layer

This provides an implementation for the Abstract Packet Layer using TCP/UDP for communication. The core class here is the PacketManager.  
**This layer is not exactly Application Logic, but is very close to application logic (probably only use by application logic).**

Currently, the PacketManagers use **a string to identify remote transporters** with local transporters. (At first this was done using the PacketTypes, but this was rather confusing)

* **ServerPacketManagerNetworked**This class is responsible for holding active NetworkPacketFactory’s, holding ClientPacketManagers and passing the factories, and provides functionality for sending, loading to the clients, or to multiple clients?
* **IClientPacketTransporterNetworked**Works in conjuction with the ClientPacketManagerNetworked  
  TODO: this interface is also used by the requester. This should be renamed and changed to some kind of more uniform interface. **This interface is for internal use!**
* **ClientPacketTransporterNetworked<T>**This is an implementation of the IClientPacketTransporter and is used in conjunction with the ClientPacketManagerNetworked.  
  **WARNING: THIS CLASS IS USED BY THE MANAGER FOR LOCKING, SO ITS INSTNACES CANNOT BE USED IN ANY OTHER LOCKING MECHANISMS**
* **IServerPacketManager (REMOVE?: unnecessary)**  
  Provides methods to send synchronously and receive synchronously to multiple remotes.   
  See Also: IClientPacketManager
* **IClientPacketManager (REMOVE?: unnecessary)**  
  Provides methods to send synchronously and receive synchronously. Asynchronous methods are currently not provided, since it is easier to make the actual game logic asynchronous than the packet sending.  
  A IClientPacketManager is supposed to be connected to a remote IClientPacketManager, and packets are send to and received from this remote manager.
* **ClientPacketManagerNetworked**Provides an implementation of IClientPacketManager using the network.Is responsible for the identification of packets between 2 remotes.   
  Supports TCP and UDP packets. Provides methods to create PacketTransporters that can send/receive packets.   
  The manager will create a list of ClientPacketTransporterNetworked instances to send each packet.  
  **IMPORTANT: THIS CLASS SHOULD BE THREAD-SAFE  
  Note that ‘Client’ in the class name does not refer to this class being part of the client, but to the fact that this class is connected to One remote (the server).   
  WARNING: THIS CLASS USES THE INSTANCE OF THE TRANSPORTERS FOR LOCKING, SO THEY CANNOT BE USED IN ANY OTHER LOCKING MECHANISMS**
  + NOTE: the first version of this class will probably only support one listener for a packet type. Maybe more listeners for one packet are needed, this functionality can be added later on. This is also important for memalloc issues, since caching is hard
  + TODO: functionality should be added that a packet can be send and a reply for that Specific packet is waited for.

#### NetworkPacket Factory Generater

These classes/attributes provides functionality to auto-generate factories for network packets.

* **NetworkPacketFactoryCodeGenerater**This class is responsible for generating c# code for NetworkPacketFactory’s and compile them to an assembly. It is also responsible for providing those generated factories.   
  TODO: an attribute should be added that allows generating factories that allow a packet buffer: a number of packets is created, these are used when a packet is received (no memalloc) and the user must release these packets when finished.
* **INetworkPacketFactoryFactory**  
  This class is for internal use!
* **NetworkPacketAttribute**Works in conjunction with the NetworkPacketFactoryCodeGenerater. This attribute defines a class/struct as a NetworkPacket for which a factory can be generated.
* **NetworkPacketIngoreAttribute**Works in conjunction with the NetworkPacketFactoryCodeGenerater .This attribute tells the generater to ignore the given field/property in serialization.
* **INetworkPacketCustomSerializable**Works in conjunction with the NetworkPacketFactoryCodeGenerater. If a NetworkPacket implements this interface, it can provide additional serialization functionality, on top of the auto-generated functionality.

##### TODO

* implement more Types (currently only int and string)
* Implement recursion into subclasses/structures, serialize them too.

#### Unsorted

* **PacketFlags**Defines additional info on how the packet will be send to the remote.

#### Server

* **GameCientListener (AL)**  
  This class is responsible for managing incoming connections and assigning and creating new gameclients when they client. It is also responsible for initializing all App Logic for connections, like initializing the UDP conn.  
  This seems to be application logic.
* **GameClient (AL)**  
  This class manages all Network connections items and privileges items.

#### Client

* **GameClient (AL)**  
  This is the counterpart of the Server’s GameClient class (and has the same name, hopefully not confusing).

## Texture Library

* Collecting of leaf Pictures
* Collecting of Grass/bushe Pictures
* Putting alpha on all of them
* Perhaps later making an easy browsing tool in combination with theTexture AtlasTool