Amphibian Classes

This document contains all of the classes seen in the uml class diagram and their content. This was done in order to keep the class diagram as simple as possible

# Core Classes

## Class Issue

This is an internal class which is used to simplify the use of issue lists

* Public String name
* Name of the issue (single or folder)
* Default internal constructor
* Default internal destructor
* Private IssueHistory history
* A class variable which is used to track the changes within the individual issues and issue lists, this will come in handy during the activity logging

## Class IssueList : Issue

A derived class used for folder simulation within the code. This class contains

* Public List<Issue> subIssues
* Used to create „folders“ within the database itself
* Default Public constructor
* Default Public destructor
* subIssue add and removal methods

## Class IssueSingle : Issue

A class which represents a single issue. This class contains

* Public User creator
* A variable pointing towards the one who created the issue
* Public Bool private
* Exploits cannot be revealed to the users for security reasons, if a user reports one of these and the devs do not know it, it should be possible to hide it from other users to prevent mass abuse and still track the issue.
* Public DateTime creationTime
* A variable used to track when the issue was created
* Public DateTime editTime
* A variable used to track when the issue was edited
* Public string description
* Used to describe the issue itself
* Public List<RelatedIssue> related
* List of special classes used to describe which issue is related to this and in what way
* Related issue add and removal methods
* Default public constructor
* Default public destructor
* Public List<Comment> comments
* List which contains comments (class Comment)
* Comment add and removal methods
* Edit method

As you can see these classes are used to represent folders or individual issues/reports. The class IssueList will be used for navigating through the different types of issues and the class IssueSingle represents the individual reports and all of its comments

## Class Comment

This is an internal class which contains the following things

* Internal constructor and destructor
* Public IssueSingle origin
* The Issue to which the comment is related
* Public User creator
* To point to the one who created the comment
* Public string content
* The comment itself
* Public DateTime creationTime
* When the comment was posted

## Class ProtectedComment : Comment

A class which stores commit messages only, not possible to edit. Can only be seen by Devs

* Public constructor and destructor
* Method for updating the CommitLog (class) when one of these is created

## Class EditableComment : Comment

Normal comments which can be edited by the creator of the comment, all QA members and all developers. Can be private (notes) and public (discussion comments)

* Public constructor and destructor
* Public bool Private
* To track if the comment is a note or a normal comment
* Public CommentHistory history
* A container class which holds previous versions of the comment, the date is not tracked because the list of previous versions is ordered already

The comment classes are forms of tracking user discussions on the given Issue/Report and tracking commits on the given issue. The commits cannot be edited and are just copied by the devs from their actual code repository and a note if they wish (serves for tracking progress of individual issues).

## Class Person

An internal class which just contains personal data about the user, the rest of the classes have methods related to the access level the user has to the repository or system

* Internal constructor and destructor
* Public String Name
* Public String LoginToken
* It is not safe to store the password in the class itself, which is because the login token is used as a key to access the database and pull the password hash, which is compared to the hashed password the user inputs. This has nothing to do with the system itself, the login can be done on a web site

## Class ProjectManager : Person

This is just the class representing the CEO, you might notice that he cannot comment or create issues, which is because this class is used for administration purposes. It has nothing in it and serves only to check if the user is the CEO, then the form will open up the CEO features on itself

* Public Constructor and Destructor

## Class User : Person

This is just the class representing the normal Users, the Project Manager and Users are in different directions because any derived class from user will be able to comment. It has no purpose other than providing info that the given person is an ordinary user. (If we exclude the fact that other classes are pointing towards it)

* Public Constructor and Destructor
* Public List<MonthlyPoints> points
* Points which are distributed to the user for activity, report quality, work quality and such things, the class will be described later on in its own section

## Class QualityAssurance : User

This is just the class representing the tester (Quality Assurance). It is derived from User because the qa can do the same things as an user, with the addition of being able to edit other people's comments and seeing private notes. It has no purpose other than providing info that the given person is a QA. All of the mentioned features are implemented in the actual GUI (Graphic User Interface) where it decides to hide/show features based on what type of user is using it

* Public Constructor and Destructor

## Class Developer : QualityAssurance

This is the class representing the Developers, which have all of the features the QA has + access to the protected comments, or commits, how it works has already been described

* Public Constructor and Destructor

As described in the classes, the Person tree is only there to provide info about the user (name, login token and type of user), the features which that user can use are managed in the GUI

This is the core section, it will only communicate with the Shell section which are extensions, containers and lists. The only exception to that is the Person tree

# Shell Classes

## Class HistoryObject

Internal class used to group things together for the ActivityLog class, for now two things are tracked: Issues and Comments

* Internal Constructor and Destructor
* Public virtual list item add and removal methods
* Purely virtual, their implementation is in the derived classes

## Class IssueHistory : HistoryObject

* Public Constructor and Destructor
* Public List<Issue> history
* A container for all of the issues, even the “deleted ones” are saved in here. When the list reaches maximum capacity the first half of it is stored somewhere externally and is freed for future use
* Public methods saveData and clearData
* As mentioned used to store the data somewhere outside of the program and clear the list itself for future use
* Public method addItem
* Adds an item to the history, this is called on each successful edit or creation in the GUI. It is not called when comments are edited
* Public method getHistory (DateTime Start, DateTime End) -> List<Issue>
* Gets a list of all the issues created between the two dates. It was not mentioned but each edit stores a new copy of the issue whose creation and edit date are the edit date. The original is just updated. So the GUI only has to check the creation dates, not the edit dates
* Public method getHistory (DateTime Start, DateTime End, User creator) -> List<Issue>
* Gets a list of all the issues in the interval given created by the specified user. The default values of the interval are: from January 1st 1900 to Today, which would give all of the issues created by the user

## Class CommentHistory : HistoryObject

* Public Constructor and Destructor
* Public List<EditableComment> history
* A container for all of the editable comments, even the “deleted ones” are saved in here. When the list reaches maximum capacity the first half of it is stored somewhere externally and is freed for future use
* Public methods saveData and clearData
* As mentioned used to store the data somewhere outside of the program and clear the list itself for future use
* Public method addItem
* Adds an item to the history, this is called on each successful edit or creation in the GUI. It is not called when comments are edited
* Public method getHistory (DateTime Start, DateTime End) -> List<Comment>
* Gets a list of all the editable comments created between the two dates. It was not mentioned but each edit stores a new copy of the issue whose creation and edit date are the edit date. The original is just updated. So the GUI only has to check the creation dates, not the edit dates
* Public method getHistory (DateTime Start, DateTime End, User creator) -> List< Comment >
* Gets a list of all the editable comments in the interval given created by the specified user. The default values of the interval are: from January 1st 1900 to Today, which would give all of the issues created by the user

Now, the reason why CommitLog is not included is because that is already on the repository where the code is (github eg). The CommitLog is just a separate tracking service the developers can but do not have to use, and only they can see it

## Class CommitLog

* Public constructor and destructor
* Public list<ProtectedComment>
* The list of the commits, once it reaches maximum capacity the data is saved somewhere outside of the system and half of the list is cleared for future use
* Public methods saveData and clearData
* As mentioned used to store the data somewhere outside of the program and clear the list itself for future use
* Public method addItem
* Adds an item to the log, this is called on each successful edit or creation in the GUI. It is not called when comments are edited

## Class IssueContainer

A container class which has all of the issues in it and fills the role of a search engine

* Public constructor and destructor
* Public method addIssue
* Called by the GUI when a new issues is created, only new issues, not edited ones are added
* Public method delIssue
* Called when an issue is deleted by using the GUI
* Public method search (parameters unknown at this point) -> List<IssueSingle>
* The parameters have not been worked out at this point, there will probably be a list of all users and the filter will be something like the html string filets used on sites. Eg “?user=135&contains=hardware”

## Class ActivityLog

A container class of all the History objects created so far with methods used to display them. Most of these methods have already been used

* Public constructor and destructor
* Public List<HistoryObject> log
* Public method giveData (DateTime start, Datetime end) -> List<HistoryObject>
* This also checks if the players is actually eligible to see the given comment, only the creator, QA and Devs can see private issues and comments
* Public methods addItem and removeItem

## Class MonthlyPoints

Multiple classes linked to each user which serve as a grading system for employees and normal users. This feature can be used by the Project manager to calculate the salary of his employees and even employ regular users if they gather enough points

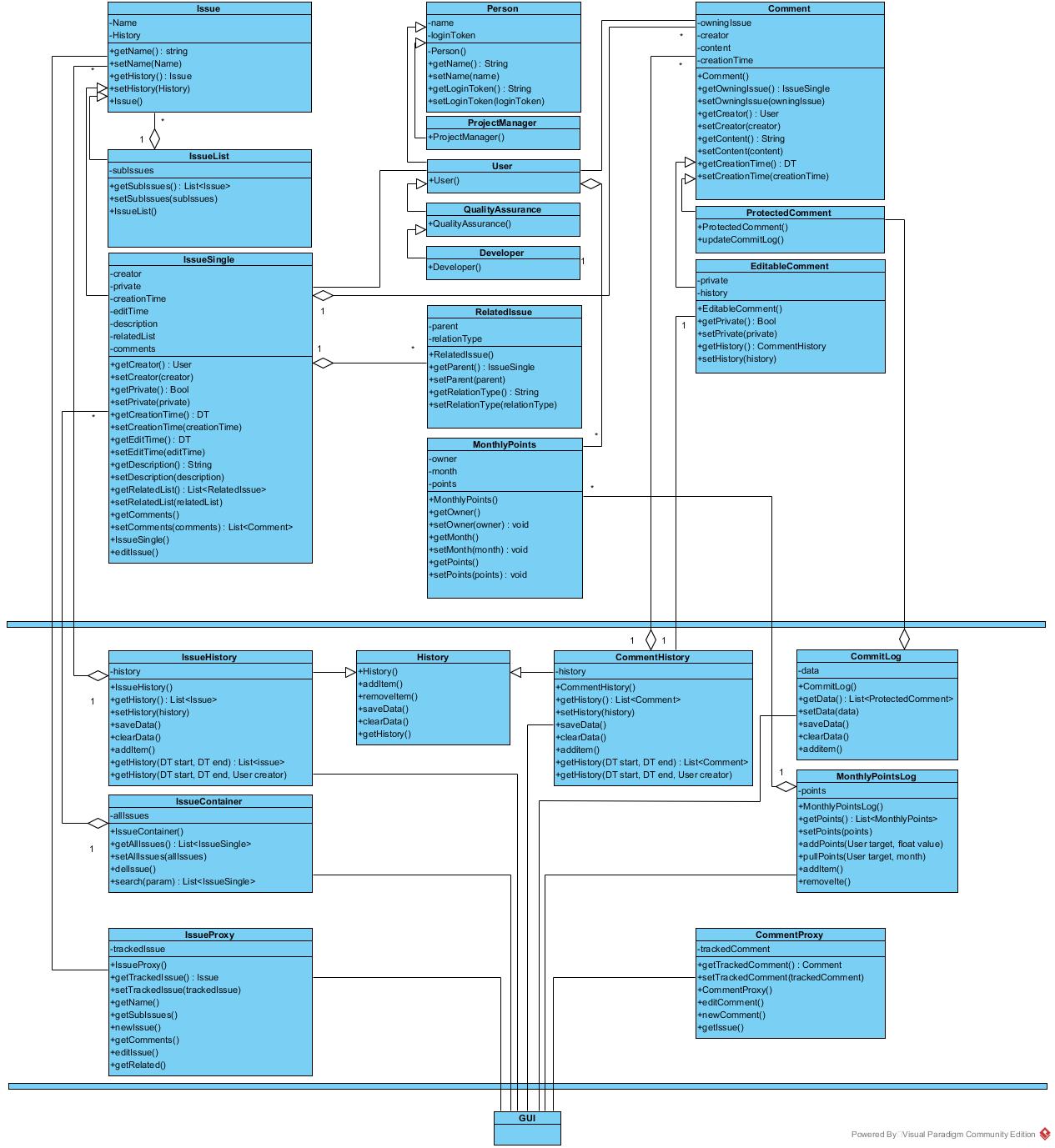
* Public constructor and destructor
* Public User pointOwner
* Public DateTime month
* Public Float points
* Public method addPoints
* Negative values are also accepted

## Class PointLog

A container class for all of the MonthlyPoints created which is also used to manage the said classes

* Public constructor and destructor
* Public list<MonthlyPoints> points
* Public method addPoints (User target, Float value)
* Used to add points to a specific user, calls the appropriate classes method addPoints. This is used by the GUI options available only to the ProjectManager
* Public method pullPoints (User target, Datetime month) -> List<MonthlyPoints>
* If a User is specified it will return his points of the given month, if not, then all points in the given month are pulled
* Public methods addItem and removeItem
* Used to add/remove point classes from the list

Those would be all of the shell classes. The shell is the mid layer, communicating directly with the core and the GUI. The GUI does not directly communicate with the core because of this and the interaction of the Shell and the GUI will be described in another diagram. For now, let’s look at the visual representation of the classes mentioned in the core and shell. The methods and attributes have not been listed as they have already been described



The same image is located in the folder in which this document is under the name “Class Diagram v2.jpg”

# GUI Classes and Forms

To avoid creating new classes or using the core ones the GUI will use proxy classes which are pointing towards the Issues and Comments, the Person tree is the only one directly accessed by the user interface. As said, the proxy classes are

## Class IssueProxy

In order for this to work as intended, a predefined IssueList has to exist which is named after the monitored project, the name of the variable is globalIssueList and it is used to track all of the folders and issues in the given project. Lets say the user of Amphibian is a gaming company, globalIssueList will be named after the tracked game and it will contain all of the issues related to that game

* Public constructor and destructor
* Private Issue viewedIssue
* Public Constructor: public IssueProxy (Issue vi)
* If vi is specified then viewedIssue becomes that, if not, viewedIssue becomes the globalIssueList
* Public method getName
* Returns the name of the viewed issue
* Public method getSubIssues
* Gets the issues contained in IssueList if that is the tracked Issue, if not, will return a null value
* Public method newIssue
* Will check if the Person calling is of type User or derived from it. If that is the case the function checks if viewedIssue is of type IssueList, if so it will show the form for creating a new issue
* Public method getComments(int page) -> List<CommentProxy>
* First checks if viewedIssue is of type SingleIssue and if that is the case it will get a list of comment proxies.
* Depending on the given integer it will give different sets of comments, for example if page==1 then the first 10 comments in that issue are displayed, if page ==3 then comments 21-30 are displayed
* Public method editIssue
* Checks if the Person meets the criteria and opens the edit form. When done it will first save the viewedIssue into the history and then replace the data with the ones in the edit form
* Public method getRelated -> List<SingleIssue>
* Gets the related issues to this one

## Class CommentProxy

Same as the IssueProxy but for comments, no need to type down the methods as you can probably assume which ones are in here

The proxies are sort of the control part of the system which transfers the orders from the forms to the shell and therefore influences that and the core, now for the forms, I will just quickly go over them

## Form NewIssue

* Just a form used to input data about the new issue, it is called by the IssueProxy and till this form is closed the thread of the IssueProxy will wait. When it is closed the data is verified and the IssueProxy saves the data
* To simplify things the same form can be used to edit, just add a Boolean to the constructor

## Form NewComment

* Same as NewIssue but uses the CommentProxy

## Form SearchForm

* Contains filters and the result datagridview. When the search button is pressed the form accesses the class IssueContainer and uses its methods to get the issue list which matches the filters, then displays that in the DataGridView
* When the users selects an item in the DataGridView the IssueProxy is set to that one and the form ViewIssue is opened

## Form ViewIssue

* Opens up the proxy specified in a separate form with all of its commets (commentProxies) and related issues
* Contains buttons used to navigate through the comments and edit/view the subtasks and the open issue

## Form MainForm

* Shows features based on which type of user is currently viewing it, multiple versions of this form can be open at a time and they contain links towards the search form and the ViewIssue form (when this is accessed it will open up the last viewed IssueProxy)

# SOLID Object Oriented Design Principles

## Single Responsibility Principle

This principle states that a class should have only one single responsibility, which as we see is present in the current classes. Each of the core classes serves only to store some basic data related to the object. The additional data and operations over those classes are implemented in other classes linked to those

## Open/Closed Prinpciple

Software should be open for extension but closed for modification has kind of been fulfilled. If we want to add new features the current classes have been created in a way which allows that without having to modify the code too much. If we want a new user type it can be attacked to the Person Tree, a new comment type, it can be attached to either of the derived comment classes. If we want to build in a login system it can be attached to the GUI or the person class can have an interface which does that

## Liskov Substitution Principle

The only case where this is not possible is if you want to switch between the Project Manager and the Users, because users can comment, create reports etc and the PM cannot, the pm can do some things the ussers cannot. This is not managed within those classes themselves