**iBudget**

**Software Design**

**Document**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Approvals: | L. Assayah | \_\_\_\_\_\_\_\_\_\_\_\_\_ | date: | \_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | C. Leung | \_\_\_\_\_\_\_\_\_\_\_\_\_ | date: | \_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | Q. Pham | \_\_\_\_\_\_\_\_\_\_\_\_\_ | date: | \_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | V. Velev | \_\_\_\_\_\_\_\_\_\_\_\_\_ | date: | \_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | J. Reimels | \_\_\_\_\_\_\_\_\_\_\_\_\_ | date: | \_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Author** | **Version** | **Reason** |
| 3/13/12 | V.Velev | 1.0 | First Draft |
|  |  |  |  |

# Introduction

## Purpose

This document describes the design of the *iBudget* personal finance software. It shows how the software system will be structured to satisfy the requirements identiﬁed in the software requirements speciﬁcation.

## Scope

This design is intended for the initial version of *iBudget*. It is intended as the basis for other versions of the software in the future.

## Definitions, acronyms and abbreviations

API: Application Programming Interface – a way for the programmer to interact with the system hardware

DBMS: Database Management System

GUI: Graphical User Interface

etc.

## References

[1] Role-playing video game SDD

[2] *Encounter* video game SDD

[3] IEEE Std 1016-1998 IEEE Recommended Practice for Software Design Descriptions

# System Architecture

## Architecture design

Top-level decomposition: the application consists of the client side, server side and back-end modules – a typical three tier client-server architecture where one server serves many clients. The top tier is the graphical user interface developed with HTML. The bottom tier is the database layer, developed through the use of MySQL that will hold all transaction, account and category information. The middle layer, or middleware, is developed using PHP, and issues functionality between the user interface and the database such as adding, removing, and editing transactions, accounts and categories.

The code structure is to use the MVC architecture pattern. Use of the MVC pattern results in separating the different aspects of the application (input logic, business logic, and GUI logic), while providing a loose coupling between these elements. The MVC was chosen as it simplifies the architecture by decoupling models and views, and to makes source code more flexible and maintainable and it maps nicely to the three-tier architecture as well.

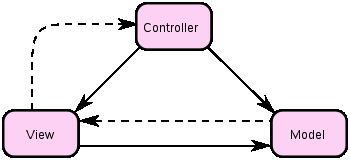


Figure 1 Model–view–controller concept.

The solid line represents a direct association, the dashed an indirect association

(via an observer for example)

The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). The model is not necessarily merely a database; the 'model' in MVC is both the data and the business/domain logic needed to manipulate the data.

The view renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.

The controller receives user input and initiates a response by making calls on model objects. A controller accepts input from the user and instructs the model and a view port to perform actions based on that input.

## Design pattern

## Factory

The factory pattern creates objects without exposing the instantiation logic to the client and refers to the newly created object through a common interface. The factory pattern is being used... *(please provide examples where and how it is being used)*

## Singleton

The singleton pattern ensures that there is exactly one instance of a given class and that it is accessible from anywhere in the application. The singleton pattern is being used in the *iBudget* application to restrict the number of database connections to only one.

Indeed it is used with all the source objects that are in the source folder. For all these files, the constructor is made private and only called once to create the object the one time. All other attempts to create another instance of the same object will result with getting the instance created before. A static attribute called $instance is created when calling for the first time the getSource() method. Then every time the method is called again, the same instance is returned. Below is an example of how it is implemented in categorysource.inc :

Attribute:

**private static** $instance = **null**;

Method:

**public static function** *getSource* () {

**if** (**self**::*$instance* == **null**) {

**self**::*$instance* = **new CategorySource();**

}

Constructor:

**private function** \_\_construct () {

**/\*content \*/**

}

And an example of how it is used in category.inc by calling the getSource() method:

**protected function** delete () {

$source = CategorySource::*getSource*();

$source->delete(**array**('ID' => **$this**->id));

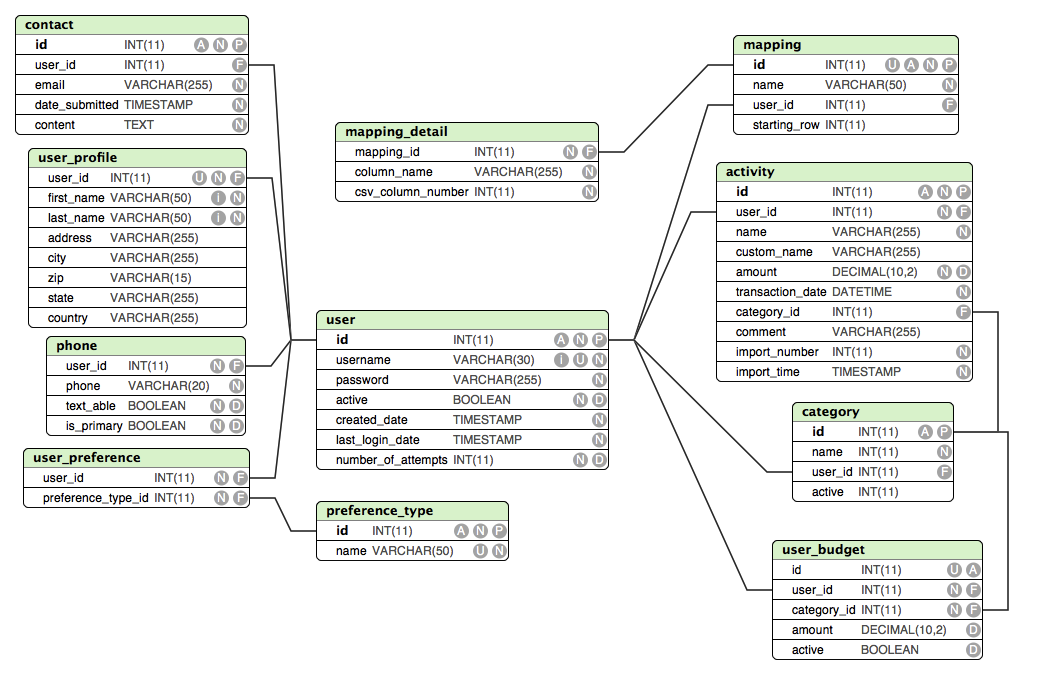
}

## More???

# Data diagram

## Data description

( describe how data is stored, processed, and organized)



## Data dictionary

(functions and parameters)

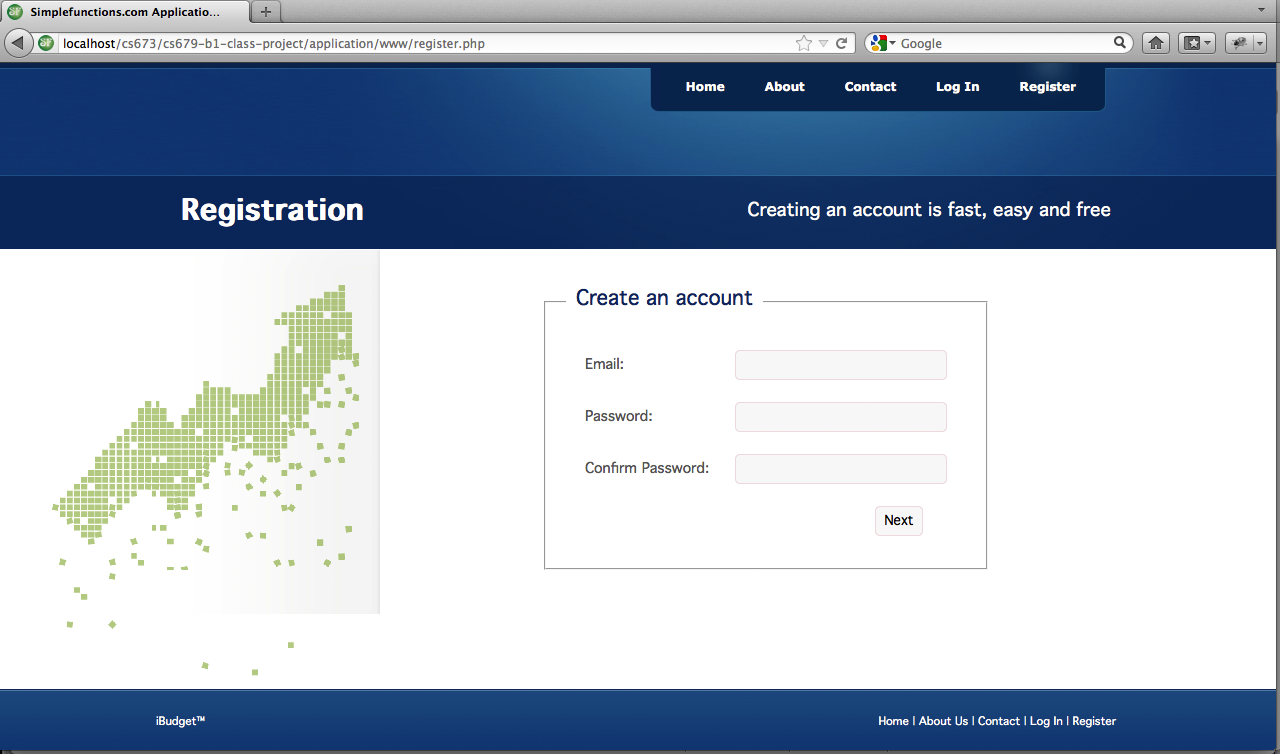
(source obj.)

# Human interface design

## Registration

## Overview

## Screenshot



## State diagram

## Login

## Overview

## Screenshot

## A description...

## State diagram

## Process CSV

## Overview

## Screenshot

## State diagram

## Contact

## Overview

## Screenshot

## State diagram

## Forgot password

## Overview

## Screenshot

## A description...

## State diagram

## Dashboard

## Overview

## Screenshot

## State diagram

## Custom mapping

## Overview

## Screenshot

## State diagram

# 5. Requirements Matrix???