

E-Commerce++

University of Urbino “Carlo Bo”
Information Science and Technology institute
Applied Computer Science
Worldwide Distance Degree Program

Software Engineering
Professor: Edoardo Bontà

Table of Contents

Problem specifications	5
Data model	7
Translation logical scheme	8
Normalization	8
Requirements analysis	10
Unprivileged user: use cases diagram	10
Unprivileged user: use cases list	10
<i>Use case #1: login (UC1)</i>	10
<i>Use case #2: register (UC2)</i>	11
<i>Use case #3: quit (UC3)</i>	11
<i>Use case #4: place a new order (UC4)</i>	11
<i>Use case #5: browse by category (UC5)</i>	12
<i>Use case #6: view product detail (UC6)</i>	12
<i>Use case #7: browse user orders (UC7)</i>	12
<i>Use case #8: show product configurations (UC8)</i>	12
Admin user: use cases diagram	13
Admin user: use cases list	13
<i>Use case #1: login (AUC1)</i>	13
<i>Use case #2: trend monthly sales (AUC2)</i>	14
<i>Use case #3: add new product (AUC3)</i>	14
<i>Use case #4: add new category (AUC4)</i>	14
<i>Use case #5: disable user (AUC5)</i>	14
<i>Use case #6: change product details (AUC6)</i>	15
<i>Use case #7: product delete (AUC7)</i>	15
Problem analysis and design	16
Robustness analysis	16
Class diagram	16
<i>Class Observable</i>	19
<i>Class Observer</i>	19
<i>Class Singleton (template)</i>	19
<i>Class Database (singleton)</i>	20
<i>Class ManagedObject</i>	20
<i>Class DataModel (singleton)</i>	22

Class User	22
Class AdminUser	23
Class NormalUser	23
Class Product and ProductProxy (virtual proxy)	24
Class Category	24
Class Order.....	25
Class Basket.....	25
Class CommandLine.....	26
Class UserMenu	26
Class BadAuthException and InvalidArgument (exceptions)	27
Sequence diagram.....	27
State diagram.....	28
Implementation	31
Application.....	31
common.h.....	31
Exceptions.h.....	32
Observer.h.....	33
Observable.h	33
Observable.cpp.....	34
Database.h.....	36
Database.cpp.....	37
DataModel.h	40
DataModel.cpp.....	40
ManagedObject.h.....	41
ManagedObject.cpp	42
User.h.....	48
User.cpp	49
Basket.h	55
Basket.cpp	55
Category.h.....	58
Category.cpp	58
Product.h	60
Product.cpp.....	62
Order.h	67
Order.cpp.....	67
CommandLine.h	70
CommandLine.cpp.....	71

<i>UserMenu.h</i>	73
<i>UserMenu.cpp</i>	74
<i>main.cpp</i>	84
<i>Makefile</i>	85
White box test	86
<i>white-box.h</i>	86
<i>white-box.cpp</i>	86
Testing	88
Unit testing	89
<i>Key/value observing</i>	89
<i>Database and DataModel functionalities</i>	90
Validation testing	90
<i>Test case #1: database connection</i>	91
<i>Test case #2: login</i>	91
<i>Test case #3: register</i>	92
<i>Test case #4: quit</i>	92
<i>Test case #5: place a new order</i>	93
<i>Test case #6: browse by category</i>	94
<i>Test case #7: view product detail</i>	95
<i>Test case #8: browse user orders</i>	95
<i>Test case #9: show product configurations</i>	96
<i>Test case #10: login</i>	96
<i>Test case #11: trend monthly sales</i>	97
<i>Test case #12: add new product</i>	97
<i>Test case #13: add new category</i>	98
<i>Test case #14: disable user</i>	98
<i>Test case #15: change product details</i>	99
<i>Test case #16: delete product</i>	100
Coding standards	101
Develop environment	102
Compile instructions	102
Bibliography.....	104

Problem specifications

The aim of this project is to realize an e-commerce tool, along with its database, for the final client who wishes to buy some product.

The first need is classifying technology products, such as computers, monitors, accessories and so on. To make products browsing easier, it is advisable to split items into categories, so that customers will be able to find what they're looking for faster and easier; for each product it is known the id, the name and its description, as far as its price together with its availability.

Whenever a potential customer chooses an item from catalogue, the system should suggest a list of configurations containing the selected product: for example, if a laptop is chosen, the system will propose to buy some kind of mice or PCCARD which belong to the same configuration.

Only registered users are allowed to browse and buy items: for each user, it's known the id, name and surname, together with his credential to access the system and a shipping address. A user must choose a unique login to be correctly registered.

A discount to total amount of customer order should be considered: total amount of an order could be different from the sum of each item.

The system should allow the system administrator to delete an item from the catalogue: this operation must be permitted only under the condition that the selected item is not part of any previous orders, otherwise there must exist another way to hide the product from catalogue browsing and leave old orders consistent.

As the user runs the tool, he can register himself as a new user or log into the system by specifying his own credential.

The interface must implement the following operations to read data:

- Browsing product catalog by category;
- Viewing details of the chosen product;
- Browsing all the configurations which include a given item;
- Browsing user's profile and all the orders he made;
- Placing a new order.

The system should let the administrator issue the following operations, which may also alter data:

- Adding a new category;
- Adding a new product;

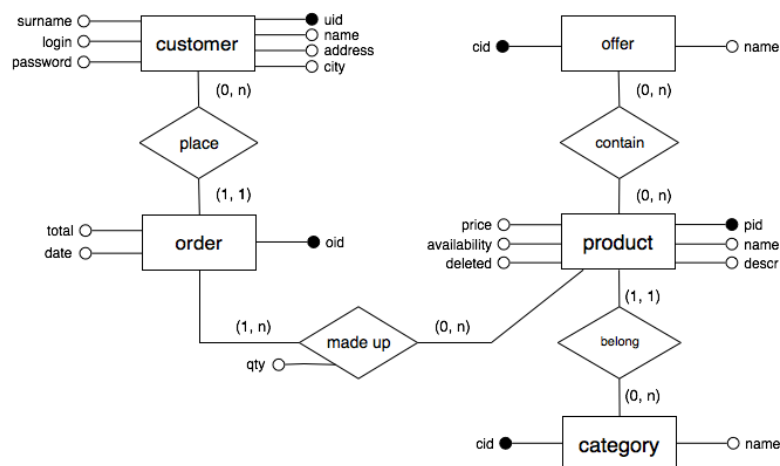
- Browsing proceeds of sales, grouped by month;
- Preventing a user from log in (lock);
- Deleting a product (even if it was already sold).

Data model

Building on the analysis of information collected in the problem specifications, we will go on with the formal identification of the database structure by using the ER primitives. We must ensure that the produced schema is:

- *Correct*: the scheme can be defined as correct if appropriately constructs of the ER model are used ;
- *Complete*: the conceptual framework must represent all the data of interest and operations that can be performed starting from the concepts described;
- *Readable*: the conceptual framework must be, as a whole, readable and self explanatory;
- *Minimum*: all specifications collected must be represented only once, avoiding redundancy.

In this scheme, we can proceed with the definition of attributes for both entities and associations, and the information of cardinality and keys.



Among all attributes, the one marked represents the primary key for the entity.

The database will also impose the following constraints that cannot be expressed by the constructs of the ER model.

ID	CONSTRAINT DESCRIPTION
v1	You cannot register more than one user with the same <i>login</i> .
v2	You cannot enter more than one item in the catalog with the same name.
v3	You cannot enter more than one category with the same name.
v4	An item can be purchased only if its availability is greater than zero.
v5	An article may be removed from the database only if it has been ever purchased in the past (i.e. if it was placed in at least one order).
v6	The total amount of an order may differ from the sum of the cost of individual items (discount).

TRANSLATION LOGICAL SCHEME

We consider the *relational model* as a tool for the translation into a relational scheme; the resulting scheme is the following¹:

CATEGORIES(**cid**, name)
PRODUCTS(**pid**, *cid*, name, descr, price, availability, deleted)
OFFERS(**oid**, name)
CONFIGURATIONS(**oid**, *pid*)
USERS(**uid**, name, surname, login, password, address, city)
ORDERS(**oid**, *uid*, date, total)
ORDER_DETAIL(**oid**, *pid*, qty)

the following restructuring has been applied:

1. The relation *madeup*, between entities *order* and *product*, is translated in the scheme *order_details*;
2. The relation *belong*, between entities *product* and *category*, is absorbed by the attributed *cid* of scheme *products*;
3. The relation *contain*, between entities *offer* and *product*, is translated in the scheme *configurations*;
4. The entity *customer* is translated in the scheme *users*.

To increase the readability of the queries, it was decided to create the following *views*:

1. AVAILABLE_PRODUCTS: displays the list of products not deleted² and having a positive stock availability;
2. HANDLED_ORDERS: displays the list of orders placed by customers and their composition in term of purchased items;
3. CATALOGUE: displays the list of products available for sale, including the belonging category.

Moreover some procedures have been developed in order to implement the following operations:

1. DELETE_PRODUCT: the procedure implements the delete operation of a product;
2. OFFERS_BY_PRODUCT: identified a product from the catalog, the procedure displays the list of complete offers containing the selected article and their relative composition.

NORMALIZATION

We analyze the relational schemes we obtained after translation; each table represents one and only one relation and has no repeating group: this condition satisfies the 1NF definition.

¹ The attributes marked in **bold** are the primary keys, those in italics the foreign keys.

² Please note that, as requirement, it is not permitted to delete a product if it is present in an handled order: in this case it should be only marked as deleted but not physically removed.

It turns out that all schemas with a single-value primary key are also in 2NF, as there cannot be attributes that functional depend on a proper subset of the key; anyway we have to analyze the schemas *configurations* and *order_details*, as they have a key with more than one attribute: the former has no non-prime attributes, while the latter presents the attribute *quantity* which functional depends only on the whole key.

There are no transitive functional dependencies, i.e. every non-prime attributes depend on the key of the table and not on any other subset of attributes: the schemas also respect the 3NF.

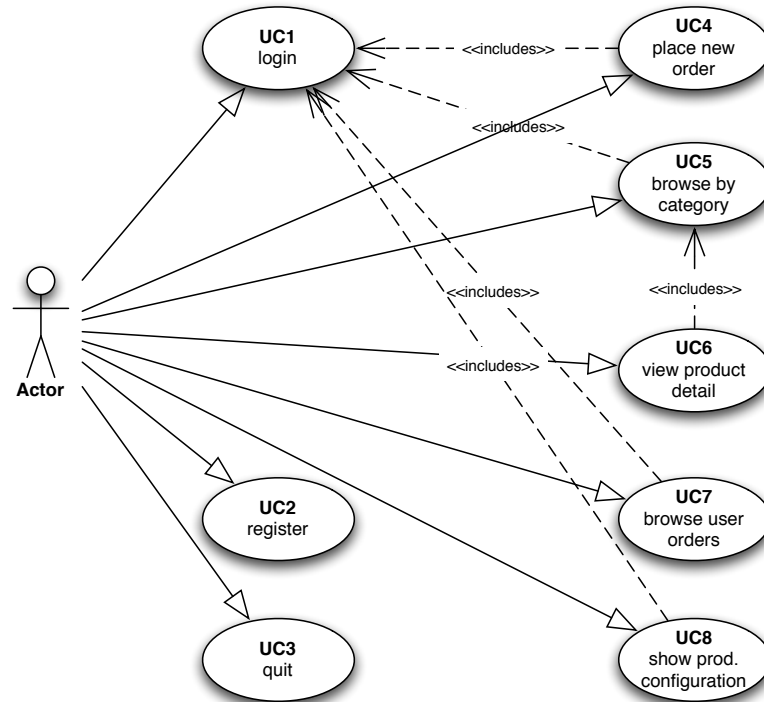
All the non-trivial functional dependencies are a dependency on a *superkey*³: this condition satisfies the BCNF requirements.

³ Informally, a superkey is a set of columns within a table whose values can be used to uniquely identify a row.

Requirements analysis

The objective of requirement analysis is to specify with higher detail level the functionalities of the system by means of UML Use Cases. There will be described two groups of use cases, one acting as a normal user and a second as administrator of the system; for each identified operation, a description of it, along with all preconditions and steps, is detailed.

UNPRIVILEGED USER: USE CASES DIAGRAM



UNPRIVILEGED USER: USE CASES LIST

Use case #1: login (UC1)

<i>Actor</i>	A1
<i>Preconditions</i>	<ol style="list-style-type: none">1. A1 runs the program;2. System connects to database;3. Main menu is shown.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 selected to log in;2. Login form is displayed;3. A1 enters his credentials;4. System verifies that user is registered and entered credentials are valid.
<i>Post conditions</i>	Administration (or user) menu is displayed.
<i>Alternative Actions</i>	<ul style="list-style-type: none">• Database connection failed: program reports the error and immediately quits;• A1 chooses to abort the operation.

Use case #2: register (UC2)

<i>Actor</i>	A1
<i>Preconditions</i>	<ol style="list-style-type: none">1. A1 runs the program;2. System connects to database;3. Main menu is shown.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 selects to register;2. Registration form is displayed;3. A1 enters his data;4. System stores user data and create a new record in the database.
<i>Post conditions</i>	<ol style="list-style-type: none">1. A new user has been registered;2. Main menu is shown.
<i>Alternative Actions</i>	Database connection failed: program reports the error and immediately quits.

Use case #3: quit (UC3)

<i>Actor</i>	A1
<i>Preconditions</i>	<ol style="list-style-type: none">1. A1 runs the program;2. System connects to database;3. Main menu is shown.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 chooses to quit;2. System quits.
<i>Post conditions</i>	
<i>Alternative Actions</i>	Database connection failed: program reports the error and immediately quits.

Use case #4: place a new order (UC4)

<i>Actor</i>	A1
<i>Preconditions</i>	User successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 chooses to place a new order;2. The system shows the product catalog and asks for the product ID to be entered;3. A1 enters the desired product ID;4. The system asks for quantity;5. A1 enters the desired quantity of chosen product;6. A1 confirms the order;7. The system places the order.
<i>Post conditions</i>	<ol style="list-style-type: none">1. A new order has been created;2. User menu is displayed to user.
<i>Alternative Actions</i>	The operation may be aborted if: <ul style="list-style-type: none">• A1 entered an invalid product ID;• A1 asked for a quantity greater than availability.

Use case #5: browse by category (UC5)

<i>Actor</i>	A1
<i>Preconditions</i>	User successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 chooses to browse user catalog;2. The system lists all available categories;3. A1 enters the desired category ID or chooses to browse all products;4. The system searches for all products belonging to chosen category;5. The system shows the result.
<i>Post conditions</i>	User menu is displayed to user.
<i>Alternative Actions</i>	

Use case #6: view product detail (UC6)

<i>Actor</i>	A1
<i>Preconditions</i>	User successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 chooses to view details of a specific product;2. The system show the list of all available categories;3. A1 chooses a specific category;4. The system shows all available products;5. A1 chooses a product ID from the list;6. The system shows all product details.
<i>Post conditions</i>	User menu is displayed to user.
<i>Alternative Actions</i>	A1 entered an invalid product ID and no detail is shown.

Use case #7: browse user orders (UC7)

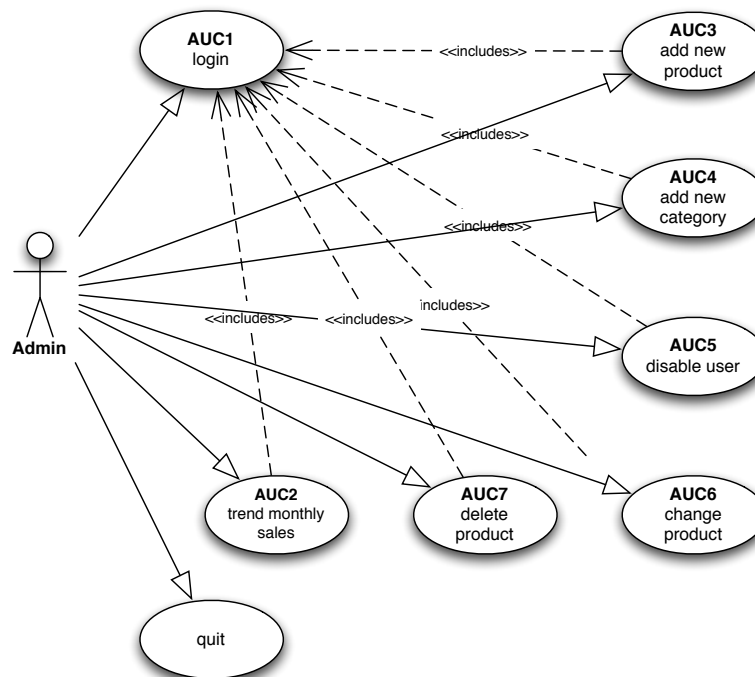
<i>Actor</i>	A1
<i>Preconditions</i>	User successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none">1. A1 chooses to see his personal profile and handled orders2. The system shows customer personal data;3. The system lists all order the customer placed until.
<i>Post conditions</i>	User menu is displayed to user.
<i>Alternative Actions</i>	

Use case #8: show product configurations (UC8)

<i>Actor</i>	A1
<i>Preconditions</i>	User successfully logged in.

<i>Basic course of events</i>	<ol style="list-style-type: none"> 1. A1 chooses to browse all configurations which include a specific product. 2. The system shows all available products and asks for a product ID; 3. The user enters the ID of wished product; 4. The system displays all pre-built configurations which include the chosen product.
<i>Post conditions</i>	User menu is displayed to user.
<i>Alternative Actions</i>	

ADMIN USER: USE CASES DIAGRAM



ADMIN USER: USE CASES LIST

Use case #1: login (AUC1)

<i>Actor</i>	A2
<i>Preconditions</i>	<ol style="list-style-type: none"> 1. A2 runs the program; 2. System connects to database; 3. Main menu is shown.
<i>Basic course of events</i>	<ol style="list-style-type: none"> 1. A2 selected to log in; 2. Login form is displayed; 3. A2 enters his credentials; 4. System verifies that user is registered and entered credentials are valid.
<i>Post conditions</i>	Administration menu is displayed.
<i>Alternative Actions</i>	<ul style="list-style-type: none"> • Database connection failed: program reports the error and immediately quits;

	<ul style="list-style-type: none"> • A2 chooses to abort the operation.
--	----------------------------------------------------------------------------------------

Use case #2: trend monthly sales (AUC2)

<i>Actor</i>	A2
<i>Preconditions</i>	Administrator successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none"> 1. A2 chooses to display trend of sales; 2. System displays the monthly trend of sales.
<i>Post conditions</i>	Administration menu is displayed.
<i>Alternative Actions</i>	

Use case #3: add new product (AUC3)

<i>Actor</i>	A2
<i>Preconditions</i>	Administrator successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none"> 1. A2 chooses to add a new product; 2. System shows the available category list; 3. A2 is asked to choose one category; 4. System starts asking new product attributes, such as name, description, price and availability; 5. System stores the new product.
<i>Post conditions</i>	<ol style="list-style-type: none"> 1. A new product has been created; 2. Administration menu is displayed.
<i>Alternative Actions</i>	A2 aborted the whole operation.

Use case #4: add new category (AUC4)

<i>Actor</i>	A2
<i>Preconditions</i>	Administrator successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none"> 1. A2 chooses to add a new category of products; 2. System asks for the name of new category; 3. A2 enters the category name; 4. System stores the new category.
<i>Post conditions</i>	<ol style="list-style-type: none"> 1. A new category has been created; 2. Administration menu is displayed.
<i>Alternative Actions</i>	A2 aborted the whole operation.

Use case #5: disable user (AUC5)

<i>Actor</i>	A2
<i>Preconditions</i>	Administrator successfully logged in.
<i>Basic course of events</i>	<ol style="list-style-type: none"> 1. A2 chooses to prevent a user from logging in; 2. System displays the list of registered users; 3. System asks for the user ID to disable; 4. A2 enters the user ID; 5. System resets selected user password.

<i>Post conditions</i>	1. The selected user is disabled; 2. Administration menu is displayed.
<i>Alternative Actions</i>	A2 aborted the whole operation.

Use case #6: change product details (AUC6)

<i>Actor</i>	A2
<i>Preconditions</i>	Administrator successfully logged in.
<i>Basic course of events</i>	1. A2 chooses to change the details of a specific product; 2. System displays the list of available products; 3. System asks for the product ID to edit; 4. A2 enters the product ID; 5. The system starts asking what information need to be changed; 6. A2 selects the proper menu item and enters the new value; 7. When finished, the system stores the new information.
<i>Post conditions</i>	1. Selected product has been updated; 2. Administration menu is displayed.
<i>Alternative Actions</i>	A2 aborted the whole operation.

Use case #7: product delete (AUC7)


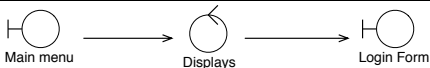
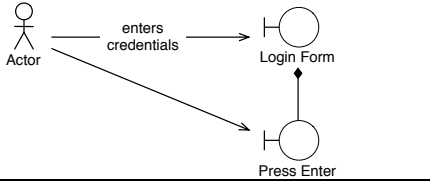
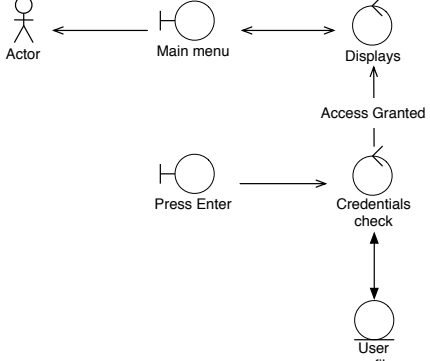
<i>Actor</i>	A2
<i>Preconditions</i>	Administrator successfully logged in.
<i>Basic course of events</i>	1. A2 chooses to delete a specific product; 2. System displays the list of available products; 3. System asks for the product ID to delete; 4. A2 enters the product ID; 5. The system checks if chosen product have ever been sold: if not, the product is physically deleted, otherwise marked as <i>deleted</i> .
<i>Post conditions</i>	1. Chosen product has been deleted; 2. Administration menu is displayed.
<i>Alternative Actions</i>	A2 aborted the whole operation.

Problem analysis and design

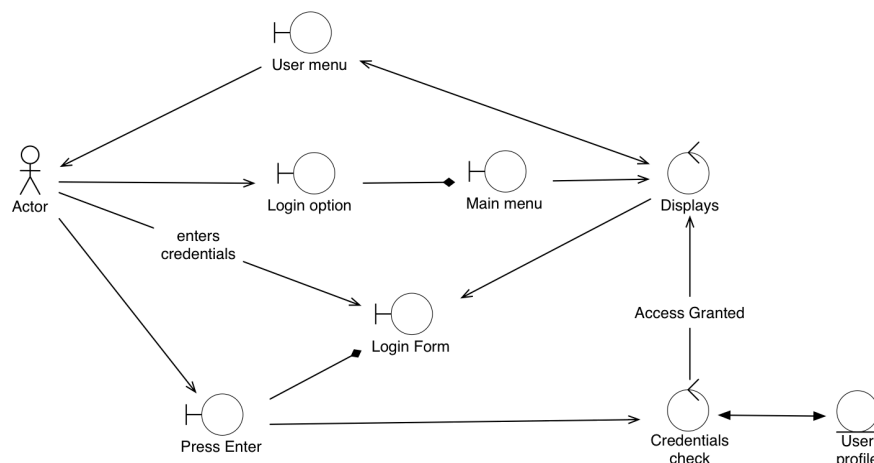
ROBUSTNESS ANALYSIS

We consider the robustness analysis of the use case UC1, user login, since authentication process is considered strategic for the system; the remaining use cases are referable to this one.

The following table summarizes each step of the use case.

ID	Step	description
1		Actor selects the option to log into the system
2		The system displays the login form
3		Actor is asked to enter his own credentials
4		The system verifies that user is registered and entered credentials are valid. User profile is accessed and "User menu" is shown to the actor.

Merging all these steps, we obtain the whole picture of the operation.



CLASS DIAGRAM

Due to its high complexity, the class diagram for the above system has been split in different parts, each representing a logical part of the whole hierarchy.

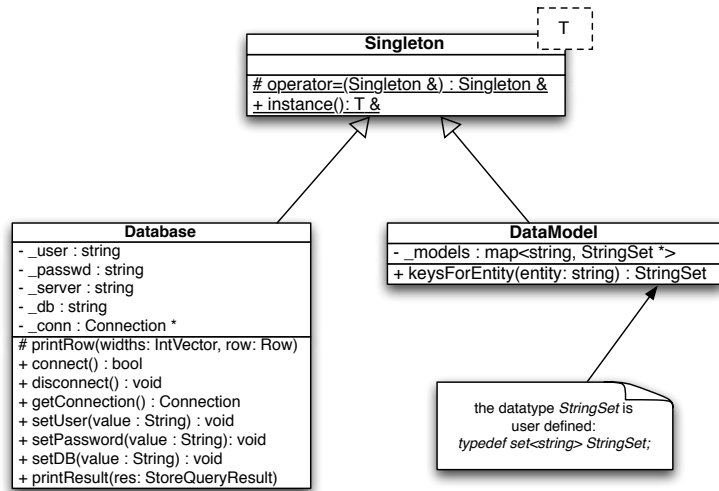


Figure 1. Class diagram #1: singletons

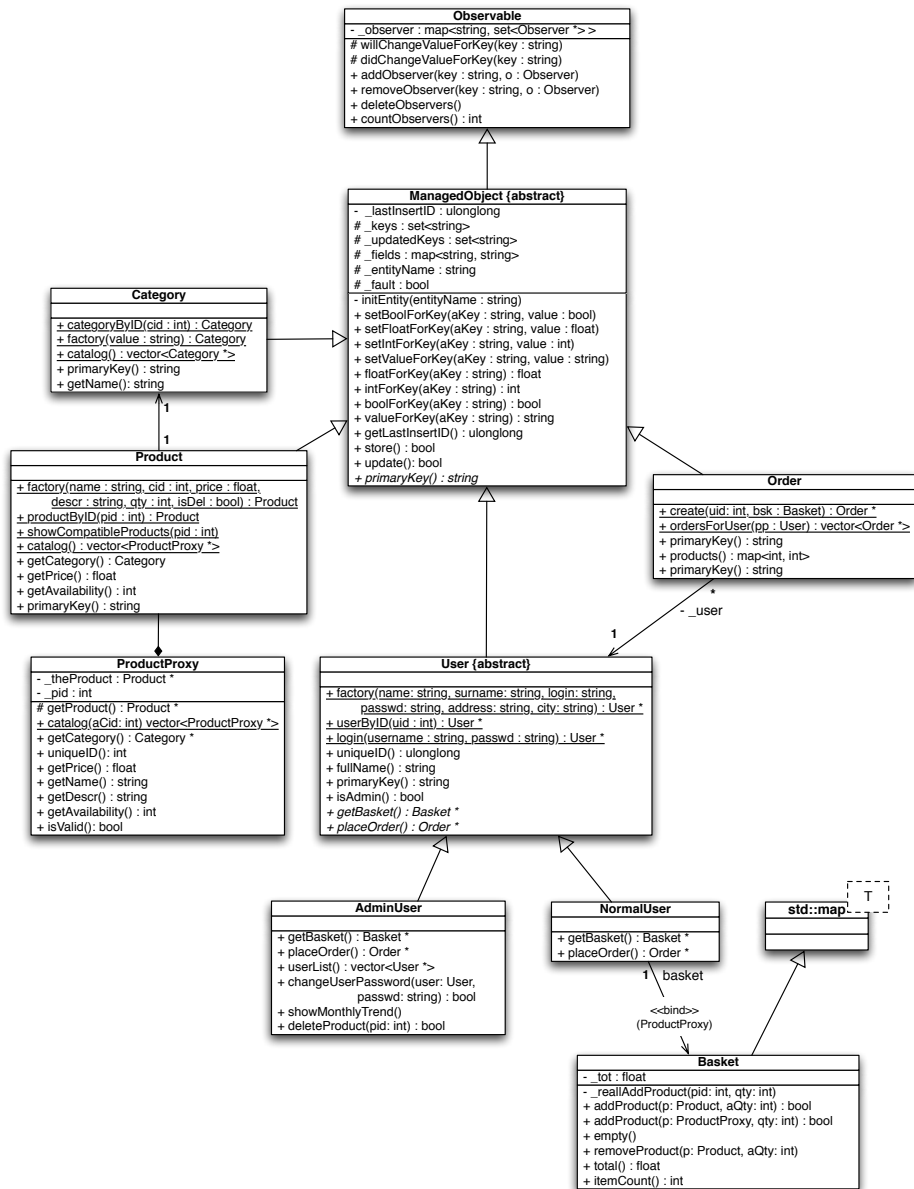


Figure 2. Class diagram #2: main class hierarchy

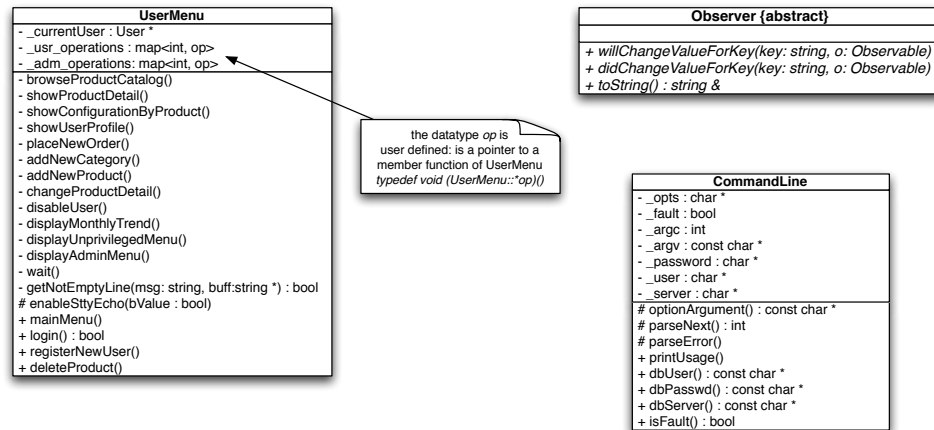


Figure 3. Class diagram #3: helper classes

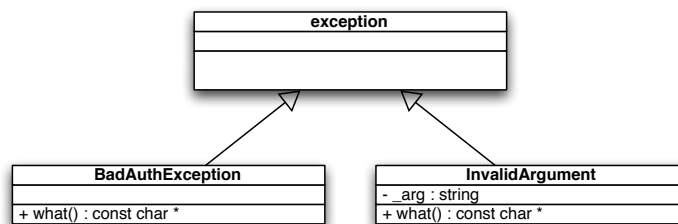


Figure 4. Class diagram #4: error handling

All diagrams conform to UML standards, all class attributes and methods can be deduced from the above schemes; according to the information hiding and black box programming concepts, all class attributes and the most of their implementation are declared as private, so that a client programmer could only interact with them by using public or static methods.

At the base of the system, we aimed to ensure three functionalities:

1. Key/value observing;
2. A generic way to interact with the database;
3. Avoid redundant query to database.

The class *Observable* exports all methods for a generic observer to register and get notified whenever a particular observed attribute changes, while the class *ManagedObject* represents a generic table of the database, with the knowledge of how to fetch, update and store a record into it, without the need of knowing the database structure. The third functionality is given by class *DataModel*, which acts as cache for some of the request issued by an instance of *ManagedObject* or its derived class.

The following is a detailed description of all classes, their roles and functionalities.

Class *Observable*

The class *Observable* defines a mechanism that allows objects to be notified of changes to specified properties of other objects. There is no central object that provides change notification for all observer: instead, notifications are sent directly to the observing objects when changes are made.

In order to be notified of changes to a property, an observing object must first register with the object to be observed by sending it an *addObserver* message, passing the attribute to be observed and the observing object. At the same way, an observer can remove itself from observing list, by calling *removeObserver* and passing the same parameters.

Observable
- observer : map<string, set<Observer *>>
willChangeValueForKey(key : string)
didChangeValueForKey(key : string)
+ addObserver(key : string, o : Observer)
+ removeObserver(key : string, o : Observer)
+ deleteObservers()
+ countObservers() : int

The class *Observable* has a private map which stores, for each attribute to monitor, a set of observer to be notified: the class *std::set* ensures that the same observer could register itself only once for the same attribute.

When the value of an observed property of an object changes, the observer receives two message, *willChangeValueForKey* before the change happens and *didChangeValueForKey* after change have been.

Class *Observer*

The class *Observer* is the counterpart of *Observable*: a class that wants to be notified of attribute changes of another object must derive from this and implements the two virtual methods of base class, respectively *willChangeValueForKey* and *didChangeValueForKey*.

The key/value notification mechanism ensures that these methods will called whenever an observer attribute is about to change or changed.

Observer {abstract}
+ willChangeValueForKey(key: string, o: Observable)
+ didChangeValueForKey(key: string, o: Observable)
+ toString() : string &

Class *Singleton (template)*

A singleton is perhaps the simplest design pattern, common to many programming languages and it's the way to allow one and only one instance of a class.

The key to creating a singleton is to prevent the client programmer from having any control over the lifetime of the object. To do this, all constructors are declared *private*, so that we also prevent the compiler from implicitly generating any constructors. Also the copy constructor and assignment operator (which intentionally have no implementations, as they will never be called) are declared private to prevent any sort of copies being made.

Singleton
operator=(Singleton &) : Singleton &
+ instance(): T &

Extending this concept, it's possible to create a template class which may be used to easily create singleton classes; the template, of course, nullifies assignment and copy operators, and declares a template method called *instance()*, used to access the only possible instance of the class.

To make a class a singleton, it's sufficient to:

1. Make its constructor *private* or *protected*;
2. Make *Singleton<MyClass>* a *friend*;
3. Derive *MyClass* from *Singleton<MyClass>*.

Let's see some real example of this pattern.

Class *Database* (singleton)

The class *Database* guarantees the physical interaction with the MySQL server: it's responsible for connecting to the database, local or remote, authenticating with the given credentials and selecting the proper schema.

This class has been designed to be a *singleton*, as it directly derives from the *Singleton* template described above.

The private attributes, along with their setters, allow to specify different credentials to use or server to connect to.

At last, the helper method *printResult* could be used to print a generic result set returned by a selection query, i.e. a SELECT: the method will create a list of returned columns, along with their width, and will print all rows (the single row is printed by the protected method *printRow*).

Database
- _user : string
- _passwd : string
- _server : string
- _db : string
- _conn : Connection *
printRow(widths: IntVector, row: Row)
+ connect() : bool
+ disconnect() : void
+ getConnection() : Connection
+ setUser(value : String) : void
+ setPassword(value : String) : void
+ setDB(value : String) : void
+ printResult(res: StoreQueryResult)

Class *ManagedObject*

The abstract class *ManagedObject* is a generic class that implements all the basic behaviour of a table of the database. So, a managed object is associated with an entity of the database⁴; this information must be passed to the constructor of the class: when an instance of the class is created, a connection to the database is automatically requested in order to fetch all the available columns of the linked table.

In some respects, the class *ManagedObject* acts like a dictionary, as it's a generic container object that efficiently provides storage for the properties fetched from the database table. The class also comes with two methods to get and set the value of these attributes: *setValueForKey* and *valueForKey*, respectively the setter and the getter of a generic key; there are also other methods, similar to these two, that act in the same

⁴ The term *entity* also refers to an instance of the ER model described above.

way but allow the programmer to specify different data types: for example, even if an integer attribute could be set with the method *setValueForKey* and automatically casted to *int*, the method *setIntForKey* is more flexible and doesn't force the programmer to convert the parameter.

Managed objects typically represent data held in the database: in some situations, a managed object may be a *fault*, i.e. the value of its attributes are different from those in the persistent store; this scenario is presented, for example, when a record is fetched from the database and then updated by the client. In order to make changes persistent, there are two methods:

- *update()*: this method must be used with a fetched record, to save its values to the database (i.e. the record already exists in the database, we're only updating it);
- *store()*: this method must be used with a new record which needs to be inserted in the store for the first time. When this method is used, the client programmer could also be interested in calling *getLastInsertID()*, which returns the value of the last auto increment generated by MySQL server.

ManagedObject {abstract}
- lastInsertID : ulonglong # _keys : set<string> # _updatedKeys : set<string> # _fields : map<string, string> # _entityName : string # _fault : bool - initEntity(entityName : string) + setBoolForKey(aKey : string, value : bool) + setFloatForKey(aKey : string, value : float) + setIntForKey(aKey : string, value : int) + setValueForKey(aKey : string, value : string) + floatForKey(aKey : string) : float + intForKey(aKey : string) : int + boolForKey(aKey : string) : bool + valueForKey(aKey : string) : string + getLastInsertID() : ulonglong + store() : bool + update() : bool + primaryKey() : string

Note that the above methods automatically creates the equivalent SQL statements to achieve the expected result.

As an abstract class, the ManagedObject must be of course derived to be used: the derived classes must implement the pure virtual method primaryKey() which returns the name of the attribute that acts as primary key for the associated table: this information is necessary to properly create the SQL statement for adding new records (see store).

The class uses some private attributes needed to maintain the key/value attribute and allow their update:

- *_keys*: the list of all available – and allowed – columns; whenever the programmer tries to set an attribute not in this list, an exception is raised;
- *_updatedKeys*: the list of updated fields only (see also the above *update()* method);
- *_fields*: the list of attributes and their values of the current record;
- *_entityName*: the name of the linked table in the database;
- *_fault*: the faulting state of the instance.

Class *DataModel* (singleton)

We know that, when an instance of *ManagedObject* is created, a request to database is established, asking for all available columns, i.e. the name of the attributes of that specific entity.

But, what would happen if two instances of same class are created? Both instances would ask the database for the same information, by wasting time and resources; for this reason, this kind of request is asked to the singleton *DataModel*, which acts as proxy with the database, but also as cache for already learned table structures.

So, the first time the method *keysForEntity()* is called, the class asks the database for attributes list of given entity, while immediately returns the same information whenever the same question is issued again.

DataModel
- _models : map<string, StringSet *>
+ keysForEntity(entity: string) : StringSet

This process is guaranteed by a *std::map*, which associates the entity name – the key – with the list of attributes fetched from the persistent store.

Class *User*

The class *User* is responsible for representing a generic person, a potential customer which interacts with the system; by requirement, there are two categories of user:

- *registered users*: a user must be registered into the system before any interaction, such as buying a product;
- *administrators*: this second type of users have the role of administering the overall system.

This abstract class, directly derived from *ManagedObject*, implements at least three important operations:

1. *user authentication*: the static method *login()* is responsible for verifying that passes parameters, representing user credentials, are valid to log into the system⁵; if credentials are valid, an instance of this class is returned;
2. *new user registration*: in order to insert a new user into the database, the class exports the method *factory()*⁶ which allows to specify all data related to the new user.

User {abstract}
+ factory(name: string, surname: string, login: string, passwd: string, address: string, city: string) : User *
+ userById(uid : int) : User *
+ login(username : string, passwd : string) : User *
+ uniqueID() : ulonglong
+ fullName() : string
+ primaryKey() : string
+ isAdmin() : bool
+ getBasket() : Basket *
+ placeOrder() : Order *

⁵ The robustness of this process will be also deeper analyzed further on.

⁶ The method *factory()* implies that *store()* must be called to make changes persistent.

The former method, *login()*, hides a polymorphic implementation: whenever the passed credentials are valid, the returned instance of *User* could be actually an instance of class *NormalUser* or *AdminUser*; see the following paragraphs to learn more about these classes.

The class *User* also introduces two new pure virtual methods, *placeOrder()* and *getBasket()*, which will be only implemented by the class *NormalUser*.

Class *AdminUser*

This class derives from *User* and is associated to the entity *customer* of the ER model. An administrator should be considered an instance of a generic user but with higher privileges, so that he can administer the system.

As derived from *User*, the class must of course implement all virtual pure methods; anyway, the operations of placing an order or getting the products basket have no meaning for an administrator: for this reason, these methods return a NULL value, which must be handled by the client programmer.

AdminUser
+ getBasket() : Basket * + placeOrder() : Order * + userList() : vector<User *> + changeUserPassword(user: User, passwd: string) : bool + showMonthlyTrend() + deleteProduct(pid: int) : bool

Moreover the class introduces new functionalities, typical of an administrator:

- *changeUserPassword()*: this method lets an administrator to change a user password; this operation is useful, for example, whenever we want to lock a user, i.e. prevent him from logging into the system;
- *userList()*: this method allows an administrator to obtain the list of all registered users;
- *showMonthlyTrend()*: all incomes are obtained by calling this method, allowing an administrator to be aware of sales trend.

Class *NormalUser*

Also this class, as far as *AdminUser*, derives from *User* and is associated with entity *customer*, detailed in ER model. A normal user must be intended as a customer who needs to browse the products catalog and, possibly, buy one or more products.

Whenever a registered user wants to buy a product, he must choose how many pieces of each product should be added to the basket and asks the system to place a new order: the class

Basket, as described further on, is the container of the products the user chose; it is clear that the implementation of virtual method *getBasket()* will return the instance of class *Basket* owned by the current user⁷.

NormalUser
+ getBasket() : Basket * + placeOrder() : Order *

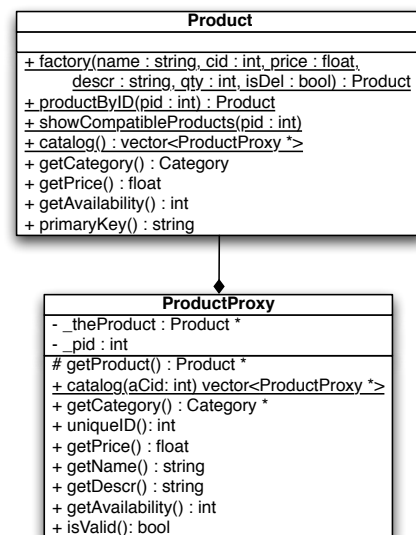
⁷ The instance of the user *Basket* is a private class variable.

On the other way, the method *placeOrder()* will analyze the content of the basket and, if not empty, will create an instance of class *Order* (see the related class description).

Class *Product* and *ProductProxy* (virtual proxy)

The class *Product* represents a generic item to be sold. As the ER model states, each product logical belongs to a category, i.e. a family of products of the same nature, such as monitor, accessories and so on.

The first class, *Product*, of course need to implements virtual methods of base class *ManagedObject*, its use is similar to previous classes described so far: the methods *factory()* allows to create a new instance of product, as well as *productByID()* fetches a product from the persistent store based on the specified criteria and *catalog()* returns the entire list of available products.



Whenever a program need to interact with a product, but not necessarily access to its attributes or call the methods described above, there is a chance to use the virtual proxy *ProductProxy*; a virtual proxy is a class that provides *lazy initialization* to create expensive objects on demand: the client code talks to this surrogate class and the real class, that does the work, is hidden behind this surrogate class. When the client program calls a function in the surrogate, it simply turns around and calls the function in the implementing class.

An example of this use is represented by the creation of a new order: the user is asked to choose a product from the catalog by specifying its ID; instead of creating an instance of class *Product*, which would come along with all its attributes fetched from the database, a *ProductProxy* is created and added to the user basket. Of course, whenever the user wants to access to the product detail, the *ProductProxy* will create a real instance of the *Product* and returns the requested information.

So, it turns out that all getter methods in the proxy, such as *getPrice()*, *getDescr()* or *getName()*, will first call the protected method *getProduct()*, responsible for lazy creation of the product, and then forward to it the original request.

Class *Category*

The class *Category* represents a logical group of products of the same nature; it's associated with the entity *category* of the ER model and derives directly from *ManagedObject*.

As requirements, each product must be assigned to a category, as well as category can contain zero or more than one product.

The static method *catalog()* can be used to obtain the entire list of available categories; the method *factory()* is used to create a new category which, anyway, need to be *store()*-ed and the *categoryByID()* returns an instance of the class after having fetched data from the persistent store based on the specified criteria.

Category
+ categoryByID(cid : int) : Category
+ factory(value : string) : Category
+ catalog() : vector<Category *>
+ primaryKey() : string

Class Order

This class realizes the main target of the overall system: let a user to buy products by placing an order; analyzing the ER model, it's possible to see that an order is a master-detail structure, in which the *master* is represented by this class and contains information about the owner of the order, the date and the overall total; the *detail* part is made up of the list of chosen products (relation *madeup* of ER model).

Order
+ create(uid: int, bsk : Basket) : Order *
+ ordersForUser(pp : User) : vector<Order *>
+ primaryKey() : string
+ products() : map<int, int>

Also this class derives from *ManagedObject* and its use doesn't differ from the previous classes; there three methods which deserve an explanation:

- *create()*: this method is similar to the *factory()* of class *User* or *Product*; it accepts the user ID who place the order and his basket; an instance of class *Order* is returned;
- *ordersForUser()*: returns a list of all orders for a given user; this method is useful for a user to access to his history;
- *products()*: this method return a dictionary whose key is the product ID and the associated value is the quantity.

Class Basket

The class *Basket* represent the container of product the user chose to buy; in consideration of its intrinsic nature, it's directly derived from *std::map*.

The class *std::map*, one of the containers offered by STL, acts like a dictionary of values: in this case the key of the dictionary is represented by the product ID, while the value to assign is the number of pieces of that product. The method *addProduct()*, used to add a new product to the basket, which accepts either an instance of *ProductProxy* or *Product*, extracts the product ID from the parameter and verifies if the same

Basket
- _tot : float
- _reallAddProduct(pid: int, qty: int)
+ addProduct(p: Product, aQty: int) : bool
+ addProduct(p: ProductProxy, qty: int) : bool
+ empty()
+ removeProduct(p: Product, aQty: int)
+ total() : float
+ itemCount() : int

product was already present in the map: in this case, no new insert will be done, but only quantity will be increased.

There are also two counterpart of *addProduct*: *empty()* will remove all products present in the basket, while *removeProduct()* allow to remove only a specific product from it.

Class *CommandLine*

The class *CommandLine* is an helper class used to parse command line arguments passed to the executable. The main usage of the class is suggested by its constructor, which expects two parameters, the *argc* and *argv*⁸ of the *main()*, the entry point of any program.

The class offers the opportunity to specify different parameters related to the database; here the list of possible values:

SPECIFIER	DESCRIPTION	DEFAULT VALUE
-u	database user	root
-p	database password	secret
-s	database server	localhost
-d	debug level	0

As instantiated, the analysis of the parameters takes place: whenever the parser finds a syntax error or a misuse of the above specifiers, parse immediately stops and the private variable *_fault* is set to true, indicating the presence of an error.

Whenever an error is found, an help message is displayed to the user, with the expected syntax to be used.

CommandLine
- _opts : char *
- _fault : bool
- _argc : int
- _argv : const char *
- _password : char *
- _user : char *
- _server : char *
optionArgument() : const char *
parseNext() : int
parseError()
+ printUsage()
+ dbUser() : const char *
+ dbPasswd() : const char *
+ dbServer() : const char *
+ isFault() : bool

After parameters analysis ended, it's possible to access parsed data with the getter methods *dbUser()*, *dbPasswd()* and *dbServer()*.

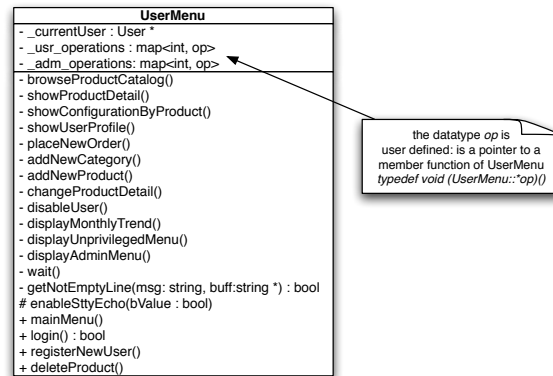
Class *UserMenu*

The interaction between the user, either an administrator or an unprivileged one, and the system is guaranteed by the class *UserMenu*.

It's responsible for creating a textual interface which, first of all, acts the role of authentication and authorization: as requirements, only registered users can interact with the system; all other user can anyway register to the system and log into it.

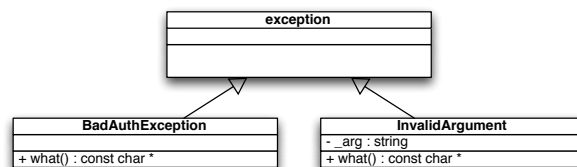
⁸ The variable *argc* is equal to the number of parameters passed to the program, while *argv* is the vector of parameters; in any case, the first value of *argv* is always the executable itself, as specified by POSIX standard.

The class also comes with a plethora of methods, each of them implements a requirement; they all shares the same prototype and they all, when invoked, check if they are called by an authenticated user and if the user has the privileges to call the method: if this is not verified, an exception is raised.



Two custom exceptions were derived from the base class *exception* in order to generate proper error messages in particular contexts. The class *BadAuthException* is raised by *UserMenu* whenever a method, which corresponds to a specific operation, is called without a properly authenticated and authorized user.

The second exception, *InvalidArgument*, is instead raised by the class *ManagedObject*, whenever the client programmer is trying to set/get an unknown key.



SEQUENCE DIAGRAM

Instead of using the primitives of previous scheme, the following diagram is made up of all the classes, represented by a box, which take part to the whole operation; the vertical line, called *lifeline*, helps to perceive the relation of a cause and its effect.

27

- *call action* and *response action*: caller waits until the receiver is ready to receive the signal and the same caller waits until the receiver processes the message and possibly returns a value;
- *object creation*: whenever an instance of a class is created.

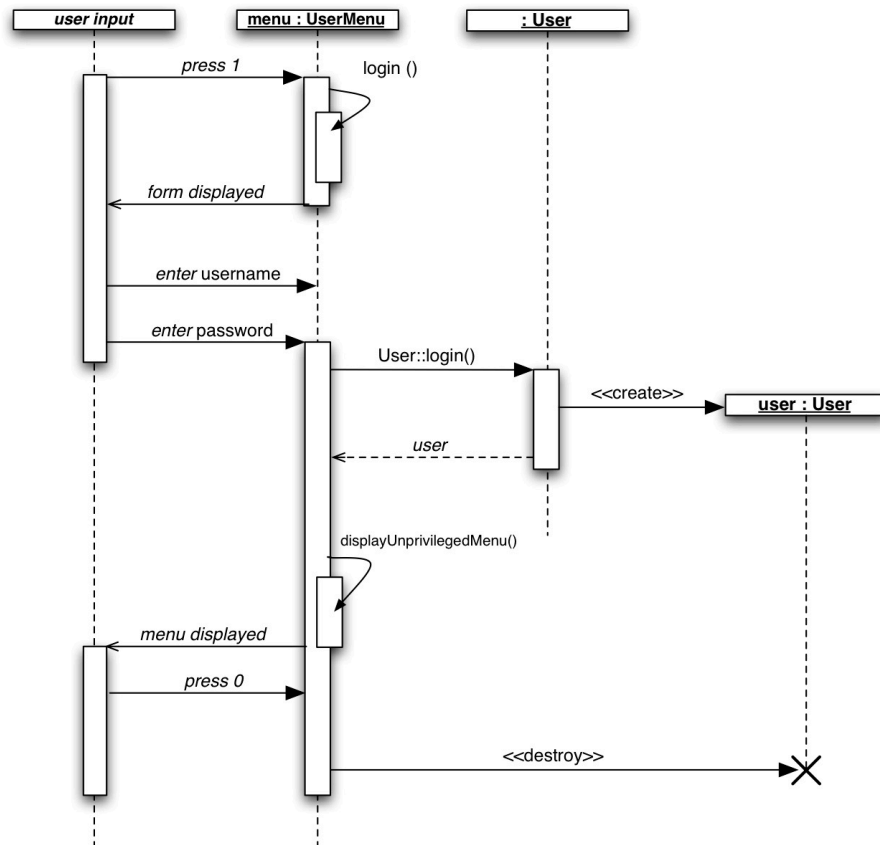


Figure 5. Sequence diagram of login

This sequence is a particular case of login process, i.e. whenever an unprivileged user successfully logs into the system. The class *UserMenu* shows the menu and handle user input; as far as the user enters his own credentials, *UserMenu* invokes the synchronous static method *User::login()* which, if credentials were valid, returns an instance of class *User*; this value is then returned to the *UserMenu* and the user menu is displayed.

The instance of class *User* will remain alive until, from the unprivileged menu, the user chooses to logout: this operation implies that *user* is destroyed.

STATE DIAGRAM

In this section we describe the possible states in which the application can be. For a better readability, first a macro vision of the possible states is shown, then each of them is exploited.

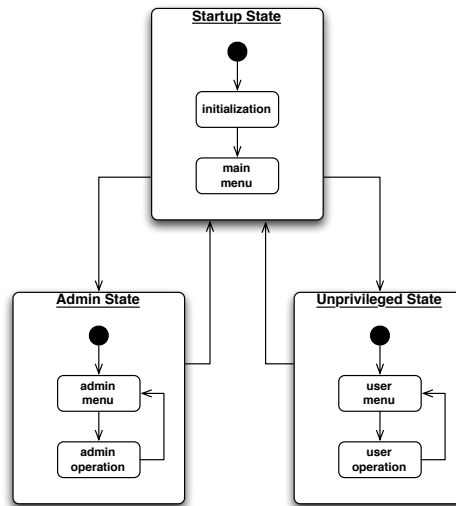


Figure 6. Representation of macro-states

After program start-up, the application automatically enters in the first macro state, *startup state*; at a deeper analysis, it's possible to split this first phase in two sub state, in which the application goes trough without any external event or action.

After initialization ended, the main menu is shown which let a user log into the system: based on authorization level of current user, the application will enter in two possible macro states, mutually exclusive: *admin state* or *unprivileged state*.

The following is the diagram of sequential sub states of *unprivileged state*. The entry point is intended to be the main menu, so passing from the *startup state* to the *unprivileged state*.

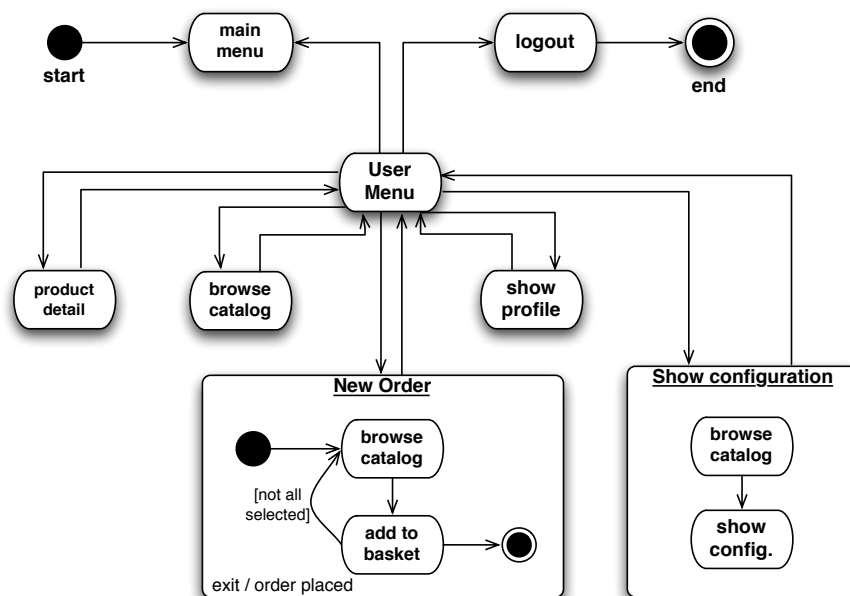


Figure 7. Unprivileged state

Each state change is driven by user input, either a menu item selection or simple press of RETURN key when the system tells that requested operation ended.

Whenever the authenticated user is recognized as an administrator, the application changes its state and enters in the *admin state*. In consideration of an higher privileges, the following diagram describes a greater interaction between the system and the administrator.

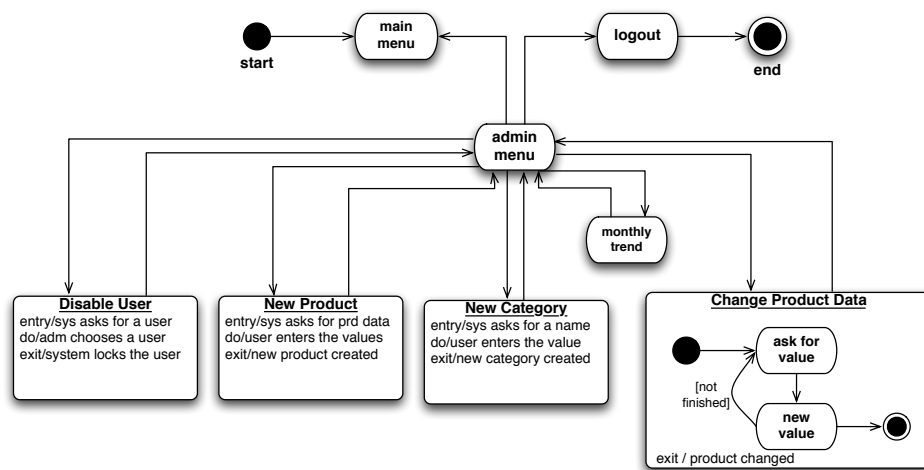


Figure 8. Administration state

Implementation

APPLICATION

common.h

```
/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: common.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __COMMON_H__
#define __COMMON_H__

#include <iostream>
#include <cstdio>
#include <iomanip>
#include <map>
#include <set>
#include <vector>

// User-defined data types
typedef std::set<std::string> StringSet;
typedef std::vector<size_t> IntVector;

/**
 * @brief Merge container values, delimited by a separator
 *
 * The function valueMerge need an iterator associated to the container,
 * an iterator that points to the end of it and a string used as
 * separator; a common use of this function is to obtain a string starting
 * from a container, such as a vector or a set.
 */
template <class InputIterator, class T>
T valueMerge(InputIterator first, InputIterator last, T delim)
{
    T value;

    while (first != last) {
        value += *first++;
        if (first != last) value += delim;
    }

    return value;
}

/**
 * Template function to deallocate a container which owns a list
 * of pointer to object.
 */
template <typename T>
struct deletePtr : public std::unary_function<bool, T>
{
    bool operator()(T *pT) const {
        delete pT;
        return true;
    }
};

/**
 * A singleton is perhaps the simplest design pattern, common to many
 * programming languages and it's the way to allow one and only one
 * instance of a class.
 *
 * The key to creating a singleton is to prevent the client programmer
 * from having any control over the lifetime of the object. To do this,
 * all constructors are declared private, so that we also prevent the
 * compiler from implicitly generating any constructors. Also the copy
 * constructor and assignment operator (which intentionally have no
 * implementations, as they will never be called) are declared private
 */
```

```

    to prevent any sort of copies being made.
    */
template <class T> class Singleton
{
private:
    Singleton(const Singleton &);
    Singleton & operator=(const Singleton &);
    T & operator=(const T &);

protected:
    Singleton() {}
    virtual ~Singleton() {}

public:
    static T & instance() {
        static T theInstance;
        return theInstance;
    }
};

extern int debugLevel;

#define CLEAR_SCREEN_CMD "clear"
#define LOG(L, ...)      if (L <= debugLevel) { \
                        printf("[%s:%d] [%s] ", __FILE__, \
                        __LINE__, __func__); \
                        printf(__VA_ARGS__); \
                        }

#define LOG_CTOR()      LOG(3, "ctor called\n")
#define LOG_DTOR()      LOG(3, "dtor called\n")

#endif /* __COMMON_H__ */

```

Exceptions.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unix@devzero.it>
 * All rights reserved.
 *
 * $Id: Exceptions.h 73 2010-08-12 17:20:17Z unix $
 */

#ifndef __EXCEPTIONS_H__
#define __EXCEPTIONS_H__

#include <iostream>
#include <exception>

using namespace std;

/**
 @brief Bad Authentication/authorization exception

 @see UserMenu
 */
class BadAuthException : public exception
{
public:
    BadAuthException() {}
    virtual ~BadAuthException() throw() {}

    virtual const char *what() const throw() {
        return "Authorization error";
    }
};

/**
 @brief Invalid argument exception

 @see ManagedObject
 */
class InvalidArgument : public exception
{
private:
    string _arg;

```



```

public:
    InvalidArgument(string anArg) : _arg(anArg) {}
    virtual ~InvalidArgument() throw() {};

    virtual const char *what() const throw() {
        string msg = "Invalid argument - " + _arg;
        return msg.c_str();
    }
};

#endif /* __EXCEPTIONS_H__ */

```

Observer.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Observer.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __OBSERVER_H__
#define __OBSERVER_H__

#include <string>

using namespace std;

// Forward declaration
class Observable;

/**
 * The class Observer is the base class to derive in order to receive
 * notifications from observed object.
 * An generic observer must first register itself as observer for an
 * attribute of an instance of a class: whenever this attribute is
 * changed, the observer is notified before and after value changed.
 *
 * @see Observable
 */
class Observer
{
public:
    virtual ~Observer() {}

    virtual void willChangeValueForKey(string, Observable *) = 0;
    virtual void didChangeValueForKey(string, Observable *) = 0;
    virtual string & toString() = 0;
};

#endif /* __OBSERVER_H__ */

```

Observable.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Observable.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __OBSERVABLE_H__
#define __OBSERVABLE_H__

#include "common.h"

using namespace std;

// Forward declaration
class Observer;

/**
 * The class Observable defines a mechanism that allows objects to be
 * notified of changes to the specified properties of other objects.
 */

```

You can observe any object properties including simple attributes.
 Observers are informed of the type of change made – as well as which
 objects are involved in the change.

```

  */
class Observable
{
private:
    /** Association between the key and the list of observers */
    map<string, set<Observer *> > _observers;

protected:
    virtual void willChangeValueForKey(string aKey);
    virtual void didChangeValueForKey(string aKey);

public:
    Observable();
    virtual ~Observable();

    virtual void addObserver(string aKey, Observer & o);
    virtual void removeObserver(string aKey, Observer & o);
    virtual void removeAllObservers();
    virtual int countObservers();
};

#endif /* __OBSERVABLE_H__ */

```

Observable.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Observable.cpp 73 2010-08-12 17:20:17Z unixo $
 */

#include "Observable.h"
#include "Observer.h"

/**
 * @brief Default constructor
 */
Observable::Observable()
{
    LOG_CTOR();
}

/**
 * @brief Default destructor
 */
Observable::~~Observable()
{
    LOG_DTOR();
}

/**
 * @brief Register an observer
 *
 * Adds an entry to the receiver's dispatch table with an observer,
 * for a notifications restricted to key.
 *
 * @param[in] aKey The name of the notification for which to register
 *                the observer
 * @param[in] o    Object registering as an observer.
 *                This value must not be NULL
 */
void Observable::addObserver(std::string aKey, Observer & o)
{
    map<string, set<Observer *> >::iterator it = _observers.find(aKey);
    set<Observer *> *aSet;

    LOG(3, "Added new observer: %s\n", o.toString().c_str());

    if (it != _observers.end())
        aSet = &(*it).second;
    else

```

```

        aSet = new set<Observer *>();

        aSet->insert(&o);
        _observers[aKey] = *aSet;
    }

    /**
     * @brief Unregister an observer
     *
     * Removes matching entries from the receiver's dispatch table for the
     * specified keys.
     *
     * @param[in]    aKey  The name of the notification for which to remove
     *                      the observer
     * @param[in]    o      Observer to remove from the dispatch table
     */
    void Observable::removeObserver(string aKey, Observer& o)
    {
        map<string, set<Observer *> >::iterator it = _observers.find(aKey);
        if (it == _observers.end())
            return;

        LOG(3, "Removed observer: %s\n", o.toString().c_str());

        set<Observer *> *aSet = &(it->second);
        aSet->erase(aSet->find(&o));
        _observers[aKey] = *aSet;
    }

    /**
     * @brief Remove all observer from the dispatch table.
     */
    void Observable::removeAllObservers()
    {
        _observers.clear();
    }

    /**
     * @brief Returns the number of keys observed.
     *
     * @return    Count of keys.
     */
    int Observable::countObservers()
    {
        return _observers.size();
    }

    /**
     * @brief Notification before value changes
     *
     * Invoked to inform the receiver that the value of a given property
     * is about to change.
     *
     * @return    aKey The name of the property that will change.
     */
    void Observable::willChangeValueForKey(string aKey)
    {
        map<string, set<Observer *> >::iterator it = _observers.find(aKey);
        if (it == _observers.end())
            return;

        LOG(3, "willChangeValueForKey('%s')\n", aKey.c_str());

        set<Observer *> *aSet = &(it->second);
        set<Observer *>::iterator sit;
        for (sit = aSet->begin(); sit != aSet->end(); sit++)
            (*sit)->willChangeValueForKey(aKey, this);
    }

    /**
     * @brief Notification after value changed
     *
     * Invoked to inform the receiver that the value of a given property
     * has changed.

```

```

        @return    aKey The name of the property that changed.
    */
void Observable::didChangeValueForKey(string aKey)
{
    map<string, set<Observer *> >::iterator it = _observers.find(aKey);
    if (it == _observers.end())
        return;

    LOG(3, "didChangeValueForKey('%s')\n", aKey.c_str());

    set<Observer *> *aSet = &(*it).second;
    set<Observer *>::iterator sit;
    for (sit = aSet->begin(); sit != aSet->end(); sit++)
        (*sit)->didChangeValueForKey(aKey, this);
}

```

Database.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unix@devzero.it>
 * All rights reserved.
 *
 * $Id: Database.h 73 2010-08-12 17:20:17Z unix $
 */

#ifndef __DATABASE_H__
#define __DATABASE_H__

#include <mysql++.h>
#include <query.h>
#include "common.h"

using namespace std;
using namespace mysqlpp;

/**
 * Manages the connection to the database server.
 * The class is intended to be used as singleton: call method
 * instance() to get an instance of Database: for this reason,
 * Database needs to be initialized early, such as in your main,
 * with default parameter (use private constructor).
 *
 * @see Singleton
 */
class Database : public Singleton<Database>
{
private:
    Connection *_conn;
    string _server;
    string _user;
    string _passwd;
    string _db;

protected:
    friend class Singleton<Database>;
    Database();

    void printRow(IntVector & widths, Row& row);

public:
    virtual ~Database();

    bool isConnected();
    bool connect();
    void disconnect();
    Connection *getConnection();
    void printResult(StoreQueryResult& res);

    void setServer(string aValue);
    void setUser(string aValue);
    void setPassword(string aValue);
    void setDB(string aValue);

    friend ostream& operator<<(ostream &, Database &);
};

```

```
#endif /* __DATABASE_H__ */
```

Database.cpp

```
/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Database.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "Database.h"

/**
 @brief Default constructor

 Init the class specifying the database informations (server to
 connect to, credentials to authenticate and database name).
 */
Database::Database()
{
    LOG_CTOR();
    setServer("localhost");
    setUser("root");
    setPassword("secret");
    setDB("seng");

    _conn = NULL;
}

/**
 @brief Default destructor

 Disconnect from the database server and destroy the instance of
 class.
 */
Database::~Database()
{
    LOG_DTOR();
    disconnect();
}

/**
 @brief Connect to database after object is created.

 If you call this method on an object that is already connected to
 a database server, the previous connection is dropped and a new
 connection is established.

 @return True if connection was established successfully.
 */
bool Database::connect()
{
    if (_conn)
        disconnect();

    try {
        _conn = new Connection(false);
        _conn->set_option(new MultiStatementsOption(true));
        _conn->connect(_db.c_str(), 0, _user.c_str(), _passwd.c_str());
    }
    catch (std::exception &e) {
        cerr << "unable to connect to database ("
            << _conn->error() << endl;
        delete _conn;
    }

    return false;
}

return _conn->connected();
}

/**
 @brief Drop the connection to the database server.
```

```

    */
void Database::disconnect()
{
    if (_conn) {
        _conn->disconnect();
        delete _conn;
        _conn = NULL;
    }
}

/**
    @brief Return a connection to the database

    @return An instance of mysqlpp::Connection
    */
Connection *Database::getConnection()
{
    return _conn;
}

/**
    @brief Change the server we going to connect to.

    If we're already connected to the server, connection is dropped: a
    new call to connect() need to be called.

    @param[in]    aValue The new server to connect to
    @see connect(), disconnect()
    */
void Database::setServer(string aValue)
{
    if (_server != aValue) {
        _server = aValue;

        if (isConnected())
            disconnect();
    }
}

/**
    @brief Change the username we are using to connect to the database
    server.

    If we're already connected to the server, connection is dropped: a
    new call to connect() need to be called.

    @param[in]    aValue The new username
    @see connect(), disconnect()
    */
void Database::setUser(string aValue)
{
    if (_user != aValue) {
        _user = aValue;

        if (isConnected())
            disconnect();
    }
}

/**
    @brief Change the password we are using to connect to the database
    server.

    If we're already connected to the server, connection is dropped: a
    new call to connect() need to be called.

    @param[in]    aValue The new password
    @see connect(), disconnect()
    */
void Database::setPassword(string aValue)
{
    if (_passwd != aValue) {
        _passwd = aValue;

        if (isConnected())

```

```

        disconnect();
    }
}

/**
@brief Database selection

Change to a different database managed by the database server
we are connected to.
*/
void Database::setDB(string aValue)
{
    if (_db != aValue) {
        _db = aValue;

        if (isConnected()) {
            _conn->select_db(_db);
        }
    }
}

/**
@brief Test connection status against the database

@return True if connection was established successfully.
*/
bool Database::isConnected()
{
    return ((_conn) && (_conn->connected()));
}

/**
@brief Print a generic row to stdout

Print to standard out the content of the given row. A vector with
all column widths need to be specified.

@param[in] widths Vector of column widths
@param[in] row The row to be printed
*/
void Database::printRow(IntVector & widths, Row & row)
{
    cout << " |" << setfill(' ');
    for (size_t i = 0; i < row.size(); ++i) {
        cout << " " << setw(widths.at(i)) << row[int(i)] << " |";
    }
    cout << endl;
}

/**
@brief Print a result set to stdou

Print to standard out the content of the given result set.
A vector with all column widths is built and passed to printRow().

@param[in] res The result set
@see printRow()
*/
void Database::printResult(StoreQueryResult& res)
{
    StoreQueryResult::size_type num_results = res.size();
    if (!res || (num_results == 0)) {
        return;
    }

    IntVector widths;
    size_t size = res.num_fields();
    for (size_t i = 0; i < size; i++) {
        widths.push_back(max(res.field(i).max_length(),
                             res.field_name(i).size()));
    }

    for (StoreQueryResult::size_type i = 0; i < num_results; ++i) {
        printRow(widths, res[i]);
    }
}

```

```

}

ostream& operator<<(ostream& aStream, Database& d) {
    return aStream << "Connection to database is" <<
        (d._conn->connected())?"":" NOT") << " established\n";
}

```

DataModel.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: DataModel.h 74 2010-08-13 16:23:12Z unixo $
 */

#ifndef __DATAMODEL_H__
#define __DATAMODEL_H__

#include "common.h"

using namespace std;

/**
 * An DataModel object describes a schema – a collection of entities
 * (data models) that you use in your application.
 * The model contains one or more objects representing the entities
 * in the schema. Each entity name object has property description
 * objects that represent the properties (or fields) of the entity
 * in the schema.
 */
class DataModel : public Singleton<DataModel>
{
private:
    map<std::string, StringSet *> _models;

protected:
    friend class Singleton<DataModel>;
    DataModel();
    virtual ~DataModel();

public:
    StringSet & keysForEntity(string anEntity);
};

#endif /* __DATAMODEL_H__ */

```

DataModel.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: DataModel.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "DataModel.h"
#include "Database.h"

/**
 * @brief Default constructor
 */
DataModel::DataModel()
{
    LOG_CTOR();
}

/**
 * @brief Default destructor
 *
 * Remove all cached models
 */
DataModel::~DataModel()
{
    LOG_DTOR();
    _models.clear();
}

```



```

}

/**
@brief Ask for the data model of given entity

The first time the method keysForEntity() is called, the class
asks the database for attributes list of given entity, while
immediately returns the same information whenever the same question is
issued again.

@param[in]    anEntity    Entity name
@return       A set of string representing table columns

@see StringSet
*/
StringSet & DataModel::keysForEntity(string anEntity)
{
    map<std::string, StringSet *>::const_iterator it =
                                                _models.find(anEntity);
    if (it != _models.end()) {
        LOG(3, "Data model for entity '%s' found in cache.\n",
            anEntity.c_str());
        StringSet *ss = (*it).second;

        return *ss;
    }

    // get an instance of the database
    Database & db = Database::instance();
    StringSet *ss = new StringSet();

    try {
        // ask Database for a valid connection to MySQL
        Connection *conn = db.getConnection();

        // obtain an instance of mysqlpp::Query and init it
        Query q = conn->query("SELECT * FROM " + anEntity);
        StoreQueryResult res = q.store();

        for (size_t i = 0; i < res.field_names()->size(); i++) {
            ss->insert(res.field_name(i).c_str());
        }

        _models.insert(pair<string, StringSet *>(anEntity, ss));
    }
    catch (const mysqlpp::BadQuery& er) {
        cerr << "Query error: " << er.what() << endl;
    }
    catch (const mysqlpp::Exception& er) {
        cerr << "Error: " << er.what() << endl;
    }
    LOG(3, "Data model for entity '%s' loaded.\n", anEntity.c_str());

    return *ss;
}

```

ManagedObject.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: ManagedObject.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __MANAGEDOBJECT_H__
#define __MANAGEDOBJECT_H__

#include "Observable.h"
#include "Exceptions.h"
#include "Database.h"

using namespace std;
using namespace mysqlpp;

```

```

/**
The abstract class ManagedObject is a generic class that implements
all the basic behaviour of a table of the database. So, a managed
object is associated with an entity of the database ; this information
must be passed to the constructor of the class: when an instance of
the class is created, a connection to the database is automatically
requested in order to fetch all the available columns of the linked
table.
*/
class ManagedObject : public Observable
{
private:
    unsigned long _lastInsertID;
    void initEntity(string anEntityName);

protected:
    /** List of possible keys associated with this entity */
    set<string> _keys;
    /** List of the updated keys to update */
    set<string> _updatedKeys;
    /** Map of key/value associated to the entity */
    map<string, string> _fields;
    /** The entity name */
    string _entityName;
    /** Fault state: if true the entity need to be serialized */
    bool _fault;

public:
    ManagedObject(string anEntityName);
    ManagedObject(string anEntityName, Row & aRow);
    virtual ~ManagedObject();

    void setBoolForKey(string aKey, bool aValue);
    void setFloatForKey(string aKey, float aValue);
    void setIntForKey(string aKey, int aValue);
    void setValueForKey(string aKey, string aValue)
        throw (InvalidArgument);

    float floatForKey(string aKey) throw (InvalidArgument);
    int intForKey(string aKey) throw (InvalidArgument);
    bool boolForKey(string aKey) throw (InvalidArgument);
    string valueForKey(string aKey) throw (InvalidArgument);

    unsigned long getLastInsertID() const;
    bool store();
    bool update();

    virtual string primaryKey() = 0;
};

#endif /* __MANAGEDOBJECT_H__ */

```

ManagedObject.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unix@devzero.it>
 * All rights reserved.
 *
 * $Id: ManagedObject.cpp 73 2010-08-12 17:20:17Z unix $
 */

#include "ManagedObject.h"
#include "DataModel.h"

/**
@brief Default constructor

Construct an instance of ManagedObject linking with a corresponding
table on database named "anEntityName".
The list of columns is fetched automatically.
*/
ManagedObject::ManagedObject(string anEntityName)
{
    LOG_CTOR();
}

```

```

        initEntity(anEntityName);
    }

    /**
     * @brief Construct an instance from a fetched record
     *
     * Construct an instance of ManagedObject linking with a corresponding
     * table on database named "anEntityName" and reading data from
     * mysqlpp::Row.
     */
    ManagedObject::ManagedObject(string anEntityName, Row & aRow)
    {
        LOG_CTOR();
        initEntity(anEntityName);

        set<string>::const_iterator it;

        for (it = _keys.begin(); it != _keys.end(); it++) {
            string key = *it;
            setValueForKey(key, (string) aRow[key.c_str()]);
        }

        _fault = false;
        _updatedKeys.clear();
    }

    /**
     * @brief Default destructor
     */
    ManagedObject::~ManagedObject()
    {
        LOG_DTOR();
    }

    /**
     * @brief Internal init
     *
     * Build the list of keys (columns) for the specified entity (table)
     */
    void ManagedObject::initEntity(string anEntityName)
    {
        _entityName = anEntityName;
        _lastInsertID = 0;
        _fault = false;

        _keys = DataModel::instance().keysForEntity(anEntityName);
    }

    /**
     * Sets the specified property of the receiver to the specified
     * string value.
     *
     * @param[in] aKey The name of one of the receiver's properties.
     * @param[in] aValue The new value for the property specified by
     * key.
     * @exception InvalidArgument Thrown if the specified key doesn't
     * belong to this entity
     * @see Observable, willChangeValueForKey, didChangeValueForKey
     */
    void ManagedObject::setValueForKey(string aKey,
                                       string aValue) throw (InvalidArgument)
    {
        // first check if the key is valid, anyway throw an exception
        set<string>::const_iterator it = _keys.find(aKey);
        if (it == _keys.end())
            throw InvalidArgument(aKey);

        // Set the value only if differs from the old one
        if (aValue != *it) {
            // notify observers that this key is going to change
            willChangeValueForKey(aKey);

            _fields[aKey] = aValue;
            _fault = true;
            _updatedKeys.insert(aKey);
        }
    }

```

```

        // notify observers that this key is changed
        didChangeValueForKey(aKey);
    }
}

/**
 @brief Set an integer value

 Sets the specified property of the receiver to the specified
 integer value.

 @param[in]    aKey    The name of one of the receiver's
                        properties.
 @param[in]    aValue  The new value for the property specified
                        by key.
 */
void ManagedObject::setIntForKey(string aKey, int aValue)
{
    setFloatForKey(aKey, (float)aValue);
}

/**
 @brief Set a float value

 Sets the specified property of the receiver to the specified
 float value.

 @param[in]    aKey    The name of one of the receiver's
                        properties.
 @param[in]    aValue  The new value for the property specified
                        by key.
 */
void ManagedObject::setFloatForKey(string aKey, float aValue)
{
    stringstream tmpSS;
    string tmpStr;

    tmpSS << aValue;
    tmpSS >> tmpStr;
    setValueForKey(aKey, tmpStr);
}

/**
 @brief Set a boolean value

 Sets the specified property of the receiver to the specified
 boolean value.

 @param[in]    aKey    The name of one of the receiver's
                        properties.
 @param[in]    aValue  The new value for the property specified
                        by key.
 */
void ManagedObject::setBoolForKey(string aKey, bool aValue)
{
    setValueForKey(aKey, (aValue?"1":"0"));
}

/**
 @brief Get an attribute value

 Returns the value as string for the property identified by a
 given key.

 @param[in]    aKey    The key to read from
 @return Value of the property

 @exception InvalidArgument Thrown if the specified key doesn't
                            belong to this entity
 */
string ManagedObject::valueForKey(string aKey) throw (InvalidArgument)
{
    // first check if the key is valid, anyway throw an exception
    set<string>::const_iterator it = _keys.find(aKey);

```

```

        if (it == _keys.end())
            throw InvalidArgument(aKey);

        return _fields[aKey];
    }

    /**
     * @brief Returns the value as boolean for the property identified
     *        by a given key.
     *
     * @param[in]    aKey The key to read from
     * @return Value of the property
     * @exception InvalidArgument Thrown if the specified key doesn't
     *        belong to this entity
     */
    bool ManagedObject::boolForKey(string aKey) throw (InvalidArgument)
    {
        string aValue = valueForKey(aKey);
        istringstream aStream(aValue);
        int i;

        if (!(aStream >> i))
            throw InvalidArgument(aKey);

        return (i == 1);
    }

    /**
     * @brief Returns the value as integer for the property identified
     *        by a given key.
     *
     * @param[in]    aKey The key to read from
     * @return Value of the property
     * @exception InvalidArgument Thrown if the specified key doesn't
     *        belong to this entity
     */
    int ManagedObject::intForKey(string aKey) throw (InvalidArgument)
    {
        string value = valueForKey(aKey);
        istringstream aStream(value);
        int i;

        if (!(aStream >> i))
            throw InvalidArgument(aKey);

        return i;
    }

    /**
     * @brief Returns the value as float for the property identified
     *        by a given key.
     *
     * @param[in]    aKey The key to read from
     * @return Value of the property
     * @exception InvalidArgument Thrown if the specified key doesn't
     *        belong to this entity
     */
    float ManagedObject::floatForKey(string aKey) throw (InvalidArgument)
    {
        string value = valueForKey(aKey);
        istringstream aStream(value);
        int f;

        if (!(aStream >> f))
            throw InvalidArgument(aKey);

        return f;
    }

    /**
     * @brief Add this instance to database
     *
     * Attempts to commit unsaved changes to the persistent store:
     * this function builds a REPLACE statement.
     * Fault state is reset if update was successful.

```

```

        @note ManagedObject::store should be called only to store new
            record and not an existing one.

        @return    True if save was successful
        @see      update
    */
    bool ManagedObject::store()
    {
        SQLQueryParms qp;
        string cols = "(";
        stringstream values;
        int i=0;

        /**
         * Iterate on all key/value in order to create the VALUE() part
         * of REPLACE statement; we also fill a SQLQueryParms to pass to
         * mysqlpp::query::execute.

         * We don't make use of valueMerge template method because we
         * need to know on which field we're iterating, in order to
         * properly fill "qp" and "values".
         */
        values << "VALUES (";
        set<string>::reverse_iterator it;
        for (it = _keys.rbegin(); it != _keys.rend(); it++) {
            cols += *it + ",";
            qp += _fields[*it];
            values << "%" << i++ << "q,";
        }

        // build the entire REPLACE statement
        cols[cols.length()-1] = ')', cols += " ";
        string sql = "REPLACE " + _entityName + cols + values.str();
        sql[sql.length()-1] = ')';

        // get an instance of the database
        Database & db = Database::instance();

        try {
            // ask Database for a valid connection to MySQL
            Connection *conn = db.getConnection();

            // obtain an instance of mysqlpp::Query and init it
            Query q = conn->query(sql);
            q.parse();

            LOG(2, "SQL: %s\n", q.str(qp).c_str());

            // check if command was successful: if not, log the error
            // and its detail
            SimpleResult r = q.execute(qp);
            if (r == false) {
                LOG(2, "An error occurred during mysqlpp:query::execute\n"
                    "ERR: %s\nQuery was: %s\n", q.error(),
                    q.str(qp).c_str());

                return false;
            }

            _lastInsertID = q.insert_id();
        }
        catch (const Exception& er) {
            cerr << "Error: " << er.what() << endl;
            return false;
        }

        // restore fault state
        _fault = false;

        return true;
    }

    /**
    @brief Make changes persistent to database.

```

Attempts to update unsaved changes to the persistent store.
Fault state is reset if update is successful.

@note ManagedObject::update should be called only to update
existing record and not an to store a new record.

@return True if update was successful
@see store

```

*/
bool ManagedObject::update()
{
    if (!_fault)
        return true;

    SQLQueryParms qp;
    set<string>::const_iterator ckit;
    vector<string> vValues;
    stringstream aValue;
    string sql;
    int i;

    /**
     * We don't make use of valueMerge template method because
     * we need to know on which field we're iterating, in order to
     * properly fill "qp" and "values".
     */
    for (i=0, ckit = _updatedKeys.begin();
         ckit != _updatedKeys.end(); ckit++, i++) {
        aValue.str("");
        aValue << *ckit << "=" << i << "q";
        vValues.push_back(aValue.str());
        qp += _fields[*ckit];
    }

    sql = "UPDATE " + _entityName + " SET " +
        valueMerge(vValues.begin(), vValues.end(), (string), "");

    // get the primary key of the entity
    string pk = primaryKey();

    // add the WHERE condition to properly identify the right record
    sql += " WHERE " + pk + " = " + _fields[pk];

    // get an instance of the database
    Database& db = Database::instance();

    try {
        // ask Database for a valid connection to MySQL
        Connection *conn = db.getConnection();

        // obtain an instance of mysqlpp::Query and init it
        Query q = conn->query(sql);

        LOG(2, "SQL: %s\n", q.str(qp).c_str());

        q.parse();

        SimpleResult r = q.execute(qp);
        if (r == false) {
            cerr << "An error occurred during Person::store()\n" << endl
                << q.error() << endl << "Query was: " << q.str() << endl;
            return false;
        }
    }
    catch (const mysqlpp::BadQuery& e) {
        // Something went wrong with the SQL query.
        cerr << "Query failed: " << e.what() << endl;
        return false;
    }
    catch (const Exception& er) {
        cerr << "Error: " << er.what() << endl;
        return false;
    }
}

```

```

        _fault = false;

        return true;
    }

    /**
     * @brief Returns the ID of the autoincrement of the last query (INSERT)
     *
     * @return The last ID
     */
    ulonglong ManagedObject::getLastInsertID() const
    {
        return _lastInsertID;
    }
}

```

User.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: User.h 74 2010-08-13 16:23:12Z unixo $
 */

#ifndef __USER_H__
#define __USER_H__

#include "common.h"
#include "ManagedObject.h"
#include "Basket.h"
#include "Order.h"

using namespace std;
using namespace mysqlpp;

/**
 * User is the base class to permit login into the system and benefit
 * of all services, such as product browsing, placing an order and
 * so on.
 * It represents instances of the table "users" but it cannot be
 * instantiated directly as it's an abstract class (see methods
 * getBasket and placeOrder).
 *
 * @note Class destructor needs to be virtual so that will be called
 *       when a derived class is used.
 *
 * @see getBasket, placeOrder
 */
class User : public ManagedObject
{
public:
    User();
    User(Row & aRow);
    virtual ~User();

    static User * factory(string aName, string aSurname, string aLogin,
                        string aPasswd, string anAddress = "",
                        string aCity = "");
    static User * userByID(int anUid);
    static User * login(string username, string passwd);

    virtual Basket * getBasket() = 0;
    virtual Order * placeOrder() throw (string) = 0;

    ulonglong uniqueID();
    string fullName();
    string loginName();
    bool isAdmin();
    string primaryKey();

    friend ostream& operator<<(ostream &, User &);
};

/**

```


This class is derived from User and is associated to the entity customer of the ER model. An administrator should be considered an instance of a generic user but with higher privileges, such that he can administer the system. As derived from User, the class must of course implements all virtual pure methods; anyway, the operations of placing an order or getting the product basket have no meaning for an administrator: for this reason, these methods return a null value, which must be handled by the client programmer.

```

*/
class AdminUser : public User
{
public:
    AdminUser(Row &aRow);
    Basket * getBasket() { return NULL; };
    Order * placeOrder() throw (string) { return NULL; };
    vector<User *> & userList();
    bool changeUserPassword(User & anUser, string aPasswd);
    void showMonthlyTrend();
    bool deleteProduct(int aPid);
};

/**
Even this class, as far as AdminUser, derives from User and is
associated with entity customer, detailed in ER model. A normal
user must be intended as a customer who need to browse the product
catalog and, possibly, buy one or more products.
Whenever a registered user wants to buy a product, he must choose
how many pieces of each product should be added to the basket and
ask the system to place a new order: the class Basket, as described
further on, is the container of products the user chose; it appears
clear that the implementation of virtual method getBasket() will
return the instance of class Basket owned by the current user.
*/
class NormalUser : public User
{
private:
    Basket basket;

public:
    NormalUser();
    NormalUser(Row &aRow);
    ~NormalUser();
    Order * placeOrder() throw (string);
    Basket * getBasket() { return &basket; };
};

#endif /* __USER_H__ */

```

User.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: User.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "User.h"
#include "Basket.h"

#define KEY_USR_UID          "uid"
#define KEY_USR_NAME        "name"
#define KEY_USR_SURNAME    "surname"
#define KEY_USR_ADDRESS    "address"
#define KEY_USR_CITY       "city"
#define KEY_USR_LOGIN      "login"
#define KEY_USR_PASSWD     "password"
#define KEY_USR_ADMIN      "admin"
#define QUERY_LOGIN        "SELECT * FROM users WHERE login = %0q AND " \
    "password = %1q"
#define QUERY_FETCH        "SELECT * FROM users WHERE uid = "
#define QUERY_ADMIN_USERLST "SELECT * FROM users WHERE admin=0 " \
    "ORDER BY surname, name"
#define QUERY_ADMIN_TREND  "SELECT DATE_FORMAT(date, '%Y') AS year, " \

```

```

        "DATE_FORMAT(date, '%m') as month, " \
        "SUM(total) AS total_sales FROM orders " \
        "GROUP BY year, month ORDER BY year, month"

/**
    @brief Default constructor

    Used to create a new instance to be saved later.

    @see factory, ManagedObject
*/
User::User() : ManagedObject("users")
{
    LOG_CTOR()
}

/**
    @brief Class constructor

    Constructs an instance of User and fill it with data taken from
    mysqlpp::Row

    @param    aRow Record fetched from the database
    @see ManagedObject, mysqlpp::Row
*/
User::User(Row &aRow) : ManagedObject("users", aRow)
{
    LOG_CTOR()
}

/**
    @brief Default destructor
*/
User::~~User()
{
    LOG_DTOR()
}

/**
    @brief Returns the primary for entity "users" (KEY_USR_UID)

    @return The primary key of the entity
*/
string User::primaryKey()
{
    return KEY_USR_UID;
}

/**
    @brief Create a new user

    Create a new instance of an unprivileged user, initialized
    with parameters.

    @param[in]    aName    The name of the user
    @param[in]    aSurname The surname of the user
    @param[in]    aLogin   The login of the user
    @param[in]    aPasswd  The password of the user
    @param[in]    anAddress The address of the user
    @param[in]    aCity    The city of the user
*/
User * User::factory(string aName, string aSurname, string aLogin,
                    string aPasswd, string anAddress, string aCity)
{
    User *nu = new NormalUser();
    nu->setIntForKey(KEY_USR_UID, 0);
    nu->setIntForKey(KEY_USR_ADMIN, 0);
    nu->setValueForKey(KEY_USR_NAME, aName);
    nu->setValueForKey(KEY_USR_SURNAME, aSurname);
    nu->setValueForKey(KEY_USR_ADDRESS, anAddress);
    nu->setValueForKey(KEY_USR_CITY, aCity);
    nu->setValueForKey(KEY_USR_LOGIN, aLogin);
    nu->setValueForKey(KEY_USR_PASSWD, aPasswd);
    if (!nu->store()) {

```

```

        delete nu;

        return NULL;
    }

    return nu;
}

/**
@brief User login

Test given user credential to log into the system: if successful
an instance of NormalUser or AdminUser is returned, according to
authorization level, NULL otherwise.

@param[in]    username The username to test
@param[in]    passwd The password to test
@return       An instance of User class
*/
User * User::login(string username, string passwd)
{
    // get an instance of the database
    Database &db = Database::instance();

    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query(QUERY_LOGIN);
    q.parse();

    StoreQueryResult res = q.store(username, passwd);
    if (!res.empty()) {
        StoreQueryResult::const_iterator it = res.begin();
        Row row = *it;

        User *newUser = NULL;

        if (row[KEY_USR_ADMIN] == "1")
            newUser = new AdminUser(row);
        else
            newUser = new NormalUser(row);

        return newUser;
    }

    return NULL;
}

/**
@brief User constructor by ID

Fetch a user by specifying its ID: if successful, an instance of
NormalUser or AdminUser is returned, according to authorization
level, NULL otherwise.

@param[in]    anUid User ID to be fetched
@return       An instance of User
*/
User * User::userByID(int anUid)
{
    // get an instance of the database
    Database &db = Database::instance();

    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query();
    q << QUERY_FETCH << anUid;
    StoreQueryResult res = q.store();
    if (!res.empty()) {
        StoreQueryResult::const_iterator it = res.begin();
        Row row = *it;

```

```

        if (row[KEY_USR_ADMIN] == "1")
            return new AdminUser(row);
        else
            return new NormalUser(row);
    }

    return NULL;
}

/**
 * @brief Returns the user ID which uniquely identifies this instance
 *
 * @return The user ID
 */
ulonglong User::uniqueID()
{
    return (ulonglong) intForKey(KEY_USR_UID);
}

/**
 * @brief Returns the full name of current user
 *
 * @return The full name
 */
string User::fullName()
{
    return valueForKey(KEY_USR_NAME) + " " + valueForKey(KEY_USR_SURNAME);
}

/**
 * @brief Returns the username used to log into the system
 *
 * @return The login name
 */
string User::loginName()
{
    return valueForKey(KEY_USR_LOGIN);
}

/**
 * @brief Returns the authorization level of current user.
 *
 * @return True if the current user has administration privileges.
 */
bool User::isAdmin()
{
    /**
     * polymorphic approach
     * (it's anyway possible to use getUser()->boolForKey(KEY_USR_ADMIN));
     */
    AdminUser *admin = dynamic_cast<AdminUser *> (this);

    return (admin != NULL);
}

ostream& operator<<(ostream& aStream, User & u) {
    return aStream << "Name : " << u.valueForKey(KEY_USR_NAME) << " "
        << u.valueForKey(KEY_USR_SURNAME) << endl
        << "Login : " << u.valueForKey(KEY_USR_LOGIN) << endl
        << "Address: " << u.valueForKey(KEY_USR_ADDRESS) << " ("
        << u.valueForKey(KEY_USR_CITY) << ")\n";
}

/**
 * @brief Class constructor by fetched record
 *
 * Constructs an instance of AdminUser and fill it with data taken from
 * mysqlpp::Row
 *
 * @param aRow Record fetched from the database
 * @see ManagedObject, User::User
 */
AdminUser::AdminUser(Row &aRow) : User(aRow)
{

```

```

    LOG_CTOR()
}

/**
@brief Registered user list

Returns the list of currently registered users, even if locked.

@return Vector of pointer to class User
@see AdminUser, NormalUser, std::vector
*/
vector<User *> & AdminUser::userList()
{
    // get an instance of the database
    Database &db = Database::instance();
    vector<User *> *users = NULL;

    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query(QUERY_ADMIN_USERLIST);
    StoreQueryResult res = q.store();
    if (!res.empty()) {
        users = new vector<User *>;
        users->reserve(res.num_rows());

        StoreQueryResult::const_iterator it;
        for (it=res.begin(); it != res.end(); it++) {
            User *anUser;
            Row row = *it;

            if (row[KEY_USR_ADMIN] == "1")
                anUser = new AdminUser(row);
            else
                anUser = new NormalUser(row);
            assert(anUser);
            users->push_back(anUser);
        }
    }

    return *users;
}

/**
@brief Delete a product

If the product has already been sold in past orders, it's marked as
deleted, otherwise it's physically removed from the database.

@param[in] aPid The product ID to delete

@return True if operation was successful
*/
bool AdminUser::deleteProduct(int aPid)
{
    assert(aPid >= 0);

    // get an instance of the database
    Database& db = Database::instance();

    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query();
    q << "CALL product_delete(" << aPid << ")";

    SimpleResult r = q.execute();
    if (r == false) {
        LOG(2, "An error occurred during mysqlpp:query::execute\n"
            "ERR: %s\nQuery was: %s\n", q.error(), q.str().c_str());
        return false;
    }
}

```

```

        return true;
    }

    /**
     @brief Change user password
     Let an administrator to change a user password: this method could
     also be used to disable a user login.
     To make this change persistent, remember to commit by calling store().

     @param[in]    anUser Instance of user to alter
     @param[in]    aPasswd New password to assign
     @return       True if change was successful
     @see ManagedObject::store()
    */
    bool AdminUser::changeUserPassword(User & anUser, string aPasswd)
    {
        assert(anUser.uniqueID() != 0);

        anUser.setValueForKey(KEY_USR_PASSWD, aPasswd);
        return anUser.update();
    }

    /**
     @brief Displays the monthly trend of sales, grouped by year/month.
    */
    void AdminUser::showMonthlyTrend()
    {
        // get an instance of the database
        Database &db = Database::instance();

        // ask Database for a valid connection to MySQL
        Connection *conn = db.getConnection();

        // obtain an instance of mysqlpp::Query and init it
        Query q = conn->query(QUERY_ADMIN_TREND);
        StoreQueryResult res = q.store();
        db.printResult(res);
    }

    /**
     @brief Default constructor

     @see User, ManagedObject
    */
    NormalUser::NormalUser() : User()
    {
        LOG_CTOR()
    }

    /**
     @brief Class constructor by fetched record

     Constructs an instance of NormalUser and fill it with data taken
     from mysqlpp::Row

     @param    aRow Record fetched from the database
     @see ManagedObject, User::User, mysqlpp::Row
    */
    NormalUser::NormalUser(Row &aRow) : User(aRow)
    {
        LOG_CTOR()
    }

    /**
     @brief Default destructor
    */
    NormalUser::~NormalUser()
    {
        LOG_DTOR()
    }

    /**

```

```

    @brief New order creation

    Call this method when the user has a filled his basket and want
    to place a new order; if successful, an instance of class Order is
    returned and the basket is cleared.

    @return    An instance of class Order if successful, NULL otherwise
    @see Order, Order::create()
    */
Order * NormalUser::placeOrder() throw (string)
{
    if (basket.size() == 0)
        throw "Basket must contain at least one product";

    Order *newOrder = Order::create(intForKey(KEY_USR_UID), basket);
    if (newOrder) {
        basket.clear();
    }

    return newOrder;
}

```

Basket.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Basket.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __BASKET_H__
#define __BASKET_H__

#include "common.h"

// forward declaration
class Product;
class ProductProxy;

/**
 * The class Basket represents the basket of the customer who
 * wants to buy one or more products: it's intended to be used
 * by User class only.
 */
class Basket : public std::map<int, int>
{
private:
    float _tot;
    void _realAddProduct(int pid, int qty);

public:
    Basket();
    ~Basket();

    bool addProduct(Product *p, int aQty = 1);
    bool addProduct(ProductProxy *p, int aQty = 1);
    float total() const;
    void empty();
    void removeProduct(Product *p, int aQty = 1);
    int itemCount();

    friend std::ostream & operator<<(std::ostream &, Basket &);
};

#endif /* __BASKET_H__ */

```

Basket.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Basket.cpp 74 2010-08-13 16:23:12Z unixo $
 */

```

```

#include "Basket.h"
#include "Product.h"

/**
 * @brief Default constructor
 */
Basket::Basket()
{
    LOG_CTOR();
    _tot = 0.0;
}

/**
 * @brief Default destructor
 */
Basket::~Basket()
{
    LOG_DTOR();
}

/**
 * @brief Add product to basket
 *
 * Add a product to the basket. It's possible to specify how many
 * pieces of the same product.
 * A check against product availability is done before adding it
 * to the basket.
 *
 * @param[in] p An instance to ProductProxy
 * @param[in] aQty How many pieces of the same product
 *                (default is 1)
 * @return True if successful, false otherwise
 */
bool Basket::addProduct(ProductProxy *p, int aQty)
{
    if (aQty > p->getAvailability())
        return false;

    _tot += p->getPrice();

    int aPid = p->uniqueID();
    _realAddProduct(aPid, aQty);

    return true;
}

/**
 * @brief Add product to basket
 *
 * Add a product to the basket. It's possible to specify how
 * many pieces of the same product.
 * A check against product availability is done before adding
 * it to the basket.
 *
 * @param[in] p An instance to Product
 * @param[in] aQty How many pieces of the same product
 *                (default is 1)
 * @return True if successful, false otherwise
 */
bool Basket::addProduct(Product *p, int aQty)
{
    if (aQty > p->getAvailability())
        return false;

    _tot += p->getPrice();

    int aPid = p->intForKey(p->primaryKey());
    _realAddProduct(aPid, aQty);

    return true;
}

/**
 * @brief Real add product

```


This function must not be used directly, but only called by addProduct() – that's why it's private. It adds the chosen product and requested quantity to the map, by first looking for an equal entry: in this case only quantity is increased.

```

    @param[in]    pid    Product ID to add
    @param[in]    qty    Requested quantity
*/
void Basket::_realAddProduct(int pid, int qty)
{
    int aValue = qty;

    map<int,int>::iterator it = find(pid);
    if (it != end())
        aValue += (*it).second;
    insert(pair<int, int>(pid, aValue));
}

/**
@brief Empty the basket
*/
void Basket::empty()
{
    _tot = 0.0;
    clear();
}

/**
@brief Remove a product from the basket

The function looks for the given product and decreases
the quantity: if this value goes down below or equal to
zero, the product is removed; otherwise only quantity is
decreased.

@param[in]    p    Pointer to Product to be removed
@param[in]    aQty    Requested quantity to remove
*/
void Basket::removeProduct(Product *p, int aQty)
{
    int aPid = p->intForKey(p->primaryKey());
    map<int,int>::iterator it = find(aPid);
    if (it != end()) {
        int aValue = (*it).second;
        aValue -= aQty;
        if (aValue <= 0)
            erase(it);
        else
            insert(pair<int, int>(aPid, aValue));
    }
}

/**
@brief Items count

Return the number of item currently present in the basket.

@return Number of items
*/
int Basket::itemCount()
{
    map<int,int>::const_iterator it;
    int count = 0;

    for (it = begin(); it != end(); it++)
        count += (*it).second;

    return count;
}

/**
@brief Return the total cost of the basket

@return The total, expressed as float
*/

```

```

float Basket::total() const
{
    return _tot;
}

ostream& operator<<(ostream& aStream, Basket & p) {
    for (map<int, int>::const_iterator it=p.begin(); it != p.end(); it++)
    {
        ProductProxy p = ProductProxy( (*it).first );
        aStream << left << setw(30) << p.getName() << " | " << right
            << (*it).second << endl;
    }

    return aStream;
}

```

Category.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Category.h 74 2010-08-13 16:23:12Z unixo $
 */

#ifndef __CATEGORY_H__
#define __CATEGORY_H__

#include "common.h"
#include "ManagedObject.h"

using namespace std;
using namespace mysqlpp;

/**
 * The class Category represents a logical group of products of the
 * same nature; it's associated with the entity category of the ER
 * model and derives directly from ManagedObject.
 *
 * As requirements, each product must be assigned to a category, as
 * well as category can contain zero or more than one product.
 */
class Category : public ManagedObject
{
public:
    Category();
    static Category *categoryByID(int aCid);
    static Category *factory(string aValue);
    static vector<Category *> & catalog();

    string primaryKey();
    string getName();

    friend ostream& operator<<(ostream &, Category &);
};

#endif /* __CATEGORY_H__ */

```

Category.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Category.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "Category.h"
#include "Database.h"

#define KEY_CAT_CID          "cid"
#define KEY_CAT_NAME        "name"
#define SQL_CATEGORY_BYID   "SELECT cid, name FROM categories WHERE " \
    "cid = "
#define SQL_CATEGORY_CAT    "SELECT cid, name FROM categories " \
    "ORDER BY cid"

```

```

/**
    @brief Class constructor
    */
Category::Category() : ManagedObject("categories")
{
    LOG_CTOR();
}

/**
    @brief Fetch a category by specifying its ID

    @param[in]    aCid Category ID to be fetched
    @return       An instance of category
    */
Category *Category::categoryByID(int aCid)
{
    Category *cat = NULL;

    // get an instance of the database
    Database &db = Database::instance();

    // obtain an instance of mysqlpp::Query and init it
    Query q = db.getConnection()->query();
    q << SQL_CATEGORY_BYID << aCid;
    StoreQueryResult res = q.store();
    if (!res.empty()) {
        cat = new Category();
        cat->setIntForKey(KEY_CAT_CID, aCid);
        cat->setValueForKey(KEY_CAT_NAME, (string) res[0][KEY_CAT_NAME]);

        return cat;
    }

    return NULL;
}

/**
    @brief Create a new category

    Create a new category to be stored into the database; the method
    store need to be called to make changes persistent.

    @param    aValue    The category name
    @return   An instance of the new category
    */
Category *Category::factory(string aValue)
{
    Category *newCategory = new Category();
    newCategory->setIntForKey(KEY_CAT_CID, 0);
    newCategory->setValueForKey(KEY_CAT_NAME, aValue);

    return newCategory;
}

/**
    @brief Returns the primary for entity "categories" (KEY_CAT_CID)

    @return The primary key of the entity
    */
string Category::primaryKey()
{
    return KEY_CAT_CID;
}

/**
    @brief Returns the list of available categories.

    @return    Vector of pointer to Category
    */
vector<Category *> &Category::catalog()
{
    // get an instance of the database
    Database& db = Database::instance();
    vector<Category *> *catalog = NULL;

```

```

try {
    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query(SQL_CATEGORY_CAT);
    StoreQueryResult res = q.store();

    if (!res.empty()) {
        catalog = new vector<Category *>;
        catalog->reserve(res.num_rows());

        for (size_t i = 0; i < res.num_rows(); ++i) {
            Category *c = new Category();
            c->setValueForKey(KEY_CAT_CID,
                             (string) res[i][KEY_CAT_CID]);
            c->setValueForKey(KEY_CAT_NAME,
                             (string) res[i][KEY_CAT_NAME]);
            catalog->push_back(c);
        }
    }
} catch (const mysqlpp::BadQuery& e) {
    // Something went wrong with the SQL query.
    cerr << "Query failed: " << e.what() << endl;
} catch (const Exception& er) {
    cerr << "Error: " << er.what() << endl;
}

return *catalog;
}

/**
 @brief Returns category name

 @return A string representing the category
 */
string Category::getName()
{
    return valueForKey(KEY_CAT_NAME);
}

ostream& operator<<(ostream& aStream, Category & c) {
    return aStream << right << setfill(' ') << setw(4) <<
        c.valueForKey(KEY_CAT_CID) << " | " << left << setw(30) <<
        c.valueForKey(KEY_CAT_NAME);
}

```

Product.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Product.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __PRODUCT_H__
#define __PRODUCT_H__

#include "common.h"
#include "ManagedObject.h"
#include "Category.h"

#define KEY_PRD_PID          "pid"
#define KEY_PRD_CID          "cid"
#define KEY_PRD_NAME         "name"
#define KEY_PRD_DESCR        "descr"
#define KEY_PRD_PRICE         "price"
#define KEY_PRD_AVAILABILITY  "availability"
#define KEY_PRD_DELETED       "deleted"

using namespace std;

```

```

using namespace mysqlpp;

// forward declaration
class ProductProxy;

/**
The class Product represents a generic item to be sold. As
the ER model states, each product logical belongs to a category,
i.e. a family of products of the same nature, such as monitor,
accessories and so on.
The class Product of course need to implements virtual methods
of base class ManagedObject, its use is similar to previous
classes described so far: the methods factory() allows to create
a new instance of product, as well as productByID() fetches a
product from the persistent store based on the specified criteria
and catalog() returns the entire list of available products.
*/
class Product : public ManagedObject
{
public:
    Product();
    Product(Row & aRow);
    ~Product();

    static Product * factory(string aName, int aCid, float aPrice,
                             string aDescr = "empty", int aQty = 0,
                             bool isDel = false);
    static Product * productByID(int aPid);
    static void showCompatibleProducts(int aPid);

    auto_ptr<Category> getCategory();
    float getPrice();
    int getAvailability();
    string primaryKey();

    friend ostream& operator<<(ostream &, Product &);
};

/**
The class ProductProxy is a virtual proxy of class Product.
There could be situation when only a proxy of the real product
is needed and not all its data are requested: for example, think
at new order placing, where only product ID is asked to be known.
Whenever other information are requested, ProductProxy provides
lazy creation of the product and forwards the request.

@see Product, getProduct()
*/
class ProductProxy
{
private:
    Product *_theProduct;
    int _pid;

protected:
    Product *getProduct() throw (InvalidArgument);

public:
    ProductProxy(const ProductProxy & pp);
    ProductProxy(int aPid);
    ~ProductProxy();

    static vector<ProductProxy *> & catalog(int aCid = 0);

    auto_ptr<Category> getCategory();
    int uniqueID() const;
    float getPrice();
    string getName();
    string getDescr();
    int getAvailability();
    bool isValid();

    ProductProxy & operator=(const ProductProxy&);
    friend ostream& operator<<(ostream &, ProductProxy &);
};

```

```
};

#endif /* __PRODUCT_H__ */
```

Product.cpp

```
/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Product.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "Product.h"
#include "Database.h"

#define SQL_CATALOG_PROXY      "SELECT pid FROM catalogue "
#define SQL_PRODUCT_PROXY     "SELECT * FROM products WHERE pid = "

/**
 * @brief Default constructor
 */
Product::Product() : ManagedObject("products")
{
    LOG_CTOR();
}

/**
 * @brief Construct a Product with values contained in a
 *        mysqlpp::Row
 *
 * @see mysqlpp::Row
 */
Product::Product(Row & aRow) : ManagedObject("products", aRow)
{
    LOG_CTOR();
}

/**
 * @brief Default destructor
 */
Product::~~Product()
{
}

/**
 * @brief New product creation
 *
 * Static method to create a new product (without saving into
 * the database): product details are filled with given parameters.
 *
 * @param[in] aName      Product name
 * @param[in] aCid       Category ID
 * @param[in] aPrice     Product price
 * @param[in] aDescr     Product detailed description
 * @param[in] aQty       Number of stocked items
 * @param[in] isDel      True if the product is not available
 *
 * @return      An instance of the new product
 */
Product *Product::factory(string aName, int aCid, float aPrice,
                          string aDescr, int aQty, bool isDel)
{
    Product *newProduct = new Product();
    newProduct->setIntForKey(KEY_PRD_PID, 0);
    newProduct->setIntForKey(KEY_PRD_CID, aCid);
    newProduct->setValueForKey(KEY_PRD_NAME, aName);
    newProduct->setValueForKey(KEY_PRD_DESCR, aDescr);
    newProduct->setFloatForKey(KEY_PRD_PRICE, aPrice);
    newProduct->setIntForKey(KEY_PRD_AVAILABILITY, aQty);
    newProduct->setBoolForKey(KEY_PRD_DELETED, isDel);

    return newProduct;
}
```

```

/**
@brief Returns a product with given ID

@param[in] aPid    The requested product ID

@return A pointer to the requested Product, NULL if not found
*/
Product * Product::productByID(int aPid)
{
    // get an instance of the database
    Database &db = Database::instance();

    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query();
    q << SQL_PRODUCT_PROXY << aPid;
    StoreQueryResult res = q.store();
    if (!res.empty()) {
        StoreQueryResult::const_iterator it = res.begin();
        Row row = *it;

        return new Product(row);
    }

    return NULL;
}

/**
@brief Primary key

Returns the primary for entity "products" (KEY_PRD_PID)

@return The primary key of the entity
*/
string Product::primaryKey()
{
    return KEY_PRD_PID;
}

/**
@brief Return the price

@return A float representing the price
*/
float Product::getPrice()
{
    return floatForKey(KEY_PRD_PRICE);
}

/**
@brief Return stock availability

@return An integer representing the availability
*/
int Product::getAvailability()
{
    return intForKey(KEY_PRD_AVAILABILITY);
}

/**
@brief Show all configurations which include a given product

@param[in] aPid    Given product ID
*/
void Product::showCompatibleProducts(int aPid)
{
    // get an instance of the database
    Database& db = Database::instance();

    // ask Database for a valid connection to MySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it

```

```

        Query q = conn->query();
        q << "CALL offers_by_product(" << aPid << ")";
        StoreQueryResult res = q.store();

        for (int i = 1; q.more_results(); ++i) {
            cout << "\n\nCONFIGURATION DETAIL\n=====\\n";
            db.printResult(res);

            cout << "\nConfiguration includes the following products:\\n"
                 << "=====\\n";

            res = q.store_next();
            db.printResult(res);
            cout << endl;

            res = q.store_next();
            StoreQueryResult res = q.store();
        }
    }

    /**
     * @brief Returns product category
     *
     * @return An instance of class Category which the product belongs
     */
    auto_ptr<Category> Product::getCategory()
    {
        return
        auto_ptr<Category>(Category::categoryByID(intForKey(KEY_PRD_CID)));
    }

    ostream& operator<<(ostream& aStream, Product & p) {
        return aStream << "PRODUCT DETAIL\\n" <<
            "Category : " << *(p.getCategory()) << endl <<
            "Name : " << p.valueForKey(KEY_PRD_NAME) << endl <<
            "Description : " << p.valueForKey(KEY_PRD_DESCR) << endl <<
            "Price : " << p.valueForKey(KEY_PRD_PRICE) << endl <<
            "Availability: " << p.valueForKey(KEY_PRD_AVAILABILITY) << endl;
    }

    /**
     * @brief Constructor with given ID
     *
     * Construct an instance of ProductProxy by specifying product unique ID.
     *
     * @param[in] aPid Product unique ID
     */
    ProductProxy::ProductProxy(int aPid)
    {
        _pid = aPid;
        _theProduct = NULL;
    }

    /**
     * @brief Default destructor
     *
     * Dealloc instance of real object if it was allocated.
     */
    ProductProxy::~ProductProxy()
    {
        if (_theProduct)
            delete _theProduct;
    }

    /**
     * @brief Obtain a real instance of the Product
     *
     * Concrete creation of an instance of Product class; when
     * client programmer tries to access to some ProductProxy methods,
     * such as getPrice(), a call to this method ensures that the product
     * is instantiated only we really needed.
     *
     * @return A pointer to class Product
     */

```



```

Product *ProductProxy::getProduct() throw (InvalidArgument)
{
    if (!_theProduct) {
        // get an instance of the database
        Database &db = Database::instance();

        // ask Database for a valid connection to mySQL and obtain a Query
        Query q = db.getConnection()->query();

        // obtain an instance of mysqlpp::Query and init it
        q << SQL_PRODUCT_PROXY << _pid;
        StoreQueryResult res = q.store();

        if (res.empty())
            throw InvalidArgument("PID");

        StoreQueryResult::const_iterator it = res.begin();
        Row row = *it;

        _theProduct = new Product(row);
    }

    return _theProduct;
}

/**
 @brief Returns the price of this product

 @return The price of the product
 @see getProduct(), Product()
 */
float ProductProxy::getPrice()
{
    return getProduct()->getPrice();
}

/**
 @brief Returns product name

 @return The name of the product
 @see getProduct(), Product()
 */
string ProductProxy::getName()
{
    return getProduct()->valueForKey(KEY_PRD_NAME);
}

/**
 @brief Returns product description

 @return The product description
 @see getProduct(), Product()
 */
string ProductProxy::getDescr()
{
    return getProduct()->valueForKey(KEY_PRD_DESCR);
}

/**
 @brief Return the product unique ID

 @return Product ID
 */
int ProductProxy::uniqueID() const
{
    return _pid;
}

/**
 @brief Return the product availability

 @return How many pieces of this product are available
 */
int ProductProxy::getAvailability()
{

```

```

        return getProduct()->intForKey(KEY_PRD_AVAILABILITY);
    }

    /**
     * @brief Return the list of products of a given category
     *
     * @param[in] aCid The category ID
     *
     * @return A vector of ProductProxy
     *
     * @note Pass zero as category ID to get all products
     */
    vector<ProductProxy *> & ProductProxy::catalog(int aCid)
    {
        // get an instance of the database
        Database& db = Database::instance();
        vector<ProductProxy *> *catalog = NULL;

        try {
            // ask Database for a valid connection to MySQL
            Connection *conn = db.getConnection();

            // obtain an instance of mysqlpp::Query and init it
            Query q = conn->query();
            q << SQL_CATALOG_PROXY;
            if (aCid != 0)
                q << "WHERE cid = " << aCid;
            q << " ORDER BY pid, category, name";
            StoreQueryResult res = q.store();

            if (!res.empty()) {
                catalog = new vector<ProductProxy *>;
                catalog->reserve(res.num_rows());

                for (size_t i = 0; i < res.num_rows(); ++i) {
                    catalog->push_back(new ProductProxy((int) res[i][0]));
                }
            }
        }
        catch (const mysqlpp::BadQuery& e) {
            // Something went wrong with the SQL query.
            cerr << "Query failed: " << e.what() << endl;
        }
        catch (const Exception& er) {
            cerr << "Error: " << er.what() << endl;
        }
        return *catalog;
    }

    /**
     * @brief Checks if the instance of product is valid
     *
     * @return True if the instance is valid, false otherwise
     */
    bool ProductProxy::isValid()
    {
        return (getProduct() != NULL);
    }

    /**
     * @brief Assignment operator
     *
     * The operator checks if the given item differs from this instance
     * and, in this case, it only copies product ID.
     */
    ProductProxy & ProductProxy::operator=(const ProductProxy &pp)
    {
        if (this != &pp) {
            _pid = pp._pid;
            _theProduct = NULL;
        }

        return *this;
    }

```

```

/**
    @brief Returns product category

    @return    An instance of class Category which the product belongs
 */
auto_ptr<Category> ProductProxy::getCategory()
{
    return getProduct()->getCategory();
}

ostream& operator<<(ostream& aStream, ProductProxy & pp) {
    Product *p = pp.getProduct();
    return aStream << *p << endl;
}

```

Order.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Order.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __ORDER_H__
#define __ORDER_H__

#include "ManagedObject.h"
#include "Basket.h"

using namespace mysqlpp;

// forward declaration
class User;

/**
    This class realizes the main target of the overall system:
    let a user to buy products by placing an order; analyzing
    the ER model, it's possible to see that an order is a
    master-detail structure, in which the master is represented
    by this class and contains information about the owner of the
    order, the date and the overall total; the detail part is made
    up of the list of chosen products (relation 'madeup' of ER model).
 */
class Order : public ManagedObject
{
private:
    User *_user;

public:
    Order();
    Order(Row &aRow);
    ~Order();

    string primaryKey();

    static Order *create(int anUid, Basket &bsk);
    static vector<Order *> &ordersForUser(User &pp);
    map<int, int>& products();

    friend ostream& operator<<(ostream &, Order &);
};

#endif /* __ORDER_H__ */

```

Order.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: Order.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "Order.h"
#include "User.h"

```

```

#include <ctime>

#define KEY_ORD_OID      "oid"
#define KEY_ORD_UID      "uid"
#define KEY_ORD_DATE     "date"
#define KEY_ORD_TOTAL    "total"

/**
 * @brief Default constructor
 */
Order::Order() : ManagedObject("orders")
{
    LOG_CTOR();
    _user = NULL;
}

/**
 * @brief Construct an instance of Order with data fetched
        from the database

    @param[in] aRow An instance of mysqlpp::Row with data
 */
Order::Order(Row &aRow) : ManagedObject("orders", aRow)
{
    LOG_CTOR();
    _user = User::userByID(intForKey(KEY_ORD_UID));
}

/**
 * @brief Default destructor

    If the user is linked to an user, dealloc the instance of class User.
 */
Order::~~Order()
{
    LOG_DTOR();
    if (_user)
        delete _user;
}

/**
 * @brief Create a new order

    @param[in] anUid    ID of user that places the order
    @param[in] bsk User basket containing products

    @return A pointer to an instance of Order if successful
 */
Order * Order::create(int anUid, Basket & bsk)
{
    Order *o = new Order();

    // get current date and format as expected by MySQL
    time_t now = time(NULL);
    struct tm *tmNow = gmtime((const time_t *)&now);

    stringstream nowStr;
    nowStr << (1900+tmNow->tm_year) << "-" << tmNow->tm_mon << "-"
        << setfill('0') << setw(2) << tmNow->tm_mday << " "
        << tmNow->tm_hour << ":" << tmNow->tm_min
        << ":" << tmNow->tm_sec;

    // set other attributes
    o->setIntForKey(KEY_ORD_OID, 0);
    o->setFloatForKey(KEY_ORD_TOTAL, bsk.total());
    o->setValueForKey(KEY_ORD_DATE, nowStr.str());
    o->setIntForKey(KEY_ORD_UID, anUid);
    if (!o->store()) {
        delete o;
        LOG(2, "Unable to place the order.\n");

        return NULL;
    }

    ulonglong oid = o->getLastInsertID();

```

```

        // get an instance of the database
        Database& db = Database::instance();

        // ask Database for a valid connection to MySQL
        Connection *conn = db.getConnection();

        // obtain an instance of mysqlpp::Query and init it
        Query q = conn->query();

        // add to "order_details" all items present in the basket
        for (Basket::const_iterator it=bsk.begin(); it != bsk.end(); it++) {
            q << "INSERT INTO order_details VALUES (" << oid << ", "
                << (*it).first << ", " << (*it).second << ")";
            q.exec();
            q.reset();
        }

        return o;
    }

/**
    @brief Return the list of all orders of a given user

    @param[in] pp    An instance of User

    @return    A vector of Order
*/
vector<Order *> & Order::ordersForUser(User & pp)
{
    vector<Order *> *orders = new vector<Order *>;

    // get an instance of the database
    Database& db = Database::instance();

    try {
        // ask Database for a valid connection to MySQL
        Connection *conn = db.getConnection();

        // obtain an instance of mysqlpp::Query and init it
        Query q = conn->query();
        q << "SELECT * FROM orders WHERE uid = " << pp.uniqueID()
            << " ORDER BY oid, date";

        StoreQueryResult res = q.store();
        if (!res.empty()) {
            orders->reserve(res.num_rows());
            StoreQueryResult::const_iterator it;

            for (it = res.begin(); it != res.end(); it++){
                Row row = *it;
                orders->push_back(new Order(row));
            }
        }
        catch (std::exception &e) {
            cerr << "an error occurred: " << e.what() << endl;
        }

        return *orders;
    }

/**
    @brief Returns the list of products (and requested quantity) of an
    order

    @return    A std::map where the key is product ID and the value the
    quantity
*/
map<int, int>& Order::products()
{
    map<int, int> *prd = NULL;

    // get an instance of the database
    Database& db = Database::instance();

```

```

try {
    // ask Database for a valid connection to mySQL
    Connection *conn = db.getConnection();

    // obtain an instance of mysqlpp::Query and init it
    Query q = conn->query();
    q << "SELECT * FROM order_details WHERE oid = "
    << valueForKey(KEY_ORD_OID);

    StoreQueryResult res = q.store();
    if (!res.empty()) {
        prd = new map<int, int>;
        StoreQueryResult::const_iterator it;

        for (it = res.begin(); it != res.end(); it++){
            Row row = *it;
            int key = atoi(row["pid"]);
            int qty = atoi(row["qty"]);
            (*prd)[key] = qty;
        }
    }
} catch (std::exception &e) {
    cerr << "an error occurred: " << e.what() << endl;
}

return *prd;
}

/**
    @brief Returns the primary for entity "orders" (KEY_ORD_OID)

    @return The primary key of the entity
    */
string Order::primaryKey()
{
    return KEY_ORD_OID;
}

ostream& operator<<(ostream& aStream, Order & o) {
    return aStream << "ORDER DETAIL\n=====\n" <<
        "Buyer : " << o._user->fullName() << endl <<
        "Date : " << o.valueForKey(KEY_ORD_DATE) << endl <<
        "Total : " << o.valueForKey(KEY_ORD_TOTAL) << endl;
}

```

CommandLine.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: CommandLine.h 73 2010-08-12 17:20:17Z unixo $
 */

#ifndef __COMMANDLINE_H__
#define __COMMANDLINE_H__

#include <string>
#include <iostream>

using namespace std;

/**
    @brief Command line parser

    The class CommandLine is an helper class used to parse
    command line arguments passed to the executable. The main
    usage of the class is suggested by its constructor, which
    expects two parameters, the argc and argv of the main(),
    the entry point of any program.

    As instantiated, the analysis of the parameters takes place:
    whenever the parser finds a syntax error or a misuse of the
    above specifiers, parse immediately stops and the private
    variable _fault is set to true, indicating the presence of an

```

```

        error. Whenever an error is found, an help message is
        displayed to the user, with the expected syntax to be used.
    */
class CommandLine
{
private:
    bool _fault;
    int _argc;
    char * const* _argv;
    const char *_opts;
    const char *_password;
    const char *_user;
    const char *_server;
    int _debug;

protected:
    const char * optionArgument() const;
    int parseNext() const;
    void parseError();

public:
    CommandLine(int argc, char * const argv[]);

    void printUsage() const;
    const char * dbUser() const;
    const char * dbPasswd() const;
    const char * dbServer() const;
    int debugLevel();
    bool isFault();

};

#endif /* __COMMANDLINE_H__ */

```

CommandLine.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: CommandLine.cpp 73 2010-08-12 17:20:17Z unixo $
 */

#include "CommandLine.h"
#include <stdlib.h>
#include <stdio.h>

/**
 @brief Default constructor

 As instantiated, the analysis of the parameters takes place:
 whenever the parser finds a syntax error or a misuse of the
 above specifiers, parse immediately stops and the private
 variable _fault is set to true, indicating the presence of
 an error.

 @param[in]   argc   Number of available arguments
 @param[in]   argv   Array of parameters
 */
CommandLine::CommandLine(int argc, char * const argv[]) :
    _argc(argc), _argv(argv), _opts("u:p:s:d:")
{
    int ch;

    _fault = false;
    _debug = 0;
    _user = "root", _password = "secret", _server = "localhost";

    while ((ch = parseNext()) != EOF) {
        switch (ch) {
            case 'p':
                _password = optionArgument();
                break;
            case 's':
                _server = optionArgument();

```

```

        break;
    case 'u':
        _user = optionArgument();
        break;
    case 'd':
        _debug = atoi(optionArgument());
        break;
    default:
        parseError();
        return;
    }
}

/**
@brief Print an error message (syntax error)
*/
void CommandLine::parseError()
{
    cerr << "Unknown parameter!\n";
    printUsage();
    _fault = true;
}

/**
@brief Print command line synopsis
*/
void CommandLine::printUsage() const
{
    cerr << "usage: ec++ [ -u user ] [ -p password ] " \
           "[ -s server ] [-d level]\n\n";
}

/**
@brief Returns database username

@return A string representing database username
*/
const char * CommandLine::dbUser() const
{
    return _fault?NULL:_user;
}

/**
@brief Returns database password

@return A string representing database password
*/
const char * CommandLine::dbPasswd() const
{
    return _fault?NULL:_password;
}

/**
@brief Returns database server address

@return A string representing database server address
*/
const char * CommandLine::dbServer() const
{
    return _fault?NULL:_server;
}

/**
@brief Returns debug log level

@return An integer representing debug level
*/
int CommandLine::debugLevel()
{
    return _debug;
}

/**
@brief Returns success of current option parse

```



```

        @return EOF if there are no more parameters to parse
    */
    int CommandLine::parseNext() const
    {
        return getopt(_argc, _argv, _opts);
    }

    /**
        @brief Returns the expected value of an option

        @return A string representing the option value
    */
    const char * CommandLine::optionArgument() const
    {
        return optarg;
    }

    /**
        @brief Returns parse exit status

        @return True if no syntax errors were detected
    */
    bool CommandLine::isFault()
    {
        return _fault;
    }

```

UserMenu.h

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: UserMenu.h 74 2010-08-13 16:23:12Z unixo $
 */

#ifndef __USERMENU_H__
#define __USERMENU_H__

#include "common.h"
#include "User.h"
#include "Exceptions.h"

class UserMenu;

typedef void (UserMenu::*op)();

/**
    @brief Display menus and handles user input
*/
class UserMenu
{
private:
    User *_currentUser;
    map<int, op> usr_operations;
    map<int, op> adm_operations;

    void printCatalog(int aCid = 0);

    // user operations
    void browseProductCatalog() throw (BadAuthException);
    void showProductDetail() throw (BadAuthException);
    void showConfigurationByProduct() throw (BadAuthException);
    void showUserProfile() throw (BadAuthException);
    void placeNewOrder() throw (BadAuthException);

    // admin operations
    void addNewCategory() throw (BadAuthException);
    void addNewProduct() throw (BadAuthException);
    void changeProductDetail() throw (BadAuthException);
    void disableUser() throw (BadAuthException);
    void displayMonthlyTrend() throw (BadAuthException);
    void deleteProduct() throw (BadAuthException);

```

```

        void displayUnprivilegedMenu() throw (BadAuthException);
        void displayAdminMenu() throw (BadAuthException);
        void wait();

        bool getNotEmptyLine(const string & msg, string *buffer);

protected:
    void enableSttyEcho(bool bValue = true);

public:
    UserMenu();
    ~UserMenu();

    void mainMenu();
    bool login();
    void display() throw (BadAuthException);
    void registerNewUser();
};

#endif /* __USERMENU_H__ */

```

UserMenu.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: UserMenu.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "UserMenu.h"
#include "Product.h"
#include "Order.h"
#include <termios.h>
#include <algorithm>

/**
 * @brief Default constructor
 */
UserMenu::UserMenu()
{
    LOG_CTOR();
    _currentUser = NULL;

    usr_operations[0] = &UserMenu::browseProductCatalog;
    usr_operations[1] = &UserMenu::showProductDetail;
    usr_operations[2] = &UserMenu::showConfigurationByProduct;
    usr_operations[3] = &UserMenu::showUserProfile;
    usr_operations[4] = &UserMenu::placeNewOrder;

    adm_operations[0] = &UserMenu::addNewCategory;
    adm_operations[1] = &UserMenu::addNewProduct;
    adm_operations[2] = &UserMenu::changeProductDetail;
    adm_operations[3] = &UserMenu::disableUser;
    adm_operations[4] = &UserMenu::displayMonthlyTrend;
    adm_operations[5] = &UserMenu::deleteProduct;

    LOG(3, "%d usr operations loaded\n", (int)usr_operations.size());
    LOG(3, "%d adm operations loaded\n", (int)adm_operations.size());
}

/**
 * @brief Default destructor
 *
 * Dealloc the logged user, if any.
 */
UserMenu::~UserMenu()
{
    LOG_DTOR();
    if (_currentUser)
        delete _currentUser;
}

/**
 * @brief Wait for user to press enter/return to continue.

```

```

    */
void UserMenu::wait()
{
    cin.clear();
    cout << "\nPress Enter/Return to continue...\n";
    cin.ignore(1, 0);
}

/**
    @brief Get input from standard input

    @param[in]    msg    String to prompt before getting input
    @param[out]   buffer Pointer to string to fill with data
    @return       True if operation was successful and buffer has been filled
    */
bool UserMenu::getNotEmptyLine(const string & msg, string *buffer)
{
    string tmp;

    //cin.clear();
    cout << msg;
    do {
        getline(cin, tmp);
    } while (tmp.size() == 0);

    *buffer = string(tmp);

    return true;
}

/**
    @brief Display user menu (for not-admin users)

    @throw BadAuthException if called without the correct level
    of authorization.
    */
void UserMenu::displayUnprivilegedMenu() throw (BadAuthException)
{
    if (!_currentUser)
        throw BadAuthException();

    int choice;

    do {
        system(CLEAR_SCREEN_CMD);
        cout << "USER MENU\n" <<
            "(1) Browse product catalog by category\n"
            "(2) View details of a chosen product\n"
            "(3) Browse all the configurations which include a given item\n"
            "(4) Browse user profile and all his order\n"
            "(5) Place a new order\n"
            "(0) LOGOUT\n\n"
            "Current user: " << _currentUser->fullName() <<
            "\n\nMake your choice: ";

        cin >> choice;
        if (choice > 0 && choice <= (int) usr_operations.size()) {
            int idx = choice - 1;
            op anOP = usr_operations[idx];
            (this->*anOP)();
        }
    } while (choice != 0);

    delete _currentUser, _currentUser = NULL;
}

/**
    @brief Browse product catalog by category

    Display all products belonging to a category or display them
    all if any category is chosen.

    @exception BadAuthException If called without being authenticated
    */
void UserMenu::browseProductCatalog() throw (BadAuthException)

```

```

{
    if (!_currentUser)
        throw BadAuthException();

    int cid;

    system(CLEAR_SCREEN_CMD);

    // list all available category and ask the user to choose one of them
    cout << "BROWSE PRODUCT CATALOG\n\nCATEGORY LIST\n";
    vector<Category *> & vc = Category::catalog();
    if (&vc && vc.size()) {
        for (int i=0; i < (int)vc.size(); i++) {
            cout << *(vc[i]) << endl;
        }
    }
    std::for_each(vc.begin(), vc.end(), deletePtr<Category>());
    cout << "\nEnter category ID [0 to browse all]: ";
    cin >> cid;
    cout << endl;
    printCatalog(cid);
    wait();
}

/**
@brief Display all product belonging to a category ID

@param[in]    aCid    Category ID

@throw BadAuthException if called without the correct level
        of authorization.
*/
void UserMenu::printCatalog(int aCid)
{
    // list all product belonging to selected category (or display them
    // all if zero was selected.
    vector<ProductProxy *> & v = ProductProxy::catalog(aCid);
    if (&v && v.size()) {
        for (int i=0; i < (int)v.size(); i++) {
            ProductProxy *pp = v[i];
            auto_ptr<Category> c = pp->getCategory();

            cout << "  |" << setfill(' ') << setw(5) << right
                 << pp->uniqueID() << " |" << left << setw(20)
                 << c->getName() << "|  "
                 << setw(35) << pp->getName() << "|  "
                 << setw(7) << right << pp->getPrice() << endl;
        }
        cout << endl;

        // dealloc container
        std::for_each(v.begin(), v.end(), deletePtr<ProductProxy>());
        delete &v;
    }
}

/**
@brief Display all details of a given product

@throw BadAuthException if called without the correct level
        of authorization.
*/
void UserMenu::showProductDetail() throw (BadAuthException)
{
    if (!_currentUser)
        throw BadAuthException();

    int pid;

    system(CLEAR_SCREEN_CMD);
    cout << "SHOW PRODUCT DETAIL\n\nProduct catalog\n";
    printCatalog();

    cout << "\nEnter product ID: ";
    cin >> pid;
}

```

```

        try {
            ProductProxy pp(pid);
            if (pp.isValid())
                cout << endl << pp;
        }
        catch (exception & e) {
            cout << "\nInvalid product ID\n";
        }
    }

    wait();
}

/**
 * @brief Show all product configurations which include a given product
 *
 * @throw BadAuthException if called without the correct level
 *        of authorization.
 */
void UserMenu::showConfigurationByProduct() throw (BadAuthException)
{
    if (!_currentUser)
        throw BadAuthException();

    int pid;

    system(CLEAR_SCREEN_CMD);
    cout << "BROWSE ALL CONFIGURATIONS WHICH INCLUDE A GIVEN ITEM\n\n"
    << "Choose a product from the following list\n\n";

    printCatalog();

    cout << "\nEnter product ID: ";
    cin >> pid;

    Product::showCompatibleProducts(pid);
    wait();
}

/**
 * @brief Show user profile and all his orders.
 *
 * @throw BadAuthException if called without the correct level
 *        of authorization.
 */
void UserMenu::showUserProfile() throw (BadAuthException)
{
    if (!_currentUser)
        throw BadAuthException();

    system(CLEAR_SCREEN_CMD);
    cout << "USER DETAILS\n" << *_currentUser << endl << endl;

    // obtain a vector containing all orders for current logged user
    vector<Order *> & orders = Order::ordersForUser(*_currentUser);

    if (orders.size()) {
        // we don't need to update this vector:
        // let's use a const_iterator (safe)
        vector<Order *>::const_iterator it;

        // iterate on all vector items and print them
        for (it = orders.begin(); it != orders.end(); it++) {
            Order *ord = (Order *) *it;
            cout << *ord << endl;

            map<int, int> &products = ord->products();
            map<int, int>::const_iterator mit;

            for (mit = products.begin(); mit != products.end(); mit++) {
                ProductProxy pp = ProductProxy( (*mit).first );
                cout << " (*) Product: " << left << setw(30)
                << pp.getName() << "Quantity: " << (*mit).second
                << endl;
            }
        }
    }
}

```

```

        cout << endl << endl;
        delete &products;
    }

    // dealloc container
    std::for_each(orders.begin(), orders.end(), deletePtr<Order>());
} else {
    cout << "[INFO] User didn't place any order at the moment.\n";
}

delete &orders;

wait();
}

/**
@brief Place a new order

Help the user to choose one or more items from available
products and add them to his basket; after last confirmation,
a new order is created.

@throw BadAuthException if called without the correct level
of authorization.
*/
void UserMenu::placeNewOrder() throw (BadAuthException)
{
    if (!_currentUser)
        throw BadAuthException();

    bool addMoreProducts = true;

    do {
        int aPid, aQty;

        system(CLEAR_SCREEN_CMD);
        cout << "CREATE ORDER\n\n";
        printCatalog();
        cout << "#" << (int)_currentUser->getBasket()->itemCount()
            << " item(s) in basket\n"
            << "\nEnter product ID to add [0 to end]: ";
        cin >> aPid;
        if (aPid == 0)
            addMoreProducts = false;
        else {
            cout << "How many pieces: ";
            cin >> aQty;

            Basket *bkt = _currentUser->getBasket();
            try {
                if (!bkt->addProduct(new ProductProxy(aPid), aQty)) {
                    cerr << "\nUnable to add requested item to basket.\n";
                    wait();
                }
            }
            catch (const exception & er) {
                cerr << "\n\nError: product not existent\n";
                wait();
            }
        }
    } while (addMoreProducts);

    if (_currentUser->getBasket()->itemCount() == 0) {
        cout << "No products were added to basket.\n";
        wait();
        return;
    }
    cout << "\nSUMMARY\n=====\n" << *(_currentUser->getBasket()) << endl
        << "TOTAL: " << _currentUser->getBasket()->total() << endl;
    _currentUser->placeOrder();
    wait();
}

/**
@brief Display all available admin operations.

```

```

        @throw BadAuthException if called without the correct level
        of authorization.
    */
    void UserMenu::displayAdminMenu() throw (BadAuthException)
    {
        if (!_currentUser)
            throw BadAuthException();

        int choice;

        do {
            system(CLEAR_SCREEN_CMD);
            cout << "ADMINISTRATION MENU\n" <<
                "(1) Add a new category\n"
                "(2) Add a new product\n"
                "(3) Change product details\n"
                "(4) Disable an user\n"
                "(5) Display monthly trend\n"
                "(6) Delete a product\n"
                "(0) EXIT\n\n"
                "Current admin: " << _currentUser->fullName() <<
                "\n\nMake your choice: ";
            cin >> choice;

            if (choice > 0 && choice <= (int) adm_operations.size()) {
                int idx = choice - 1;
                op anOP = adm_operations[idx];
                (this->*anOP)();
            }
        } while (choice != 0);
    }

    /**
    @brief Admin operation to create a new category of products.

    @exception BadAuthException If called without being authenticated
    */
    void UserMenu::addNewCategory() throw (BadAuthException)
    {
        if (!_currentUser)
            throw BadAuthException();

        Category *newCat;
        string name;
        bool bValue;

        system(CLEAR_SCREEN_CMD);
        cout << "ADD A NEW CATEGORY\n\n";
        bValue = getNotEmptyLine("Enter the name of new category "
                                "[0 to quit]: ", &name);
        if (!bValue || name == "0")
            cerr << "Operation aborted.\n";
        else {
            newCat = Category::factory(name);
            newCat->store();
            cout << "\nOperation was successful.\n";
        }
        wait();
    }

    /**
    @brief Admin operation to display monthly trend of sales

    @exception BadAuthException If called without being authenticated
    or not authorized
    */
    void UserMenu::displayMonthlyTrend() throw (BadAuthException)
    {
        AdminUser *anAdmin = dynamic_cast<AdminUser *> (_currentUser);

        if (!_currentUser || !anAdmin)
            throw BadAuthException();

        system(CLEAR_SCREEN_CMD);
    }

```

```

        cout << "MONTHLY TREND\n\n | YEAR | MONTH |          TOTAL |\n"
              << "-----\n";
        anAdmin->showMonthlyTrend();
        wait();
    }

    /**
     * @brief Admin operation to add a new product.
     * @exception BadAuthException If called without being authenticated
     */
    void UserMenu::addNewProduct() throw (BadAuthException)
    {
        if (!_currentUser)
            throw BadAuthException();

        string name, descr = "", cidStr, priceStr, qtyStr;
        float price;
        int qty, cid;
        bool success;

        system(CLEAR_SCREEN_CMD);
        cout << "ADD NEW PRODUCT\n"
              << "\nChoose the category of new product from list\n";

        vector<Category *> &v = Category::catalog();
        if (&v && v.size()) {
            for (int i=0; i < (int)v.size(); i++) {
                cout << *(v[i]) << endl;
            }
        } else {
            cerr << "There are no categories available." << endl
                  << "Create a new category, then add a product." << endl;
            wait();
            return;
        }

        // we can dealloc container, categories are only printed
        std::for_each(v.begin(), v.end(), deletePtr<Category>());
        delete &v;

        success = getNotEmptyLine("\nEnter category ID [0 to abort]: ",
                                  &cidStr);
        if (!success || cidStr == "0") {
            cerr << "\nOperation aborted.\n";
            wait();
            return;
        }

        getNotEmptyLine("Product name", &name);
        getNotEmptyLine("Product description", &descr);
        getNotEmptyLine("Price", &priceStr);
        getNotEmptyLine("Availability", &qtyStr);

        price = atof(priceStr.c_str());
        qty = atoi(qtyStr.c_str());
        cid = atoi(cidStr.c_str());

        if ((cid <= 0) || (price <= 0.0)) {
            cerr << "\nOperation aborted.\n";
        } else {
            Product *p = Product::factory(name, cid, price, descr, qty);
            p->store();
            delete p;
        }

        wait();
    }

    /**
     * @brief Admin operation to update an existing product.
     * @exception BadAuthException If called without being authenticated
     */
    void UserMenu::changeProductDetail() throw (BadAuthException)

```



```

{
    if (!_currentUser)
        throw BadAuthException();

    Product *p;
    int pid, attr;
    string attribute;

    // display the product catalog
    printCatalog();

    // identify the product to edit or abort the operation
    cout << "Enter product ID to change [0 to abort]: ";
    cin >> pid;
    if (pid == 0) {
        cout << "\nOperation aborted.\n";
        wait();
        return;
    }

    // retrieve the product from database
    p = Product::productByID(pid);
    if (!p) {
        cerr << "[ERR] Unable to find specified product. Abort.\n";
        wait();
        return;
    }
    cout << "\nREVIEW PRODUCT DETAIL\n " << *p << endl;

    do {
        cout << "\nCHOOSE WHICH ATTRIBUTE YOU WANT TO EDIT:\n"
              << "[1] Name\n[2] Description\n[3] Price\n[0] End\n\n"
              << "Enter attribute ID: ";
        cin >> attr;
        if (attr != 0) {
            if (getNotEmptyLine("Enter new value: ", &attribute)) {
                switch (attr) {
                    case 1:
                        p->setValueForKey(KEY_PRD_NAME, attribute);
                        break;
                    case 2:
                        p->setValueForKey(KEY_PRD_DESCR, attribute);
                        break;
                    case 3:
                        p->setValueForKey(KEY_PRD_PRICE, attribute);
                        break;
                    default:
                        attr = 0;
                        break;
                }
            }
        }
    } while (attr != 0);

    if (p->update())
        cout << "Operation successfully completed\n";
    else
        cerr << "Something went wrong. Operation aborted.\n";

    wait();
}

/**
@brief Admin operation to disable a registered user

@exception BadAuthException If called without being authenticated
or not authorized

@see AdminUser::changeUserPassword()
*/
void UserMenu::disableUser() throw (BadAuthException)
{
    AdminUser *anAdmin = dynamic_cast<AdminUser *> (_currentUser);

    if (!_currentUser || !anAdmin)

```

```

        throw BadAuthException();

system(CLEAR_SCREEN_CMD);
cout << "USER LOCK\n\n";

vector<User *> & uc = anAdmin->userList();
int pid, i;

for (i=0; i<(int)uc.size(); i++) {
    User *anUser = uc[i];
    cout << setw(4) << i+1 << " | " << anUser->fullName()
        << " (" << anUser->loginName() << ")" << endl;
}
cout << "\nEnter user ID to disable: ";
cin >> pid;
if ((pid <= 0)) {
    cerr << "\nInvalid selection\n";
    wait();
    return;
}

// Check if given user ID properly identifies an user: otherwise
// abort the operation.
User *anUser = User::userByID(pid);
if (!anUser) {
    cerr << "\nInvalid user ID entered. Operation aborted.\n";
} else {
    bool bValue = anAdmin->changeUserPassword(*anUser, "*LK*");

    cout << "\nOperation was " << (bValue?"":"NOT ")
        << "successfully completed\n";
}

// dealloc container
std::for_each(uc.begin(), uc.end(), deletePtr<User>());
delete &uc;

wait();
}

/**
@brief Delete a product (even if already sold).

@throw    BadAuthException If called by client programmer without
          any logged user

@see AdminUser::deleteProduct()
*/
void UserMenu::deleteProduct() throw (BadAuthException)
{
    AdminUser *anAdmin = dynamic_cast<AdminUser *> (_currentUser);

    if (!_currentUser || !anAdmin)
        throw BadAuthException();

    int pid;

    system(CLEAR_SCREEN_CMD);
    cout << "PRODUCT DELETE\n\n";

    // Display product catalog
    printCatalog();

    cout << "\nEnter product ID to delete [0 to abort] :";
    cin >> pid;
    if (pid <= 0) {
        cerr << "Operation aborted.\n";
    } else {
        bool b = anAdmin->deleteProduct(pid);
        if (b)
            cout << "\nProduct successfully deleted.\n";
        else
            cerr << "\nSomething went wrong. Operation aborted.\n";
    }
    wait();
}

```

```

}

/**
 @brief Display a menu to the screen, according to authorization
        level of current user.

 @throw    BadAuthException If called by client programmer without
           any logged user

 @see displayAdminMenu(), displayUnprivilegedMenu()
 */
void UserMenu::display() throw (BadAuthException)
{
    if (!_currentUser)
        throw BadAuthException();

    if (_currentUser->isAdmin())
        displayAdminMenu();
    else
        displayUnprivilegedMenu();
}

/**
 @brief User login

 Ask user to enter his credentials and test the access: if
 successful, an instance of User is allocated and assigned to
 _currentUser.

 @return    True if user logged in successfully (credentials were
           valid).

 @see User::login(), enableSttyEcho()
 */
bool UserMenu::login()
{
    string username;
    string passwd;

    do {
        system(CLEAR_SCREEN_CMD);
        cout << "USER LOGIN\nLogin [0 to quit]: ";
        cin >> username;
        if (username == "0") {
            cerr << "\nOperation aborted.\n";
            wait();
            return false;
        }

        cout << "Password          : ";
        enableSttyEcho(false);
        cin >> passwd;
        enableSttyEcho();

        if ((_currentUser = User::login(username, passwd))
            return true;

        cerr << "\n\nInvalid username/password.\n";
        wait();
    } while (1);

    return false;
}

/**
 @brief Main menu

 Displays the main menu: user is allowed to login into the
 system, register himself as new user or quit the application.

 @see login(), registerNewUser(), display()
 */
void UserMenu::mainMenu()
{
    int choice;

```

```

do {
    system(CLEAR_SCREEN_CMD);
    cout << "MAIN MENU\n\n(1) User login\n(2) New user registration\n"
        << "(0) Quit\n\nMake your selection: ";
    cin >> choice;
    if (choice == 1) {
        if (login())
            display();
    } else if (choice == 2)
        registerNewUser();
    } while (choice != 0);
}

/**
@brief Give the user the opportunity to register as new user

@see User::factory
*/
void UserMenu::registerNewUser()
{
    string name, surname, login, passwd, address, city;
    bool success = false;

    system(CLEAR_SCREEN_CMD);
    cout << "NEW USER REGISTRATION\n\n";

    success = getNotEmptyLine("Enter name      : ", &name);
    if (success) {
        getNotEmptyLine("Enter surname : ", &surname);
        getNotEmptyLine("Enter address : ", &address);
        getNotEmptyLine("Enter city    : ", &city);
        getNotEmptyLine("Enter login   : ", &login);
        getNotEmptyLine("Enter password: ", &passwd);

        if (User::factory(name, surname, login, passwd, address, city) ==
            NULL)
            cerr << "\nUnable to register new user."
                << "Try to review your input data.\n";
        else
            cout << "\nOperation successfully completed.\n";
    }

    wait();
}

/**
@brief Enable or disable STTY echo

This function disables (or enables) the echo of input coming
from the STTY, such as the keyboard; calling this method is
useful when the user is asked to enter his password and this
data don't need to be echo-ed to screen.

@see login()
*/
void UserMenu::enableSttyEcho(bool bValue)
{
    struct termios tty;

    tcgetattr(STDIN_FILENO, &tty);
    if (!bValue)
        tty.c_lflag &= ~ECHO;
    else
        tty.c_lflag |= ECHO;

    (void) tcsetattr(STDIN_FILENO, TCSANOW, &tty);
}

```

main.cpp

```

/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 */

```

```

* $Id: main.cpp 74 2010-08-13 16:23:12Z unixo $
*/

#include "Database.h"
#include "UserMenu.h"
#include "CommandLine.h"

int debugLevel = 0;

int main (int argc, char * const argv[])
{
    // parse command line
    CommandLine cmd(argc, argv);
    if (cmd.isFault())
        return 1;

    // set debug level
    debugLevel = cmd.debugLevel();

    // get an instance of the database
    Database& db = Database::instance();
    db.setServer(cmd.dbServer());
    db.setUser(cmd.dbUser());
    db.setPassword(cmd.dbPasswd());

    // try to connect to database: halt the program if not
    if (!db.connect()) {
        cerr << "Unable to connect to database\n";
        return 2;
    }

    // display main menu
    UserMenu menu;
    menu.mainMenu();

    return 0;
}

```

Makefile

```

#####
# Ferruccio Vitale (unixo@devzero.it)
# 20/07/2010
#
# Applied Computer Science - Urbino University "Carlo Bo"
# Software Engineering
# Professor Edoardo Bonta'
# Academic Year 2009 - 2010
#####

# Use this definition to add debug information
# CFLAGS = -I/usr/local/include/mysql++ -I/usr/local/mysql/include \
#         -O0 -ggdb

# Use this under MacOS X
# CFLAGS = -I/usr/local/include/mysql++ -I/usr/local/mysql/include \
#         -O -Wall -Werror

CPP      = g++
CFLAGS   = -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror \
          -Wno-unused-result
LIBS     = -L/usr/local/lib -lmysqlpp
OBJS     = Basket.o Category.o Database.o ManagedObject.o Order.o \
          Product.o User.o UserMenu.o DataModel.o Observable.o \
          CommandLine.o

.PHONY: all
all: ec++ white-box

ec++: $(OBJS) main.o
      ${CPP} ${LIBS} ${OBJS} main.o -o ec++

white-box: $(OBJS) white-box.o
      ${CPP} ${LIBS} ${OBJS} white-box.o -o white-box

```

```
.cpp.o:
    ${CPP} ${CFLAGS} ${INCLUDES} -c $<

.PHONY: clean
clean:
    rm -f *.o core *~ ec++ white-box
```

WHITE BOX TEST

white-box.h

```
/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: white-box.h 74 2010-08-13 16:23:12Z unixo $
 */

#include "common.h"
#include "Observer.h"
#include "Observable.h"
#include "User.h"

class TestObserver : public Observer
{
private:
    string _name;

public:
    TestObserver(string s) : _name(s) { };

    virtual void willChangeValueForKey(string k, Observable *) {
        cout << "[TestObserver::willChangeValueForKey] key: '" << k
              << "' observer: '" << _name << "'\n";
    };

    virtual void didChangeValueForKey(string k, Observable *) {
        cout << "[TestObserver::didChangeValueForKey] key: '" << k
              << "' observer: '" << _name << "'\n";
    };

    string & toString() { return _name; };
};

class TestObservable : public Observable
{
private:
    string _attr;

public:
    void setKey(string &aValue) {
        willChangeValueForKey("attr");
        cout << "[TestObservable::setKey] setting new value\n";
        _attr = aValue;
        didChangeValueForKey("attr");
    };
};
```

white-box.cpp

```
/*
 * Copyright (c) 2010 Ferruccio Vitale <unixo@devzero.it>
 * All rights reserved.
 *
 * $Id: white-box.cpp 74 2010-08-13 16:23:12Z unixo $
 */

#include "white-box.h"
#include "CommandLine.h"

int debugLevel = 3;
```

```

/**
    @brief Test key/value observing framework
 */
void testObserver()
{
    cout << "OBSERVER/OBSERVABLE TEST #1\n";

    TestObservable *master = new TestObservable();
    TestObserver *slave1 = new TestObserver("slave1");
    TestObserver *slave2 = new TestObserver("slave2");
    string str1 = "value1", str2 = "value2";

    master->addObserver("attr", *slave1);
    master->addObserver("attr", *slave2);
    master->setKey(str1);

    master->removeObserver("attr", *slave1);
    master->setKey(str2);

    delete master, delete slave1, delete slave2;

    cout << endl;
}

/**
    @brief Test DataModel and Database classes
 */
void testDataModel()
{
    cout << "DATAMODEL TEST #2\n";

    User *usr1 = User::login("unixo", "secret");
    User *usr2 = User::userByID(2);

    delete usr1, delete usr2;
}

int main (int argc, char * const argv[])
{
    CommandLine cmd(argc, argv);
    if (cmd.isFault())
        return 1;

    // get an instance of the database
    Database& db = Database::instance();
    db.setServer(cmd.dbServer());
    db.setUser(cmd.dbUser());
    db.setPassword(cmd.dbPasswd());
    if (!db.connect()) {
        cerr << "Unable to connect to database\n";
        return 2;
    }

    // call unit tests
    testObserver();
    testDataModel();

    return 0;
}

```

Testing

Application testing is one of the most important development phase, during which many different types of checks are applied in order to verify that the application doesn't suffer of any kind of problem, bugs or failures.

To realize a solid software we used different techniques, such as:

- *defensive programming*: many crucial blocks of code are enriched with the `assert()` statement, a construct common to many different languages, which simply checks that a condition is verified or abort signal is immediately sent to the program;
- *non-blocking debug messages*: some macros have been defined in order to log debug messages and help the programmer to find the right block of instructions that may fail; these messages are automatically disabled unless the user increases the debug level by specifying the `-d` parameter in the command line, at application start up;
- *const_iterator vs iterator*: code safeness is also ensured by the use of `const_iterator`, whenever possible, to enumerate containers; this approach, together with returning a reference instead of a pointer to a class instance, reduces the undesired possibility of altering data structures;
- *compiler warnings*: two g++ parameters have been used to be sure that the generated code conforms as much as possible to standards; the first, `-Wall`, enables all compiler warnings and the second, `-Werror`, treats warnings as error so that even a warning would stop the compiler.

Due to `mySQL++` libraries, it's not possible to use the parameter `"Weffc++"`.

The following are examples of the above techniques.

```
vector<User *> & AdminUser::userList()
{
    [omissis]
    if (row[KEY_USR_ADMIN] == "1")
        anUser = new AdminUser(row);
    else
        anUser = new NormalUser(row);
    assert(anUser);
    users->push_back(anUser);
    [omissis]
}
```

Figure 9. `assert()` example

This block code explains the use the `assert()` macro: whenever the local variable `anUser` is NULL, i.e. the `new()` operator failed to create the class instance, the condition is not verified and the programs halts; it's

advisable to use this approach to test condition that *should* never happen and so they don't need to be handled by some code.

```
[ManagedObject.cpp:28] [ManagedObject] ctor called
[DataModel.cpp:13] [DataModel] ctor called
[DataModel.cpp:56] [keysForEntity] Data model for entity 'users'
                    loaded.
[User.cpp:48] [User] ctor called
[Basket.cpp:14] [Basket] ctor called
[User.cpp:321] [NormalUser] ctor called
```

Figure 10. Debug messages example

This second example shows the output of macros `LOG_CTOR()` and `LOG()` which are printed only when the debug level is increased by the user; note that the macro `LOG_xx()` also shows which file contains the instruction, the line number and the invoked method; an example of this usage is shown in the following class constructor:

```
ManagedObject::ManagedObject(string anEntityName)
{
    LOG_CTOR();
    initEntity(anEntityName);
}
```

UNIT TESTING

An ad-hoc module has been developed in order to test the most important features offered by the library, the key/value observing and the database/data-model infrastructure. In line with *white-box testing*, a main module has also been developed, to act as driver with the aim of create specific data structures and pass them proper parameters.

So, the project also comes with a module that, along with an high debug level, shows the proper message flow between classes and test the following functionalities.

Key/value observing

To complete the first test, we needed to define two new classes, *TestObservable* and *TestObserver*: the former, which acts as master, offers an attribute that it's possible to change, while the latter, the slave, is added to *master* observer list.

The test shows that, whenever the master attributed is changed, all observers are notified before the key is changing and after the change has done.

```
[Observable.cpp:13] [Observable] ctor called
[Observable.cpp:36] [addObserver] Added new observer: slave1
[Observable.cpp:36] [addObserver] Added new observer: slave2
[Observable.cpp:102] [willChangeValueForKey] willChangeValueForKey('attr')
[TestObserver::willChangeValueForKey] key: 'attr' observer: 'slave1'
[TestObserver::willChangeValueForKey] key: 'attr' observer: 'slave2'
[TestObservable::setKey] setting new value
[Observable.cpp:123] [didChangeValueForKey] didChangeValueForKey('attr')
[TestObserver::didChangeValueForKey] key: 'attr' observer: 'slave1'
[TestObserver::didChangeValueForKey] key: 'attr' observer: 'slave2'
[Observable.cpp:63] [removeObserver] Removed observer: slave1
[Observable.cpp:102] [willChangeValueForKey] willChangeValueForKey('attr')
[TestObserver::willChangeValueForKey] key: 'attr' observer: 'slave2'
[TestObservable::setKey] setting new value
[Observable.cpp:123] [didChangeValueForKey] didChangeValueForKey('attr')
[TestObserver::didChangeValueForKey] key: 'attr' observer: 'slave2'
```

```
[Observable.cpp:18] [~Observable] dtor called
```

Figure 11. Key/Value observe testing

Two instances of *TestObserver* are created, added as *TestObservable* observer and then the attribute of the latter is changed: both instances are notified of the events; the first instance of *TestObserver* is therefore unregistered as observer and again the attribute is changed: this time, only the second instance is notified.

Database and DataModel functionalities

This second unit checks that, whenever a table schema is requested to the singleton *DataModel*, only the first request is forwarded to the database, by creating a connection and executing a SELECT command: subsequent requests must return cached value.

```
[Observable.cpp:13] [Observable] ctor called
[ManagedObject.cpp:32] [ManagedObject] ctor called
[DataModel.cpp:13] [DataModel] ctor called
[DataModel.cpp:56] [keysForEntity] Data model for entity 'users' loaded.
[User.cpp:52] [User] ctor called
[User.cpp:249] [AdminUser] ctor called
[Observable.cpp:13] [Observable] ctor called
[ManagedObject.cpp:32] [ManagedObject] ctor called
[DataModel.cpp:26] [keysForEntity] Data model for entity 'users' found in
cache.
[User.cpp:52] [User] ctor called
[Basket.cpp:14] [Basket] ctor called
[User.cpp:352] [NormalUser] ctor called
[User.cpp:60] [~User] dtor called
[ManagedObject.cpp:51] [~ManagedObject] dtor called
[Observable.cpp:18] [~Observable] dtor called
[User.cpp:357] [~NormalUser] dtor called
[User.cpp:60] [~User] dtor called
[ManagedObject.cpp:51] [~ManagedObject] dtor called
[Observable.cpp:18] [~Observable] dtor called
[DataModel.cpp:18] [~DataModel] dtor called
[Database.cpp:34] [~Database] dtor called
```

Figure 12. DataModel testing

Lines marked in bold show the right behaviour of the cache. Note that two different managed objects have been instantiated, *NormalUser* and *AdminUser*, but they both share the same data model, derived from class *User*, so the request type is the same.

The classes *Database* and *ManagedObject* are also tested indirectly, when we request to fetch two users from the store.

VALIDATION TESTING

Even if all these techniques help the programmer to build software as solid as possible, it's clear that they're not sufficient to ensure that the output of development chain responds to requirements.

So, another and deeper approach of analysis need to be applied: the *black box testing*, particularly useful from the point of view of interaction

with the end-user. We start from the analysis of the application use cases, which describe all user requirements, in order to define the use test.

Test case #1: database connection

As the program is started, it immediately tries to connect to the database with default credentials; whenever the database is not available or given credentials are wrong, the application fails to connect to MySQL and aborts.

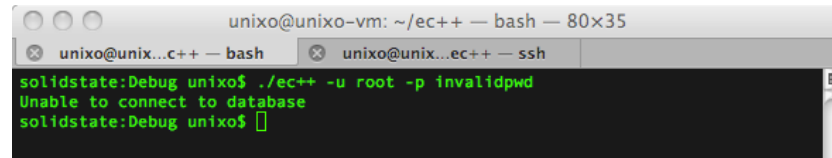
A terminal window titled 'unixo@unixo-vm: ~/ec++ -- bash -- 80x35'. It shows the command './ec++ -u root -p invalidpwd' being executed. The output is 'solidstate:Debug unixo\$ Unable to connect to database' followed by a new prompt 'solidstate:Debug unixo\$'.

Figure 13. The application fails to connect to database

Test case #2: login

When the program is launched, the main menu is shown, allowing the user to log into the system or register himself if he's a new user. If the user chooses to log in, the system asks him to enter his credential.

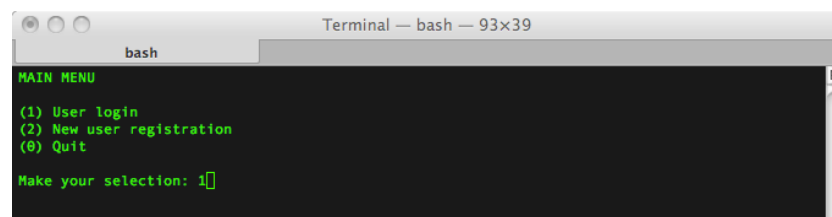
A terminal window titled 'Terminal -- bash -- 93x39'. It displays a 'MAIN MENU' with options: (1) User login, (2) New user registration, and (0) Quit. Below the menu, it says 'Make your selection: 1' followed by a cursor.

Figure 14. Main menu at start up

When user entered the credential, the system must check if they are valid and the user is authorized to log in.

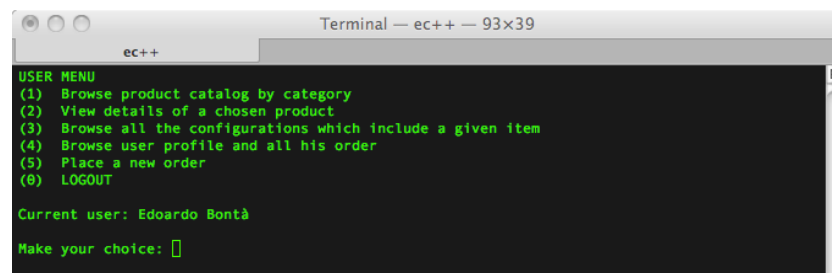
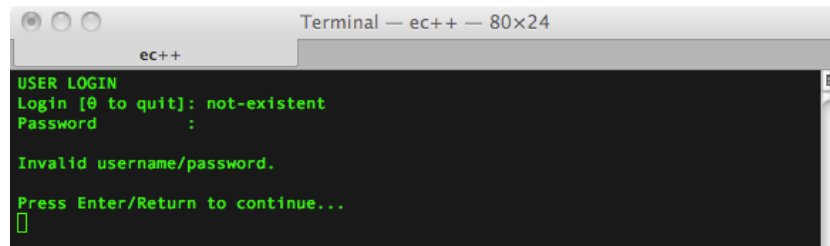
A terminal window titled 'Terminal -- ec++ -- 93x39'. It displays a 'USER MENU' with options: (1) Browse product catalog by category, (2) View details of a chosen product, (3) Browse all the configurations which include a given item, (4) Browse user profile and all his order, (5) Place a new order, and (0) LOGOUT. Below the menu, it says 'Current user: Edoardo Bontà' and 'Make your choice: ' followed by a cursor.

Figure 15. User menu after successful login

Otherwise, if data entered are not valid, the system returns an error message indicating that the combination of username and password was not valid (to avoid user enumeration).



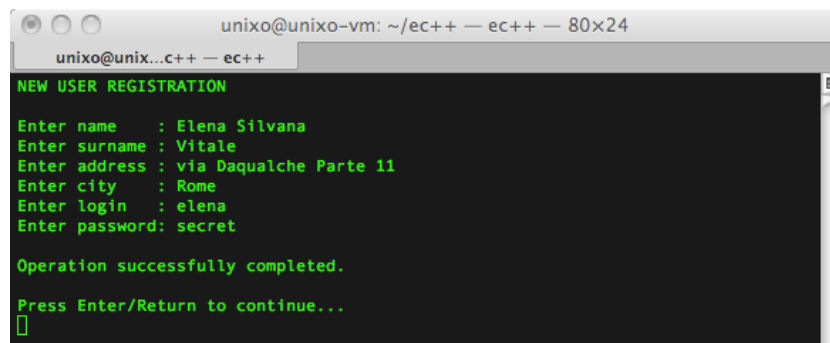
```
Terminal — ec++ — 80x24
ec++
USER LOGIN
Login [0 to quit]: not-existent
Password      :
Invalid username/password.
Press Enter/Return to continue...
█
```

Figure 16. Failed login

Test case #3: register

As requirement, only registered users can access the system: so, this feature permits new user to register and interact with the service. At start up, it's possible to enter this form from the main menu.

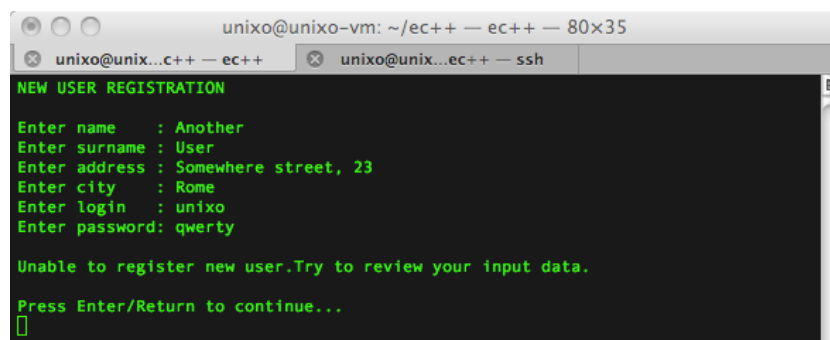
The user is asked to enter some personal data, such as the shipping address and the credential he will use to log in. After this phase successfully ends, the new user can access the system.



```
unixo@unixo-vm: ~/ec++ — ec++ — 80x24
unixo@unix...c++ — ec++
NEW USER REGISTRATION
Enter name      : Elena Silvana
Enter surname   : Vitale
Enter address   : via Daqualche Parte 11
Enter city      : Rome
Enter login     : elena
Enter password  : secret
Operation successfully completed.
Press Enter/Return to continue...
█
```

Figure 17. Registration of a new user

As requirement, a new user must choose an unique username, i.e. different from all other existing values; if the user doesn't respect this requirement, registration process fails.



```
unixo@unixo-vm: ~/ec++ — ec++ — 80x35
unixo@unix...c++ — ec++  unixo@unix...ec++ — ssh
NEW USER REGISTRATION
Enter name      : Another
Enter surname   : User
Enter address   : Somewhere street, 23
Enter city      : Rome
Enter login     : unixo
Enter password  : qwerty
Unable to register new user.Try to review your input data.
Press Enter/Return to continue...
█
```

Figure 18. Registration failed (not unique username)

Test case #4: quit

Whenever a user wants to quit the application, even if not logged in, the *quit* menu item allows to return to the shell. If the user already logged in, he must first logout and then quit the application.

```

Terminal — bash — 93x39
bash
MAIN MENU
(1) User login
(2) New user registration
(0) Quit
Make your selection: 0
bash-3.2$

```

Figure 19. Quit to shell

Test case #5: place a new order

The main goal of the system is order placing by users. As logged in, a user can access the form of new order creation: the system will prompt available products and let the user choose which product and how many pieces need.

This process continues until the user enters the proper value (zero) and the order is created.

```

unixo@unixo-vm: ~/ec++ — ec++ — 80x41
unixo@unix...c++ — ec++
CREATE ORDER
| 1 |Notebook      | Apple MacBook Pro      | 2150
| 2 |Notebook      | Apple iMac 24          | 1200
| 3 |Monitor       | External monitor Acer  | 800
| 4 |Monitor       | Monitor LCD Full HD    | 1230
| 5 |Keyboards      | Italian keyboard       | 15
| 6 |Keyboards      | Bluetooth keyboard     | 45
| 8 |PCCARD         | TV Card                | 243
| 9 |PCCARD         | NETGEAR WG511 Wireless-G | 30
|10 |Hard disk      | HD esterno 1.5Tb       | 160
|11 |Computer Desktop | PACKARD BELL iMax mini | 249
|12 |Computer Desktop | FUJITSU-SIEMENS Amilo Li 3740 | 249
|13 |Computer Desktop | HP Pavilion Elite m9441.it | 369
|14 |Audio          | LOGITECH LS11          | 19
|15 |Audio          | HERCULES XPS 2.0 Lounge | 34
|26 |Wi-Fi Antennas  | Extra range III        | 175
|27 |PCCARD         | USB to PCCARD converter | 25

#4 item(s) in basket
Enter product ID to add [0 to end]: 0

SUMMARY
=====
Apple MacBook Pro      | 2
TV Card                | 1
USB to PCCARD converter | 1

TOTAL: 2418
Press Enter/Return to continue...

```

Figure 20. New order placing

Of course, before placing the order, the system checks the contents of user basket: a new order is created only if:

- the basket is not empty;
- user requested to add valid products;
- user didn't try to add more items than their availability.

if at least one of above conditions is not guaranteed, the process is aborted, as shown in the following picture.

```

unixo@unixo-vm: ~/ec++ — ec++ — 80x41
unixo@unix...c++ — ec++
CREATE ORDER

| 1 |Notebook      | Apple MacBook Pro      | 2150
| 2 |Notebook      | Apple iMac 24          | 1200
| 3 |Monitor       | External monitor Acer  | 800
| 4 |Monitor       | Monitor LCD Full HD   | 1230
| 5 |Keyboards     | Italian keyboard       | 15
| 6 |Keyboards     | Bluetooth keyboard     | 45
| 8 |PCCARD        | TV Card                | 243
| 9 |PCCARD        | NETGEAR WG511 Wireless-G | 30
| 10 |Hard disk     | HD esterno 1.5Tb       | 160
| 11 |Computer Desktop | PACKARD BELL iMax mini | 249
| 12 |Computer Desktop | FUJITSU-SIEMENS Amilo Li 3740 | 249
| 13 |Computer Desktop | HP Pavilion Elite m9441.it | 369
| 14 |Audio         | LOGITECH LS11          | 19
| 15 |Audio         | HERCULES XPS 2.0 Lounge | 34
| 26 |Wi-Fi Antennas | Extra range III        | 175
| 27 |PCCARD        | USB to PCCARD converter | 25

#0 item(s) in basket

Enter product ID to add [0 to end]: 1
How many pieces: 100

Unable to add requested item to basket.

Press Enter/Return to continue...

```

Figure 21. User requested more items than available

```

Terminal — ec++ — 80x24
ec++

| 3 |Monitor       | External monitor Acer  | 800
| 4 |Monitor       | Monitor LCD Full HD   | 1230
| 5 |Keyboards     | Italian keyboard       | 15
| 6 |Keyboards     | Bluetooth keyboard     | 45
| 8 |PCCARD        | TV Card                | 243
| 9 |PCCARD        | NETGEAR WG511 Wireless-G | 30
| 11 |Computer Desktop | PACKARD BELL iMax mini | 249
| 12 |Computer Desktop | FUJITSU-SIEMENS Amilo Li 3740 | 249
| 13 |Computer Desktop | HP Pavilion Elite m9441.it | 369
| 14 |Audio         | LOGITECH LS11          | 19
| 15 |Audio         | HERCULES XPS 2.0 Lounge | 34
| 26 |Wi-Fi Antennas | Extra range III        | 175
| 27 |PCCARD        | USB to PCCARD converter | 25

#0 item(s) in basket

Enter product ID to add [0 to end]: 40
How many pieces: 1

Error: product not existent

Press Enter/Return to continue...

```

Figure 22. User tried to add a non-existent product

Test case #6: browse by category

User requirements suggest to group available products into different categories, so that is easier to browse catalog.

By specifying the proper value (zero), it's possible to browse all product anyway.

```

Terminal — ec++ — 93x39
ec++
BROWSE PRODUCT CATALOG

CATEGORY LIST
7 | Audio
6 | Computer Desktop
5 | Hard disk
3 | Keyboards
2 | Monitor
1 | Notebook
4 | PCCARD
8 | pluto

Enter category ID [0 to browse all]: 2

| 3 | Monitor          | External monitor Acer          |
| 4 | Monitor          | Monitor LCD Full HD           |

Press Enter/Return to continue...

```

Figure 23. Browse monitor category

Test case #7: view product detail

Beside viewing the list of all available products, a logged user can also ask for details of a specific product. The system must let the user to choose an item from the list and display all related data.

```

Terminal — ec++ — 80x33
ec++
SHOW PRODUCT DETAIL

Product catalog
| 1 | Notebook          | Apple MacBook Pro              |
| 2 | Notebook          | Apple iMac 24                  |
| 3 | Monitor            | External monitor Acer          |
| 4 | Monitor            | Monitor LCD Full HD           |
| 5 | Keyboards           | Italian keyboard               |
| 6 | Keyboards           | Bluetooth keyboard             |
| 8 | PCCARD              | TV Card                        |
| 9 | PCCARD              | NETGEAR WG511 Wireless-G      |
| 10 | Hard disk           | HD esterno 1.5Tb               |
| 11 | Computer Desktop    | PACKARD BELL iMax mini         |
| 12 | Computer Desktop    | FUJITSU-SIEMENS Amilo Li 3740  |
| 13 | Computer Desktop    | HP Pavilion Elite m9441.it     |
| 14 | Audio               | LOGITECH LS11                  |
| 15 | Audio               | HERCULES XPS 2.0 Lounge        |
| 17 | pluto               | ferruccio                      |
| 18 | Audio               | 4                              |

Enter product ID: 2

PRODUCT DETAIL
Category   : 1 | Notebook
Name       : Apple iMac 24
Description : Desktop monitor 24 inches + HDD 640Gb
Price      : 1200
Availability: 2

Press Enter/Return to continue...

```

Figure 24. Details of a specific product

Test case #8: browse user orders

It's possible to see any previous placed order by accessing user profile; along with his personal data, the list of orders is displayed: each of them comes with the date of the order and the list of bought products.

```

Terminal — ec++ — 80x33
ec++
USER DETAILS
Name : Edoardo Bontà
Login : eddy
Address: unknown (unknown)

ORDER DETAIL
=====
Buyer : Edoardo Bontà
Date : 2010-07-07 18:20:58
Total : 2393

(*) Product: Apple MacBook Pro          Quantity: 2
(*) Product: TV Card                    Quantity: 1

Press Enter/Return to continue...

```

Figure 25. User profile and his orders

Test case #9: show product configurations

As requirement, when a user chooses a product from the catalog, the system must suggest one or more prebuilt configurations which include the given product.

After logged in, the user chooses the proper menu item and the system prompts with product catalog: the user enters the desired product ID and the configurations, if any, are displayed.

```

unixo@unixo-vm: ~/ec++ — ec++ — 114x34
unixo@unixo-vm: ~/ec++ — ec++
unixo@unixo-vm: ~/ec++ — ssh
| 14 | Audio | LOGITECH LS11 | 19
| 15 | Audio | HERCULES XPS 2.0 Lounge | 34
| 26 | Wi-Fi Antennas | Extra range III | 175
| 27 | PCCARD | USB to PCCARD converter | 25

Enter product ID: 3

CONFIGURATION DETAIL
=====
| Total Discount Offer | 800 |

Configuration includes the following products:
| 3 | 2 | External monitor Acer | Width 32 inches | 800 | 1 | 0 | Monitor |
| 5 | 3 | Italian keyboard | 102 keys | 15 | 1 | 0 | Keyboards |
| 6 | 3 | Bluetooth keyboard | NULL | 45 | 10 | 0 | Keyboards |

CONFIGURATION DETAIL
=====
| Desktop 4 everyone | 1094 |

Configuration includes the following products:
| 3 | 2 | External monitor Acer | Width 32 inches | 800 | 1 | 0 | Monitor |
| 6 | 3 | Bluetooth keyboard | NULL | 45 | 10 | 0 | Keyboards |
| 11 | 6 | PACKARD BELL iMax mini | HD 160GB - RAM 1GB | 249 | 11 | 0 | Computer Desktop |

Press Enter/Return to continue...

```

Figure 26. Product #3 is included in two different configurations

Test case #10: login

This test case produces the same effects of test case #1.

After user successfully logged in, the admin menu is displayed instead of user one.


```

unixo@unixo-vm: ~/ec++ -- ec++ -- 80x42
unixo@unix...c++ -- ec++
unixo@unix...ec++ -- ssh
ADMINISTRATION MENU
(1) Add a new category
(2) Add a new product
(3) Change product details
(4) Disable an user
(5) Display monthly trend
(6) Delete a product
(0) EXIT

Current admin: Ferruccio Vitale

Make your choice: 

```

Figure 27. Administration menu

Test case #11: trend monthly sales

Even if it's not a user requirement, an administrator need to know and analyze the trend of all sales: by selecting this menu item, the system will display this data grouped by years and, for each year, by months.

```

Terminal -- ec++ -- 80x33
ec++
MONTHLY TREND

| YEAR | MONTH | TOTAL |
-----|-----|
| 2009 | 07 | 830 |
| 2009 | 10 | 2300 |
| 2009 | 12 | 3705 |
| 2010 | 01 | 800 |
| 2010 | 07 | 2393 |

Press Enter/Return to continue...

```

Figure 28. Monthly trend of sales

Test case #12: add new product

Adding a new product to the catalog means increase the offer and, possibly, increase the sales. After logging in, the administrator can choose to add a new product: the system will ask for its name and description, along with the price and stock availability.

If data are properly entered, the system will create a new record in the store.

```

unixo@unixo-vm: ~/ec++ -- ec++ -- 80x41
unixo@unix...c++ -- ec++
ADD NEW PRODUCT

Choose the category of new product from list
1 | Notebook
2 | Monitor
3 | Keyboards
4 | PCCARD
5 | Hard disk
6 | Computer Desktop
7 | Audio
28 | Wi-Fi Antennas

Enter category ID [0 to abort]: 4
Product name : USB to PCCARD converter
Product description : USB 2.0 compatible converter
Price : 25
Availability : 10

Press Enter/Return to continue...

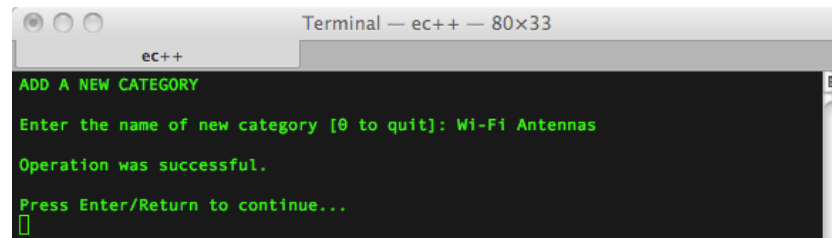
```

Figure 29. New product creation

Test case #13: add new category

Categories are groups of similar products, created to easily browse the catalog: an administrator need of course to create as many categories as needed.

After logged in, the administrator chooses the right menu item and the system asks for the new category name: if this process successfully ends, the new category is created and new products could be added to it.

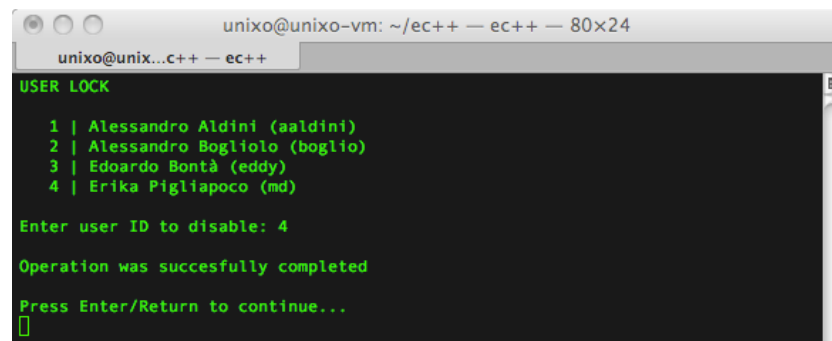


```
Terminal — ec++ — 80x33
ec++
ADD A NEW CATEGORY
Enter the name of new category [0 to quit]: Wi-Fi Antennas
Operation was successful.
Press Enter/Return to continue...
█
```

Figure 30. New category created

Test case #14: disable user

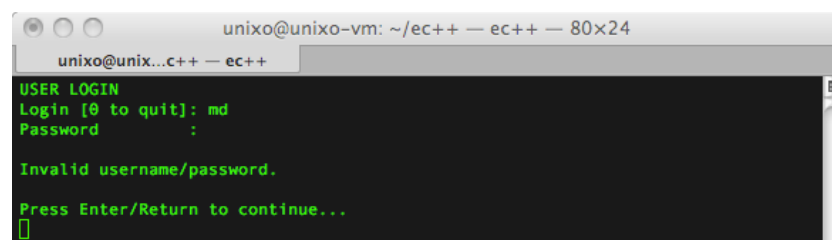
Whenever an administrator wants to inhibits a user from logging into the system, he can access the proper menu item from administration menu: the system will display all registered users and asks for the user ID to disable; if the input is properly entered, the user password is reset and he can log until the administrator changes his password.



```
unixo@unixo-vm: ~/ec++ — ec++ — 80x24
unixo@unix...c++ — ec++
USER LOCK
1 | Alessandro Aldini (aaldini)
2 | Alessandro Bogliolo (boglio)
3 | Edoardo Bontà (eddy)
4 | Erika Pigliapoco (md)
Enter user ID to disable: 4
Operation was succesfully completed
Press Enter/Return to continue...
█
```

Figure 31. User with UID #4 can't login anymore

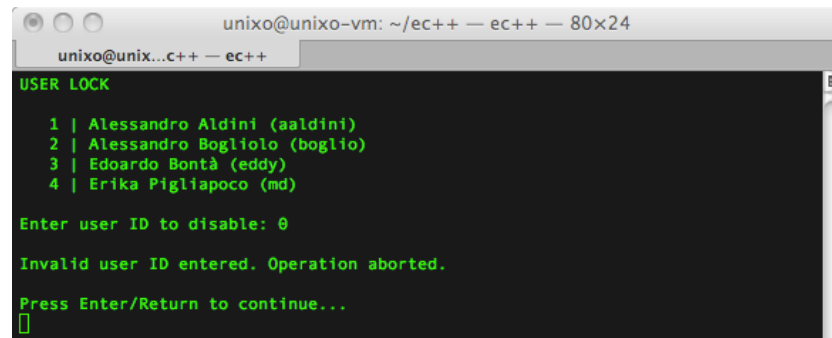
If the user tries to access again, the system will deny the operation.



```
unixo@unixo-vm: ~/ec++ — ec++ — 80x24
unixo@unix...c++ — ec++
USER LOGIN
Login [0 to quit]: md
Password :
Invalid username/password.
Press Enter/Return to continue...
█
```

Figure 32. The system prevents user from log in

Of course, the system detects if the user ID identifies a valid user or not: if an error is detected, the whole operation is aborted.



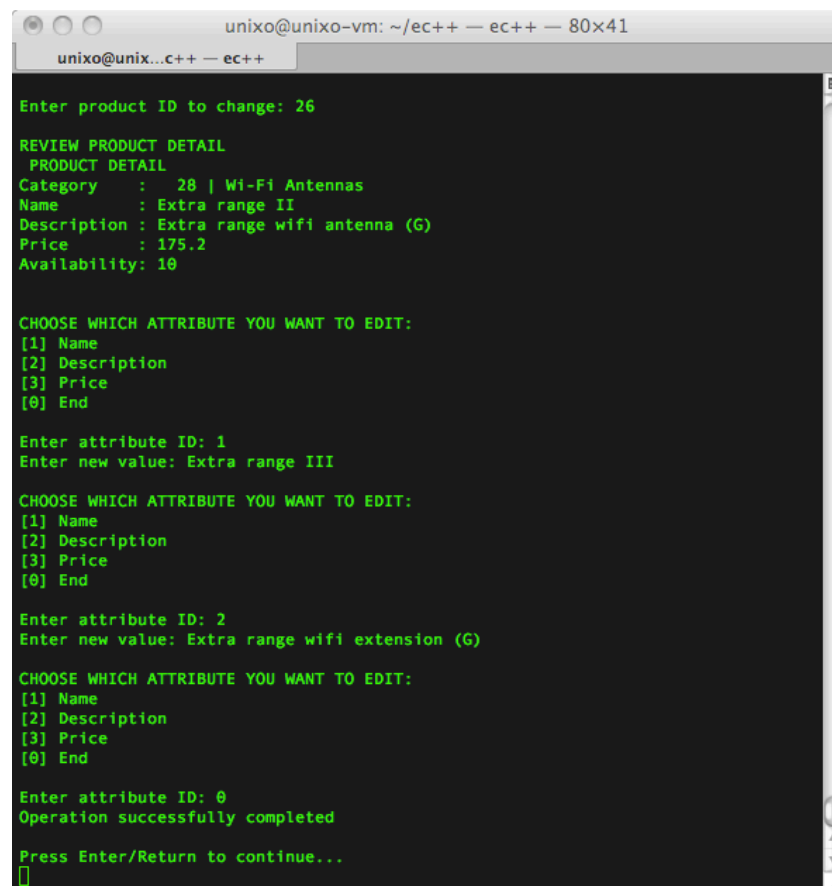
```
unixo@unixo-vm: ~/ec++ — ec++ — 80x24
unixo@unix...c++ — ec++
USER LOCK
1 | Alessandro Aldini (aaldini)
2 | Alessandro Bogliolo (boglio)
3 | Edoardo Bontà (eddy)
4 | Erika Pigliapoco (md)
Enter user ID to disable: 0
Invalid user ID entered. Operation aborted.
Press Enter/Return to continue...
█
```

Figure 33. User lock aborted (invalid ID)

Test case #15: change product details

It's possible that the administrator enters wrong data or, due to marketing policy, a price review is needed; in this case, it's possible to change some of the product attributes, such as its name, the description or the price.

The system shows the list of available products and let the administrator choose one of them; after this phase, the user is asked which attribute need to update: the system asks for the new value and so on, until the administrator commits the updates.



```
unixo@unixo-vm: ~/ec++ — ec++ — 80x41
unixo@unix...c++ — ec++
Enter product ID to change: 26
REVIEW PRODUCT DETAIL
PRODUCT DETAIL
Category   : 28 | Wi-Fi Antennas
Name       : Extra range II
Description : Extra range wifi antenna (G)
Price      : 175.2
Availability: 10

CHOOSE WHICH ATTRIBUTE YOU WANT TO EDIT:
[1] Name
[2] Description
[3] Price
[0] End

Enter attribute ID: 1
Enter new value: Extra range III

CHOOSE WHICH ATTRIBUTE YOU WANT TO EDIT:
[1] Name
[2] Description
[3] Price
[0] End

Enter attribute ID: 2
Enter new value: Extra range wifi extension (G)

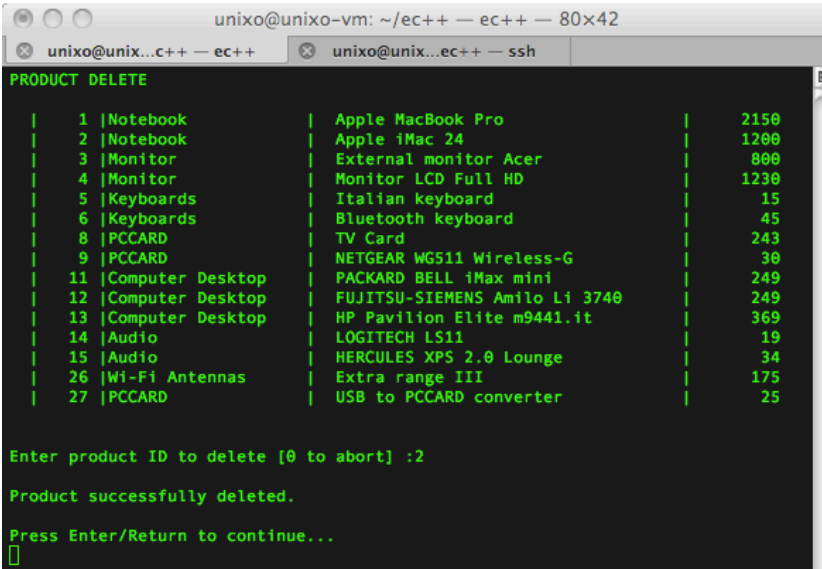
CHOOSE WHICH ATTRIBUTE YOU WANT TO EDIT:
[1] Name
[2] Description
[3] Price
[0] End

Enter attribute ID: 0
Operation successfully completed
Press Enter/Return to continue...
█
```

Figure 34. Update of two product detail

Test case #16: delete product

From administration menu, the user can choose to delete a product, as requirement. The system let the administrator choose a product to delete by displaying the whole catalog: the user enters the product ID and the system deletes the item.



```
unixo@unixo-vm: ~/ec++ -- ec++ -- 80x42
unixo@unix...c++ -- ec++
unixo@unix...ec++ -- ssh

PRODUCT DELETE

| 1 |Notebook      | Apple MacBook Pro      | 2150
| 2 |Notebook      | Apple iMac 24          | 1200
| 3 |Monitor       | External monitor Acer  | 800
| 4 |Monitor       | Monitor LCD Full HD   | 1230
| 5 |Keyboards     | Italian keyboard       | 15
| 6 |Keyboards     | Bluetooth keyboard     | 45
| 8 |PCCARD        | TV Card                | 243
| 9 |PCCARD        | NETGEAR WG511 Wireless-G | 30
| 11 |Computer Desktop | PACKARD BELL iMax mini | 249
| 12 |Computer Desktop | FUJITSU-SIEMENS Amilo Li 3740 | 249
| 13 |Computer Desktop | HP Pavilion Elite m9441.it | 369
| 14 |Audio         | LOGITECH LS11         | 19
| 15 |Audio         | HERCULES XPS 2.0 Lounge | 34
| 26 |Wi-Fi Antennas | Extra range III        | 175
| 27 |PCCARD        | USB to PCCARD converter | 25

Enter product ID to delete [0 to abort] :2

Product successfully deleted.

Press Enter/Return to continue...
█
```

Figure 35. Product deletion

Coding standards

Different coding standards have been applied in order to maximize portability and readability, allowing people to easily and quickly understand the code.

Even if some of these coding approaches aren't standardized by any Institute, they are recognized by the worldwide developer community; these standards range, for example, from the use of the *underscore* to mark private class variables and distinguish class variable from method parameters, to class members syntax as naming convention.

Sources files are formatted to be 80 characters wide; source code is split in chunks of instructions, divided by comments, formatted by DoxyGen syntax.

Develop environment

The project has been entirely developed under MacOS X Snow Leopard (10.6.4), using the following tools:

- Apple XCode 3.2.3;
- GNU g++ 4.2.1;
- GNU gdb 6.3.50;
- GNU make 3.81;
- MySQL server 5.1.48;
- Subversion 1.6.9;
- Doxygen 1.6.2.

Portability has also been tested under Ubuntu Linux 2.6.32, virtualized by Parallels Desktop: sources compile without any warning or error on both platforms.

COMPILE INSTRUCTIONS

First of all, the database must be created and the MySQL process need to be running and accepting incoming connection. To create the schema, import the file *seng_dump.sql* with your favourite tool; for example, it's possible to create the required structure with the command:

```
mysql -u username -p password < seng_dump.sql
```

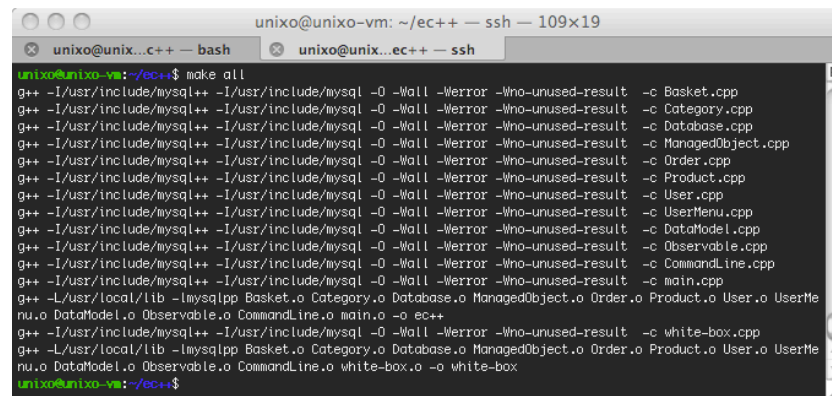
This command will create a schema called *seng* with some demonstration values.

To compile the project, at least the g++ compiler and the *make* command are needed. The programs need also the following dependencies:

- MySQL headers and dynamic libraries;
- MySQL++ connector headers and dynamic library (libmysqlpp)

Edit your *Makefile* and properly set the variables *CFLAGS* and *LIBS*: the former represents the parameters passed to the compiler and the latter the options for the linker; if your installation paths differ from those specified in the *Makefile*, change to reflect your configuration.

At shell prompt, issue the command *make* in order to build both the binary *ec++*, the program, and the binary *white-box*, the unit test.



```
unixo@unixo-vm: ~/ec++ - ssh - 109x19
unixo@unixo-vm: ~/ec++$ make all
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c Basket.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c Category.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c Database.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c ManagedObject.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c Order.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c Product.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c User.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c UserMenu.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c DataModel.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c Observable.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c CommandLine.cpp
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c main.cpp
g++ -L/usr/local/lib -lmysqlpp Basket.o Category.o Database.o ManagedObject.o Order.o Product.o User.o UserMe
nu.o DataModel.o Observable.o CommandLine.o main.o -o ec++
g++ -I/usr/include/mysql++ -I/usr/include/mysql -O -Wall -Werror -Wno-unused-result -c white-box.cpp
g++ -L/usr/local/lib -lmysqlpp Basket.o Category.o Database.o ManagedObject.o Order.o Product.o User.o UserMe
nu.o DataModel.o Observable.o CommandLine.o white-box.o -o white-box
unixo@unixo-vm: ~/ec++$
```

Figure 36. Example of sources compilation

Bibliography

Edoardo Bontà. *Software Engineer course*. University of Urbino, 2009.

Alessandro Aldini. *Databases and information systems*. University of Urbino, 2009.

Tangent Soft. *MySQL++ v3.1.0 User manual*. MySQL AB, Educational Technology Resources, June 2010.