Introduction

This is a tutorial on object oriented design methodology. It is based around to the solution for the assignment 3 for the CSE2305 course (Object Oriented Software Engineering and C++ Programming) at Monash University (spring semester 2006). This document is ***not*** a model solution and there is absolutely no claim about the amount of marks this solution would receive if submitted for the assignment. Moreover, this solution is not complete; it concentrates on some parts of the assignments, while leaving other important parts in the shadow.

This tutorial should be taken as a guide on how an object oriented system design can be attempted. The author has many years of professional experience designing B2C trading systems, database systems and pirate shops; he is also a tutor for the CSE2305 course. He therefore hopes that this tutorial will be a useful resource for CSE2305 and other Software Engineering students.

It should be noted, that hardly any Software Engineering problem will have a single correct solution. The author believes that the system presented here is one of many ways to go about the example problem presented in this tutorial and there are many other possible approaches. Some of them will be similar to the approach taken here; other will follow a completely different strategy. The only objective judgement one can take on this matter is that a solution must be clear and concise. It should be object oriented and modular and lead to a robust and easy to maintain system.

Contents

Introduction........1
Contents........1
The problem statement........2
Scope........4
High level design........4
Conventions........5
Business logic........6
Persistence........11
User interface........14
Dynamic model........14
Testing........14
Conclusion........15

The problem statement

Pirate Pete runs a shop that leases ships to pirates intent on causing mayhem and destruction. He has a ship for all occasions. Recently he has found it hard to keep track of which ships have been loaned and which are moored at his wharf. The problem is exacerbated by the dangers of the trade... his ships are often sunk whilst they are out on loan! Pete has decided to computerise his inventory, keeping track of which ships are "in" and which are "out", and also when the ships that are "out" are to be returned.
Pete has three types of ship:

Spanish galleons

Schooners

... and Canoes

(for the pirate down on his luck)

Spanish galleons, schooners and canoes may come mounted with guns. You can only fit a single gun to a canoe.

Every boat comes free with a talking parrot that incessantly reminds the skipper to return the ship in one piece. The ship can't leave the harbour unless it has a parrot on board. Sometimes parrots fly off and leave a ship during its voyage never to be seen or heard again.

For any item let by the Ship Shop we are interested in the following information:

- The type of item (galleon, schooner, canoe, sail, cannon, parrot)
- The number of items of this type currently stored at the wharf
- The number of items of this type currently out at sea
- The wholesale price (The cost for Pete to buy a new item of this type)
- Rental fraction (The percentage of the wholesale price it costs to rent the item per week)
- The rental price per week (This is equal to the wholesale price multiplied by the rental fraction (apart from parrots which are free). For example, if the wholesale cost is 2000 gold coins and the rental fraction 50%, the rental price is 1000 gold coins per week.)
- Deposit fraction (Before a ship is rented out to a pirate, a deposit must be paid up front. For example, this deposit might be 200% of the cost of the ship. That way if the ship gets lost at sea Pirate Pete isn't out of pocket. The deposit is returned after the ship is returned to Pete. Pete doesn't even care who borrows his ships. Actually, sometimes he prefers it if they are never seen again since he can then pocket the deposit!)

The system is to have the following features:

- The ability to add and delete items from the database of stocked items as Pete buys more goods or goods are lost at sea.
- The ability to increase the number of copies of an existing item (this is done when new stock arrives).
- The ability to read and save the database to Pirate Pete's computer hard disk.
- The ability to rent an item (hence decreasing the number of that item held at the wharf). It should not be possible to rent an item that is currently out at sea. Pete only rents out ships that he has at his wharf as many ships never return!
- The ability to list an inventory of all items at the wharf or all items currently owned (even if they are out at sea).

- The ability to display all items that are out of stock.
- The ability to see if an item of a particular type is at the wharf and available for rent.
- Searches that match should display all the information about that item type, including the number available and their deposit cost and cost to rent.

Any interaction with the system should be via a simple text interface. The system should automatically load the database on start-up, and save it before quitting. You should be able to specify the name of the database file as a command line argument.

Develop an object-oriented design for this system using UML diagrams.

Also, you will need to devise a test plan for the software you design. The test plan should test all the features of the system as specified.

What to submit for assessment:

- A design of the various classes required by the system, including appropriate class, object and interaction (sequence) diagrams (in PDF format).
- Written description of key classes and their functions.
- A plan for testing the completed system.
- Written components of the assignment should be done in plain text (i.e. readable using a standard text editor such as vi).

Scope

In this tutorial we will concentrate on designing a static program structure for the system. The author believes that for a course like CSE2305 (which covers advanced programming and an introduction to Software Engineering) this part will be the most important one to cope with: If you do not get your static structure right, you will not be able to design the dynamic structure of the software well. Remember, however, in order to do well on the assignment, you will need to put a lot of work into all requirements equally.

Our modelling tool will be UML and we will design some class diagrams and explain this process as we go.

High level design

Let us consider the overall task. Essentially, we are building a stock management system. This task has been often done and we want to draw on the experience of the professionals as much as we can. One established and very important pattern in software design is the separation of the business, the presentation, and the persistence logic. So, before we can start, it is essential to understand what this means.

N-tier applications

Almost any modern business application will consist of some sort of user interface (UI) which controls some functionality required by the business. That functionality works on data saved in some persistent storage. It turns out that these three layers (the UI, the business logic, and the persistence functionality) present three loosely coupled modules which can and should function separately.

Consider our ship shop. Imagine this was a real job for a real customer. We would design and code our application and Pirate Pete would be happily using it for a while. But at some point he decides that he is not happy to store the data in flat files any more. He would like to increase security and performance by using an off-the-shelf database (such as MySQL). Pete wants us to update the system. Now imagine, that the same class that calculates the rent for a ship, also stores the ship's data on the disk. We would have to read and understand that class (we have probably forgotten how it works since we developed it), and change it, which is very likely to introduce new bugs. It would be much better if one class (e.g. `Ship`) performed all the calculations, such as calculating the rent for a certain amount of weeks, while a different class (e.g. `ShipSaver`) took care of saving (and loading) the ship's data. These two classes could communicate through the `Ship`'s public interface. This way, when changing the way data is made persistent, we could rely on the fact that it does not affect the correct function of the rest of the system. In fact, we would not even need to understand the whole of the system. We could only concentrate on the saving and loading. Our system would be much more modular, manageable and extendable.

Now consider that Pete decided to replace the text UI with a graphical UI (GUI). Again, if the class `Ship` took care of displaying its own data on the screen, we would have to change it. We would add colours, input fields and all sorts of fancy features and not even notice how we break the core functionality while doing that. Now imagine that our `Ship` class merely has getter and setter functions for all its values. We have a second class, for instance `ShipTextUIPainter`, which contains a function such as

`displayShipDets(const & Ship ship)` that prints out the ship's data on the screen. Now all we have to do is to replace that class with a new `ShipGUIPainter` class, and out new GUI is ready.

Our 3-tier Ship Shop

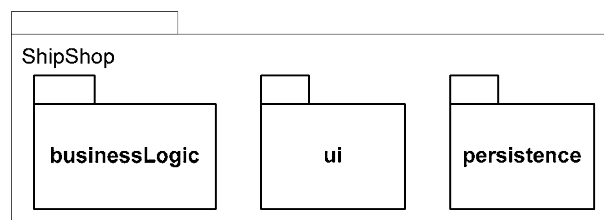
This gives us a very important pattern - our application must strictly separate the UI, the core business functionality and the persistence:

- The **UI module** reads input from the user (in our case – the keyboard) and displays the results on the screen. Under no circumstances is it allowed to perform core calculations, such as summing or multiplying properties of ships or other items. If such an action is required by the user, it must be requested through a function from the appropriate business class.

- The **business logic module** holds all required data in the RAM and performs all required operations on it. Under no circumstances will any instance variables be made `public` or any class outside this module `friend`. All data must only be accessed through corresponding `getX` and `setX` functions, however simple that data item might seem. Under no circumstances will classes of this module perform any input reading or parsing or any printing, however little a message might be (unless for debug or error diagnostic purposes). Under no circumstance will we do any saving or loading, or any writing or reading to or from streams in this module.

- The **persistence module**. Here we only deal with files. To load a database, we open and parse a stream, while creating and initialising the objects of the business module through a well defined public interface. To save a database, we examine the business object using their public getter methods.

To emphasise of this structure we will have three different packages:



The most interesting bit is, of course, the business logic, so we will concentrate on that. Later, we describe the persistence package in more detail.

Conventions

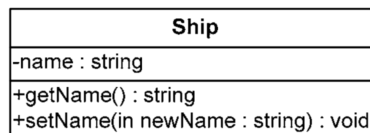
We will be using the UML Class Diagram notation to denote the static structure of the system. First of all let us clarify our notation.

No implementation details. We will not display any private or local information which may be required for a specific implementation of the system. We only concentrate on the conceptual information.

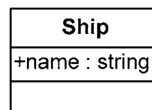
Public variables are not public variables. Public variable are always a bad idea. Always! You might think for instance – changing a ship's name does not imply any action, so why can't I make the name of a

ship a public string variable? Well, in theory you could. But imagine that in future you discover that each ship's parrot's name is made up of the name of the ship, and changing the ship's name requires changing the parrot's name too (sounds weird, but pirates are sometimes weird). You would have to search through all of you code and everywhere you change a ship's name you would have to modify the code to change the parrot's name as well. That is tedious and error-prone. But if the ship's name was private and you always used a setter-function to modify it (even from within the `Ship`-class!) you would have to modify the code only at a single place – inside the setter-function!
So, since we know that all variables are private, we can save some space. In our diagrams we will mark a number of variables as public. This denotes that there is a private variable which has both, a getter and a setter function.

For instance, in order to denote that:

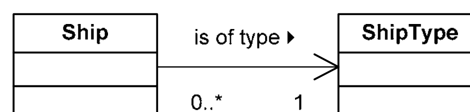


we simply use this:



This way we save a lot of space and concentrate on the essentials. Note that if variable has ether a setter or a getter-function, but not both of them, we have to use the long notation.

No unnecessary information. We do not want to use up space by saying things twice. From a diagram like



we should be able to deduce that the class `Ship` has functions like `setType` and `getType` or similar without having to explicitly denote that.

Business logic

Now we have outlined our strategy and our notational conventions and we can start the work.

Let us consider the items available in the shop first. From the requirements we notice that we have to deal with properties such as a wholesale price of an item, as well as an item's current location. Let us consider (say) cannons. It is not clear from the requirements whether each cannon has a unique wholesale price or whether all cannons are the same. In real life it is very often the case that the customer does not specify the requirements in sufficient detail. Normally we would have to contact the customer and to clarify this before we go on. But Pete is not available and we have to take our best guess. Back in the old day, each item was hand-made and probably had its own unique price. But in the modern industrial age items of the same type usually have the same price. The lessons drawn from this tutorial will most likely be applied in modern days (unless you invent a time-machine), so we will assume the latter case.

It is a very bad practice to store properties of a whole type of items together with the properties of each individual item, because it would be very difficult to change the type-properties in that case. Therefore we need two separate concepts: an `Item` and an `ItemType`.

An `ItemType` will store properties shared by all items of a certain type: like the name of the type ("sail", "cannon") and its pricing information.

In cases when a system is required to work in a multi-currency environment, it is a custom to introduce a `Currency` or `Price` (or similar) class, which encapsulates all relevant information and currency specific calculations. However, in simple systems working in a single currency environment the common practice is to use an unsigned integer to specify the smallest currency denomination. In Australia, for example, we would store Cents. For simplicity, we will assume that Pete only trades in gold coins. 1 gold coin is divided into 100 silver coins. For example, if a cannon costs 499.95 gold coins, we would say price=49995 (silver coins).

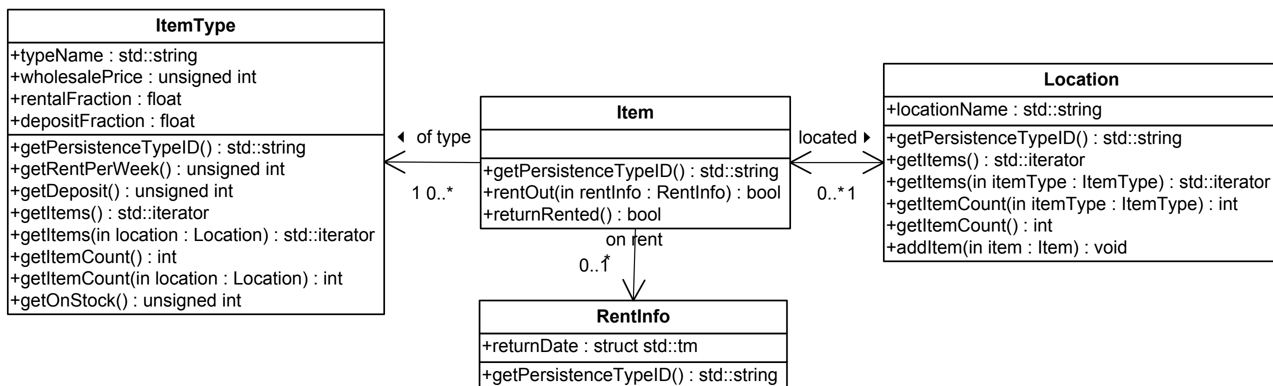
An `ItemType` also needs functions to calculate prices and costs based on the primary properties (e.g. rent per week = wholesale price * rental fraction). Furthermore, it needs functions to obtain all individual items of a type, access the number of such items available for rent or at certain locations, and similar.

An `Item` will store information relevant for individual items, such as current location (at sea, in the wharf, …) and rent information.

Pete is not interested in who is renting his items. In fact, he thinks that it is much healthier not to know too much. But he is interested in the date when an item should be returned. We encapsulate this information in a `RentInfo` class, because this makes it easier to add more information in future. Also, an item will only have an "`on rent`"-association when it is actually on rent. When an item is in the wharf, its "`on rent`"-association will be `NULL`.

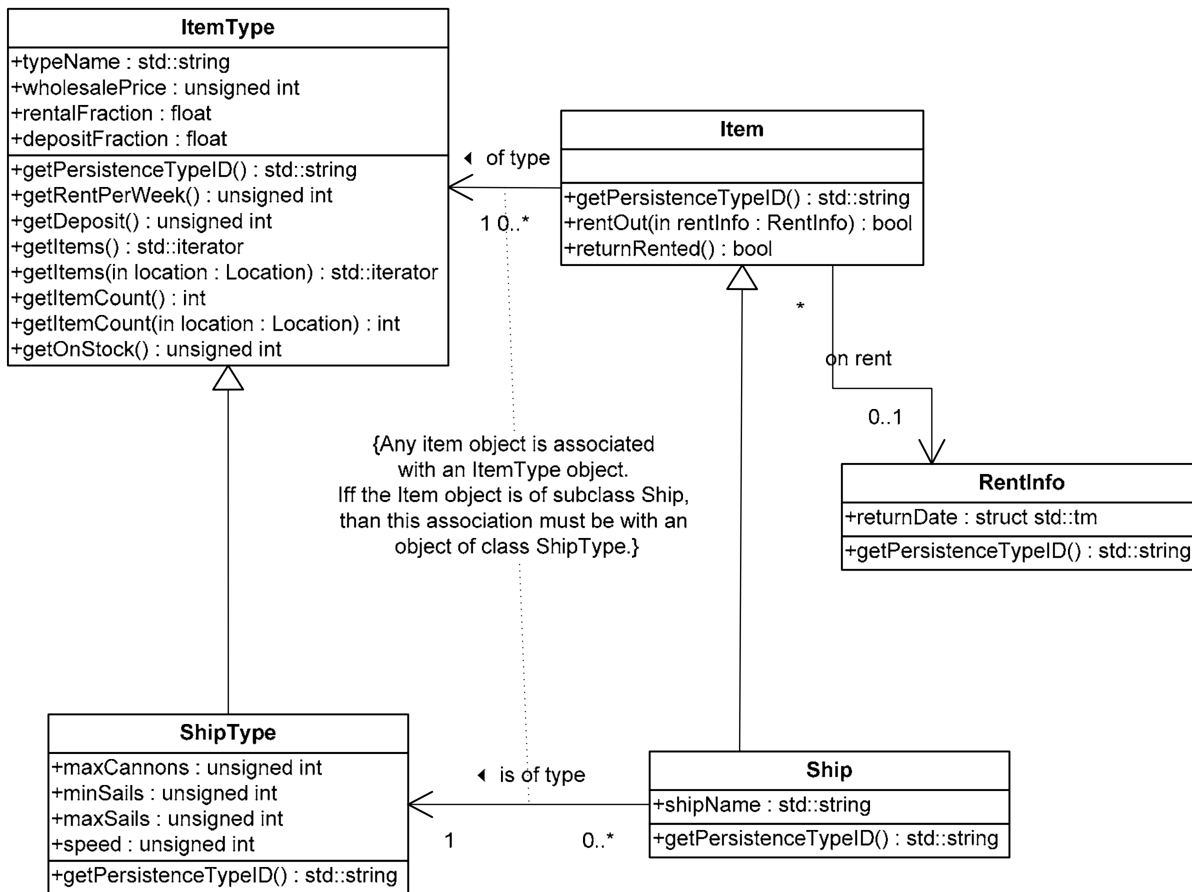
Finally, objects of the `Location` class represent various possible locations and provide functions for accessing the items at the locations.

This yields the following basis for our business logic:

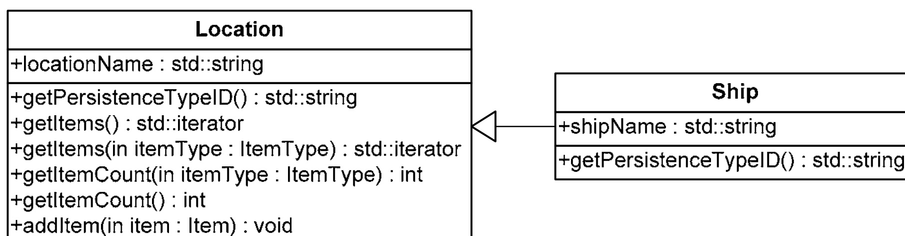


You probably noticed that we did not include some important properties of the ships. For instance, how many cannons can a certain ship take on board?

Ships are a special kind of items and this relationship is best represented by a specialisation/generalisation relationship. Similar, the `ShipType` is a special kind of an `ItemType`. Since `Ship` is a subclass of `Item` and because an `Item` object is always associated with an `ItemType` object, a `Ship` object must also always be associated with an `ItemType` object. We introduce the additional constraint that the `ItemType` object associated to a `Ship` must be of subclass `ShipType`:



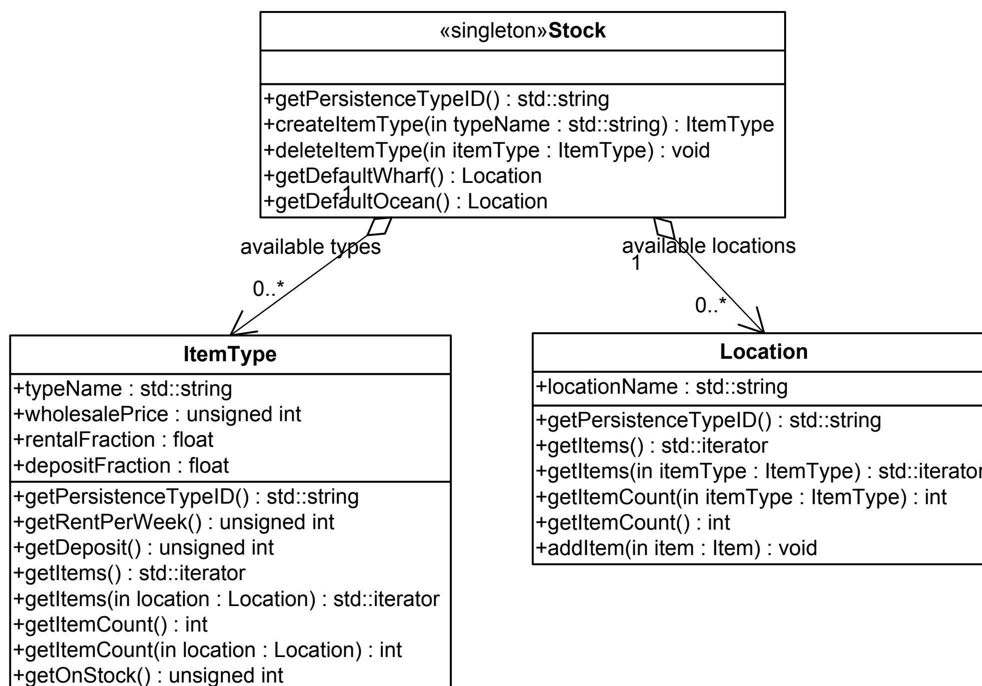
Consider now the case when a ship gets sunk and is never returned. When the return date has passed Pete writes off the ship. But what about the cannons and sails (and the parrot) which went with it? They also have to be written off. To make this possible we introduce the following trick: the location of the equipment rented out will not be "sea", but the actual ship they are on. The fact that they are at sea can still be determined recursively through the ship. This means, that a `Ship` is a special kind of `Location`:



This makes our program very flexible:

- We have a special location "wharf" (and we may easily add more wharfs as Pete's business grows).
- We have a further special location "sea" (and we may add more seas or oceans as Pete's business becomes global).
- Each ship is a location.
- Each item is directly associated with exactly one location. But it may be indirectly associated with more locations through recursion (e.g. a cannon is located on "The Jolly Roger" which is in turn located on the Treasure Ocean).

Finally, we need a manager object which holds a collection of all our information. For that purpose we define a `Stock` class. This will follow a singleton pattern (urgently find out how this very important technique works! (Google will help)). The `Stock` will hold a collection of all `ItemType`-s and all `Location`-s. Through those we will be able to access all our `Item`-s.

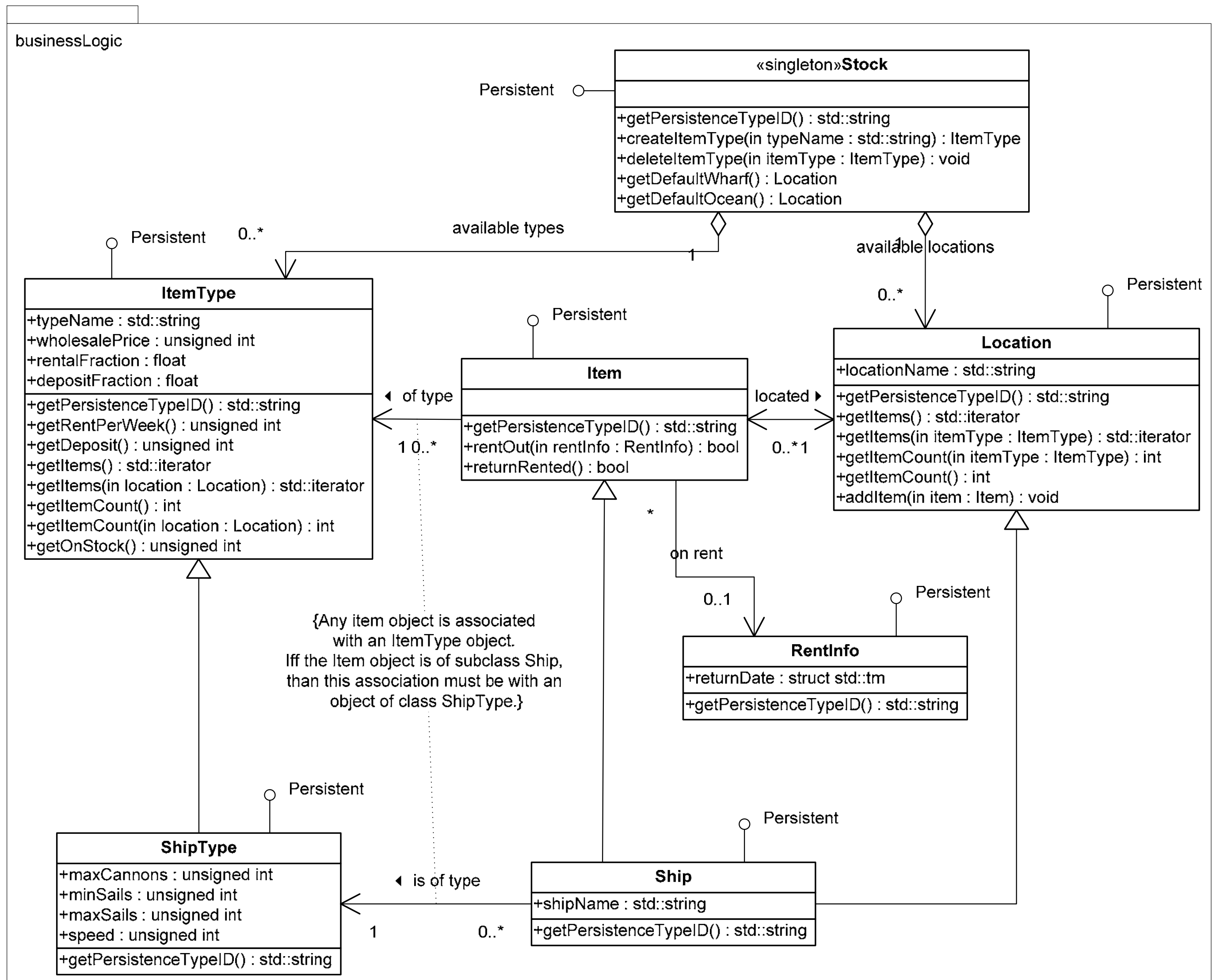


When you first read the assignment statement you were probably thinking you would have classes like `Canoe`, `Schooner` or `Parrot`. You might be wondering why we didn't include those. The answer is that we model our system on a different level of abstraction. This is the main art of Software Engineering: finding the right abstractions. For Pete things like these are important. Cannons, parrots, galleons and sails are the terms he thinks in. But we must have the ability to abstract away from those things to what is really important for our system. Would our `Sail` class have any function different from a `Cannon` class? And does a `Schooner` really differ from a `Galleon` for our purposes? If not, than what would be a justification for creating such classes?

As it was pointed out in the introduction, there are many possible solutions to this problem. Some of them may include classes to describe each type of item and there is certainly nothing wrong with that in principle. Whichever strategy you choose, the important thing is that you think carefully about what you are doing and do not do things just because they seem intuitive at the start. Finding the right

abstractions is an art which requires a lot of experience and you should be able to justify whichever choice you take.

Let us now combine our business logic into one overview-diagram:



If you were reading the diagrams carefully, you might have noticed one function which we have not yet explained: `getPersistenceTypeID() : string`. We will discuss persistence in the next section.

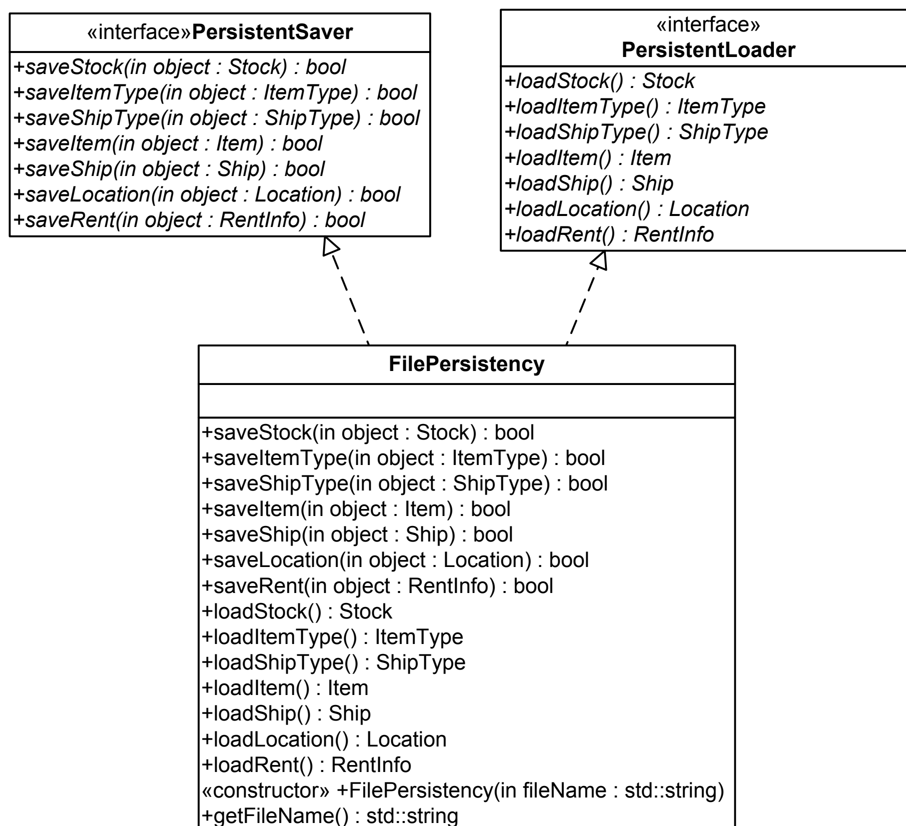
Persistence

We have previously discussed the need to separate the business logic from the persistence logic of the application (see section "High level design"). Having designed the business logic in the previous section, we will now design our persistence logic.

Let us define a `PersistentLoader` interface. A class implementing that interface must provide functions to load the data for an object of any of our business classes from some permanent location. Such a location could be a file, a database, a network resource or whatever. For each permanent storage type we can write a new class implementing the `PersistentLoader` interface. This will be the only change required to the whole system to integrate a new type of storage backend. Neat, isn't it?

To make our system not read-only we require a `PersistentSaver` interface which works in just the same way.

For the first version, we only require flat file persistence functionality. To achieve this we create a class `FilePersistency` that implements both of our interfaces:



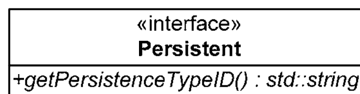
How does the `FilePersistency` class work? Say, want to save our database. We create an instance of `FilePersistency` and initialise it with a file name. We then pass the singleton instance of `Stock` to the `saveStock` function. This function saves the `Stock`-specific data to the file and then iterates through the encapsulated `ItemType` and `Location` objects. Each of those is passed to `saveItemType` or `saveLocation` functions respectively. Those functions save the data specific to "their" classes in the file and recursively call the appropriate `saveX` functions for all instance-objects until the whole database, including all items is saved.

So, why do we need `getPersistenceTypeID`??

Because of polymorphism, when the function `saveStock` iterates through the ItemType-s, it doe not know which `ItemType`-s are "normal" and which are actually `ShipType`-s. If we were using a language like Java, where run-time introspection is available, we could simply find that out using the `instanceof` operator. But what possibility do we have in C++?

If we made our `save` function a virtual member of `ItemType` and `ShipType`, the compiler would do the job for us and call the right function. But this would badly violate the N-tier concept. In future the `ItemType` class would grow to include functions like `saveInDatabase`, `saveInSomeOtherDatabase` and so on, and our program would become unmanageable.

A commonly used solution is to force each object which wants to be persistent to return an ID, which can be used to determine which saver-function is right for that object. Classes that want their objects to be persistent implement the `Persistent` interface:

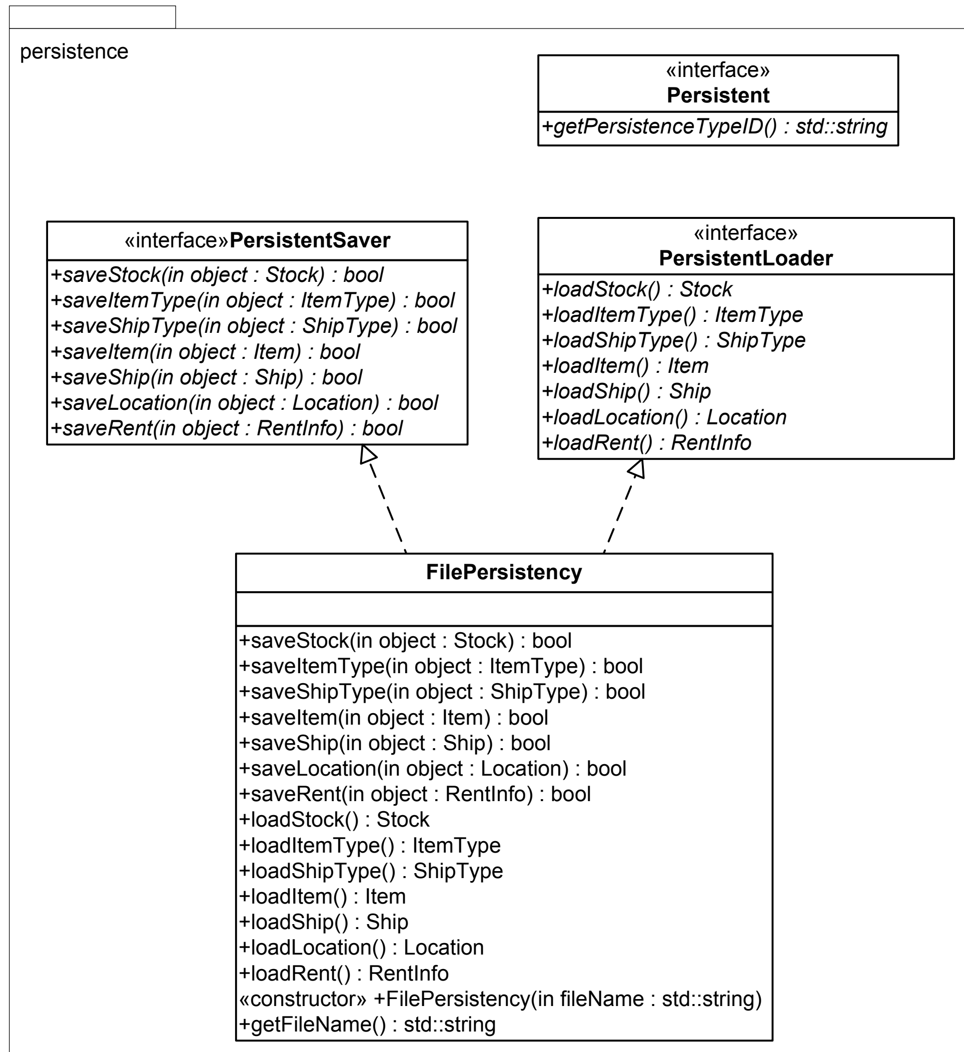


We have chosen `string` to be the type of ID, but this choice is somewhat arbitrary. For instance, integer constants are also widely used for this purpose. Examine the overview diagram of the business logic package again. You will now notice that most classes are specified to implement the `Persistent` interface.

We can now implement the top-level save function as follows:

```
FilePersistency::saveStock(const Stock & stock) {

      obtain file stream;

      save Stock-specific variables to the file (use the getter functions to access them);

      obtain the ItemType iterator;
      for each ItemType object itemType {
            switch(itemType.getPersistenceTypeID()) {
                  case PERSISTENCEID_ITEMTYPE:
                        saveItemType(itemType);
                        break;
                  case PERSISTENCEID_SHIPTYPE:
                        saveShipType(itemType);
                        break;
            }
      }

      obtain the Location iterator;
      for each Location object location {
            switch(location.getPersistenceTypeID()) {
                  case PERSISTENCEID_LOCATION:
                        saveLocation(location);
                        break;
                  case PERSISTENCEID_SHIP:
                        // Do not need to save ships here as they will be saved by their locations
                        break;
            }
      }

      Close file stream;
}
```

Here is the overview of out persistence package:



User interface

We will leave the detailed design of the user interface package to you as an exercise. In order to find out how to stick to our N-tier design, you should find out about the model-view-controller pattern. That design pattern is a standard technique for designing flexible and efficient user interfaces. A Google search is a good place to start.

For a text-only UI a single `UI`-class combined with a couple `view` (or `painter`) classes should do the job. Make sure that your `UI` class contains a separate function for each menu function of your software. This allows for non-hierarchical menu navigation and avoids long function bodies with confusing indentations.

Dynamic model

The problem statement required you to design a dynamic model for each of the use cases. Luckily the use cases are explicitly listed in the problem statement:

- Add and delete items from the database
- The ability to increase the number of copies of existing items.
- The ability to read and save the database to hard disk.
- The ability to rent an item (it should not be possible to rent an item that is currently out at sea).
- The ability to list an inventory of all items at the wharf or all items currently owned.
- The ability to display all items that are out of stock.
- The ability to see if an item of a particular type is at the wharf and available for rent.
- Display all the information about an item type.

Each of these use cases should be modelled with a UML Communication (Collaboration) Diagram, a Sequence Diagram or any other UML diagram if you can justify why you deem it appropriate.

We will leave this as a further exercise ☺. You should start with the saving, as the corresponding pseudo-code is already partly given in the section about persistence.

Testing

An often underestimated issue in software development is Testing. Research shows that this is one of the main reasons behind a large number of failed software projects. You should find out about Unit Testing and how you could use it in you object oriented design.

The least you need to do to satisfy the expectations expressed in the problem statement is to provide a detailed test log.

After you have modelled each of the use cases in detail you need to design a test plan for each of them. In order to do that for each use case, you need to identify all types of valid and invalid inputs into the system (including user mistakes) and identify the boundary conditions between them. For each program screen (menu/function/choice/…) you need to list all input types (choices/possibilities/…) including examples. For each of the examples you need to list what output you expect from the system. Do not forget to test for negative and wrong inputs – these must be handled gracefully.

While it might sound tedious at first, this is a lot less tedious then debugging a not properly tested system. So do not take any shortcuts here!

Conclusion

We have examined the design of the static software structure for a stock management system for Pirate Pete's Ship Shop. We have introduced the N-tier design methodology and demonstrated how it can be realised in a business application. We designed a business logic component for the system and explained the design choices step by step. We then demonstrated how a persistence module for the business logic can be realised. Finally, we discussed some issues of dynamic system design and system testing.

You are welcome to email the author if you have any questions.