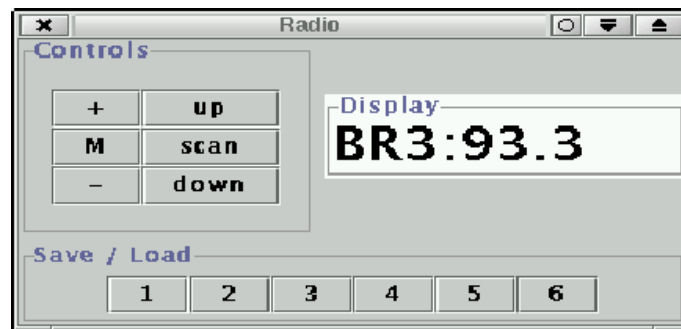


Assignment 2 Part 1 / Object Oriented Systems Design – Design Patterns and Evolution of Software-Architectures

Kenan Esau

November 2000



Tutor: Mr. Karl-Heinz Sylla
Course: M.Sc Distributed Systems Engineering
Lecturer: Mr. Prowse

Contents

1	Introduction	3
2	Evolution of a Storage System	3
2.1	Simple Storage System	3
2.2	Persistent Storage System	4
2.3	Storage System with GPS-Information	5
2.3.1	The new Design with the Bridge-Pattern	5
2.3.2	Implementation of the new Design	7
2.4	Choosing the Implementation at Start-up	8
2.5	Usage of JUnit	8
3	Conclusions	9

1 Introduction

This assignment is the third and last part of the assignments dealing with Java and the radio-simulator. This part deals with the evolution of software-architectures and the usage of the testing-framework JUnit. Part 1 of the assignments [4] already gave an overview about the basic structure of the radio simulator and the used design patterns. This part only deals with the storage system of the radio simulator and its different implementations at the different software evolution steps.

2 Evolution of a Storage System

In the following four sections the incremental development from a very simple storage system to a very flexible approach is described. Each section describes a new major step which introduces new functionality of some kind to the storage system.

2.1 Simple Storage System

Before the first implementation of a storage system the radio simulator was only able to store frequencies of a station in form of integer values in an array. The first implementation of the storage system will be able to store station information consisting of the station name and its frequency. The class `StationEntry` was introduced for this purpose. This class is a simple class which is only used for data storage – it has no own “intelligence”. The class `StationStorageSys` is introduced to store references of objects of the class `StationEntry`. There for this class uses a `Vector` and a `HashMap`. The `Vector` is needed to assure the order of the entries and the `HashMap` is needed for fast access to them.

Each time a new frequency is set the radio has to “ask” the `StationStorageSys` which stores all available stations whether there is a station on that specific frequency-value or not. For this special purpose the `HashMap` uses the frequencies as a key and the station names as a value.

All in all there are two storage systems for stations in the radio-simulator. One of them – which was already described above – stores all stations which are available to the simulator and the other one stores only six entries which can be chosen by the user. This is for the “save/load”-station feature and since there are only six buttons to save and load a station there have to be only six entries in this storage system.

2.2 Persistent Storage System

As a second major step the storage system for the `StationEntries` is made persistent. To serialize an object of a class it has to implement the interface `Serializable` or the interface `Externalizable`. The persistent storage system is implemented in the class named `PersistentStationStorageSys`. This class implements the interface `Externalizable`. It does not matter whether you implement `Serializable` or `Externalizable`.

In figure 1 you can see the relations between the new class `PersistentStationStorageSys` and the old class `StationStorageSys`.

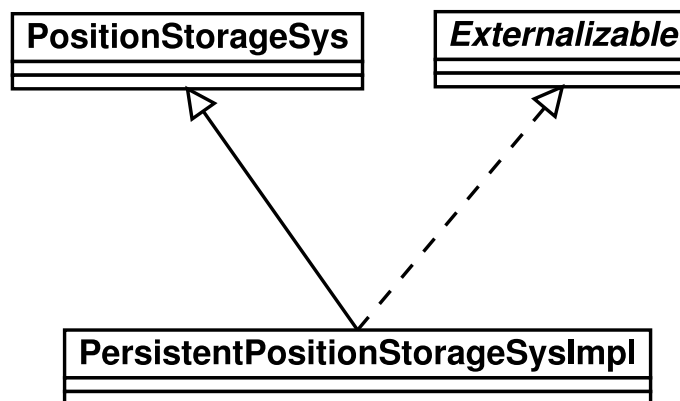


Figure 1: *Static relations of the class `PersistentStationStorageSys`*

A class which implements the interface `Externalizable` has to implement the methods `writeExternal()` and `readExternal()`. In the little piece of code below you can see the two methods as they are implemented in the class `PersistentStationStorageSys`. All references to the saved `StationEntry`-objects are saved in the `Vector` named `stations`. The `HashMap` named `hash` contains the corresponding key value pairs. By saving these two variables to the file the complete state of an object of the type `PersistentStationStorageSys` can be saved.

```
public void writeExternal(ObjectOutput out) throws \
    IOException{
    out.writeObject(stations);
    out.writeObject(hash);
}

public void readExternal(ObjectInput in) throws IOException, \
    ClassNotFoundException {
    stations = (Vector)in.readObject();
    hash = (HashMap)in.readObject();
}
```

2.3 Storage System with GPS-Information

Besides this the class `PersistentStationStorageSys` derives from the class `StationStorageSys`. Every change in the saved stations is immediately written to the disk. To achieve this `PersistentStorageSys` overwrites every method of `StationStorageSys` which is able to change the data. As you can see below the overwritten versions of the methods call the implementations in their super class and after this the method `saveToDisk` is called where all changes are written to the disk. This method opens an `ObjectOutputStream` writes the actual object to the disk and closes the stream again.

The effort to create this class was minimal. The methods which had to be implemented additionally are described above and the methods which change the saved content just call the implementation of the father-class and the method `saveToDisk()`. For example the method `remove()`:

```
public StationEntry remove(int pos){
    StationEntry entry = super.remove(pos);
    saveToDisk();
    return entry;
}
```

Most effort in creating this new class was the new constructor which has to check whether it has to load stored data from a file or not. The signature of this constructor looks like this:

```
public PersistentStationStorageSysImpl(boolean load, String \
    fileName)
```

The first parameter says if the constructor should try to load data from the file specified in the variable named `fileName`. If not it assumes it has to create this file.

2.3 Storage System with GPS-Information

Section 2.3.1 describes the new design using the “bridge”-pattern and the section 2.3.2 describes the changes to the code which were needed.

2.3.1 The new Design with the Bridge-Pattern

Since now it should be able to store different kinds of entries to a storage system I decided to restructure the class hierarchy a little bit. As you can see below in figure 2 it was decided to build a “bridge”. The bridge-pattern is introduced in [2]. It is used to split an abstraction from its implementation. In this case there are two different implementations. One implementation which stores the data only in memory and where changes are lost after a restart of the program and the second one is a persistent implementation where all data is saved in a file.

2.3 Storage System with GPS-Information

These two implementations can be used by all classes which are of the type `StorageSys`. At the moment there are two sub classes of this class which can make use of the two different implementations:

1. `StationStorageSys`
2. `PosistionStorageSys`

The “old” configuration of a persistent storage system which stores `Station-Entries` can now be achieved by combining the classes `StationStorageSys` and `PersistentStorageSysImpl`.

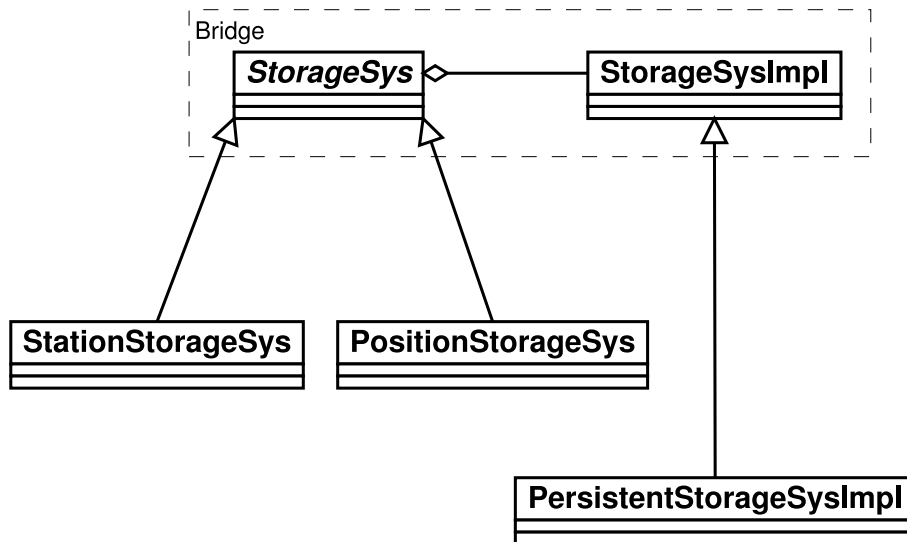


Figure 2: *Bridge-Pattern*

The new implementation of `StationStorageSys` and the new class `PositionStorageSys` are only responsible for the type safety. Their major task is the correct delegation of method calls to the appropriate implementation-class (eg. `StorageSysImpl` or `PersistentStorageSysImpl`).

The new class `PositionStorageSys` provides an interface which is capable to store `PositionEntry`-objects to the implementations. It works like the class `StationStorageSys`. It assures type-safety and delegates the method-call to the appropriate methods in the used implementation-class.

The decision which implementation should be used is made via the constructor of one of the classes derived from `StorageSys`. The next line of code shows an example how a certain storage system can be utilized to be persistent or not. In this example a storage system which stores `PositionEntries` uses a persistent implementation.

```
PositionStorageSys = new PositionStorageSys(new \
    PersistentStorageSysImpl(true, "positions.dat"));
```

2.3 Storage System with GPS-Information

2.3.2 Implementation of the new Design

To create the new class `StorageSysImpl` it was possible to use a lot of code from the old class `StationStorageSys`. The methods to modify the stored data look the same as in the old implementation of `StationStorageSys` except of the fact that the class `StorageSysImpl` does not store `StationEntries` any more but `AbstractEntries`. `AbstractEntry` is an interface which has to be implemented by all classes which have to be stored within a class of the type `StorageSysImpl`.

The additional changes which have to be done for the existing class `StationEntry` are minimal. This class only has to implement the two methods `getKey()` and `getValue()` of the interface `AbstractEntry`.

For the `PositionStorageSys`-class it is the same as with the `StationStorageSys` – here it was possible to use a lot of code from the class `StationStorageSys` too since its interface is very similar to the interface of `PositionStorageSys`.

The persistent implementation-class `PersistentStorageSysImpl` is derived from `StorageSysImpl`. Again – it was possible to use most of the code from the class `PersistentStationStorageSys`. The main work was to replace the special types of the method-signatures by the more generic ones (eg. replace `StationEntry` with `AbstractEntry`). Below you can see an example how easy it was to turn the class `PersistentStationStorageSys` into a more generic class which is able to handle very different types of entries.

The first piece of code shows the method `remove()` of the class `PersistentStationStorageSys`. This class was used as a base for the new class `PersistentStationImpl`. The second part shows the new implementation of the method `removeObj()` in the class `PersistentStationImpl`.

```
public StationEntry remove(int pos){
    StationEntry entry = super.remove(pos);
    saveToDisk();
    return entry;
}
```

As you can see the changes are very small. The name of the method was changed in `removeObj()` to indicate that it's now possible to remove more generic instances from the storage system. In fact these entries are not of the type `Object` but of the type `AbstractEntry` nevertheless I think the name is quite good. The return-type of the method was changed from the special type `StationEntry` to the more generic type `AbstractEntry`.

```
public AbstractEntry removeObj(int pos){
    AbstractEntry entry = super.removeObj(pos);
```

2.4 Choosing the Implementation at Start-up

```
        saveToDisk();
        return entry;
    }
```

Similar changes to the one described above had to be applied to all methods. But the changes were just a matter of renaming – which could be done very easily with a good editor (eg. Emacs, Nedit, VI, ...).

2.4 Choosing the Implementation at Start-up

The last part is the most simple one. The essential part is just the `if`-construct you can see below. The boolean variable named `isPersistent` is set to `true` if the user wants to have persistent entries and it is set to `false` if the user does not want to have persistent entries. The decision for the storage system for the `PositionEntries` whether it should be persistent or not can be made in an equivalent way as shown below.

```
    if ( isPersistent ){
        System.out.println("creating_persistent_store(s)");
        savedStations = new StationStorageSys(new \
            PersistentStorageSysImpl(true, "stations.dat"));
    }
    else {
        System.out.println("creating_non-persistent_store(s)");
        savedStations = new StationStorageSys(new StorageSysImpl(\
            entries));
    }
}
```

2.5 Usage of JUnit

Although I separated this section about testing from the sections about the implementation and the development of the storage system the development of the test cases was done in parallel. This fact is crucial to effective and useful testing. If the test cases would have been developed after the storage system the benefits of them would have been much less.

Although it was only mandatory to provide test-cases for the different kinds of storage systems I already started to write some test cases for the threads assignment. For the threads assignment there is one test case, which is not complete, for the station-search functionality and one test case for the queue which acts like a buffer (see [5]). These test cases are not complete since they have been the first tries – just to play around a little bit with JUnit – but in case of the queue they helped me a lot to find some problems.

The remaining test of the suite `RadioSuite` deal with the different kinds of storage systems. Perhaps the name “StorageSystemSuite” would be

3 Conclusions

more appropriate. There are the following tests cases which deal with the storage systems:

1. **StorageSysImplTest**
This test deals with the implementation which stores its entries just in memory – the class `StationStorageSysImpl`.
2. **PersistentStorageSysImplTest**
This test case checks the class `PersistentStationStorageSysImpl` which is derived from `StationStorageSysImpl`. The test case was just created by cutting and pasting the test case from point 1. Just the names of the classes had to be replaced.
3. **StationEntryTest**
With this test case the class `StationEntry` is checked.
4. **StationStorageSysTest**
This test class contains two test cases. One test case for the configuration with the non-persistent `StorageSysImpl` and one for the persistent configuration.
5. **PositionEntryTest**
With this test case the class `PositionEntry` is checked.
6. **PositionStorageSysTest**
Like the test class `StationStorageSysTest` this test class contains two test cases. One for a persistent and one for a non-persistent configuration.

3 Conclusions

As you can see in this assignment incremental software development/evolution is supported by design patterns very fine. A good design simplifies later software changes and it enables new developers to understand the design of a complete project more easily. The usage of design patterns not necessarily means that it is a good design but it makes it a lot easier to make a good design even for less experienced developers.

The usage of JUnit simplifies the development very much since a lot of errors can be discovered very early in the development process. I think especially in bigger projects this methods can be applied really effectively but even in this little example the test cases were a great help to the developer to check if the classes work properly.

References

- [1] “Description of the task for the assignment for workshop AW2”
- [2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides “Design Patterns”
- [3] Goll, Weiß, Rothländer “Java als erste Programmiersprache”
- [4] Esau “Assignment Object-Oriented Systems Design – A Radio Simulator”
- [5] Esau “Assignment 2 Part 2 Object-Oriented Systems Design – Threads”