Ministerul Educației și Tineretului al Republicii Moldova  
Universitatea Tehnică a Moldovei  
Catedra Tehnologii Informaționale

Laboratory #3  
PSI

Executed by: Tatiana Moldovan, st.gr.FAF-091

Verified by: Ivan Zarea

Chișinău – 2012 –

## 1. Code inspection

For this laboratory work we have used Sonar. It was a pain in the ass installing it – but in the end it worked (and everybody was happy :D)

**Sonar Architecture**

|  |  |  |
| --- | --- | --- |
| **Concept** | **Definition** | **Related Topics** |
| **Analyzer** | Batch that analyzes the source code to compute **snapshots**. | See [Architecture Overview](http://docs.codehaus.org/display/SONAR/Architecture+Overview). |
| **Client** | Batch that runs the **analyzers.** | See [Architecture Overview](http://docs.codehaus.org/display/SONAR/Architecture+Overview). |
| **Database** | Stores:   * configuration of Sonar * **snapshots** | See [Architecture Overview](http://docs.codehaus.org/display/SONAR/Architecture+Overview). |
| **Server** | Web Server that is used to navigate the **snapshots** and make configuration | See [Architecture Overview](http://docs.codehaus.org/display/SONAR/Architecture+Overview) . |

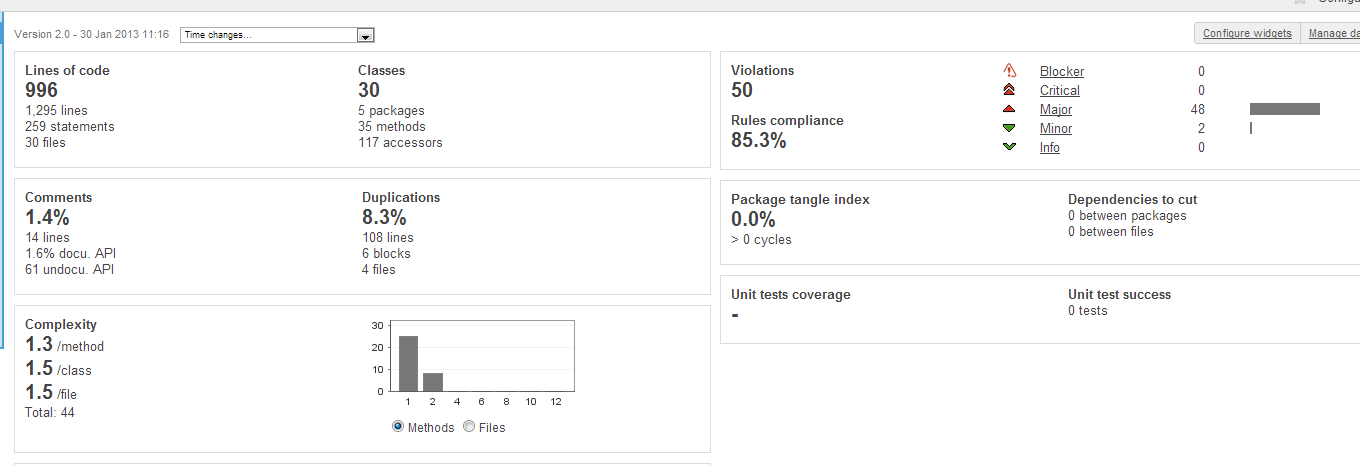
**Quality**

|  |  |  |
| --- | --- | --- |
| **Concept** | **Definition** | **Related Topics** |
| **Check** | Check = **Coding Rule**. |  |
| **Coding Rule** | A good coding practice. Violating coding rules leads to quality flows.  Coding rules can check quality on files, unit tests or packages. | See [Viewing Unit Tests > Violations Tab](http://docs.codehaus.org/display/SONAR/Resource+Viewer#ResourceViewer-viewingUnitTests) |
| **Metric** | A property of a **resource**. Examples: number of lines of code, complexity, etc.  A metric may be either:   * *qualitative:* gives a quality indication on the **resource** (ex: density of duplicated lines, line coverage by unit tests, etc.) * or*quantitative:* does not give a quality indication on the **resource** (ex: number of lines of code, complexity, etc.)   The value of the metric for a given **resource** is called **measure**. | See [detailed documentation on metrics](http://docs.codehaus.org/display/SONAR/Metric+Definitions). |
| **Measure** | The value of a **metric** for a given **resource** at a given time.  Examples:   * 125 lines of code on class myClass * Density of duplicated lines of 30.5% on project myProject |  |
| **Quality Profile** | A set of **coding rules**.  Each **snapshot** is based on a single quality profile. | See [Quality Profile](http://docs.codehaus.org/display/SONAR/Quality+Profiles). |
| **Resource** | A piece of software. Examples: project, package, module, file, etc. |  |
| **Snapshot** | A set of **measures** and **violations** on a given **resource** at a given time.  A snapshot is generated for each Sonar analysis. |  |
| **Violation** | When a **resource** does not comply with a **coding rule**, Sonar logs a violation into the **snapshot**.  A violation can be logged on a file or on a unit test or on a package. |  |

**Web Interface**

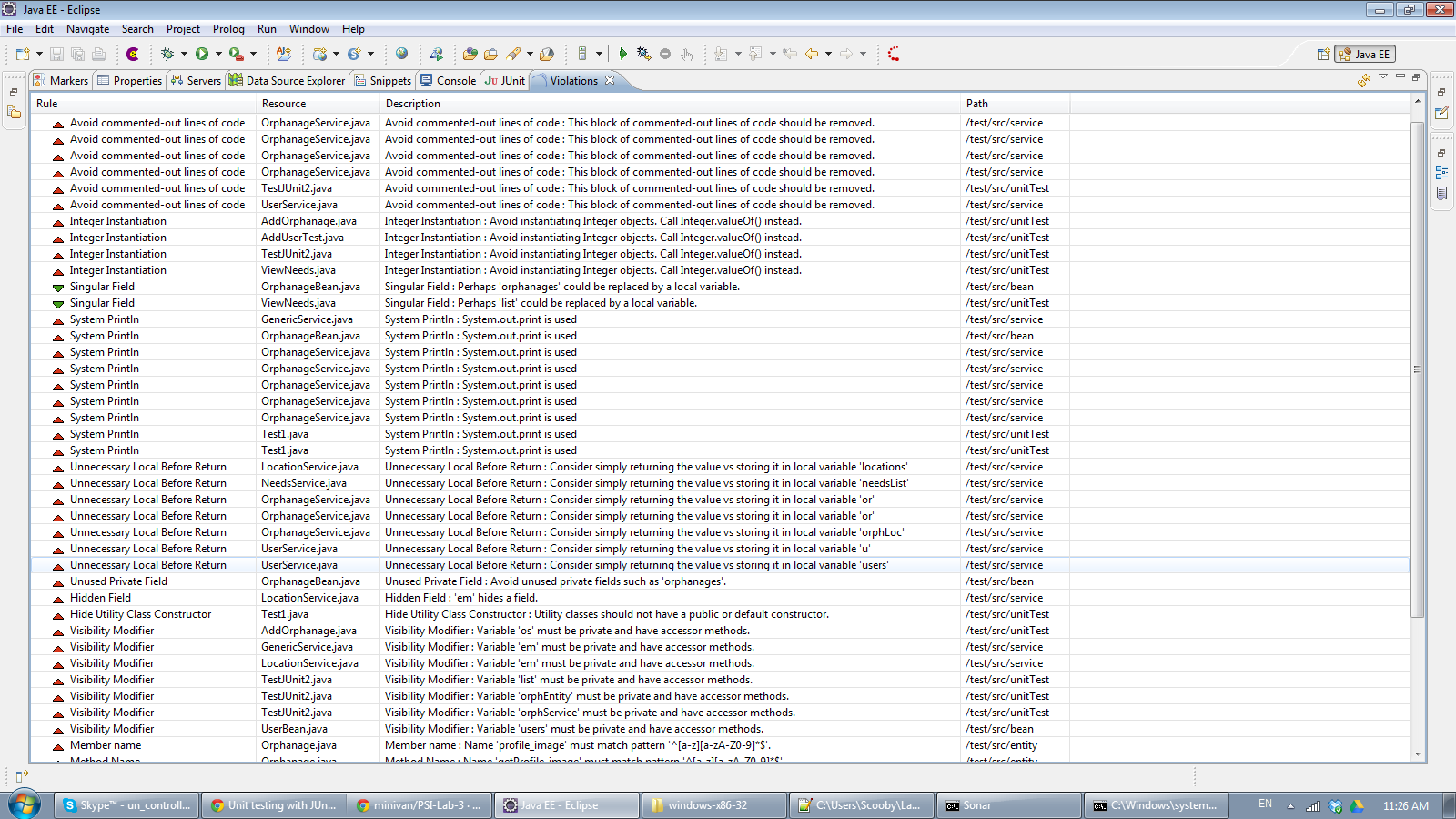
|  |  |  |
| --- | --- | --- |
| **Concept** | **Definition** | **Related Topics** |
| **Dashboard** | Web page that provides a way to display any data stored in the **database**.  A dashboard is a set of **widgets**. | See [Dashboards](http://docs.codehaus.org/display/SONAR/Dashboards). |
| **Widget** | It is a box that displays data on a **dashboard**.  There are two types of widget:   * *Global widget* that can display data from multiple projects * *Project widget* that only displays data from a specific project |  |

After inspecting the code with Sonar, we found this:



8.3% of duplication – not bad :D

But then, we’ve decided to dig deeper and to run some analysis, and this is what we’ve obtained:



Quite a lot of violations. Everywhere.

After seeing all this statistics, we decided to move straight forward to refactoring.

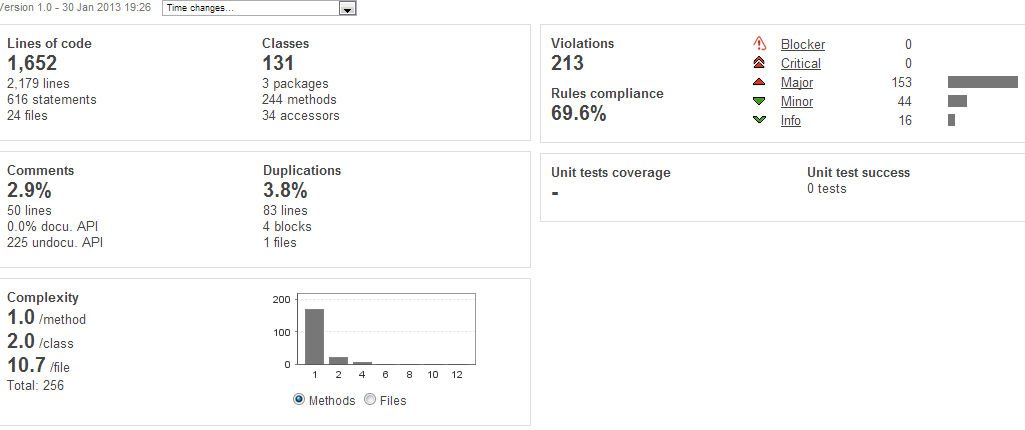
*-----------------A little bit of fun-----------------*

I’ve a project with 23 design patterns implemented and I wanted to see what is their state and quality.

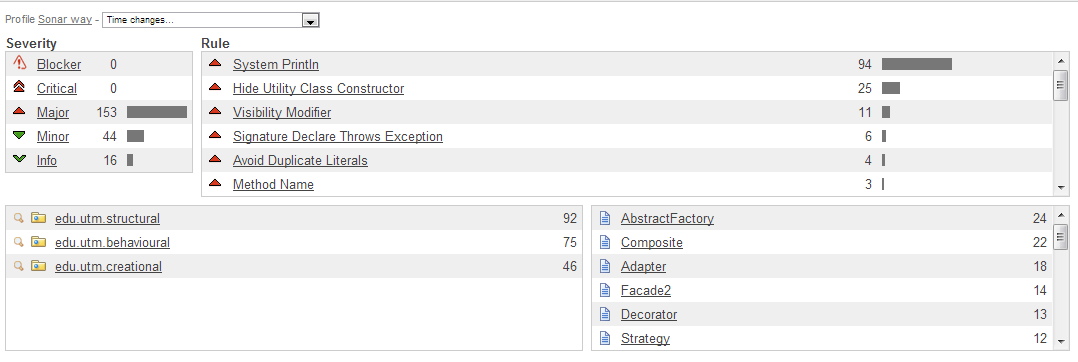
I’ve discovered a couple of things:

* Installing Sonar is not really that bad once you actually read everything and don’t skip the important lines :D
* Boooy, I need to review my code :D

So, here what the situation was (now remember, there is absolutely no dependencies between these files and probably there are similar classes along the patterns (they almost all were based on wizards and the wonders of Harry Potter)):



And if looking closely into all the violations:



*---------- And now to serious stuff----------*

## 2. Refactoring

First of all, we’ve eliminated all the duplications (well, at least those that are not necessary), unnecessary lines in methods (like System.out.println (since it does not belong to the method and it adds additional functionality (bad practice)).

Methods of refactoring used:

**1. Renaming**

The variables that were named not by the standards or had misleading name – were changed.

**2. Extract Superclass**

For connecting to the database through JPA we have to use Persistence Context. Till the refactoring processes, each function that accessed the database – used its own freshly new created Persistence Context.

Instead, we’ve created a GenericService that creates the PersistenceContext, and all the function have to do is to extends the class and use that context. As a result the code becomes cleaner and easier to maintain

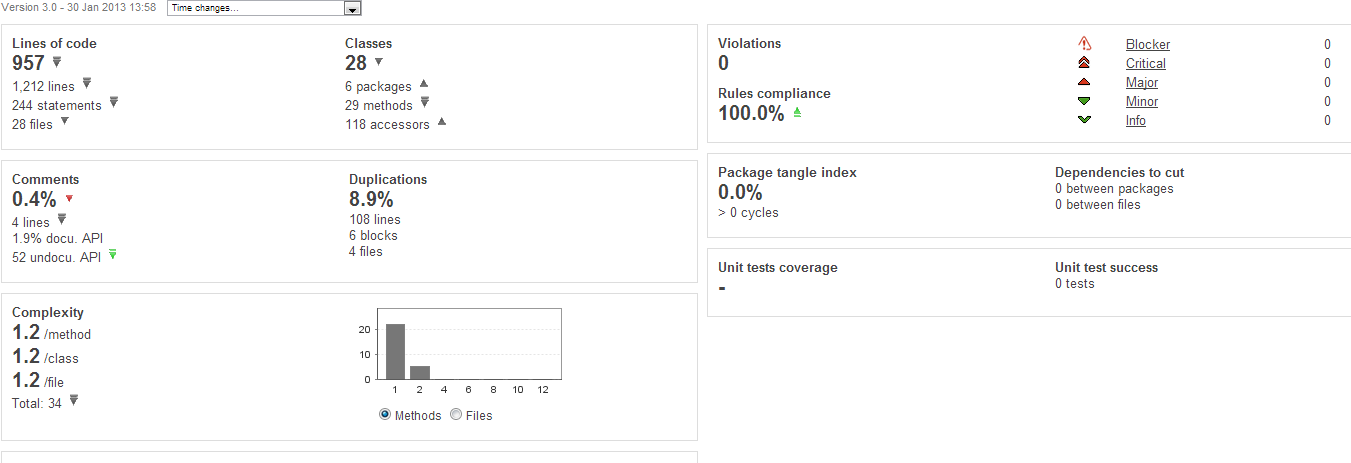
**3. Encapsulated all the fields**

The variables that were accessed directly were “retyped” as private and proper getters and setter method were provided.

**4 + 5. Move classes and packages**

All the classes were moved in the proper packages (also the interfaces).

After refactoring the analysis:



Well, the duplication code increased (it is basically closing all the resources that were opened). But:

- Rules compliance – 100%

- Violations – 0

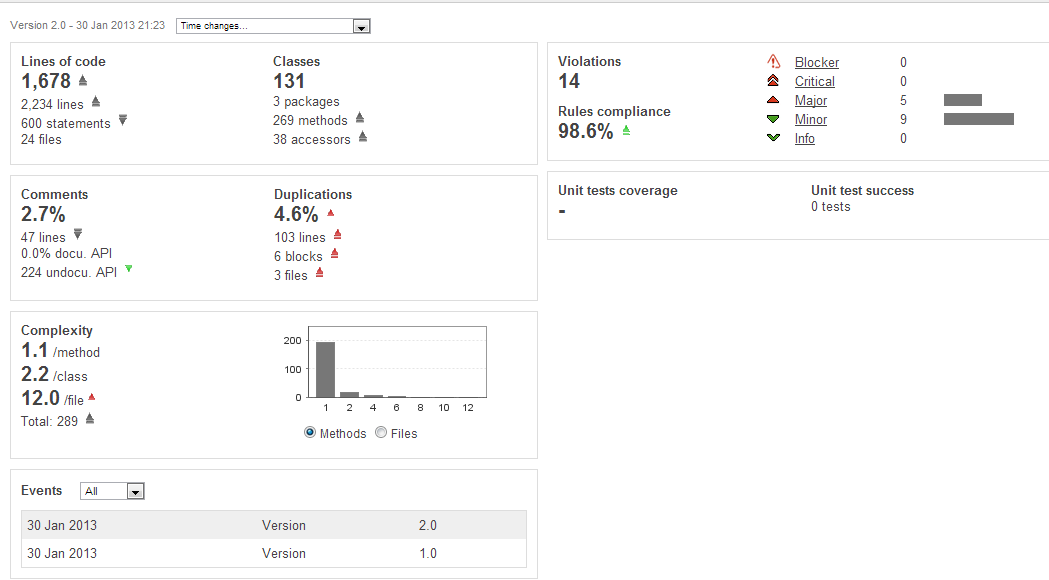
Which is pretty good if you ask me ☺

*----------Back to the fun part -----------------*

So, after seeing all those 200++ violations, I’ve started removing them one by one.

The most common one was the change from “System.out.println” to “System.out.format” – I guess Eclipse or Sonar don’t quite like the println statement.

Anyways, after some time, I’ve got to these results:



Well, the number of violations decreased considerably and the percentage of rules compliance also increased a lot.

Since the java files were not connected one to another and there are absolutely no dependencies between them – the fact that duplication increased – it is not a bad a sign (or a good one).

Looking closely to the causes of violations:



Maybe I still didn’t get used to Sonar, but some of its rules have to have exceptions (well, I want to have protected values in my code, since Java allows it and they actually have their benefits – but Sonar doesn’t want to let me. ☺)

From the refactoring, the most commonly used one was:

**Pull Up – in OOP, move to a superclass**

There was a lot of common functionality in several of methods. It was rewritten just once (and used as many times as we need).

*------------ Coming back to our system ---------------*

## 3. Architectural patterns

Architectural patterns represent the highest-level patterns in the pattern system. They help you to specify the fundamental structure of an application. Every development activity that follows is governed by this structure – for example, the detailed design of the subsystems, the communication and collaboration between different parts of the system, and its later extension.

Each architectural pattern helps you to achieve a specific global system property, such as the adaptability of the user interface. Patterns that help to support similar properties can be grouped into categories.

Basically, the patterns can be grouped in 4 categories:

* **From mud to structure**. Patterns in this category help you avoid a “sea” of components or objects. In particular, they support a controlled decomposition of an overall system task into cooperation subtask. The category includes the Layers pattern, the Pipes and Filters pattern and the Blackboard pattern.
* **Distributed Systems.** This category includes one pattern – Broker and refers to two patterns in other categories, Microkernel and Pipes and Filters. The Broker pattern provides a complete infrastructure for distributed applications.
* **Interactive Systems.** This category comprises two patterns, the Model-View-Controller pattern and the Presentation-Abstraction-Control pattern. Both patterns support the structuring of software that feature human-computer interaction.
* **Adaptable Systems.** The Reflection pattern and the Microkernel pattern strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Let take each architectural pattern and analyze it:

**The Layers pattern** helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

It describes the most widespread principle of the architectural subdivision. Many of the block diagrams we see in the system architecture documents seem to imply a layered architecture.

This is the pattern we used for our application (see lab.work #1)

**The Pipes and Filters pattern** provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

This pattern is not suited for our system since we don’t have that much information circling between different filters. The information goes from the user to the database and vice versa, and it is not passed through different filters or channels. In fact, it will not use channels and pipes at all.

**The Blackboard pattern** is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

This pattern is also is not suited to our system, since it is designed also to implement complex and non deterministic control strategies.

Our system is pretty much predictable, we know what it has to do, what users will do and how everything works.

**Broker Pattern**

Many complex software systems run on multiple processors or distributed computers. There are a number of reasons to distribute software across computers, for example:

* A distributed system can take advantage of the computing power of multiple CPUs or a cluster of low-cost computers.
* Certain software may only be available on specific computers.
* Parts of the software may have to run on different network segments due to security considerations.
* Some services may be provided by business partners and may only be accessed over the Internet.

This pattern is also not suitable for our system since there is no need to use multiple processors or distributed computers.

**Model-View-Controller pattern**

It is a software architecture pattern that separates the representation of information from the user's interaction with it. The *model* consists of application data and business rules, and the *controller* mediates input, converting it to commands for the model or view.A *view* can be any output representation of data, such as a chart or a diagram. Multiple views of the same data are possible, such as a pie chart for management and a tabular view for accountants. The central ideas behind MVC are code reusability and separation of concerns.

Initially, our system was designed for 3-tier architecture and layered architecture. Although they may seem similar, the difference is in topology.

Conceptually the 3-tier architecture is linear. However, the MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

So, MVC pattern could suite our system, but we would have to write the code in such a manner that all 3 components interact.

**Presentation-Abstraction-Control pattern**

It is a software architectural pattern. It is interaction-oriented software architecture, and is somewhat similar to model–view–controller (MVC) in that it separates an interactive system into three types of components responsible for specific aspects of the application's functionality. The abstraction component retrieves and processes the data, the presentation component formats the visual and audio presentation of data, and the control component handles things such as the flow of control and communication between the other two components.

In contrast to MVC, PAC is used as a hierarchical structure of agents, each consisting of a triad of presentation, abstraction and control parts. The agents (or triads) communicate with each other only through the control part of each triad. It also differs from MVC in that within each triad, it completely insulates the presentation (view in MVC) and the abstraction (model in MVC), this provides the option to separately multithread the model and view which can give the user experience of very short program start times, as the user interface (presentation) can be shown before the abstraction has fully initialized.

This pattern is not quite suitable for our system because since PAC is more oriented to agents (e.g. air traffic control)

**Reflection pattern**

It provides a mechanism for changing structure and behavior of software system dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system proprieties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

This pattern is also not suited for our system, since we have a well established structure and it was not designed to change ☺

**Microkernel pattern**

The Microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

The pattern may be applied in the context of complex software systems serving as a platform for other software applications. Such complex systems usually should be extensible and adaptable to emerging technologies, capable of coping with a range of standards and technologies. They also need to possess high performance and scalability qualities; as a result, low memory consumption and low processing demands are required. Taken together, the above requirements are difficult to achieve.

This pattern is more suitable and used for OS. Therefore, it is not applicable for our system.

There is one another architecture I want to discuss (I avoided the word pattern or style, since different sources call it different): three-tier architecture.

Three-tier architecture:

Three-tier is a client–server architecture in which the user interface, functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms.

Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology. For example, a change of operating system in the presentation tier would only affect the user interface code.

Three-tier architecture has the following three tiers:

* **Presentation tier**

This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network.

* **Application tier (business logic, logic tier, data access tier, or middle tier)**

The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application’s functionality by performing detailed processing.

* **Data tier**

This tier consists of database servers. Here information is stored and retrieved. This tier keeps data neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance.

But how this works in web development?

In the web development field, three-tier is often used to refer to websites, commonly electronic commerce websites, which are built using three tiers:

1. A front-end web server serving static content, and potentially some cached dynamic content. In web based application, Front End is the content rendered by the browser. The content may be static or generated dynamically.
2. A middle dynamic content processing and generation level application server, for example Ruby on Rails, Java EE, ASP.NET, PHP, ColdFusion, Perl platform.
3. A back-end database or data store, comprising both data sets and the database management system or RDBMS software that manages and provides access to the data.

**References:**

<http://www.javagyan.com/articles/refactoringtechniques>

<http://www.methodsandtools.com/archive/archive.php?id=4>

<http://refactoring.com/>

<http://docs.codehaus.org/display/SONAR/Configuring+Sonar+in+Eclipse>

<http://docs.codehaus.org/display/SONAR/Sonar+Concepts>

<http://docs.codehaus.org/display/SONAR/Java+Metric+Definitions>

<http://docs.codehaus.org/display/SONAR/Analyzing+with+Sonar+Runner>

<http://docs.codehaus.org/display/SONAR/Installing+and+Configuring+Sonar+Runner>

<http://en.wikipedia.org/wiki/Three-tier_(computing)#Three-tier_architecture>

<http://en.wikipedia.org/wiki/Software_Architecture_styles_and_patterns#Catalog_of_architectural_patterns>

<http://en.wikipedia.org/wiki/Software_Architecture_styles_and_patterns#Catalog_of_architectural_patterns>