blog content

Billboard Management System

Design Patterns and TDD

Completed on 31 May, 2020

The relevance of Design Patterns is a topic of ongoing debate within the software industry. Initially, these patterns were highly valuable as they facilitated effective communication of architectural ideas among professionals. However, with the decline in the tradition of studying Design Patterns among new practitioners, their utility has been called into question. Whether this shift is beneficial or detrimental, the fact remains that I have the esteemed text by Erich Gamma et al. on my shelf, which serves as a valuable resource for understanding and applying these patterns

As part of a team of four computer science students, I had the opportunity to contribute to the development of the Billboard Management System. Throughout the project, we leveraged various design patterns to improve the system's architecture and also delved into the practice of test-driven development (TDD). My official contributions to the project, as documented in the paperwork, include actively implementing design patterns to enhance the system's functionality and advocating for the adoption of TDD principles, ensuring robust code quality and facilitating future modifications.

- Test Driven Development of Billboard.Java
- Design, writing and refactor of Billboard class
- Wiring design, implementation and refactor of GUIBillboardPanel class
- Writing and testing of XMLFileOpener class
- Writing and testing of IMGFileOpener class
- Writing and testing of URL Handler class
- Writing of report walk through

In this project, students were provided with a requirement specification document along with an overview of the business case, commonly referred to as the "scenario.". In

essence, this was:

Our development team was contracted by a client corporation to create a comprehensive system for managing their electronic billboards. The client, with multiple screens spread across their offices nationwide, required a solution that would facilitate the display of announcements, advertising, and motivational content. Our task was to design and develop a robust system that would streamline the management and scheduling of content across these electronic billboards, ensuring efficient and impactful communication within the client's organisation.

This corporation requires three separate applications:

1. **1. Billboard Control Panel**
    ◦ This application serves as a user interface that enables users to perform Create, Read, Update and Delete (CRUD) operations on Billboards. Additionally, it facilitates the scheduling of a Billboard and allows an admin to create new users and define their permissions over billboard creation and scheduling.
2. **2. Billboard Viewer**
    ◦ This application enables the viewing of a Billboard on a device. While running, this application sends a request for the next scheduled Billboard.
3. **3. Billboard Server**
    ◦ This application functions as a server for both the Billboard Viewer and Billboard Control Panel applications. It handles incoming requests to carry out various actions, such as creating a new billboard through the Billboard Control Panel and retrieving the next billboard to be displayed through the Billboard Viewer.

The applications in question were intended to be interconnected through the corporation's intranet, ensuring that the Billboard viewer and control panel would not communicate directly. The primary function of the billboard viewer was to display content according to the instructions provided by the Billboard server. The high-level network architecture, depicted in the figure below, illustrates the overall connectivity and flow of information within the system.

The Model-View-Controller (MVC) architectural design pattern is employed in this project. In MVC, the user interacts with the controller to make changes to the model. The model serves as the authoritative data source, updating the viewer to reflect these changes for the user to observe.

MVC provided the team with 3 main benefits:

1. **1. Model-View Cardinality**
    ◦ One of the key benefits of the Model-View-Controller (MVC) pattern is its inherent scalability, particularly when it comes to accommodating multiple viewers. With MVC, only a single instance of each viewer needs to be created, which can then be displayed on multiple screens. This one-to-many relationship between the viewers and screens proved to be crucial in fulfilling our business case requirements.
2. **2. Asynchronicity**
    ◦ The advantage of asynchronous development is that it empowers our team to be autonomous and efficient by enabling independent development of each application.
3. **3. Unit testing**
    ◦ The built-in modularity provided by MVC development allows us to achieve better test coverage across each application with greater ease.

I welcome the reader to explore the repo to get a detailed understanding of how this system is constructed. The following will discuss my experience using Driven Development (TDD), our use of mocking and finally tracer-bullets.

Test-driven development

```
publicclassBillboard {

private String createdBy;
private String message;
private String information;
private String picture;
private String bgColour;
private String messageColour;
private String informationColour;
private String name;
```

Test-Driven Development (TDD) is a software development practice that prioritizes defining the functional utility of a method before its mechanics. This approach involves writing functional tests prior to writing any code. The benefits of TDD are significant, as it results in code with extensive test coverage, leading to easier maintenance and scalability.

In our system, we employ a class called "billboard" to encapsulate all the necessary information for representing a billboard object. This decision was made due to its

frequent usage across multiple classes involved with the Control Panel Client. By addressing this early in the development process, we aimed to mitigate future risks effectively.

To adhere to TDD principles, our team divided the test writing and fulfillment responsibilities between myself and a team member. Test fulfillment was considered achieved when the tests passed in the JUnit output window (note: JUnit is the testing utility available in IntelliJ). Once fulfilled, the tests could be refactored and re-tested to ensure they maintained the same functionality. Throughout the development process, as the application grew, these tests, along with others, were continuously run to ensure the desired behavior of the class.

My personal experience with TDD was overwhelmingly positive for several reasons. Firstly, the process compelled me to conceptualize and plan the entire class and its uses before writing any methods. It served as a blueprint, allowing me to assert and validate requirements only after all tests had passed. This ensured that once the tests were successful, the functionality was considered complete, and any additional work focused on improving the code.

Secondly, TDD provided a sense of peace of mind while working on the class. Knowing that the JUnit tests were passing gave me confidence that the related methods were functioning as intended. It allowed me to concentrate deeply on the problem at hand by abstracting myself from the exact inner workings of each function.

Lastly, TDD offered a new perspective for gauging development progress. Since tests are written first, it becomes easier to track the completion of each method, assuming the project requires a testing suite. Having tests written beforehand ensures that at the end of each development cycle, each method has its own test(s). This approach is more efficient than estimating development speed upfront, as it avoids the potential waste of efforts in implementing a function that fails to meet its requirements.

In summary, TDD not only improved the overall quality of the codebase but also provided a structured and focused approach to development. It encouraged thoughtful planning, instilled confidence through passing tests, and offered valuable insights into development progress.

Requests, Responses and Mocking

In the MVC pattern, the controller assumes the responsibility of modifying the model. In the context of the BMS (Billboard Management System), it is the control panel GUI that empowers users to perform actions such as creating, editing, importing, and scheduling billboards on the viewer. Consequently, a significant portion of the information exchange occurs from user interactions on the control panel, which is then transmitted to the server. Apart from billboard management, users also have the ability to create and assign privileges to other users for specific features. Given the extensive range of features that involve modifying the model, it became imperative for us to establish a standardized approach for passing data from the control panel to the server. To address this requirement, we implemented a

request

and

response

pattern between these two applications, extending its implementation to the database interface as well. This pattern facilitated consistent and structured data exchange, offering several immediate benefits to our system's functionality and performance.

1. Readable code

helpers class

1. Communication speed
2. Extendability

Tracer-bullets

Developing software as an undergraduate student has exposed me to various technologies and programming patterns that were previously unfamiliar to me. While I had a basic understanding of object-oriented programming (OOP) and Java before embarking on this project, I lacked knowledge on how to network multiple Java applications together. This created uncertainty about whether our chosen data encapsulation approach would seamlessly transfer across the network. Making critical design decisions early on without considering the network implications could result in the need to write unnecessary parsers or undertake extensive code refactoring. My prior experience had taught me the costliness of reworking components within a data pipeline.

To mitigate the risks associated with major refactors later on, I advocated for the use of tracer bullets. This method served as a guide, allowing us to trace how data flows between different technologies within the application(s). By understanding this process early in the development stage, we could determine how data needed to be handled at each point along the pipeline. For instance, it was crucial to comprehend how a user's action, such as pressing the delete button on the billboard control panel, would translate into a MySQL command for removing a row from the billboards table. Through the use of tracer bullets, we discovered that we couldn't directly pass objects in our request functions to the server. Instead, objects had to be explicitly serialized. Consequently, we developed a function that would convert our "request" message, which included encapsulated data, into bytes that could be transmitted across the network to the server.

Summary

Throughout this project, the value of employing pre-defined design patterns became evident. While MVC (Model-View-Controller) is not a new pattern, its continued relevance in modern, large-scale applications indicates its versatility across a wide range of use cases. By embracing Test-Driven Development (TDD), our team confidently progressed in the development process, ensuring that the first classes we created operated precisely as intended. Additionally, our client-server interface utilized a straightforward request and response pattern that all software developers could easily comprehend, guaranteeing the code's extensibility.

Developing the databases early on could have been a costly endeavor, considering the potential need for frequent updates and maintenance throughout the development process. To address this, we devised a clever solution by implementing a Mock database. This allowed us to generate pseudo data instead of relying on actual SQL results. Consequently, we avoided making changes to tables and scripts while the database requirements were still in the formulation phase.

Lastly, our utilization of tracer-bullets provided valuable insights into how data should flow across our applications, significantly reducing the risk of misunderstanding how we should encapsulate our data. These tracer-bullets acted as our trusty guides, ensuring our data journeyed seamlessly through our system without any confusion or misadventures along the way.