Design Goals

FDM05

Contents

1. Design Considerations

The two main categories in designing software applications are efficiency and elegance. Both will be investigated in-depth alongside industry standard good practices and design patterns

1. Efficiency

Designing for efficiency should improve the user experience (UX) of an application.

1.1.1. Performance

Designing for performance aims to reduce the time taken to perform system actions, to improve UX for a system. It can be argued that UX is important for customer satisfaction and therefore customer retention (Badran, 2018), which makes performance a viable design goal. The goal is also particularly important where actions may be time sensitive, such as buying or selling shares on the stock market at a price which is subject to change over time.

Implementing a performant system requires intelligent usage of memory, CPU, and network resources. This includes choosing the correct communication methods and architectures between application layers, such as: HTTP REST APIs, Graph QL, Microservices, WebSocket etc. Using these tools intelligently with project requirements in mind can streamline performance immensely.

In addition to network performance, the written code and algorithms must have a quick run time through careful use of memory and processing power. However typically, good practices and performance are not necessarily complimentary of one another. There may be trade-offs to be had in the design phase to allow for rapid development and performant code (Ayata, 2010), and there is merit in not over-engineering too early. Performance can be easier factored in retroactively when bottlenecks are visible in a complete application, through stress-testing or deployment into a production environment.

There are potential UX workarounds to mask performance issues, which communicates to a user that the system is processing information. In relation to this application, trade prices are time sensitive, but techniques such as immediately locking in the user's requested price until completed can reduce frustration as the application functionality is not impeded. Web is also an ideal medium to convey information to users where actions are taking a long time, due to an abundance of front-end and visual tools. Performance would only be entirely crucial if a more high-frequency trading functionality was required.

1.1.2. Reliability

Reliability in software design encapsulates the expectation for an application to function correctly. Correctness can be determined by all functional requirements being met, the absence of bugs or crashes during runtime, and proper handling of incorrect and invalid data or inputs. Reliable code is paramount in any application. It must function as expected, and security is crucial for customer details, especially where banking information or financial circumstances are concerned.

Reliability can be enforced through the implementation of a robust test plan. Such areas could include unit, component, integration, automation, and penetration testing, which would cover a significant amount of the codebase at both a high and low level.

Security is a particularly interesting focus as there are an abundance of flaws to avoid in web applications which can be designed against. The 'OWASP Top 10' (OWASP Top Ten Web Application Security Risks, 2021) is a perfect starting resource to guide developers on what to look out for. For example, SQL injection prevention is specifically listed as a requirement for this project, which sits at number one in the top ten, and one way to mitigate this scenario is by escaping user input (removing characters that may close a string, allowing SQL queries to be modified outside of the string) as demonstrated by OWASP (SQL Injection Prevention - OWASP Cheat Sheet Series, 2021).

Due to the points discussed, reliability is a significant design goal for this project. It will be measured on completion with fully documented and completed tests in each area.

1.2. Elegance

Designing for elegance makes an application easier to develop and maintain.

1.2.1. Readability

A key design consideration in software development is readability – ensuring the codebase can be easily understood. Therefore, it will be easier to make sense of the code for a new developer or on return to previously written areas. Readable code is a standard that should be upheld in most cases and is typically an industry standard when working in or across teams of developers.

This can be done using design patterns. These recurring patterns between codebases can provide familiarity to developers, as they are seen widely throughout the industry.

However, readability is sometimes a barrier to efficient, or more specifically, performant code (Ayata, 2010). Managing memory and processing power effectively can sometimes create difficult to understand code, and the existence of design patterns and their increase in abstraction can introduce more computational complexity in favour of readability and familiarity.

Unfortunately, designing for readability at a high-level is difficult as it should really be enforced through good practice. The benefits also don't outweigh the *potential* performance impacts, as there is no need for readability between multiple developers in this project, while also being difficult to measure. While readable code will always be a consideration throughout development, it will not be paramount in the design phase.

1.2.2. Reusability

Another consideration within software elegance is reusability and writing multipurpose code only once. This inherently means less code, which in turn means less development time, fewer testing requirements, and lower costs in commercial projects. Reusability is a standard that should be strived for in most cases, as it reduces resource requirements significantly.

Object-oriented programming (OOP) good practices, and SOLID principles encourage reusability, and the importance of such show how crucial this design goal can be (Turan and Tanriöver, 2018). Therefore, techniques such as DRY (Don't Repeat Yourself), inheritance, and polymorphism are great measures in ensuring this goal is met. However, it can sometimes be difficult to foresee what elements of the code *should* or *can* be re-used effectively and can be an iterative process in an agile environment.

Due to the simplicity of measuring this design goal through evidence of SOLID, and the impact it can have when limited by resources, reusability will be a great key focus for this project.

1.2.3. Maintainability

Finally, maintainability is another crucial element in application design. Code must be written in a way that is adaptable to shifting requirements, therefore allowing functionality to be swapped in and out much easier. While it is typically important to focus on this in an agile environment where requirements are more subject to change, building an application for maintainability is generally good practice as it may help in supporting the application in future, fixing bugs, or making slight design changes down the line (Mendes and Tempero, 2009).

Well defined and loosely coupled components (e.g., microservices) would be an ideal way to approach maintainability, as it would be theoretically simpler to add and remove small pieces of functionality, rather than untying large knots in the code. The less references there are to a piece of code, the easier it will be to remove.

This project contains some robust and clear project requirements and there is no anticipation that they will change, however, due to the nature of good OOP practice it would be a great focus area for this project. It is worth noting that this area would also be one of the most difficult to implement into an existing application, as a tightly coupled codebase would be a monolithic task to modularise.

Maintainability is also a difficult area to measure until additional features are required, and maybe could be proved easier with the use of a tiered scoping system, or categorisation of features into an MVP (minimum viable product). Nevertheless, it could be measured upon completion and reflection of the development process, by assessing how easy functionality was added throughout; a maintainable design should not hinder the development process.

2. Development Methodology

The following measures may be used in software design to support or contradict certain design goals.

2.1. Cohesion and Coupling

Two valuable metrics to consider in software engineering for elegance are **cohesion** and **coupling**.

Cohesion describes how cohesive a particular module is, by evaluating the actions it was built to perform. A module with **low cohesion** would perform a variety of actions that aren't necessarily linked in any meaningful way, which may cause confusion in any large project. However, a module with **high cohesion** would be focused in its design, where its actions are very similar to one another.

Coupling refers to the interconnection of different modules and can be evaluated by understanding each modules' reference to one another. **Tight coupling** would be descriptive of a codebase containing modules that contain many references to other modules. In contrast, **loose coupling** describes a codebase containing modules which contain very little reference to other modules.

High cohesion and loose coupling are very indicative of clean code and are an important consideration in software design. There are many methods discussed below in which these metrics can be targeted.

2.2. OOP Pillars

Clean object-oriented code is generally defined by employing the 4 pillars:

- **Abstraction:** Encourages clean code by hiding extraneous detail away from the usage of a particular piece of code.
- **Encapsulation:** Employed similarly to abstraction, make use access modifiers to hide data from external classes.
- **Inheritance:** Ensure object hierarchy allows reusability from parent to child classes.
- **Polymorphism:** An extension of inheritance whereby child classes may include various implementations for the same function.

2.3. SOLID

SOLID is a set of design principles outlined by Robert C. Martin to help in ensuring software designs are more understandable, flexible, and maintainable (2000; 2003).

- **Single Responsibility Principle:** One class should only have a single reason to change. This principle is synonymous with high cohesion.
- **Open / Closed Principle:** Build code to be extended, not modified. New code changes should inherit existing behaviour and refrain from touching existing source code.
- **Liskov Substitution:** Derived classes should be substitutable for their parent. In extension of the open / closed principle, this will ensure subclasses are extending behaviour correctly.
- **Interface Segregation:** Interface clients should not be forced to implement unused behaviour. When using an object-oriented language, creating behaviour specific interfaces can help with sticking to this principle.
- **Dependency Inversion:** Build classes to depend only on abstractions. This principle can encourage loose coupling by removing references to concrete implementations throughout the codebase.

There is recent debate over SOLID and its applicability to modern software (Mardjan, 2019; Dunn, 2021). However, the principles were written in the golden age of OOP, and some of the advice doesn’t always translate well into modern development trends. In fact, it may just be simpler to focus on writing simple code as it is needed than overengineering through SOLID guidelines. The overall verdict seems to be: implement SOLID where appropriate and don’t write unnecessary abstraction.

2.4. Design Patterns

There are many design patterns that may support or contradict the key design goals. In addition to the twenty-three classic patterns (Gamma et al., 1995), more patterns are discovered as an ever-growing set of problems are solved over time. Common design patterns are easily maintainable due to their applicability to common problems throughout many codebases. This is because they are easily recognised, and over time can become a crucial tool within a developer’s toolset.

However, design patterns are not perfect pathways to success and should be used only for the appropriate problems.

3. Discussion

Each potential design goal is equally weighted in importance, and a perfect world would allow for a design to conform perfectly to each. However, the project requirements provide guidance in which may benefit the software solution best.

Based on the analysis provided above, the design goals in focus for this project are:

- Reliability
- Reusability
- Maintainability

References

Ayata, M., 2010. *Effect of Some Software Design Patterns on Real Time Software Performance.* Postgraduate. The Graduate School of Informatics of Middle East Technology. Available at: <https://open.metu.edu.tr/bitstream/handle/11511/19615/index.pdf>.

Badran, O., 2018. The Impact of Software User Experience on Customer Satisfaction. *Journal of Management Information and Decision Sciences,* [online] 21(1). Available at: <https://www.abacademies.org/articles/The-impact-of-software-user-experience-on-customer-satisfaction-1532-5806-21-1-116.pdf>.

Dunn, S., 2021. SOLID - is it still relevant?. [Blog] *Steve Dunn*, Available at: <https://dunnhq.com/posts/2021/solid-relevance/> [Accessed 3 May 2021].

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. *Design patterns: elements of reusable object-oriented software.* 1st ed. Reading, MA: Addison-Wesley.

Mardjan, M., 2019. Rise and fall of SOLID programming principles. [Blog] *No Complexity*, Available at: <https://nocomplexity.com/solid-programming/> [Accessed 5 May 2021].

Martin, R., 2003. *Agile software development*. 1st ed. Upper Saddle River, N.J.: Pearson Education.

Martin, R., 2000. *Design Principles and Design Patterns.* [ebook] Available at: <http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf> [Accessed 7 May 2021].

Mendes, E. and Tempero, E., 2009. A systematic review of software maintainability prediction and metrics. In: *Empirical Software Engineering and Measurement (ESEM).* [online] Lake Buena Vista, FL: IEEE, pp.367-377. Available at: <https://www.academia.edu/14385485/A_systematic_review_of_software_maintainability_prediction_and_metrics>

OWASP. 2021. *OWASP Top Ten Web Application Security Risks.* [online] Available at: <https://owasp.org/www-project-top-ten/> [Accessed 10 April 2021].

OWASP. 2021. *SQL Injection Prevention - OWASP Cheat Sheet Series.* [online] Available at: <https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html> [Accessed 10 April 2021].

Turan, O. and Tanriöver, Ö., 2018. An Experimental Evaluation of the Effect of SOLID Principles to Microsoft VS Code Metrics. *Online Academic Journal of Information Technology*, 9(34). Available at: <https://dergipark.org.tr/tr/pub/ajit-e/issue/54418/740683>