Nicole P. Satiembre **March 27, 2024**

Cebu Technological University-Main Campus

Clean Code

Chapter 11 - Systems

How would you build a City? Managing requires teams of people that work on different things such as the traffic systems, water systems, law inforcements, etc. Some of those people are responsible for the bigger picture while the others are for the details. This system is possible in building a City because there is an appropriate abstraction of tasks where people can manage to work effectively without the need to understand the bigger picture. Although it is possible for software teams to be organized like that too, the systems they work on often don't have the same separation of concerns and levels of abstraction. Clean code helps us achieve this at the lower levels of abstraction.

Separate Constructing a System from Using it. It is important to distinguish between the construction usage phases in both physical infrastructure, like building a hotel, and software systems. We must apply separation of concerns, particularly in software development, urging developers to isolate the startup process from runtime logic to enhance maintainability. The drawbacks of lazy initialization, such as hard-coded dependencies and violation of the Single Responsibility Principle, are highlighted, stressing the importance of adopting practices like dependency injection and ensuring a global, consistent strategy for resolving dependencies.

Separation of Main. This shows how abstraction us applied in separating construction from usage. You just need to simply to move all aspects of construction to main, or modules called by main, and to design the rest of the system assuming that all objects have been constructed and wired up appropriately. This means that the application has no knowledge of main or of the construction process.

Factories. There are instances where we need to make an application responsible for when an object gets created. We can give an application control on building a part of the system but keeping the details of that construction separate from the application code.

Dependency Injection. A powerful mechanism for separating construction from use is Dependency Injection (DI), the application of Inversion of Control (IoC) to dependency management. Inversion of Control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the Single Responsibility Principle. True Dependency Injection involves classes being completely passive regarding dependency resolution, instead relying on setter methods or constructor arguments to inject dependencies.

Scaling Up. Just like cities starting from small towns to big skyscrapers, software systems also scale up from a small crappy program to what it is today. It is a myth that we can get systems “right the first time.” Instead, we should implement only today’s stories, then refactor and expand the system to implement new stories tomorrow. This is the essence of iterative and incremental agility. Test-driven development, refactoring, and the clean code they produce make this work at the code level.

Cross-cutting Concerns. The EJB2 architecture attempts to separate concerns by declaring transactional, security, and persistence behaviors in deployment descriptors independently of source code. However, cross-cutting concerns like persistence often lead to the scattering of implementation code across multiple objects. Aspect-oriented programming (AOP) anticipates this issue by allowing aspects to specify consistent modifications to behavior across the system, enabling non-invasive modifications to target code.

Java Proxies. Java proxies are suitable for simple situations, such as wrapping method calls in individual objects or classes. However, the dynamic proxies provided in

the JDK only work with interfaces. To proxy classes, you have to use a byte-code manipulation library, such as CGLIB, ASM, or Javassist. Code volume and complexities are the two drawbacks of proxies. They make it hard to create clean code! Also, proxies don't provide a mechanism for specifying system-wide execution "points" of interest, which is needed for a true AOP solution.

Pure Java AOP Frameworks. Fortunately, most of the proxy boilerplate can be handled automatically by tools. Proxies are used internally in several Java frameworks, for example, Spring AOP and JBoss AOP, to implement aspects in pure Java.

AspectJS Aspects. The most full-featured tool for separating concerns through aspects is the AspectJ language,17 an extension of Java that provides "first-class" support for aspects as modularity constructs.

Test Drive the System Architecture. The passage emphasizes the importance of starting software projects with a straightforward, flexible architecture that focuses on delivering working features quickly. It suggests that overcomplicating things with complex APIs, like early versions of EJB, can hinder progress. Instead, it advocates for a simple approach using basic Java Objects, integrated together smoothly using easy-to-implement tools. This way, teams can stay agile and adapt to changes easily, ensuring they deliver value to customers efficiently. This approach enables test-driven development, fostering efficient delivery of optimal value to customers by focusing creative efforts on implementing user stories rather than architectural constraints.

Optimize Decision Making. When building a large system, it is best to give the decision making responsibilities to the qualified people. However, we often forget that it is also best to postpone decisions until the last possible moment. It lets us make informed choices with the best possible information. A premature decision is a decision made with suboptimal knowledge. We will have that much less customer feedback, mental reflection on the project, and experience with our implementation choices if we decide too soon.

Use Standards Wisely, When They Add Demonstratable Value. Standards make it easier to reuse ideas and components, recruit people with relevant experience, encapsulate good ideas, and wire components together. However, the process of creating standards can sometimes take too long for industry to wait, and some standards lose touch with the real needs of the adopters they are intended to serve.

Systems Need Domain Specific Languages. Building construction, like most domains, has developed a rich language with a vocabulary, idioms, and patterns that convey essential information clearly and concisely. In software, there has been renewed interest recently in creating Domain-Specific Languages (DSLs), which are separate, small scripting languages or APIs in standard languages that permit code to be written so that it reads like a structured form of prose that a domain expert might write. A good DSL minimizes the “communication gap” between a domain concept and the code that implements it, just as agile practices optimize the communications within a team and with the project’s stakeholders.