APPENDIX A

Software Design Principles

The Oxford English Dictionary defines "principle" as "a fundamental truth or proposition that serves as the foundation for a system of belief or behavior or for a chain of reasoning." Software design principles lay down fundamental guidelines for designers and developers to effectively build, understand, and maintain a software system.

A number of design principles have been documented in literature. This appendix only lists the principles that have been mentioned in this book for your quick reference.

A.1 ABSTRACTION

The principle of abstraction advocates the simplification of entities through reduction and generalization: reduction is by elimination of unnecessary details and generalization is by identification and specification of common and important characteristics [2].

A.2 ACYCLIC DEPENDENCIES PRINCIPLE

Acyclic dependencies principle (ADP) says that "The dependencies between packages must not form cycles" [10]. ADP mainly targets package relationships and thus the general tendency is to consider its applicability at the architecture level only. However, cyclic dependencies are undesirable at the class level as well. Dependency cycles impose a constraint: Entities that are part of a cycle need to be used, tested, developed, and understood together. In case of cyclic dependencies, changes in one class/package (say A) may lead to changes in other classes/packages in the cycle (say B). However, because of the cyclic nature, changes in B can have ripple effects on the class/package where the change originated (i.e., A). Large and indirect cyclic dependencies are usually difficult to detect in complex software systems and are a common source of subtle bugs. To summarize, avoid introducing direct/indirect cycles between classes/packages.

A.3 DON'T REPEAT YOURSELF PRINCIPLE

The DRY principle states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [11]. The principle advocates "single source of truth" philosophy and is broadly applicable to all software artifacts including documents, architecture and design, test code, and source code. In the context of our discussion of detailed design, duplication in design entities and code could manifest in the form of duplicated type names and duplicated implementation.

Intentional or inadvertent duplication is a common source of subtle defects. Therefore, duplication in any form should be avoided in software systems.

A.4 ENCAPSULATION

The principle of encapsulation advocates separation of concerns and information hiding through techniques such as hiding implementation details of abstractions and hiding variations.

A.5 INFORMATION HIDING PRINCIPLE

The information hiding principle advocates identifying difficult or likely-to-change design decisions, and creating appropriate modules or types to hide such decisions from other modules or types [41]. This design approach helps to protect other parts of the program from extensive modification if the design decision is changed. In this book, we also refer to the principle of information hiding to mean that the public interface of an abstraction should expose the "what" aspects of the abstraction and not the "how" aspects.

Adherence to this principle makes the clients of the abstraction less vulnerable to change when implementation details of the abstraction are modified, thereby improving the changeability and extensibility of the software design.

A.6 KEEP IT SIMPLE SILLY

The keep it simple silly (KISS) principle states that simplicity should be a key goal while designing a software system by avoiding the introduction of unnecessary complexity. In this book, we refer to the KISS principle as stated by Tony Hoare [12]: "… there are two ways of constructing a software design: *One way is to make it so simple that there are obviously no deficiencies* and the other way is to make it so complicated that there are no obvious deficiencies."

A.7 LISKOV'S SUBSTITUTION PRINCIPLE

The formal definition of Liskov's substitution principle (LSP) [36] is: "…What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T." An informal description of LSP [14] is: "derived classes must be usable through the base class interface without the need for the user to know the difference." In other words, all subtypes must provide the behavior at least promised by

the supertype and an instance of a subtype should be replaceable wherever a reference of its supertype is used.

A.8 HIERARCHY

The principle of hierarchy advocates the creation of a hierarchical organization of abstractions using techniques such as classification, generalization, substitutability, and ordering.

A.9 MODULARIZATION

The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition.

A.10 OPEN/CLOSE PRINCIPLE

The Open/Close principle states that a "module should be open for extension but closed for modification". In particular, a module should be able to support new requirements without its code being modified. According to Bertrand Meyer [24], once the implementation of a type is complete, the type could only be modified to fix bugs; any new requirement that needs to be supported has to be achieved by extending the class without modifying the code within that class.

A.11 SINGLE RESPONSIBILITY PRINCIPLE

According to Robert C. Martin [15], "There should never be more than one reason for a class to change." Each responsibility is an axis of change, thus each change should impact a single responsibility. When a class has multiple responsibilities, it takes more time and effort to understand each responsibility, how they relate to each other in the abstraction, etc. This impacts understandability negatively. Further, it is difficult to figure out what members should be modified to support a change or enhancement. Further, a modification to a member may impact unrelated responsibilities within the same class. This makes maintenance of the class difficult.

A.12 VARIATION ENCAPSULATION PRINCIPLE

The variation encapsulation principle advocates a form of information hiding and suggests encapsulating the concept that varies [54]. This design principle is reflected in a number of design patterns including Strategy, Bridge, and Observer.