OBJECT-ORIENTED DESIGN PRINCIPLES FOR PACKAGES

BY PANKAJ KAMTHAN

1. INTRODUCTION

There can be no doubt that all our knowledge begins with experience.
— Immanuel Kant

There is a need for **experiential knowledge** during software design. The object-oriented design principles (OODP)[1] are embodiment of such knowledge that is applicable to object-oriented design (OOD).

This document presents an **adapted** and **annotated** collection of OODP, amalgamated from various **resources** [De Marco, 1979; Wirth, 1995; Meyer, 1997; Knoernschild, 2002, Chapter 1; Garland, Anthony, 2003, Section 11.3; Martin, Martin, 2006; Dooley, 2011, Chapter 10; Almugrin, 2015]. These OODP reflect a confluence of **collective expertise and experience** of many people in the software engineering community. The OODP included here are relevant to **software architecture description**, especially towards **package design** aiming for **longevity**.

2. MOTIVATION FOR OODP

The best designers sometimes disregard the principles of design. When they do so, however, there is usually some compensating merit attained at the cost of the violation. Unless you are certain of doing as well, it is best to abide by the principles.
— William Lidwell, Kritina Holden, Jill Butler

I didn't fail ten thousand times. I successfully **eliminated, ten thousand times**, materials and combinations which wouldn't work.
— Thomas Alva Edison

The most difficult design task is to find the most appropriate decomposition of the whole into a module hierarchy, minimizing function and code duplications.
— Niklaus Wirth

[1] The term OODP is used both as singular and plural.

There are a number of reasons for a commitment to OODP:

- The OODP are a culmination of (past) experience, including **successes and failures**[2], in designing object-oriented software systems.

- The notion of **quality and principles are intrinsically related**. The OODP could be viewed as one of the **preventative means** for **improving the quality (especially, related to evolvability or maintainability) of OOD**.

- The use of OODP can help avoid the **development and accumulation** of **grime** in design [Griffith, 2015].

- The OODP **transcend** the **idiosyncrasies** of particular programming languages. In fact, OODP are **independent** of any programming language.

3. HISTORY AND EVOLUTION OF OODP

The history of OODP is goes at least as far back the **Principle of Information Hiding** [Parnas, 1972], in general, and the **Law of Demeter (Principle of Least Knowledge)**[3] [Lieberherr, Holland, 1989], in particular.

4. PRELIMINARIES

There is a need to introduce certain basic terms and their definitions prior to a discussion of OODP.

OBJECT

Definition [Object] [Wirfs-Brock, McKean, 2003]. An object embodies a set of roles with a designated set of responsibilities.

Remark. There are **other definitions** of an object: some define an object in terms of its **properties** (identity, behavior, and state); others define an object in terms of the **class** to which the object belongs to.

[2] In today's rapid, consumerized, software development environment, an organization cannot afford to have ten thousand failures in its path to success!
[3] URL: http://www.ccs.neu.edu/research/demeter/ .

CLASS

Definition [Class]. A class is an organizational unit of objects with similar structure and behavior.

Remark. The notion of class is necessary for **class-based** object-oriented programming languages (OOPL).

PACKAGING AS A DAILY ROUTINE

In daily lives, it is common for **human beings** to **organize (semantically-related) items** that **change and grow** overtime, such as **clothing, bills, groceries**, and so on.

To do that, there are organizing entities, such as **racks, files, cupboards**, and so on, respectively. It helps people to be more **productive**. For example, such organization can help people easily find or move collections of physical products.

There is also a need to **organize information** at all stages of software development and, subsequently, operation. For example, people use folders to organize personal and professional documents.

PACKAGE

The **increase in size and complexity** of software systems necessitates a **higher level of abstraction** during design and implementation of the systems in question. This abstraction is provided by the notion of a package.

Definition [Package]. A package is an organizational unit consisting of a collection of (related) classes.

For example, a package can represent a **sub-system** in a **software system**, or a **component** in a **software architecture**.

Remark. The notion of 'package' **evolved** over time. It became a part of the object-orientation vocabulary after inception of the Java programming language.

Remark. The notion of 'package' is important as a distribution mechanism for **release engineering** in open source software development [Abate, Cosmo, Treinen, Zacchiroli, 2013].

Remark. The growth in **size and complexity** of software systems has motivated the need for some kind of high-level organization. Indeed, a typical software system may have a large number of classes. The grouping of classes into packages allows **reasoning** about the **design** at a higher level of abstraction.

Remark. The **Unified Modeling Language (UML)** is aimed towards modeling **object-oriented systems**, and therefore has **native support** for diagram types that can model (the structure and/or behavior, as appropriate, of) objects, classes, and packages.

DESIGN

Definition [Top-Down Design]. In a top-down approach to design, the packages are designed **before** classes.

For a Top-Down Design approach to make sense, it should be possible to identify packages relatively easily as compared to classes.

Definition [Bottom-Up Design]. In a bottom-up approach to design, the classes are designed **before** packages.

5. A CLASSIFICATION OF OODP

The OODP can be broadly classified into **two categories**, those related to classes and those related to packages:

1. **Class-Level OODP.** The OODP related to classes address design concerns for managing the information in classes and relationships between classes.

2. **Package-Level OODP.** The OODP related to packages address concerns for managing the information in packages and relationships between packages. This can be relevant to **software architecture**.

The **principles of package cohesion** help towards **allocation of classes to packages**. These principles depend on the fact that at least some of the classes and their interrelationships have been discovered. Thus, these principles take a **bottom-up view** of partitioning.

The **principles of package coupling** help towards determining how **interrelationships between packages** should be realized.

6. A COLLECTION OF OODP

The more science becomes divided into specialized disciplines, the more important it becomes to find unifying principles.
— Hermann Haken

The principles are there to help us eliminate bad smells. They are not a perfume to be liberally scattered all over the system.
— Robert C. Martin

Table 1 presents a list of design principles.

Scope	Concern	OOD Principle
Package	Cohesion	• The Release-Reuse Equivalency Principle (REP) • The Common Closure Principle (CCP) • The Common Reuse Principle (CRP)
	Coupling	• The Acyclic Dependencies Principle (ADP) • The Stable Dependencies Principle (SDP) • The Stable Abstractions Principle (SAP)

Table 1. A collection of object-oriented design principles along with their scope of applicability.

6.1. CHARACTERISTICS OF OODP

The **name** of an OODP is **illustrative** of its **meaning**.

OODP are **paradigm-dependent**, but **implementation language-independent**.

It should become evident from later discussion that the OODP mentioned in Table 1 are **not** necessarily **mutually exclusive**. Indeed, in some cases, they can **assist** each other.

6.2. SITUATING OODP WITH RESPECT TO OTHER FORMS OF EXPERIENTIAL KNOWLEDGE

There are different forms of experiential knowledge for engineering software. It is therefore useful to see how OODP relates to these other forms of experiential knowledge.

6.2.1. SITUATING OODP WITH RESPECT TO OTHER PRINCIPLES

It could be said that **high-level decisions govern low-level decisions** at any stage of a (software) project.

The OODP (are expected to) conform to other principles, such as **enterprise architecture principles** [Lippert, Roock, 2006, Section 3.1; Greefhorst, Proper, 2011] and **software engineering principles** [Ghezzi, Jazayeri, Mandrioli, 2003], at a relatively **higher level**, as shown in Tables 2 and 3.

Enterprise Architecture Principle
Software Engineering Principle
Design Principle
Object-Oriented Design Principle

Table 2. A hierarchy of principles, from general to specific, relevant to software design.

Only in Response to Business Needs are Changes to IT Systems Made
Anticipation of Change
Design for Change
Dependency Inversion Principle

Table 3. An example of hierarchy of principles, from general to specific, relevant to software design.

6.2.2. SITUATING OODP WITH RESPECT TO ARCHITECTURAL TACTICS

Definition [Tactic] [Bass, Clements, Kazman, 2003]. A **design decision** that influences the control of a **quality attribute** response.

There are certain architectural tactics that are related to OODP. For example, **Restrict Dependencies, Use an Intermediary, Hide Information, and Maintain Existing Interface** are tactics [Sangwan, 2015, Section 4.4.3] that are related to **ADP and SDP**.

6.3. VIOLATION OF OODP DUE TO DECAY OF MACRO-ARCHITECTURE DESIGN PATTERNS

It has been said that software systems, like humans, **age** [**Parnas, 1994**].

Definition [Decay] [Parnas, 1994]. The **deterioration** of the structure of design of a software system.

A **design pattern** is a kind of **pre-defined design decision**. This motivates the following definitions:

Definition [Design Pattern Rot] [Izurieta, Bieman, 2013]. A kind of decay where there is a breakdown of the structural integrity of a design pattern realization.

Definition [Design Pattern Grime] [Izurieta, Bieman, 2013]. A kind of decay where there is an accumulation of unrelated artifacts in classes that play roles in a design pattern realization.

Rot

Grime

(Source: Google Images.)

The **difference** between rot and grime is that while **rot is subtractive** (breaks the structural integrity of the solution of a pattern), **grime is additive** (adds but does not break the structural integrity of the solution of a pattern).

There can be different kinds of grime: **class, modular, and organizational**.

Figure 1 summarizes the relationships among different kinds of design pattern-related decay. The design pattern rot and design pattern grime are **mutually exclusive**. There could be design pattern-related decay that is neither design pattern rot, nor design pattern grime, but that is an **open problem**.

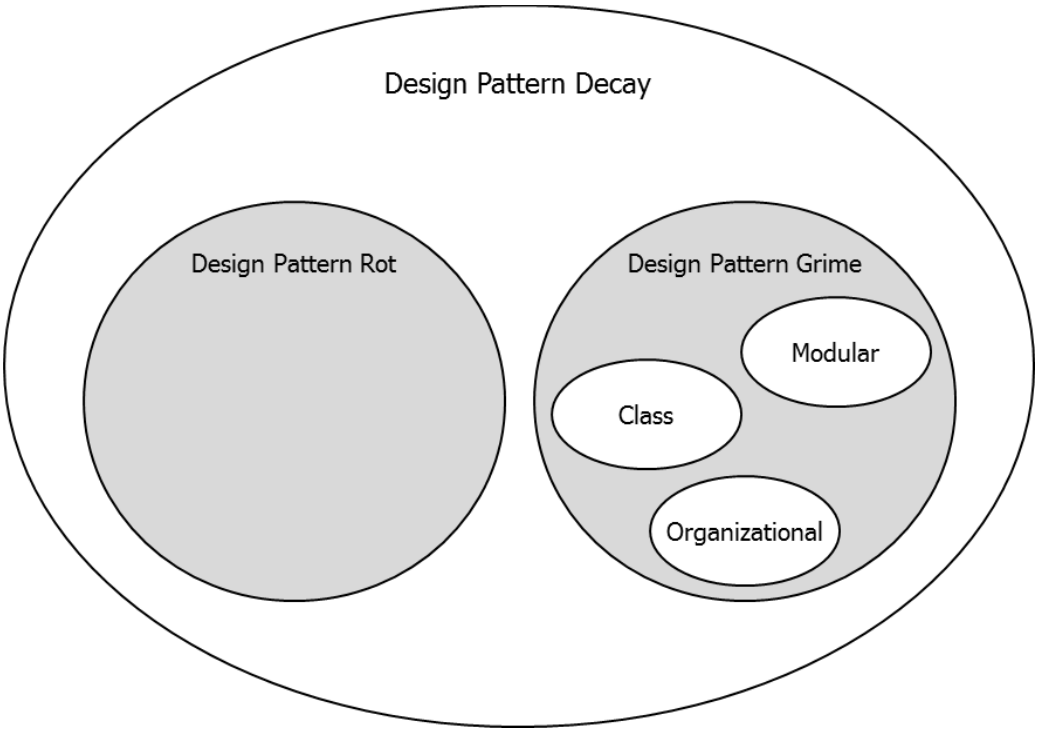

Figure 1. A **non-exhaustive** classification of design pattern-related decay.

Definition [Organizational Design Pattern Grime] [Izurieta, Bieman, 2013; Griffith, 2015]. The accumulation of design pattern grime due to the allocation of pattern classes to packages, namespaces, or modules within a software system.

The **violation** of OODP is among the causes of the **development and accumulation** of the **organizational grime**.

It is evident that both rot and grime are maintainability concerns. In particular, **grime** can **reduce adaptability and testability** of a software system [Izurieta, Bieman, 2013].

Remark. In [Schanz, 2011], based on empirical studies of certain open source software, **modular grime has been classified**.

Remark. In [Griffith, 2015], based on empirical studies of certain open source software, the **effects of design pattern grime** on **software product quality and technical debt** have been examined.

6.4. OODP IN PERSPECTIVE

The best designers sometimes disregard the principles of design. When they do so, however, there is usually some compensating merit attained at the cost of the violation. Unless you are certain of doing as well, it is best to abide by the principles.
— William Lidwell, Kritina Holden, Jill Butler

The OODP are **not 'laws'**. There are no (known) legal implications for their (non-)use.

The OODP are **not standards**, even though software engineering standards may be based on them or refer to them.

The OODP are **not 'absolute'** or 'unconditional' or 'universally applicable'. For example, a principle may be **context (situation) dependent**.

The default target for OODP is a **new design**. However, if (1) a design **'smell'** is discovered and (2) it is possible to **'refactor'**, then OODP, particularly those for classes, can also be used to improve the quality of an old (legacy) design.

It could be noted that **not** all OODP are used equally or frequently.

6.5. AN ORGANIZATION OF OODP COVERAGE

The following structure is used to organize the OODP discussed in the rest of the document:

Related Quality Attribute. The list of relevant quality attribute(s) that the OODP impacts in some manner is given.

Statement. The statement of OODP is given.

The statement of OODP is followed by an analysis of the OODP. This may include **observations, remarks, and examples**.

Related Principle. The list of relevant principle(s), as per Table 2, is given.

7. THE RELEASE-REUSE EQUIVALENCY PRINCIPLE (REP)

Related Quality Attribute. Reusability.

Statement. The package of reuse is the package of release.

Definition [Release] [ISO/IEC/IEEE, 2010].
1. [A] collection of new or changed configuration items that are tested and introduced into a live environment together.
2. [A] delivered version of an application which may include **all or part** of an application.

Observation. The term 'reuse' of any aspect of a software system, including a package, is a **commitment to others**. This is because a thing does not become reusable automatically; it requires **foresight** as well as **deliberate and considerate effort** to make that thing reusable. In other words, there are **social** and **technical** aspects of reusability.

7.1. SOCIAL ASPECTS OF REUSE

From a social perspective, the target **user** of the package and their **needs** should be clear. It can lead to waste of space and time if any one of these is not clear.

7.2. TECHNICAL ASPECTS OF REUSE

From a technical perspective, it is evident that the term **'release'** in REP is being used only as a **metaphor**: it mimics **commercial-off-the-self (COTS)** software.

REP essentially suggests that a package destined to be released for reuse should have the same characteristics (namely, release **identification and tracking**) as of a complete software system. In other words, such a package should be like a **standalone mini system** to be used on its own.

By indirect reference, REP also provides hints on partitioning of classes, that is, **reusable packages should contain reusable classes**. In fact, a package designed for reuse should **not contain any classes not designed for reuse**. Therefore, the conclusion is that there is **no 'partial reuse'**; **either all the classes in a package are reusable, or none of them are**. In short, the package aims for **homogeneity**.

7.3. PROPERTIES OF REUSE

Let P be a package. Then, according to REP, P can be **effectively reused** if it has the following properties:

- **Completeness.** P is **complete** in the sense that it has properly defined interfaces and is supported by documentation.

- **Maintenance.** P is equipped with a **guarantee** that it will be maintained for a specific amount of time.

- **Backward Compatibility.** P is equipped with a **guarantee** that in the event of any **future changes** to its external interfaces or internal functionality (1) a **notification** will be provided, (2) there will be an **option** for rejecting the new version, and (3) there will be **support** of use of the old version for a specified period of time.

Related Software Engineering Principle. Separation of Concerns.

8. THE COMMON CLOSURE PRINCIPLE (CCP)

When it is not necessary to change,
It is necessary not to change.
— Lord Falkland

I suspect that some programmers think that their program will be so good that it won't have to be changed. This is foolish. The only programs that don't get changed are those that are so bad that nobody wants to use them. Designing for change is designing for success.
— David L. Parnas

"Come Together"
— The Beatles, Abbey Road (1969)

Related Quality Attribute. Modifiability.

Statement. The classes in a package should be closed together against the same kinds of changes.

Observation [Acknowledgement]. CCP is based on the assumption that designs cannot be completely temporally static; some **volatility is necessary** if the design is to be maintained. CCP is therefore about **change-driven-reason package management**.

Observation [Formalization].

The following propositions aim to explain the meaning of "closure".

Proposition 1. Let $C = \{C_1, \ldots, C_n\}$ be a set of classes. Let F and H be two different kinds of changes. Let P and Q be two different packages. (The goal is to assign elements of C to P and to Q, accordingly.) Let F affect C_i and let F affect C_j, for some $i \leq n$ and $j \leq n$. Let H affect C_s and let H affect C_t, for some $s \leq n$ and $t \leq n$. Then, according to CCP, C_i and C_j belong to P, and C_s and C_t belong to Q.

Proposition 2 is an **extension** of Proposition 1.

Proposition 2. Let $C = \{C_1, \ldots, C_n\}$ be a set of classes. Let F, G, and H be three different kinds of changes. Let P and Q be two different packages. (The goal is to assign elements of C to P and to Q, accordingly.) Let F and G affect C_i and let F and G affect C_j, for some $i \leq n$ and $j \leq n$. Let H affect C_s and let H affect C_t, for some $s \leq n$ and $t \leq n$. Then, according to CCP, C_i and C_j belong to P, and C_s and C_t belong to Q.

Observation [Seclusion]. CCP suggests that a change that affects a closed package affects all the classes in that package and no other packages.

This, with the aid of Proposition 1, is illustrated in Figure 2.

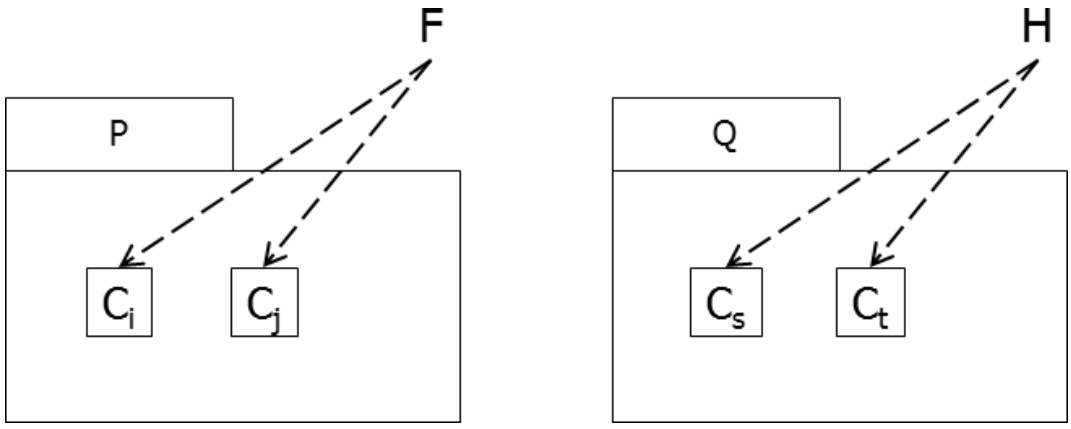

Figure 2. F has impact on all classes in P. H has impact on all classes in Q. F has no impact on classes in Q, and H has no impact on classes in P.

Remark. CCP suggests that the change is a relation between classes, and leads to **equivalence classes of packages**. This is similar to isolating each package and placing it in **quarantine**.

Observation [Change Localization]. CCP is about **'localizing' the change** to a **minimal number of packages**. Instead of being broadly distributed, CCP suggests assembling, in one place, **all** the classes that are susceptible to change for the **same reasons**. This leads to a **change-reason-driven** organization of classes.

For example, two **classes** that are strongly related (conceptually or otherwise) in such a manner that they **always change together** belong in the **same package**.

Observation [The SRP Connection]. CCP is the **Single Responsibility Principle (SRP)** restated for packages. CCP says that a package should **not** have **multiple reasons** to change.

Related Software Engineering Principle. Anticipation of Change.
Related Design Principle. Design for Change, Modularity (Decrease Coupling).
Related OODP. SRP, OCP.

9. THE COMMON REUSE PRINCIPLE (CRP)

Related Quality Attribute. Reusability.

Statement. The classes in a package are reused together.

Observation. CRP is based on the assumption that **reusable classes** often **collaborate** to **solve a larger problem**, and therefore have a strong dependency on each other.

CRP suggests that **classes that tend to be reused together belong in the same package**.

Example [Matrix Computations]. A package for finding the **LU decomposition of square matrices** should have classes for **Gaussian elimination** (or another numerical algorithm with the same purpose), **inverse of a matrix**, **determinant of a matrix**, and **multiplication of matrices**.

CRP suggests that if **one** of the classes in a package is reused, then **all** the classes in the package must be reused.

By **indirect reference**, CRP also suggests that a distribution of reusable classes into multiple different packages (that is, a **violation of CRP**) will **increase coupling** between packages.

Related Software Engineering Principle. Anticipation of Change.
Related Design Principle. Design for Change, Modularity (Decrease Coupling).

10. THE ACYCLIC DEPENDENCIES PRINCIPLE (ADP)

Related Quality Attribute. Buildability (Constructability), Testability.

Statement. A package dependency graph should not allow any cycles.

Observation.

"Houston, we have a problem."
— Jim Lovell, Apollo 13 (1995)

The **history** of ADP can be traced back to the **1970s** [Parnas, 1979].

10.1. RELATIONSHIPS AMONG PACKAGES

There can be (at least) two types of salient relationships among packages [Mancinelli, Boender, Cosmo, Vouillon, 2006]:

1. **Dependency Relationship:** In case of a dependency, one package needs another to work properly. Furthermore, syntactically, a dependency has a direction.

2. **Conflict Relationship:** In case of a conflict, the packages involved cannot work together. (It is evident that a conflict does not have a direction.)

10.2. UNDERSTANDING ORIGINS OF DEPENDENCIES AMONG PACKAGES

If a class contains relationships to other classes, then corresponding packages containing those classes also must contain relationships to other packages.

Example. Figure 3 shows a UML Class Diagram depicting two packages, P_1 and P_2. P_1 contains class **`Client`** and B contains class **`Server`**. If class **`Client`** references in any way class **`Server`**, then **`Client`** has a structural relationship to class **`Server`**. This implies that any changes to the **`Server`** class may impact **`Client`**.

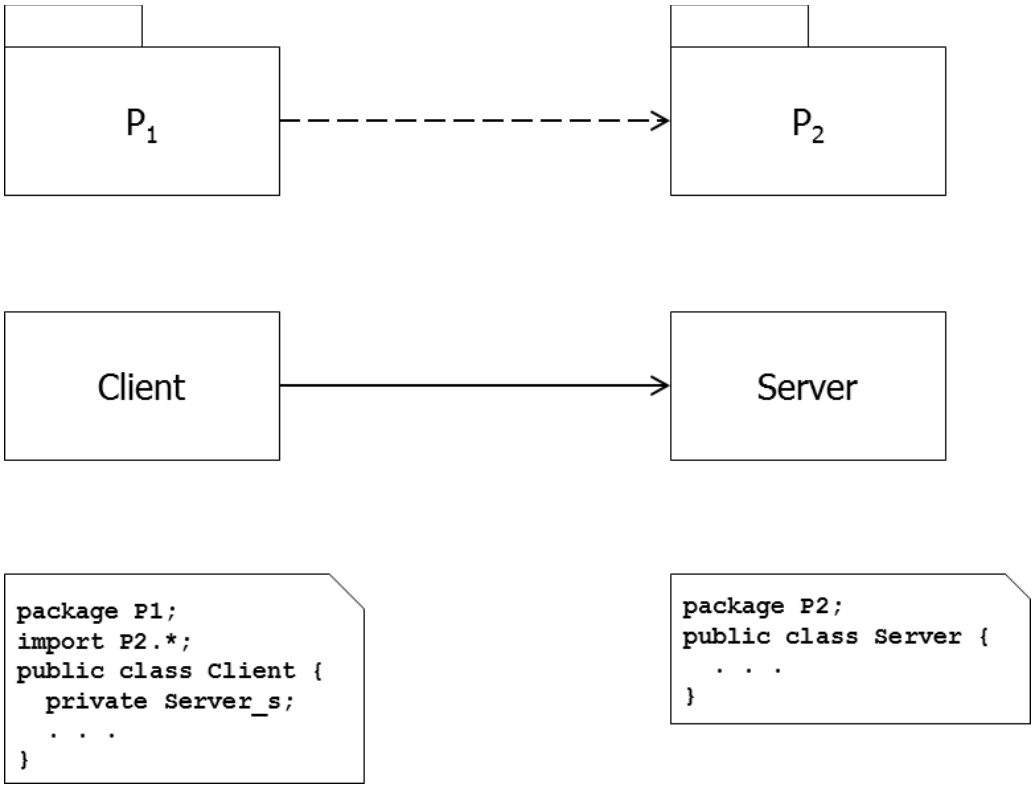

Figure 3. A collection of package and corresponding relationships among classes.

If the contents of package P_1 are dependent on the contents of package P_2, then P_1 has a dependency on P_2, and if the contents of P_2 change, then this impact may be noticeable in P_1. Therefore:

If changing the contents of a package P_1 may impact the contents of another package P_2, then P_1 has a package dependency on P_2.

A package may not only contain classes but also other packages. A nested package may or may not be visible. Therefore, there can be two kinds of visibility:

1. **Opaque Visibility:** A dependency on a package with nested packages does not imply access to these nested packages.

2. **Transparent Visibility:** A dependency on a package with nested packages carries with it implicit dependencies.

UML and Java have different positions regarding the visibility of nested packages. In UML, the question of how to deal with nested packages is left to the development team.

In Java, importing the contents of a package implies that one has access only to the classes within that package and do not have access to classes in any nested packages.

10.3. TYPES OF DEPENDENCIES AMONG PACKAGES

There are two different types of dependencies:

1. **Conjunctive Dependency:** This is a dependency which can only be satisfied in one way.

2. **Disjunctive Dependency:** This is a dependency which may be satisfied by any one of a list of packages.

Example. If one wants to install the package A, then one needs to install B (a conjunctive dependency) and C or D (a disjunctive dependency).

10.4. DEPENDENCY GRAPH

A **dependency graph** is a model of software project artifacts and their relationships.

A **package dependency graph** is a **dependency graph** in which the software project artifacts are packages (in some OOD).

ADP aims to solve the following **problem**:

> Let A_1 be the author of a package P_1 that is in a working state at time T_1.
> Let A_2 be the author of a package P_2.
>
> P_1 depends on package P_2.
>
> A change to P_2 by A_2 at time $T_2 > T_1$ can lead to P_1 not working at time T_2.

Example. In many **software-intensive organizations**, **periodic builds** are common. Indeed, **"Release Early, Release Often"** is one of the **open source software development philosophies**, and **"Deliver Working Software Frequently"** is one of the **agile software development principles** as per the Agile Manifesto.

T_i: Day. In case of daily build, the aforementioned problem is known as the **'morning-after syndrome'**.

A **lengthening** of the **build schedule** (such as daily build to weekly build) does not **fundamentally** solve the problem.

T_i: Week. In case of weekly build, it becomes increasingly **difficult to integrate and test**, and there is a **loss of rapid feedback**. In other words, moving from daily build to weekly build **increases project risks**. For example, one such risk could be **waste of resources** by **implementing undesired functionality** during lapsed time (week, in this case).

The solution to this problem is to do the following:

- The development environment is **partitioned** into **releasable packages**.
- There is a **single 'author' group** (one or more persons) who is **responsible** for each package, at any given time. (This is to ensure the existence of a dedicated **communication channel** in case there are problems.)
- If a package is working, a copy of it is released (with a release identification, such as a **version control number**) in a **public area** (such as an organization's network) to be used by other authors.
- The evolution to the package can continue in a **private area**.
- The process is repeated with further releases. Therefore, **none** of the groups depend on the others.

However, the previous solution is based on the **assumption** that the **packages are indeed releasable**. For that, according to ADP, their dependency graph should **not** have any cycles [Lippert, Roock, 2006, Section 3.4.1]. Indeed, the presence of a cycle is an **architectural smell** [Babar, Brown, Mistrik, 2014, Section 3.6].

In general, ADP enables **maintenance** [Spinellis, 2006, Section 7.1]. This is because, for any package, its **dependencies** can be followed down to the **leaves** to see the effect of a change. Furthermore, **building the system** involves **compiling the packages from the leaves, up to the trunk**. (This is an example of using the top-down approach and the bottom-up approach to design, **together**.)

The packages in an OOD are **not** isolated from each other[4]. Indeed, their structure forms a **dependency graph**.

ADP suggests that the dependency graph of packages in an OOD **must** form a **directed acyclic graph (DAG)**. Let A, B, and C be three packages in an OOD. Then, according to ADP, if A depends on B and B depends on C, then C **should not depend** upon A.

Figure 4 provides an abstract example that does not conform to ADP, and an abstract example that removes the cycle and reinstates the dependency graph by 'refactoring' the design.

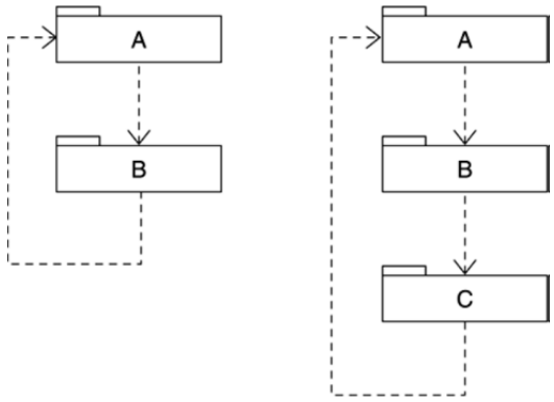

Figure 4(a). A collection of packages that do not conform to ADP. (Source: [Knoernschild, 2002, Chapter 1].)

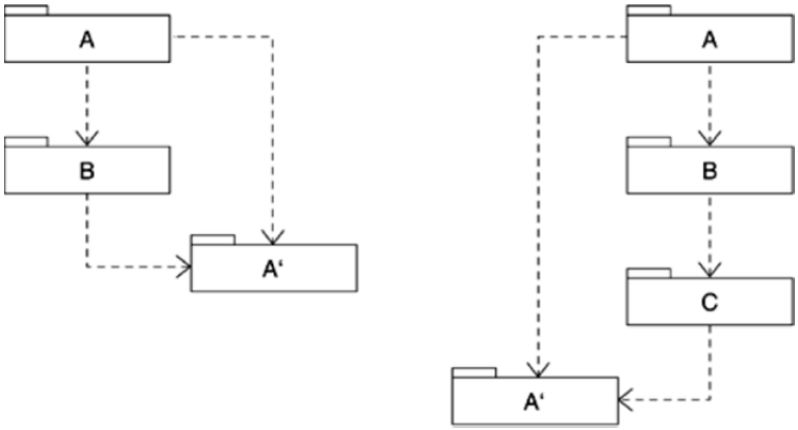

Figure 4(b). A collection of packages that conform to ADP. (Source: [Knoernschild, 2002, Chapter 1].)

[4] The classes in a package often have dependencies on other classes across package 'boundaries'. In these cases, the packages will have dependencies with each other.

Figure 5 provides a concrete example that does, and a concrete example that does not, conform to ADP.

For example, in Figure 3(a), if the **`MyDialogs`** package is released, it impacts **only** the dependent packages, namely **`MyTasks`** and **`MyApplication`**.

Therefore, the impact of the change is **local**. It also allows a **bottom-up release** of the software system.

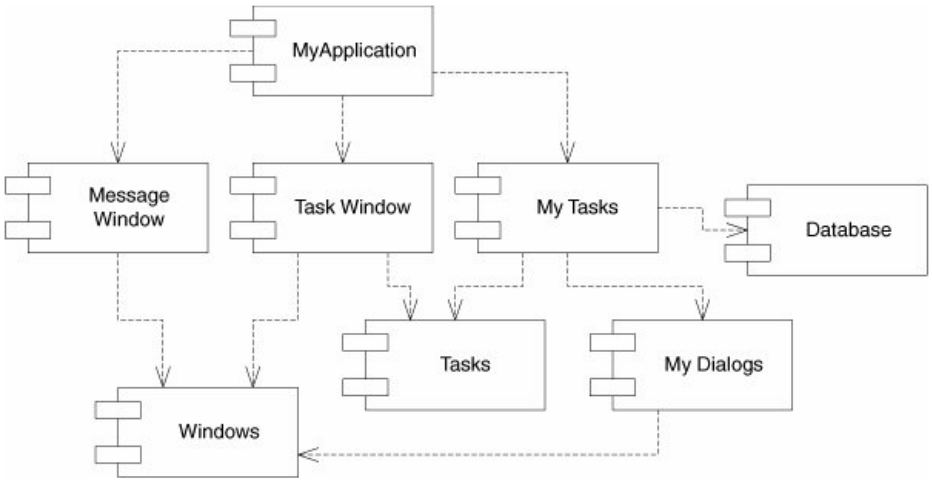

Figure 5(a). A collection of packages that conform to ADP.

However, Figure 5(b) shows one of the **worst** kinds of dependencies possible.

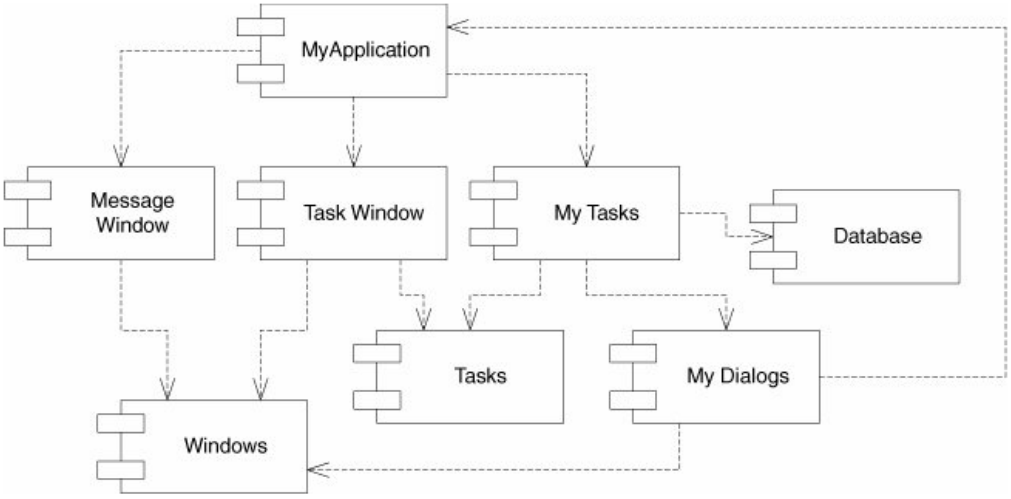

Figure 5(b). A collection of packages that has a cycle and therefore violates ADP.

The **cycle** is the **culprit** for an **effective release**. It leads to the following problems:

- The cycle, for example, makes **`MyTasks`** package difficult to release: its release impacts **all** the packages in the software system. (This can affect the **compile time adversely**.)

- The cycle forces **`MyApplication`**, **`MyTasks`**, and **`MyDialogs`** to always be released at the **same** time.

- The cycle makes **`MyTasks`** difficult to **(unit) test**: its test requires building the entire software system. (In this case, the **'unit' is a package**.)

It is possible to **remove the cycle** in Figure 5(b) and reinstate the dependency graph by **'refactoring'** the design. There are different ways of doing that. Figure 5(c) illustrates **one** way of doing that.

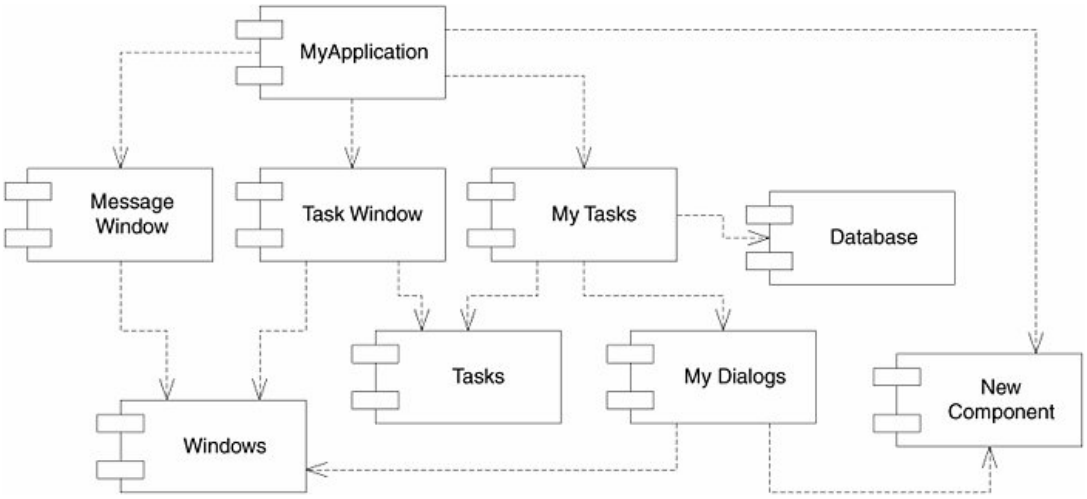

Figure 5(c). A collection of packages in which a cycle is removed by an addition of a new package, namely **`New Component`**.

10.5. ADP IN THE REAL WORLD

Example. Apache Velocity is a template engine in Java, and aims to provide a separation between model and view, so as to satisfy the philosophy of the **MODEL-VIEW-CONTROLLER (MVC) pattern** for (Enterprise) Web Applications.

Figure 6 shows a cycle in **Apache Velocity, Version 1.6.2** [Oyetoyan, 2015]. In this case, a cyclic dependency is formed because components depend on one another in a **circular manner**. The cycle is caused by the VISITOR pattern that involves the abstract visitor, the abstract element, and the concrete element. This relationship covers both direct and indirect connection between the components involved.

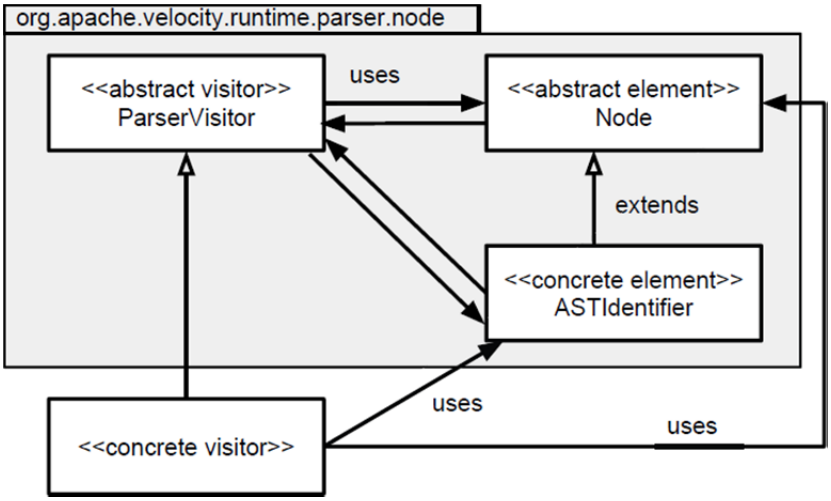

Figure 6. A cycle in Apache Velocity 1.6.2. (Source: [Oyetoyan, 2015].)

Example. The **Sub-Type Knowledge (STK)** is a **design anti-pattern** [Riel, 1996]. STK occurs in a cycle that contains at least one inheritance (extends) or realization (implements) edge, and a back reference path connecting the target of the edge to its source. This type of anti-pattern violates the principle that super-types should not know anything about their sub-types. An instance of the solution of STK is a cycle.

Figure 7 shows a STK cycle in the **Java Runtime Environment (JRE), Version 1.7.0** [Oyetoyan, 2015; Oyetoyan, Falleri, Dietrich, Jezek, 2015]. This is a **class-level cycle** that also induces a **package-level cycle** between `java.awt` and `javax.swing`. (Moreover, the superclass **cannot be understood or used** without its subclass, which violates one of the design heuristics: superclass should not "know" its subclasses.)

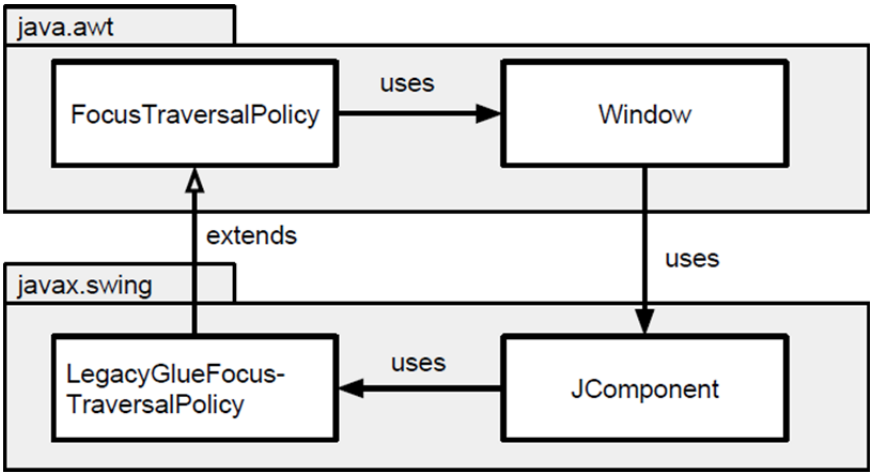

Figure 7. An STK cycle in an implementation of Java. (Source: [Oyetoyan, Falleri, Dietrich, Jezek, 2015].)

Remark. A cycle may be **desirable or undesirable** [Laval, Falleri, Vismara, Ducasse, 2012]. In general, a **package-level cycle** is undesirable; a **class-level cycle** may or may not be undesirable.

Example. JFace[5] is the main widget library used in the **IBM Eclipse Integrated Development Environment (IDE)**.

The package `jface.text` is dedicated to the **text widgets**, and provides classes such as `TextViewer`. The package `jface.text.hyperlink` is dedicated to the **management of textual hyperlinks**.

In JFace, there is a **cycle** between `jface.text` and `jface.text.hyperlink`. This cycle is **desirable** [Laval, Falleri, Vismara, Ducasse, 2012]. The `TextViewer` class is able to display texts containing hyperlinks, and therefore `jface.text` depends on `jface.text.hyperlink`. The package `jface.text.hyperlink` uses a several

[5] URL: http://wiki.eclipse.org/index.php/JFace .

classes and interfaces defined in `jface.text`. (For instance, a hyperlink is able to trigger text events and therefore depends on the `TextEvent class`, which is defined in the `jface.text` package.) Therefore, `jface.text.hyperlink` depends on `jface.text`. The nature of the hyperlink motivates its isolation in a package of its own, namely `jface.text.hyperlink`. Yet, it is not necessary to break the cycle with `jface.text` as it would not make sense to release one without the other.

Remark. ADP is the basis for the **Tangle** metric[6], the degree to which package dependencies are cyclic. Tangle and other metrics are part of products from **Structure101**.

Related Software Engineering Principle. Anticipation of Change.
Related Design Principle. Design for Change, Modularity (Decrease Coupling).
Related OODP. DIP.

[6] URL: http://structure101.com/products/structure101/measure-complexity.php .

11. THE STABLE DEPENDENCIES PRINCIPLE (SDP)

Related Quality Attribute. Modifiability, Stability.

Statement. The dependencies should be in the direction of stability.

Example. Figures 8(a) and 8(b) illustrate the **notion of stability and instability**, respectively.

Figure 8(a) shows a package X. There are three packages that depend on X, and therefore X has three reasons not to change. Since X does not depend on anything, it has no external influence to make it change. Therefore, **X is stable**.

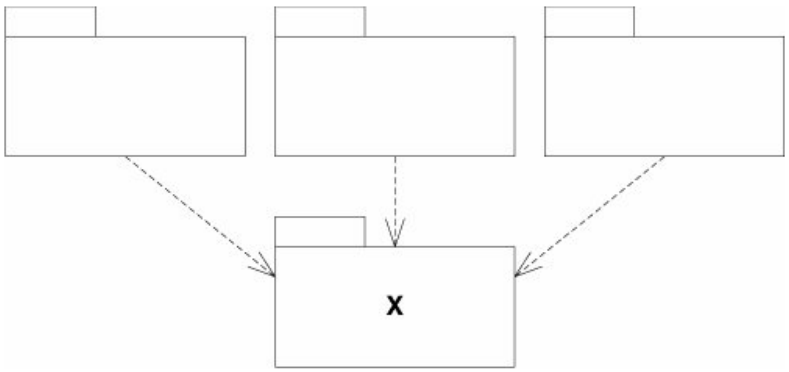

Figure 8(a). A collection of packages that depend in the direction of stability.

Figure 8(b) shows a package Y. Y depends on three packages, and therefore changes may come from at least three external sources. Therefore, **Y is instable**.

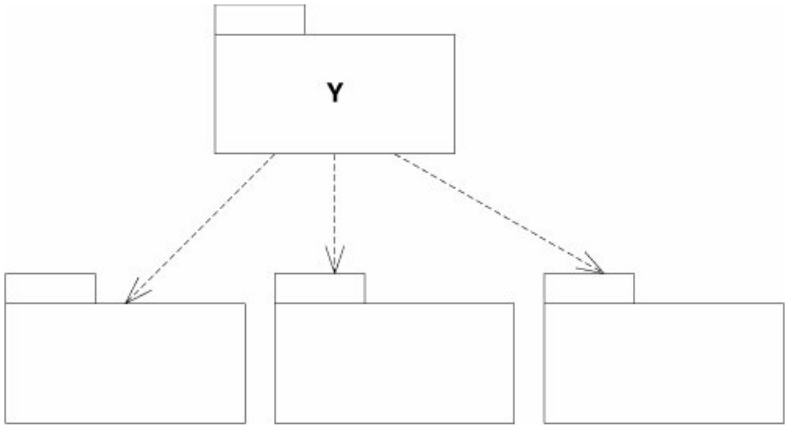

Figure 8(b). A package that depends in the direction of instability.

As an **analogy** from **classical mechanics**, Figure 9(a) could represent a **potential well** (a region surrounding a **local minimum** of **potential energy**), and Figure 9(b) could represent a **potential hill** (a region surrounding a **local maximum** of **potential energy**). This phenomenon is shown in Figure 9.

Figure 9. The motion of a sphere on a potential well and on a potential hill, respectively.

Example. Let A, B, and C be three packages in an OOD. Then, according to SDP, if A depends on B and B depends on C, then C should be the **most stable** among the three packages.

Observation. A conformance to CCP allows creating packages that are sensitive to certain kinds of changes. The packages that are designed to be **volatile** are **expected to change**, and so should be **easy to change**. (For example, a volatile package could be one that is based on a new, not necessarily mature, user interface theory.)

Therefore, any package that is expected to be volatile should **not** be depended upon by a package that is difficult to change. If not, the volatile package will also be difficult to change.

Observation. Let **`A`**, **`B`**, and **`C`** be three packages in an OOD such that **`A`** depends on **`B`** and **`B`** depends on **`C`**. Then, according to SDP,

$$\texttt{deg(C)} \geq \texttt{deg(B)} \geq \texttt{deg(A)},$$

where **`deg`** is the **degree of difficulty of changing a package**.

In other words, SDP suggests that packages that are intended to be easy to change are **not** depended upon by packages that are more difficult to change than they are.

Example. In [Spinellis, 2006, Section 7.1], SDP has been applied to **JUnit** and **Java Management Extensions (JME)**. It is concluded that there are more stable dependencies in JUnit than in JME.

Related Software Engineering Principle. Anticipation of Change.
Related Design Principle. Design for Change, Modularity (Decrease Coupling).
Related OODP. CCP.

12. THE STABLE ABSTRACTIONS PRINCIPLE (SAP)

Related Quality Attribute. Modifiability, Stability.

Statement. A package should be as abstract as it is stable.

Observation. SAP provides a connection between **stability and abstractness**.

SAP suggests that a **stable package should also be abstract** (consist of abstract classes) so that it can be extended.

SAP also suggests that an **instable package should be concrete** (consist of concrete classes), since its instability allows the concrete source code in it to be changed easily.

Example. Let **`A`**, **`B`**, and **`C`** be three packages in an OOD such that

`C ---|> B ---|> A`

and

`A > B > C`,

where **`---|>`** denotes the generalization relationship and **`X > Y`** denotes that **`X`** is more stable than **`Y`**. Then, according to SAP, **`A`** should be more abstract than **`B`** and **`C`**.

Observation. SAP + SDP = DIP (for packages), where DIP is the abbreviation for the Dependency Inversion Principle.

Observation [Similarity]. According to SDP, **dependencies should run in the direction of stability** and, according to SAP, **stability implies abstraction**. Therefore, **dependencies run in the direction of abstraction**.

Observation [Difference]. There is a **difference** between DIP (for classes) and DIP (for packages). A class can be either abstract or not, and so **abstractness** is **dichotomous**. However, a package can be **partially** abstract, and so **abstractness** is on a **spectrum**.

Related Software Engineering Principle. Anticipation of Change.
Related Design Principle. Design for Change, Modularity (Decrease Coupling).
Related OODP. DIP, SDP.

ACKNOWLEDGEMENT

The inclusion of images from external sources is only for non-commercial educational purposes, and their use is hereby acknowledged.

REFERENCES

[Abate, Cosmo, Treinen, Zacchiroli, 2013] A Modular Package Manager Architecture. By P. Abate, R. Di Cosmo, R. Treinen, S. Zacchiroli. Information and Software Technology. Volume 55. Issue 2. 2013. Pages 459-474.

[Almugrin, 2015] Definitions and Validations of Metrics of Indirect Package Coupling in an Agile, Object-Oriented Environment. By S. A. Almugrin. Ph.D. Thesis. Kent State University. Kent, U.S.A. 2015.

[Babar, Brown, Mistrik, 2014] Agile Software Architecture: Aligning Agile Processes and Software Architectures. By M. A. Babar, A. W. Brown, I. Mistrik (Editors). Morgan Kaufmann. 2014.

[Bass, Clements, Kazman, 2003] Software Architecture in Practice. By L. Bass, P. Clements, R. Kazman. Second Edition. Addison-Wesley. 2003.

[De Marco, 1979] Structured Analysis and Systems Specification. By T. De Marco. Prentice-Hall. 1979.

[Dooley, 2011] Software Development and Professional Practice. By J. Dooley. Apress. 2011.

[Garland, Anthony, 2003] Large-Scale Software Architecture: A Practical Guide using UML. By J. Garland, R. Anthony. John Wiley and Sons. 2003.

[Ghezzi, Jazayeri, Mandrioli, 2003] Fundamentals of Software Engineering. By C. Ghezzi, M. Jazayeri, D. Mandrioli. Second Edition. Prentice-Hall. 2003.

[Greefhorst, Proper, 2011] Architecture Principles: The Cornerstones of Enterprise Architecture. By D. Greefhorst, E. Proper. Springer-Verlag. 2011.

[Griffith, 2015] Design Pattern Decay: A Study of Design Pattern Grime and its Impact on Quality and Technical Debt. By I. D. Griffith. Ph.D. Thesis Proposal. Montana State University. Bozeman, U.S.A. 2015.

[ISO/IEC/IEEE, 2010] ISO/IEC/IEEE 24765:2010. Systems and Software Engineering -- Vocabulary. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC)/The Institute of Electrical and Electronics Engineers (IEEE) Computer Society. 2010.

[Izurieta, Bieman, 2013] A Multiple Case Study of Design Pattern Decay, Grime, and Rot in Evolving Software Systems. By C. Izurieta, J. M. Bieman. Software Quality Journal. Volume 21. Number 2. 2013. Pages 289-323.

[Knoernschild, 2002] Java Design: Objects, UML, and Process. By K. Knoernschild. Addison-Wesley. 2002.

[Laval, Falleri, Vismara, Ducasse, 2012] Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems. By J. Laval, J.-R. Falleri, P. Vismara, S. Ducasse. Journal of Object Technology. Volume 11. Number 1. 2012. 1-24.

[Lieberherr, Holland, 1989] Assuring Good Style for Object-Oriented Programs. By K. Lieberherr, I. M. Holland. IEEE Software. 1989. Volume 6. Issue 5. Pages 38-48.

[Lippert, Roock, 2006] Refactoring in Large Software Projects. By M. Lippert, S. Roock. John Wiley and Sons. 2006.

[Mancinelli, Boender, Cosmo, Vouillon, 2006] Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. By F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon. The Twenty First IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). Tokyo, Japan. September 18-22, 2006.

[Martin, Martin, 2006] Agile Principles, Patterns, and Practices in C#. By R. C. Martin, M. Martin. Prentice-Hall. 2006.

[Meyer, 1997] Object-Oriented Software Construction. By B. Meyer. Second Edition. Prentice-Hall. 1997.

[Oyetoyan, 2015] Dependency Cycles in Software Systems: Quality Issues and Opportunities for Refactoring. By T. D. Oyetoyan. Ph.D. Thesis. Norwegian University of Science and Technology. Trondheim, Norway. 2015.

[Oyetoyan, Falleri, Dietrich, Jezek, 2015] Circular Dependencies and Change-Proneness: An Empirical Study. By T. D. Oyetoyan, J.-R. Falleri, J. Dietrich, K. Jezek. The Twenty Second IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015). Montreal, Canada. March 2-6, 2015.

[Parnas, 1972] On the Criteria To Be Used in Decomposing Systems into Modules. By D. L. Parnas. Communications of the ACM. Volume 15. Number 12. 1972. Pages 1053-1058.

[Parnas, 1979] Designing Software for Ease of Extension and Contraction. By D. L. Parnas. IEEE Transactions on Software Engineering. Volume 5. Number 2. 1979. Pages 128-138.

[Parnas, 1994] Software Aging. By D. L. Parnas. The Sixteenth International Conference on Software Engineering (ICSE 1994). Sorrento, Italy. May 16-21, 1994.

[Sangwan, 2015] Software and Systems Architecture in Action. By R. S. Sangwan. CRC Press. 2015.

[Schanz, 2011] A Taxonomy of Modular Grime in Design Patterns. By T. S. Schanz. M.Sc. Thesis. Montana State University. Bozeman, U.S.A. 2011.

[Spinellis, 2006] Code Quality: The Open Source Perspective. By D. Spinellis. Addison-Wesley. 2006.

[Wirfs-Brock, McKean, 2003] Object Design: Roles, Responsibilities, and Collaborations. By R. Wirfs-Brock, A. McKean. Addison-Wesley. 2003.

[Wirth, 1995] A Plea for Lean Software. By N. Wirth. IEEE Computer. Volume 28. Number 2. 1995. Pages 64-68.