Six Principles of Package Design

1. Reuse–Release Equivalence Principle
2. Common-Reuse Principle
3. Common-Closure Principle
4. Acyclic-Dependencies Principle
5. Stable-Dependencies Principle
6. Stable-Abstractions Principle

REP: The Reuse–Release Equivalence Principle

- Anything we reuse must also be released and tracked
- Package author should guarantee:
  - maintenance
  - notifications on future changes
  - option for a user to refuse any new versions
  - support for old versions for a time

REP (cont’d)

- Primary political issues
  - software must be partitioned so that humans find it convenient
- Reusable package must contain reusable classes
  - either all the classes in a package are reusable or none of them are
- Reusable by the same audience

CRP: The Common-Reuse Principle

- The classes in a package are reused together.
  - If you reuse one of the classes in a package, you reuse them all.

CRP (cont’d)

- If one class in a package uses another package, there is a dependency between the packages
  - whenever the used package is released, the using package must be revalidated and re-released
  - when you depend on a package, you depend on every class in that package!
- Classes that are tightly bound with class relationships should be in the same package
  - these classes typically have tight coupling
  - example: container class and its iterators
- The classes in the same package should be inseparable – impossible to reuse one without another

CCP: The Common-Closure Principle

- The classes in a package should be closed together against the same kind of changes.
  - A change that affects a closed package affects all the classes in that package and no other packages.
CCP (cont’d)

- SRP restated for packages
  - A package should not have multiple reasons to change
- Maintainability often more important than reusability
  - Changes should occur all in one package
  - Minimizes workload related to releasing, revalidating and redistributing
- Closely related to OCP
  - Strategic closure: close against types of changes that are probable
  - CCP guides to group together classes that are open to the same type of change

ADP: The Acyclic-Dependencies Principle

- Allow no cycles in the package dependency graph.
- Without cycles it is easy to compile, test and release 'bottom-up' when building the whole software.
- The packages in a cycle will become de facto a single package, testing becomes difficult since a complete build is needed to test a single package.
- Developers can step over one another since they must be using exactly the same release of each other's packages.

The 'Morning-After Syndrome'

- Developers are modifying the same source files trying to make it work with the latest changes somebody else did → no stable version
- Solution #1: the weekly build
  - Developers work alone most of the week and integrate on Friday
  - Works on medium-sized projects
  - For bigger projects, the iteration gets longer (monthly build?) → rapid feedback is lost
- Solution #2: Partition the development environment into releasable packages
  - Ensure ADP

Release-Control

- Partition the development environment into releasable packages
  - Package = unit of work
  - Developer modifies the package privately
  - Developer releases the working package
  - Everyone else uses the released package while the developer can continue modifying it privately for the next release
- No developer is at the mercy of the others
  - Everyone works independently on their own packages
  - Everyone can decide independently when to adapt the packages to new releases of packages they use
  - No 'big bang' integration but small increments
- To avoid the 'morning-after syndrome' the dependency tree must not have any cycles

Package Structure as a Directed Acyclic Graph

Breaking the Cycle with DIP
Breaking the Cycle with a New Package

MyApplication

Message Window

Task Window

MyTasks

Database

Tasks

MyDialogs

Windows

NewPackage

Breaking the Cycle – a Corollary

- The package structure cannot be designed top–down but it evolves as the system grows and changes
- Package dependency diagrams are not about the function of the application but they are a map to the buildability of the application

SDP: The Stable-Dependencies Principle

Depend in the direction of stability.

- Designs cannot be completely static
  - some volatility is required so that the design can be maintained
  - some packages are sensitive to certain types of changes
- A volatile package should not be depended on by a package that is difficult to change
  - a package designed to be easy to change can (accidentally) become hard to change by someone else hanging a dependency on it!

Stable and Instable Packages

- Package dependency diagrams are not about the function of the application but they are a map to the buildability of the application

Stability Metrics

- Affarent couplings $C_a$
  - the number of classes outside this package that depend on classes within this package
- Effarent couplings $C_e$
  - the number of classes inside this package that depend on classes outside this package

Instability $I$

- $I = C_e / (C_a + C_e)$
- $I = 0$: maximally stable package
- $I = 1$: maximally instable package

Dependencies

- C++: #include
- Java: import, qualified names

Instability $I$

- The $I$ metric of a package should be larger than the $I$ metrics of the packages that depends on

SDP

- 'Stable' = not easy to change
  - how much effort is needed to change a package: size, complexity, clarity, incoming dependencies
- If other packages depend on a package, it is hard to change (i.e. stable)
Fixing the Stability Violation Using DIP

Flexible

Stable

Flexible

Stable

SAP: The Stable-Abstractions Principle

A package should be abstract so that it is stable.

- A stable package should be abstract so that stability does not prevent it from being extended.
- An unstable package should be concrete since the instability allows the concrete code to be changed easily.
- SDP + SAP = DIP for packages.
- Since packages have varying degrees of abstractness, we need a metric to measure the abstractness of a package.

Measuring Abstractness

- The number of classes in the package $N_c$.
- The number of abstract classes in the package $N_a$.
  - Abstract class = at least one pure interface and cannot be instantiated.
- Abstractness $A$
  - $A = N_a / N_c$.
  - $A = 0$: no abstract classes.
  - $A = 1$: only abstract classes.

The Abstractness-Instability Graph

- The main sequence of pain and uselessness.

Package Cohesion and Coupling

- REP, CRP, and CCP: cohesion within a package.
  - “bottom-up” view of partitioning.
  - Classes in a package must have a good reason to be there.
  - Classes belong together according to some criteria.
  - Political factors.
  - Dependencies between the packages.
  - Package responsibilities.
- ADP, SDP, and SAP: coupling between packages.
  - Dependencies across package boundaries.
  - Relationships between packages.
  - Technical.
  - Political.
  - Volatile.

FACTORY

- DIP: prefer dependencies on abstract classes.
  - Avoid dependencies on concrete (and volatile!) classes.
  - Any line of code that uses the new keyword violates DIP:
    ```java
    Circle c = new Circle(origin, 1);
    ```
  - The more likely a concrete class is to change, the more likely depending on it will lead to trouble.
- How to create instances of concrete objects while depending only on abstract interfaces → FACTORY.
Example: Creating Shapes Violates DIP

```
<creates>
Application

<interface>
Shape

Square Circle
```

Example: Shapes Using FACTORY

```
<interface>
Shape

+makeSquare()
+makeCircle()

<interface>
ShapeFactory

Application

ShapeFactory Implementation

Square Circle
```

Example: Removing the Dependency Cycle

```
public interface ShapeFactory {
    public Shape make(Class<? extends Shape> t);
}

public class ShapeFactoryImplementation implements ShapeFactory {
    public Shape make(Class<? extends Shape> t) {
        if (t == Circle.class) return new Circle();
        else if (t == Square.class) return new Square();
        throw new Error();
    }
}

ShapeFactory sf = new ShapeFactoryImplementation();
Shape s1 = sf.make(Circle.class);
Shape s2 = sf.make(Square.class);
```

Benefits of FACTORY

- Implementations can be substituted easily
- Allows testing by spoofing the actual implementation

FACTORY – the Flip Side

- Factory is a powerful abstraction
  - Strictly thinking DIP entails that you should use factories for every volatile class
- Do not start out using factories
  - Can cause unnecessary complexity
  - Add them when the need becomes great enough

Reading for the Next Week

- Section 5: The Weather Station Case Study
  - Chapter 23: COMPOSITE
  - Chapter 24: OBSERVER – Backing into a Pattern
  - Chapter 25: ABSTRACT SERVER, ADAPTER, and BRIDGE
  - Chapter 26: PROXY and STAIRWAY TO HEAVEN: Managing Third Party APIs
  - Chapter 27: Case Study: Weather Station