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

Cohesion (1–3)

Coupling (4–6)

REP: The Reuse–Release Equivalence Principle

THE GRANULE OF REUSE IS THE GRANULE OF RELEASE.

- Anything we reuse must also be released and tracked
- Package author should guarantee
  - maintanance
  - 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 reason to change
- Maintainability often more important than reusability
  - changes should occur all in one package
  - minimizes workload related 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
  - compile-times increase
  - 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

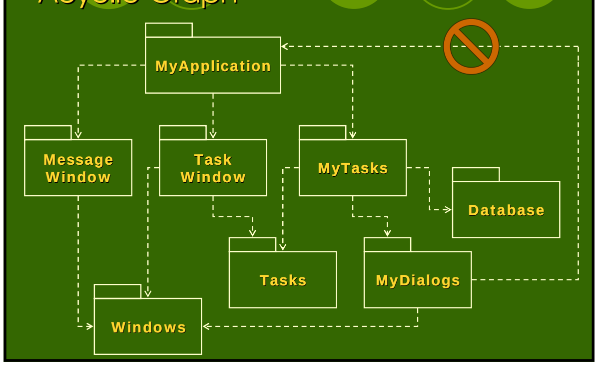
The 'Morning-After Syndrome'

- 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

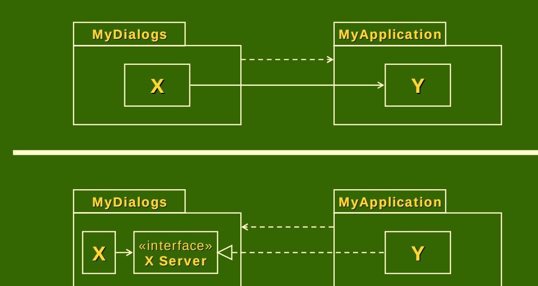
Release-Control

- 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 the packages they use
  - no 'big bang' integration but small increments
- To avoid the 'morning-after syndrome' the dependency tree must not have any cycles

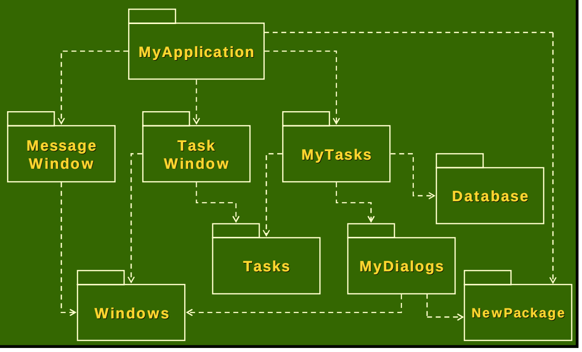
Package Structure as a Directed Acyclic Graph



Breaking the Cycle with DIP



Breaking the Cycle with a New Package



Breaking the Cycle – a Corollary

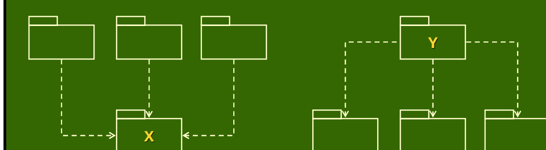
- The package structure cannot be designed top–down but it evolves as the system grows and changes
- Package depency diagrams are not about the function of the application but they are a map to the *buildability* of the application

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
  - CCP: 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



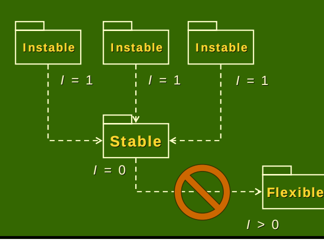
- 'Stable' = not easy to change
  - how much effort is needed to change a package: size, complexity, clarity, incoming dependencies
- If other packages depend on a package, it is hard to change (i.e. stable)

Stability Metrics

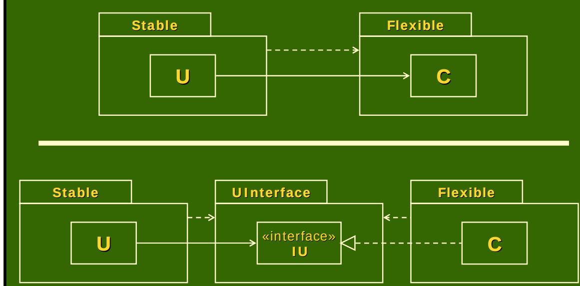
- Affarent couplings C_a
  - the number of classes outside this package that depend on classes within this package
- Efferent couplings C_e
  - the number of classes inside this package that depend on classes outside this package
- Instability I
  - $I = C_e / (C_a + C_e)$
  - $I = 0$: maximally stable package
  - $I = 1$: maximally instable package
- Dependencies
  - C++: `#include`
  - Java: `import`, qualified names

SDP

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



Fixing the Stability Violation Using DIP



SAP: The Stable-Abstractions Principle

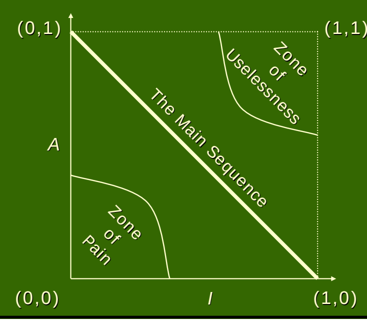
A PACKAGE SHOULD BE AS ABSTRACT AS IT IS STABLE.

- A stable package should be abstract so that stability does not prevent it from being extended
- An instable package should be concrete since the instability allows the concrete code to be changed easily
- SDP + SAP = DIP for packages
  - dependencies run in the direction of abstractions
- 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



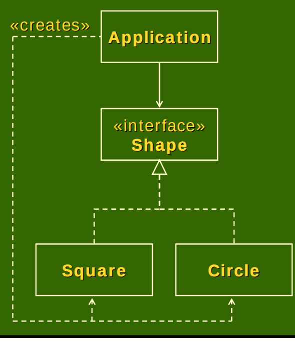
Package Cohesion and Coupling

- REP, CRP, and CCP: cohesion within a package
  - 'bottom–up' view of partitioning
  - classes in a packages 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 accross package boundaries
  - relationships between packages
    - technical
    - political
    - volatile

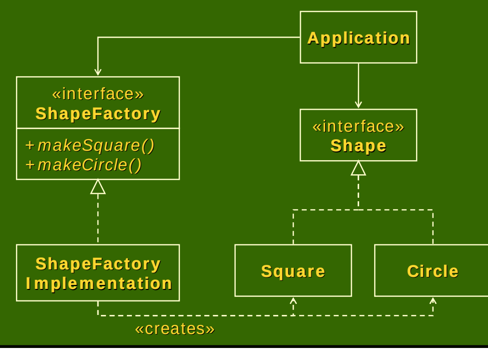
FACTORY

- DIP: prefer dependencies on abstract classes
  - avoid dependencies on concrete (and volatile!) classes
  - any line of code that uses the new keyword violates DIP:
    `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



Example: Shapes Using FACTORY



Example: Removing the Dependency Cycle

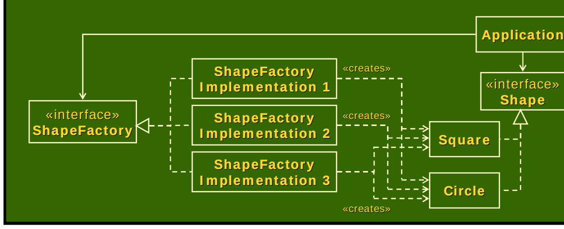
```
public interface ShapeFactory {
    public Shape make(Class<? extends Shape> t);
}

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