Ingineria Programării

Cursul 6 – 1 aprilie

OOD principles for classes

- The **S**ingle Responsibility Principle
- The **O**pen Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle

- SOLID

The Single Responsibility Principle

- *A class should have one, and only one, reason to change.*
- A responsibility to is “a reason for change.”
- Each responsibility is an axis of change.

```
interface Modem
  {
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
  }
```

The Single Responsibility Principle

- Should these responsibilities be separated?
  - If the implementations for the communication and connection management change independently, separately
  - If the implementations only change together, do not separate
- Corollary: An axis of change is only an axis of change if the changes actually occur.

The Open Closed Principle

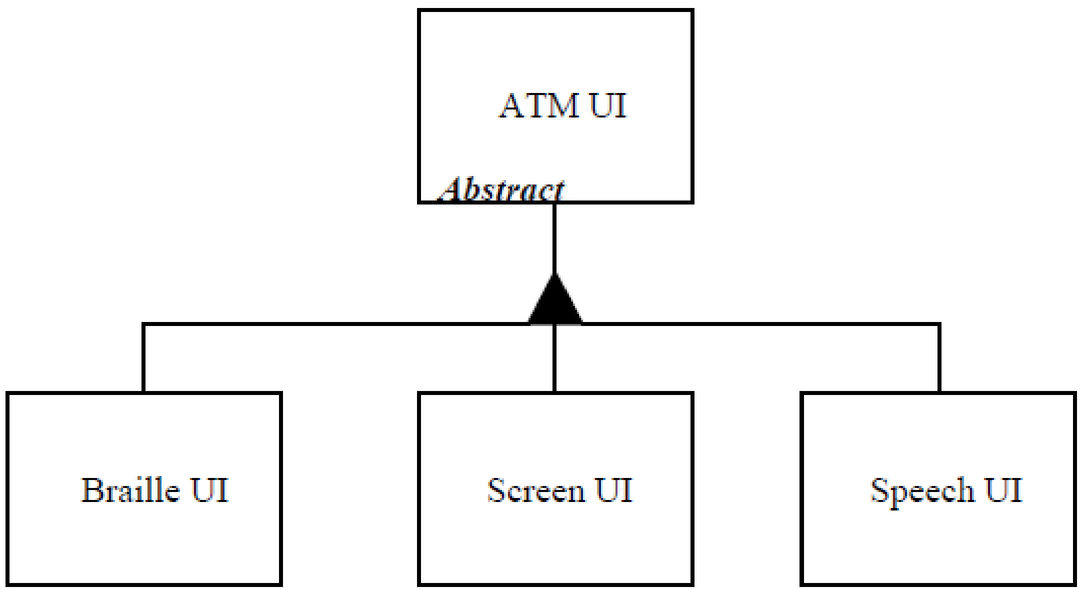
- *You should be able to extend a class' behavior without modifying it.*
- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- The primary mechanisms behind the Open-Closed principle are abstraction and polymorphism.

The Liskov Substitution Principle

- *Derived classes must be substitutable for their base classes.*
- Makes applications more maintainable, reusable and robust
- If there is a function which does not conform to the LSP, then that function uses a reference to a base class, but must know about all the derivatives of that base class.
- Such a function violates the Open–Closed principle

The Interface Segregation Principle

- *Clients should not be forced to depend upon interfaces that they do not use.*
- *Make fine grained interfaces that are client specific.*



The Dependency Inversion Principle

- What is it that makes a design bad?
  - most software eventually degrades to the point where someone will declare the design to be unsound
  - Because of the lack of a good working definition of “bad” design.

The Dependency Inversion Principle

- The Definition of a “Bad Design”
  - It is hard to change because every change affects too many other parts of the system. (Rigidity)
  - When you make a change, unexpected parts of the system break. (Fragility)
  - It is hard to reuse in another application because it cannot be separated from the current application. (Immobility)

The Dependency Inversion Principle

- A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
- B. Abstractions should not depend upon details. Details should depend upon abstractions.

Object-Oriented Design

- The most common types of programming are Structured Programming and Object Oriented Programming
- It has become difficult to write a program that does not have the external appearance of both structured programming and object oriented programming
  - Do not have **goto**
  - class based and do not support functions or variables that are not within a class
- Programs may look structured and object oriented, but looks can be decieving

OOD principles for deliverables

- Cohesion
  - The Release Reuse Equivalency Principle
  - The Common Closure Principle
  - The Common Reuse Principle

- Coupling
  - Acyclic Dependencies Principle
  - The Stable Dependencies Principle
  - The Stable Abstractions Principle

R – GRASP

- Principii, responsabilități
- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Controller

Granularity (Cohesion)

- As software applications grow in size and complexity they require some kind of high level organization.
- The class is too finely grained to be used as an organizational unit for large applications.
- Something "larger" than a class is needed => packages.

Designing with Packages

- What are the best partitioning criteria?
- What are the relationships that exist between packages, and what design principles govern their use?
- Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?
- How are packages physically represented? In the programming language? In the development environment?
- Once created, how will we use these packages?

The Reuse/Release Equivalence Principle

- Code copying vs. code reuse
- I reuse code if, and only if, I never need to look at the source code. The author is responsible for maintenance
  - I am the customer
  - When the libraries that I am reusing are changed by the author, I need to be notified
  - I may decide to use the old version of the library for a time
  - I will need the author to make regular releases of the library
  - I can reuse nothing that is not also released

The Reuse/Release Equivalence Principle

- The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package.

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.*
- Which classes should be placed into a package?
    - Classes that tend to be reused together belong in the same package.
- Packages to have physical representations that need to be distributed.

The Common Reuse Principle

- I want to make sure that when I depend upon a package, I depend upon every class in that package or I am wasting effort.

The Common Closure Principle

- The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.
- If two classes are so tightly bound, either physically or conceptually, such that they almost always change together; then they belong in the same package.

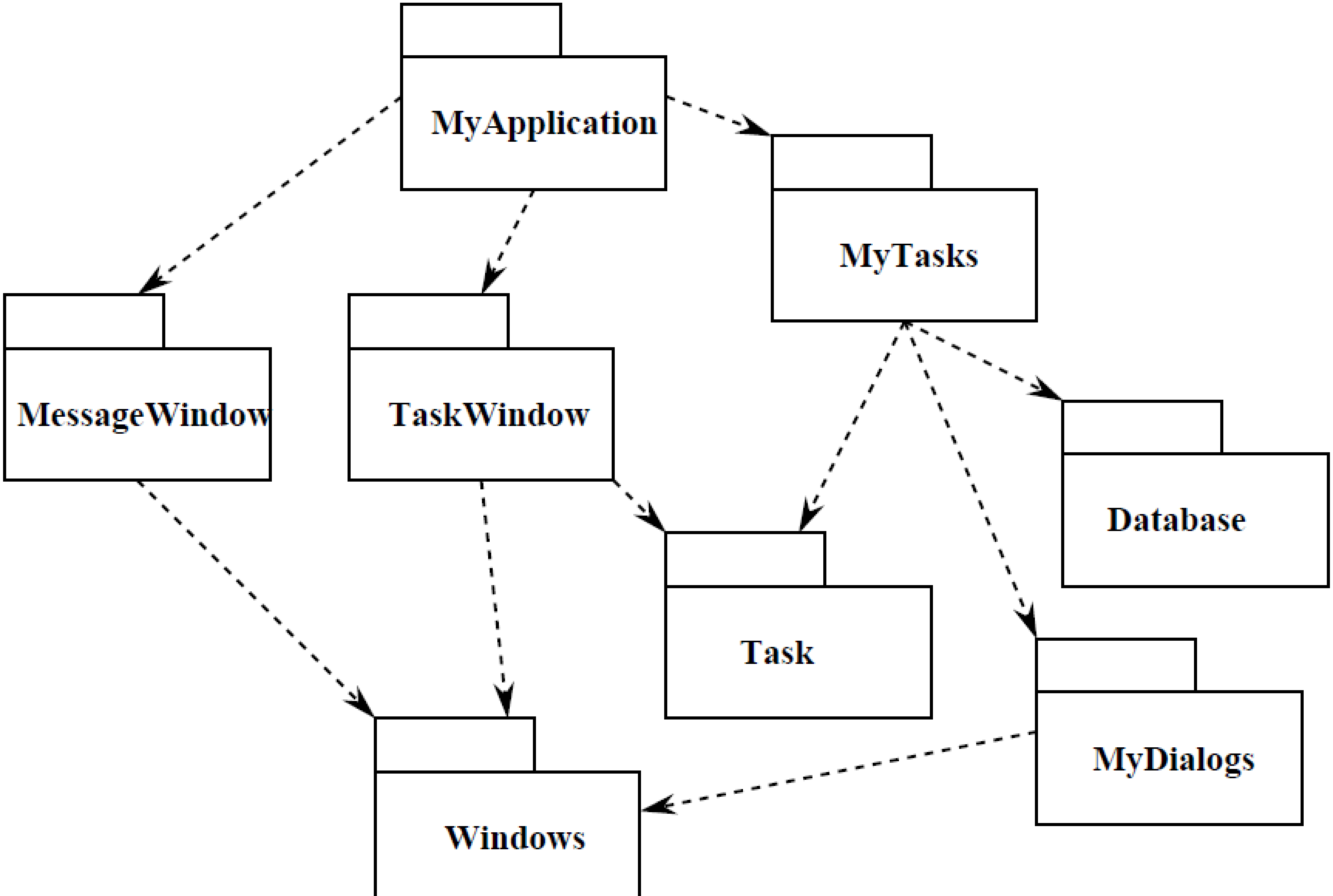
The Acyclic Dependencies Principle

- The morning after syndrome: you make stuff work and then gone home; next morning it longer works? Why? Because somebody **stayed later than you**!
- Many developers are modifying the same source files.
- Partition the development environment into releasable packages
- You must *manage the dependency structure of the packages*

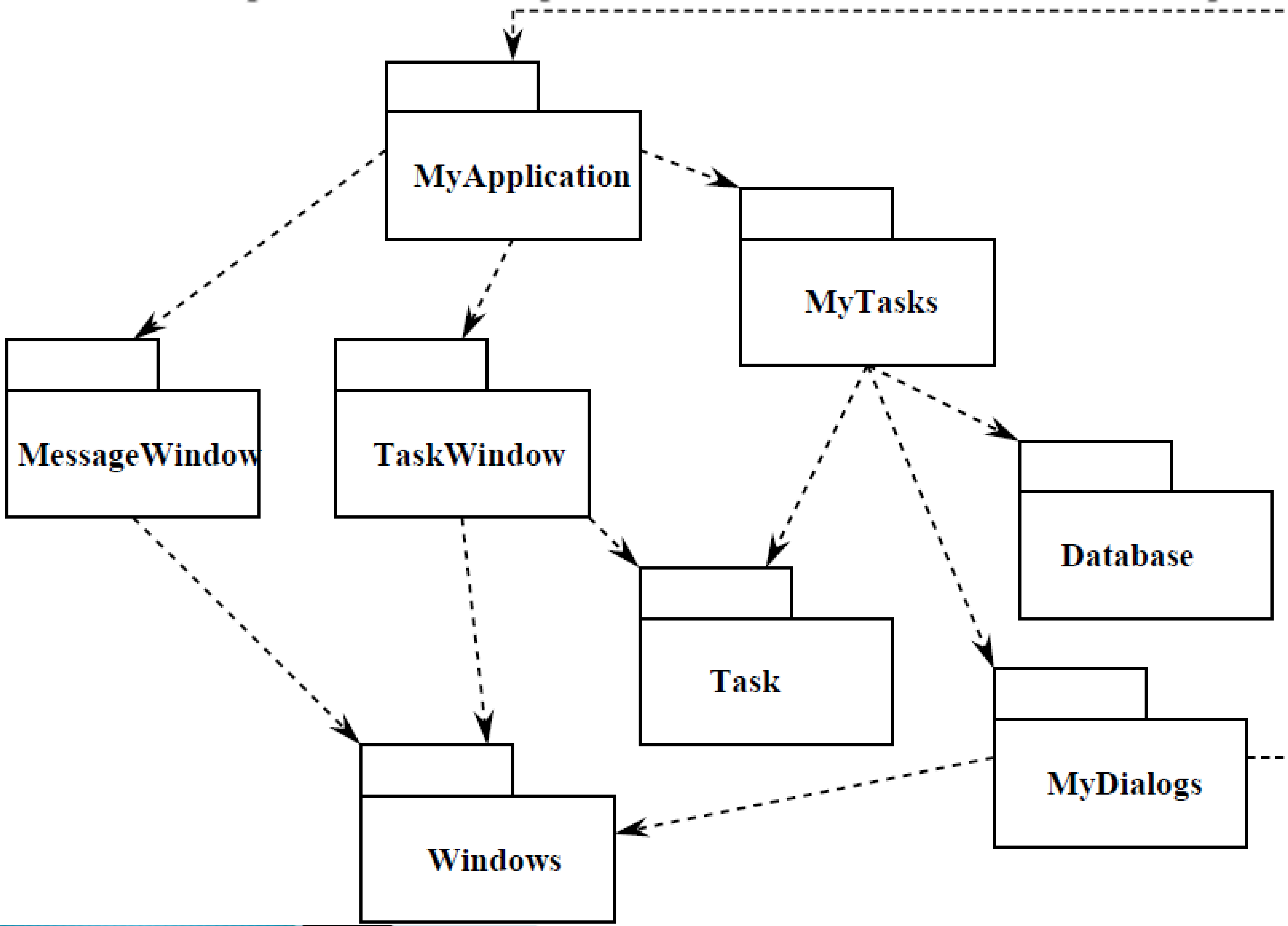
The Acyclic Dependencies Principle

- The dependency structure between packages must be a directed acyclic graph (DAG). That is, there must be no cycles in the dependency structure.

The Acyclic Dependencies Principle



The Acyclic Dependencies Principle



The Acyclic Dependencies Principle

- Breaking the Cycle
  - Apply the Dependency Inversion Principle (DIP). Create an abstract base class
  - Create a new package that both MyDialogs and MyApplication depend upon. Move the class(es) that they both depend upon into that new package.

- The package structure cannot be designed from the top down.

Stability

- Not easily moved
- A measure of the difficulty in changing a module
- Stability can be achieved through
  - Independence
  - Responsibility
- The most stable classes are Independent and Responsible. They have no reason to change, and lots of reasons not to change.

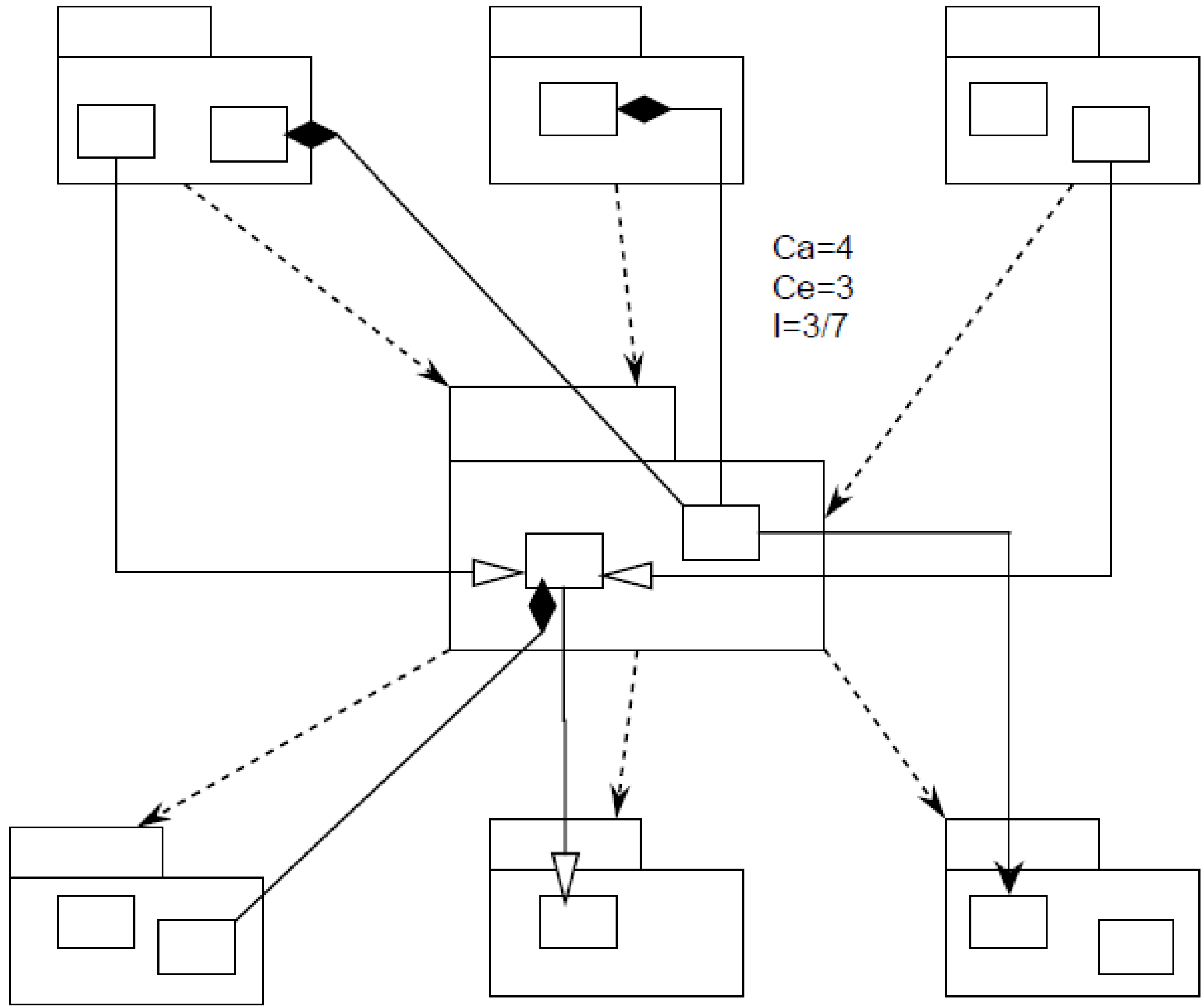
The Stable Dependencies Principle

- The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is.
- We ensure that modules that are designed to be unstable are not depended upon by modules that are more stable

Stability Metrics

- **Ca** : Afferent Couplings : The number of classes outside this package that depend upon classes within this package.
- **Ce**: Efferent Couplings : The number of classes inside this package that depend upon classes outside this package.
- **I** : Instability : (Ce/(Ca+Ce)) I=0 maximally stable package. I=1 maximally instable package.

Stability Metrics



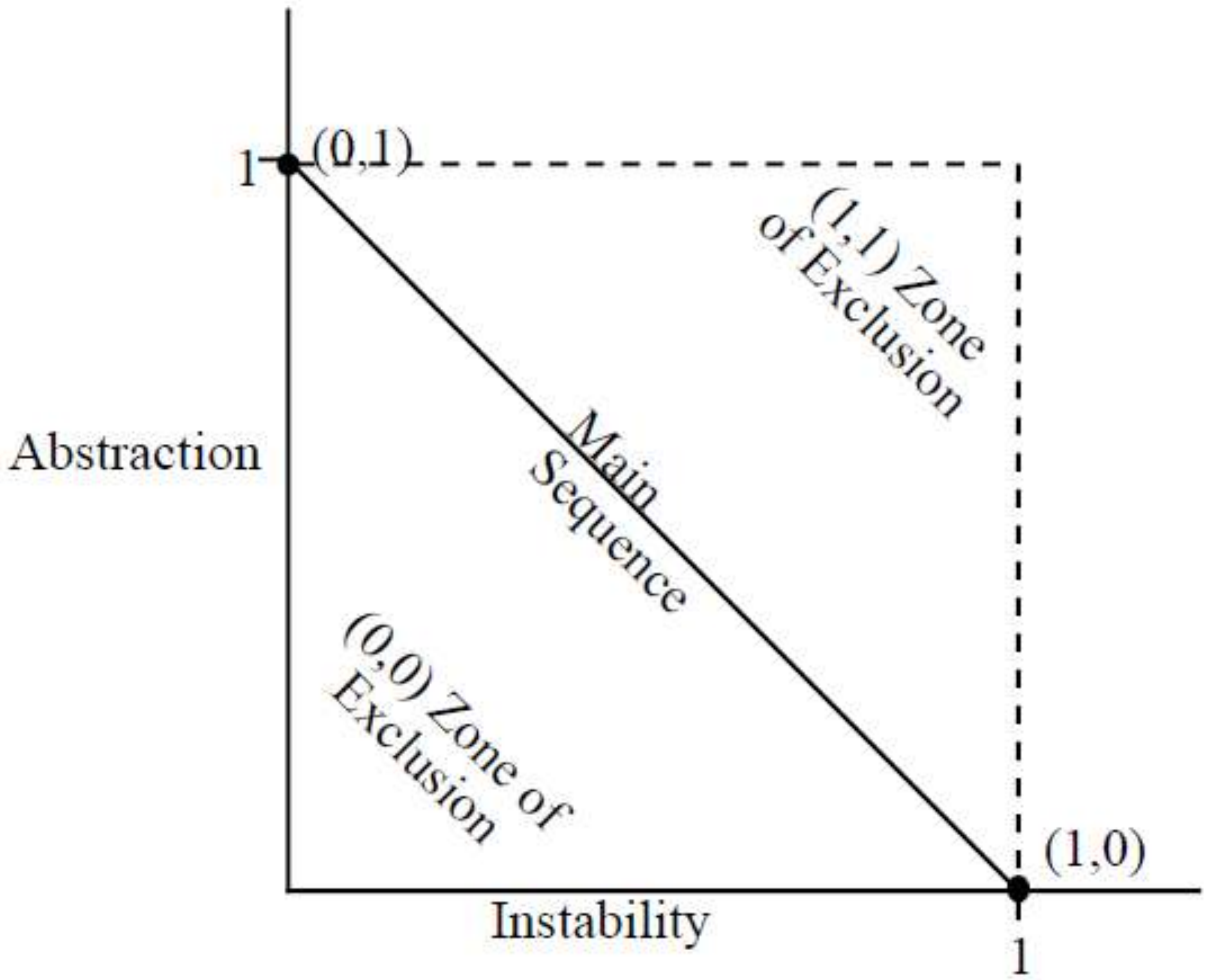
The Stable Dependencies Principle

- Not all packages should be stable
- The software the encapsulates the high level design model of the system should be placed into stable packages
- How can a package which is maximally stable (I=0) be flexible enough to withstand change?
  - classes that are flexible enough to be extended without requiring modification => abstract classes

The Stable Abstractions Principle

- Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstraction of a package should be in proportion to its stability.
- Abstraction (A) is the measure of abstractness in a package. $A = AC/TC$

Main Sequence



Design Patterns – Why?

- **If a problem occurs over and over again**, a solution to that problem has been used effectively (solution = pattern)

- When you make a design, you **should know the names of some common solutions**. Learning design patterns is good for people to **communicate each other effectively**

Design Patterns – Definitions

- “Design patterns capture solutions that have developed and evolved over time” (GOF – ***Gang-Of-Four*** (because of the four authors who wrote it), *Design Patterns: Elements of Reusable Object-Oriented Software)*

- **In software engineering (or computer science), a design pattern is a general repeatable solution to a commonly occurring problem in software design**

- The **design patterns** are language-independent strategies for solving common object-oriented design problems

Gang of Four

- Initial was the name given to a leftist political faction composed of four Chinese Communist party officials
- The name of the book (“Design Patterns: Elements of Reusable Object-Oriented Software”) is too long for e-mail, so “book by the gang of four” became a shorthand name for it
- That got shortened to "**GOF book**“**.** Authors are: *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*
- The **design patterns** in their book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*

Design Patterns – Elements

1. Pattern name
2. Problem
3. Solution
4. Consequences

Design Patterns – Pattern name

- A handle **used to describe a design problem**, its solutions, and consequences in a word or two
- Naming a pattern immediately increases our **design vocabulary**. It lets us design at a higher level of abstraction
- Having a **vocabulary** for patterns lets us talk about them with our colleagues, in our documentation
- Finding good names has been one of the hardest parts of developing our catalog

Design Patterns – Problem

- Describes **when** to apply the pattern. It explains the problem and its **context**
- It might describe specific design problems such as how to represent **algorithms** as objects
- It might describe **class** or **object** structures that are symptomatic of an inflexible design
- Sometimes the problem will include a **list of conditions** that must be met before it makes sense to apply the pattern

Design Patterns – Solution

- Describes the elements that make up the **design, their relationships, responsibilities,** and **collaborations**
- The solution **doesn't describe a particular concrete design or implementation**, because a pattern is like a template that can be applied in many different situations
- Instead, the pattern provides an **abstract description of a design problem** and **how** a general arrangement of elements (classes and objects in our case) **solves it**

Design Patterns – Consequences

- Are the results and trade-offs of applying the pattern
- They are critical for **evaluating design alternatives** and for **understanding the costs and benefits** of applying the pattern
- The consequences for software often concern **space** and **time trade-offs**, they can address **language and implementation issues** as well
- Include its impact on a system's **flexibility, extensibility, or portability**
- Listing these consequences explicitly helps you **understand and evaluate** them

Example of (Micro) pattern

- **Pattern name**: Initialization
- **Problem**: It is important for some code sequence to be executed only once at the beginning of the execution of the program.
- **Solution**: The solution is to use a static variable that holds information on whether or not the code sequence has been executed.
- **Consequences**: The solution requires the language to have a static variable that can be allocated storage at the beginning of the execution, initialized prior to the execution and remain allocated until the program termination.

Describing Design Patterns 1

- **Pattern Name and Classification**
- **Intent** – the answer to question: *What does the design pattern do*?
- **Also Known As**
- **Motivation** – A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem
- **Applicability** – *What are the situations in which the design pattern can be applied*? *How can you recognize these situations*?
- **Related Patterns**

Describing Design Patterns 2

- **Structure** – A graphical representation of the classes in the pattern
- **Participants** – The classes and/or objects participating in the design pattern and their responsibilities
- **Collaborations** – How the participants collaborate to carry out their responsibilities
- **Consequences** – *How does the pattern support its objectives*?
- **Implementation** – *What techniques should you be aware of when implementing the pattern*?
- **Sample Code**
- **Known Uses** – Examples of the pattern found in real systems

Design Patterns – Classification

- **Creational patterns**
- **Structural patterns**
- **Behavioral patterns**
- NOT in GOF: Fundamental, Partitioning, GRASP, GUI, Organizational Coding, Optimization Coding, Robustness Coding, Testing, Transactions, Distributed Architecture, Distributed Computing, Temporal, Database, Concurrency patterns

Creational Patterns

- **Abstract Factory** groups object factories that have a common theme
- **Builder** constructs complex objects by separating construction and representation
- **Factory Method** creates objects without specifying the exact class to create
- **Prototype** creates objects by cloning an existing object
- **Singleton** restricts object creation for a class to only one instance
- Not in GOF book: Lazy initialization, Object pool, Multiton, Resource acquisition (is initialization)

Structural Patterns

- **Adapter** allows classes with incompatible interfaces to work together
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently
- **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behavior in an existing method of an object
- **Facade** provides a simplified interface to a large body of code
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity

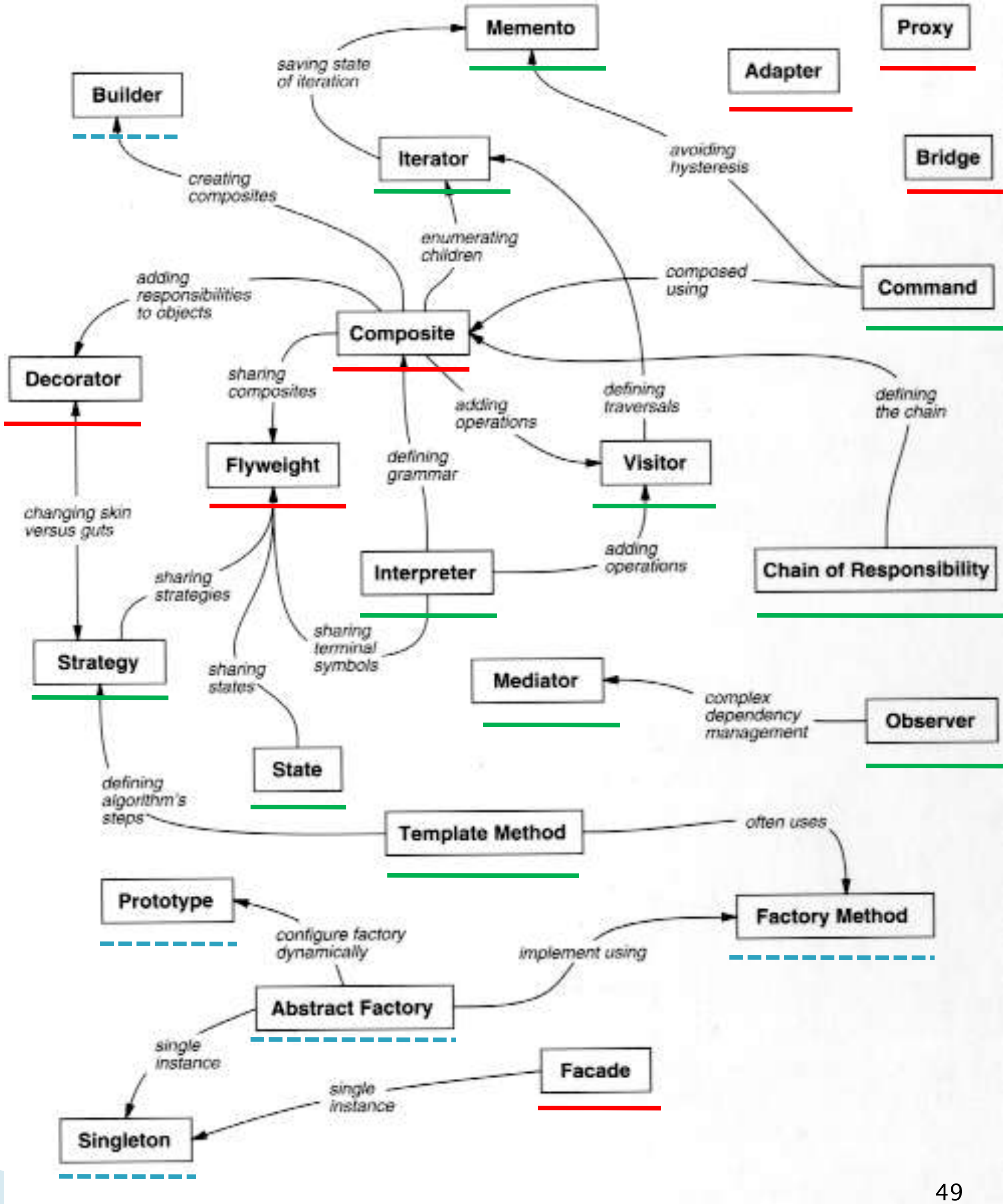
Behavioral patterns 1

- **Chain of responsibility** delegates commands to a chain of processing objects
- **Command** creates objects which encapsulate actions and parameters
- **Interpreter** implements a specialized language
- **Iterator** accesses the elements sequentially
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods
- **Memento** provides the ability to restore an object to its previous state

Behavioral patterns 2

- **Observer** allows to observer objects to see an event
- **State** allows an object to alter its behavior when its internal state changes
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime
- **Template** defines an algorithm as an abstract class, allowing its subclasses to provide concrete behavior
- **Visitor** separates an algorithm from an object structure
- Not in GOF book: Null Object, Specification

- Patterns
  - Creational
  - Structural
  - Behavioral



Design Pattern Relationships

How to Select a Design Pattern?

- With more than 20 design patterns to choose from, it might be hard to find the one that addresses a particular design problem

- Approaches to finding the design pattern that's right for your problem:
  1. *Consider how design patterns solve design problems*
  2. *Scan Intent sections*
  3. *Study relationships between patterns*
  4. *Study patterns of like purpose (comparison)*
  5. *Examine a cause of redesign*
  6. *Consider what should be variable in your design*

How to Use a Design Pattern?

1. *Read the pattern once through for an overview*
2. *Go back and study the Structure*, Participants, and Collaborations sections
3. *Look at the Sample Code section to see a concrete example*
4. *Choose names for pattern participants that are meaningful in the application context*
5. *Define the classes*
6. *Define application-specific names for operations in the pattern*
7. *Implement the operations to carry out the responsibilities and collaborations in the pattern*

Bibliography

- Robert C. Martin, Engineering Notebook columns for The C++ Report
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)