Ingineria Programării

Cursul 12 – 17-18 mai

Outline

- Recap – Software testing
- Non-functional software testing
  - Security
  - Performance
  - Usability
  - Internationalization
- SOLID Refactoring
  - Degradation of code
  - Designing with classes
  - Designing with packages

Recap

- Software quality assurance
- Functional software testing
  - Automatic testing
  - Manual testing

Non Functional Software Testing

- Verifies that the software functions properly even when it receives invalid or unexpected inputs
- Methods:
  - **Performance testing** or **Load Testing** checks to see if the software can handle large quantities of data or users (software scalability).
  - **Usability testing** checks if the user interface is easy to use and understand.
  - **Security testing** is essential for software which processes confidential data and to prevent system intrusion by hackers.
  - **Internationalization and localization** is needed to test these aspects of software, for which a pseudo localization method can be used.

Software Performance Testing

- **Types**
  - **load testing** – can be the expected concurrent number of users on the application (database is monitored)
  - **stress testing** – is used to break the application (2 x users, extreme load) (application's robustness)
  - **endurance testing** – if the application can sustain the continuous expected load (for memory leaks)
  - **spike testing** – spiking the number of users and understanding the behavior of the application whether it will go down or will it be able to handle dramatic changes in load

Performance Testing Analysis

Usability testing

- A technique used to evaluate a product by testing it on users
- Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose.
- **Examples** of products that commonly benefit from usability testing are web sites or web applications, computer interfaces, documents, or devices
- **Goals**
  - *Performance* – How much time, steps?
  - *Accuracy* – How many mistakes did people make?
  - *Recall* – How much does the person remember afterwards or after periods of non-use?
  - *Emotional response* – How does the person feel about the tasks completed? Is the person confident, stressed? Would the user recommend this system to a friend?

Usability testing steps

Security testing

- The Process to determine that an Information System protects data and maintains functionality as intended.
- The six basic security concepts that need to be covered by security testing are:
  - **Confidentiality**,
  - **Integrity** – information which it receives has not been altered in transit or by other than the originator of the information
  - **Authentication** – validity of a transmission, message, or originator,
  - **Authorization** – determining that a requester is allowed to receive a service or perform an operation,
  - **Availability** – Assuring information and communications services will be ready for use when expected,
  - **Non-repudiation** – prevent the later denial that an action happened, or a communication that took place

Security logo

Internationalization and localization

- Means of adapting computer software to different languages and regional differences
- **Internationalization** is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes.
- **Localization** is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

Measuring software testing

- Usually, quality is constrained to such topics as **correctness, completeness, security**
- Can also include capability, reliability, efficiency, portability, maintainability, compatibility, and usability
- There are a number of common software measures, often called "metrics", which are used to measure the state of the software or the adequacy of the testing.

Degradation of code

- Initial code is clean and well structured
- As it is developed, it degrades because of incremental changes
- Eventually, it becomes difficult to maintain, extend or debug
  - Rigid, Fragile, Immobile, Viscous
    - Design Principles and Design Patterns. Robert C. Martin

Degradation of code

- **Rigidity** – the tendency for software to be difficult to change, even in simple ways
- **Fragility** – the tendency of the software to break in many places every time it is changed
- **Immobility** – the inability to reuse software from other projects or from parts of the same project
- **Viscosity** – it is easy to do the wrong thing, but hard to do the right thing

Refactoring

- "… *the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*." – Martin Fowler in *Refactoring Improving The Design Of Existing Code*
- Making code cleaner, simpler and more elegant
- The refactored program must be functionally equivalent to the initial program

Refactoring

- Refactoring doesn't prevent changing functionality, it just says that it's a different activity from rearranging code
- It is easier to improve code quality if you don't also change its functionality
- Refactoring is not rewriting
  - refactoring does not change functionality
  - rewriting does change it

Why to Refactor?

- Increase cohesion and decrease coupling
- Refactored code is easier to maintain and extend
- Simpler to use design patterns
- Improves performance and usability of code

When is Refactoring Necessary?

- Duplicate Code
- Long Methods
- Large classes
- Long lists of parameters
- Instructions switch
- Speculative generality
- Intense communication between objects
- Chaining Message

How to Refactor

- Prepare a set of automatic tests for the program to be refactored
- Change the code in small iterations
  - Test for each iteration to ensure semantic equivalence
  - If any test fails, undo the change and try a different one
- After numerous iterations the cumulated changes will be large

Refactoring Techniques

- **Increase abstraction**
  - **Encapsulate Field** – force code to access the field with getter and setter methods
  - **Generalize Type** – create more general types to allow for more code sharing
  - **Replace type** – checking code with State/Strategy
  - **Replace conditional** with polymorphism
- Split code into more logical pieces
  - **Extract Method**, to turn part of a larger method into a new method.
  - **Extract Class** moves part of the code from an existing class into a new class.

Refactoring Techniques

- **Improving names and location of code**
  - **Move Method** or **Move Field** – move to a more appropriate Class or source file
  - **Rename Method** or **Rename Field** – changing the name into a more representative new one
  - **Pull Up** – in OOP, move to a superclass
  - **Push Down** – in OOP, move to a subclass

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 Liskov Substitution Principle

```
public class Rectangle
{
public void SetWidth(double w) {itsWidth=w;}
public void SetHeight(double h) {itsHeight=w;}
public double GetHeight() const {return itsHeight;}
public double GetWidth() const {return itsWidth;}
protected double itsWidth;
protected double itsHeight;
};
```

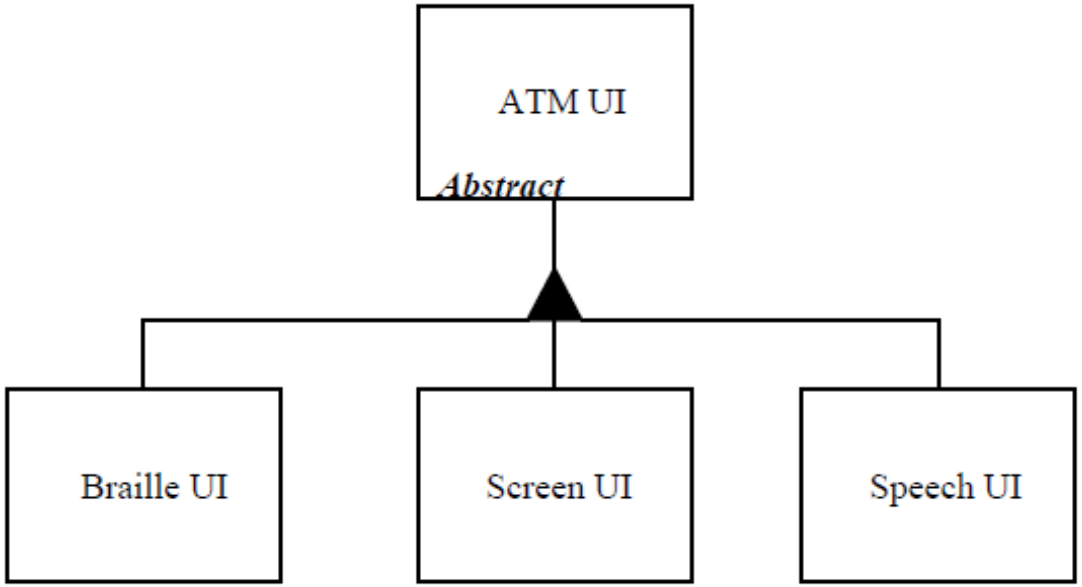
```
public class Square extends Rectangle
{
public void SetWidth(double w) {super.SetWidth(w);
                                 super.SetHeight(w);}
public void SetHeight(double h) {super.SetWidth(h);
                                 super.SetHeight(h);}
```

The Liskov Substitution Principle

```
public class Test
{
  public static void f(Rectangle r, float a)
  {
     r.SetWidth(a);
  }
}
public class Main
{
  public static void main(String args[])
  {
     Square square = new Square();
     square.SetWidth(10);
     Test test = new Test();
      test.f(square);
     //at this point the square is not a mathematical square
     //because its height is different from its length
  }
};
```

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 deceiving

Organization of Classes

- 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.

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

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.

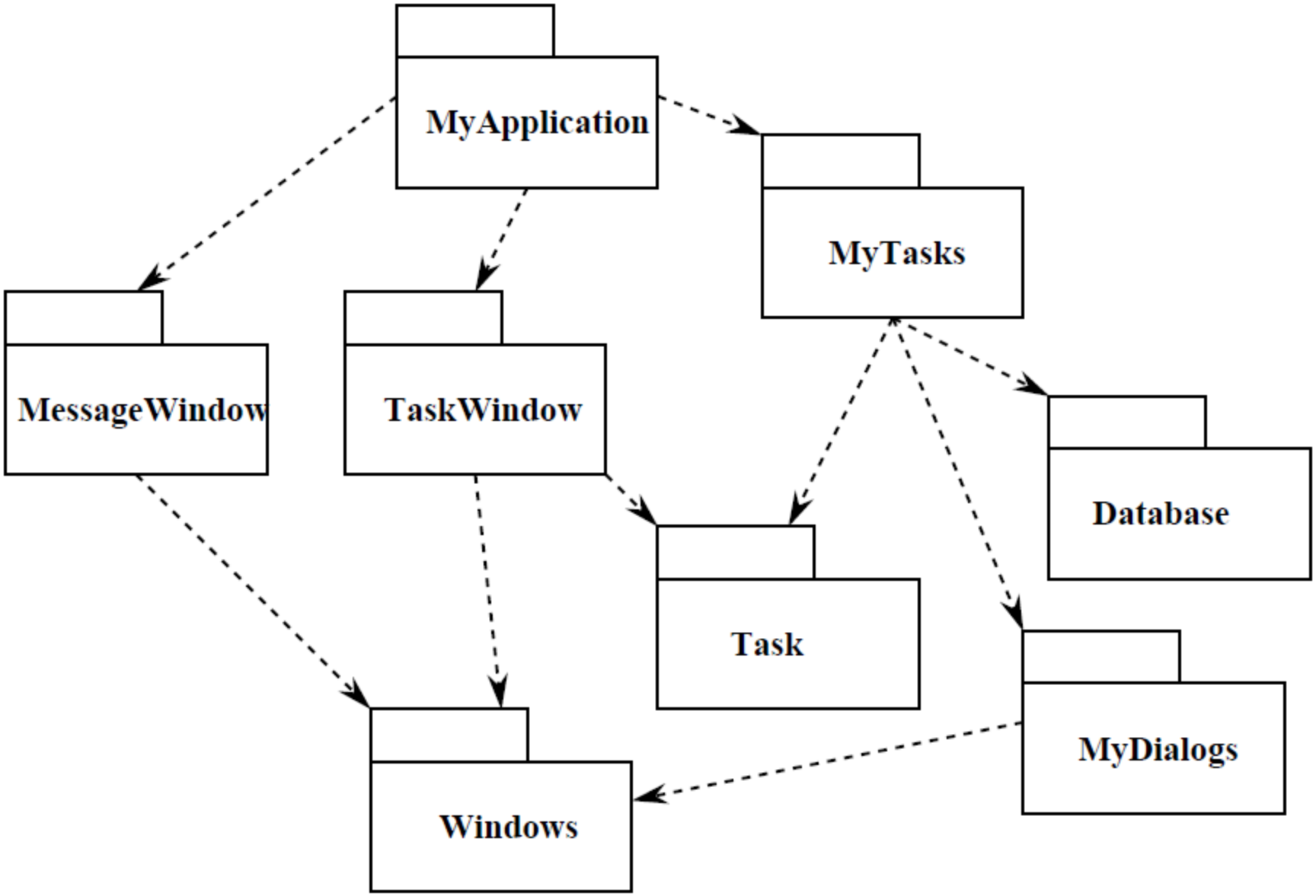
The Acyclic Dependencies Principle

- 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*

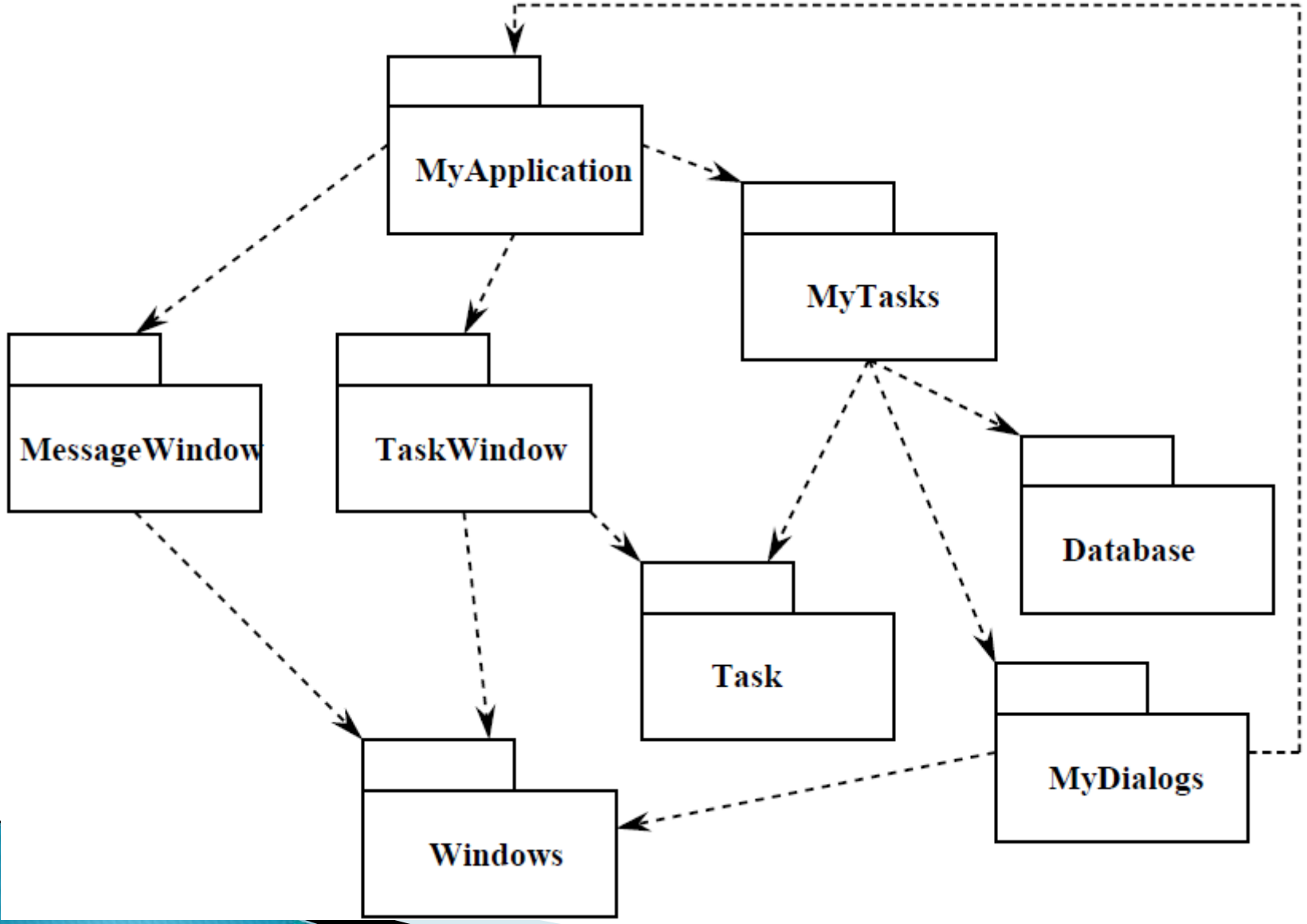
The Acyclic Dependencies Principle

- 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.

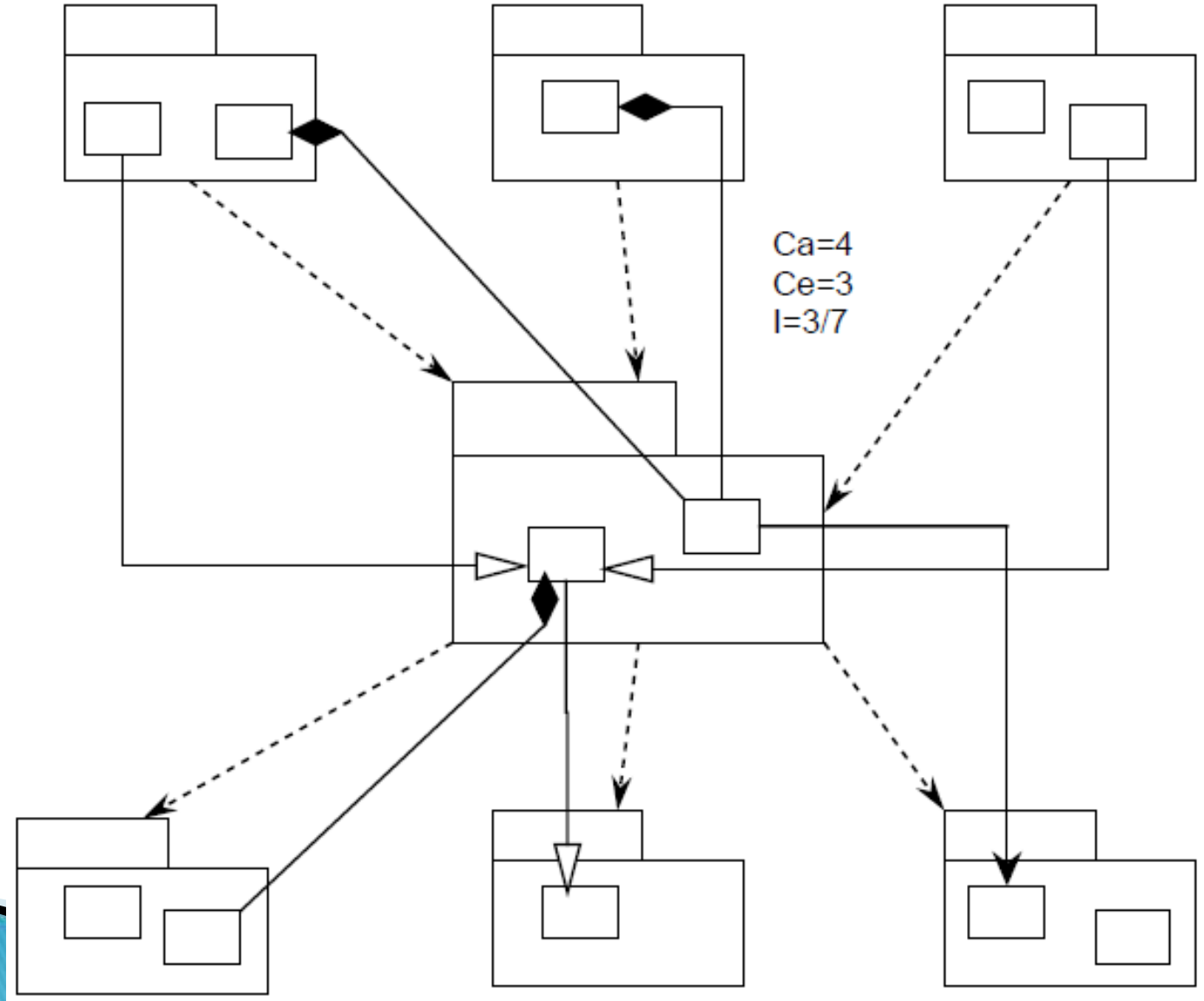
The Stable Dependencies Principle

- 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



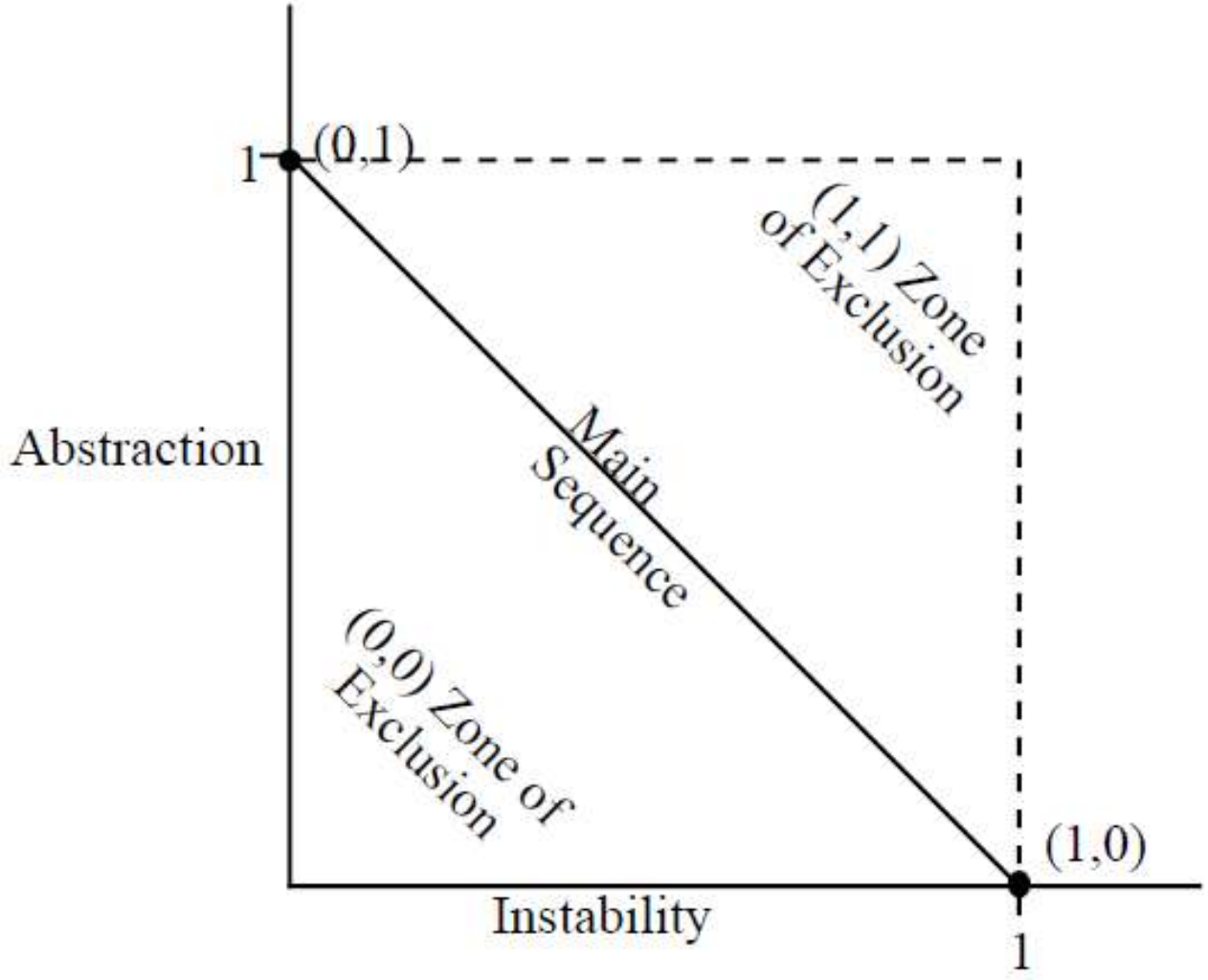
The Stable Dependencies Principle

- 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

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
- Design Principles and Design Patterns. Robert C. Martin.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)
- Adrian Iftene, Advanced Software Engineering Techniques