Principles of Design Patterns

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 22 — 03/31/2011

Goals of the Lecture

- Cover the material in Chapters 14 of our textbook
  - Principles of Design Patterns

Principles of Design Patterns (I)

- One benefit of studying design patterns is that they are based on good object-oriented principles
  - learning the principles increases the chance that you will apply them to your own designs
- **We've encountered several principles this semester already**

- **Code to an interface**
- **Encapsulate What Varies**
- **Only One Reason to Change**
- **Classes are about behavior**
  - **Prefer delegation over inheritance**
- **Dependency Inversion Principle**

Principles of Design Patterns (II)

- **Code to an interface**
  - If you have a choice between coding to an interface or an abstract base class as opposed to an implementation or subclass, choose the former
  - Let polymorphism be your friend
- Pizza store example
  - Two abstract base classes: Pizza and Pizza Store
  - There were a LOT of classes underneath, all hidden

Principles of Design Patterns (III)

- **Encapsulate What Varies**
  - Identify the ways in which your software will change
  - Hide the details of what can change behind the public interface of a class
    - Combine with previous principle for powerful results
      - Need to cover a new region? New PizzaStore subclass
      - Need a new type of pizza? New Pizza subclass

Principles of Design Patterns (IV)

- **Only One Reason to Change**
  - Each class should have only one design-related reason that can cause it to change
    - That reason should relate to the details that class encapsulates/hides from other classes
  - The FeatureImpl class discussed during last lecture has only one reason to change
    - a new CAD system requires new methods in order to fully access its features

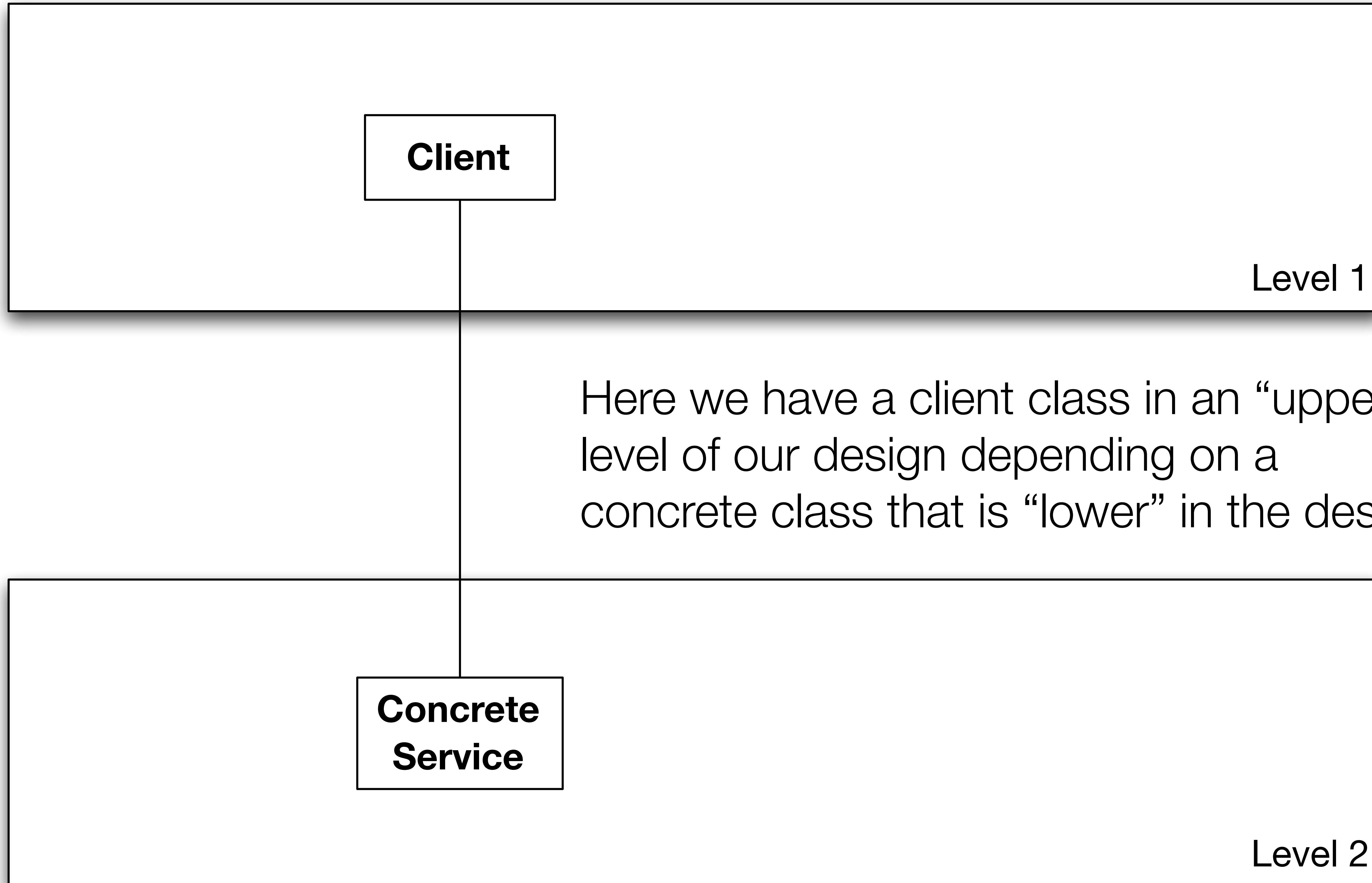
Principles of Design Patterns (V)

- **Classes are about behavior**
  - Emphasize the behavior of classes over the data of classes
    - Do not subclass for data-related reasons; It's too easy in such situations to violate the contract associated with the behaviors of the superclass
      - Think back to our Square IS-A/HAS-A Rectangle example
- Related: **Prefer Delegation over Inheritance**; to solve the Square/Rectangle problem, we resorted to delegation; it provides a LOT more flexibility, since delegation relationships can change at run-time

Principles of Design Patterns (VI)

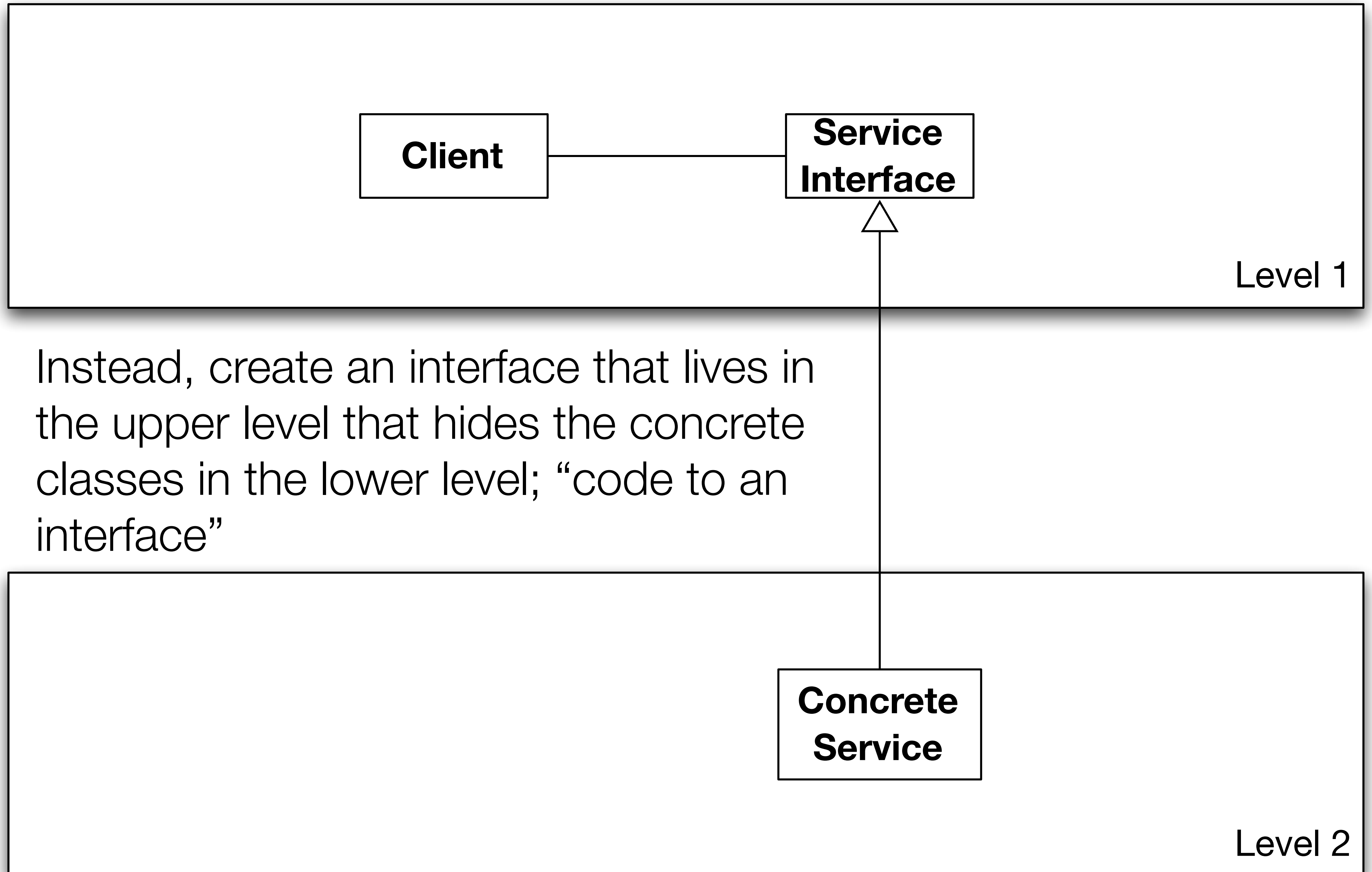
- **Dependency Inversion Principle**
- "Depend upon abstractions. Do not depend upon concrete classes."
- Normally "high-level" classes depend on "low-level" classes;
  - Instead, they BOTH should depend on an abstract interface
- We saw this when discussing the Factory Method back in lecture 9

Dependency Inversion Principle: Pictorially



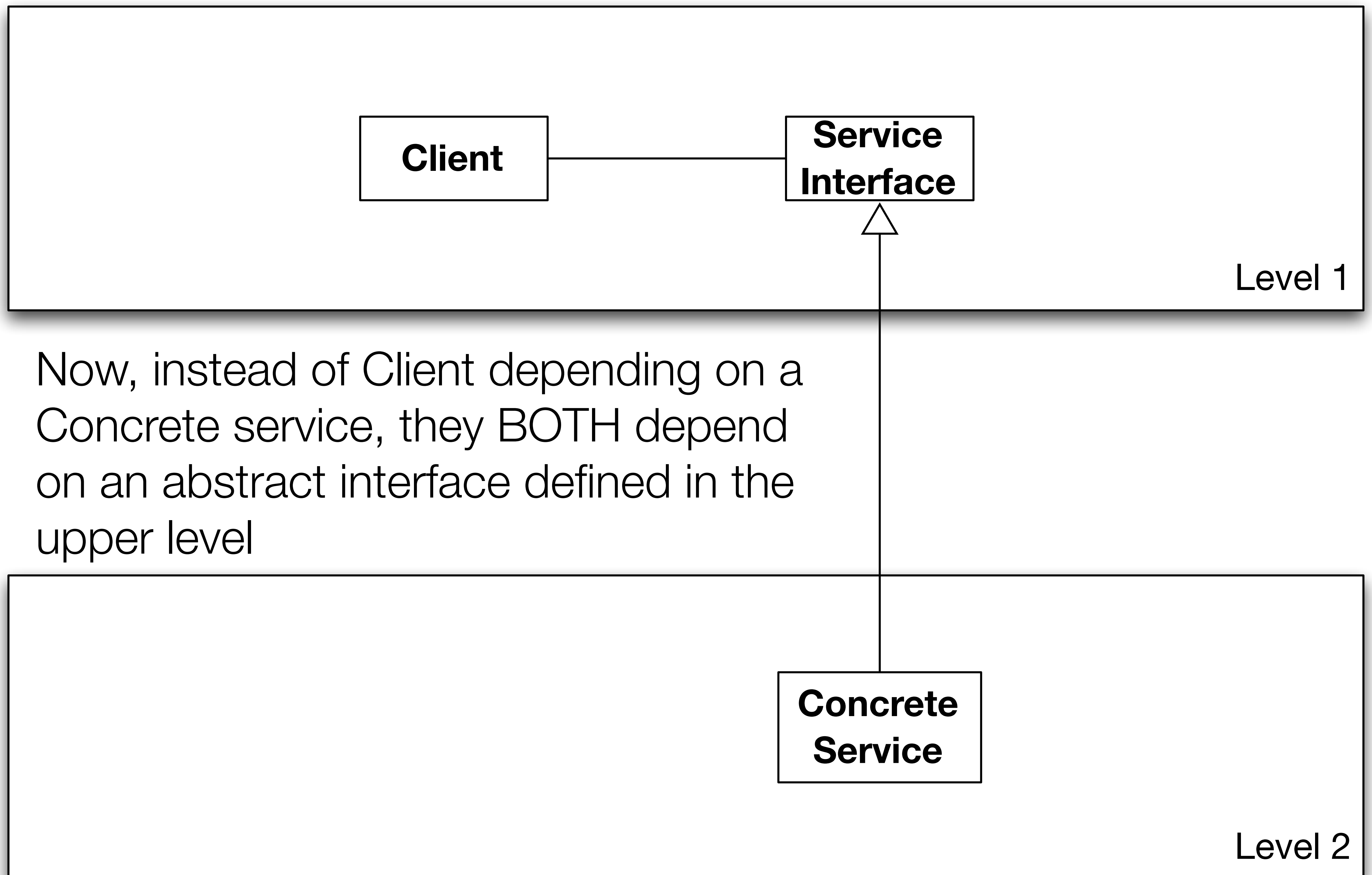
Here we have a client class in an “upper” level of our design depending on a concrete class that is “lower” in the design

Dependency Inversion Principle: Pictorially



Instead, create an interface that lives in the upper level that hides the concrete classes in the lower level; “code to an interface”

Dependency Inversion Principle: Pictorially



Now, instead of Client depending on a Concrete service, they BOTH depend on an abstract interface defined in the upper level

Principles of Design Patterns (VII)

- Let's learn about a few more principles
  - Open-Closed Principle
  - Don't Repeat Yourself
  - Single Responsibility Principle
  - Liskov Substitution Principle
- Some of these just reinforce what we've seen before
  - This is a GOOD thing, we need the repetition…

Open-Closed Principle (I)

- Classes should be open for extension and closed for modification
- Basic Idea:
  - Prevent, or heavily discourage, changes to the behavior of existing classes
    - especially classes that exist near the root of an inheritance hierarchy
  - You've got a lot of code that depends on this behavior
    - It should not be changed lightly

Open-Closed Principle (II)

- If a change is required, one approach would be to create a subclass and allow it to extend/override the original behavior
  - This means you must carefully design what methods are made public and protected in these classes
  - private methods cannot be extended

Is this just about Inheritance? (I)

- Inheritance is certainly the easiest way to apply this principle
    - but its not the only way
- Think about the delegate pattern we saw in iOS
    - We can customize a class's behavior significantly by having it assume the existence of a delegate
    - If the delegate implements a delegate method, then call it, otherwise invoke default behavior

Is this just about Inheritance? (II)

- In looking at Design Patterns, we see that **composition and delegation offer more flexibility in extending the behavior of a system**
  - Inheritance still plays a role but we will try to rely on delegation and composition first

Open-Closed Principle (III)

- Returning to the open-closed principle, the key point is to get you to **be reluctant to change working code**
  - look for opportunities to extend, compose and/or delegate your way to achieve what you need first

Don't Repeat Yourself (I)

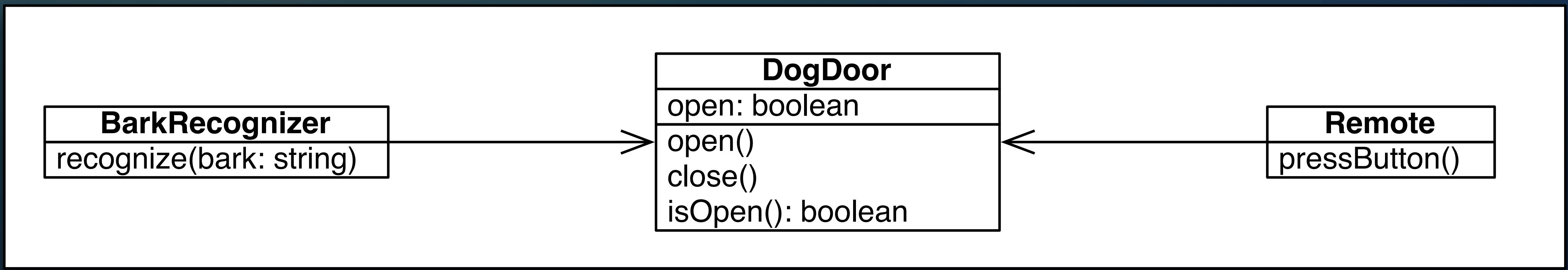
- Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- Basic Idea
  - Duplication is Bad!
    - … at all stages of software engineering: analysis, design, implement, and test

Don't Repeat Yourself (II)

- We want to avoid duplication in our requirements & use cases
- We want to avoid duplication of responsibilities in our code
- We want to avoid duplication of test coverage in our tests
- Why?
  - Incremental errors can creep into a system when one copy is changed but the others are not
  - Isolation of Change Requests (a benefit of Cohesion)
    - We want to go to ONE place when responding to a change request

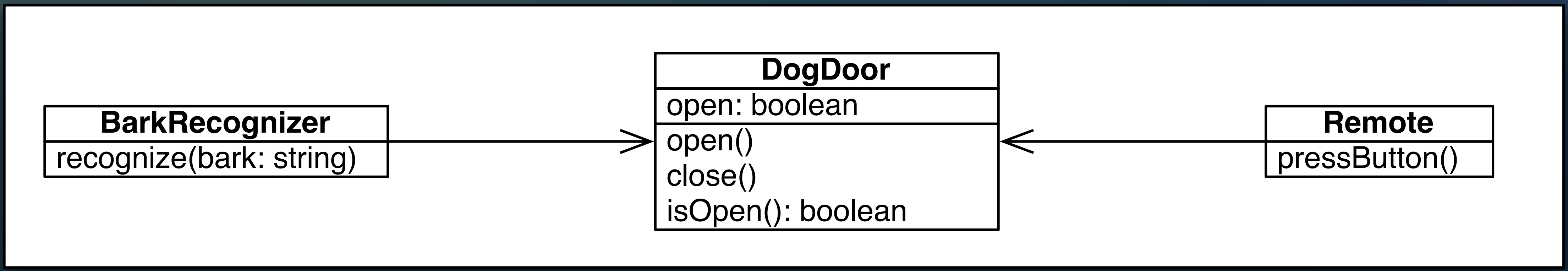
Example (I)

- Duplication of Code: Imagine the following system



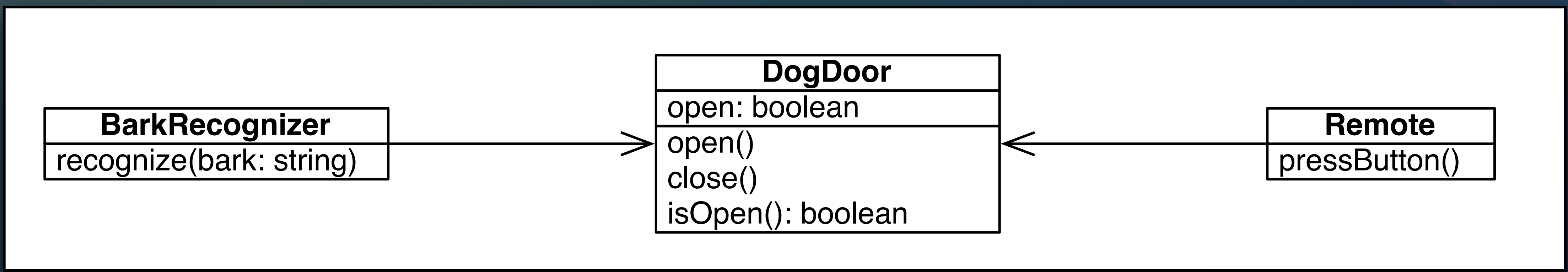
- Suppose we had the responsibility for closing the door live in the Remote class (which was implemented first)
- When we add the BarkRecognizer, the first time we use it we'll discover that it won't auto-close the door

Example (II)



- We then have a choice:
  - we could add the code from Remote for closing the door automatically to the BarkRecognizer
- But that would violate Don't Repeat Yourself

Example (III)



- OR
  - we could remove the auto-close code from Remote and move it to DogDoor
  - now, the responsibility lives in one place

Don’t Repeat Yourself (III)

- DRY is really about ONE requirement in ONE place
  - We want each responsibility of the system to live in a single, sensible place
- To aid in this, you must make sure that there is no duplication hiding in your requirements

Example (I)

- New Requirements for the Dog Door System: Beware of Duplicates
  - The dog door should alert the owner when something inside the house gets too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
  - The dog door should make a noise if the door cannot open because of a blockage outside
  - The dog door will track how many times the dog uses the door
  - When the door closes, the house alarm will re-arm if it was active before the door opened

Example (II)

- New Requirements for the Dog Door System: Beware of Duplicates
  - The dog door should alert the owner when something inside the house gets too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system to make sure it doesn't activate when the dog door is open
  - The dog door should make a noise if the door cannot open because of a blockage outside
  - The dog door will track how many times the dog uses the door
  - When the door closes, the house alarm will re-arm if it was active before the door opened

Example (III)

- New Requirements for the Dog Door System
  - The dog door should alert the owner when something is too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system
  - The dog door will track how many times the dog uses the door
- Duplicates Removed!

Example (IV)

- Ruby on Rails makes use of DRY as a core part of its design
  - focused configuration files; no duplication of information
  - for each request, often single controller, single model update, single view
- But, prior to Ruby on Rails 1.2, there was duplication hiding in the URLs used by Rails applications
  - `POST /people/create     # create a new person`
  - `GET /people/show/1      # show person with id 1`
  - `POST /people/update/1   # edit person with id 1`
  - `POST /people/destroy/1 # delete person with id 1`

Example (V)

- The duplication exists between the HTTP method name and the operation name in the URL
  - `POST /people/create`
- Recently, there has been a movement to make use of the four major "verbs" of HTTP
  - PUT/POST == create information (create)
  - GET == retrieve information (read)
  - POST == update information (update)
  - DELETE == destroy information (destroy)
- These verbs mirror the CRUD operations found in databases
  - Thus, saying "create" in the URL above is a duplication

Example (VI)

- In version 1.2, Rails eliminates this duplication; Now URLs look like this:
    - POST /people
    - GET /people/1
    - PUT /people/1
    - DELETE /people/1
- And the duplication is **logically** eliminated
    - Disclaimer: … but not actually eliminated… Web servers do not universally support PUT and DELETE "out of the box". As a result, Rails uses POST
        - POST /people/1
          Post-Semantics: Delete

Single Responsibility Principle (I)

- Every object in your system should have a single responsibility, and all the object's services should be focused on carrying it out
  - This is obviously related to the "One Reason to Change" principle
  - If you have implemented SRP correctly, then each class will have only one reason to change

Single Responsibility Principle (II)

- The "single responsibility" doesn't have to be "small", it might be a major design-related goal assigned to a package of objects, such as "inventory management" in an adventure game
- We've encountered SRP before
  - SRP == high cohesion
  - "One Reason To Change" **promotes** SRP
  - DRY is often used to achieve SRP

Textual Analysis and SRP (I)

- One way of identifying high cohesion in a system is to do the following
  - For each class C
    - For each method M
      - Write "The C Ms itself"
  - Examples
    - The Automobile drives itself
    - The Automobile washes itself
    - The Automobile starts itself

Textual Analysis and SRP (II)

- If any one of the generated sentences doesn't make sense then investigate further.
  - "The Automobile puts fuel in itself."
- You may have discovered a service that belongs to a different responsibility of the system and should be moved to a different class (Gas Station)
  - This may require first creating a new class before performing the move

Liskov Substitution Principle (I)

- Subtypes must be substitutable for their base types
- Basic Idea
  - Instances of subclasses do not violate the behaviors exhibited by instances of their superclasses
    - They may constrain that behavior but they do not **contradict** that behavior

Liskov Substitution Principle (II)

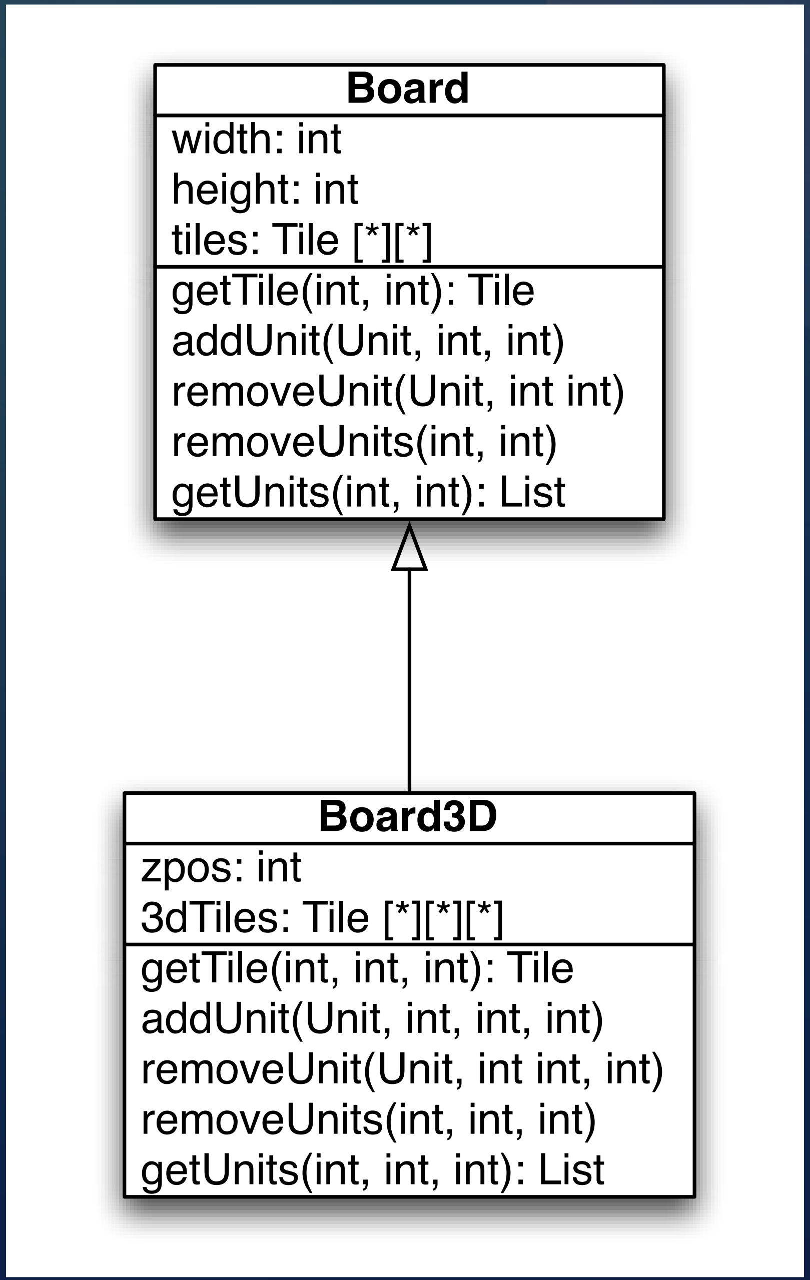
- Named after Barbara Liskov who co-authored a paper with Jeannette Wing in 1993 entitled ***Family Values: A Behavioral Notion of Subtyping***
  - *Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.*
- Properties that hold on superclass objects, hold on subclass objects
  - **Return to Rectangle/Square**: *WidthAndHeightMayBeDifferent(Rectangle)* equals true for Rectangles and equals false for Square

Well-Designed Inheritance

- LSP is about well-designed inheritance
  - When I put an instance of a subclass in a place where I normally place an instance of its superclass
    - the functionality of the system must remain **correct**
    - (not necessarily the **same**, but correct)

Bad Example (I)

- Extend Board to produce Board3D
- Board handles the 2D situation
  - so it should be easy to extend that implementation to handle the 3D case, right? RIGHT?
- Nope



Bad Example (II)

- But look at an instance of Board3D…
  - Each attribute and method in bold is meaningless in this object
  - Board3D is getting nothing useful from Board except for width and height!!
  - We certainly could NOT create a Board3D object and hand it to code expecting a Board object!
  - As a result, this design violates the LSP principle; How to fix?

: Board3D
width: int height: int zpos: int **tiles: Tile [*][*]** 3dTiles: Tile [*][*][*]
getTile(int, int): Tile **addUnit(Unit, int, int)** **removeUnit(Unit, int int)** **removeUnits(int, int)** **getUnits(int, int): List** getTile(int, int, int): Tile addUnit(Unit, int, int, int) removeUnit(Unit, int int, int) removeUnits(int, int, int) getUnits(int, int, int): List

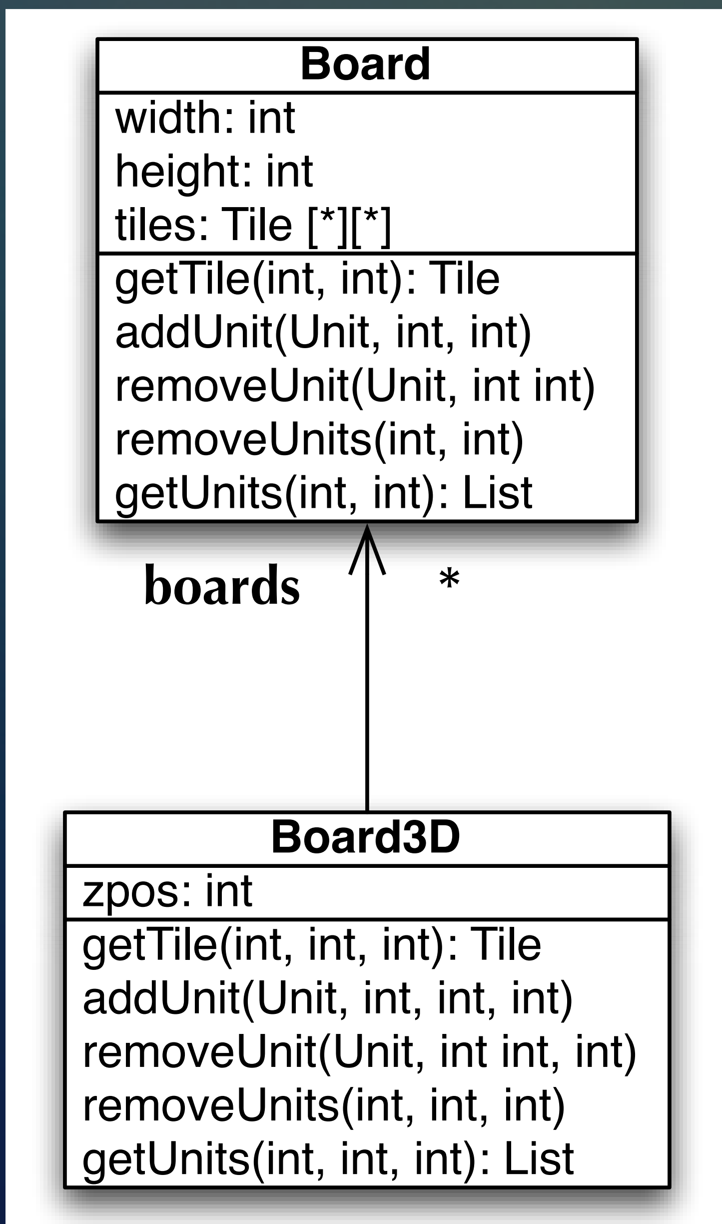
Delegation to the Rescue! (Again)

- You can understand why a designer thought they could extend Board when creating Board3D
    - Board has a lot of useful functionality and a Board3D should try to reuse that functionality as much as possible
    - However, the Board3D has no need to CHANGE that functionality and the Board3D doesn't really behave in the same way as a board
        - For instance, a unit on "level 10" may be able to attack a unit on "level 1"; such functionality doesn't make sense in the context of a 2D board

Delegation to the Rescue! (Again)

- Thus, if you need to use functionality in another class, but you don't want to change that functionality, consider using **delegation** instead of inheritance
  - Inheritance was simply the wrong way to gain access to the Board's functionality
  - Delegation is when you hand over the responsibility for a particular task to some other class or method

New Class Diagram



Board3D now maintains a list of Board objects for each legal value of "zpos"

It then delegates to the Board object as needed

```
public Tile getTile(int x, int y, int z) {
    Board b = boards.get(z);
    return b.getTile(x,y);
}
```

Summary of New Principles

- **Open-Closed Principle (OCP)**
  - Classes should be open for extension and closed for modification
- **Don’t Repeat Yourself (DRY)**
  - Avoid duplicate code by abstracting out things that are common and placing those things in a single location
- **Single Responsibility Principle (SRP)**
  - Every object in your system should have a single responsibility, and all the object’s services should be focused on carrying it out
- **Liskov Substitution Principle (LSP)**
  - Subtypes must be substitutable for their base types

Use of Principles in Design Patterns

- When you look at a pattern, you'll see evidence of these principles everywhere
- Strategy Pattern **So simple yet so powerful!**
  - Code to an interface (the algorithm)
  - Prefer delegation over inheritance
  - Inheritance used between the abstract algorithm and the concrete algorithms because they will all behave similarly; Liskov Substitution Principle
  - Dependency Inversion Principle (everything depends on algorithm)
  - Encapsulate What Varies (concrete algorithms hidden behind abstract)
  - Open Closed Principle; client object is not modified directly, new behavior comes from a new concrete algorithm subclass

The Principle of Healthy Skepticism

- Chapter 14 ends with a warning not to depend on patterns for everything
- "Patterns are useful guides but dangerous crutches…"
  - Patterns are useful in guiding/augmenting your thinking during design
    - use the ones most relevant to your context
    - but understand that they won't just hand you a solution… creativity and experience are still key aspects of the design process

Problems (I)

- Problems that can occur from an over reliance on patterns
- **Superficiality**: selecting a pattern based on a superficial understanding of the problem domain
- **Bias**: When all you have is a hammer, everything looks like a nail; a favorite pattern may bias you to a solution that is inappropriate to your current problem domain
- **Incorrect Selection**: not understanding the problem a pattern is designed to solve and thus inappropriately selecting it for your problem domain

Problems (II)

- Problems that can occur from an over reliance on patterns
- **Misdiagnosis**: occurs when an analyst selects the wrong pattern because they don't know about alternatives; has not had a chance to absorb the entire range of patterns available to software developers
- **Fit:** applies a pattern to a set of objects that do not quite exhibit the range of behaviors the pattern is supposed to support; the objects don't "fit" the pattern and so the pattern does not provide all of its benefits to your system

Wrapping Up

- Principles of Design Patterns
  - We've now encountered ten OO design principles
  - Looked at how they are applied in certain patterns
  - Cautioned against an over reliance on patterns
    - They are useful but they can't be your hammer
      - They are one tool among many in performing OO A&D

Coming Up Next

- Homework 6 due on Friday
- Homework 7 assigned on Friday
- Lecture 23: Commonality and Variability Analysis & The Analysis Matrix
    - Chapters 15 and 16
- Lecture 24: Decorator, Observer, Template Method
    - Chapters 17, 18 and 19