Principles of Design Patterns

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 25

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Goals of the Lecture

- Quick review of principles of OO related to Design Patterns
- Touch on Code Complete class construction

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 looked at the OO principles almost every lecture…

OO Principles

- General
  - Program to interfaces not implementations
  - Encapsulate what varies
  - A class should have only one reason to change
  - Classes are about behavior, not specialization
  - Favor delegation (composition) over inheritance
  - Strive for loosely coupled designs between objects that interact
  - Only talk to your (immediate) friends (Law of Demeter, Principle of Least Knowledge)
  - Don’t call us, we’ll call you (the Hollywood Principle)
- SOLID
  - Single Responsibility Principle (SRP)
  - Open-Closed Principle - Classes should be open for extension, but closed for modification
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle - Depend on abstractions, not concrete classes
- Other
  - Don’t repeat yourself (DRY Principle)
  - You Aren’t Going to Need It (YAGNI Principle)
  - The Principle of Healthy Skepticism

Principles of Design Patterns (II)

- **Program to interfaces not implementations**
- **Aka 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 Code To An Interface 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)

- **A Class Should Have 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, not specialization**
  - 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 SpecialOrderBlinkingSpinningPentagon example
- Related: **Prefer Delegation over Inheritance**
  - to solve the Special Pentagon issues, we resorted to delegation; it provides a LOT more flexibility, since delegation relationships can change at run-time

SOLID Principles of OO

- **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
- **Open-Closed Principle (OCP)**
  - Classes should be open for extension and closed for modification
- **Liskov Substitution Principle (LSP)**
  - Subtypes must be substitutable for their base types
- **Interface Segregation Principle (ISP)**
  - Define subsets of functionality as interfaces
- **Dependency Inversion Principle (DIP)**
  - Depend upon abstractions. Do not depend upon concrete classes.

- Principles were presented together in 2000 in "Uncle Bob" Martin's "Design Principles and Design Patterns"
- SOLID acronym was introduced later by Michael Feathers, author of "Working Effectively with Legacy Code"

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

- 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
- If any one of the generated sentences does not 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

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
  - 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
- Inheritance is certainly the easiest way to apply this principle, but…
- 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 (II)

- For 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

Liskov Substitution Principle (I)

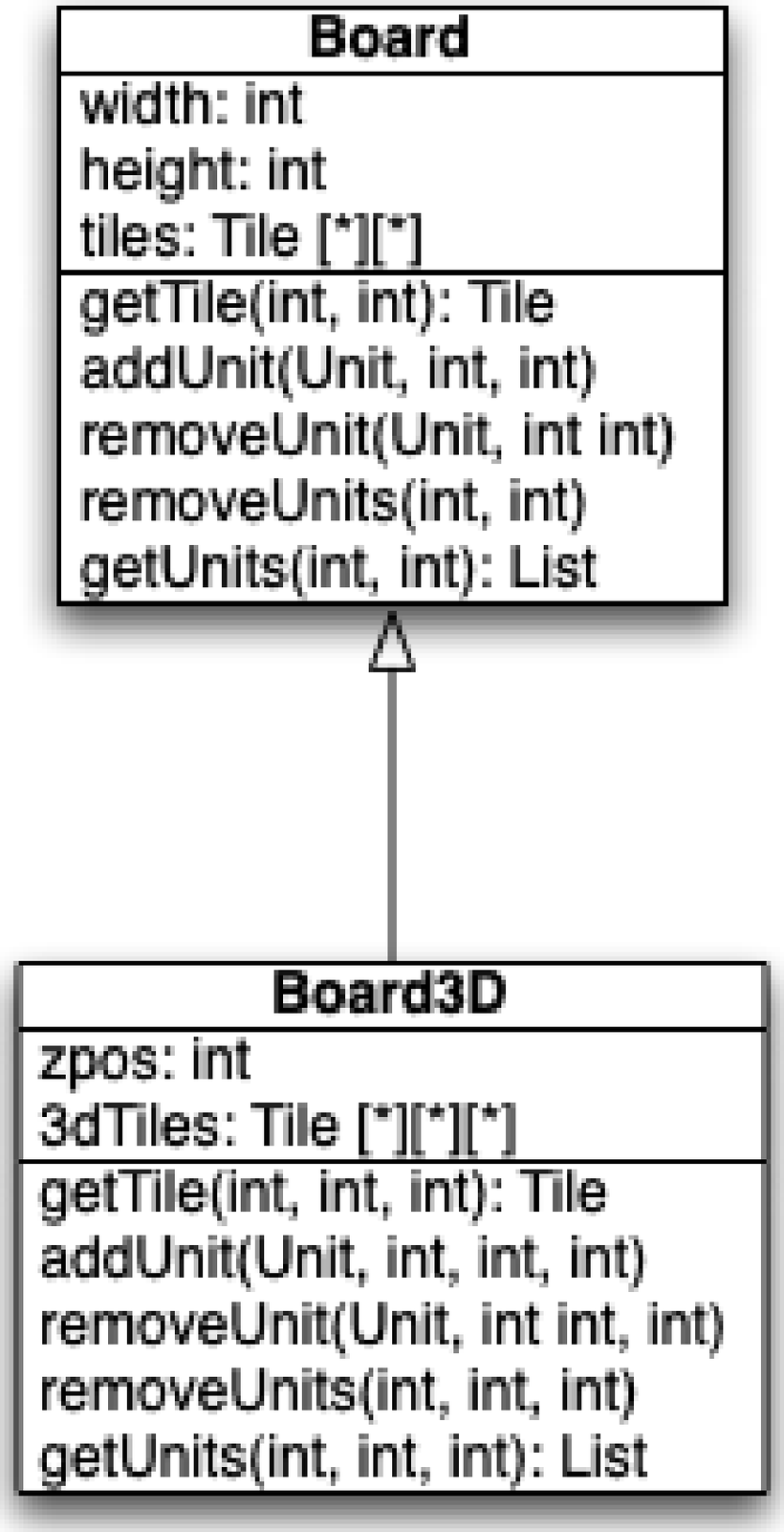
- Subtypes must be substitutable for their base types
- Basic Idea
  - Instances of subclasses do not violate the behaviors exhibited by instances of their superclasses
  - They may constrain that behavior, but they do not contradict that behavior
- Named after Barbara Liskov who co-authored a paper 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

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)

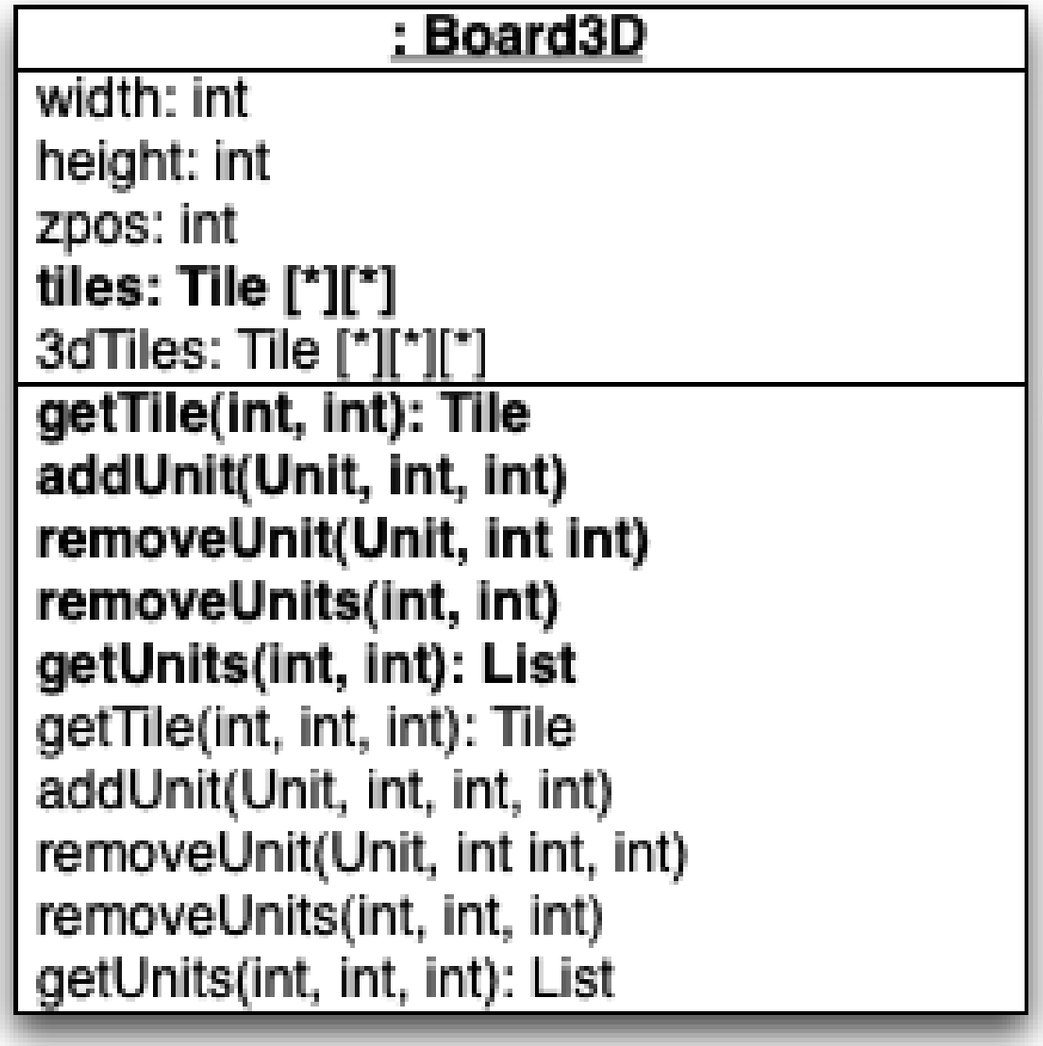
Bad Example (I)

- 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

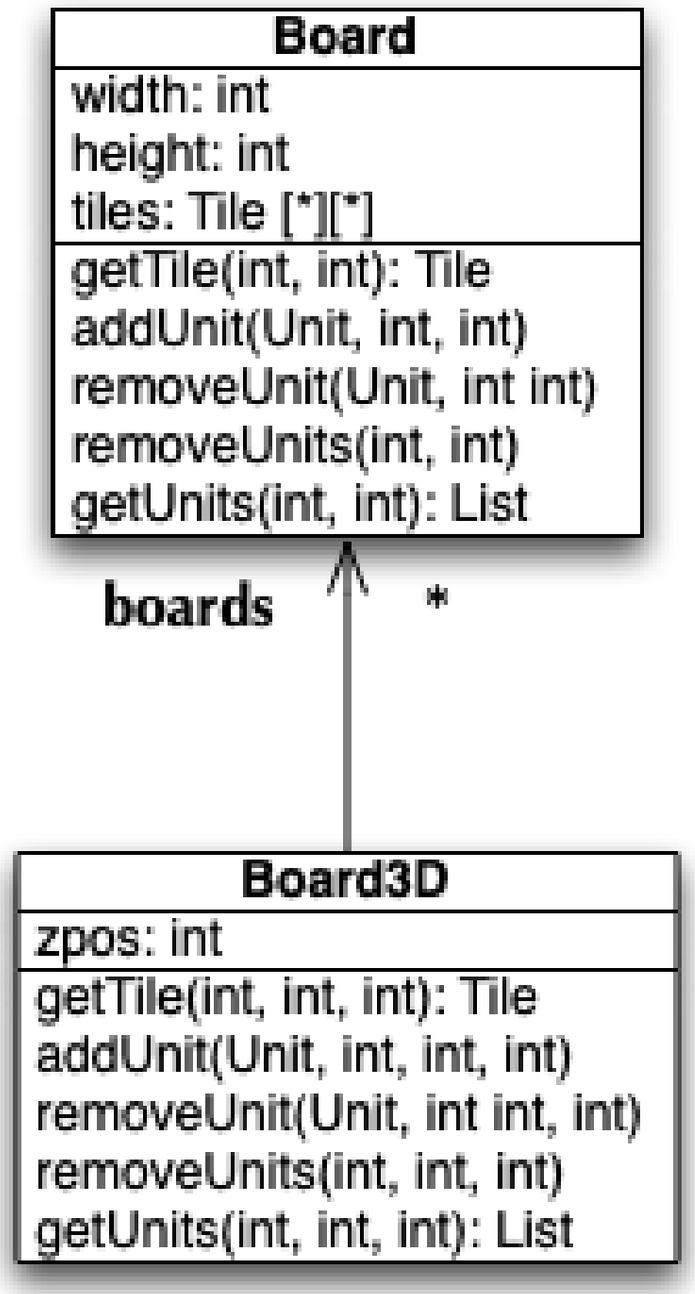


Bad Example (II)

- 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?



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);
}
```

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 does not 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
- 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

Interface Segregation Principle (ISP)

- Interface Segregation Principle as defined by Bob Martin – no client should be forced to depend on methods it doesn't use
  - Related to single responsibility…
- The goal is to reduce the impact and frequency of changes by splitting software into multiple independent parts
- Correct abstraction is a key to this – keeping interfaces small and cohesive
- Also need the ability to inherit multiple interfaces if required

- https://www.oodesign.com/interface-segregation-principle.html

Interface Segregation Principle (ISP)

```
// interface segregation principle - bad example
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {...working…}
        public void eat() {…eating…}
}
```

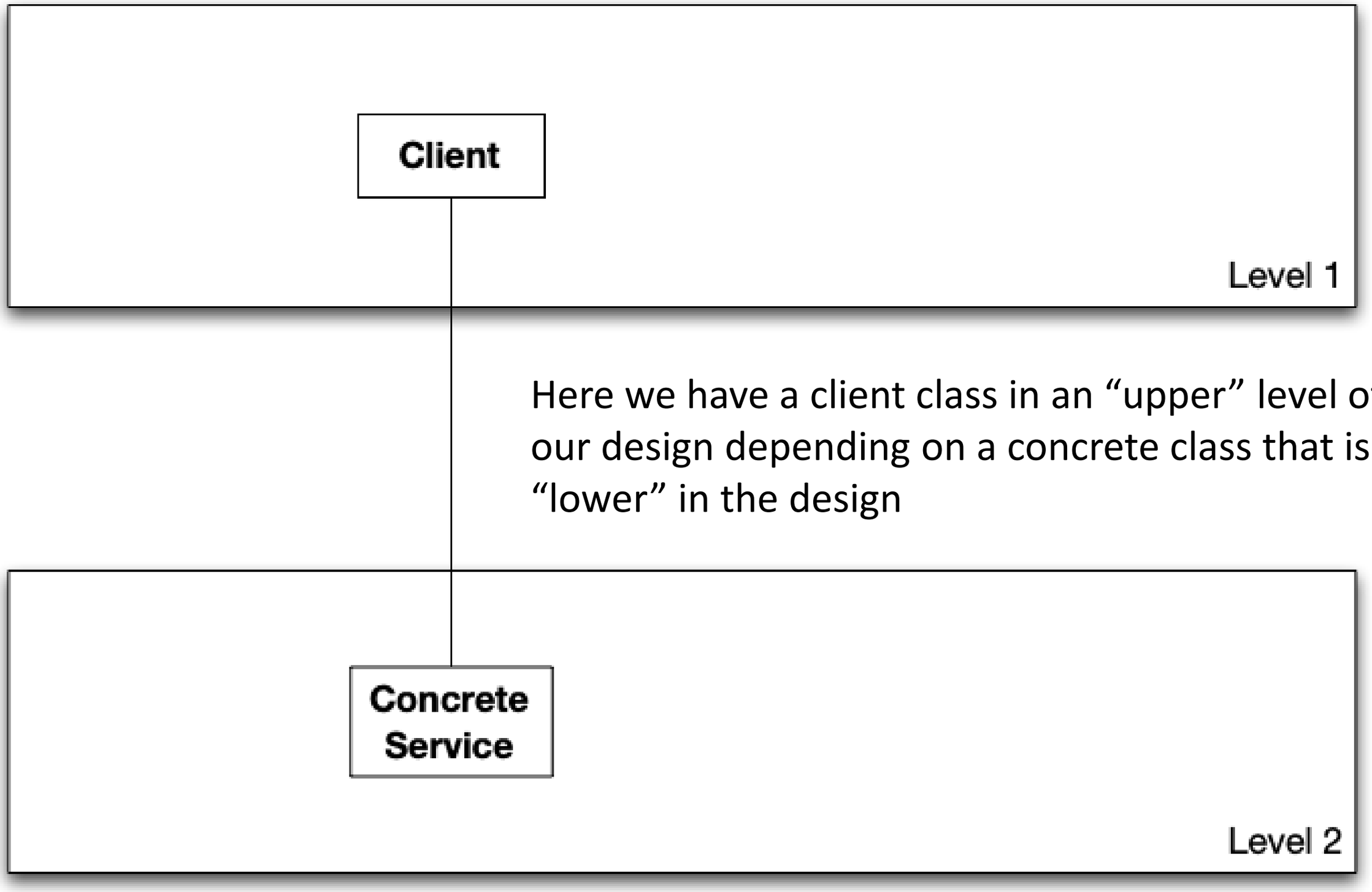
https://www.oodesign.com/interface-segregation-principle.html

```
// interface segregation principle – good example

interface IWorkable {
        public void work();
}

interface IFeedable{
        public void eat();
}

class Worker implements IWorkable, IFeedable{
        public void work() {...working…}
        public void eat() {…eating…}
}

class Robot implements IWorkable{
        public void work() {...just working…}
}
```

Dependency Inversion Principle

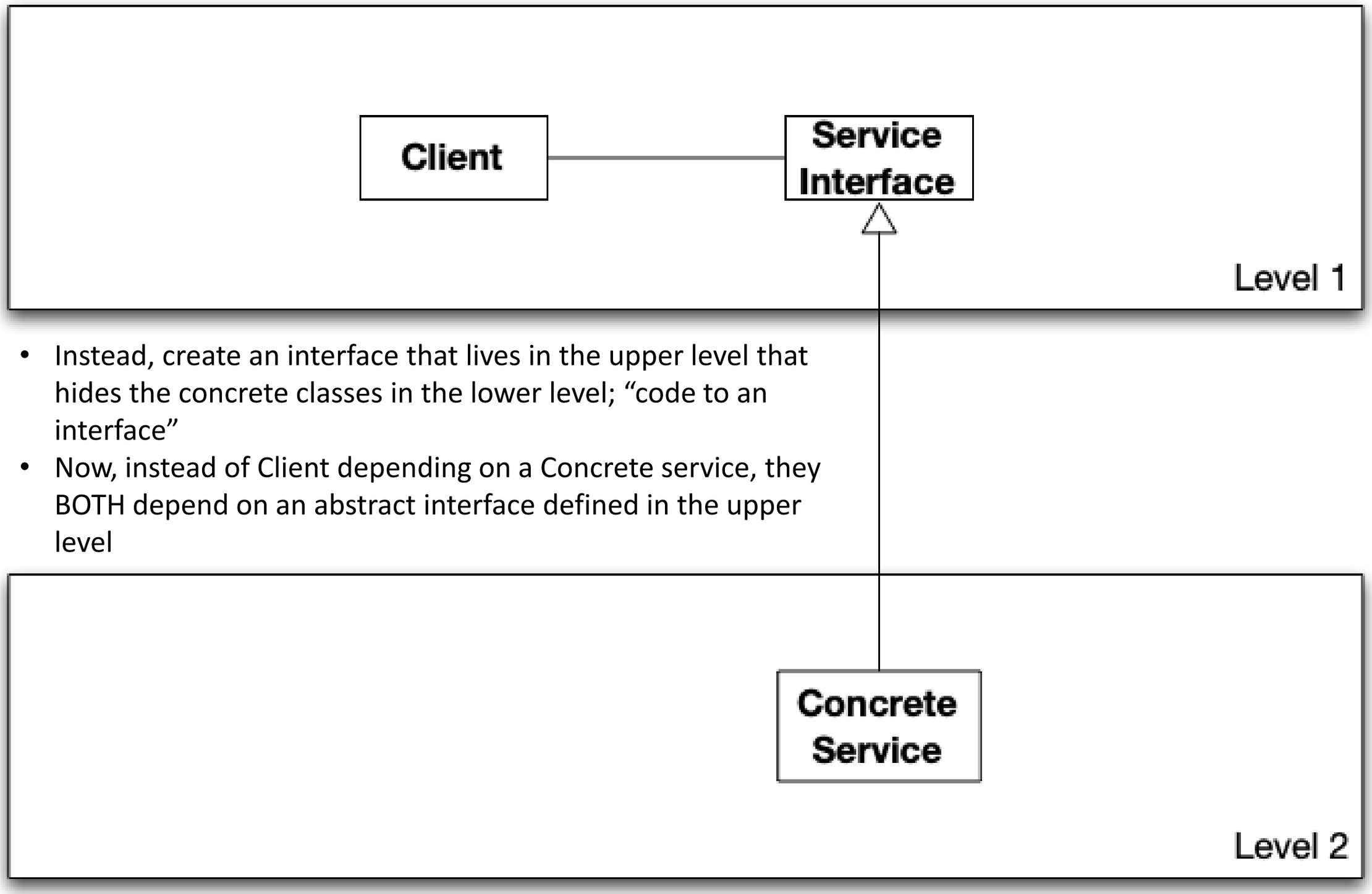
- “Depend upon abstractions. Do not depend upon concrete classes.”
- Normally “high-level” classes depend on “low-level” classes;
- Instead, they BOTH should depend on an abstract interface
- We saw this when discussing the Factory Methods

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



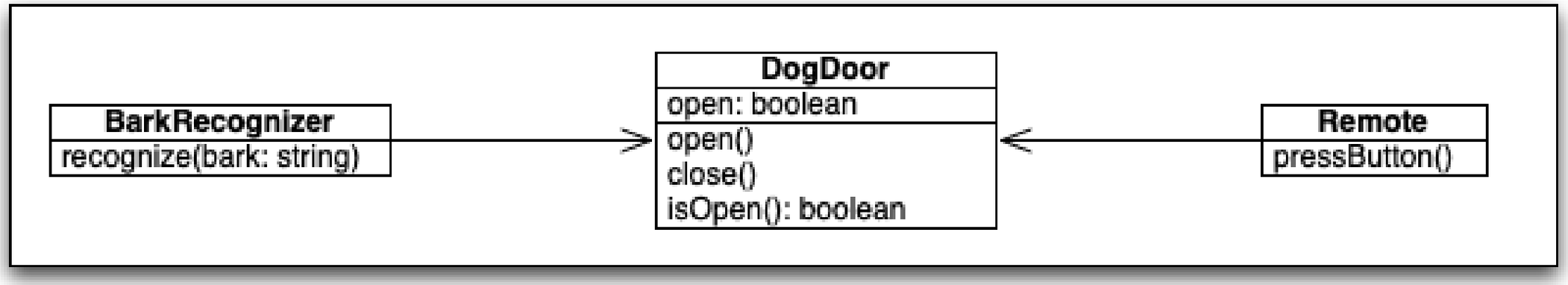
- Instead, create an interface that lives in the upper level that hides the concrete classes in the lower level; “code to an interface”
- Now, instead of Client depending on a Concrete service, they BOTH depend on an abstract interface defined in the upper level

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!
- 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

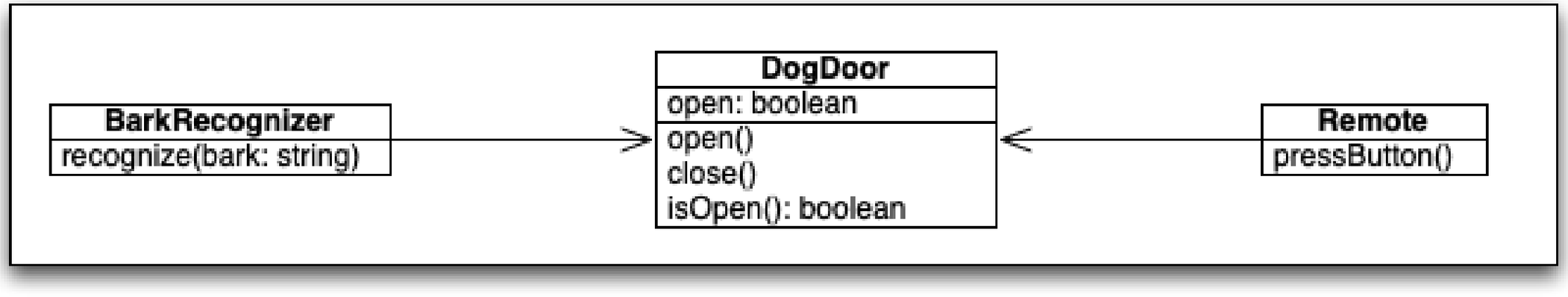
Example (I)

- Duplication of Code: Imagine the following system



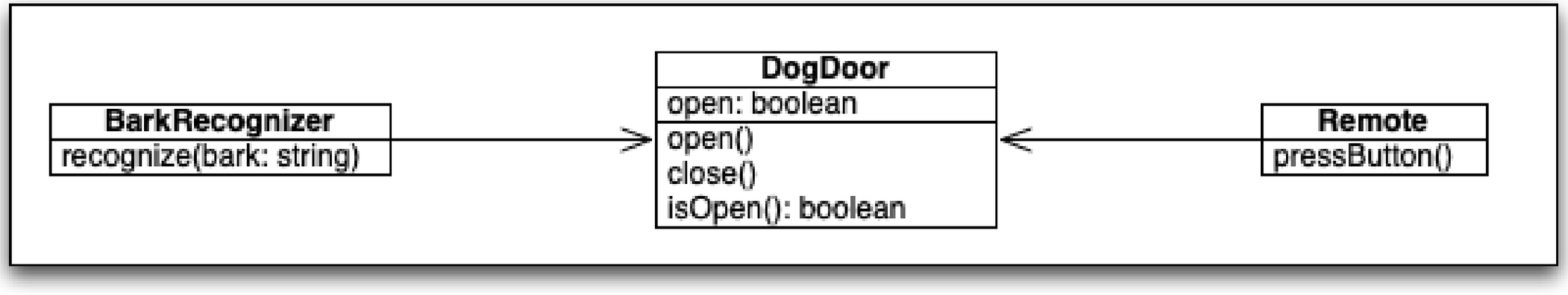
- Suppose we had the responsibility for closing the door live in the Remote class (which was implemented first)
- When we add the BarkRecognizer, the first time we use it we'll discover that it won't auto-close the door

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 (II)

- 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)

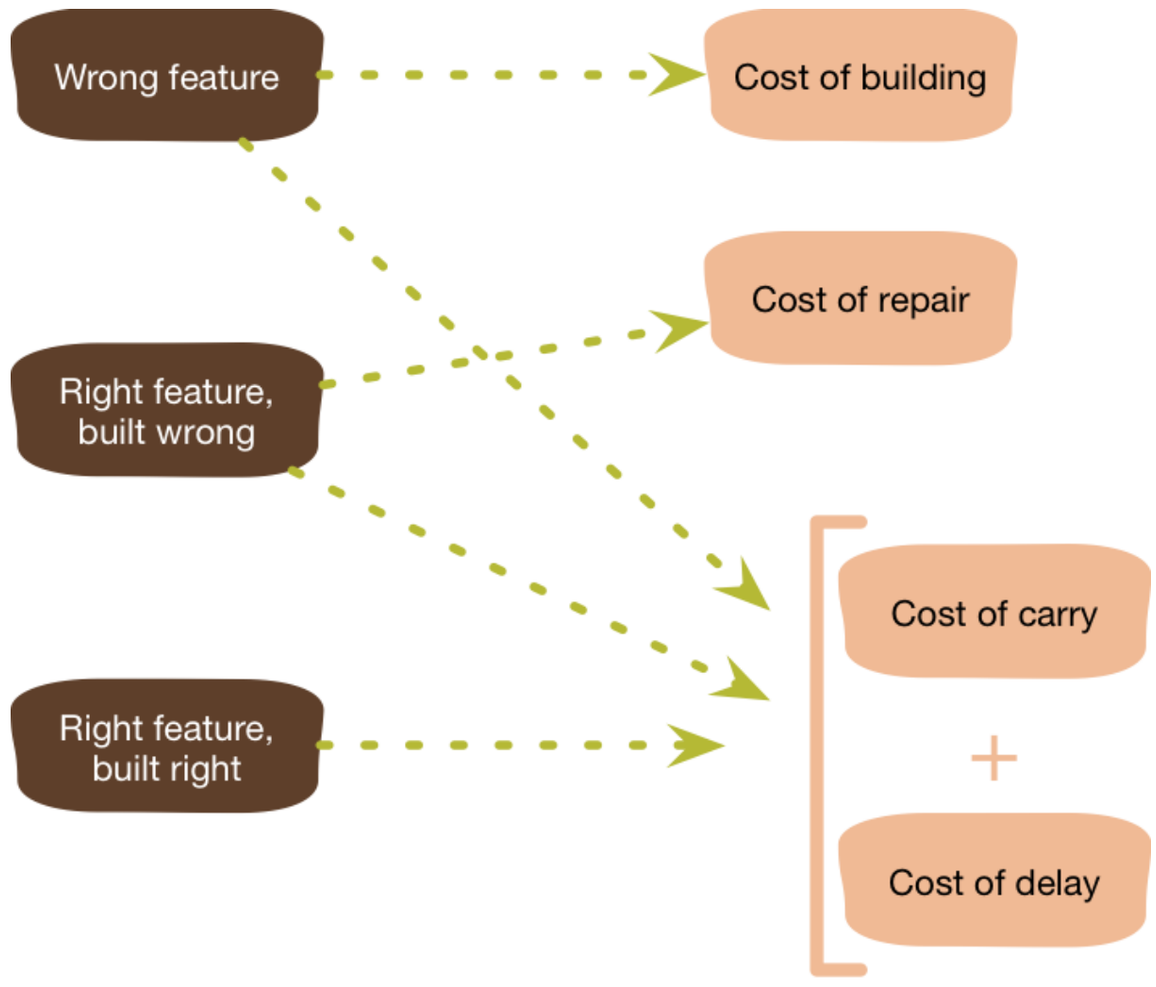
- New Requirements for the Dog Door System: Beware of Duplicates
  - The dog door should alert the owner when something inside the house gets too close to the dog door
  - The dog door will open only during certain hours of the day
  - The dog door will be integrated into the house's alarm system to 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!

YAGNI: You Ain’t Gonna Need It (aka You Aren’t Going to Need It)

- YAGNI comes from Extreme Programming’s Simple Design rule and talks to avoiding “presumptive” features for code
- There are clear costs associated with writing code before it’s needed
  - Cost of build/refactoring time
  - Cost of delay
  - Cost of carry
  - Cost of repair
- https://martinfowler.com/bliki/Yagni.html



The Principle of Healthy Skepticism

- Chapter 14 in Shalloway/Trott 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
- Related to Fred Brooks "No Silver Bullet"
  - Famous 1986 Paper
  - No single software development or technique, management or technology, will yield an order of magnitude productivity, reliability, or simplicity increase

Problems

- 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
  - **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

Code Complete 2 on Reasons to Create a Class

- Model a real-world object
- Model an abstract object
- Reduce complexity
- Isolate complexity
- Hide implementation details
- Limit the impact of change
- Hide “global” data

- Steve McConnell, Code Complete 2

- Streamline parameter passing
- Make central points of control
- Facilitate reusable code
- Plan for a family of programs
- Package related operations
- Accomplish specific refactoring

Classes to avoid:
- “God” classes
- Irrelevant classes
- Classes that are verbs (behavior only)

Code Complete 2 on Class Construction

- Abstraction - the ability to view a complex operation in a simplified form
  - Be sure YOU know what abstraction the class is implementing
  - Beware of abstraction erosion over time (with modifications)
- Abstraction and cohesion go together
  - A class with a clear abstraction is likely cohesive around its functionality
  - A cohesive class is more likely to have a clearly presented abstraction (not always)
- Minimize access to attributes and methods
  - Generally err on the side of more hiding than less
  - Avoid friend classes (classes with access to private/protected data/methods)
- Many methods will naturally need an opposite
- Etc.

- Steve McConnell, Code Complete 2

Code Complete 2 – Class Construction Guidelines

- Abstraction
  - Does the class have a central purpose?
  - Is the class well named, and does its name describe its central purpose?
  - Does the class's interface present a consistent abstraction?
  - Does the class's interface make obvious how you should use the class?
  - Is the class's interface abstract enough that you don't have to think about how its services are implemented?
  - Can you treat the class as a black box?
  - Are the class's services complete enough that other classes don't have to meddle with its internal data?
  - Has unrelated information been moved out of the class?
  - Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
  - Are you preserving the integrity of the class's interface as you modify the class?

- Steve McConnell, Code Complete 2

Code Complete 2 – Class Construction Guidelines

- Encapsulation
  - Does the class minimize accessibility to its members?
  - Does the class avoid exposing member data?
  - Does the class hide its implementation details from other classes as much as the programming language permits?
  - Does the class avoid making assumptions about its users, including its derived classes?
  - Is the class independent of other classes? Is it loosely coupled?
- Inheritance
  - Is inheritance used only to model “is a” relationships — that is, do derived classes adhere to the Liskov Substitution Principle?
  - Does the class documentation describe the inheritance strategy?
  - Do derived classes avoid “overriding” non-overridable routines?
  - Are common interfaces, data, and behavior as high as possible in the inheritance tree?
  - Are inheritance trees fairly shallow?
  - Are all data members in the base class private rather than protected?

- Steve McConnell, Code Complete 2

Code Complete 2 – Class Construction Guidelines

- Other Implementation Issues
  - Does the class contain about seven data members or fewer?
  - Does the class minimize direct and indirect routine calls to other classes?
  - Does the class collaborate with other classes only to the extent absolutely necessary?
  - Is all member data initialized in the constructor?
  - Is the class designed to be used as deep copies (object and references) rather than shallow copies (just the object) unless there's a measured reason to create shallow copies?
- Language-Specific Issues
  - Have you investigated the language-specific issues for classes in your specific programming language?

- Steve McConnell, Code Complete 2

Summary

- Principles of Design Patterns
  - We've now encountered several OO design principles
  - Looked at how they are applied in certain cases
  - Cautioned against an over reliance on patterns
    - They are useful but they can't be your only hammer
    - They are one tool among many in performing OO A&D and general quality software development
- Code Complete 2 Guidelines for Class Construction

- Program to interfaces not implementations
- Encapsulate what varies
- A class should have only one reason to change
- Classes are about behavior, not specialization
- Favor delegation (composition) over inheritance
- Strive for loosely coupled designs between objects that interact
- Only talk to your (immediate) friends (Law of Demeter, Principle of Least Knowledge)
- Don't call us, we'll call you (the Hollywood Principle)
- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle - Classes should be open for extension, but closed for modification
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle - Depend on abstractions, not concrete classes
- Don't repeat yourself (DRY Principle)
- You Aren't Going to Need It (YAGNI Principle)
- The Principle of Healthy Skepticism