Software Design Principles

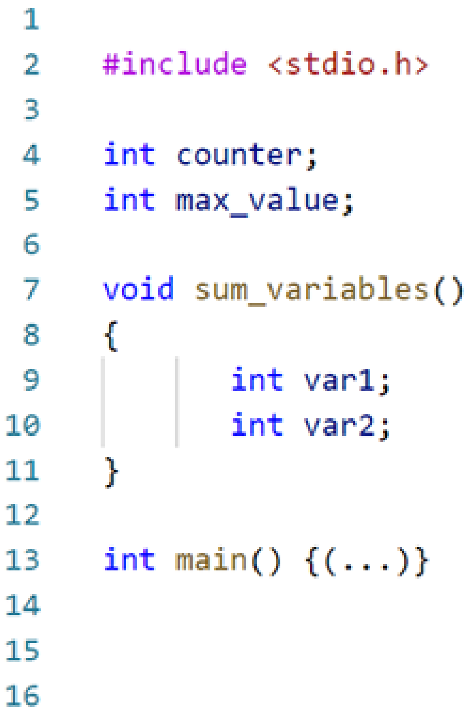
Program Design

Outline

- Overall Program Structure
- Modules and Functions
- Cohesion and Coupling
- Functional and Object Oriented Design
- Inheritance and Composition

Overall Program Structure

A C language Example



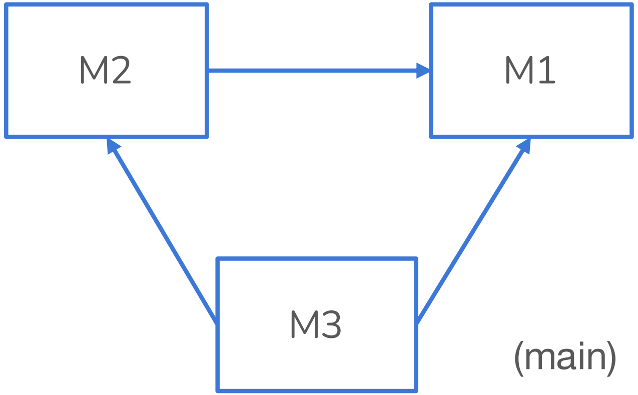
```

#include <stdio.h>

int counter;
int max_value;

void sum_variables()
{
        int var1;
        int var2;
}

int main() {(...)}



```

Global variables

Modules

Main function

Overall Program Structure - Example



Design Considerations

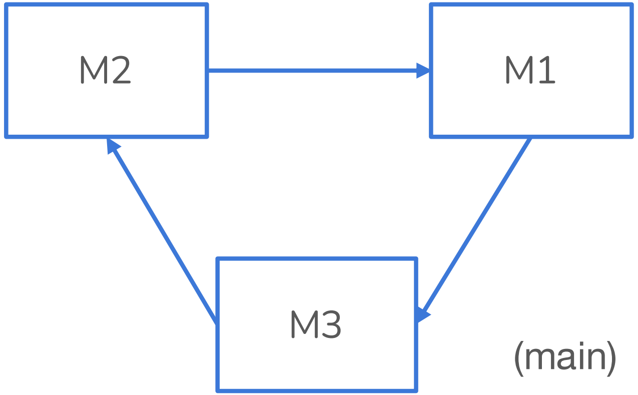
Interdependency

- M3 testing requires both M2 and M1
- M2 testing requires M1
- M1 can be tested alone

Order of Development

1. M1
2. M2
3. M3

Overall Program Structure - Example

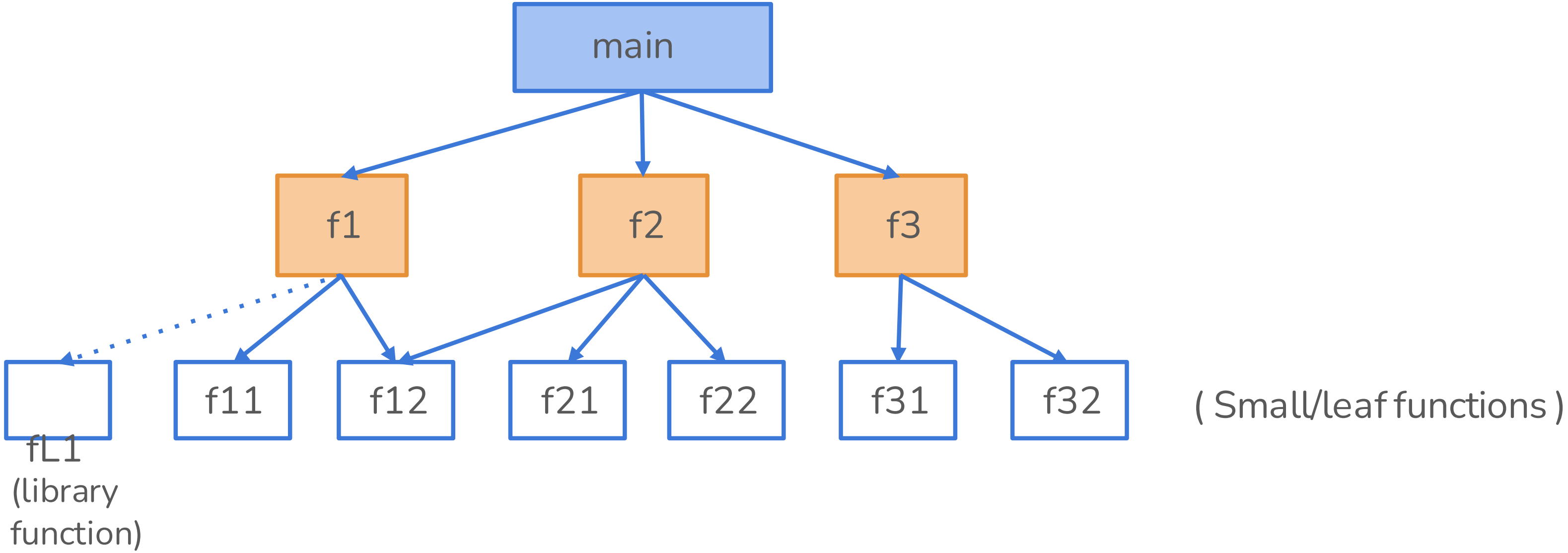


Bad Design Structure!

Why?

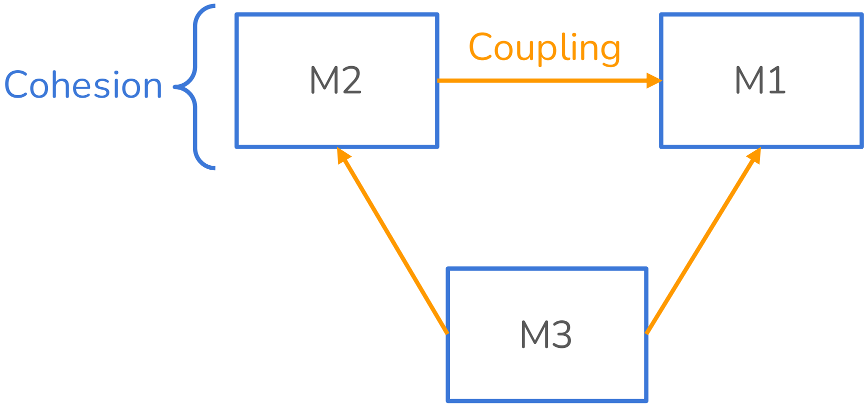
- All three modules are needed to test any functionality
- Interdependency forces joint module development

Functions



How do functions interact with each other?

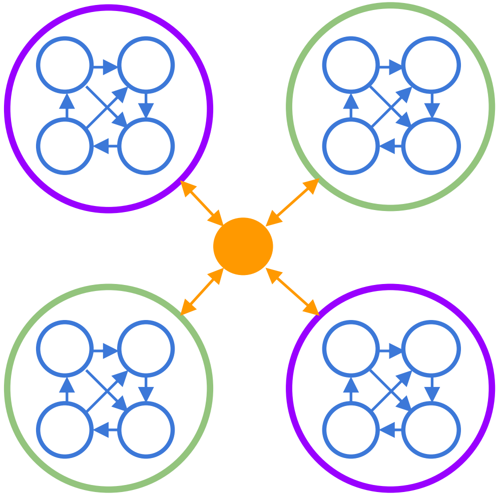
Cohesion and Coupling



Cohesion - A module should have only related functions within

Coupling - Modules should be well coupled with each other

Cohesion and Coupling



Cohesion - Strength of relationship between elements, within a module

Coupling - Interdependence / degree of closeness between modules

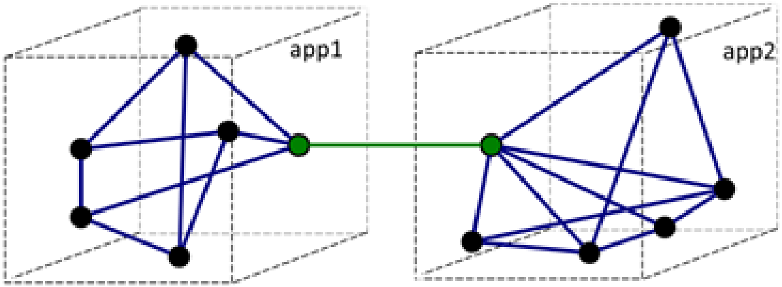
Good Program Design

- ➔ Internal integrity : Strong Cohesion
- ➔ Interface complexity : Loose Coupling

How to Design and Measure ?

Cohesion

Good Cohesion



Bad Cohesion

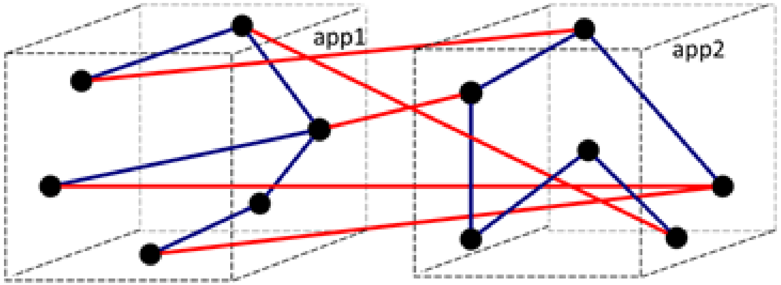
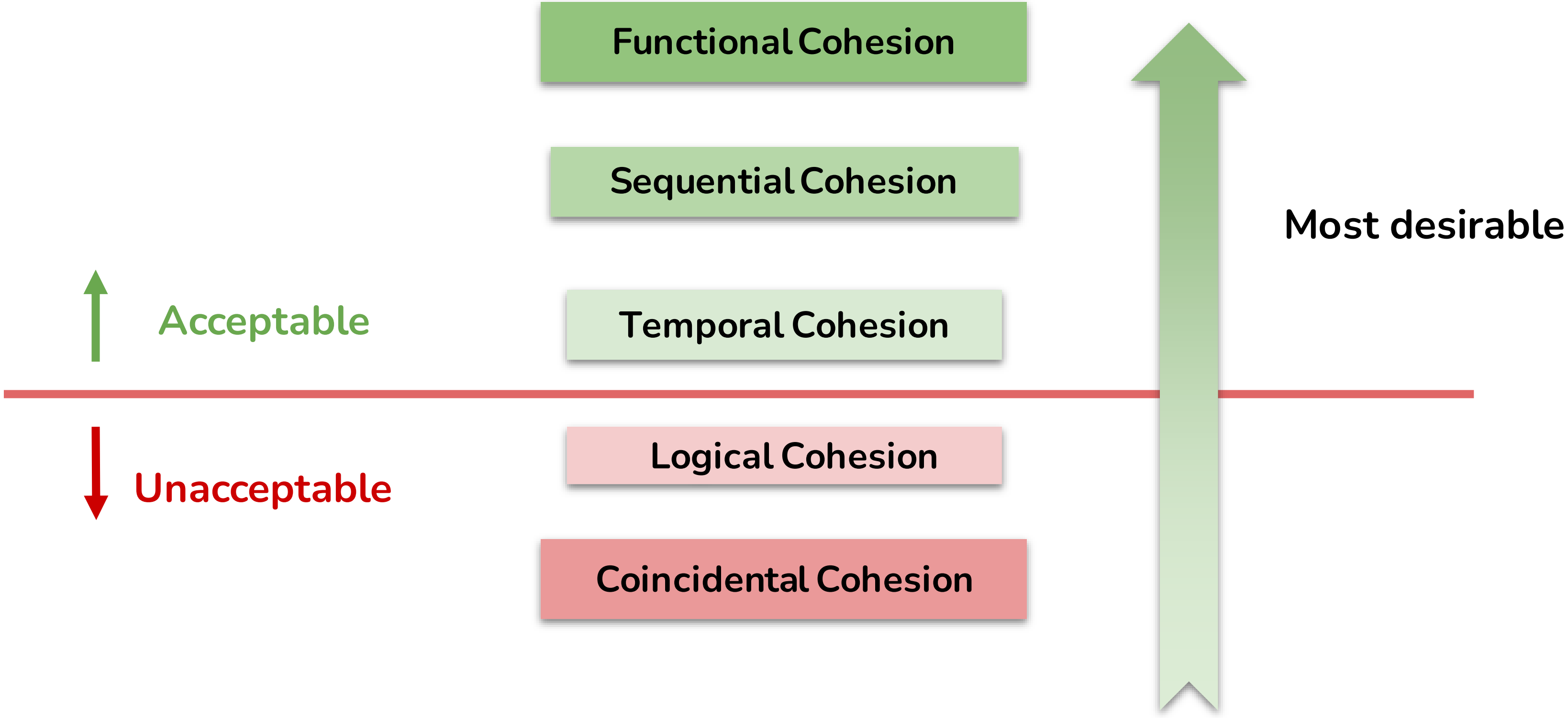


image credits-commons.wikimedia
https://upload.wikimedia.org/wikipedia/commons/b/bc/Good%2C_bad_apps.png

Types of Cohesion



Functional Cohesion

Sequential Cohesion

Temporal Cohesion

Logical Cohesion

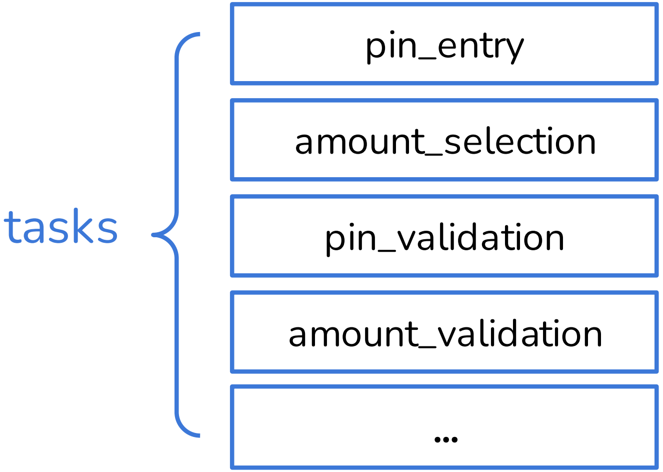
Coincidental Cohesion

Most desirable

Acceptable

Unacceptable

Cohesion in a module

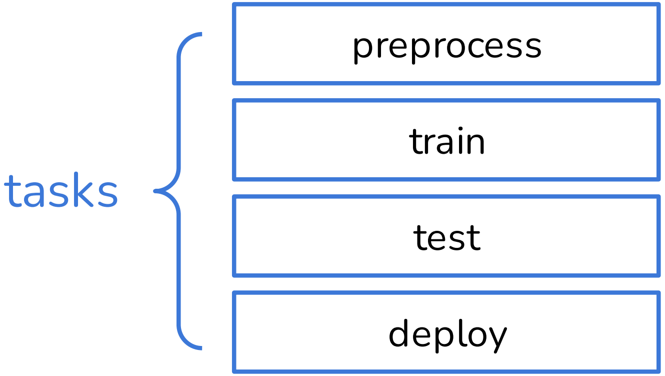


Functional Cohesion

Various parts contribute to a well defined task

Example - ATM withdrawal flow

Cohesion in a module

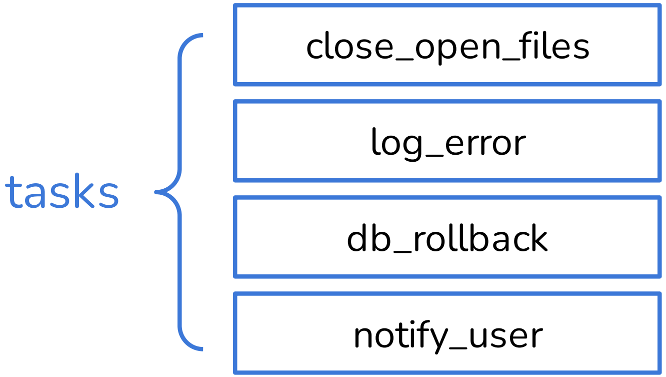


Sequential Cohesion

Assembly line flow of tasks with data passing through

Example - A machine learning module with shared data

Cohesion in a module


tasks

close_open_files

log_error

db_rollback

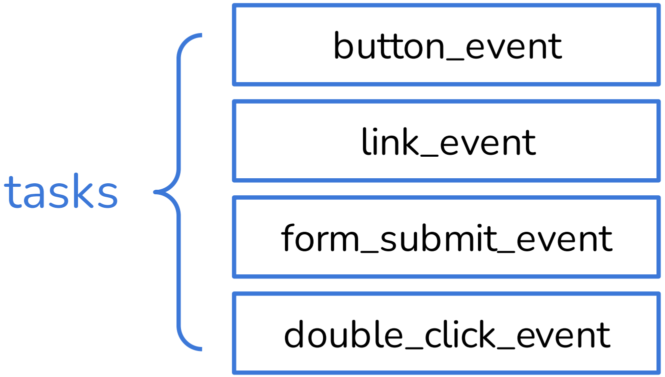
notify_user

Temporal Cohesion

Unrelated tasks executed together due to the same trigger

Example - Code exception

Cohesion in a module



tasks

button_event

link_event

form_submit_event

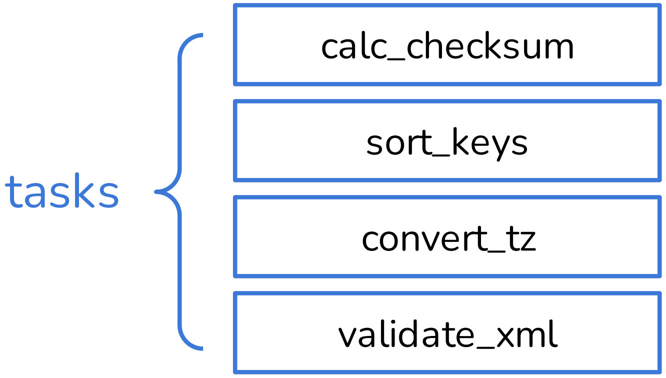
double_click_event

Logical Cohesion

Logically similar but functional unrelated tasks

Example - Event handling of various html DOM elements

Cohesion in a module



tasks

calc_checksum

sort_keys

convert_tz

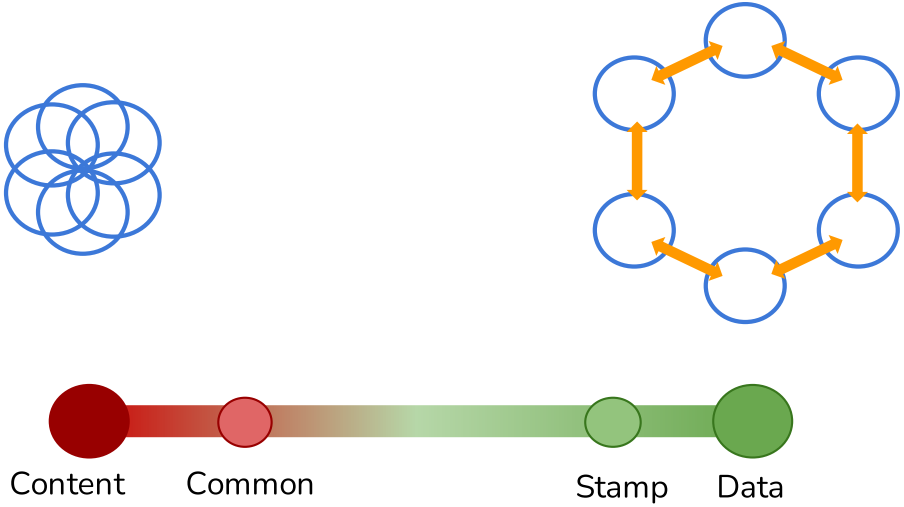
validate_xml

Coincidental Cohesion

Unrelated tasks grouped together arbitrarily

Example - Utility module

Types of Coupling



More Interdependency

More Coordination

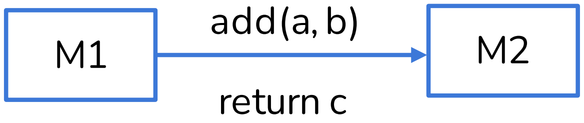
More Information Flow

Less Interdependency

Less Coordination

Less Information Flow

Types of Coupling

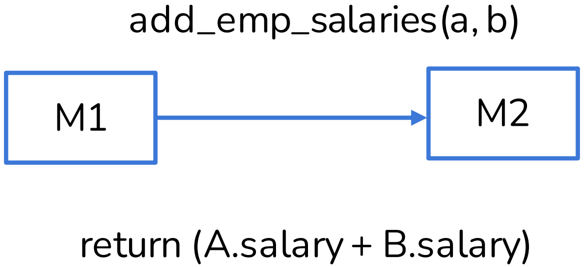


Data Coupling

Communication is through a parameter list

Minimal information shared

Types of Coupling

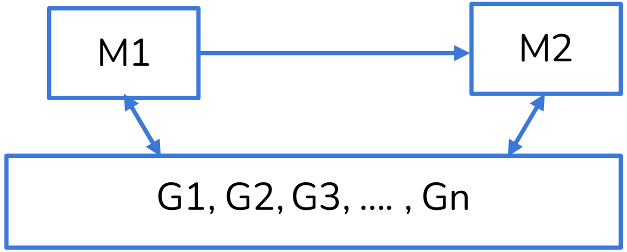


Stamp Coupling

Passing whole object when only specific elements are needed

Unnecessary object passing and structure knowledge

Types of Coupling

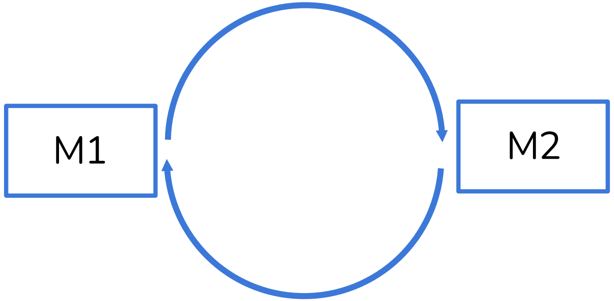


Common Coupling

Modules can access global variables and modify them

Unknown dependencies outside of direct interaction

Types of Coupling



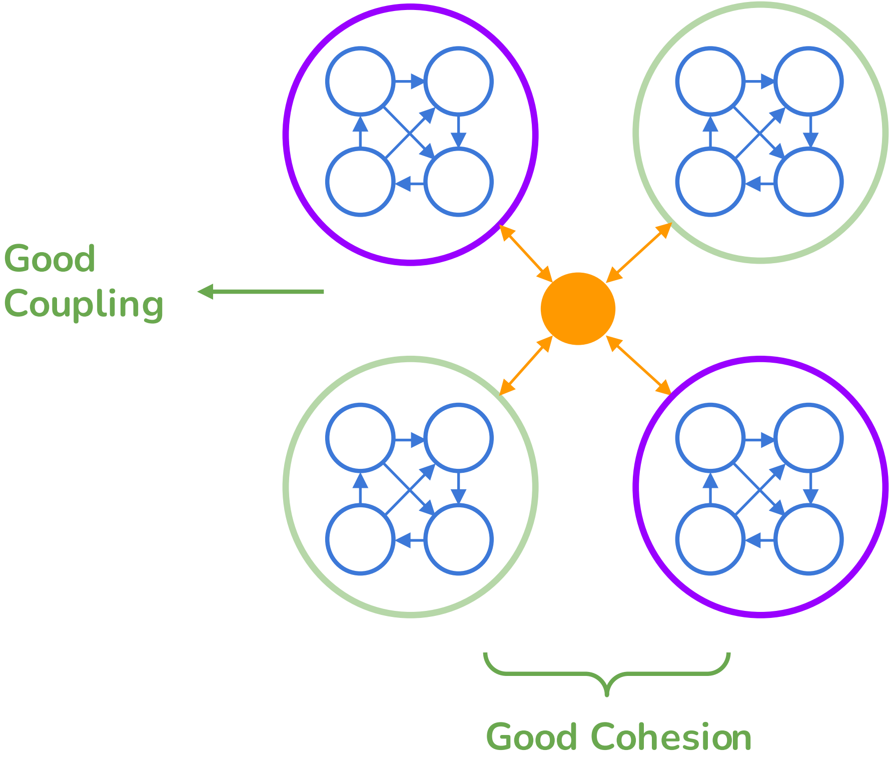
Content Coupling (Pathological)

M1 refers to M2’s local data or branches into M2

Worst form of coupling, breaks multiple design principles

Good Program Structure Design

- ✓ **Hierarchical** Modular structure
- ✓ **Tight Cohesion** within a module
- ✓ **Loose Coupling** between the modules



Design Approaches

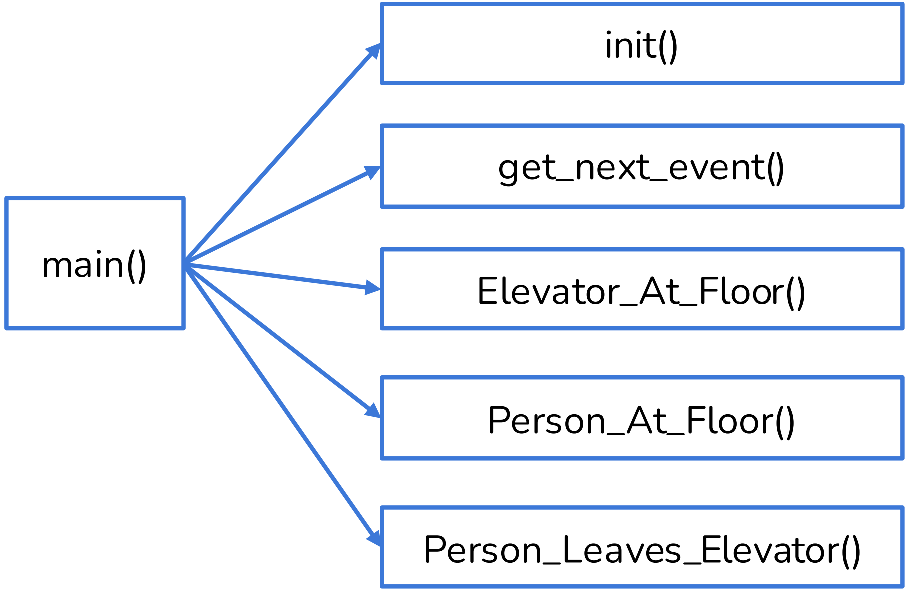
- Procedural Design
- Object Oriented Design

Design Approaches - Elevator Simulation

Let’s try designing an Elevator

Design Approaches - Elevator Simulation

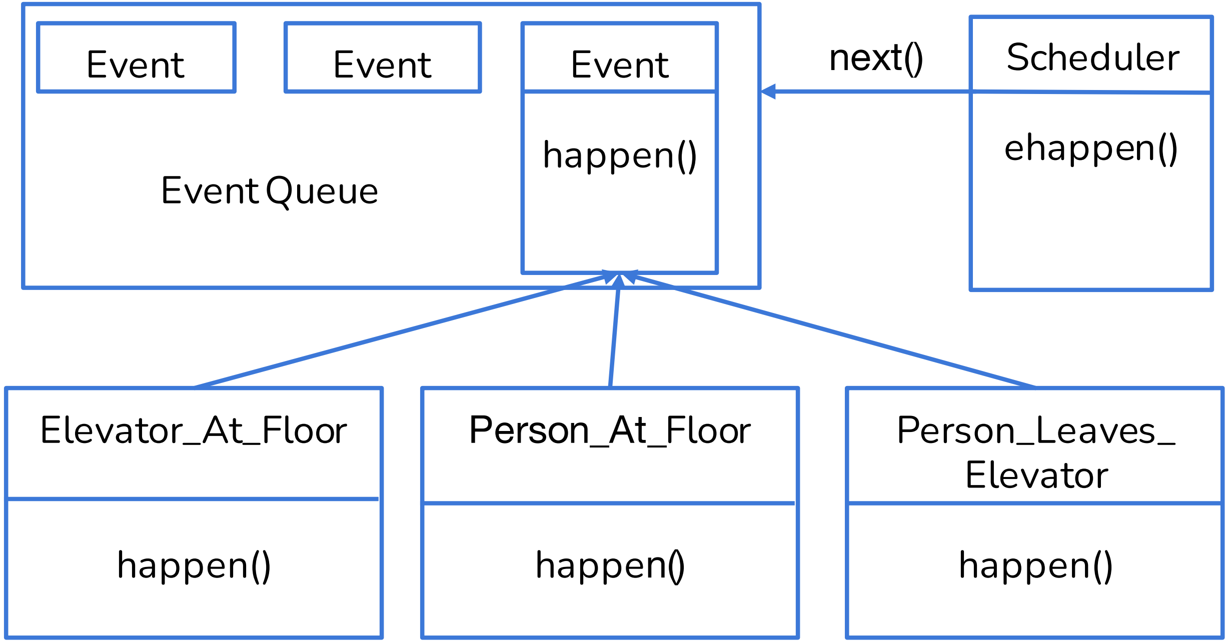
Procedural Design



- Interdependent business logic, lacks clarity
- High maintenance cost
- Extension requires multiple changes

Design Approaches - Elevator Simulation

Object Oriented Design



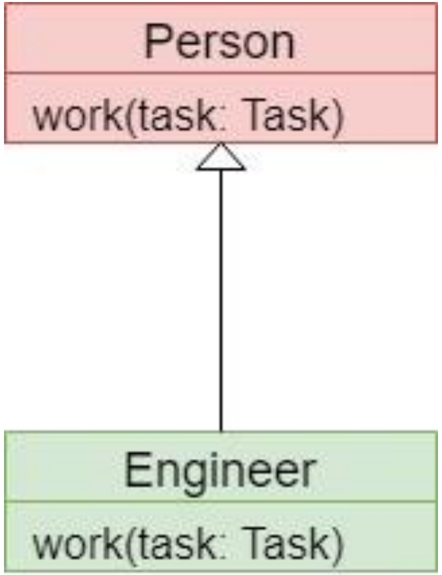
- Interdependent discrete events
- Easily extensible
- Clarity in Business Logic

Inheritance vs Composition

Inheritance

- Inheritance: **IS-A** relationship
- Example: An Engineer **IS-A** Person

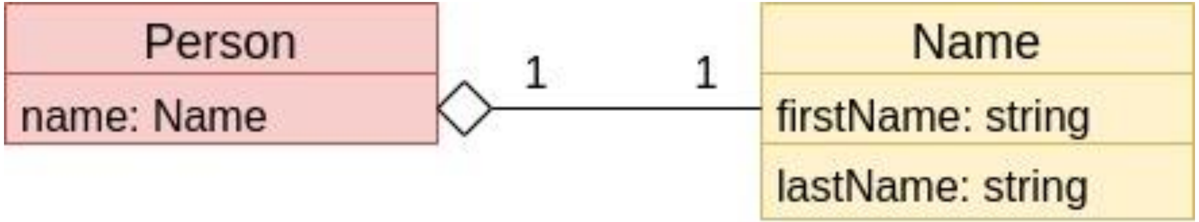
- Compile-time dependency



Inheritance vs Composition

Composition

- Composition: **HAS-A** relationship
- Example: Person **HAS-A** name



- Run-time relationship

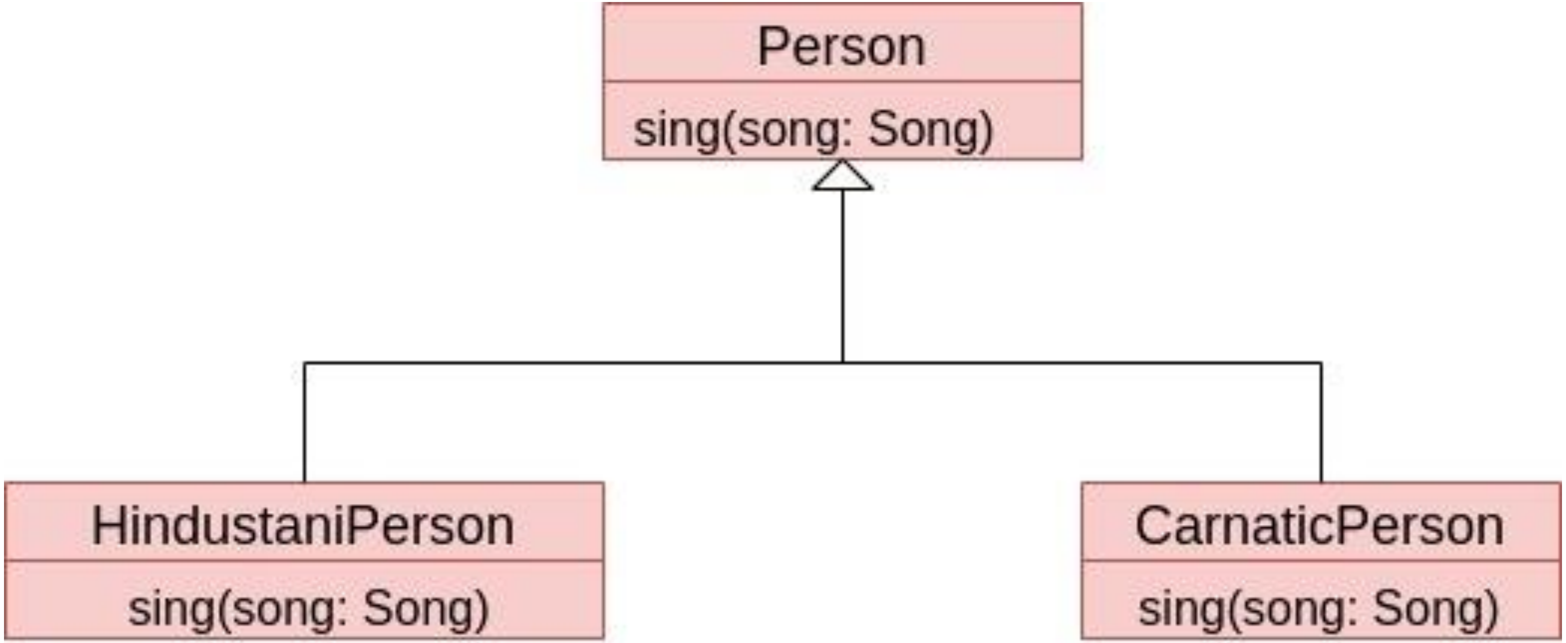
Inheritance vs Composition

Inheritance and Composition Example

- ➔ A Person named Laxman can sing both Carnatic and Hindustani.
- ➔ **Goal:** Allow Laxman to sing either Carnatic or Hindustani, at run-time.
- ➔ **How to model?**

Inheritance vs Composition

Just Inheritance?



Person

sing(song: Song)

HindustaniPerson

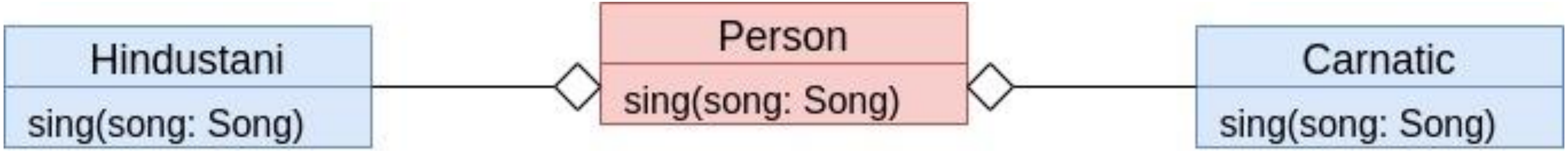
sing(song: Song)

CarnaticPerson

sing(song: Song)

Inheritance vs Composition

Just Composition?



Inheritance vs Composition

Classes should achieve polymorphism and code reuse using composition rather than inheritance.

IS-A

[Laxman **IS A** Singer]

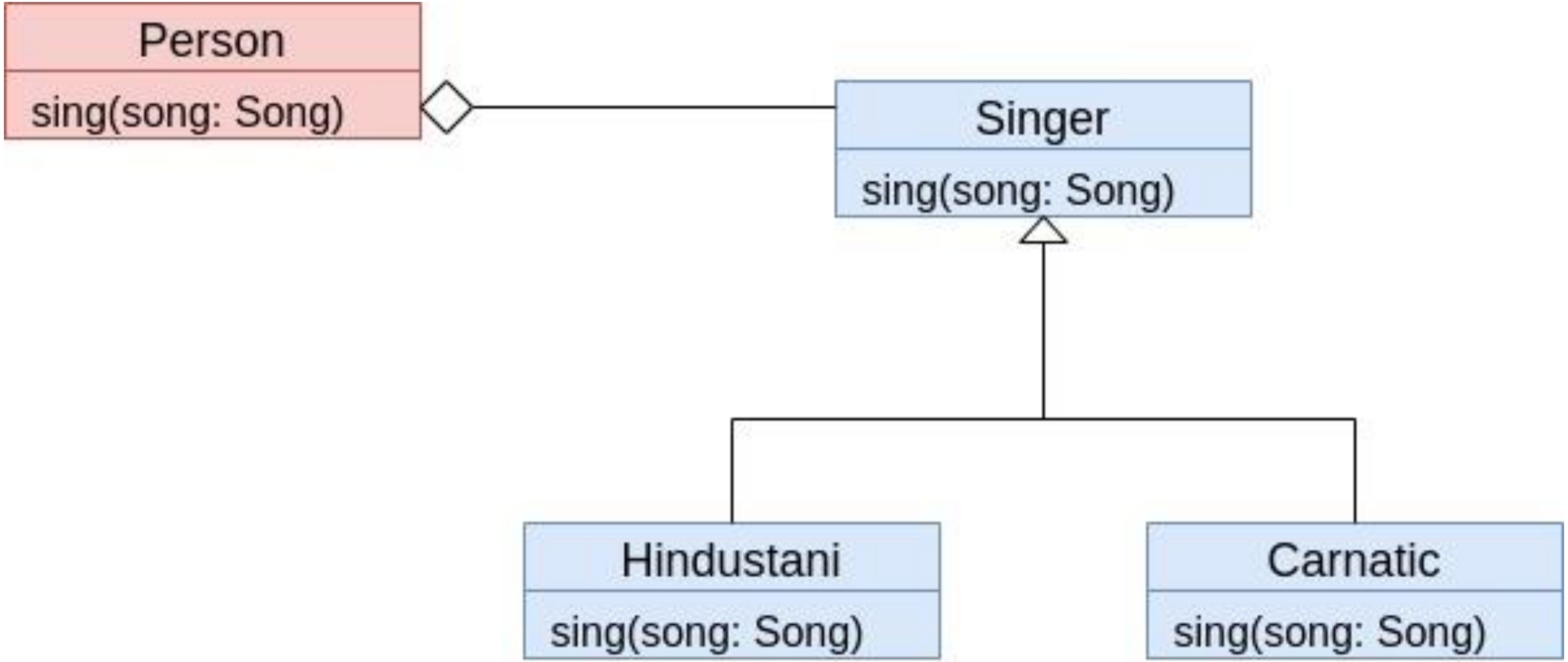
vs

HAS-A

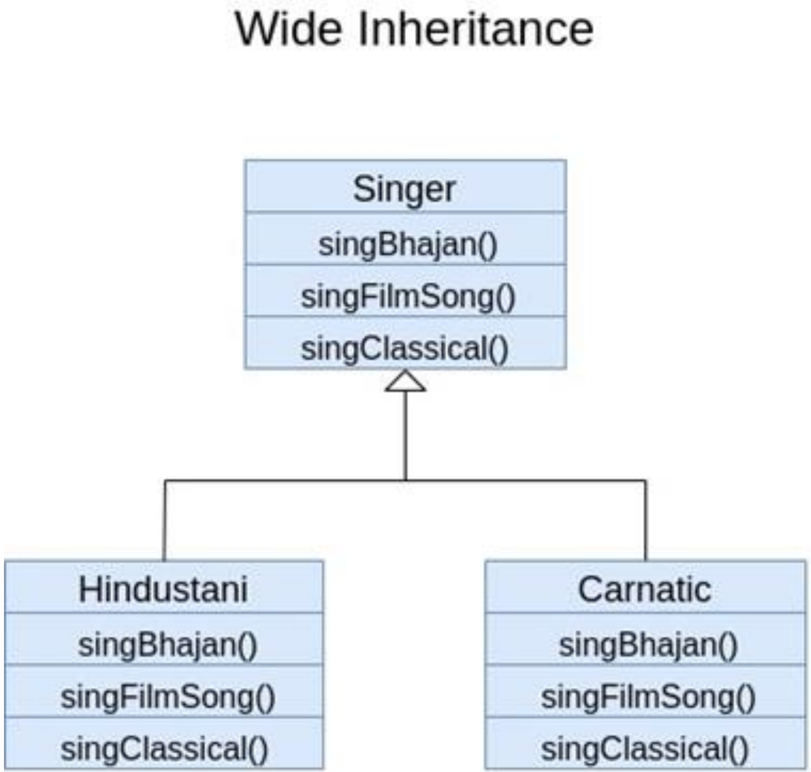
[Laxman **HAS A** skill of Singer]

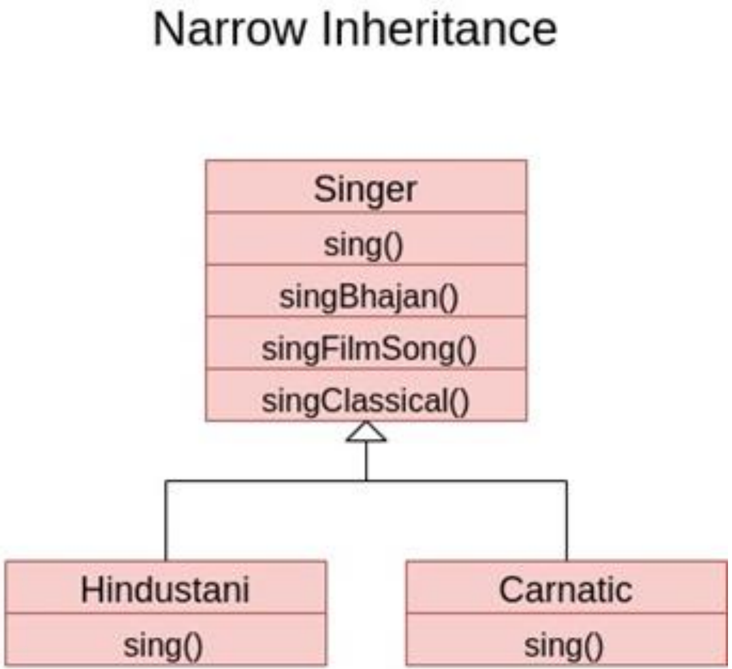
Inheritance vs Composition

Inheritance & Composition - Let’s use Both



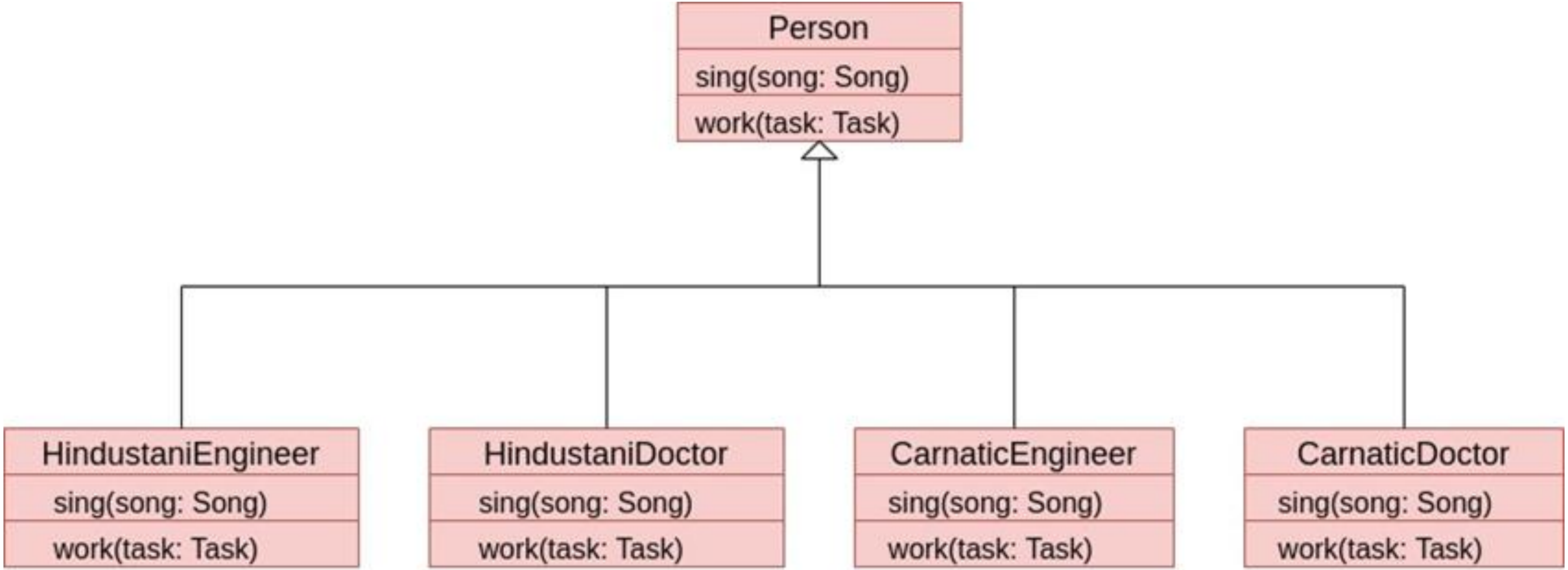
Wide And Narrow Inheritance





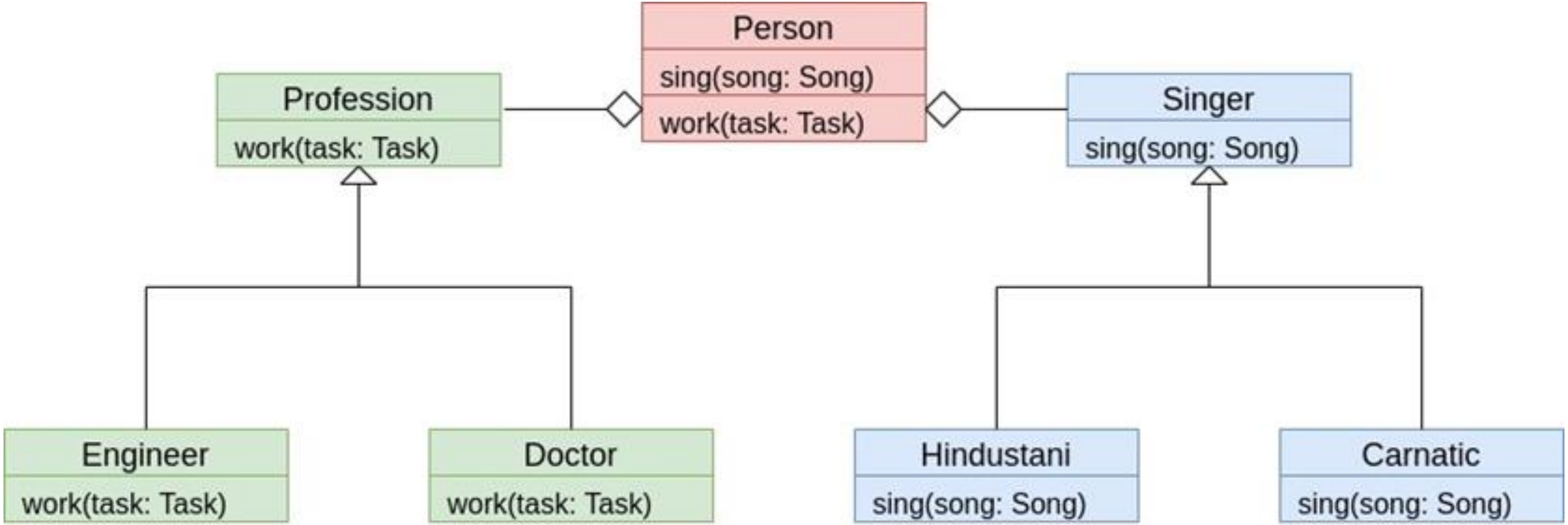
Inheritance

Poor Inheritance Design



Inheritance & Composition

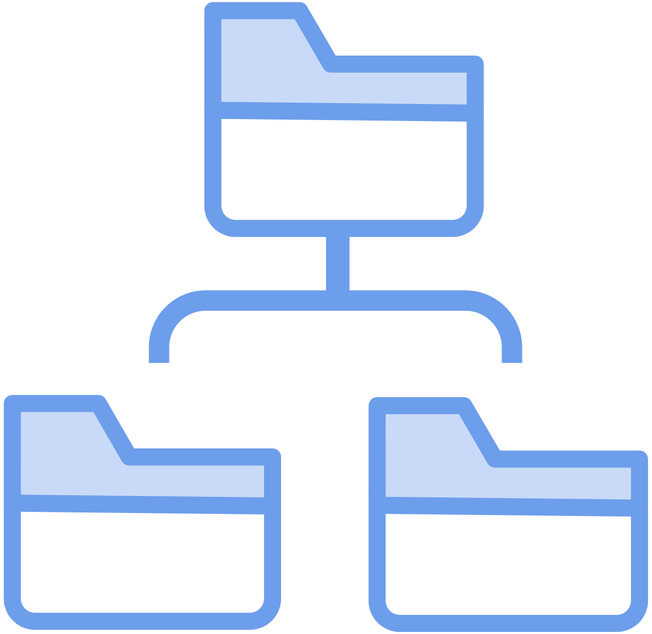
Multiple Hierarchies



Inheritance & Composition

Modeling Folders

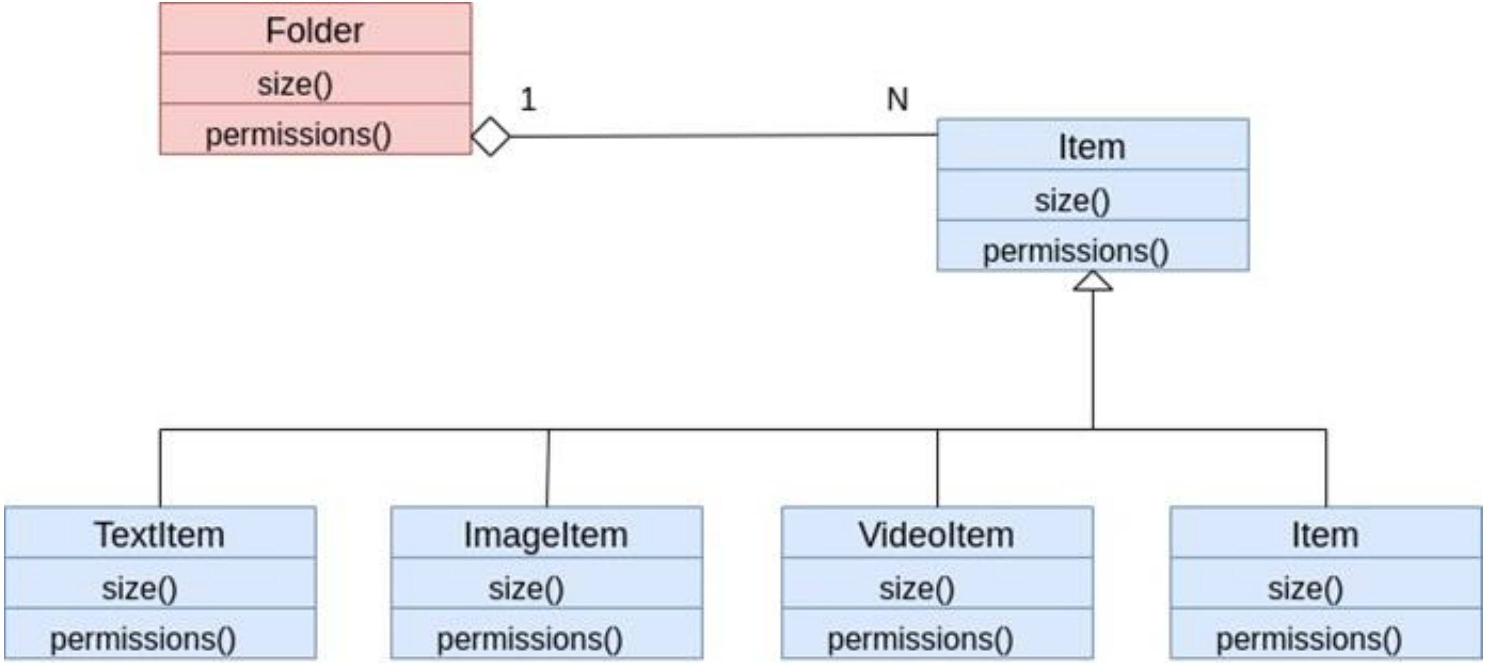
- A Folder could contain
  - Text / Image / Video Item
  - A Slideshow
  - More Folders
- Each Item has:
  - A thumbnail icon
  - A set of access apps
- How to Model?



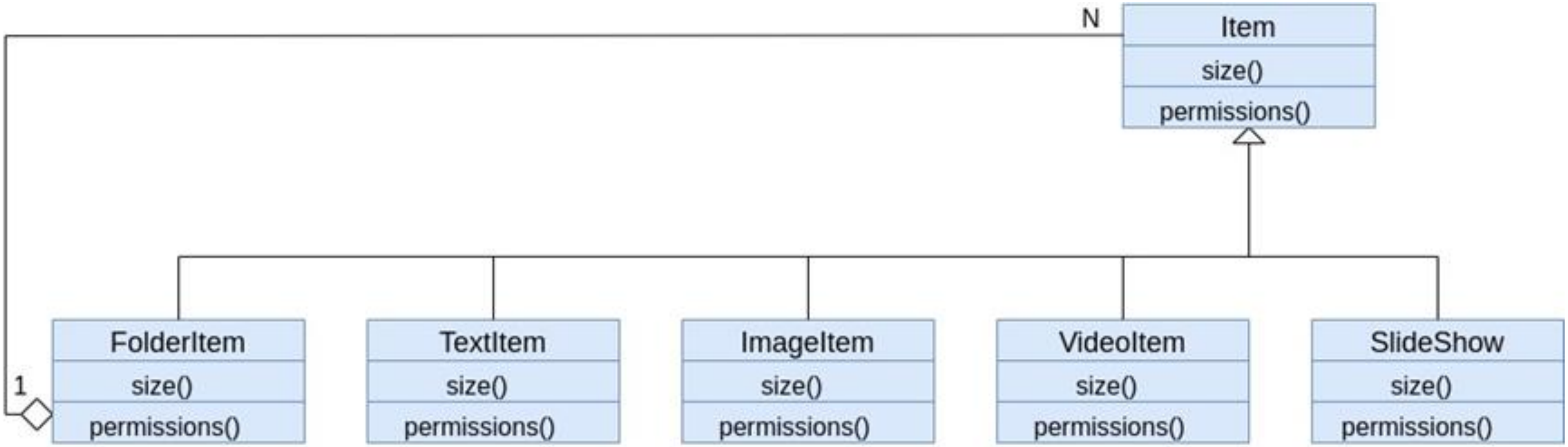
Inheritance & Composition

Modeling Folders

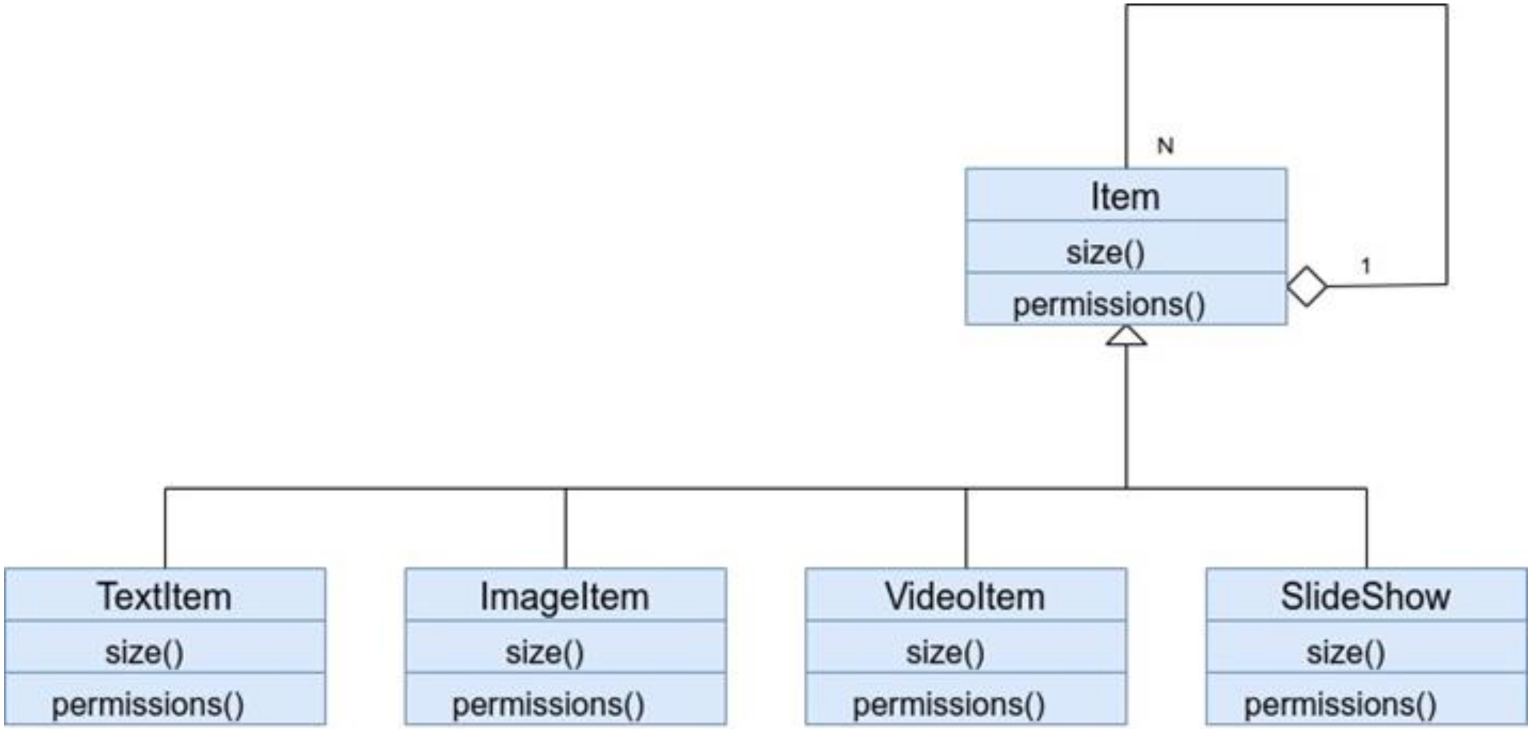
What about Sub-folders?



Inheritance & Composition



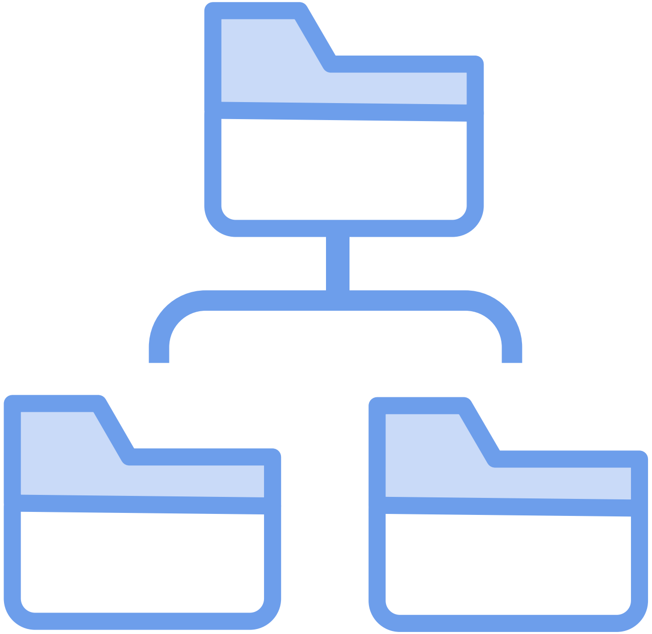
Inheritance & Composition



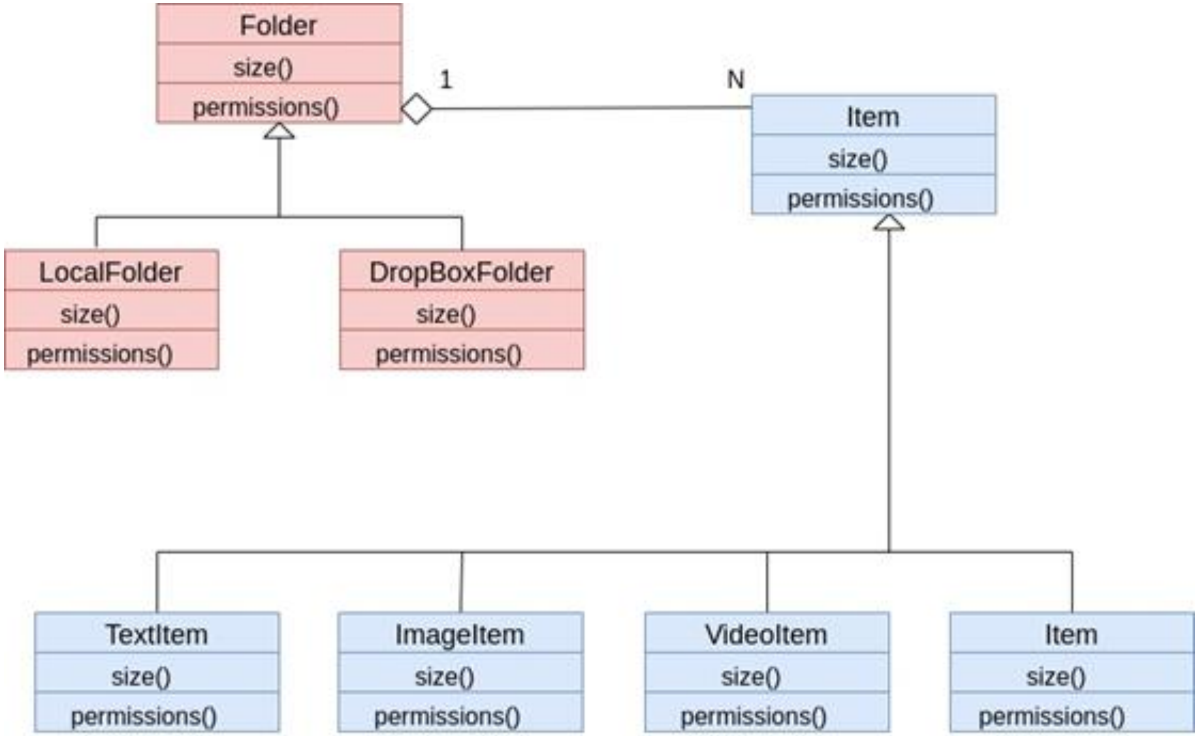
Inheritance & Composition

More Meta Patterns

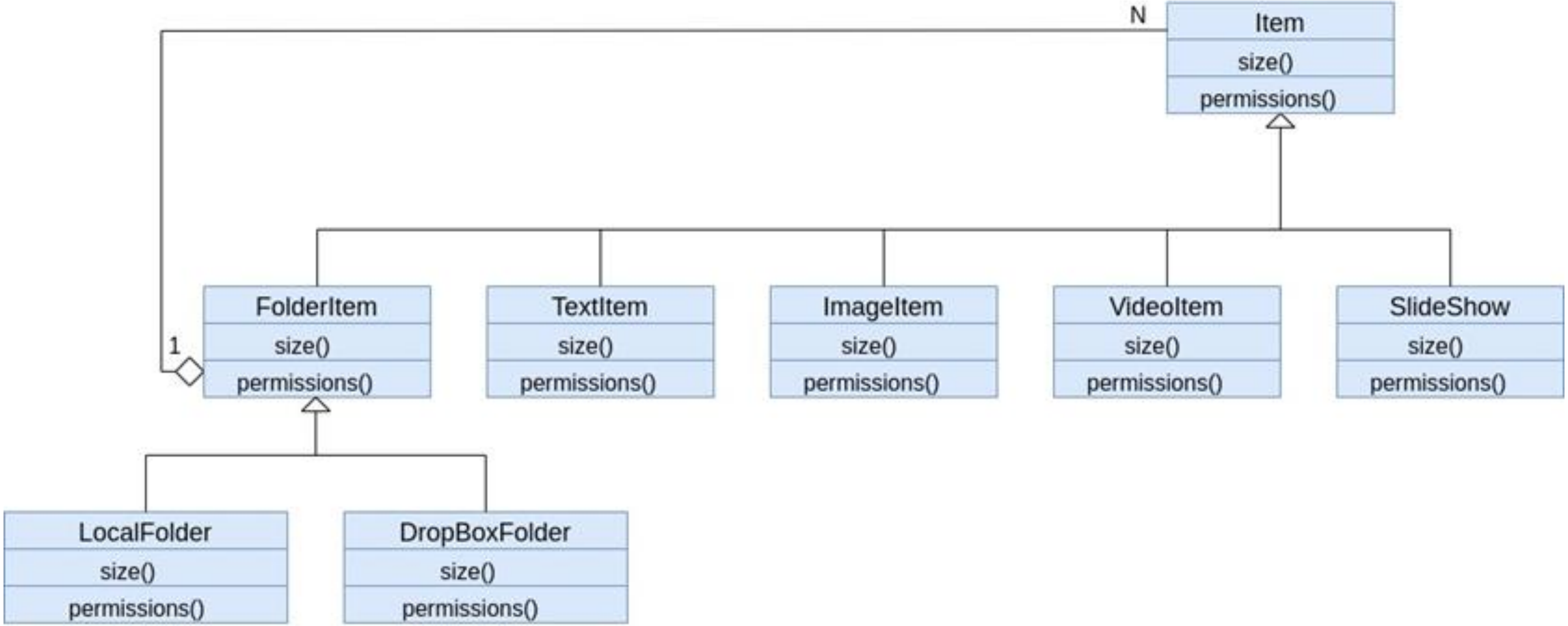
- **So far:** a fixed type object composed of varying object types
  - A folder consisting of items
- **Next:** Let’s vary the composite object. Folder could be a
  - Local Folder
  - Dropbox Folder
  - Network Folder



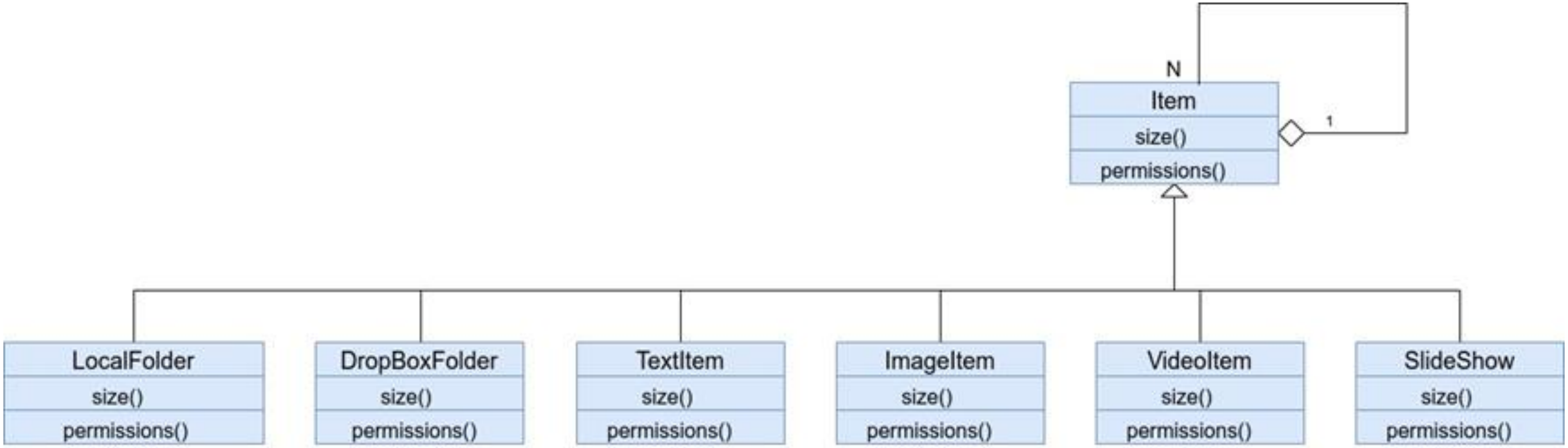
Inheritance & Composition



Inheritance & Composition



Inheritance & Composition



Attribution

Image Credits:

1) Pixabay: https://rb.gy/ez9syv
2) Shutterstock: https://www.shutterstock.com/image-illustration/software-architect-designing-uml-model-on-287660663
3) Pixabay: https://pixabay.com/photos/elevator-berlin-ludwig-erhard-haus-1598431/
4) Shutterstock: https://www.shutterstock.com/image-vector/vector-illustration-south-indian-man-singing-1074997097

SOLID Design Principles

Agenda

- Design Principles
- Why is design important?
- Impact of improper design
- SOLID Principles with code walkthroughs
    - SRP
    - OCP
    - LSP
    - ISP
    - DIP

Design principles

Design principles are **universally** applicable concrete design rules to **guide** a developer, designer, or architect during **design conceptualization or maintenance** phases

Why is design important?

Why is design important?

Design is important because

- It enables structured problem solving approach
- Provides elegant solution
- It is the basis of the functional product

Improper design can cause…

- **Rigidity** -Single change causes cascade of changes

- **Fragility** - Single changes causes breaks in many other, often unrelated areas. Modules constantly in need of repair, always on the bug list.

- **Immobility** - Contains parts that could be useful elsewhere, but too much work to separate those parts. Very common!

- **Needless Repetition** - Cut-and-paste can be disastrous code-editing operations! Miss an abstraction, miss an opportunity to make system easier to understand and maintain.

- **Opacity** - Tendency to be difficult to understand. Code seems clear when first written. Later even same developer may not understand it. Code reviews! Refactor as you go!

SOLID Principles

Single Responsibility Principle : A class should change for only one reason or it should have only one responsibiltiy

Open/Closed Principle : Software (classes, modules, functions, etc.) can only be open for adding new extensions, but it should be closed for changes or modifications.

Liskov Substitution Principle: Subtypes should be replaceable for their base types.

Interface Segregation Principle : Methods or method dependencies should not forced on clients that they do not intend to use.

Dependency Inversion Principle: There should be no dependency of high-level modules on low-level modules. Instead, both should depend on abstractions.

- Abstract elements should not be dependent on details. Rather, details should depend on abstractions.

SRP: The Single-Responsibility Principle

Cohesion

- We want our classes to be “cohesive” – but what does that mean?
- One way to make specific: Single Responsibility Principle (SRP)
- Functions that change together, exist together.

Single Responsibility – Example (Modem)

```
class Modem:
   def connect(self, ph_no):
      pass
   def disconnect(self):
      pass
   def send(self, data):
      pass
   def recv(self):
      pass
```

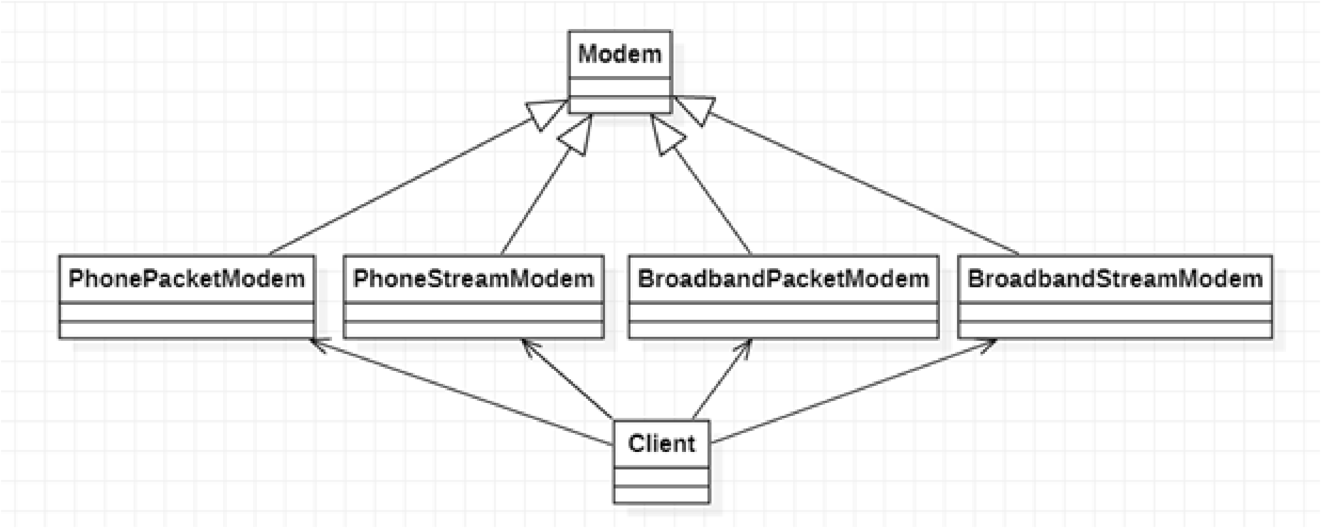
Seems like good set of modem functions – but is it one or two sets of responsibilities?

- Connection management (connect and disconnect)
- Data communication (send and recv)

Separate if connection functions are changing – otherwise have rigidity.

Don’t separate if no changes expected – otherwise have complexity.

Single Responsibility – Example (No SRP Modem UML)



Single Responsibility – Example (No SRP Modem)

```
class PhonePacketModem(Modem):
    def connect(self, ph_no):
        print("PhonePacketModem: Dial phone")
    def disconnect(self):
        print("PhonePacketModem: Hangup phone")
    def send(self, data):
        print("PhonePacketModem: Send packet")
    def recv(self):
        print("PhonePacketModem: Receive packet")
```

```
class PhoneStreamModem(Modem):
    def connect(self, ph_no):
        print("PhoneStreamModem: Dial phone")
    def disconnect(self):
        print("PhoneStreamModem: Hangup phone")
    def send(self, data):
        print("PhoneStreamModem: Send stream")
    def recv(self):
        print("PhoneStreamModem: Receive stream")
```

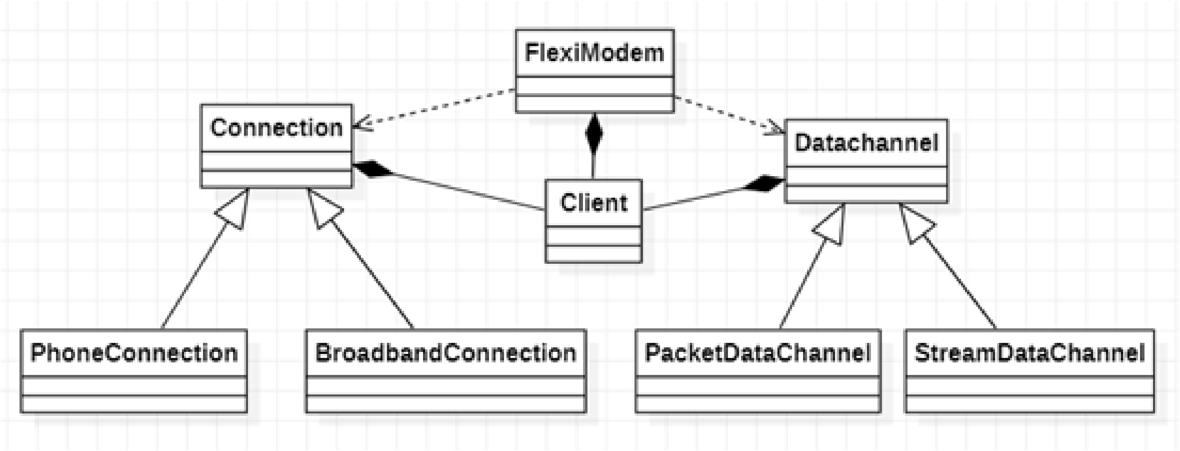
```
class BroadbandPacketModem(Modem):
    def connect(self, ph_no):
        print("BroadbandPacketModem: Dial Broadband")
    def disconnect(self):
        print("BroadbandPacketModem: Hangup Broadband")
    def send(self, data):
        print("BroadbandPacketModem: Send packet")
    def recv(self):
        print("BroadbandPacketModem: Receive packet")
```

```
class BroadbandStreamModem(Modem):
    def connect(self, ph_no):
        print("BroadbandStreamModem: Dial Broadband")
    def disconnect(self):
        print("BroadbandStreamModem: Hangup
Broadband")
    def send(self, data):
        print("BroadbandStreamModem: Send stream")
    def recv(self):
        print("BroadbandStreamModem: Receive stream")
```

When new connections and type of communications are added we see following issues

- Code duplication where connection and communication types are same.
- Increased rigidity and reduced flexibility in design. (the entire code has to be compiled for similar connections or communication type modems)

Single Responsibility – Example (SRP Modem UML)



Single Responsibility – Example (SRP Modem)

```
class Connection:
    def connect(self, ph_no):
        pass
    def disconnect(self):
        pass
```

```
class PhoneConnection(Connection):
    def connect(self, ph_no):
        print("PhoneConnection:  Dial  phone")
    def disconnect(self):
        print("PhoneConnection:  Hangup  phone")

class BroadbandConnection(Connection):
    def connect(self, ph_no):
        print("BroadbandConnection:  Dial Broadband")
    def disconnect(self):
        print("BroadbandConnection:  Hangup  Broadband")
```

```
class Datachannel:
    def send(self, ch_data):
        pass
    def recv(self):
        pass
```

```
class PacketDataChannel(Datachannel):
    def send(self, ch_data):
        print("PacketDataChannel:  Send packet")
    def recv(self):
        print("PacketDataChannel:  Receive  packet")

class StreamDataChannel(Datachannel):
    def send(self, ch_data):
        print("StreamDataChannel:  Send packet")
    def recv(self):
        print("StreamDataChannel:  Receive  packet")
```

Single Responsibility – Example (SRP Modem)

```
class FlexiModem:
    def __init__(self, Connection, Datachannel):
        self.connection = Connection
        self.datachannel = Datachannel

    def do_connection(self):
        self.connection.connect("121.55.825.56")
        self.datachannel.send("Hello SRP")
        self.datachannel.recv()
        self.connection.disconnect()
```

```
if __name__ == '__main__':
    flexi_modem = []
    fm1 = FlexiModem(BroadbandConnection(), PacketDataChannel())
    fm2 = FlexiModem(BroadbandConnection(), StreamDataChannel())
    fm3 = FlexiModem(PhoneConnection(), PacketDataChannel())
    fm4 = FlexiModem(PhoneConnection(), StreamDataChannel())

    flexi_modem.append(fm1)
    flexi_modem.append(fm2)
    flexi_modem.append(fm3)
    flexi_modem.append(fm4)

    for fm in flexi_modem:
        fm.do_connection()
```

Single Responsibility – Summary

- The SRP is frequently used design principles in Software design. It can be applied at all levels and types of software entities. (like classes, software components, and microservices etc)
- The software entities (like classes, software components, and microservices etc) is allowed to have only one responsibility.
- This increases the cohesion, and reduces the technical coupling between software entities and reduces the need to change the code.

OCP: The Open-Closed Principle

Open-Closed Principle

Software (classes, modules, functions, etc.) can only be open for adding new extensions, but it should be closed for changes or modifications

- Open for extension – extend with new behaviors
- Closed for modification – behavior extension should not change source code or binary (.exe, DLL, .jar)

If single change causes cascade of changes, *refactor*. Further changes should be achieved by adding new code.

How to achieve this?

Procedural Shapes – non OCP

```
import enum
class ShapeType(enum.Enum):
    circle = 1
    square = 2

# square
class Square:
    def __init__(self, type, side, topleft):
        self.type = type
        self.side = side
        self.topleft = topleft

# clrcle
class Circle:
    def __init__(self, type, radius, center):
        self.type = type
        self.radius = radius
        self.center = center
```

```
# draw square
def draw_square():
     print("Drawing Square")

# draw circle
def draw_circle():
        print("Drawing Circle")

# drawing all shapes
def DrawAllShapes(shape):
   if shape == ShapeType.square:
        draw_sqaure()
   elif shape == ShapeType.circle:
        draw_circle()
```

```
if __name__ == '__main__':
    for shape in (ShapeType):
        DrawAllShapes(shape)
```

Can we add a Triangle shape without changing the existing code? NO!!!

Procedural Shapes – non OCP

```
import enum
class ShapeType(enum.Enum):
    circle = 1
    square = 2
    triangle = 3 ←

# square
class Square:
    def __init__(self, type, side, topleft):
        self.type = type
        self.side = side
        self.topleft = topleft

# clrcle
class Circle:
    def __init__(self, type, radius, center):
        self.type = type
        self.radius = radius
        self.center = center

# triangle
class Triangle:
…
```

```
# draw square
def draw_square():
    print("Drawing Square")

# draw circle
def draw_circle():
    print("Drawing Circle")

# drawing all shapes
def DrawAllShapes(shape):
    if shape == ShapeType.square:
        draw_sqaure()
    elif shape == ShapeType.circle:
        draw_circle()
→   elif shape == ShapeType.triangle:
        draw_triangle()
```

```
if __name__ == '__main__':
    for shape in (ShapeType):
        DrawAllShapes(shape)
```

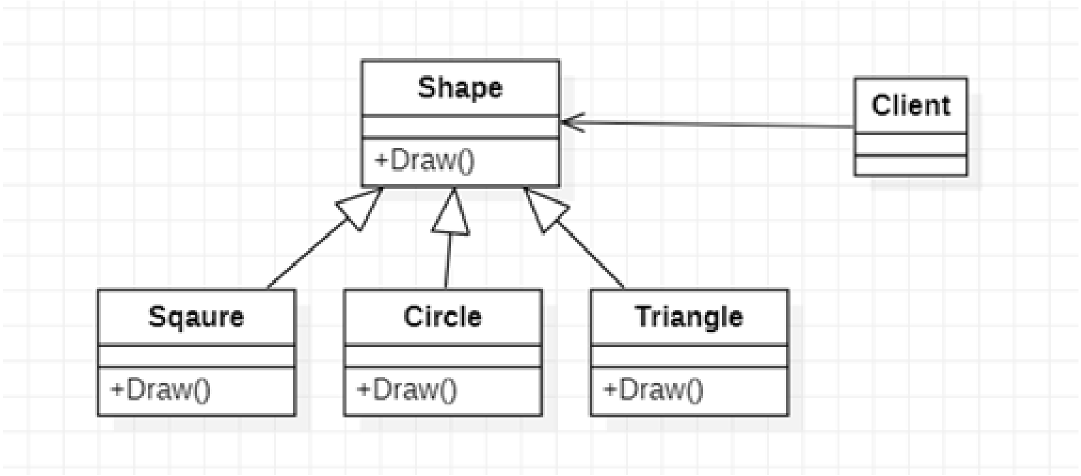
Existing enum must be modified
Existing DrawAllShapes must be modified
NOT CLOSED!!

(and a real system might have more switch statements to update)

Procedural Code - analysis

- Rigid – try to add a Triangle, must recompile Shape, Circle, Square, DrawAllShapes
- Fragile – many switch/case and if/else statements to understand and modify
- Immobile – try to reuse DrawAllShapes must include Square and Circle, even if application has none of those

OCP Shapes - UML



OCP Shapes

```
class Shape:
    def Draw(self):
        pass

# square
class Square(Shape):
    def Draw(self):
        print("Drawing Square")

# circle
class Circle(Shape):
    def Draw(self):
        print("Drawing Circle")
```

```
def DrawAllShapes(shapes):
    for shape in shapes:
        shape.Draw()
```

```
# triangle
class Triangle(Shape):
    def Draw(self):
        print("Drawing Triangle")
```

```
if __name__ == '__main__':
    shapes = []
    shapes.append(Square())
    shapes.append(Circle())

    shapes.append(Triangle())
    DrawAllShapes(shapes)
```

Changes required

No change to Shape class!
No change to Square or Circle!
No change to DrawAllShapes!

Adjust to change appropriately

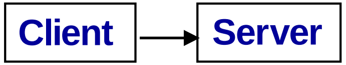
- When change happens, implement abstractions to protect from future changes *of that kind*.
- Modem example: might combine functions originally. As soon as need to change connection, add an abstraction (in this example, an interface).*

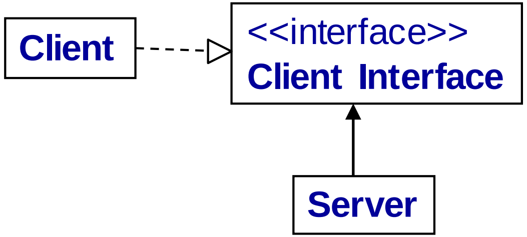
*** like DRY – do it the *first* time you repeat code. Here we switch to OCP the *first* time we need an extension.**

Abstraction is the Key!

Abstraction may be fixed
Behavior can be extended by creating new derivatives of the abstraction

The what, not the how





STRATEGY pattern

Behavior in Client can be extended and modified by creating new subtypes of ClientInterface.

OCP - Summary

Conformance to OCP yields great benefits

Just using an OO language does not guarantee OCP

Judicious use of abstraction is critical!

- apply abstraction where needed (as soon as you detect a possible future change)
- resist premature abstraction

LSP: The Liskov Substitution Principle

What makes a good inheritance hierarchy?

LSP: Subtypes must be substitutable for their base types.

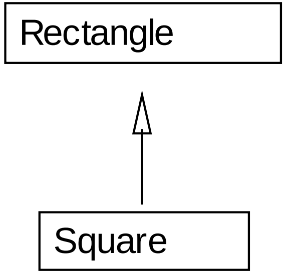
Meyers: Everything that is true of the base type must be true of the subtype.*

What if this were not true?

Methods that use base class references should be able to use objects of child classes without their knowledge.

Is a Square a Rectangle??

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def set_width(self, width):
        self.width = width
    def set_height(self, height):
        self.height = height
    def get_width(self):
        return width
    def get_height(self):
        return height
    def get_area(self):
        return height * width
```



Rectangle

Square

Issues:

- Square doesn’t need both itsWidth and itsHeight
- Having both SetWidth and SetHeight inappropriate

Naïve fixes do not work!

- We could override set_height and set_width. But since these methods aren't virtual*, so we have problem in these cases (do you understand why?!).
- If you are not still convinced, consider the following client code.

```
def g(rect):
   rect.set_width(5)
   rect.set_height(4)
   assert rect.get_area() == 20
```

```
class Square(shape):
   def __init__(self, width):
      self.width = width
   def get_width(self):
      return width
   def set_width(self, width):
      self.width = width
   def get_area(self):
      return self.width * self.width
```

The author of g assumed that changing the height of a rectangle would not change its width – a reasonable assumption for a rectangle!

Function g shows that there are functions that take references to Rectangle objects that do not operate properly on Square objects. So Square is NOT a valid substitute for a Rectangle. Making it a child violates the LSP principle!

* Meyers item 36: **Never redefine an inherited non-virtual function**.

Validity is not Intrinsic – Behavior Counts!

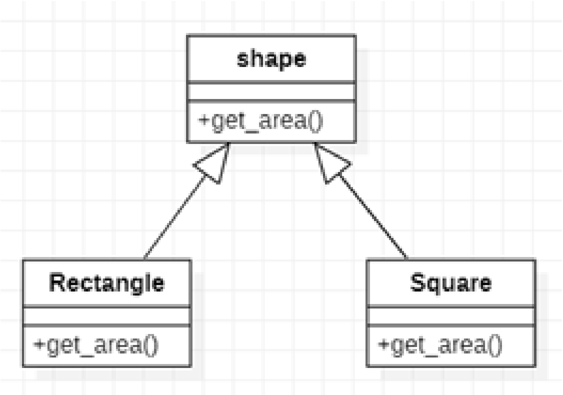
So an inheritance hierarchy cannot be evaluated abstractly. In the abstract sense, a square *is* a rectangle. But when we look at the *behavior* (as expressed in code), we see the issue.

"is-a" is only an approximate way to identify parent-child relationships.

So how do we fix this? Factoring!

If a set of classes all support a common responsibility, they should inherit that responsibility from a common superclass.

Factored to LSP!



```
class shape:
    def __init__(self):
        pass
    def get_area(self):
        pass
```

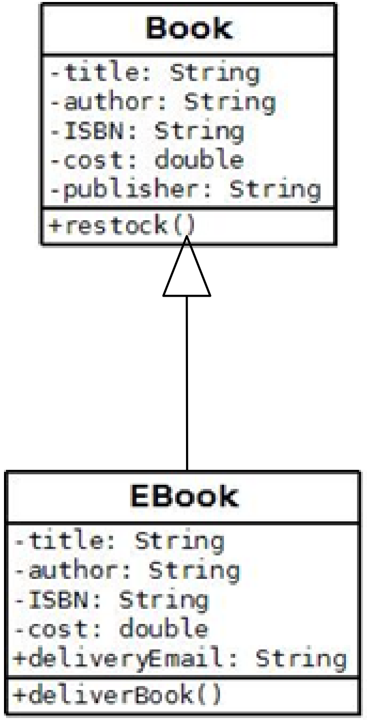
```
class Rectangle(shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def set_width(self, width):
        self.width = width
    def set_height(self, height):
        self.height = height
    def get_width(self):
        return width
    def get_area(self):
        return self.height * self.width
```

```
class Square(shape):
    def __init__(self, width):
        self.width = width
    def get_width(self):
        return width
    def set_width(self, width):
        self.width = width
    def get_area(self):
        return self.width * self.width
```

```
if __name__ == '__main__':
    rectangle = Rectangle(5,4)
    square = Square(5)
    print(rectangle.get_area())
    print(square.get_area())
```

More Examples!

Is an E-Book a Book?



Book

-title: String
-author: String
-ISBN: String
-cost: double
-publisher: String

+restock()

EBook

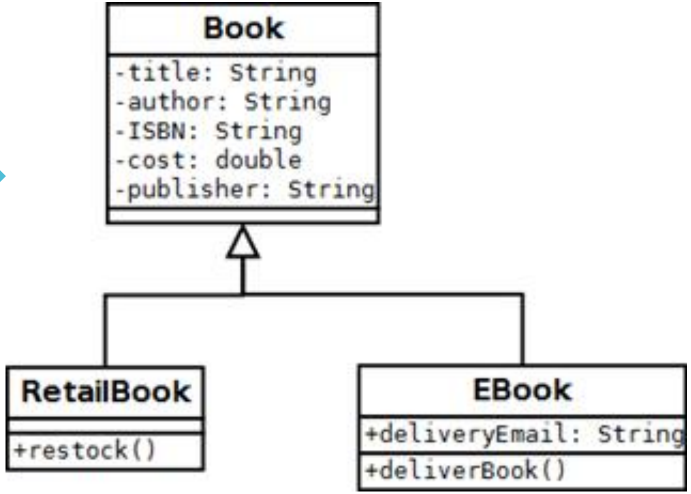
-title: String
-author: String
-ISBN: String
-cost: double
+deliveryEmail: String

+deliverBook()

Does restock apply to an Ebook? NO!

Is there commonality between these classes? YES!

What to do? FACTOR!



Book

-title: String
-author: String
-ISBN: String
-cost: double
-publisher: String

RetailBook

+restock()

EBook

+deliveryEmail: String

+deliverBook()

Heuristics

- A derivative that does *less* than its base is usually not substitutable for that base, and therefore violates LSP.

- If the users of the base class don’t expect exceptions, adding them to methods of the derivatives is not substitutable.

- IS-A is too broad to be a definition of a subtype. Substitutable is more appropriate, where substitutability is defined by an interface or contract.

is-a vs has-a

Assume you want to implement a Stack class. You already have a LinkedList class. Would you do:

a) public class Stack extends LinkedList

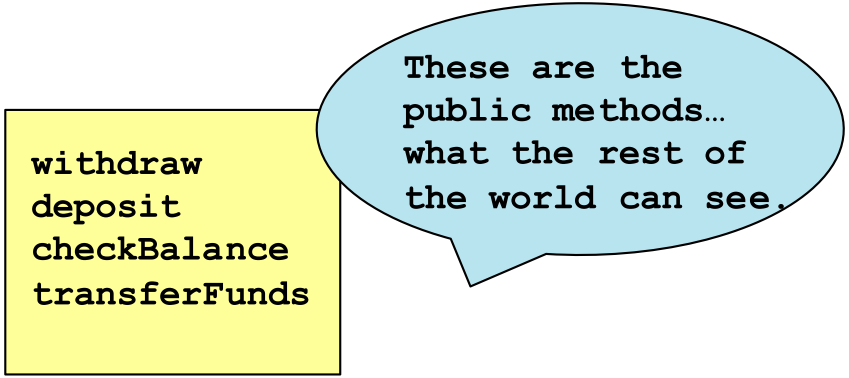
a) public class Stack {
private LinkedList theData;

ISP: The Interface-Segregation Principle

Don’t make your interface “fat”

Fat interfaces: classes that are not cohesive

Interface here means the public methods of the class, NOT the Java concept of interface. Every C++ class has an interface, for example.



```
withdraw
deposit
checkBalance
transferFunds
```

These are the public methods… what the rest of the world can see.

Animal – DON’T DO THIS

```
class Animal
    def swim(self):
        pass
    def purr(self):
        pass
    def bark(self):
        pass
    def fly(self):
        pass
```

Interfaces may contain groups of methods, where each group serves a different set of clients. Best to separate.

UGLY!

```
class cat(Animal):
    def swim(self):
        print("Cat is swimming")
    def purr(self):
        print("purr purr purr")
    def bark(self):
        print("Undefined behaviour")
    def fly(self):
        print("Undefined behaviour")
```

Similar for Fish and Dog

More ISP violations

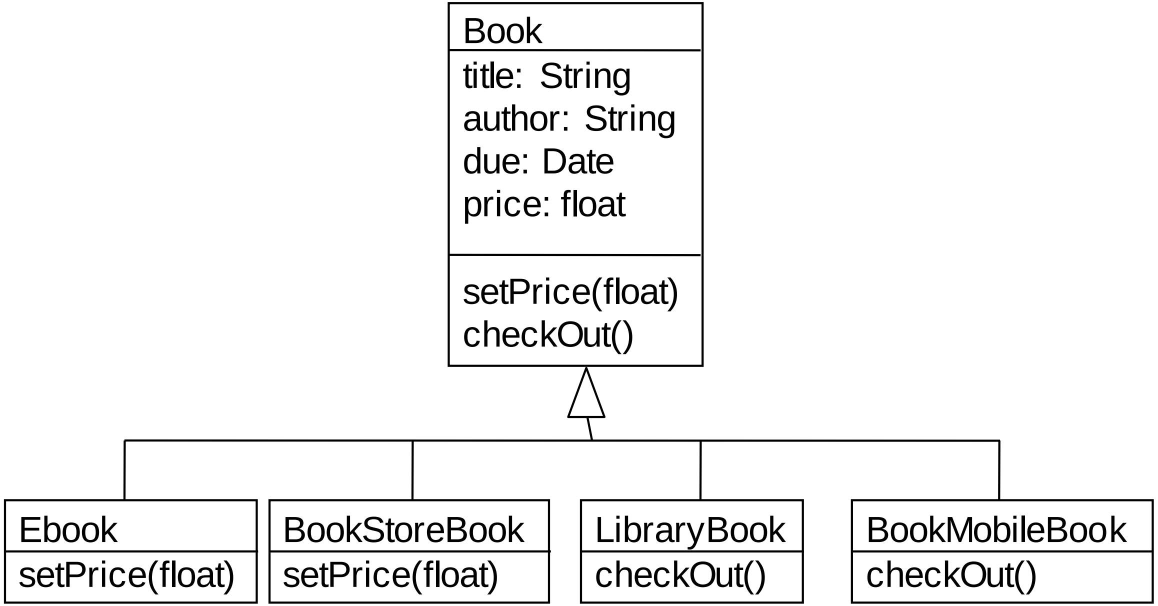
```
class Bird:
    def chirp(self):
        pass
    def eat(self):
        pass
    def walk(self):
        pass
    def fly(self):
        pass
```

Is a Penguin a bird?

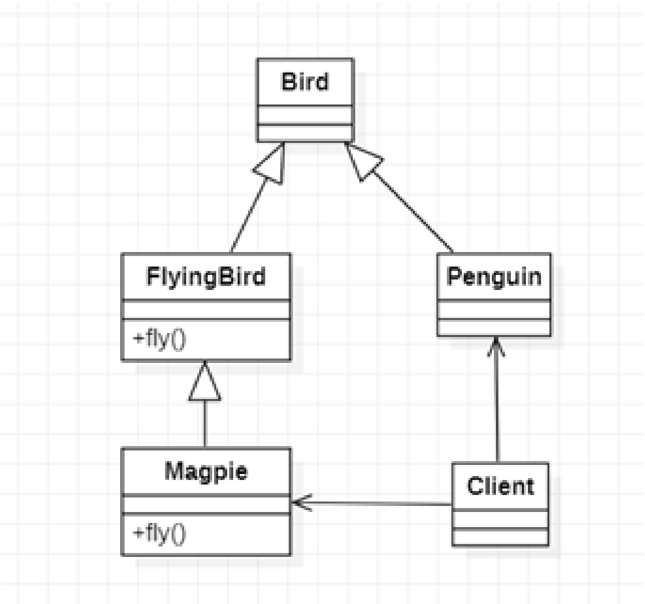
```
class CellPhone:
    def play_music(self):
        pass
    def stop_music(self):
        pass
    def take_photo(self):
        pass
    def zoom_camera(self):
        pass
    def place_call(self):
        pass
    def receive_call(self):
        pass
```

Is this a good design?

Another ISP violation example



Bird fixed!



```
class Bird:
    def chirp(self):
        pass
    def eat(self):
        pass
    def walk(self):
        pass


class FlyingBird(Bird):
    def fly(self):
        pass
```

```
if __name__  == '__main__':
    penguin  = Penguin()
    magpie  = Magpie()
    penguin.eat()
    magpie.fly()
```

```
class Penguin(Bird):
    def chirp(self):
        print("chirp")
    def eat(self):
        print("eat")
    def walk(self):
        print("walk")

class Magpie(FlyingBird):
    def chirp(self):
        print("chirp")
    def eat(self):
        print("eat")
    def walk(self):
        print("walk")
    def fly(self):
        print("fly")
```

Interface-Segregation Principle (ISP)

Methods or method dependencies should not forced on clients that they do not intend to use.. *

When a client A depends on a class that contains methods that are not used by it, but other clients use, then client A will be affected by changes those *other clients* force upon the class.

So it is better to remove such couplings whereever possible, by separating the interfaces.

*** This is the important point – not memorizing the name ISP**

DIP: The Dependency-Inversion Principle

What does it say…

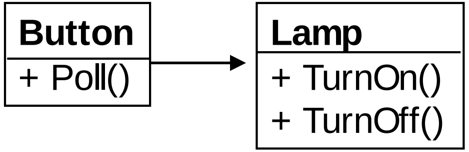
- There should be no dependency of high-level modules on low-level modules. Instead, both should depend on abstractions.

- Abstract elements should not be dependent on details. Rather, details should depend on abstractions.

Simple Example

Button object. Has Poll method to determine whether or not user has “pressed” it. Could be icon on GUI, home security system, physical button, etc. Detects if activated or deactivated.

Lamp object. Has TurnOn and TurnOff methods.



Naïve Model

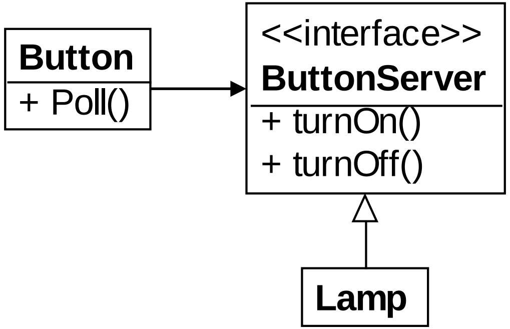
Button can’t be reused. Changes to Lamp might affect Button.

Find the abstraction…

Abstraction: Find an on/off behavior from a user and send that behavior to a target object

Mechanism to detect behavior – Irrelevant!

What is the target object – Irrelevant!



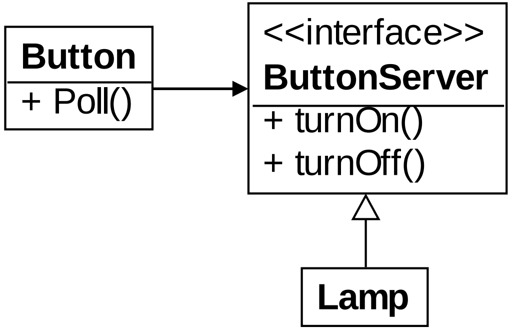
Now Button can control any device that implements ButtonServer

Notice that ButtonServer does NOT depend on Button. So could change it to SwitchableDevice. Then we could have a Button or a Switch do the polling.

How DIP helps…

Thus we have implemented DIP where a high level module (Button) and low level module (Lamp) are dependent on abstraction (ButtonServer)

Also, the abstraction (ButtonServer) does not depend on details (Lamp), but the details (Lamp) depend on the abstraction (ButtonServer)



The advantage is that the Button and Lamp classes are loosely coupled classes, (by including a reference of the ButtonServer interface). So now, we can easily use another class (Fan, for ex.,) which implements ButtonServer with a different implementation.

No-DIP code

- In the example, the BusinessLayer class uses the concrete DataAccess class. Therefore, it is tightly coupled
- BusinessLayer → high-level module and DataAccessLayer → low-level module.
- So BusinessLayer should not depend on the concrete DataAccess class. (According to 1st rule of DIP it is a violation!)

```
class DataAccessLayer:
   def __init__(self):
      return
   def get_cust_name(self, id):
      return "Dummy customer"
   # read from data base for a real application

class BusinessLayer:
   def __init__(self):
      return
   def get_cust_name(self, id):
      data_access_layer = DataAccessLayer()
      return data_access_layer.get_cust_name(id)

if __name__ == '__main__':
   business_layer = BusinessLayer()
   print(business_layer.get_cust_name(1))
```

- Both classes should depend on abstractions, meaning both classes should depend on an interface or an abstract class. In the example there is no abstract class. (According to 2ndst rule of DIP it is again a violation!)

DIP code!

```
class ICustData:
    def GetCustomerName(self, id):
        pass

class CustData(ICustData):
    def __init__(self):
        return
    def get_cust_name(self, id):
        return "Dummy customer"
    # read from data base for a real application
```

```
class DataAccess:
    @staticmethod
    def get_customer_data():
        return CustData()

class BusinessLayer:
    def __init__(self):
        self.customer_data_access = DataAccess.get_customer_data()
    def get_cust_name(self, id):
        return self.customer_data_access.get_cust_name(id)
```

```
if __name__ == '__main__':
    business_layer = BusinessLayer()
    print(business_layer.get_cust_name(1))
```

- Here the high-level module (BusinessLayer) and low-level module (DataAccess) are dependent on an abstraction (ICustData).
- Also, the abstraction (ICustData) does not depend on details (DataAccess), but the details depend on the abstraction.

SOLID Principles

Single Responsibility Principle : A class should change for only one reason or it should have only one responsibiltiy

Open/Closed Principle : Software (classes, modules, functions, etc.) can only be open for adding new extensions, but it should be closed for changes or modifications.

Liskov Substitution Principle: Subtypes should be replaceable for their base types.

Interface Segregation Principle : Methods or method dependencies should not forced on clients that they do not intend to use.

Dependency Inversion Principle: There should be no dependency of high-level modules on low-level modules. Instead, both should depend on abstractions.

- Abstract elements should not be dependent on details. Rather, details should depend on abstractions.

Questions?

Thank You!!