

# Coupling, Cohesion, & GRASP

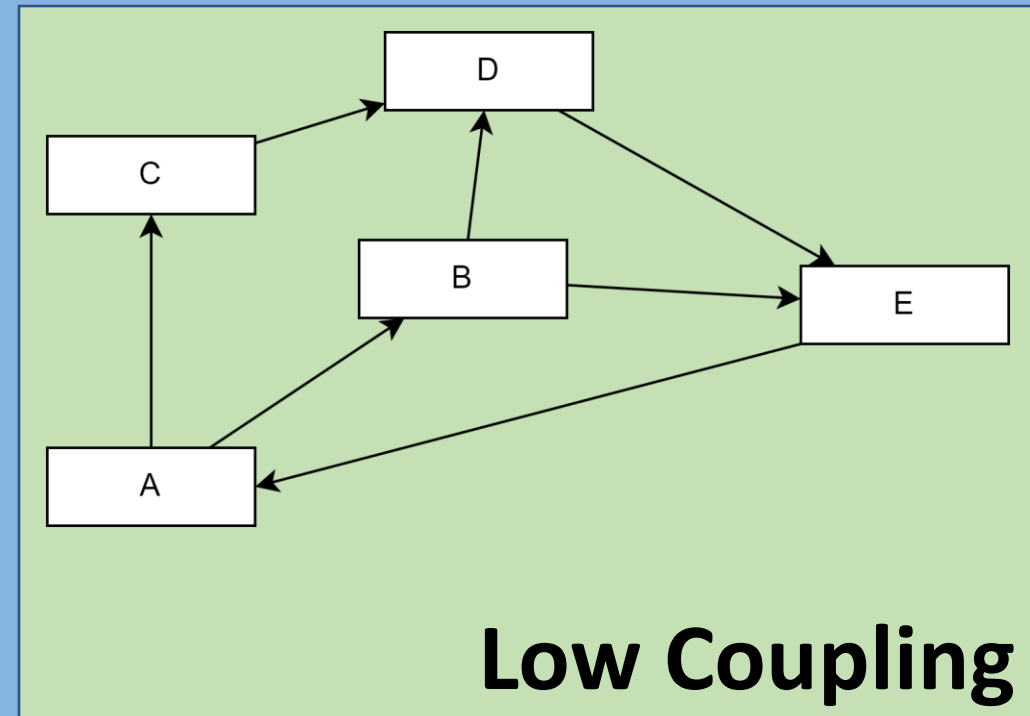
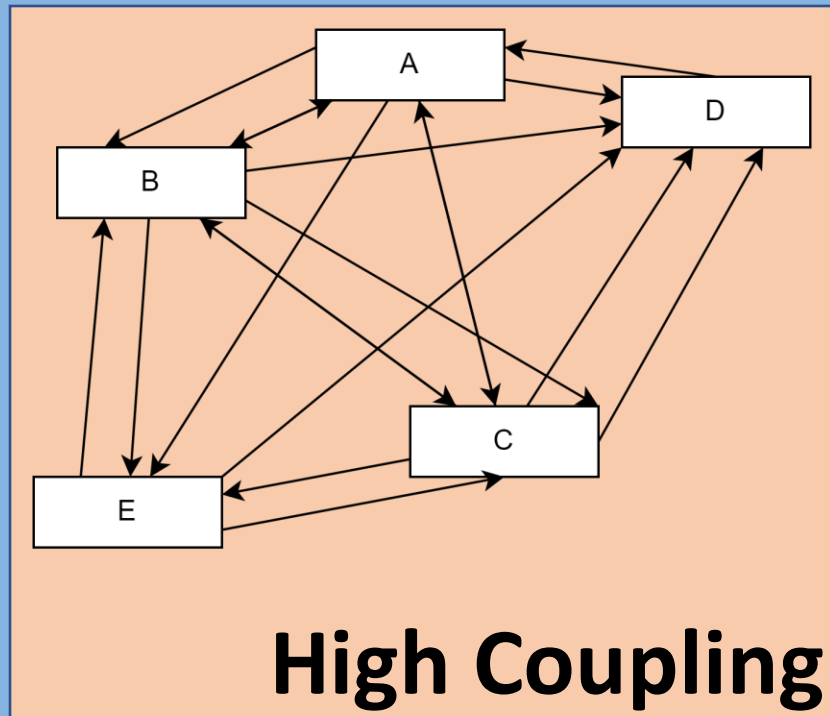
Hunter Squires, Jenna-Luz Pura, & Alphonse Crittenden

# General Outline

- Introductions
- Coupling -- Hunter
- Cohesion -- Al
- Visibility – Jenna [pura0273@vandals](mailto:pura0273@vandals).
- GRASP
- Critique Gantt charts
- Overview of how our Master Gantt charts work (if there's time)

# Coupling

- Describes how connected software modules are to each other



# Levels of Coupling

**W** Content

Common

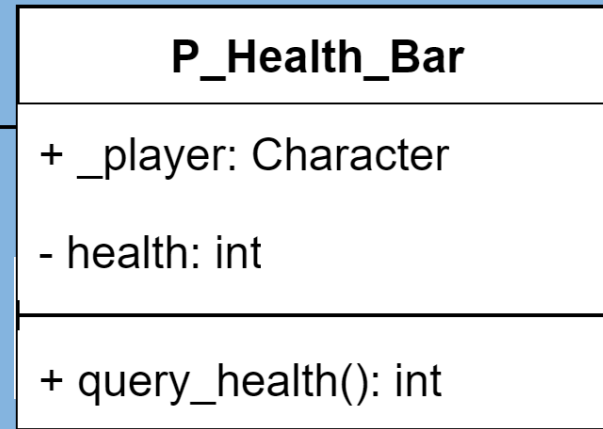
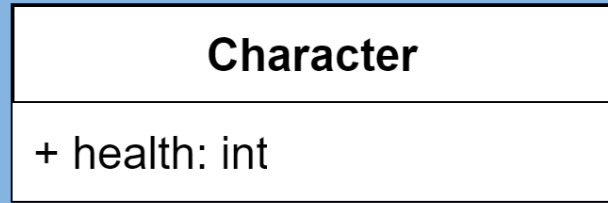
Control

Stamp

**B** Data

# Content Coupling

- Worst level of coupling
- Occurs when one module directly references or changes another module's contents
- This forces modules to be highly dependent on each other



```
public class Character : MonoBehaviour
{
    ...

    public int health;

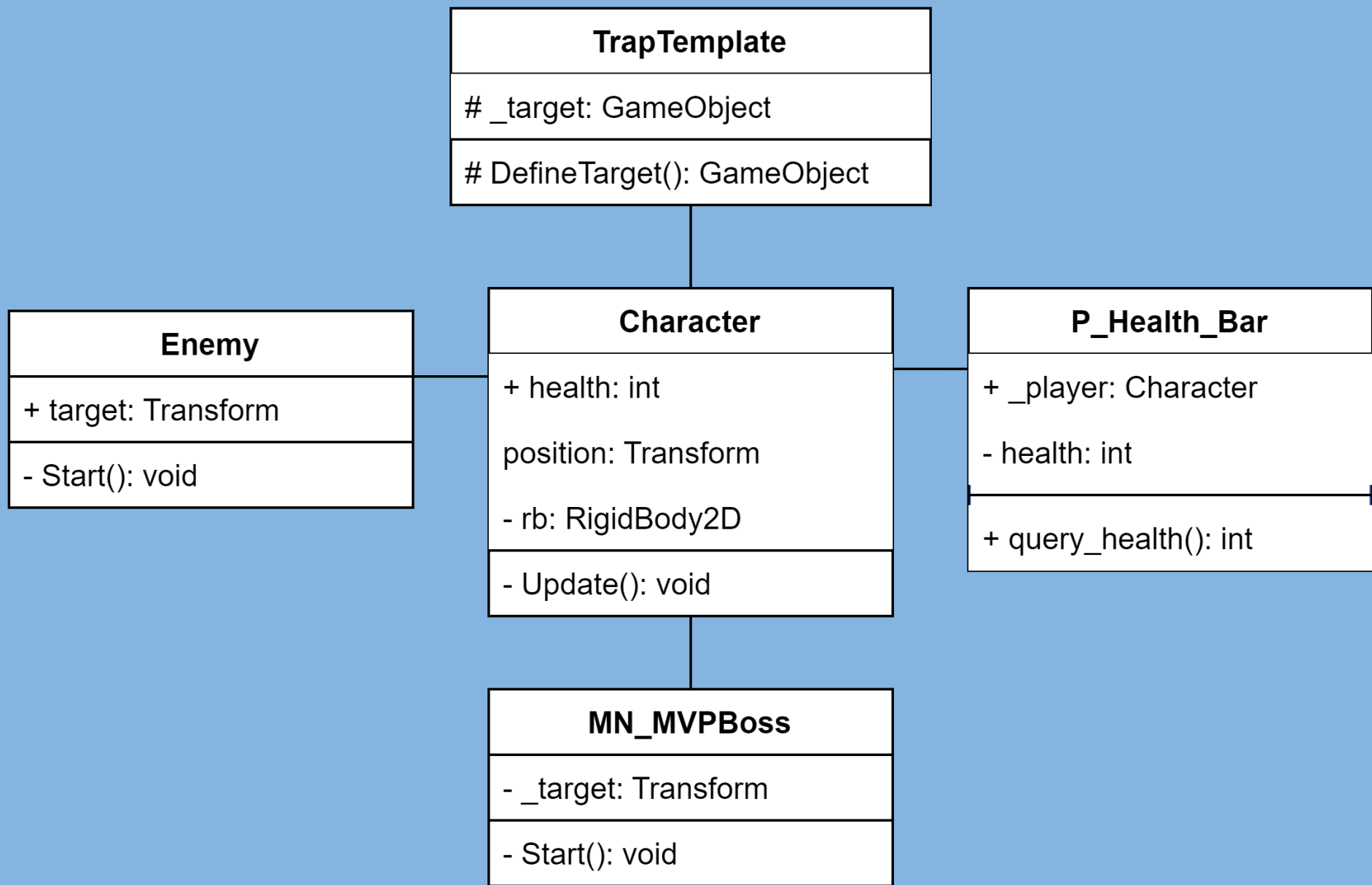
    ...
}
```

```
public class P_Health_Bar : MonoBehaviour
{
    ...
    public Character _player;
    private int health;
    ...
    public int query_health()
    {
        health = _player.health;
        return health;
    }
    ...
}
```



# Common Coupling

- Occurs when multiple modules can read and/or write to the same global resource
- This can cause unintended consequences when changes are made to the global resource





# Control Coupling

- Occurs when one module of code passes control to another module
- Requires one module to tell another module what to do, rather than letting that second module make decisions based on its own state.

# Stamp Coupling

- Occurs when one module passes an entire data structure to another module, when the second module only needs select fields from the first module.
- This passes more data between modules than necessary

**Character**

**TrapTemplate**

# \_target: GameObject

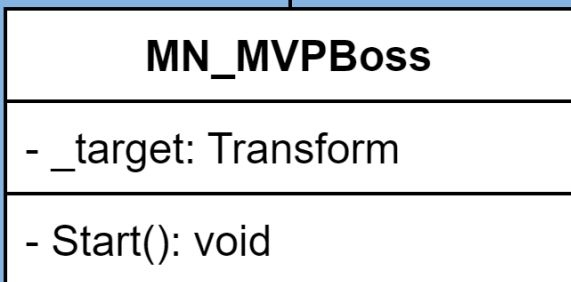
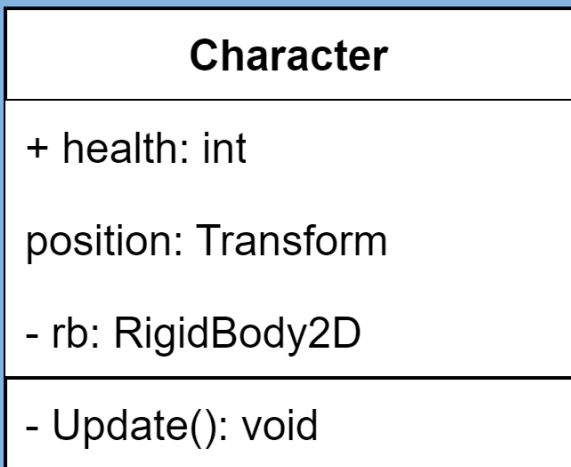
# DefineTarget(): GameObject

```
public class Character : MonoBehaviour
{
    ...
}

public abstract class TrapTemplate : MonoBehaviour
{
    ...
    protected GameObject _target;
    ...
    protected GameObject DefineTarget()
    {
        GameObject target = GameObject.FindGameObjectWithTag("Player");
        _target = target;
        return target;
    }
    ...
}
```

# Data Coupling

- The best level of coupling
- Occurs when relationships between modules are simple operations, and there are no unused fields in shared data structures
- This lowers the dependencies between modules



```
public class Character : MonoBehaviour
{
    ...
    Rigidbody2D rb;
    ...
    void Update()
    {
        //Moves the player based upon WASD input
        float horizontal = Input.GetAxis("Horizontal");
        float vertical = Input.GetAxis("Vertical");
        Vector3 temp = new Vector3(horizontal, vertical, 0);
        temp = temp.normalized * speed * Time.deltaTime;
        rb.MovePosition(rb.transform.position + temp);
    }
}
```

```
public class MN_MVPBoss : MonoBehaviour
{
    ...
    private Transform _target;

    void Start()
    {
        ...
        _target = GameObject.FindGameObjectWithTag("Player").transform;
    }
    ...
}
```

# Subclasses and Coupling

- When 2 modules are in a subclass and superclass relationship, there is inherently a strong coupling between them

```
public abstract class TrapTemplate : MonoBehaviour
{
    protected int p_trapDamage;
    protected float p_trapSpeed;
    protected Vector3 p_projectileSpeed;
    protected GameObject p_trapObject;
    protected GameObject _target;
    public GameObject trapProjectile;
    protected Rigidbody2D p_trapBody;
    protected bool _isLoading = true;
    ...
}
```

```
public class CrossbowTrap : TrapTemplate
{
    void Start()
    {
        p_trapBody = GetComponent<Rigidbody2D>();
        p_trapObject = gameObject;
        p_trapSpeed = 1f;
        p_projectileSpeed = transform.up * 10;
        DefineTarget();
    }
    ...
}
```

# Cohesion

- The measure at which elements within a module interact to increase use/performance
- The Goal
  1. Increased clarity
  2. Ease of maintenance
  3. Reusability

# Levels of Cohesion

Worst

Coincidental



Logical



Temporal



Procedural



Communicational



Best

**Functional or Informational**



# Coincidental Cohesion (worst)

- When elements of a module are loosely related or unrelated entirely

```
public class HealthManager : MonoBehaviour
{
    2 references
    private GameObject player;

    2 references
    private Rigidbody2D rb;

    1 reference
    public float force;

    1 reference
    public static bool isOver = false;

    1 reference
    public Image healthBar;

    [SerializeField]
    5 references
    public float healthAmt = 100f;

    [SerializeField]
    1 reference
    public float damage = 34f;

    //Snowball - shoot in the direction of the palyer
    0 references
    void Start() ...
    //snowball boundary collision
    0 references
    private void OnCollisionEnter2D(Collision2D collision) ...

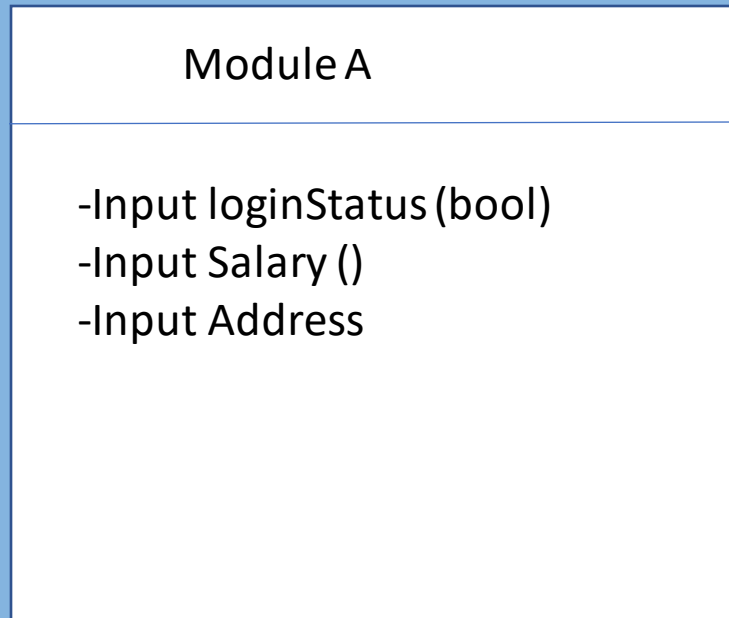
    // Update is called once per frame
    0 references
    void Update() { }
    //player collision upon boss give damage
    0 references
    private void OnCollisionEnter2D(Collision2D collision) ...

    1 reference
    public void takeDamage(float damage) ...
}
```

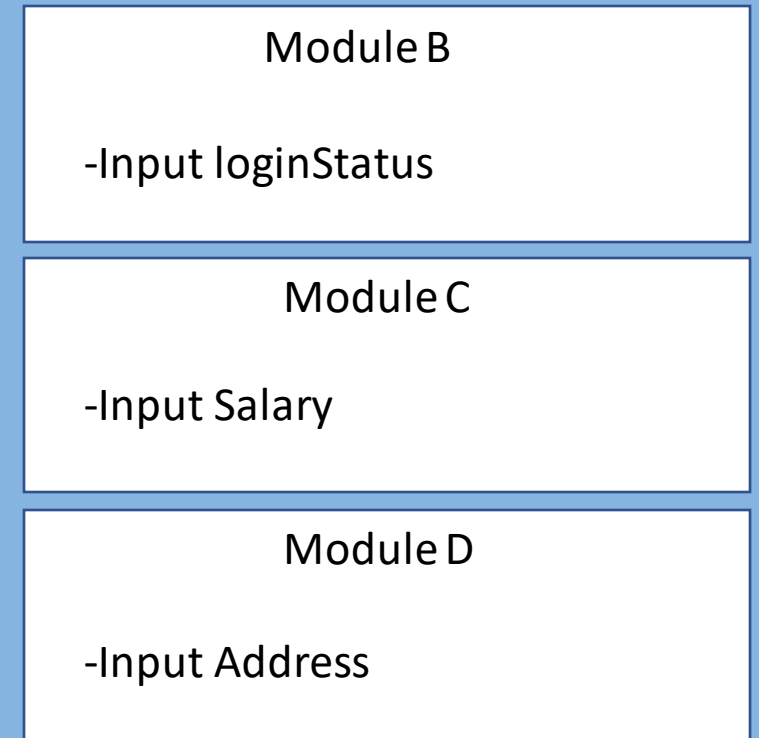
# Logical Cohesion

- Module containing elements of similar actions typical I/O

• Ex.



**BAD**



# Temporal Cohesion

- Elements within a module are related by time

- Ex.

MorningRoutine
-checkNews -makeBreakfast -brushTeeth

# Procedural Cohesion

- Elements within a module are related by the sequence followed by product
- Modules of this type : Combine decision making and task execution
- Ex.

RevenueReport
-Print revReport -Update revExpenses

# Communicational Cohesion

- Elements within a module are related by the data they all operate upon and contributing to the same output

• Ex.

Module A
-Calculate Mean -Calculate Mode
+ return Mean, mode

# Functional Cohesion (Best)

- Elements within a module performs an action in a 1-to-1 case
- Ex.

```
public static class PlayerSingleton {  
    4 references  
    static Player m_Player;  
  
    //Public getter  
    1 reference  
    public static Player Player {  
        get {  
            if (m_Player != null) {  
                return m_Player;  
            } else {  
                try {  
                    m_Player = GameObject.FindWithTag("Player").GetComponent<Player>();  
                } catch (Exception e) {  
                    throw new Exception("Got an exception while finding the player: " + e.Message);  
                }  
                return m_Player;  
            }  
        }  
    }  
}
```

# Informational Cohesion (Best)

- Elements within a module each have their own entry point, exit, and independent code

• Ex.

```
public float horizontal {  
    get {  
        return m_horizontal;  
    }  
}  
  
1 reference  
public bool vertical {  
    get {  
        return m_vertical;  
    }  
}
```

```
public class PlayerComputer : PlayerController {  
  
    0 references  
    public new float horizontal {  
        get {  
            return m_horizontal;  
        }  
        set {  
            m_horizontal = value;  
        }  
    }  
}
```

# Visibility

- How accessible is a variable in a program?
- Ability for one object to see and refer to another object
- For object A to send a message to object B, B must be visible to A





# Types of Visibility

Attribute

Parameter

Local

Global

# Attribute Visibility

- B is an attribute of A
- A: *class Hero*
- B: *public float jumpForce*
- Permanent visibility
  - Remains if A and B exists

```
public class Hero : MonoBehaviour
{
    public float jumpForce;
    public float movementSpeed;

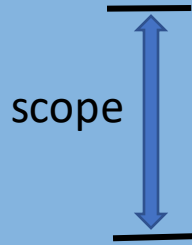
    private Rigidbody2D rb;
    //private Animator anim;
    private SpriteRenderer sprite;
```



Nathan, Hero.cs

# Parameter Visibility

- B is passed as a parameter to a method of A
- Method of A: *void OnTriggerEnter2D*, B: *Collider2D collision*
- Temporary visibility
  - Remains only within the scope of the method



```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "Hero") {
        AudioSource.PlayClipAtPoint(collectSound, transform.position);
        this.Collectected();
        Destroy(gameObject);
    }
}
```



# Local Visibility

- B is declared as a local object in a method of A
- Method of A: *void CreateRandomChunk*, B: *int randX*
- Temporary visibility
  - Remains only within the scope of the method

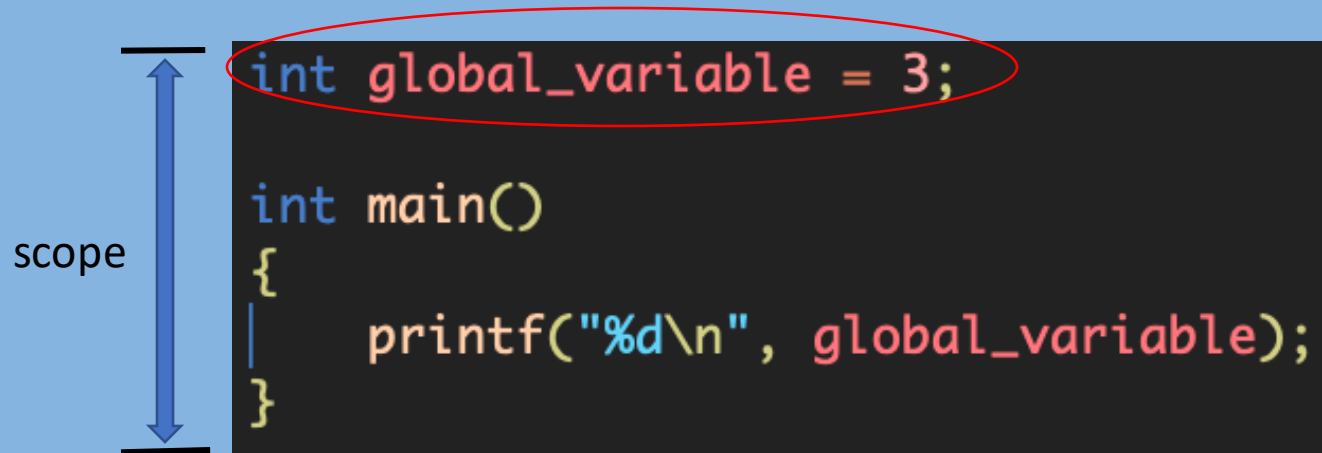
scope

```
public void CreateRandomChunk(Vector3 heroPos){  
    System.Random rnd = new System.Random();  
    int randX = rnd.Next(-10, 10);  
    int randY = rnd.Next(-5, 5);  
    randLoc = new Vector2(heroPos.x + randX - 0.5f, heroPos.y + randY - 0.5f);  
    Instantiate(terrain, randLoc, Quaternion.identity);  
}
```



# Global Visibility

- B is global to A
- A: *int main*, B: *global\_variable*
- Permanent visibility
  - Remains if A and B exists



```
int global_variable = 3;

int main()
{
    printf("%d\n", global_variable);
}
```

**General**

**Responsibility**

**Assignment**

**Software**

**Patterns**

- *Grasp* these principles to successfully design object-oriented software
- *Grasp* these principles to succeed on the POSTMORTEM!
- Different "patterns" from previously mentioned patterns

# 9 GRASP Patterns

Creator

Information Expert

Low Coupling

Controller

High Cohesion

Polymorphism

Pure Fabrication

Indirection

Protected Variations

# Creator

## Problem

- Suppose you have an object of class A that needs to have several instances created. Who should be responsible for creating those new instances of class A?





# Creator (cont.)

## **Solution:**

Select a class B that bears the responsibility of creating objects of class A if one or more of the following is true:

- B contains A
- B records A
- B closely uses A
- B holds the data that will be passed to A (i.e. B is an 'Expert' of A)

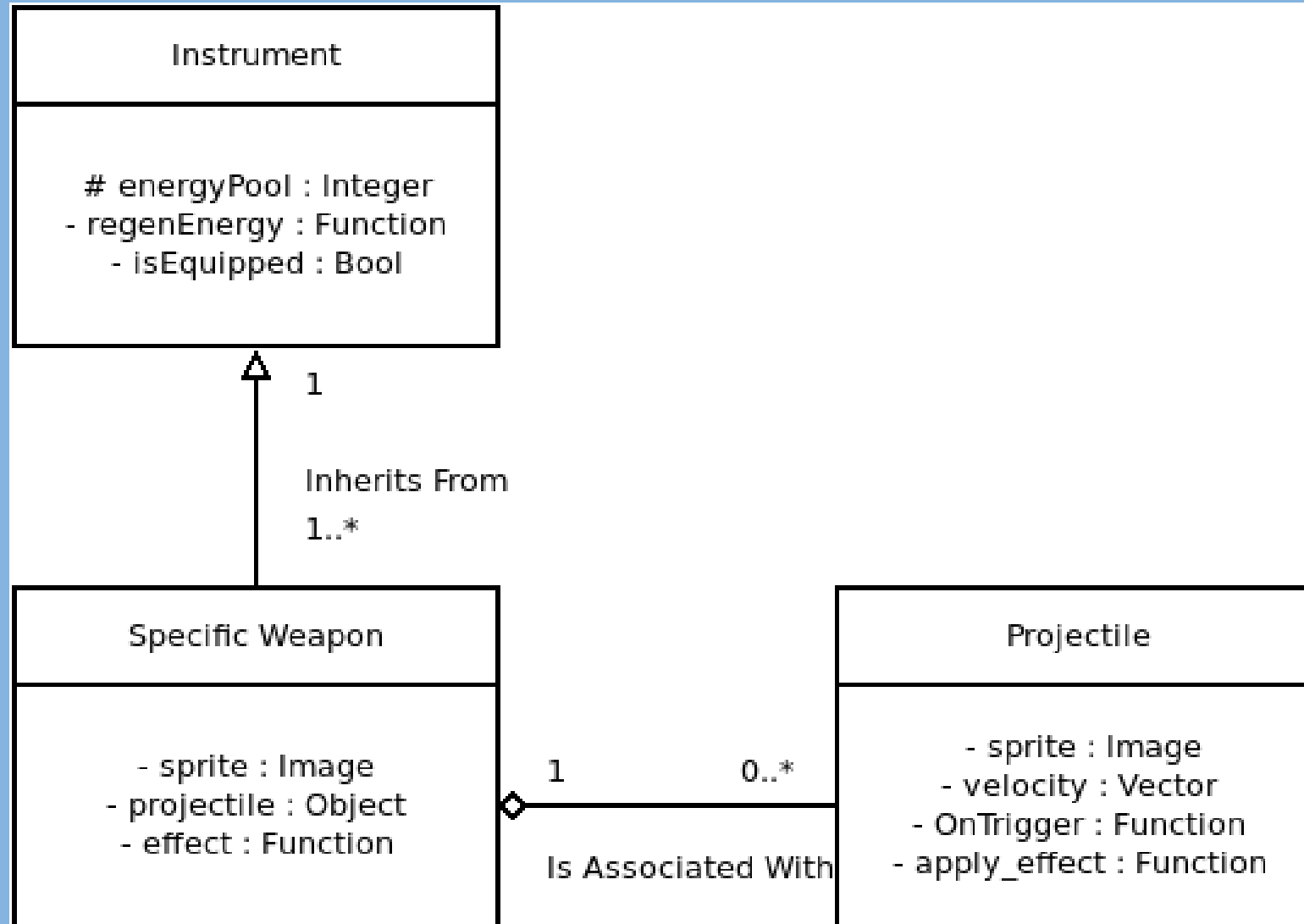
These conditions also imply that class A is visible to class B

# Selecting a Creator



- Determine an object's creator based on how many of these criteria are met
- The more criteria met, the higher the coupling between the creator and the created object
  - Picking an already strongly coupled class to be the creator prevents further coupling

# Creator Example



# Information Expert

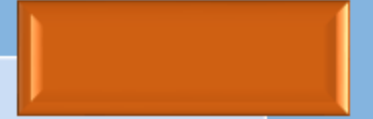
## Problem

- What is a general principle of assigning responsibilities to objects?

## Solution

- Assign responsibilities to an 'Expert Class', which is a class with the information to complete the task.

# Information Expert (cont.)



## Benefits

- Encapsulates data well
- Class behavior is spread across smaller, lightweight classes

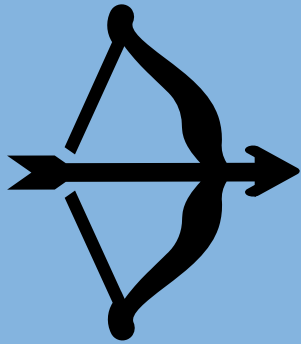
## Detriments

- Tends to lower cohesion by adding responsibility to a class
- Can raise coupling if the added responsibility requires communication with a separate module

# Information Expert Example

## CrossbowBolt

- \_bolt: Rigidbody2D  
+ boltSpeed: Vector3



## Character

+ health: int



Who should measure that the collision occurred to maximize cohesion, minimize coupling, and best utilize visibility?

# Low Coupling – How to implement it?

- Use private variables and methods
- Limit data sharing between modules
- When designing modules, classes, and objects make sure they are designed to lower coupling and not raise it

# Controller

## Problem

When a request comes from the UI layer, what is the first object that should receive the message to controls/coordinates within the domain layer to accomplish the request?

## Solutions

1. Assign it to a class that represents the overall system
    - Façade Controller
  2. Assign it to a class that represents a use case handling a specific system operation
    - Use Case controller
- + which may imply High Cohesion, Low coupling, and better visibility



# Controllers (cont.)

## Pros

1. Increases potential of reuse and pluggable interfaces
2. Opportunity to reason about the state of the use case

## Cons

1. Over-assignment of responsibility (bloated controller)

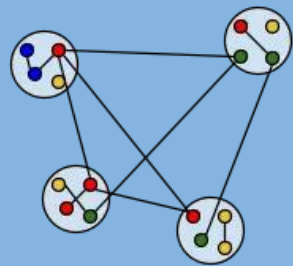
# High Cohesion

## Problem

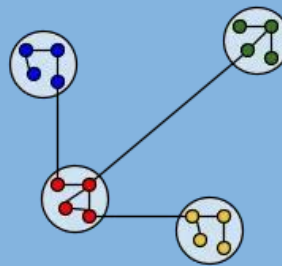
Ability to keep objects Focused, Understandable, and Maintainable

## Solution

Assign a responsibility so that cohesion remains high



low cohesion  
high coupling



high cohesion  
low coupling

# Polymorphism

## **Problem**

How to handle alternatives based on type, and to create pluggable software components

## **Solution**

When related alternatives or behaviors vary by type(class), assign responsibility for the behaviors using polymorphic operations to types for which the behaviors differ

# Pure Fabrication

## **Problem**

Which object should be assigned responsibility when you run the risk of violating high cohesion and low coupling?

## **Solution**

Assign a highly cohesive set of responsibilities to a new class

# Pure Fabrication Example

```
void make_account()
{
    SAConnection con;
    SACommand;

    try {
        con.Connect("Banking", ...);
        cmd.setConnection(&con);
        cmd.SetCommandText("insert into Accounts (first_name,
            last_name) values ('stu', 'dent');
        cmd.Execute();
    }
}
```

Branch

omer\_num():int

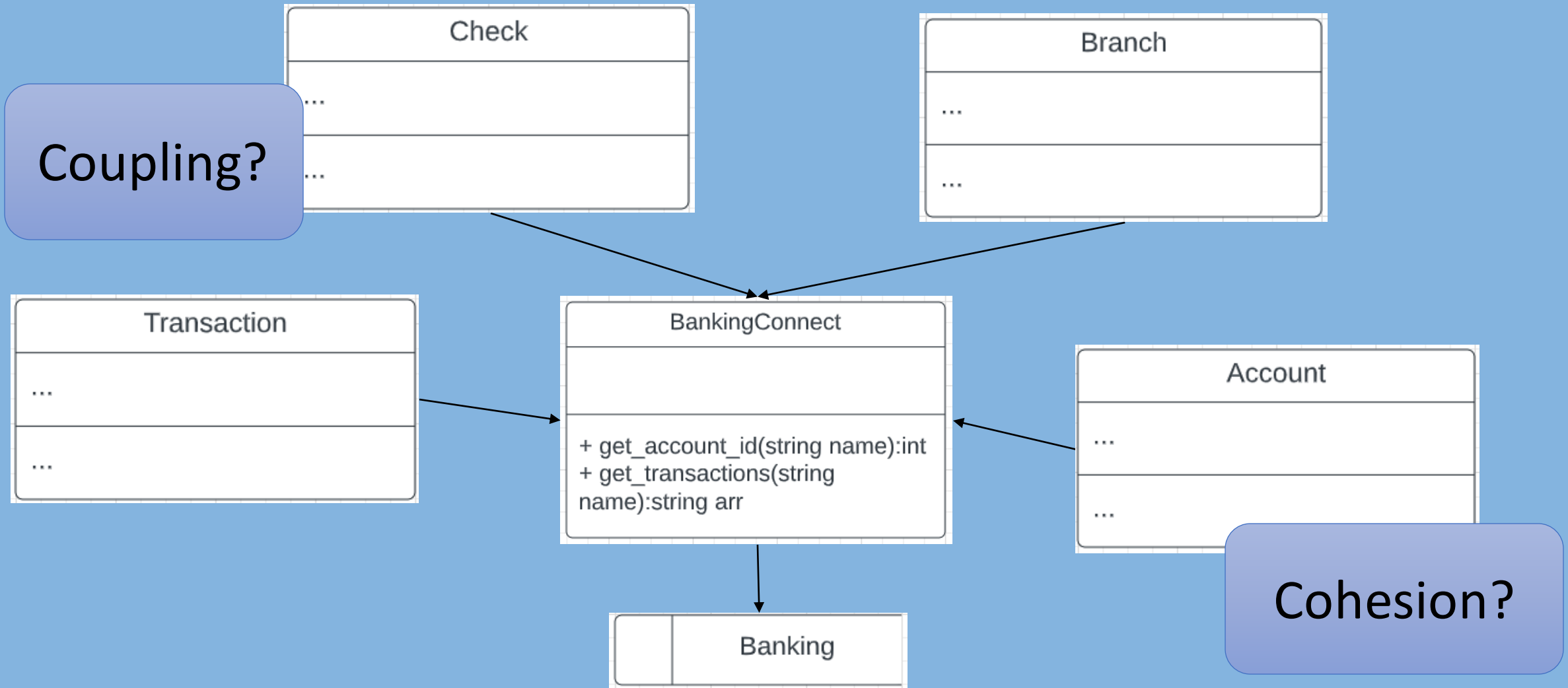
Account

...

- make\_account()
- get\_account\_id():int

Banking

# Pure Fabrication Example (cont.)



# Pure Fabrication Example (cont.)

## **Low coupling**

- Classes and databases are now more independent
- Changes to the database will not affect interface to database

## **High cohesion**

- Many different classes w/ private database methods → single database class w/ public database methods

## **Reusability**

- One database methods can be generalized using parameters

# Indirection

## **Problem**

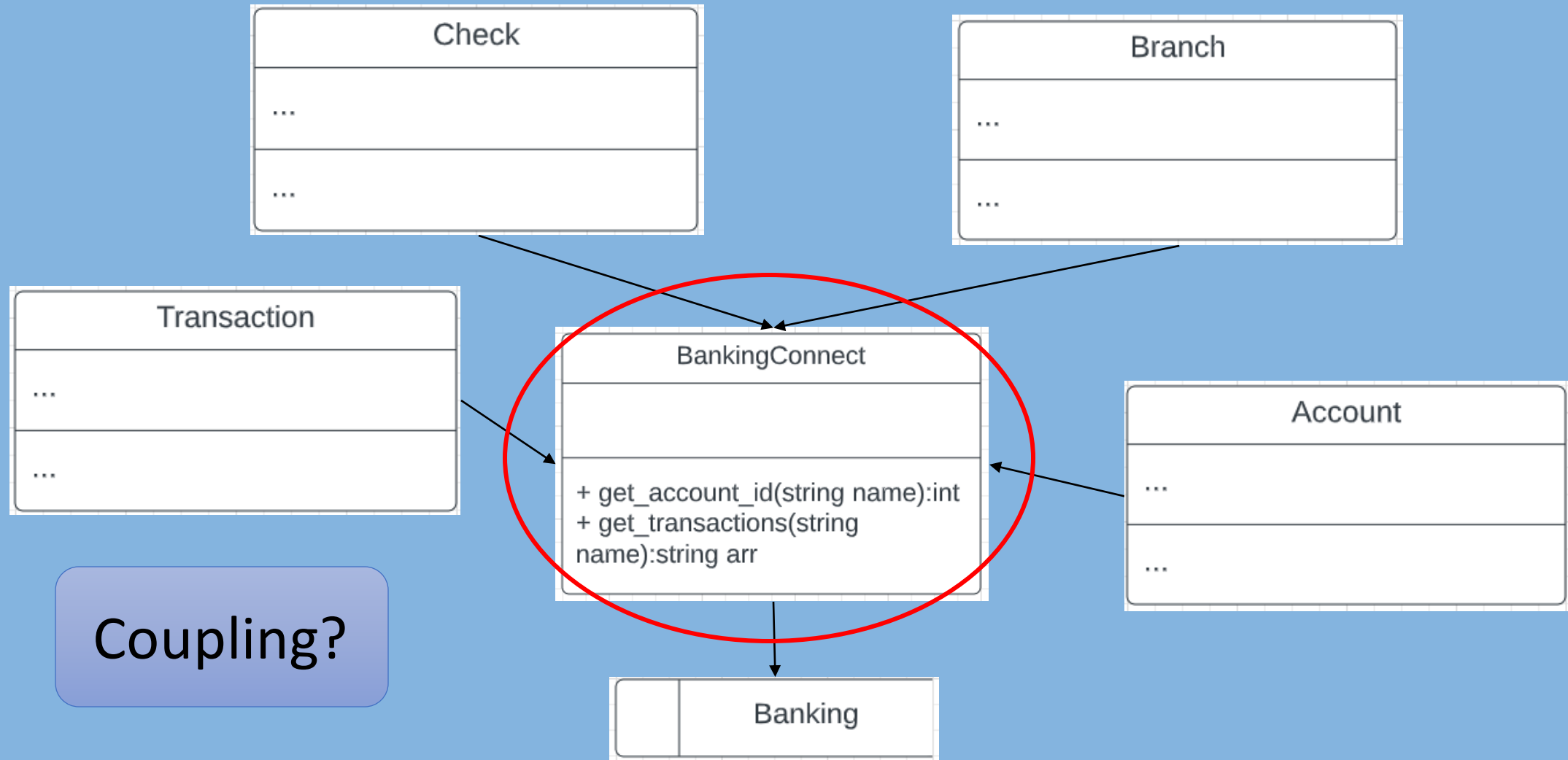
Where do we assign responsibility when we want to avoid direct coupling between two or more objects?

## **Solution**

Assign responsibility to an intermediate object to mediate the between the other objects.



# Indirection Example



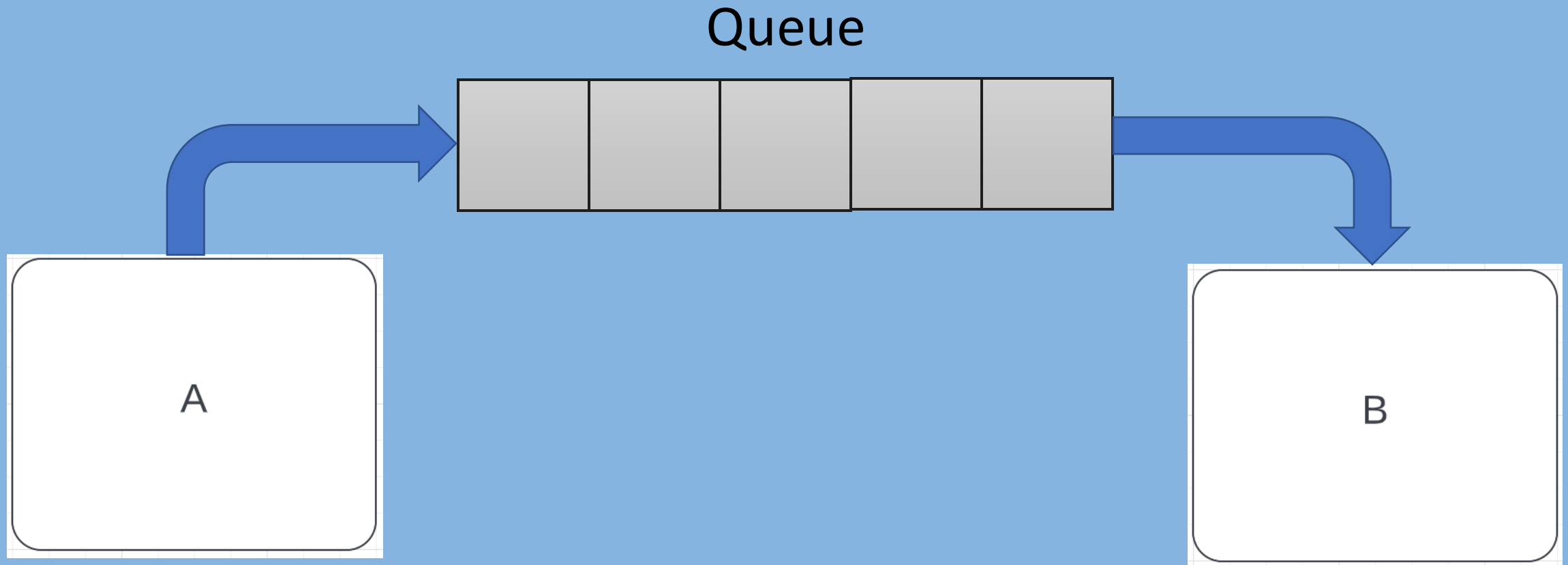
# Indirection Example (cont.)

## **Low coupling**

- Avoids direct coupling between database and other classes
- Changes to database does not affect other classes

# Indirection Example (cont.)

**Queues!**



# Protected Variations

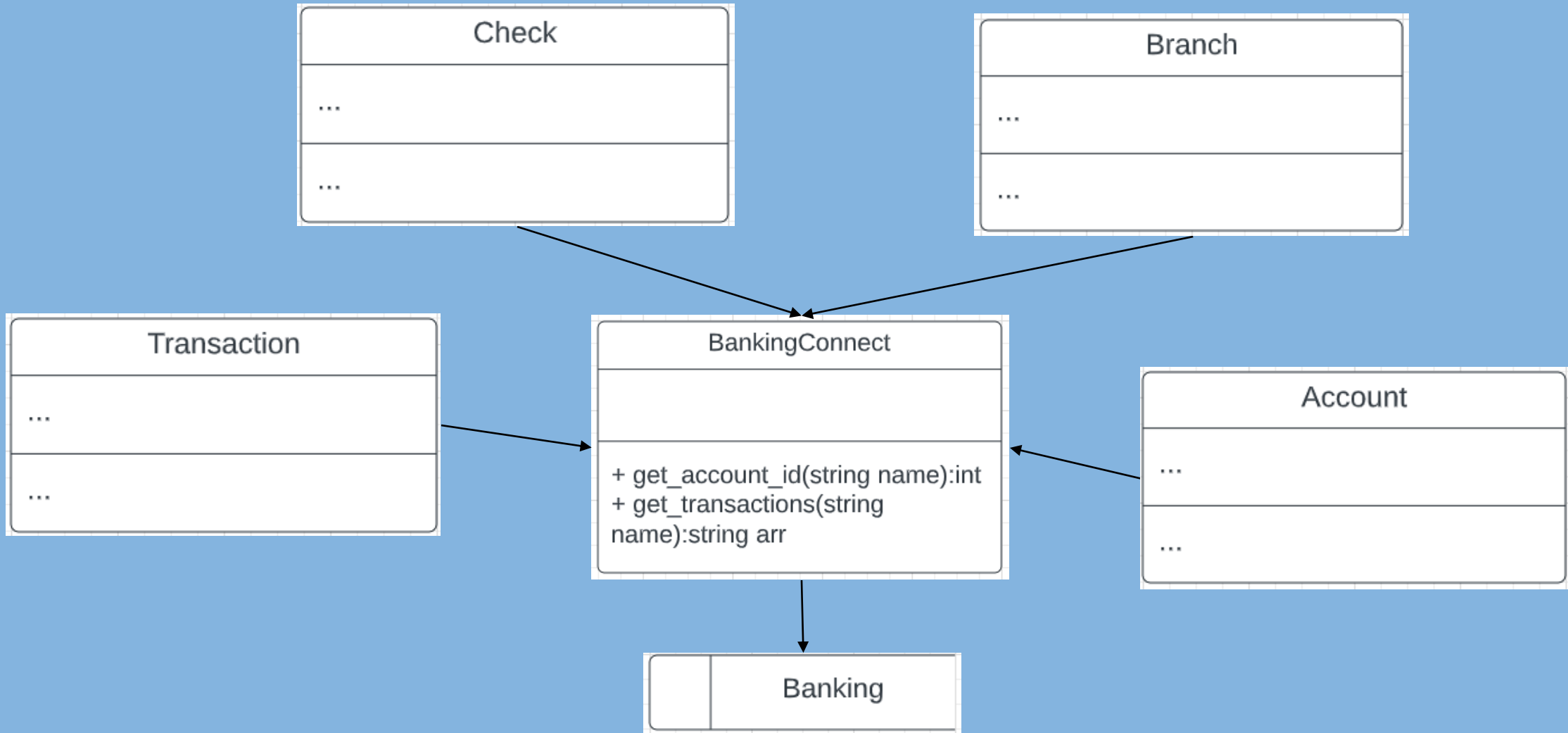
## **Problem**

How can I design a system where variations in some elements does not inadvertently affect the functionality of other objects?

## **Solution**

Create an interface for these varying objects to make they appear invariable.

# Protected Variations Example



# Protected Variations Example (cont.)

## Low coupling

- Prevent changes in the database from affecting banking system classes
- Classes are *protected* from the *variations* of the database
- Especially important for frequently changing code

# GANTT Chart Critiques!

# Gruntworx





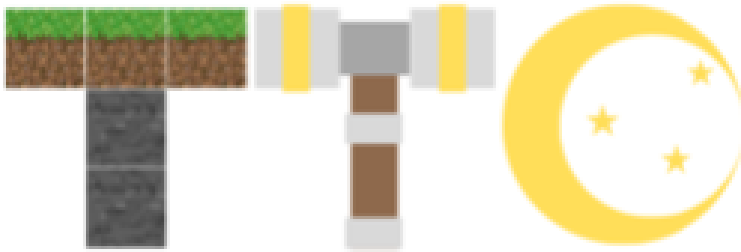
# Clear Lake Studios



# Clover Games



# TTC



# Exodus

**EXODUS**  
Studios

