software engineering ⋰ developpement

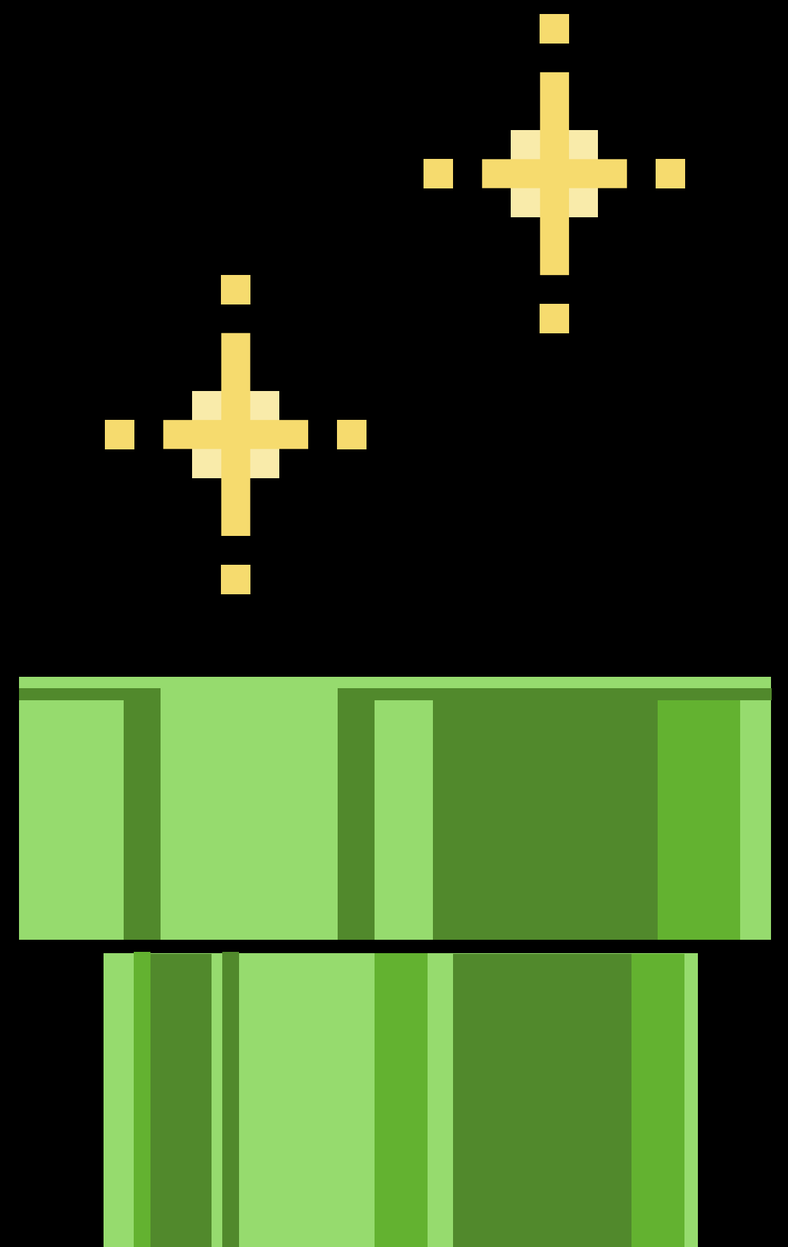
CODEQUEST

WHERE CODE AND FUN COLLIDE

mohamed amine el bacha
younes menfalouti

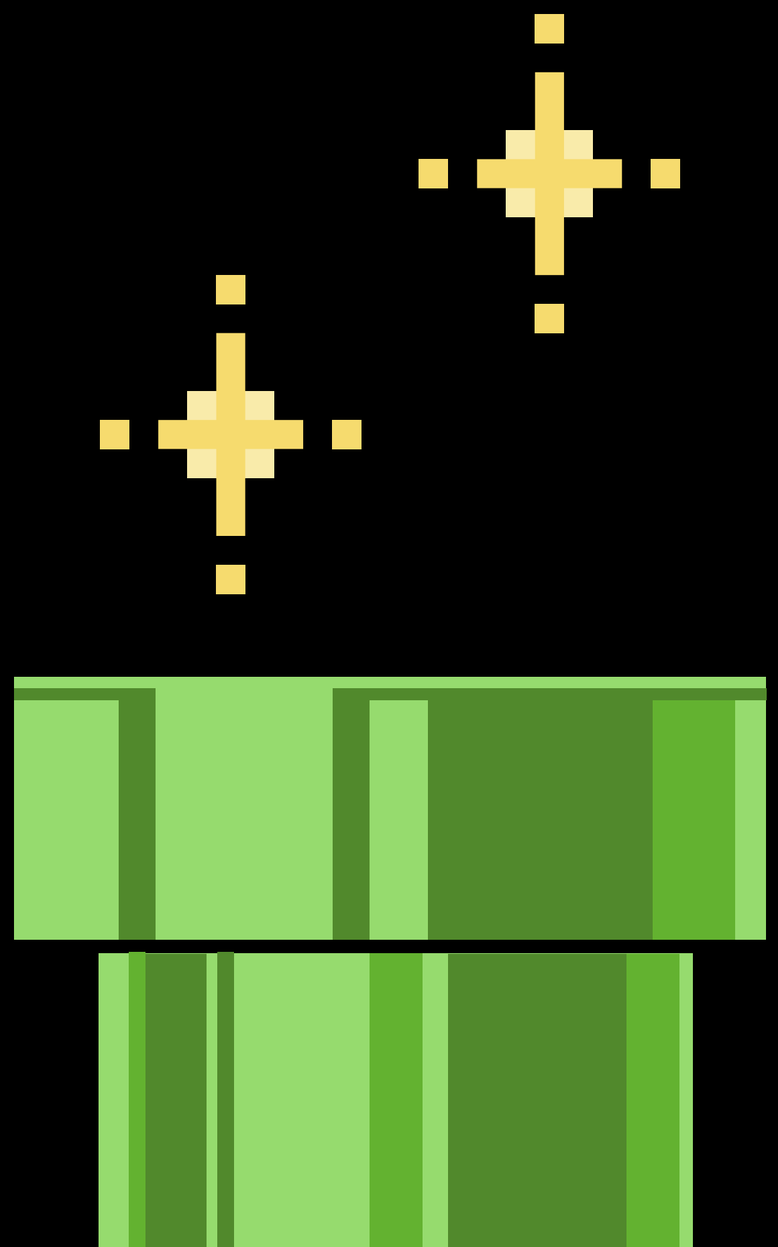
Ok

INTRODUCTION



MAIN PROBLEME

Programming education lacks engagement and immediate practical context.



CONCRET MANIFESTATION

DISCONNECTED ABSTRACTION

Students learn concepts (for, if, variables) without seeing their immediate usefulness in a real context

DELAYED FEEDBACKS

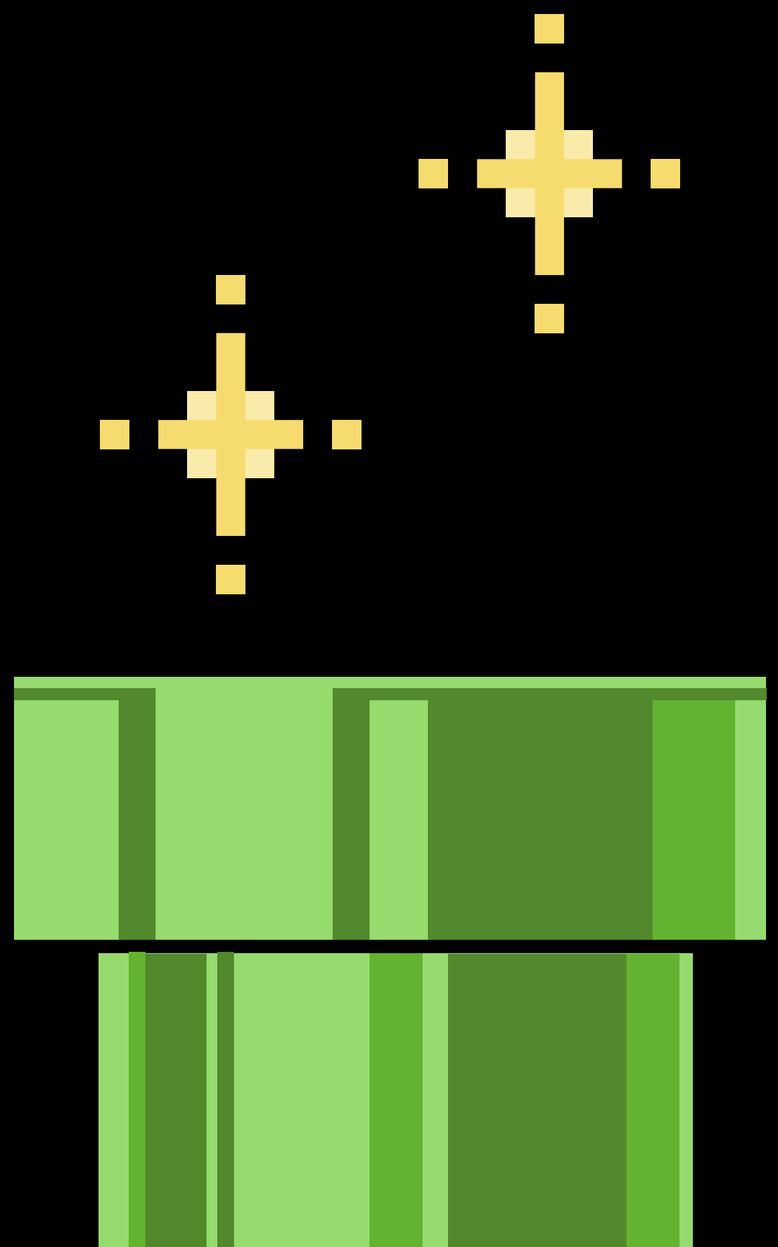
In traditional environments, you must compile → execute → wait → see result → restart

STEEP LEARNING CURVE

Going directly from "Hello World" to complex programs without natural progression

LIMITED MOTIVATION

Classic exercises (calculating factorials, sorting arrays) don't create emotional engagement



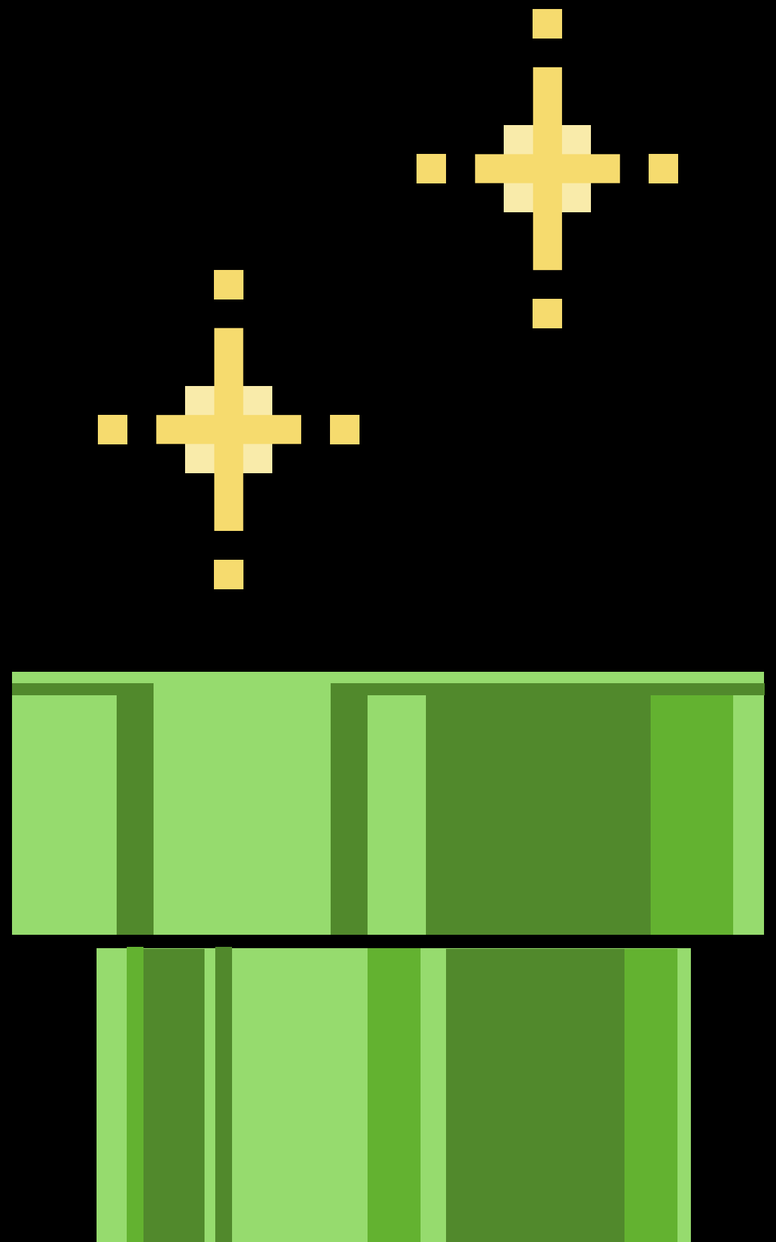
LIMITATION OF EXISTING SOLUTIONS

EXISTING OF OVER-SIMPLIFICATION SOLUTIONS (SCRATCH, BLOCKY) :

- Going directly from "Hello World" to complex programs without natural progression
- Isolated exercises without narrative context
- visual blocks are differenet form real life coding exprience

EXISTING OF OVER-COMPLICATION SOLUTIONS :

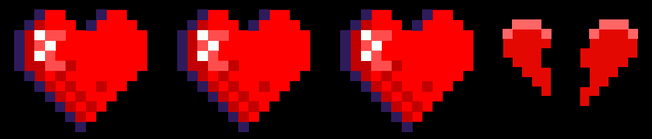
- Diving directly into complete languages without pedagogical progression
- Text-only interface, no immediate visualisation
- Full syntax from the beginning without understanding the core concept



SOLUTION

CODE QUEST

CodeQuest is a 2D educational game that teaches programming fundamentals through interactive gameplay where players control their character by writing real Python-style code commands.



MAIN USERS

PERSONNA 1

SARAH - THE BEGINNER STUDENT

- Age: 14-16 years old
- Education: Middle school-High school
- Programming Experience: None or very limited
- Tech Comfort: Uses smartphones-tablets daily

MAIN USERS

PERSONNA 2

KHALID - A HIGHT SCHOOL TEACHER

- Age: 30-45 years old
- Role: Computer Science/STEM teacher
- Class Size: 20-30 students
- Tech Resources: Computer lab with projector

MAIN USERS

PERSONNA 3

AMINA - THE PARENT

- Age: 35-50 years old
- Role: Parent helping child or private tutor
- Programming Knowledge: Limited to moderate
- Investment: Wants child to learn valuable skills

TECHNICAL FEASIBILITY

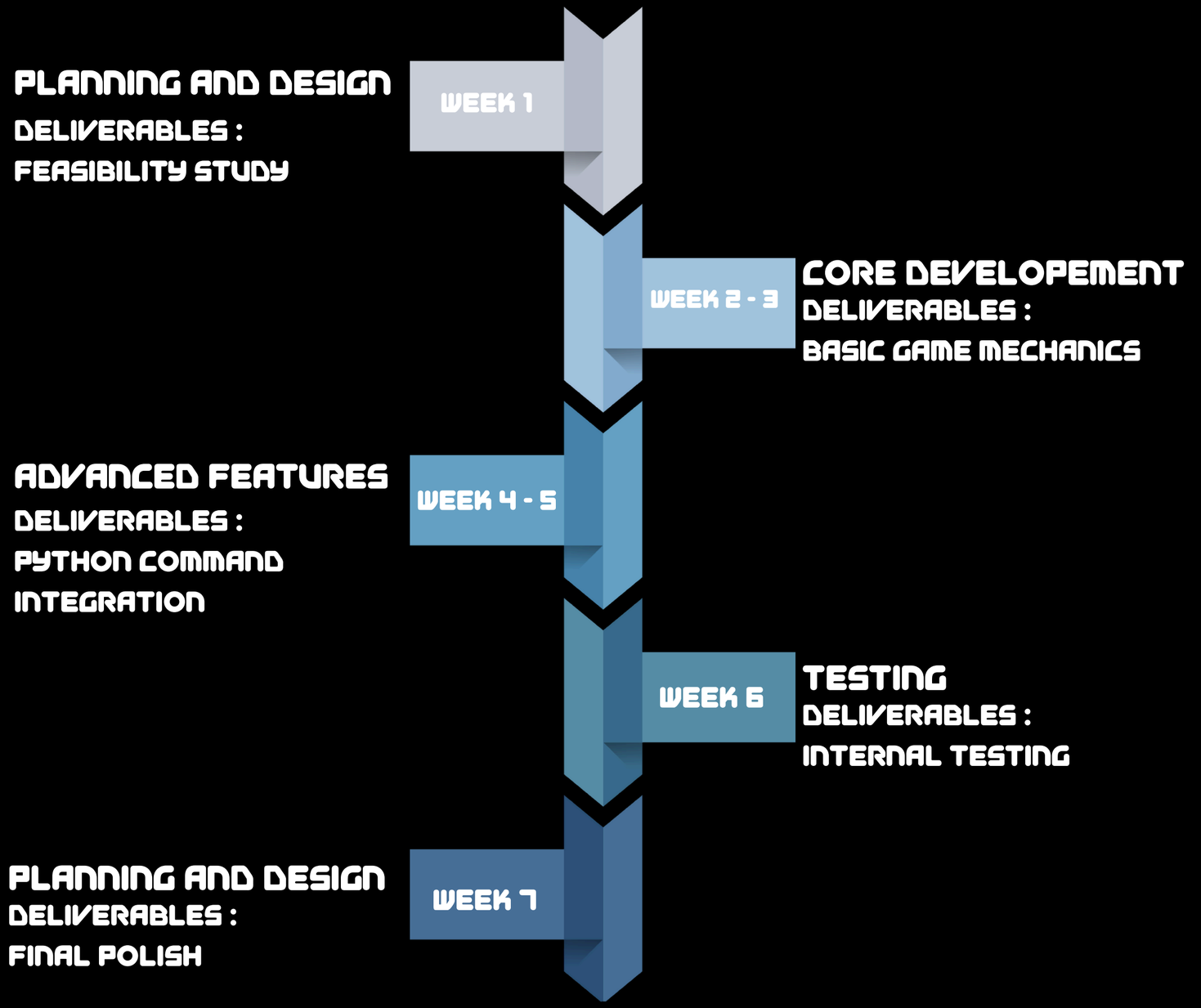
TOOLS
java 25.0.1

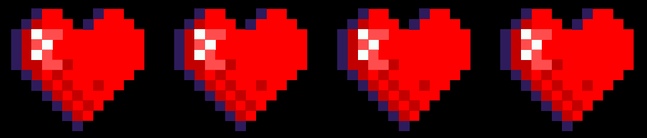
TEAM SKILLS
foundational Java and
knowledge and
game development basic
understanding

GAP ANALYSIS
the team has the
knowledge and ressources
to pass this challenge

KEY CHALLENGE
executing player challeges
without arbitrary game
execution and game
crashes

SCHEDULE FEASIBILITY

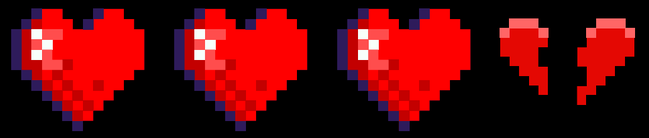




DESIGNED PATTERN

CODEQUEST HAS SEVEREAL DESIGN PATTERNS

- singleton : single asset manager
- Factory : centralised object creation
- Observer : auto UI apdate



SINGLETON DESIGN PATTERN

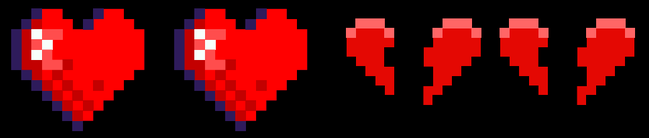
- Prevents loading same images multiple times
- Centralized asset management
- Memory efficiency :all sprites loaded onc

IMPLEMENTATION

```java
// Singleton class that loads and manages all game images (Singleton pattern)
public class AssetHandler {
    private static AssetHandler instance; // Single instance  3 usages
    private java.util.HashMap<String, BufferedImage> assets; // Map of name -> image  5 usages

    // Private constructor prevents external instantiation
    private AssetHandler() {  1 usage
        assets = new java.util.HashMap<>();
        loadAssets(); // Load all assets on creation
    }

    // Get the single instance (creates it if doesn't exist)
    public static AssetHandler getInstance() {
        if (instance == null) {
            instance = new AssetHandler();
        }
        return instance;
    }
```



FACTORY DESIGN PATTERN

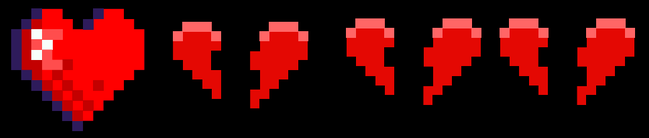
- Separates object creation from usage
- Easy to add new object types
- Consistent initialization of complex objects

IMPLEMENTATION

```
public static MapObject createObject(String name, int x, int y) {  1 usage
    MapObject obj = new MapObject();
    obj.name = name;

    switch (name) {
        case "wall":
            // Standard wall - full tile collision
            obj.collision = true;
            obj.solidArea = new Rectangle( x: 0,  y: 0,  width: 64,  height: 64);
            obj.solidAreaDefaultX = obj.solidArea.x;
            obj.solidAreaDefaultY = obj.solidArea.y;
            obj.image = AssetHandler.getInstance().getImage( key: "wall");
            break;

        case "wall2":
            // Top corner wall - extended height (96px) with vertical offset
            obj.collision = true;
            obj.solidArea = new Rectangle( x: 0,  y: -32,  width: 64,  height: 96);
            obj.solidAreaDefaultX = obj.solidArea.x;
            obj.solidAreaDefaultY = obj.solidArea.y;
            obj.image = AssetHandler.getInstance().getImage( key: "wall_top_corner");
```



OBSERVER DESIGN PATTERN

- Decouples game logic from UI
- Automatic synchronization
- Multiple UI elements can observe same data

IMPLEMENTATION

- Observer interface

```java
/**
 * Observer interface - part of the Observer Design Pattern
 * Observers are notified when the Subject they're watching changes state
 * Used for updating UI elements when game systems (health, keys, chests) change
 */
public interface Observer {  18 usages  1 implementation
    /**
     * Called when the observed Subject's state changes
     * Implementing classes should update their display or state accordingly
     */
    void update();  1 implementation
}
```

OBSERVER DESIGN PATTERN

- Decouples game logic from UI
- Automatic synchronization
- Multiple UI elements can observe same data

IMPLEMENTATION

- subject interface

```
public interface Subject {  6 usages  3 implementations
    /**
     * Add an Observer to the notification list
     * @param o The Observer to register
     */
    void registerObserver(Observer o);  3 usages  3 implementations

    /**
     * Remove an Observer from the notification list
     * @param o The Observer to unregister
     */
    void unregisterObserver(Observer o);  no usages  3 implementations

    /**
     * Notify all registered Observers that state has changed
     * Calls update() on each Observer
     */
    void notifyObservers();  6 usages  3 implementations
}
```

DEMO

OOP CONCEPTS

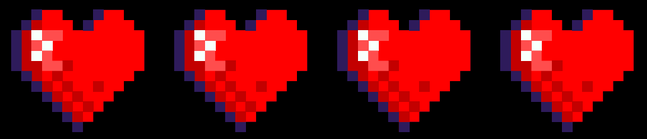
encapsulation

Inheritance

polymorphisme

abstraction

interfaces



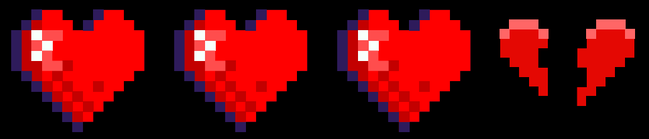
UNLOCK THEE POWER OF ENCAPSULATION

CONTEXT

Player Class Implementation
Protects position and speed

Health Management System
Handles damage and healing

Asset Loading Handler
Encapsulates resource access



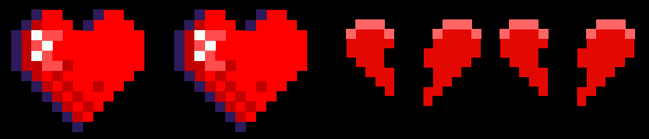
UNLOCK THEE POWER OF INHERITANCE

CONTEXT

Subclassing in Game Design
Enables code reuse effectively.

Hierarchies for Organization
Simplifies the code structure.

Extending Base Functionality
Adds unique behaviors easily.



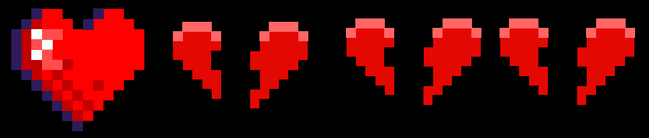
UNLOCK THEE POWER OF POLYMORPHISM

CONTEXT

Dynamic Object Behavior
Multiple forms enhance functionality

Override for Specific Actions
Players customize movement behavior

Unified Interface Interaction
Handle diverse objects seamlessly



UNLOCK THEE POWER OF ABSTRACTION

CONTEXT

Simplifying Complex Systems

Hides implementation details effectively.

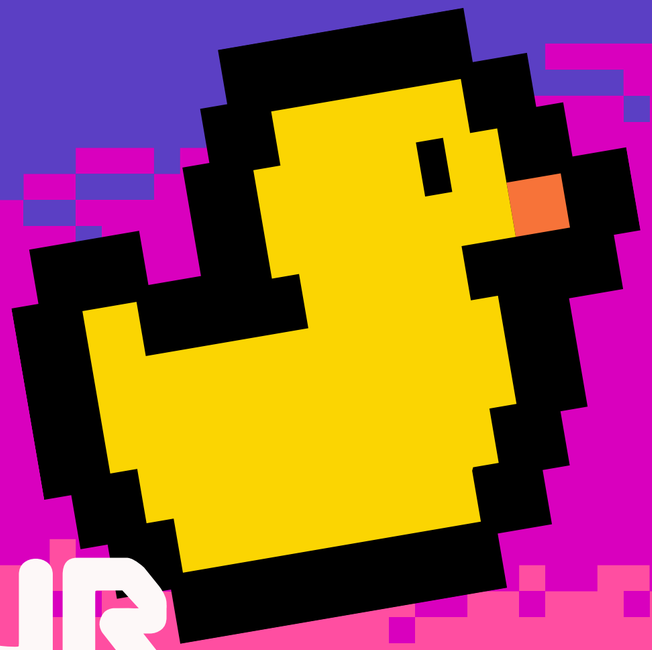
Enhancing Code Clarity

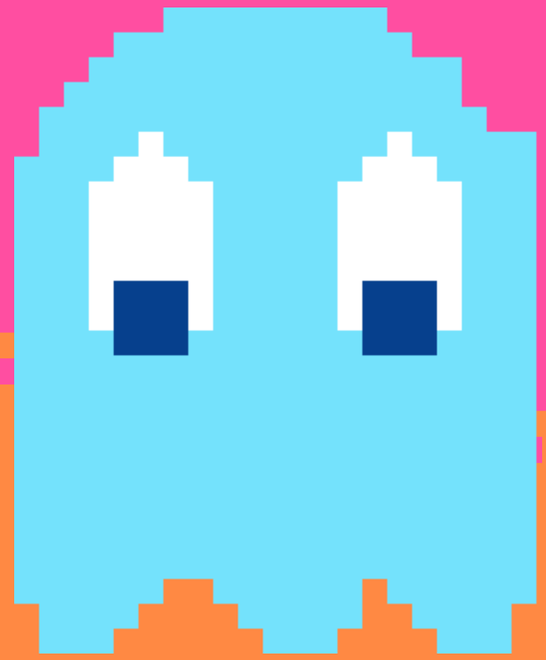
Focuses on essential features.

Promoting Code Reusability

Facilitates easier maintenance and updates.

THANKS FOR YOUR ATTENTION





LOG OUT