COSC 3P91 – Assignment 1 – Question 2

DESIGN DESCRIPTION DOCUMENT, Brock University, Canada

This document presents an object-oriented design for a strategic territory conquest simulation system. The architecture employs abstraction hierarchies, polymorphic behaviors, and modular packaging to create an extensible framework. Seven distinct modules organize functionality: territorycore for domain management, fortifications for defensive capabilities, dwellers for population entities, assaultunits for offensive forces, gamecontrol for orchestration logic, playerspace for participant data, and frontend for interaction mechanisms.

1 INTRODUCTION

This document provides a comprehensive description and justification of the object-oriented design for a village-war strategy game system. The game follows the mechanics of popular mobile strategy games where players build and upgrade their villages, train armies, and attack other players' villages to collect resources and increase their ranking.

The design employs fundamental object-oriented principles including abstraction, inheritance, polymorphism, and encapsulation to create a flexible and extensible system. The architecture is organized into six distinct packages, each with clearly defined responsibilities, promoting separation of concerns and maintainability.

2 SYSTEM OVERVIEW

The village-war strategy game system is designed to support the following core behaviors:

- **Building**: Players can construct various buildings in their village, each occupying space and providing specific functionality
- **Training**: Players can train villagers and army units with different specializations
- **Upgrading**: Both buildings and inhabitants can be upgraded to higher levels, improving their capabilities
- **Village Generation**: The system generates AI-controlled villages for players to attack
- **Attack Exploration**: Players can explore and find villages to attack
- **Attacking**: Players can deploy armies to attack other villages and collect loot

The system manages resources (gold, iron, wood, food), time-based mechanics (building/training time, guard periods), and complex attack simulations that determine battle outcomes.

3 PACKAGE STRUCTURE

The design is organized into six packages, each with a specific responsibility:

3.1 Resources Package

The `resources` package handles all resource-related functionality. It contains:

- `ResourceType` enum defining the four resource types (GOLD, IRON, WOOD, FOOD)
- `Cost` class representing the cost of actions in terms of resources and time
- `ResourceManager` class managing resource storage, capacities, and consumption

Author's address: Design Description Document, Brock University, 1812 Sir Isaac Brock Way, St. Catharines, ON, L2S 3A1, Canada.

This package is kept separate to ensure resource management logic is decoupled from game entities, promoting reusability and maintainability.

3.2 Buildings Package

The `buildings` package contains all building-related classes organized in an inheritance hierarchy. This modular organization allows for easy addition of new building types while maintaining common functionality in abstract base classes.

3.3 Inhabitants Package

The `inhabitants` package manages all population entities including workers and military units. The hierarchical structure allows for shared behavior while enabling specialization for different inhabitant types.

3.4 Core Package

The `core` package contains the central game entities:

- `Village` - Manages all aspects of a player's village
- `Player` - Represents player information and actions
- `Army` - Manages army composition for attacks

3.5 Engine Package

The `engine` package provides game orchestration and simulation logic:

- `GameEngine` - Main game loop and time management
- `AttackSimulator` - Simulates combat between armies and villages
- `VillageGenerator` - Creates AI villages
- `ScoreCalculator` - Computes various game scores
- `AttackResult` - Encapsulates attack outcomes

3.6 UI Package

The `ui` package separates the user interface from game logic, containing the `GameUI` class that handles player interactions and display.

4 BUILDING HIERARCHY

The building system uses a carefully designed inheritance hierarchy to maximize code reuse while allowing for specialization.

4.1 Abstract Building Class

The `Building` abstract class serves as the base for all structures. It defines common attributes and behaviors:

- Level progression (current level, max level)
- Hit points for damage modeling
- Position on the village grid
- Cost and build time
- Upgrade mechanics with time tracking

Key methods include `upgrade()`, `getUpgradeCost()`, `takeDamage()`, and `repair()`. The upgrade cost scales with level using a multiplier (1.5x per level), balancing progression.

4.2 VillageHall

The `VillageHall` is the most important building, determining the village level and constraining other building upgrades. Players must upgrade the VillageHall to unlock higher levels for other structures.

4.3 Production Buildings

An intermediate abstract class `ProductionBuilding` extends `Building` to add production mechanics:

- Worker assignment (maxWorkers, assignedWorkers)
- Production rate that increases with assigned workers
- Resource generation methods

Concrete production buildings inherit from this class:

- **Farm**: Produces food and increases population capacity
- **GoldMine**: Generates gold resources
- **IronMine**: Generates iron resources
- **LumberMill**: Generates wood resources

4.4 Defense Buildings

Similarly, `DefenseBuilding` extends `Building` with defensive attributes:

- Damage output
- Attack range
- Attack speed
- Target prioritization

Concrete defensive structures include:

- **ArcherTower**: Long-range defense with limited ammunition
- **Cannon**: High damage but slower attack speed

This hierarchy allows adding new building types easily while inheriting common functionality.

5 INHABITANT HIERARCHY

The inhabitant system models the village population using a similar hierarchical approach.

5.1 Abstract Inhabitant Class

The `Inhabitant` abstract class provides base functionality for all population entities:

- Level and hit points
- Training cost and time
- Food consumption
- Upgrade mechanics
- Damage handling and healing

5.2 Workers

The `Worker` class represents general-purpose villagers who perform various tasks:

- Building construction
- Food production
- Building repairs
- Task tracking (idle vs. busy)

Workers are essential for village development but don't participate in combat.

5.3 Resource Collectors

An abstract `Collector` class extends `Inhabitant` for specialized resource gathering. It adds collection rate and target resource type. Concrete collectors include:

- **GoldCollector**: Collects from gold mines
- **IronCollector**: Collects from iron mines
- **WoodCollector**: Collects from lumber mills

5.4 Army Units

The `ArmyUnit` abstract class extends `Inhabitant` with combat capabilities:

- Damage output
- Attack range
- Movement speed
- Attack methods

Concrete military units provide variety in combat:

- **Soldier**: Melee fighter with shield for blocking
- **Archer**: Ranged attacker with faster movement
- **Knight**: Heavy armored unit with high damage
- **Catapult**: Siege unit effective against buildings

6 VILLAGE CLASS

The `Village` class is the central hub managing all aspects of a player's settlement:

- **Identity**: Name and level
- **Population**: Current and maximum population limits
- **Space**: Area usage for building placement
- **Buildings**: Collection of constructed buildings
- **Inhabitants**: Collection of trained population
- **Resources**: ResourceManager instance
- **Scores**: Defense and attack capability calculations
- **Protection**: Guard period for new players

Key methods include:

- `addBuilding()`: Validates space and adds buildings
- `addInhabitant()`: Checks population limits
- `calculateDefenseScore()`: Computes defensive strength
- `canBuild()`: Validates building placement

The guard period (7 days for new villages) prevents attacks on new players, encouraging fair gameplay.

7 ARMY CLASS

The `Army` class (in the core package) manages attack force composition:

- Collection of ArmyUnit objects
- Total army size tracking
- Methods to add/remove units
- Attack power calculation
- Casualty management during battles

This separate class allows players to compose custom armies from their trained units.

8 GAME ENGINE

The `GameEngine` class orchestrates the overall game flow:

- **Time Management**: Tracks game time and speed
- **Player Management**: Maintains active players list
- **Upgrade Processing**: Completes time-based upgrades
- **Validation**: Enforces game rules (e.g., VillageHall level requirements)

Supporting classes provide specialized functionality:

- **AttackSimulator**: Runs battle simulations considering unit types, defenses, and random factors
- **VillageGenerator**: Creates balanced AI villages for players to attack
- **ScoreCalculator**: Computes rankings based on attack/defense performance
- **AttackResult**: Encapsulates battle outcomes (success, loot, casualties, stars)

9 PLAYER AND UI

The `Player` class represents individual players:

- Player identification (name, unique ID)
- Village ownership
- Statistics (attack/defense wins/losses, ranking, total loot)
- Action methods (building, upgrading, attacking)

The `GameUI` class separates presentation from logic:

- Display methods for village, menus, and battle results
- Input handling for player actions
- Reference to current player and game engine

This separation allows changing the UI (e.g., from console to GUI) without modifying game logic.

10 DESIGN PRINCIPLES AND JUSTIFICATION

10.1 Abstraction and Inheritance

The design extensively uses abstract base classes (`Building`, `Inhabitant`, `ProductionBuilding`, `DefenseBuilding`, `Collector`, `ArmyUnit`) to:

- Share common functionality across related classes
- Reduce code duplication
- Enforce consistent interfaces
- Enable polymorphic behavior

This hierarchy makes adding new building or inhabitant types straightforward—simply extend the appropriate abstract class and implement specific behaviors.

10.2 Encapsulation

All classes use private or protected attributes with public accessor methods, ensuring:

- Internal state is protected from invalid modifications
- Implementation details can change without affecting clients
- Validation can be enforced (e.g., resource limits, population caps)

10.3 Polymorphism

Abstract classes and inheritance enable polymorphic collections:

- `Village` stores `Building` references, treating all building types uniformly
- `Army` manages `ArmyUnit` references regardless of specific unit type
- Upgrade and damage calculations work on base class references

This allows the system to handle new entity types without modifying existing code.

10.4 Separation of Concerns

The package structure cleanly separates different aspects:

- Resource management isolated from game entities
- UI separated from game logic
- Game orchestration separated from domain objects
- Building/inhabitant hierarchies in separate packages

This modularity improves maintainability and allows team members to work on different packages independently.

10.5 Extensibility

The design supports future extensions:

- New building types: Extend `ProductionBuilding` or `DefenseBuilding`
- New inhabitants: Extend `Collector` or `ArmyUnit`
- New resources: Add to `ResourceType` enum
- Enhanced UI: Implement new UI classes using the same game engine interface

10.6 Scalability

The architecture supports large-scale gameplay:

- Collections for buildings and inhabitants scale with village size
- Time-based mechanics allow asynchronous processing
- Score calculations enable ranking thousands of players
- Village generation creates unlimited opponents

11 CONCLUSION

This design provides a robust foundation for a village-war strategy game. The careful use of object-oriented principles creates a system that is maintainable, extensible, and aligned with the required behaviors. The modular package structure, thoughtful inheritance hierarchies, and clear separation of concerns enable both current functionality and future enhancements while maintaining code quality and reducing complexity.

A RESOURCES PACKAGE - RESOURCETYPE ENUM

```
package resources;

public enum ResourceType {
    GOLD,
    IRON,
    WOOD,
    FOOD
}
```

B RESOURCES PACKAGE - COST CLASS

```
package resources;

public class Cost {
    private int gold;
    private int iron;
    private int wood;
    private int time;

    public Cost(int gold, int iron, int wood, int time) {
        this.gold = gold;
        this.iron = iron;
        this.wood = wood;
        this.time = time;
    }

    public int getTotalValue() {
        return gold + iron + wood;
    }

    public Cost multiply(double factor) {
        return new Cost(
            (int)(gold * factor),
            (int)(iron * factor),
            (int)(wood * factor),
            (int)(time * factor)
        );
    }

    public int getGold() { return gold; }
    public int getIron() { return iron; }
    public int getWood() { return wood; }
    public int getTime() { return time; }
}
```

C RESOURCES PACKAGE - RESOURCEMANAGER CLASS

```
package resources;

public class ResourceManager {
    private int gold;
    private int goldCapacity;
    private int iron;
    private int ironCapacity;
    private int wood;
    private int woodCapacity;
    private int foodProduction;
    private int foodConsumption;

    public void addGold(int amount) {
        gold = Math.min(gold + amount, goldCapacity);
    }

    public void addIron(int amount) {
        iron = Math.min(iron + amount, ironCapacity);
    }

    public void addWood(int amount) {
        wood = Math.min(wood + amount, woodCapacity);
    }

    public boolean consumeResources(Cost cost) {
        if (hasEnoughResources(cost)) {
            gold -= cost.getGold();
            iron -= cost.getIron();
            wood -= cost.getWood();
            return true;
        }
        return false;
    }

    public boolean hasEnoughResources(Cost cost) {
        return gold >= cost.getGold() &&
               iron >= cost.getIron() &&
               wood >= cost.getWood();
    }

    public Cost getAvailableLoot() {
        return new Cost(gold / 2, iron / 2, wood / 2, 0);
    }

    public boolean canSupport(int population) {
        return foodProduction >= foodConsumption + population;
    }
}
```

D BUILDINGS PACKAGE - BUILDING ABSTRACT CLASS

```java
package buildings;

import resources.Cost;

public abstract class Building {
    protected int level;
    protected int maxLevel;
    protected int hitPoints;
    protected int maxHitPoints;
    protected int positionX;
    protected int positionY;
    protected Cost buildCost;
    protected int buildTime;
    protected boolean isUpgrading;
    protected long upgradeStartTime;

    public Building(int maxLevel, int maxHitPoints, Cost buildCost) {
        this.maxLevel = maxLevel;
        this.maxHitPoints = maxHitPoints;
        this.hitPoints = maxHitPoints;
        this.buildCost = buildCost;
        this.level = 1;
        this.isUpgrading = false;
    }

    public void upgrade() {
        if (!isMaxLevel() && !isUpgrading) {
            isUpgrading = true;
            upgradeStartTime = System.currentTimeMillis();
        }
    }

    public Cost getUpgradeCost() {
        return buildCost.multiply(level * 1.5);
    }

    public boolean isMaxLevel() {
        return level >= maxLevel;
    }

    public void takeDamage(int damage) {
        hitPoints = Math.max(0, hitPoints - damage);
    }

    public void repair() {
        hitPoints = maxHitPoints;
    }

    public void completeUpgrade() {
        if (isUpgrading) {
            level++;
            isUpgrading = false;
            maxHitPoints = (int)(maxHitPoints * 1.2);
            hitPoints = maxHitPoints;
        }
    }

    public int getLevel() { return level; }
    public int getHitPoints() { return hitPoints; }
}
```

E BUILDINGS PACKAGE - VILLAGEHALL CLASS

```
package buildings;

import resources.Cost;

public class VillageHall extends Building {
    private int villageLevel;

    public VillageHall() {
        super(10, 5000, new Cost(1000, 500, 500, 600));
        this.villageLevel = 1;
    }

    public int getAllowedBuildingLevel() {
        return villageLevel;
    }

    public int getVillageLevel() {
        return villageLevel;
    }

    @Override
    public void completeUpgrade() {
        super.completeUpgrade();
        villageLevel = level;
    }
}
```

F BUILDINGS PACKAGE - FARM CLASS

```
package buildings;

import resources.Cost;

public class Farm extends ProductionBuilding {
    private int foodPerHour;
    private int populationSupport;

    public Farm() {
        super(8, 1000, new Cost(100, 50, 200, 120));
        this.foodPerHour = 50;
        this.populationSupport = 10;
        this.maxWorkers = 3;
        this.productionRate = 20;
    }

    public int getFoodProduction() {
        return foodPerHour + getProduction();
    }

    public int getPopulationSupport() {
        return populationSupport * level;
    }
}
```

G BUILDINGS PACKAGE - ARCHERTOWER CLASS

```
package buildings;

import resources.Cost;

public class ArcherTower extends DefenseBuilding {
    private int arrowCount;

    public ArcherTower() {
        super(8, 1500, new Cost(300, 200, 100, 240));
        this.damage = 50;
        this.range = 10;
        this.attackSpeed = 1.0;
        this.arrowCount = 100;
    }

    public void reload() {
        arrowCount = 100;
    }
}
```

H INHABITANTS PACKAGE - INHABITANT ABSTRACT CLASS

```
package inhabitants;

import resources.Cost;

public abstract class Inhabitant {
    protected int level;
    protected int maxLevel;
    protected int hitPoints;
    protected int maxHitPoints;
    protected Cost trainingCost;
    protected int trainingTime;
    protected int foodConsumption;
    protected boolean isUpgrading;

    public Inhabitant(int maxLevel, int maxHitPoints) {
        this.maxLevel = maxLevel;
        this.maxHitPoints = maxHitPoints;
        this.hitPoints = maxHitPoints;
        this.level = 1;
        this.isUpgrading = false;
    }

    public void upgrade() {
        if (!isUpgrading && level < maxLevel) {
            isUpgrading = true;
        }
    }

    public Cost getUpgradeCost() {
        return trainingCost.multiply(level * 1.5);
    }

    public void takeDamage(int damage) {
        hitPoints = Math.max(0, hitPoints - damage);
    }

    public boolean isAlive() {
        return hitPoints > 0;
    }
```

```
    public void heal() {
        hitPoints = maxHitPoints;
    }
}
```

I INHABITANTS PACKAGE - WORKER CLASS

```
package inhabitants;

import resources.Cost;
import buildings.Building;

public class Worker extends Inhabitant {
    private boolean isIdle;
    private String currentTask;

    public Worker() {
        super(5, 100);
        this.trainingCost = new Cost(50, 0, 0, 30);
        this.foodConsumption = 1;
        this.isIdle = true;
        this.currentTask = "";
    }

    public void buildStructure(Building b) {
        isIdle = false;
        currentTask = "Building";
    }

    public void produceFood() {
        isIdle = false;
        currentTask = "Producing Food";
    }

    public void repair(Building b) {
        isIdle = false;
        currentTask = "Repairing";
    }

    public void setIdle(boolean idle) {
        this.isIdle = idle;
        if (idle) {
            currentTask = "";
        }
    }
}
```

J INHABITANTS PACKAGE - SOLDIER CLASS

```
package inhabitants;

import resources.Cost;

public class Soldier extends ArmyUnit {
    private int shield;

    public Soldier() {
        super(6, 150);
        this.trainingCost = new Cost(100, 50, 0, 60);
        this.damage = 30;
        this.attackRange = 1;
        this.movementSpeed = 2;
        this.shield = 20;
        this.foodConsumption = 2;
    }

    public void block() {
        // Implementation for blocking attacks
```

```
    }
}
```

K CORE PACKAGE - VILLAGE CLASS

```
package core;

import java.util.List;
import java.util.ArrayList;
import buildings.Building;
import buildings.VillageHall;
import inhabitants.Inhabitant;
import resources.ResourceManager;

public class Village {
    private String name;
    private int level;
    private int population;
    private int maxPopulation;
    private int area;
    private int maxArea;
    private VillageHall villageHall;
    private List<Building> buildings;
    private List<Inhabitant> inhabitants;
    private ResourceManager resources;
    private int defenseScore;
    private int attackScore;
    private long guardPeriodEnd;
    private boolean inGuardPeriod;

    public Village(String name) {
        this.name = name;
        this.level = 1;
        this.population = 0;
        this.maxPopulation = 50;
        this.area = 0;
        this.maxArea = 100;
        this.buildings = new ArrayList<>();
        this.inhabitants = new ArrayList<>();
        this.resources = new ResourceManager();
        this.villageHall = new VillageHall();
        this.inGuardPeriod = true;
        this.guardPeriodEnd = System.currentTimeMillis()
            + (7 * 24 * 60 * 60 * 1000);
    }

    public boolean addBuilding(Building b) {
        if (canBuild(b)) {
            buildings.add(b);
            return true;
        }
        return false;
    }

    public void removeBuilding(Building b) {
        buildings.remove(b);
    }

    public boolean addInhabitant(Inhabitant i) {
        if (population < maxPopulation) {
            inhabitants.add(i);
            population++;
            return true;
        }
        return false;
    }

    public int calculateDefenseScore() {
```

```
        defenseScore = 0;
        for (Building b : buildings) {
            defenseScore += b.getLevel() * 10;
        }
        return defenseScore;
    }

    public boolean canBuild(Building b) {
        return area + 10 <= maxArea;
    }
}
```

L CORE PACKAGE - PLAYER CLASS

```
package core;

import buildings.Building;
import inhabitants.Inhabitant;
import engine.AttackResult;

public class Player {
    private String playerName;
    private String playerId;
    private Village village;
    private int rank;
    private int attackWins;
    private int attackLosses;
    private int defenseWins;
    private int defenseLosses;
    private int totalLoot;

    public Player(String name) {
        this.playerName = name;
        this.playerId = generatePlayerId();
        this.village = new Village(name + "'s Village");
        this.rank = 0;
    }

    private String generatePlayerId() {
        return "P" + System.currentTimeMillis();
    }

    public boolean buildBuilding(Building b) {
        return village.addBuilding(b);
    }

    public boolean upgradeBuilding(Building b) {
        b.upgrade();
        return true;
    }

    public boolean upgradeInhabitant(Inhabitant i) {
        i.upgrade();
        return true;
    }

    public void updateRank() {
        rank = (attackWins * 10) - (attackLosses * 5)
            + (defenseWins * 8) - (defenseLosses * 3);
    }

    public Village getVillage() {
        return village;
    }
}
```

M ENGINE PACKAGE - GAMEENGINE CLASS

```
package engine;

import java.util.List;
import java.util.ArrayList;
import core.Player;
import core.Village;
import buildings.Building;

public class GameEngine {
    private long currentTime;
    private double gameSpeed;
    private AttackSimulator attackSimulator;
    private VillageGenerator villageGenerator;
    private ScoreCalculator scoreCalculator;
    private List<Player> players;

    public GameEngine() {
        this.currentTime = System.currentTimeMillis();
        this.gameSpeed = 1.0;
        this.attackSimulator = new AttackSimulator();
        this.villageGenerator = new VillageGenerator();
        this.scoreCalculator = new ScoreCalculator();
        this.players = new ArrayList<>();
    }

    public void startGame() {
        currentTime = System.currentTimeMillis();
    }

    public void updateTime() {
        currentTime = System.currentTimeMillis();
    }

    public void processUpgrades(Village v) {
        // Process building and inhabitant upgrades
    }

    public boolean allowBuild(Village v, Building b) {
        return v.canBuild(b);
    }

    public void addPlayer(Player p) {
        players.add(p);
    }
}
```

N ENGINE PACKAGE - ATTACKRESULT CLASS

```
package engine;

import resources.Cost;

public class AttackResult {
    private boolean success;
    private double successRate;
    private Cost loot;
    private int attackerLosses;
    private int defenderLosses;
    private int stars;

    public AttackResult(boolean success, Cost loot) {
```

```
        this.success = success;
        this.loot = loot;
        this.attackerLosses = 0;
        this.defenderLosses = 0;
        this.stars = success ? 3 : 0;
    }

    public boolean isSuccess() { return success; }
    public Cost getLoot() { return loot; }
    public int getStars() { return stars; }
}
```

O UI PACKAGE - GAMEUI CLASS

```
package ui;

import core.Player;
import engine.GameEngine;

public class GameUI {
    private Player currentPlayer;
    private GameEngine gameEngine;

    public GameUI(Player player, GameEngine engine) {
        this.currentPlayer = player;
        this.gameEngine = engine;
    }

    public void displayVillage() {
        System.out.println("=== Village ===");
        // Display village information
    }

    public void displayBuildMenu() {
        System.out.println("=== Build Menu ===");
        System.out.println("1. Farm");
        System.out.println("2. Gold Mine");
        System.out.println("3. Archer Tower");
    }

    public void displayAttackMenu() {
        System.out.println("=== Attack Menu ===");
        // Display attack options
    }

    public void handleUserInput() {
        // Handle user input for game actions
    }
}
```