COS20007-S2-2024
SUPPLYMENTARY MATERIAL FOR CUSTOM PROGRAMS

Below is a list of suggested real-world topics and ideas for custom tasks. This list is just a recommendation. The project is yours, so please optionally select a topic and then define your own software requirements using OOP principles. **Please consult with the tutor of your allocated labs to see whether your software requirement is significant**.

We hope this will be a great opportunity for you to put OOP into practice. Our tutors are available to guide and discuss your project based on your interests and preferences.

There are two types of ideas, either from Business domain (**see page 1)** or game applications **(see page 3).**

Suggested Business Ideas:

- Car sale applications
- Real estate applications
- Airbnb applications
- Uber applications
- Finding nearby cabs
- Flexicar
- Social network with messaging applications
- Salesforce loyalty management (Customer with credit/velocity points)
- E-learning management system
- Automated stock trading system
- Jobseeker portal
- Online voting system
- Online e-assessment/exam
- Private health management system
- Payroll system
- Insurance claim management system
- Customer relationship management system
- Professional team collaboration application
- Freight management software
- Flight booking system
- Password management system for different users/business domains
- SplashKit-enabled applications with new functionalities in drafting, engineering, architecture, illustration, presentations, radiology, and visualization

COS20007-S2-2024
SUPPLYMENTARY MATERIAL FOR CUSTOM PROGRAMS

Below is the list of some real-world useful libraries/framework if you want to use in your OOP project.

- Use .Net framework to map Object Oriented classes with database management tools, such as ADO.net ,Entity framework, and Language Integrated Query (LinQ)
    - https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/
    - https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/how-to-map-database-relationships
    - https://learn.microsoft.com/en-us/ef/
    - https://learn.microsoft.com/en-us/dotnet/csharp/linq/
- Apple's Medical API, https://developer.apple.com/documentation/healthkit/samples/accessing_health_records
- Australian My Health Record API, https://www.aihw.gov.au/reports-data/myhospitals/content/api
- The U.S. Medline Plus Health API services, https://medlineplus.gov/about/developers/webservices/
- Involve an artificial intelligent (AI) player/agent in your projects. For example, chess with AI, chat with AI chatbot
    - Reference. https://github.com/SciSharp/BotSharp
    - https://learn.microsoft.com/en-us/dotnet/ai/quickstarts/get-started-openai?tabs=azd&pivots=openai
    - https://swharden.com/blog/2024-02-19-local-ai-chat-csharp/
- Integrate with free online web services
    - Stock market API, https://www.alphavantage.co/ , https://finnhub.io/
    - Weather API https://www.weatherapi.com/
- Integrate with Google Place APIs, https://developers.google.com/maps/documentation/places/web-service/overview
- Write/store data to Microsoft Excels or SQL database management
- Logistics with Route Planning framework,
    - https://github.com/route4me/route4me-net-core
    - https://github.com/bulentsiyah/Using-Genetic-Algorithm-the-road-to-Point-to-Point-and-Route-Planning
- Crypto applications features/ to encrypt/decrypt data with C# https://asecuritysite.com/csharp/

Implementation Tip:

Ensure you plan how the classes interact and where the design patterns fit into the overall architecture before coding. This approach will make the project more organized and showcase your understanding of OOP principles effectively.

Suggested Game Ideas with Characters/Actors/Players:

- Chess
- King's Quest
- Counter-strike
- Dota or Diablo

Here are a few more game ideas that can incorporate multiple classes and design patterns:

1. **Adventure RPG**

- **Concept**: A role-playing game where the player navigates through a world, interacts with NPCs (Non-Player Characters), and completes quests.

- **Classes**:

    - `Player`, `NPC`, `Quest`, `Inventory`, `Item`, `World`, `Enemy`

- **Design Patterns**:

- **Factory Pattern**: For creating different types of items and NPCs.

- **Observer Pattern**: For notifying the player of quest updates or inventory changes.

- **Strategy Pattern**: For different types of combat or interaction behaviors.

2. **Tower Defense Game**

- **Concept**: The player places defensive structures to stop waves of enemies from reaching a certain point.

- **Classes**:

    `Tower`, `Enemy`, `Wave`, `Map`, `Player`, `Projectile`, `GameManager`

- **Design Patterns**:

- **Singleton Pattern**: For the `GameManager` class to manage the game state.

- **Decorator Pattern**: To add different abilities or enhancements to towers.

- **Observer Pattern**: For updating the game state when enemies are defeated or when waves are spawned.

3. **Card Game**

- **Concept**: A digital version of a collectible card game (like Magic: The Gathering or Hearthstone).

- **Classes**:, Card`, `Player`, `Deck`, `Hand`, `Game`, `Board`, `Effect`

- **Design Patterns**:

- **Factory Pattern**: For creating different types of cards.

- **Command Pattern**: For handling actions like playing a card, attacking, or triggering effects.

- **Observer Pattern**: To update the game state when cards are played or effects are triggered.

4. **Space Shooter**

- **Concept**: A 2D or 3D space shooter where the player controls a spaceship and fights waves of enemies.

- **Classes**:

    - `PlayerShip`, `EnemyShip`, `Bullet`, `PowerUp`, `Level`, `GameManager`, `Score`

- **Design Patterns**:

- **Factory Pattern**: For creating different types of enemy ships and power-ups.

- **Observer Pattern**: For updating the player's score or health based on game events.

- **State Pattern**: For managing different game states, like playing, paused, or game over.

5. **Platformer Game**

- **Concept**: A side-scrolling platformer where the player controls a character to navigate levels, avoid obstacles, and defeat enemies.

- **Classes**:

    - `Player`, `Enemy`, `Level`, `Obstacle`, `PowerUp`, `Score`, `GameManager`

- **Design Patterns**:

- **State Pattern**: For handling different player states (e.g., standing, jumping, running).

- **Factory Pattern**: For generating different types of enemies and obstacles.

- **Observer Pattern**: For updating the score or triggering level transitions.

Implementation Tip:

Ensure you plan how the classes interact and where the design patterns fit into the overall architecture before coding. This approach will make the project more organized and showcase your understanding of OOP principles effectively.