1. Software development: in your opinion, what are some of the best practices to follow when doing software development? (30 points)

In my opinion, best practices of software development include pursuing high code quality, applying design patterns cleverly, and following good plan & review norms.

- **Pursuing High Code Quality**

When developing a complicated software system, pursue high code quality and cleaner design ***from the beginning point***. Following this practice, future changes and maintenances will be much easier compared to a quick but messy start.

Code of high quality should have these properties: clean, good readability, good maintainability, good reusability, high cohesion & loose coupling.

a) **Clean**: It should be as simple as possible. It should present ***no code smell***, violating fundamental design principles and indicating deeper problems.
b) **Readability**: It should be easy to understand, follow professional naming/style ***conventions***. It should also be ***commented clearly*** whenever necessary, but no unnecessary comments.
c) **Maintainability**: It should be easy to isolate/correct defects, repair/replace worn-out components without affecting others, be coped with a changed environment. It should also be fairly easy to ***refactor***, reduce technical debt, and add new features.
d) **Reusability**: It should be modular, hiding information and ***separating concerns*** properly.
e) **High Cohesion**: Each class/function should focus on a small number of specific tasks.
Loose Coupling: Interdependency between components should be low.

- **Applying Design Patterns Cleverly**

Applying appropriate design patterns ***saves time spent on solving common problems and communication*** significantly. Good use of design patterns will also help programmer ***avoid human errors*** and mistakes better, decreasing risk of defect. While good use of design patterns improves code quality, bad use does harm to code quality.

To apply design patterns cleverly and avoid bad practices, we should:

a) When stuck on a problem, ***go check if there is design pattern aimed at that problem***. Most commonly occurring problems have a complete and well-tested solution structure. Don't waste too much time on struggling with those problems.
b) Before applying a design pattern, ***learn about its four essential elements*** (name, problem, solution, consequence) thoroughly, especially its consequences (pros and cons, impact on reusability, portability, extensibility). Make sure use design pattern matches the situation.
c) Only apply design pattern to solve problem, ***never use a pattern just because you can***. Keep code simple. Every line should have its own clear goal.
d) ***Avoid anti-patterns***, which generate negative consequences with a super high probability.
e) When apply a design pattern, always ***leave comments*** to let others know. In this way, the code can be readable and maintainable.

- **Good Plan and Review Norms**

For a team developing a complicated software system, setting and following good plan and review norms are extremely critical. Good norms include the following:

a) **Plan early.** So that every task can be considered carefully and whole project can be done in a reasonable timespan. Always leave some time for unexpected problems.
b) **Start early.** NEVER LEAVE WORK TO THE LAST MINUTE. A process of begin-expand-revise-polish is necessary for any good work.
c) **Read teammates' code and comments seriously.** Code of every member will work together for one purpose in the end. Understand other's code will help team merge every part together smoothly. This will save a lot of time on communication and make future maintenance much easier.
d) **Do code review seriously.** Good review should identify potential problems of the code, which the author may not realize. Writing helpful code review will help the whole team improve code quality and prevent possible negative consequences of bad practices as soon as possible.

2. Team project processes and results: Discuss your experience on the team project and what you learned from it. Do so as if you were answering this question as part of a job interview. Be sure to also briefly describe how successful you think your team's project was, and what you would do differently if starting over. (30 points)

- **Experience on Team Project**

In two and a half months, our team of six developed a 2D interactive game imitating *The Legend of Zelda* NES edition. The final work presents a complete game allows user control the character, Link, to adventure in various environments facing various enemies. In the view of whole project, we followed paradigm and principles of **Object-Oriented Programming** to build a stable and flexible complicated system. We applied **event-based programming** to handle interaction with user. Main tools we used:

- Language: **C#**
- Environment: **Visual Studio**
- Framework: **MonoGame**
- Development Platform: **GitHub**

In these two and a half months, we divide the development process into five stages. For each stage, we set and finished a milestone. After each milestone done, we did code review for everyone's code and a reflection on whole team's performance.

In the first stage (Sprint 1), the goal was to get familiar with **team-working development** platform we selected to use, GitHub, and make sure everyone is competent with branch, push, merge and other operations.

In the second stage (Sprint 2), first, we built a fundamental framework for the game. We designed **interfaces** for essential components, including sprite, player, enemy, item, and object. Classes implementing these interfaces were created to test and ensure them function properly. Then, we created classes implementing sprite drawing, moving, and animation. In this process, we got confident with basic **2D graphic techniques** and **game content management**. Command, State, and Factory Method design pattern were applied at this point.

In the third stage (Sprint 3), **core features of game environment and collision were implemented.** Most classes we wrote in Sprint2 were refactored to match new background. Firstly, we finished **environment and enemy setting** for whole game. Different segmented screens of "rooms" were added with own corresponding subset of objects displaying. Then, we implemented **collision detecting and handling** between character and environment, based on location information of each sprite. Damaged character behavior was added using Decorator design pattern.

In the fourth stage (Sprint 4), we **made whole project more like a real game** by restructuring game framework and adding more game elements. Code in Sprint3 provided a good foundation for this stage. Firstly, we built game state structure to provide a **controllable game flow**. Different game states' screens and corresponding interaction were designed. We also added **scrolling and transition** between rooms, so that user can walkthrough whole game by controlling character. Collision between enemy and character' projectiles were added. HUD, camera, and sound were also implemented here.

In the fifth stage (Sprint 5), **more entertaining features** were added. We added a secret room user can continue exploring after win the game. We also added a timer to track time user spent to finish the game and provide a time cost rank screen. **Code quality** was also emphasized and improved at this time. We remove unnecessary elements to make code clean, without any code smell. All private fields and public properties are properly set.

– **What we did successfully**

Reflecting at this point, out team did successfully on several points. We let everyone choose parts each of we interested in, and warp everything up before a set time for the milestone. Each of us keep being responsible for the parts we select and keep improving them towards the final goal. In this way, each of us could do research on several focused topic thoroughly. When problems shown up, the one who need to take the responsibility was clear.

– **What we did not do well**

One thing we did not done well and if starting over I would definitely do differently are the ***team collaboration and project planning***. Under this special time, we could not sit together and discuss face to face. We suffered from the limitation and inconvenience of online communication producing almost nothing useful for several weeks (which is a great amount considering we only have 2.5 month). If we could take this issue seriously at the beginning of project, maybe we could reach an agreement on clearer communication and task division forms. Following forms everyone agrees with, we could work together like a mature team and avoid the situations that everyone was just working by self.

The other thing I would do totally differently was the I would use a much better ***structured approach to design and develop the large software***. Known nearly nothing about software development, my early steps and code were messy, implementing functions roughly without structure and design. Almost all early code was rewritten later. If starting over, with a deeper sense of software development process and code quality understanding, I would apply a structured approach to develop systematically. I would design following principles of high cohesion and low coupling, separating concerns in code as early as possible, so that cost of maintaining code could be decease significantly. I would also ***set each state's task and goal more clearly and focused*** (e.g. interfaces → foundation: sprites, state, … → functionality: command, collision, …→ expand: other features → polish: refactor code…). So that, each stage of the whole process could be done and polished in a reasonable time span.

- **What I Learned**

First time doing a complicated engineering project, I learned quite a lot from this course and this project. What I gained can be divided into following three parts:

a) **Interactive 2D System Implementation**

The most practical knowledge I learned is definitely 2D graphics techniques and interactive system implementation skills. I am now competent with 2D graphics objects, rendering, and animation. I also practiced game content creation and editing, understanding what happens behind 2D/3D animation in real commercial games. I gained a more thorough understanding about ***event-based programming*** in the process of handling user interaction and control.

b) **Software Engineering Development**

This part is the most important thing I learned. As the course was designed, the main goal of this project is "design, development and documentation" an interactive system. After this project, I am now clear about ***how to design, implement and evaluate a software system to meet desired demands*** (e.g. different functions work at same time in the game). This project brought me to a higher level where I see a bigger-picture of software engineering development.

I now have a sense of ***how a CSE professional should apply appropriate techniques, skills and tools to solve engineering problems***. For example, a good programmer should concern with code quality, design pattern, related programming principles and be clear about how they influence further development. A good programmer should also ***be clear about what decision he/she make and why*** in process of development. In the process of development, I start asking myself these questions: "why I design the interface with these methods?", "why this method must have these parameters?", "why I need this variable as public property here?". So that, I can make sure that we are ***"design"*** but not doing something just because it is easy. In this way, we will also know what we really value when facing future potential tradeoffs.

c) **Teamwork and Documentation**

Different from team experience I have before, in this huge and complicated project, everyone's works are interdependent closely. I learned a lot about ***project progress tracing and source code control***. To avoid one operation ruining source code which provides foundation for whole team, all merge should be done under full ***communication, test and documentation***. Individual's good documentation will help whole team understand his/her work, making progress tracing and future maintenance much easier. At the beginning of project, I only document what features I added. But soon I found it is not enough to draw other's attention and make them know what I am doing. After then, for each part of my work, I provided a detailed description besides the brief introduction, including a list of new files I added and its function, a list of files I updated and why, what operation or computing it based on…It turned out to help a lot for future reference. I gained a deeper understanding of what good documentation looks like and why we need documentation for users and for teammates.

3. Software development question: Select a 2D game and describe how you would design your own implementation of it. (40 points)

Game: ***Splatoon*** 2D Version
Sample Game Walkthrough: https://www.youtube.com/watch?v=_Aybng7A0lI
The specific part I want to implement is a **simplified Turf War with two players**. Players can transform between two forms. In humanoid form, they can shoot colored link to hurt enemy or color a set amount of surface under one tap/click operation. In cephalopodic form, they can swim on surfaces in own ink. The game **ends immediately when one of player die or reach a set time limit**. The player still alive or cover more surface wins.

In my understanding, the core task of designing implementation of a game is to **design a system of interfaces and classes** for it, applying proper design and conventions.

- **Interface**

Define contract of a certain component type needed in the game. Contain definitions for necessary methods, properties, or events.

a) **IGame**: define a type for different game states (title screen, pause, in progress, finish…).
Methods: Update, Draw, (Pause, Resume).
b) **IBackground**: define a type for background of the Turf.
Methods: Update, Draw.
c) **IController**: define a type for different controllers (keyboard/gamepad).
Methods: Update.
d) **ICommand**: define a type for different commands, carry out corresponding actions.
Methods: Execute.
e) **ISprite**: define a type for all sprites used in game.
Methods: Update, Draw.
f) **IPlayer**: define a type for each player/character.
Methods: Update, Draw. Properties: health, color, state.
g) **IPlayerForm**: define a type for different forms of player.
Properties: form (humanoid, cephalopodic, dead).
h) **IPlayerPhysicalState***: define a type for different physical behaviors of player.*
Methods: Fall, Jump
i) **IPlayerMovementState**: define a type for different movement behaviors of player.
Methods: Stay, GoLeft, GoRight, Climb, Shoot.
j) **IObject**: define a type for different static/dynamic objects set in the background of the Turd.
Methods: Update, Draw. Properties: sprite, position.
k) **ISurface**: extends IObject. Define a type for surface of background in the Turf.
Methods: Update, Draw. Properties: color (color1, color2, none).
l) **Iink:** extends IObject. Define a type for ink shoot by player.
Methods: Update, Draw. Properties: state (fall, onSurface), color (color1, color2).

- **Class**

Implementation with functional code. If implementing interface supporting all parts of contract.

a) IGame
- **TitleScreenState/PauseState/EndState**: present corresponding screen and user's options.
- **GameInProgressState**: invoke, update, draw all necessary components in Turf.

b) IBackground
- **Background**: update, draw sprite of background

c) IController
- **KeyboardController/GamePadController**: map operations to corresponding commands.

d) ICommand
- *Game Commands* - **BeginGameCommand/PauseGameCommand/ExitGameCommand**: invoke begin/pause/exit game operations, carry out corresponding game states
- *Player Commands* – **Humanoid/Cephalopodic/DeadPlayerCommand**: invoke corresponding physical state operations, carry out corresponding players' states.
- *Player Commands* – **Stay/MoveRight/MoveLeft/Jump/Climb/Shoot/HurtPlayerCommand**: invoke corresponding movement state operations, carry out players' states.
- *Surface Commands* – **ColorSurfaceCommand**: invoke color surface operations
- *Ink Commands* – **ShootInkCommand/FallInkCommand/DropOnSurfaceInkCommand**: invoke corresponding ink state operations, carry out corresponding ink's states.

e) ISprite
- **StaticObjectSpirte/DynamicObjectSprite**: implement all objects' sprites and animation.
- **HumanPlayerStay/MoveLeft/MoveRight/Climb/Shoot/HurtSprite**: implement corresponding player states' sprites and animation in humanoid form.
- **CephaloPlayerStay/MoveLeft/MoveRight/Climb/HurtSprite**: implement corresponding player state's sprites and animation in cephalopodic form.
- **DeadPlayerSprite***:* implement player's dead sprite
- **SurfaceNoneColor/Color1/Color2Sprite***:* implement different color surfaces' sprites.
- **InkShoot/Falling/DropOnSurfaceSprite***:* implement different inks' sprites and animation.

f) IPlayer
- **Player1/Player2**: implement specific players, set corresponding sprite, state, heath, color.

g) IPlayerForm
- **HumanoidPlayerForm/Cephalopodic/DeadPlayerForm**: implement different player forms, carry out corresponding forms' sprites.

h) IPlayerPhysicalState
- **PlayerFall/JumpState**: implement different physical states of player, carry out animation.

i) IPlayerMovementState
- **PlayerStay/MoveLeft/MoveRight/Climb/Shoot/HurtState:** implement different movement states of players, carry out different states' animation.

j) IObject
- **Block/Balloon/Tree/Building**: implements different objects' behaviors set in Turf, carry out corresponding sprites and animation.

k) ISurface

- **NoneColor/Color1/Color2Surface**: implements different surface states in Turf, carry out corresponding sprites and animation.

l) Iink

- **Shoot/Falling/DropOnSurfaceInk**: implements different behaviors of ink in different states, carry out corresponding sprites and animation.

m) Collision Detector and Handler

- **PlayerObjectCollisionDetector**: detect if player collides with other player, any objects in Turf (surface, block, tree, building).
- **PlayerInkCollisionDetector**: detect if player detect enemy ink, carry out specific response.
- **InkCollisionDetector**: detect if ink collides with any objects (surface, block, tree, building).
- **PlayerSurface/Block/Tree/Building/PlayerCollisionHandler**: response collisions of player, carry out stop or change player's movement state.
- **InkSurface/Block/Tree/BuildingCollisionHandler***:* response collision of ink and objects, carry out change of ink and object's state.
- **PlayerInkCollisionHandler**: response collision of player and enemy ink, carry out hurt and change in player's state.

n) Factory

- **BackgroundFactory**: implement generation of background sprite
- **ObjectSpriteFactory**: implement generation of all object sprites
- **PlayerSpriteFactory**: implement generation of all player sprites
- **InkSpriteFactory***:* impelemnt generation of all ink sprites

o) Other

- **Camera**: implement scrolling background and adjusting view as player move.
- **HUD**: implement HUD information display and control.
- **Sound**: implement sound generation and all sound effect.
- **UtilityClass**: hold all important variable values used in system.
- **Program**: main entry point of whole game.
- **Game1**: declare basic variables, load all content, entry point of all Update, Draw methods.

- **Design Pattern**

a) **Behavior - Command Pattern**

Used in classes: ***all command related classes*.**

Command pattern is used heavily in this system to decrease code coupling. The benefit is that all classes invoking operation are decoupled from the class executing the operations, encapsulate information needed to perform an action.

b) **Behavior - State Pattern**

Used in classes: ***all objects/surface/ink/player state classes*.**

State pattern is used heavily in this system to increase code cohesion. The benefit is that each state class can focus on implementing specific behaviors of the specific state, and other classes can focus on other tasks (i.e. change state).

c) **Creation - Factory Method Pattern**

Used in classes: ***all sprite factories*.**

Factory method pattern is used heavily in this system to increase code reusability and cohesion. The benefit is that all sprites all generated and prepared together at a starting point, then all other classes can use sprites by calling simpler method instead of repeating constructors.

d) **Structure - Decorator Pattern**

Used in classes: ***classes of damaged player*.**

Decorator pattern is used here to allow a specific behavior (e.g. damage) of a class change dynamically, without affecting other behaviors (e.g. form, movement) of that class. It will provide new behavior at run time for each object incrementally.

e) **Behavior - Strategy Pattern**

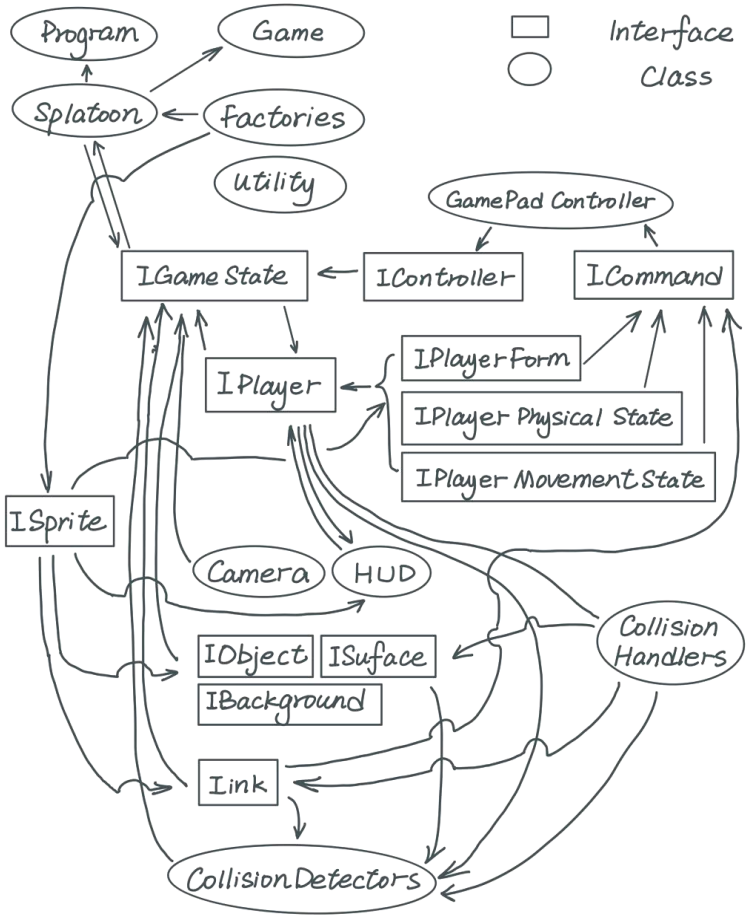
May used in classes: ***class to calculate player' score, time and health*.**

Strategy pattern may be used here to calculate player's score, time and health by an algorithm selected at runtime. A family of algorithms is designed and encapsulated in specific classes.

f) **Creation - Singleton Pattern**

Singleton pattern will be beneficial when we need it to ensure a class has only on instance during execution and provide a global access point to the instance. May used in classes: game.

- **Core Relation between Interfaces** (and some Classes not implementing interfaces)



4. Extra

Time spend on this final report: about 10 hours

The point I want to mention specifically is that this course pushed me to further practice ***life-long learning skills***. Beginning from Code Quality and Design Pattern sub-webpages provided by Dr. Matt, I was exposed to the boundless internet world about game design and software development. To implement functionality and features required in our project, I had to ***read & run existing code, identify the problem, search on the internet, select the most value materials, and learn, try, finish the work by myself*.** This process improves my learning and problem-solving skills significantly. It will be kept in my mind deeply in my future study.

This course also changes my view of game and game industry significantly. Unstoppable, when I play game like Minecraft and Splatoon again, I start thinking about how these games were designed and developed. I become wondering how *RPG, non-linear games, first/third person view*, and other features/functions were designed and implemented.

Thank all instructors and assistants in this class for opening a door of bigger computer science and game industry world to me. Appreciate everyone's effort in this special time.