CMPT 414 – Game Programming I

Quiz #3, Form: A

Name: _____

Date: _____

Section 1. OOP Topics in C++

1. **(Weight=1)** For each access specifier listed below, briefly explain how it affects who (i.e., which other code) can access members under the corresponding section of a class declaration.

   (a) `public`

   (b) `private`

   (c) `protected`

2. **(Weight=2)** All C++ classes have a special function called a *destructor*.

   (a) Suppose we write a class named `ZombieState`. What is the name of the destructor for this class?

   (b) What is the purpose of a destructor? I.e., what logic or operations likely appear in a destructor?

   (c) Finally, if we want to extend the `ZombieState` class, what modifier keyword must we add to the base class destructor in order to ensure that derived class destructors are called correctly?

3. **(Weight=2)** The C++ standard library provides several *smart pointer* classes to help make memory dynamic memory management simpler and less error-prone.

   (a) A *smart pointer* is just a simple lightweight wrapper encapsulating a raw pointer. How does this approach eliminate the need for manually releasing heap memory? That is, why do we not need write a `delete` statement if we are correctly using a smart pointer?

   (b) In our `BeeState` example from class, we used the `std::unique_ptr` class template. What are the ownership rules for this kind of smart pointer? (Consider the case where we create a new `BeeState` and then assign it as the `currentState` of our `Bee` game object.)

Section 2. Managing State-Dependent Behavior

4. **(Weight=2)** Recall the STATE design pattern, including the extra functionality described by Nystrom for handling state transitions more cleanly. For each method listed below, identify any argument(s) and return value, then briefly explain what it does (i.e., its purpose within the pattern).

   (a) `handle` -

   (b) `update` -

   (c) `enter` -

   (d) `exit` -

5. **(Weight=3)** The STATE pattern is a good fit for altering the behavior of a game object in response to changes in its internal state. Imagine that we wish to apply this pattern to *Zombie Arena* in order to implement a more interesting type of zombie, as follows:

   Our new zombie begins by attacking the player, but it speeds up each time it gets hit while it's health is high enough (e.g., above 25%). When its health drops below 25%, it will flee for 2 seconds before resuming the attack. And finally, when it's health reaches zero, it doesn't die immediately and instead pauses for a second before exploding.

   (a) If our base class for the zombie's state is called `ZombieState`, name the concrete subclasses would you create to implement the behavior described above? That is, what states our zombie will need?

   (b) For each subclass state, identify which other state(s) the zombie may transition to and what triggers the transition.

   (c) For which of these state subclass might the `enter` and `exit` methods be useful, and how?

Section 3. Encapsulating Behavior as Commands

6. **(Weight=2)** Summarize the COMMAND design pattern using *either* a UML diagram *or* some C++-like pseudocode. Your response should include two classes and at least one method.

7. **(Weight=3)** Nystrom discusses various applications of this pattern for different situations.

   (a) Explain how we would implement or apply the pattern differently for AI-controlled characters as opposed to a player-controlled character.

   (b) This pattern is also useful in situations where we want to be able to undo or redo actions. Explain how we can employ a *list* data structure to implement this mechanism.

Section 4. Reducing Memory Footprint

8. **(Weight=3)** Imagine we are creating a new game called M3GA (Medieval 3-d Garden Adventure) in which one level sees the player fighting an army of Bee warriors amidst a rich, colorful environment densely packed with flowers.

   (a) Assuming that various Bee instances differ only by a few properties (e.g., position, size, health, animation frame number), and similarly for the Flowers, what design pattern can help us reduce the memory footprint of our game by taking advantage of the fact of the large amount of data that is common to each instance of a given type of game object?

   (b) Illusrtrate this pattern in the context of our hypothetical garden RPG by *either* drawing a UML diagram *or* writing some C++-like pseudocode.

Answer Key for Exam A

Section 1. OOP Topics in C++

1. **(Weight=1)** For each access specifier listed below, briefly explain how it affects who (i.e., which other code) can access members under the corresponding section of a class declaration.

   (a) `public`

   (b) `private`

   (c) `protected`

2. **(Weight=2)** All C++ classes have a special function called a *destructor*.

   (a) Suppose we write a class named `ZombieState`. What is the name of the destructor for this class?

   (b) What is the purpose of a destructor? I.e., what logic or operations likely appear in a destructor?

   (c) Finally, if we want to extend the `ZombieState` class, what modifier keyword must we add to the base class destructor in order to ensure that derived class destructors are called correctly?

3. **(Weight=2)** The C++ standard library provides several *smart pointer* classes to help make memory dynamic memory management simpler and less error-prone.

   (a) A *smart pointer* is just a simple lightweight wrapper encapsulating a raw pointer. How does this approach eliminate the need for manually releasing heap memory? That is, why do we not need write a `delete` statement if we are correctly using a smart pointer?

   (b) In our `BeeState` example from class, we used the `std::unique_ptr` class template. What are the ownership rules for this kind of smart pointer? (Consider the case where we create a new `BeeState` and then assign it as the `currentState` of our `Bee` game object.)

Section 2. Managing State-Dependent Behavior

4. **(Weight=2)** Recall the STATE design pattern, including the extra functionality described by Nystrom for handling state transitions more cleanly. For each method listed below, identify any argument(s) and return value, then briefly explain what it does (i.e., its purpose within the pattern).

   (a) `handle` -

   (b) `update` -

   (c) `enter` -

   (d) `exit` -

5. **(Weight=3)** The STATE pattern is a good fit for altering the behavior of a game object in response to changes in its internal state. Imagine that we wish to apply this pattern to *Zombie Arena* in order to implement a more interesting type of zombie, as follows:

   Our new zombie begins by attacking the player, but it speeds up each time it gets hit while it's health is high enough (e.g., above 25%). When its health drops below 25%, it will flee for 2 seconds before resuming the attack. And finally, when it's health reaches zero, it doesn't die immediately and instead pauses for a second before exploding.

   (a) If our base class for the zombie's state is called `ZombieState`, name the concrete subclasses would you create to implement the behavior described above? That is, what states our zombie will need?

   (b) For each subclass state, identify which other state(s) the zombie may transition to and what triggers the transition.

(c) For which of these state subclass might the `enter` and `exit` methods be useful, and how?

```
class SingletonThing
{
public:
    static SingletonThing* getInstance() {
        if (THE_ONE_AND_ONLY_INSTANCE == nullptr)
            THE_ONE_AND_ONLY_INSTANCE = new SingletonThing();
        return THE_ONE_AND_ONLY_INSTANCE;
    }
private:
    SingletonThing() { /* implementation... */ }
    static SingletonThing* THE_ONE_AND_ONLY_INSTANCE;
};

SingletonThing* SingletonThing::THE_ONE_AND_ONLY_INSTANCE = nullptr;
```

Section 3. Encapsulating Behavior as Commands

6. **(Weight=2)** Summarize the COMMAND design pattern using *either* a UML diagram *or* some C++-like pseudocode. Your response should include two classes and at least one method.

7. **(Weight=3)** Nystrom discusses various applications of this pattern for different situations.

   (a) Explain how we would implement or apply the pattern differently for AI-controlled characters as opposed to a player-controlled character.

   (b) This pattern is also useful in situations where we want to be able to undo or redo actions. Explain how we can employ a *list* data structure to implement this mechanism.

Section 4. Reducing Memory Footprint

8. **(Weight=3)** Imagine we are creating a new game called M3GA (Medieval 3-d Garden Adventure) in which one level sees the player fighting an army of Bee warriors amidst a rich, colorful environment densely packed with flowers.

   (a) Assuming that various Bee instances differ only by a few properties (e.g., position, size, health, animation frame number), and similarly for the Flowers, what design pattern can help us reduce the memory footprint of our game by taking advantage of the fact of the large amount of data that is common to each instance of a given type of game object?

   (b) Illusrtrate this pattern in the context of our hypothetical garden RPG by *either* drawing a UML diagram *or* writing some C++-like pseudocode.