|  |  |  |
| --- | --- | --- |
|  | Faculty of Computing, Engineering and Science |  |

**Assessment Cover Sheet and Feedback Form** 2020-21

|  |  |  |  |
| --- | --- | --- | --- |
| Module Code:  CS2S560 | Module Title:  Data Structures and Algorithms with Object Oriented Programming | | Module Team:  Emlyn Everitt, Janusz Kulon |
| Assessment Title and Tasks:  NPC Generator | | | Assessment No.  2 |
| Date Set:  **23-Sep-20** | | Submission Date:  **19-Mar-21** | Return Date:  **16-Apr-21** |

**IT IS YOUR RESPONSIBILITY TO KEEP RECORDS OF ALL WORK SUBMITTED**

|  |
| --- |
| **Marking and Assessment** |
| This assignment will be marked out of 100%  This assignment contributes to 50% of the total module marks. |
| **Learning Outcomes to be assessed** (as specified in the validated module descriptor [https://icis.southwales.ac.uk/](https://icis.southwales.ac.uk/studentmodules/13599/studentmodulespecifications) ):  1) Demonstrate knowledge, comprehension and discernment in the efficient application of common data structures and algorithms, and collections.  2) Demonstrate knowledge, comprehension and discernment in the efficient application of object-oriented programming. |
| *Provisional mark only: subject to change and / or confirmation by the Assessment Board* |

Tasks

## Standard Scenario (60%)

You work for Blizzhard, a well-known game company. One day your boss comes to you to say that he/ she wants you to develop the NPC generator for their latest game: Battlecraft III. The generator’s role will be to limit the amount of spawned NPCs (to prevent too many existing at any one time) and ensure that they are only equipped to the level dictated by available resources.

Criteria for the generator:

* The generator should only display basic success/ error messages: **no attempt should be made to implement user menus or prompt for user input in any way**.
* The generator will only respond to method calls made to the public interface of NPCGenerator class (see **Appendix A**). **Note: methods will be expected to return a Boolean true when a requested operation results in a valid transition only, and a Boolean false when not.**
* **None of the provided code (see appendix A) should be changed**. Implementation code will need to be added, but **the existing code and the public APIs (methods) should remain the same**. **The code should also continue to function as provided – i.e. not circumvented or commented out**
* The generator should implement the following state diagram without modification (i.e. you should not attempt to add or remove states / transitions).



* The state diagram and the API specification together represent the formal specification for the project, and a failure to adhere to them will result in lost marks.
* You should employ appropriate design patterns in the delivery of this project. Any solution that is not implemented using appropriate design patterns will be judge a failure, even if it works.
* **DO NOT** **INCLUDE** **the ‘main’ function** in your submission. You code should be complete apart from this omission (you can use your own main function whilst testing you code, just not include it when submitting).
* The only libraries you should include are:<string>, <iostream> and <vector> (to support the storing of State classes within the StateContext class only – e.g. do not try to implement the NPC/ Weapon classes using vectors). **No other libraries should be used.**
* **Exceptions should not be used**.
* **Your code must compile to the specification given in Appendix A.** During testing (by me), a main function will be added to your code that will verify that it meets the provided specification and if your code does not compile because it has deviated from the specification, marks will be lost.
* When submitting your work, please submit your code in **a single .cpp file** to Blackboard in the format: *put\_your\_student\_ID\_here*.cpp – **DO NOT** submit the entire Visual Studio project directory and **DO NOT** submit a zip/rar file. Marks will be lost if you do.
* The generator should spawn a range of different NPCs in accordance with an option code passed to the NPCGenerator via the makeSelection method as specified by the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| NPCs | Option Code | Cost | Base Damage |
| Peon | 1 | 100 | 10 |
| Grunt | 2 | 150 | 20 |
| Raider | 4 | 200 | 30 |
| Tauren | 8 | 250 | 40 |
| Shaman | 16 | 300 | 50 |

Note: only one NPC type should be selected.

## Bonus Section (20%)

Bonus marks will be awarded for the implementation of weapons. Your generator should allow the addition of one or more weapons to your NPC[[1]](#footnote-1), with additional descriptions and costs being calculated and made available via the NPC class public interface using **an appropriate design pattern** and without changing the declaration of the make\_selection method**.**

Option codes for weapons are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Weapons | Option Code | Cost | Damage |
| Dagger | 32 | 20 | 20 |
| Staff | 64 | 20 | 20 |
| Spear | 128 | 50 | 30 |
| Axe | 256 | 70 | 40 |
| Sword | 512 | 100 | 50 |

**Please note: If you are unable to implement this section, ignore these additional option codes in your code.**

## Mega Bonus Section (20%)

Additional bonus marks will be awarded for implementing the following, which will form part of the NPC class public API (see **appendix A**):

virtual int bestDamage(int best\_weapon\_damage = 0); //returns the best damage an NPC can do - i.e. base damage + highest damage weapon. Should return 0 if not implemented

virtual Weapon\* ReturnHighestCostWeapon(void); //returns a pointer to the highest cost weapon. Should return nullptr if not implemented

If you are unable to implement this section of the coursework, you should still ensure that the methods return 0 and nullptr respectively (see above), because this is required by the provided specification.

# Appendix A

#include <vector> //only to be used to contain dispenser states and nothing else

#include <iostream>

#include <string>

using namespace std;

enum state { NO\_NPCS, NO\_RESOURCES, HAVE\_RESOURCES, SPAWN\_NPC };

enum stateParameter { NPCS, RESOURCES };

class StateContext;

class State

{

protected:

StateContext\* CurrentContext;

public:

State(StateContext\* Context)

{

CurrentContext = Context;

}

virtual ~State(void)

{

}

};

class StateContext

{

protected:

State\* CurrentState=nullptr;

int stateIndex=0;

vector<State\*> availableStates;

vector<int> stateParameters;

public:

virtual ~StateContext()

{

for (int index = 0; index < this->availableStates.size(); index++) delete this->availableStates[index];

this->availableStates.clear();

this->stateParameters.clear();

}

virtual void setState(state newState)

{

this->CurrentState = availableStates[newState];

this->stateIndex = newState;

}

virtual int getStateIndex(void)

{

return this->stateIndex;

}

virtual void setStateParam(stateParameter SP, int value)

{

this->stateParameters[SP] = value;

}

virtual int getStateParam(stateParameter SP)

{

return this->stateParameters[SP];

}

};

class Transition

{

public:

virtual bool addResource(int) { cout << "Error!" << endl; return false; }

virtual bool makeSelection(int){ cout << "Error!" << endl; return false; }

virtual bool returnResource(void){ cout << "Error!" << endl; return false; }

virtual bool addNPCs(int){ cout << "Error!" << endl; return false; }

virtual bool spawn(void){ cout << "Error!" << endl; return false; }

};

class NPCState : public State, public Transition

{

public:

NPCState(StateContext\* Context) : State(Context){}

};

class NoNPCs: public NPCState

{

public:

NoNPCs(StateContext\* Context) : NPCState(Context){}

bool addResource(int resource);

bool makeSelection(int option);

bool returnResource(void);

bool addNPCs(int number);

bool spawn(void);

};

class NoResources : public NPCState

{

public:

NoResources(StateContext\* Context) : NPCState(Context) {}

bool addResource(int resource);

bool makeSelection(int option);

bool returnResource(void);

bool addNPCs(int number);

bool spawn(void);

};

class HaveResources : public NPCState

{

public:

HaveResources(StateContext\* Context) : NPCState(Context) {}

bool addResource(int resource);

bool makeSelection(int option);

bool returnResource(void);

bool addNPCs(int number);

bool spawn(void);

};

class SpawnNPC : public NPCState

{

public:

SpawnNPC(StateContext\* Context) : NPCState(Context) {}

bool addResource(int resource);

bool makeSelection(int option);

bool returnResource(void);

bool addNPCs(int number);

bool spawn(void);

};

class Weapon;

class NPC

{

protected:

string \_description = "NPC";

int \_cost = 0;

int \_damage = 0;

public:

string description(void) { return this->\_description; };

int cost(void) { return this->\_cost; }

int damage(void) { return this->\_damage; }

virtual int bestDamage(int best\_weapon\_damage = 0); //returns the best damage an NPC can do - i.e. base damage + highest damage weapon. Should return 0 if not implemented

virtual int costAll(void) { return this->cost(); } //returns NPC base cost

virtual string describeAll(void) { return this->description(); } //returns description of NPC

virtual Weapon\* ReturnHighestCostWeapon(void) { return nullptr; } //returns nullptr

};

class Weapon : public NPC

{

protected:

NPC\* FilledNPC = nullptr;

public:

~Weapon(void);

void equipNPC(NPC\* NewNPC);

int costAll(void); //returns cost of NPC + all weapons

int bestDamage(int best\_weapon\_damage = 0); //returns the best damage an NPC can do - i.e. base damage + highest damage weapon

string describeAll(void); //returns string description of NPC + all weapons

Weapon\* ReturnHighestCostWeapon(void); //returns a pointer to the high cost weapon. Should return nullptr if not implemented

};

class NPCGenerator: public StateContext, public Transition

{

friend class SpawnNPC;

friend class HaveResources;

private:

NPCState\* NPCCurrentState=nullptr;

bool itemDispensed = false; //indicates whether a NPC is there to be retrieved - needed for ownership managemnt

NPC\* SpawnedNPC=nullptr; //

bool itemRetrieved = false; //indicates whether a NPC has been retrieved - needed for ownership managemnt

public:

NPCGenerator(int inventory\_count);

~NPCGenerator(void);

bool addResource(int resource);

bool makeSelection(int option);

bool returnResource(void);

bool addNPCs(int number);

bool spawn(void);

NPC\* getNPC(void);

virtual void setStateParam(stateParameter SP, int value);

virtual int getStateParam(stateParameter SP);

};

NPCGenerator::~NPCGenerator(void)

{

if (!this->itemRetrieved)

{

delete this->SpawnedNPC;

}

}

bool NPCGenerator::addResource(int resource)

{

NPCCurrentState = (NPCState\*)this->CurrentState;

return this->NPCCurrentState->addResource(resource);

}

bool NPCGenerator::makeSelection(int option)

{

NPCCurrentState = (NPCState\*)this->CurrentState;

return this->NPCCurrentState->makeSelection(option);

}

bool NPCGenerator::returnResource(void)

{

NPCCurrentState = (NPCState\*)this->CurrentState;

return this->NPCCurrentState->returnResource();

}

bool NPCGenerator::addNPCs(int number)

{

NPCCurrentState = (NPCState\*)this->CurrentState;

return this->NPCCurrentState->addNPCs(number);

}

bool NPCGenerator::spawn(void)

{

NPCCurrentState = (NPCState\*)this->CurrentState;

return this->NPCCurrentState->spawn();

}

NPC\* NPCGenerator::getNPC(void)

{

//handles transfer of NPC object ownership

if (this->itemDispensed)

{

this->itemDispensed = false;

this->itemRetrieved = true;

return this->SpawnedNPC;

}

return nullptr;

}

void NPCGenerator::setStateParam(stateParameter SP, int value)

{

this->stateParameters[SP] = value;

}

int NPCGenerator::getStateParam(stateParameter SP)

{

return this->stateParameters[SP];

}

1. NPCs can have more than one weapon allocated to them, but only a maximum of one of each type – e.g. a Grunt could have a spear and a sword, but not more than one sword. [↑](#footnote-ref-1)