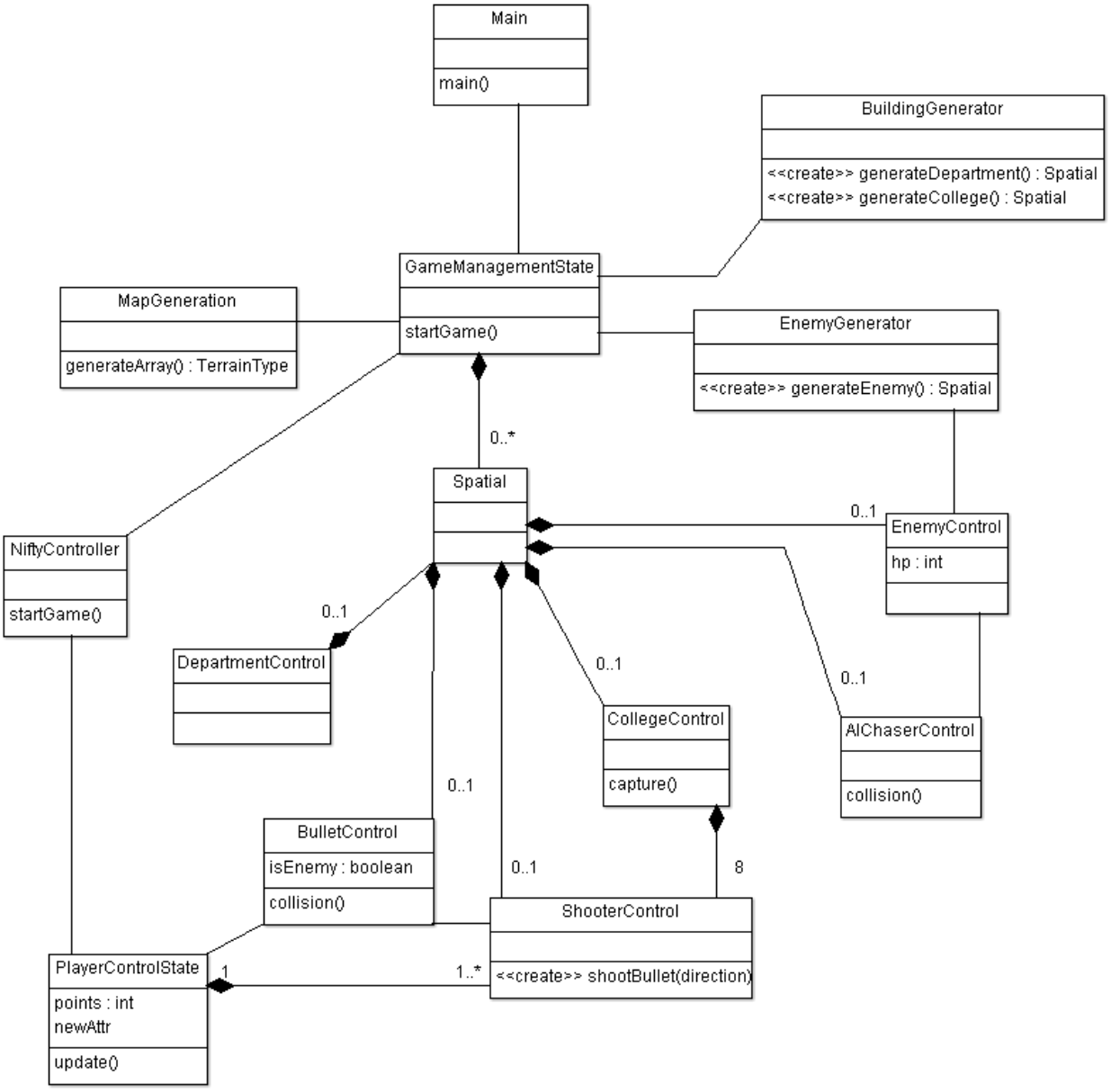
Team iPatch

Assessment 2: Architecture report

Christian Pardillo Laursen
Filip Makosza
Joseph Leigh
Mingxuan Weng
Oliver Relph

Architecture model



The architecture of the code is represented with a UML diagram created in ArgoUML. When the program is launched, Main is initialised. In jMonkeyEngine, the programmer has little access to the Main class at runtime, beyond certain methods related to fetching objects which are defined at a library level, like the rootNode which keeps track of the relationships between game entities and is used to render them every frame. As such, we have the Main method do little more than instantiate the GameManagerState. The GameManagerState is primarily used to instantiate game entities and other **AppStates** which manage specific elements of the game. jMonkeyEngine is designed around the principle of composition over inheritance, and app states are the result. App states are distinct pieces of code responsible for handling a certain part of the program, which can communicate with each other using facilities provided by the engine's app state manager. As described previously,

the Main method instantiates the GameManagementState app state, which is stored in a list containing the active app states in the program. It initialises the NiftyController app state, which is responsible for controlling UI behaviour and the handling of player input on the UI, When the player clicks a button displayed by the UI, a method is executed in the NiftyController, which makes it communicate with the GameManagementState and initialise the rest of the game. This would be impossible to do *without* the GameManagementState, as all custom methods within Main are inaccessible from the app states.

This instantiates a BulletAppState responsible for physics calculations within jMonkeyEngine. Unlike NiftyController, the BulletAppState is part of a predefined library which can be used from the get-go. The NiftyController class extends a ScreenController class provided by the Nifty GUI libraries, but the class itself is only made functional when the programmer adds specialised code to handle the interface described in a custom XML file. The manager state also instantiates the PlayerControlState, an app state which handles multiple aspects of the game involving the player such as health and points alongside handling keyboard input. Besides app states, the manager state also instantiates several classes with specialised functions. The MapGeneration class creates voxel-based islands used to make the map feel less like an empty ocean, and the BuildingGenerator and EnemyGenerator classes have factory methods that create template entities for stationary and mobile entities.

The majority of game functionality is defined within **Controls**. Controls are part of jMonkeyEngine and are similar to app states, except a control is instantiated and attached to an entity in the game. In a way, controls are app states which are responsible for handling a single entity, such as an individual enemy or bullet. Because technically only one instance of an app state should exist at any one time, controls are necessary to implement functionality for entities which there are many of. The EnemyGenerator, for example, creates entities with an EnemyControl attached. The EnemyControl does little more than give the entity health, allowing it to be destroyed when it it hit with enough bullets. An AIChaserControl can be attached to the same spatial, which will allow it to chase after the player (or any other target entity) and crash into them, dealing damage on impact. A single entity can therefore have multiple controls, although most controls are made specifically for a certain type of entity.

The CollegeControl makes entities produced by the BuildingGenerator act like a fortress of cannons, while a DepartmentControl makes them act like a shop for the player to visit. The ShooterControl is attached to an object to create a constant point from which cannonballs are fired, and every time an entity fires the ShooterControl instantiates a BulletControl to handle the bullet and any collisions it gets into. The flexibility of controls makes it possible for both the player and shooting enemies to use different instances of the same ShooterControl and BulletControl class.

Because of the way Controls and AppStates interact, the concrete structure of the program is remarkably loose and easy to extend.

Justification

Our concrete architecture builds on the abstract architecture by adapting it to fit our engine's (jMonkeyEngine) requirements and provided constructs. Within the engine there are two classes most of our code inherits from, which are the Control and the AppState. Controls are the building blocks for the entities in the game, as they can be attached to Spatials, the graphical objects, to make them implement some behaviour. AppStates, on the other hand, add global behaviour to the game, such as an entity manager or a GUI.
This is all based on an approach favouring composition over inheritance. We initially set out with the intent to avoid inheritance and other classic OOP game architecture principles, and it turned out that jMonkeyEngine heavily encouraged the approach we planned to take anyway. While we didn't have a robust EntityBase as we originally planned, the majority of its function was already carried out by jMonkeyEngine's Node and Spatial classes. The PlayerSystem was implemented almost exactly as the PlayerControlState, and the GameManagementState carries out many of the same tasks we wanted GameManager to do. On an abstract level, we stuck close to our original abstract architecture. We did not include certain elements for assessment 2, such as the minigame which would have taken place on the UI layer, and we had classes for generating enemies which we did not include in the previous architecture description. Despite this, the design is flexible and extensible and follows the commitment to composition we originally described. We do not use inheritance except when required by jMonkeyEngine's libraries, which should make it easier for any groups which may have to extend our software to grasp the code.
For user requirement [R1], all system requirements were implemented within the PlayerControlState. This makes sense on an abstract level, as properties of the player are kept in the same logical space as methods acting upon the player, such as the update method which handles movement.
[R2], [R8] and [R16] are fulfilled by multiple elements of the architecture, with the actual damage code being part of the damage-dealing projectiles themselves. This means that there is no distinction between a player's bullet and an enemy's bullet beyond the difference in a single boolean value stored in the BulletControl. This implementation is flexible and robust, and allows for a lot of code reuse. We enable shooting for a spatial by attaching a ShooterControl, which has a shootBullet method that shoots a bullet in the direction specified. In addition, the AIChaserControl can be added to spatials to make them chase the player and deal damage on collision.
[R3], [R7], and [R10] are heavily intertwined within NiftyController, as that app state is responsible both for managing the user interface and testing conditions like whether the player has enough gold for an upgrade. Allowing the same app state control the layout and implement the functionality of the UI helps keep the interface the user sees working synchronously with the underlying logic that processes their inputs.

[R9] is satisfied fully by MapGeneration. It algorithmically generates random island entities which act as an obstacle for the player, helping the game map appear less barren. Because map generation only has to be carried out once at the start of each level, its methods are simply called directly and their output merged into the game world. This helps save on resources by not keeping an instance of the generator in memory and offers a simple interface which the programmer can access from anywhere in the code with no hassle.
[R6] is implemented primarily by the PlayerControlState, as that is the app state actually responsible for tracking the player's points. Allocating points is flexible and up to the programmer, as its methods can be accessed by basically any piece of code through jMonkeyEngine's app state manager. Currently the player is only given points by the BulletControl when a bullet detects that it delivered the final blow to a particular enemy, but it would be trivial to extend the program to give the player points for time spent alive or distance travelled.
[R12] is implemented by the CollegeControl and its capture method, which is called when the college's health reaches zero. However, stages have not yet been added so [R12.b] is not yet fully met.
[R10] is similarly implemented by the DepartmentControl and the NiftyController which work in tandem to provide a shop GUI when the player comes near the departments.