3 (b)

The codebase we inherited contained a number of significant, systematic issues such as code duplication and violation of the principle of locality. While there was an attempt to logically separate the codebase into modules of related classes, in practice, discrete functionality was often implemented across functions scattered among various modules' classes. Furthermore, related classes often contained sections of nearly identical functionality.

We decided to refactor this codebase to resolve these issues so as to allow us to implement our requirements in a more robust, effective fashion. Details of these refactorings are tabulated in detail in the Changes to implementation subsection. Below is a broad, conceptual level explanation of our method and motivation for the changes:

- Leverage of the OOP model: common functionality was pushed down into base classes, such as the abstract Entity
- Entity-component framework: functionality was separated into components which could be composed into Entity classes to enable behaviours
- Principle of locality & single responsibility principle: classes were refactored to contain only functionality relevant to their behaviour. Other necessary, enabling functionalities, such as lifecycle management or asset retrieval, were refactored into manager classes.
- Separation of model and view: data was localised to the relevant classes, UI classes were built to be provided with data to display
- Singleton pattern: where possible, classes and their methods and fields were refactored to be static, allowing easy access from across the class hierarchy and easy mocking for tests

The result of these extensive refactorings empowered us to implement our new requirements with ease and rapidity. The details of the code changes related to implementing requirements are tabulated separately in the Additions to implementation section.

Changes to implementation

Previous	New	Justification	Classes	Issues
Game entities represented as large classes combining most functionality necessary to the class. Lots of code duplication between classes.	Classes changed into an entity hierarchy with common functionality captured within a base Entity class, and further speciation via subclassing, e.g. Ship->Pirate->Boss. Entity-component model implemented to segment out functionality.	Facilitaties implementation of new functionality, e.g. obstacles, as well as easing testing due to the code sharing allowed by base classes and components.	com.ducks. entities.*	9, 13
Texture and animation code included within each object, lots of duplication	Components implementing said functionality	Reduce code duplication, encapsulate logic separately to objects' behaviour	ChestAnimation, components. HealthBar, ShipAnimation, Texture	5
Box2D bodies and fixtures for collision and entity sensors included within each object	Components implementing said functionality	Reduce code duplication, encapsulate logic, hide large amounts of boilerplate which obscured object behaviour code	RigidBody	1
Multiple ListsOf... classes contain functionality for lifecycle management of relevant objects and some of their functionality	EntityManager introduced to handle all lifecycle management for in game objects. Functionality common to types of object generalised as interfaces.	Reduce code duplication, introduce common ways of handling similar objects, less testing due to smaller codebase	EntityManager, IDrawable, IShooter	19
MainGameScreen loaded assets into private fields, passing them as parameters to other classes	AssetManager is a singleton loading and providing all game assets	Easier testing, faster development of code (no need to chain pass parameters down)	AssetManager, MainGameScreen	
Contact listener contained parts of functionality for the contact events, signalled actions by changing fixture user data strings	EntityData stores a stack of contacting fixtures, added by EntityContactListener during contact events. Entities handle contacts as necessary by popping the stack each update.	Encapsulates functionality within the entities classes, helping to reason about behaviour and implement new behaviours.	EntityContactListener, EntityData	
Duplicate code for handling keyboard inputs existed in Player and MainGameScreen classes, former selecting animation frames, latter applying force to b2dbody	InputParser encapsulates all input handling. Opponent movement is simulated in the form of random keyboard inputs, minimising new code. Force is applied inside entity classes based on InputParser output	Movement handled in one place. Easy to reason about NPC movement as it leverages same mechanics as player. Easy to provide directions to animation components.	InputParser	2
Hud class combining the storage of player stats such as gold and health with drawing of relevant UI components	Hud solely responsible for drawing UI elements. StatsManager is a singleton holding the data. Health is an Entity-level field. Player health is displayed via the ui.HealthBar class.	Separation of model and view enabling easier testing and UI changes	Hud, StatsManager, ui.HealthBar	
Subtitle contained a mix of font assets, UI elements, text data, and functionality	Subtitle is a UI element for displaying notices with text and icons, used throughout the game	Wider reuse of functionality, separation of model and view	Subtitle	

Changes to implementation				
Previous	**New**	**Justification**	**Classes**	**Issues**
Unique texture assets for each button	Generic PlainButton class	Can create new buttons in code	PlainButton	
Shooting mechanics handled through combination of loosely related classes	Shooter component encapsulates of all shooting functionality for all Entities.	Easier testing, code reuse	Shooter	19
Minimap displayed the location of enemy colleges, Tutorial prompted the player to explore the map	Player pointed to college location dynamically via Indicators	More effective game mechanics	Removed: Minimap, Tutorial	21
Multiple prompts were given to player at beginning and end of play, serving as tutorial and storyline	Eliminated in favour of a static message showing controls and an endgame stats screen	Mechanics unsuited for kiosk-style play	Removed: InitialStorylineScreen, FinalStorylineScreen	
Content provided utility methods for loading game asset textures	Use of TextureAtlas provides same mechanics natively	Surplus to requirements	Removed: Content	
Game textures loaded within MainGameScreen	Static TextureAtlas instance stores all Entity textures, static Skin provides all UI drawables	Ease of access, reduction of code	AssetManager	5

Additions to implementation

New	Relevant requirement	Classes
QuestManager manages the lifecycle of Quests. Quest encapsulates data such as objectives and rewards, communicating to the player via Indicator and Subtitle. Indicator guides the user to the objective.	FR_QUEST_CREATION, FR_QUEST_COMPLETION	QuestManager, Quest, Indicator
PowerupManager handles the lifecycle of Powerups. Powerup implements them as an Entity	FR_POWERUP_SPAWN, FR_POWERUP_FUNCTION, FR_POWERUP_DISPLAY	PowerupManager, Powerup
ShopManager allows purchasing of randomly spawning powerups. PauseMenu displays the relevant UI elements including ShopButtons. Triplet is a convenience class for storing the data describing shop items	FR_SHOP_FUNCTION	ShopManager, ShopButton, PauseMenu, Triplet
Button that handles the purchase of powerups and display of relevant info	FR_SHOP_FUNCTION	ShopButton
SaveManager is a singleton providing save/load functionality in tandem with new methods added to EntityManager in accord with its lifecycle management responsiblity. Data is stored by classes implementing the ISaveData interface.	FR_SAVE, FR_LOAD	com.ducks.tools.Saving.*
DifficultyControl stores the difficulty level and provides a generic convenience method for selecting the appropriate data	FR_DIFFICULTY	DifficultyControl
Whirlpool implements them as an Entity, creating a player obstacle. Lifecycle managed by EntityManager	FR_WHIRLPOOL_SPAWNING, FR_WHIRLPOOL_MOVEMENT, FR_WHIRLPOOL_EFFECT	Whirlpool