Wroclaw University of Science and Technology
Faculty of Information and Communication Technology

Field of study: **Computer Engineering**

A Python implementation of the classic board game for Advanced Topics of Object-Oriented Programming

Wiktor Konieczny, 272956

Piotr Koronczok, 272955

Adam Krawczyk, 272978

Supervisor

PhD. Jacek Cichosz

WROCŁAW 2025

Contents

1. INTRODUCTION

Ticket to Ride is a popular board game designed by Alan R. Moon and published in 2004. The game focuses on building train routes between cities, set collection, and strategic planning. It has become widely recognized for its accessible rules and engaging gameplay, making it a staple in both family and hobby gaming communities.

1.1. PROJECT MOTIVATION

The main motivation for this project was to combine practical skills in object-oriented programming with the challenge of implementing a full-featured desktop game. We wanted to deepen our knowledge of design patterns, GUI programming, and event-driven architectures while recreating a widely known and appreciated game. This project also allowed us to practice teamwork, agile development, and the use of professional tools like Git and UML.

1.2. PROJECT SCOPE:

This project aims to create a digital implementation of the board game "Ticket to Ride" as a standalone desktop application. The scope is focused on designing and developing a object-oriented program that represents the game mechanics and rules. The implementation will support 2-5 players and will feature turn-based gameplay with an graphical user interface. The project will be developed using Python and Pygame as the primary technologies, following object-oriented programming principles and an event-driven approach to handle game actions and state changes.

1.3. SYSTEM DESCRIPTION AND FUNCTIONALITY

The system will recreate the Ticket to Ride board game experience, allowing players to:

— Set up a game with 2-5 players with appropriate initial resource distribution (train cards, destination tickets, and train cars)
— Draw train cards from either face-up display or the deck
— Draw destination tickets and select which ones to keep
— Claim routes between cities by playing matching train cards
— Track scores for claimed routes and completed destination tickets
— Calculate bonuses for longest continuous path
— Determine when the game ends and calculate final scores

1.4. TECHNOLOGIES AND TOOLS FOR SYSTEM DESIGN AND IMPLEMENTATION

The following technologies and tools will be utilized for the development of the Ticket to Ride application:

1. **Programming Language:** Python 3
2. **GUI Framework:** Pygame
3. **Design Documentation:** UML (Unified Modeling Language)
4. **Development Methodology:** Agile approach
5. **Version Control:** Git

2. REQUIREMENTS ANALYSIS

2.1. FUNCTIONAL REQUIREMENTS

The functional requirements for the Ticket to Ride application are organized by game phase and functionality:

1. **Game Setup:**
   - board is a map with cities and routes that connects them. Each route from city A to city B has it's cost which is number of specific train cards.
   - allow setup for 2-5 players
   - distribute 45 train cars to each player
   - deal 4 random train cards to each player at game start
   - deal 3 random destination tickets to each player, allowing them to return up to 1
   - display 5 face-up train cards on the game board
2. **Player Turn Actions:**
   - allow players to draw 2 train cards (from face-up display or deck)
   - handle locomotive wildcards according to game rules (count as one card when taken face-up)
   - allow players to claim routes by playing matching train cards
   - allow players to draw destination tickets and keep at least 1
   - enforce the "one action per turn" rule
   - replenish face-up train cards after a player draws
3. **Route Management:**
   - validate route claims based on card color requirements
   - handle gray routes that can be claimed with any single color

— enforce route ownership (prevent multiple claims on the same route)
— visually update the game board when routes are claimed
— deduct the appropriate number of train cars when a route is claimed

4. **Scoring System:**
   — award points immediately when routes are claimed
   — track points for completed destination tickets
   — subtract points for incomplete destination tickets
   — calculate and award the longest continuous path bonus
   — determine the game winner based on final score
5. **Game Flow Control:**
   — manage turn order among players
   — trigger the final round when a player has 2 or fewer trains remaining
   — provide all players with one final turn after the end-game trigger
   — calculate and display final scores at game end
6. **User Interface:**
   — display the game board with cities and routes
   — show player information (train cars, cards, score)
   — provide visual feedback for valid and invalid actions
   — display destination tickets and their completion status

2.2. NON-FUNCTIONAL REQUIREMENTS

The non-functional requirements define the quality attributes of the Ticket to Ride application:

1. **Usability:**
   — the user interface shall be intuitive for new players to learn
   — display the current game state clearly at all times
2. **Portability:**
   — the application shall run on Windows, macOS, and Linux operating systems

2.3. USE CASE DIAGRAM

The use case diagram provided illustrates the interactions between the three main actors (Player, Game Controller, and GUI) and the system. The diagram shows the key functionality of the Ticket to Ride game.

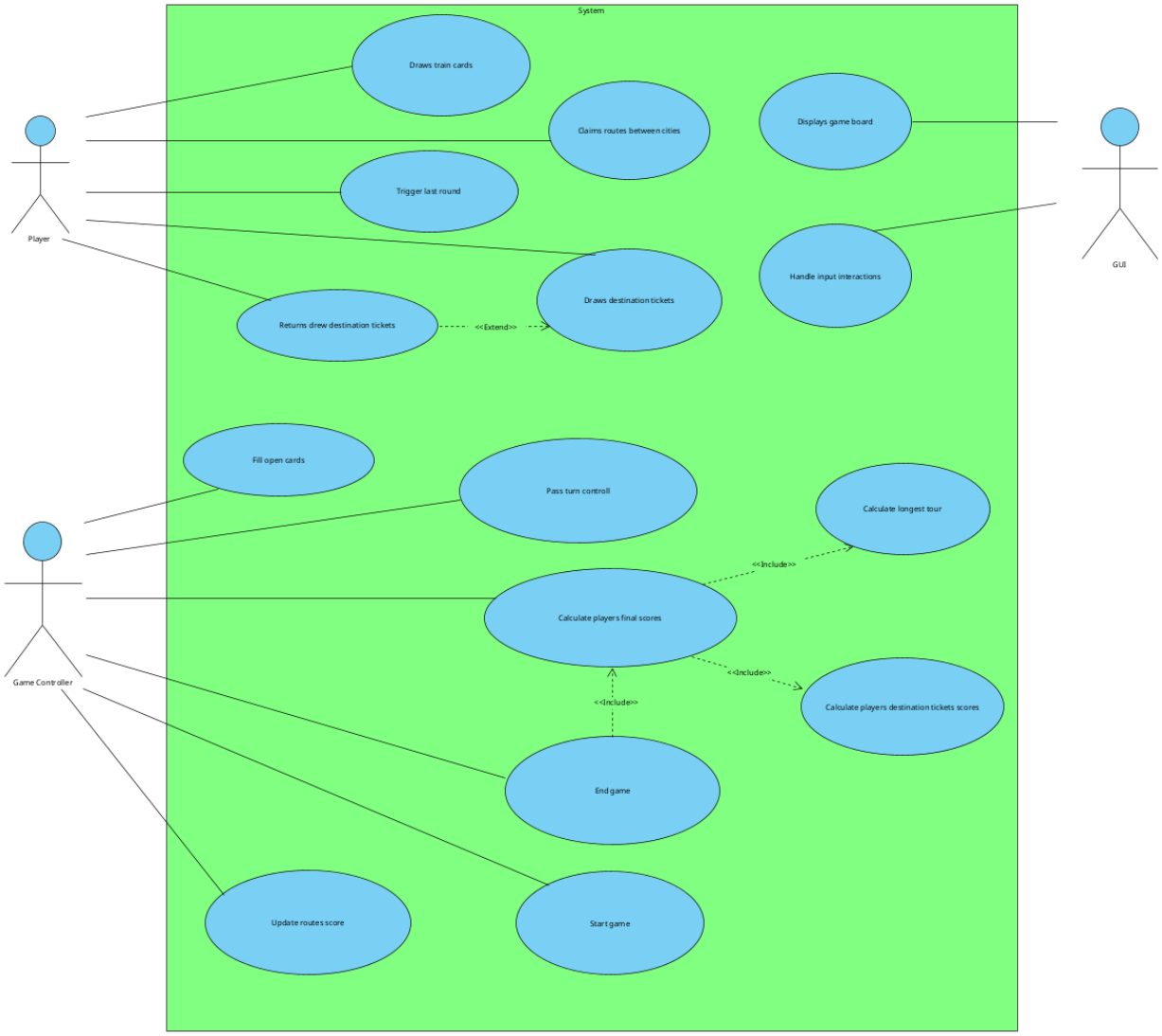

Figure 1. Use case diagram

2.4. USE CASE DESCRIPTIONS

Use Case	Actor	Description
Claim Route	Player	Player selects a route on the map and claims it by playing matching train cards. The system validates the action, updates the map, and adjusts player resources accordingly.
Draw Train Cards	Player	Player chooses to draw train cards from the open display or deck. System deals cards according to rules and updates the display.
View Scores	Player	Player requests to view current scores. The system displays a score summary for all players.
Draw Destination Tickets	Player	Player draws destination tickets and chooses which to keep. System updates player's tickets and deck.
View Map	Player	Player can view the game map with cities, routes, and claimed connections.
Choose Parallel Route	Player	When parallel routes exist, player selects which route to claim from available options.
View Player Hand	Player	Player can view their current train cards and destination tickets.
View Game State	Player	Player can see current game state including open cards, decks, and other players' claimed routes.

Table 1. Use Case Descriptions

3. IMPLEMENTATION CONSIDERATIONS

The application was implemented in Python 3 using Pygame for the graphical user interface. The design follows object-oriented principles with a clear class structure, ensuring maintainability and extensibility. UML diagrams (class and use-case diagrams) were prepared to document the system architecture. Source code management utilized Git version control.

Implementation is available on GitHub [6].

3.1. OBJECT-ORIENTED ASPECTS

3.1.1. Fundamental OOP Principles

— **Encapsulation:** The design uses data encapsulation within classes, hiding implementation details and exposing only necessary interfaces.
— **Inheritance:** Utilized in the card class hierarchy (`Card` → `TrainCard`, `DestinationTicketCard`), allowing for shared behavior and code reuse.
— **Polymorphism:** Implemented in the scoring system and the handling of different card types, enabling flexible and extensible code.

— **Abstraction:** Applied to model key game concepts, such as cities, connections, and cards, allowing for high-level management of game entities.

3.1.2. Modularity

— The codebase is split into logical modules and files, with each class having a single, well-defined responsibility.
— A clear separation exists between game logic and user interface, improving readability and maintainability.

3.1.3. Main Classes

— `GameWindow`: Manages the graphical interface, event handling (mouse/keyboard), and rendering game elements.
— `Game`: Controls the overall game state, manages player turns, and coordinates different game components.
— `City` and `CityConnection`: Represent cities and their connections on the board, including route and scoring management.
— `Player`: Manages player state, including cards and points.
— `Card` System: Hierarchy for different card types and deck management.

3.1.4. Class Relationships

— The `Game` class controls all main components of the application.
— `Player` objects own cards and claimed routes.
— `City` instances are connected via `CityConnection` objects in a many-to-many relationship.
— The GUI collaborates with all components to visualize the game state.

3.1.5. Design Patterns

— **Model-View-Controller (MVC):** Separates game logic from the user interface and controls data flow.
— **Singleton:** Used in the main `Game` class to ensure a single instance manages the game state.
— **Factory:** Applied for creation of cards and connections.
— **Observer:** Implements event and state update notifications.
— **State:** Manages different game states (e.g., turn, animation, waiting for move) in an organized way.

3.1.6. SOLID Principles

— **Single Responsibility:** Each class has a single, well-defined responsibility.

— **Open/Closed:** The system is designed for easy extension of functionality without modifying existing code.
— **Interface Segregation:** Interfaces are minimal and consistent, tailored to specific needs.
— **Dependency Inversion:** Abstractions are used to decouple high-level modules from low-level details.

3.1.7. Good Practices

— Clear code structure and consistent naming conventions.
— Proper documentation and modular architecture.
— Effective resource management and error handling.

3.1.8. Extensibility

— Easy addition of new card types and game mechanics.
— Implementation of different gameplay strategies.
— Extension of the user interface and visualization features.

3.2. OBJECT-ORIENTED DESIGN ASPECTS

Based on an analysis of the code, the following object-oriented programming principles and patterns have been applied in the *Ticket to Ride* project:

3.2.1. Encapsulation

— The `GameWindow` class encapsulates all logic related to game window display and user interaction.
— Internal state and behavior are protected through private fields and methods inside the relevant classes.
— The `City` class encapsulates the data and behavior associated with a city and its connections.
— The `CityConnection` class contains the logic for connections between cities, including scoring and claim validation.

3.2.2. Inheritance and Polymorphism

— The scoring system is implemented polymorphically through the method `get_score_for_claiming()` in the `CityConnection` class, allowing different connection types to define custom scoring behavior.

3.2.3. Abstraction

— The `City` class models an abstract concept of a city on the map.

— The `CityConnection` class abstracts the logic of connections between two cities, regardless of graphical representation or specific route type.
— GUI methods are abstracted to high-level actions such as `draw()`, `update()`, hiding implementation details of screen rendering.

3.2.4. Modularity

— The codebase is divided into logically separated modules and files.
— Each class is designed with a single, well-defined responsibility.
— The project structure clearly separates game logic from the graphical user interface, increasing maintainability and testability.

3.2.5. Cohesion and Encapsulation

— Each class has a cohesive responsibility, and its data is protected and only accessible through public methods.
— The GUI logic does not interfere with game logic, and vice versa.

3.2.6. Single Responsibility Principle (SRP)

— The GUI handles only rendering and user interaction.
— Data model classes such as `City` and `CityConnection` do not contain any GUI logic.

3.2.7. Design Patterns

— **Observer Pattern:** Can be used to notify the GUI of changes in the game state.
— **State Pattern:** Different gameplay phases can be implemented as separate state classes (e.g., waiting for move, animation).
— **MVC:** Separation of Model (game logic), View (Pygame), and Controller (reaction to actions).

3.2.8. Interfaces and Extensibility

— The design allows for easy extension, such as adding new route types or game rules, without modifying existing classes.
— Class interfaces are clearly defined and constrained.

3.2.9. Open/Closed Principle

— The system is open to extension but closed to modification.

3.2.10. Composition

— The GUI consists of components (e.g., board, player panel, buttons) that can be independently modified or replaced.

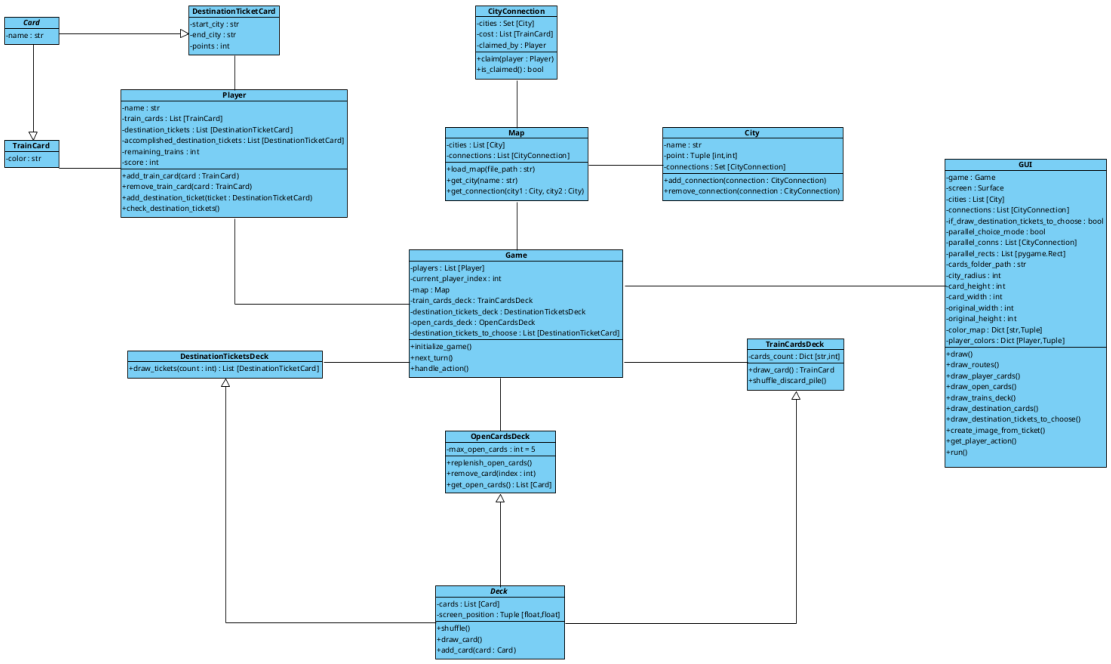

Figure 2. Class diagram

3.2.11. Polymorphism

— Different types of actions and states can be represented as objects implementing a common interface.

4. APPLICATION INSTALLATION, LAUNCHING, AND TESTING

4.1. INSTALLATION AND LAUNCHING

The repository can be cloned from GitHub and executed directly using Python 3. The only external dependency is Pygame, which can be installed using pip. To launch the application:

1. Install Python 3 if not already present.
2. Install Pygame: `pip install pygame`
3. Clone the repository:
   `git clone https://github.com/triplo098/Ticket-to-ride.git`
4. Enter the project directory and run the main application file (e.g. `python main.py`).

4.2. TESTING

Testing was performed manually and through scripted scenarios to verify core gameplay logic (such as route claiming, ticket completion, and scoring).

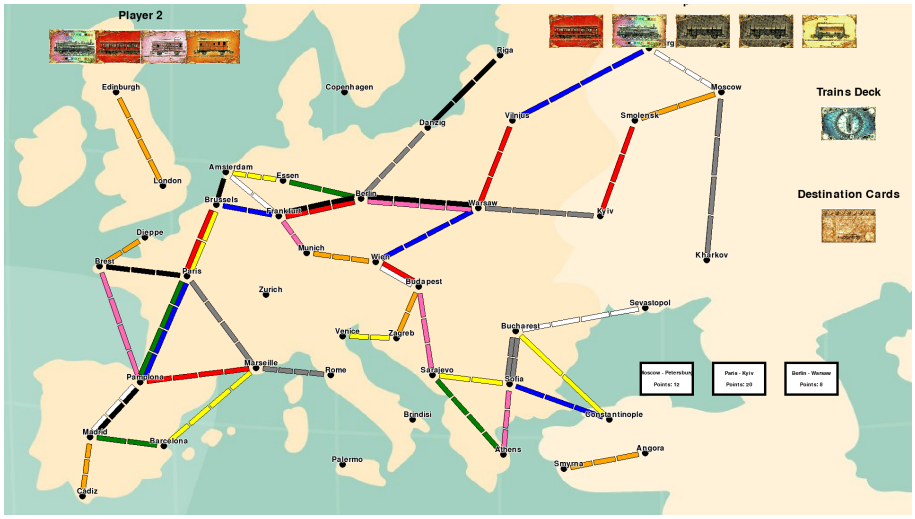

Figure 3. Graphical user interface of the Ticket to Ride game

5. SUMMARY AND CRITICAL DISCUSSION

The project demonstrates practical use of object-oriented programming and design patterns in Python, with an interactive GUI built on Pygame. The architecture separates game logic from the interface, ensuring clarity and extensibility. The main strengths of the solution are modularity and ease of adding new features. Limitations include the lack of network multiplayer and basic AI. Potential future improvements involve:

— online game-play
— AI opponents for single-player game
— additional maps, routes

Bibliography

[1] Website: https://www.python.org/about/

[2] Website: https://www.mysql.com/

[3] Website: https://www.fullstackpython.com/flask.html

[4] Russ, M., Kim, H. *Learning UML 2.0*, O'Reilly Media, 2006.

[5] Website: https://docs.vultr.com/how-to-install-mysql-on-ubuntu-24-04

[6] GitHub Repository: https://github.com/triplo098/Ticket-to-ride.git