Homework #4: Carcassonne

In this assignment, you will design and implement a board game called Carcassonne™. Carcassonne[1] is a tile-based board game for two to five players where the goal is to claim regions of adjacent tiles.

This assignment focuses on the design and implementation of a medium-sized program. The goals are for you to:

- Demonstrate a comprehensive design and development process including object-oriented analysis, object-oriented design, and implementation.
- Demonstrate the use of design goals to influence your design choices, assigning responsibilities carefully, using design patterns where appropriate, discussing trade-offs among alternative designs, and choosing an appropriate solution. The core logic of your solution must be testable and completely independent from your solution's eventual graphical user interface (GUI).
- Communicate design ideas clearly, including design documents that demonstrate fluency with the basic notation of UML class diagrams and interaction diagrams, the correct use of design vocabulary, and an appropriate level of formality in the specification of system behavior.
- Demonstrate basic fluency in GUI implementations, including an understanding of event handling and the observer pattern.

This assignment is much larger and more challenging than the previous assignments in this course. You will design and implement your game over the period of a month, with three main milestones:

- Milestone A: Object-oriented analysis and design (due October 4th)
- Milestone B: Core implementation and testing of game features (due October 18th)
- Milestone C: Full implementation with GUI (due October 25th)

We recommend you start your work for future milestones early whenever possible. For example, ideally, you should start your implementation (Milestone B) before the Milestone A deadline.

Late days for the separate milestones are independent because the milestones have separate deadlines. For example, if you use one late day for Milestone A and two late days for Milestone B, that costs you three late days, not two.

[1] Pronounced car·kuh·SON: "car" as in the vehicle, "kuh" as in the first syllable of "cousin," and "son" rhyming with "gone."

The Game: Carcassonne

In this section we describe the basic rules of Carcassonne. In addition to reading this section, you should read the official rules of Carcassonne, which your solution must implement. You should also play some existing implementation of the game, such as the actual board game or a Flash implementation of the game.[2] The official rules especially clarify many details of the game – such as scoring and the distribution of tiles – and contain examples of gameplay.[3]

In the game of Carcassonne, players take turns placing tiles so that they abut[4] tiles that have already been placed. For the placement of a tile to be legal, the adjacent *segments* of abutting tiles must have the same *feature types*. There are four feature types: road, field, city, and cloister.

Gameplay

Play begins with a designated tile (by convention, a tile with a single straight road) placed face up. Player order is fixed arbitrarily. A player starts her turn by randomly drawing a tile from the stack of (face-down) unused tiles. She then places the tile face-up in a legal position (possibly having rotated it) and optionally places a follower on a segment of the newly placed tile. If a tile cannot be legally placed in any position, it is discarded from the game. The game ends when all tiles are placed or discarded.

Adjacent segments (touching, same-type segments on abutting tiles) connect to each other to form a *feature*. By convention, field features are called farms, and road and city features are just called roads and cities. Cloister segments appear only in the center of a tile and never connect to other cloisters; a cloister feature is instead formed from the tiles in the 3×3 grid centered at the cloister's tile.

Throughout the game, each player controls 7 *followers*.[5] As part of her turn, a player may play a follower on a segment of the tile she places, as long as the segment is not connected (over any distance) to another segment containing a follower. A follower will remain on its segment until the segment's feature is *completed*; the feature is then scored, and its followers are returned to their players. A road is completed if all ends of the road lead to a city, an intersection of roads, or a closed loop. A city is completed if it is completely enclosed by a city wall. A cloister is completed if it is surrounded by 8 tiles. A farm is

[2] Scoring seems broken in this Flash implementation, but basic gameplay seems well-demonstrated. Other online versions have various strengths and weaknesses.

[3] The official rules also include many details specific to board games (such as physical devices for scoring) that are not relevant to a computer implementation.

[4] Two tiles *abut* each other if they are next to each other edge-to-edge. Diagonally touching tiles (with only corners touching) are not considered to be abutting.

[5] Followers are called "meeple" in some versions of the game.

never considered to be completed.

A follower is sometimes called a *knight* if it is placed on a city, a *farmer* if it is placed on a field, a *monk* if it is placed on a cloister, and a *thief* if it is placed on a road. There is no significance to these names.

Scoring

A completed feature is scored based on the feature type as well as the ownership of followers placed on the feature. The official rules summarize the scoring of different feature types. The points for a feature go to the player with the most followers on the feature.[6] (In the case of a tie, the full points go to each player involved in the tie.) Completed features are scored when they are completed, and incomplete features are scored (differently) at the end of the game; see the official rules for details.

At the end of the game, each farm is scored based on the number of completed cities adjoining the farm. The player with the most followers on that farm receives 3 points for each completed city adjoining the farm. Again, in the case of a tie, the full points go to each player involved in a tie.

[6] It is possible for a feature to contain multiple followers by connecting previously unconnected features, even though you can't directly play a follower on a feature containing another follower.

Milestone A: Object-oriented analysis and design (due Oct 4 at 11:59 p.m.)

For this milestone, you must analyze and design your game. You should focus on the game-related functionality of the program, not its user interface. (Think of playing the game by calling a sequence of methods; it is also helpful to think about and possibly sketch out the GUI and how it interacts with the game at this early stage.)

This milestone consists of the following tasks:

1. Create a domain model describing the important concepts of the game.
2. Create a system sequence diagram describing the interactions between a player and the game.
3. Describe the system's behavioral contract for playing a tile (without a follower).
4. Respond to two design scenarios (given below) and provide at least one object-level interaction diagram for each scenario.
5. Create an object model for your game.
6. Justify your design choices, describing the rationale for your design.
7. Get feedback from the course staff.

We describe each of these tasks below.

The domain model

Create a domain model for the game. Your domain model should be represented by a UML class diagram; you may optionally include a glossary. For more information on domain models, see Chapter 9 of Larman's *Applying UML and Patterns*, linked from the course webpage.

The system sequence diagram

Create a system sequence diagram identifying all interactions between the user and the system when the user plays the game. The system sequence diagram should help you determine what interactions the high-level system makes available to its users. For more information on system sequence diagrams, see Chapter 10 of Larman's *Applying UML and Patterns*.

Behavioral contracts

Provide behavioral contracts for the following interaction initiated by the user:

The user attempts to play a tile, without a follower.

The contract should explicitly describe the preconditions and postconditions for the interaction, and your behavioral contract should be consistent with your domain model and interaction diagrams. Constructing behavioral contracts should help you envision important changes of internal state of the game when a player interacts with the game. You may provide explicit examples to clarify your contract. For more information on contracts, see Chapter 11 of Larman's *Applying UML and Patterns*.

Object-level interaction diagrams for two design scenarios

Create object-level interaction diagrams for the two scenarios below. You may split a scenario into multiple interaction diagrams if that improves the clarity of your visualization. Your interaction diagram should be a UML sequence diagram, optionally accompanied by prose. For more information on interaction diagram notation, see Chapter 15 of Larman's *Applying UML and Patterns*.

1. *Describe how the placement of a tile (without a follower) is validated by the game.* You do not need to include any scoring-related details in this scenario.

2. Suppose a valid tile placement (without a follower) completes one or more previously played cloisters (possibly containing followers). *Describe how the game detects newly completed, previously played cloisters, determines whether they contain followers, scores the cloisters as needed, and returns any scored followers to their players.*

These scenarios are just examples of many scenarios worth considering as you design your game. Later in this section, we provide a set of additional informal questions that can help you design your game. We expect formal interaction diagrams only for the two scenarios above. You do not need to respond to the informal scenarios posed elsewhere in this document. You may include interaction diagrams and explanations for additional scenarios if you want.

The object model

Create an object model of your game, documented as a UML class diagram. Your object model should demonstrate that you understand the distinctions between a domain model and an object model. The object model should describe the classes and interfaces of your design, as well as their key associations (with cardinalities), attributes, and methods. The objects and methods in your object model should correspond to the objects in your interaction diagrams above and should reflect the interactions defined in your system sequence diagram. For more information on class diagrams, see Chapter 16 of Larman's *Applying UML and Patterns*.

In Milestone C of this assignment, you will implement a GUI for this game—and your game's methods will be called by the GUI—but your design here should be independent of any specific GUI implementation. Your object model should just model Carcassonne. Do not include any GUI elements in your design.

Justifying your design decisions

Write a short (≤ 2 pages) description of your object model justifying your design choices. To support your justification, refer to design goals, principles, and patterns where appropriate. To help formulate your justification, we recommend that you consider design decisions related to the formal scenarios above, and the informal questions that we provide later in this section.

Showing your work to the course staff

To get full credit for this assignment you *must* show your work to and get feedback from a course staff member *before* the Milestone A deadline. We will schedule design review meetings before the Milestone A deadline and announce the meeting schedule on Piazza well before the deadline. You should finish a reasonably complete draft of all documents before your design review meeting.

In our experience, software design appears to be a simple task, but *good* software design is very difficult and requires many iterations of design and revision. The problem is that, for software design, there are no automated tools such as auto-graders or compilers to provide immediate feedback. Instead, you should get feedback from more experienced software designers—such as the course staff—before you turn in your homework.

We encourage you to discuss your design decisions with your peers. However, you must fully understand your design decisions and generate your own UML diagrams and rationale.

Turning in your work

Turn in the following files for Milestone A in the `design_documents` directory:

- `domain.pdf`: The domain model for your game.
- `system_sequence.pdf`: The system sequence diagram for your game.
- `behavioral_contract.pdf`: The behavioral contract for placing a tile without a follower.
- `interaction_tile_validation.pdf`: Your response to our scenario about validating a placed tile.
- `interaction_cloister_scoring.pdf`: Your response to our scenario about scoring completed regions following a tile placement.
- `object.pdf`: The object model for your game.
- `rationale.pdf`: Your justification of your design decisions.
- `README.md`: Any additional information about your design.

All documents may contain text and figures. Label your final Git commit with a descriptive log message so we know which homework submission to grade and whether to charge you with any late days.

Some design hints

When you are done, Carcassonne will be a medium-sized program with numerous interactions between game components, some of which are complex. Here are some helpful questions you should consider when creating your design. You do not need to formally respond to these questions, but we recommend that you consider them when planning your object model and interaction among game components. It might be helpful to create some interaction diagrams for these questions:

- How can a player interact with the game? What are the possible actions a player can perform?
- How would someone start a game with multiple players? How would players share the same screen?
- How is a player's action of placing a tile with a follower represented differently from placing a tile with no follower?

- How are the current placements of tiles represented? How are tile rotations represented?
- When a new tile is placed, how does the game determine the newly completed features to be scored? How does the game determine if a follower placement conflicts with a previously played follower?
- How does the game support different scoring methods for different features? For example, cloisters are completed when they are surrounded by 8 tiles, but cities are completed when there is no tile placement that could expand the city.
- How are invalid operations handled? To what extent are they processed internally vs. exposed to clients using the game? Are invalid operations exposed as exceptions, boolean or null return values, or other structures that represent the error?
- How will the game be represented in a GUI? What information does a GUI need to access from the game?

Again, you do not need to respond to these questions, but we believe that a good design will clearly address these issues.

A note on notation

To ease communication and avoid ambiguity, we expect all models to use UML notation for class and sequence diagrams. Chapters 9, 10, 15, and 16 of Larman's *Applying UML and Patterns* provide many details and guidance on UML notation. We do not require much formality, but we expect that relationships (such as association, inheritance, and aggregation) are described correctly in your diagrams, and that each relationship includes the cardinalities of the relationship. Attributes and methods should be specified correctly, but we do not require precise descriptions of visibility or types.

It is important that your models demonstrate an understanding of appropriate levels of abstraction. For example, your domain model should not refer to implementation artifacts, and your object model should not include highly specific details such as getter and setter methods, unless they aid the general understanding of your design.

UML contains notation for many advanced concepts, such as loops and conditions in interaction diagrams. You may use UML notation for these advanced concepts, but we do not require you to do so. You may describe such concepts with your own notation or textual comments, as long as you clearly communicate your intent.

To maximize clarity, we recommend that you draw UML diagrams with software tools. We do not require specific tools, and you may share tool-related tips on Piazza. We strongly recommend that you do not mechanically extract models from a software implementation; such mechanically generated models are almost always at an inappropriate level of abstraction. We will accept handwritten models or photographs of models (such as whiteboard sketches) if the models are clearly legible.

Milestone B: Core implementation (due Oct 18 at 11:59 p.m.)

For this milestone, you must complete four tasks related to your game implementation:

1. Improve your design based on our feedback, if necessary.
2. Implement the core features of your game.
3. Test the major components of your core implementation.
4. Document your implementation.

We describe each of these tasks below.

Revising your design based on our feedback

We will provide written feedback on physical copies of your Milestone A submission. (We will also push less detailed feedback to `grades/hw4a.md`.) This feedback might include suggestions for improving your design. We recommend that you revise your design based on our feedback, if possible.

Implementing the core features of the game

Implement the core features of the Carcassonne game. By "core features" we mean that your program should include all the functionality needed to play a full game of Carcassonne. Hence, the core implementation is only playable by calling a sequence of methods and not via a user interface. Your implementation should be able to set up a game and allow you to test the correctness of basic features. For example, your implementation should include a method (or sequence of methods) to place a tile, evaluate the legality of a placement, and update the score.[7]

Note, however, that we are *not* asking you to implement any user interface (graphical or otherwise) in the core implementation. That is, when "setting up a game", you are not adding any user interactions; instead, you should only be calling the methods that might be invoked by a GUI and testing that those methods are correct.

We recommend that you finish implementing the core features of the game before you start to develop the GUI. The core features must be implemented and tested independently of a user interface. You may commit (possibly incomplete) GUI code in the

[7] A unit test could act as the user in a system sequence diagram, initiating several actions across the boundary of the system and then confirming that the state of the system is as expected.

`edu.cmu.cs.cs214.hw4.gui` package, but we will not consider this when grading Milestone B. Make sure that your solution compiles without files in the `edu.cmu.cs.cs214.hw4.gui` package.

Testing the major components of your implementation

Test your implementation with JUnit tests. We do not have any coverage requirements, but you should be confident that the major features of your implementation work. As in homework 3, follow best practices for unit testing. We do recommend that you achieve at least 80% method coverage and thoroughly test the various scoring scenarios. Automate your build and tests with Gradle so that your tests are automatically run on Travis CI.

Documenting your implementation

As usual, your implementation should be well documented using Javadoc comments and regular comments where appropriate.

If your implementation is substantially conceptually different from your Milestone A design it might help to update your design documents, but we won't evaluate your updated documents until after the Milestone C deadline.

Turning in your work

Turn in the following files for Milestone B:

- `src/main/java/edu/cmu/cs/cs214/hw4/core` (and sub-packages): Your implementation of the game's core features.
- `src/test/java/edu/cmu/cs/cs214/hw4/core`: JUnit tests for core implementation.
- `build.gradle`: A build script so `gradle test` compiles and executes all tests.

Label your final Git commit with a descriptive log message so we know which homework submission to grade and whether to charge you with any late days.

Milestone C: Full implementation with GUI (due Oct 25 at 11:59 p.m.)

For this milestone, you will complete your implementation of the game and assess your design process throughout the project.

Implementing a GUI

Your full implementation of the Carcassonne game must include a graphical user interface that allows a player to play the game. The game must be playable, but we impose no requirements on the visual design of the interface. A fully functional game will earn more credit than a cool-looking-but-broken implementation.

We recommend that you write a Swing-based GUI. You may, however, use a different GUI framework (including a web-based GUI or an Android application) if you specifically ask for and obtain our permission in a private message on Piazza.

You may revise your design and core implementation of the game (from Milestone B) as necessary, but the core implementation must remain independent of your GUI implementation. All GUI-related code should be contained within the `edu.cmu.cs.cs214.hw4.gui` package, and your core implementation should never import anything from this package. You do not need to submit tests for the GUI-related components of your implementation.

Extend your Gradle script so that `gradle run` automatically starts your game. If instructions are needed on how to play your game, include them in your `README.md` file.

Design reflection

Update your design documents and rationale to reflect your design of the core game as implemented. Discuss how you changed your design from your initial design in Milestone A to your final implementation in Milestones B and C. Explain why you made these changes. You should explicitly describe improvements you made to your design documents and your design. For Milestone C we will re-evaluate your design changes (if you describe them and we can find them) and partially restore credit lost in Milestone A.

Turning in your work

Turn in all files related to this project in your `homework/4` directory and its subdirectories. These files should include:

- `discussion.pdf`: A discussion of how you changed your design over the milestones.
- `rationale.pdf`: An updated description of your design.
- `*.pdf`: Updated design documents from Milestone A.
- `README.md`: A description of how to play the game as you implemented it, and of any design changes we should re-evaluate.
- `build.gradle`: A Gradle build file supporting the targets `test` and `run`.

Evaluation

Overall, this homework is worth 320 points. Milestones A and B are each worth 120 points, and Milestone C is worth 80 points. If you lose design-related points in Milestone A you can regain some of those points in Milestone C by improving your solution. For example, if your design has serious problems, we will provide feedback and outline what improvements are necessary to regain some of the lost points. The feedback you receive for Milestones A and B will contain additional details and instructions. Exceptional submissions may receive up to 10 points extra credit.

For full credit, we expect:

- A domain model that describes the vocabulary of the problem.
- Design documents that describe the core structures and behaviors of the game, at an appropriate level of abstraction and detail. Specifically, your interaction diagrams should clearly describe object-level interactions within the game. Your object model should correspond to your interaction diagrams and describe the core components of the game, following the design goals discussed in lecture.
- Models that generally follow UML notation, as discussed in the *A note on notation* section above. We will focus more on your overall design than on the specifics of UML notation, but we still expect reasonable notation.
- Plausible and insightful design discussions. It is possible to achieve full credit with a slightly flawed model as long as your design discussions convince us that you carefully considered your decisions. You should refer to the design goals, design principles, and design patterns introduced in lecture.

- JUnit tests that follow good practices and are automated with Gradle and Travis CI.
- A core implementation that is independent of any GUI implementation.
- A functional game implementation playable by multiple players.
- Readable code that follows standard naming conventions and good style.

When in doubt, use your best judgment: make reasonable assumptions and document them.

With your permission, we might feature your game in class if your game, GUI, or general implementation is exceedingly awesome. Have fun!