Settlers of Catan: Developing an Implementation of an Emerging Classic Board Game in Java

Vladimir Costescu
Professor David P. Walker
Princeton University
August 28th, 2014

"This paper represents my own work in accordance with University regulations" - Vladimir Costescu

Contents

Abstract

The goal of this project was to study the workings of the Settlers of Catan® board game, seemingly simple but in reality quite complex, and faithfully implement a computer program in the Java™ programming language through which the game can be played. Furthermore, in the process of creating this implementation, I intended to learn the basics of producing a complex piece of software on my own and from scratch, with planning and design constituting major challenges in this endeavor. In this paper, I will discuss the background of the game and related work, the rules of the game, the steps and reasoning behind my implementation, the challenges I encountered in the course of development, and features of the game I would like to implement in the future.

1. Motivation and Goals for the Project

As with all human undertakings, large and time-consuming projects begin with an underlying motivation, a drive that propels those who engage in such endeavors to start a project and also to continue working on it, despite inevitable hardship during its course, until some goals are met and/or a calendar deadline comes to pass. This senior thesis project is no different. Two years ago, I was first exposed to Settlers of Catan, a three to four person board game, when a number of my fellow eating club members invited me to join them in playing a strange and complicated-looking board game. Having nothing better to do at the time, I accepted their offer and started learning how to play the game. Not surprisingly, it took me some time to begin to grasp the complex rules of the game, and I lost quite a few games, coming in dead last consecutively and consistently. I even became angry at my inability to win, and my friends, tauntingly, told me to "please continue to rage" as they laughed amongst themselves. With some practice, I was able to win sporadically, and I learned to not take the game so seriously and enjoy the social interactions facilitated by Settlers of Catan.

Although those responsible for piquing interest in the game have since graduated, the knowledge that I gained during those spring days remained dormant in my mind for some time, until my memories of the game resurfaced this past fall and I decided to invest in a specimen of the game in order to play with my suitemates. During an ordinary gameplay session, an idea struck me: as a Computer Science major and Settlers of Catan aficionado, why not implement the game as a computer program for my senior thesis? I mulled over the idea briefly and decided that it would make for an interesting research opportunity, and thus I chose it as my topic of investigation for this paper. What follows is a report of the multifaceted learning experience in which I engaged in my

study of Settlers of Catan, including the sum of small successes encountered throughout and also the many moments of frustration that inevitably crept up when things did not go according to plan.

2. Background and Related Work

2.1. Klaus Teuber and Die Siedler von Catan

The story behind the genesis of the game is a curious and unexpected one, beginning with a frustrated dental technician who tinkered with board game ideas in his spare time in his basement workshop and culminating in a wildly successful commercial enterprise that has taken the world by storm and even prompted observers to posit that Settlers of Catan has the potential to displace the classic Parker Brothers game Monopoly in terms of household popularity [10]. While such predictions have not come to pass, the profits generated by Catan and its numerous offshoots and adaptations have certainly enabled Klaus Teuber, the game's creator, to abandon his career as a dental technician and focus on board game development as a full-time job under the banner of his company, Catan GmbH.

Born in 1952 to a father specializing in the dental technician profession, Teuber studied chemistry as an undergratuate and then pursued the same career track as his father, becoming an apprentice in his father's dental lab in 1978 [12]. Just a few years later, in 1981, Teuber created his first game, a predecessor to Barbarossa, and embarked upon what would become a fruitful journey into the world of board games [12]. Throughout his dental career, Teuber continued experimenting with game design, fatefully releasing Die Siedler von Catan in Germany in 1995 and then the translated version, Settlers of Catan, in the United States in 1996 [10]. Experiencing unexpected success with Catan, Teuber was able to afford quitting his job as dental lab managing director in 1999 and subsequently established his company, Catan GmbH, together with his son Guido in 2002 [12].

2.2. Impact of Catan

At the turn of the 21st century, Catan was still a niche product despite enjoying solid success in its home country of Germany after Teuber won the 1995 Spiel des Jahres, but by 2011 it had sold over

18 million copies worldwide [10, 15]. As Catan expanded beyond Germany, consumer demand grew so quickly that the resource needs for manufacturing the game exceeded available supply, and as a result Mayfair Games, Catan's publisher, had to reach out to American companies in order to acquire the necessary materials [10]. Now available for purchase at major retailers such as Amazon.com, Barnes & Noble, Target, and Walmart, Catan is considered mainstream enough to constitute serious competition for more entrenched titles such as Monopoly or Risk [4, 6, 10, 15].

Not only does Catan compete with Monopoly for marketshare, it also competes for mindshare, offering an alternative to the strictly adversarial, capitalistic, and zero-sum world depicted in Monopoly [4, 15]. In Monopoly, there is little opportunity for players to cooperate in order to improve each other's fortunes, nor is there any incentive to do so. Instead, the goal of Monopoly is to drive other players to financial ruin, and the game does not end until one player succeeds in driving the others into bankruptcy, an outcome that can take hours to achieve and likely induces boredom in the players who know that they will lose [4]. Furthermore, although Monopoly's vicious and cutthroat nature unfortunately remains relevant today, more than 75 years after its creation, its teachings also serve to perpetuate the harmful behaviors that have led the world into catastrophic financial crisis, much in the same way as the solitary winner in the game leads all other participants to their economic downfall [4]. In such a toxic climate, Catan attempts to change the status quo through its simple and innovative gameplay.

In contrast to Monopoly and Risk, Settlers of Catan encourages a cooperative rather than confrontational play style, and in fact it is quite difficult, if not impossible, to win the game without engaging with other players in trade for resources [4, 15]. Due to the nature of the game, victory is seldom assured for any one player, and it is not uncommon for players in last place to make a miraculous comeback and win the game. This property of the game keeps players engaged and makes strategic planning more difficult, since each player's actions can provide a material advantage to another player and change the tide of the game [4]. Furthermore, the interconnectedness of players' fortunes serves to discourage any excessively hostile actions by any one player, since the other players might then conspire to frustrate that player's progress in the game and even strip him

or her of any chance of achieving victory. Finally, the conditions that align to create this deterrence mimic the real world quite well, making Catan a useful learning tool and a simplified model for how things actually work in practice, particularly in the domain of international relations [4].

2.3. Other Software Implementations of Catan

Settlers of Catan has been implemented as a software program on a number of platforms, including PC / Mac, iPad / iPhone, Android, Xbox 360, and even Amazon Kindle [13]. In addition to these official versions, there also exist numerous open source incarnations of the game, including SolitaireSettlers, Pioneers, Settlers3d, JSettlers2, and HTML5Settlers [1, 3, 5, 7, 9]. Given the nature of the project at hand, I did not peruse the source code of any of these projects, since I did not want to taint my implementation with ideas taken from sources other than the Settlers of Catan physical board game that I purchased from Amazon.com and the rule booklet included within. It is worth noting that the Settlers of Catan legal team may have, at some point in the past, been involved in the removal of some non-official implementations of the game from the Web, meaning that the list of implementations provided here may not be complete [8].

3. Catan Rules [14]

3.1. Game Components

Before a description of the Settlers of Catan game rules can be attempted, it is necessary to first enumerate and explain the components of the game. The most basic element of the game is the board, which is composed of nineteen hexagonal tiles surrounded by six sea frame pieces. Eighteen of the tiles depict one of five attainable resources in the game, Wool, Grain, Lumber, Brick, and Ore, while the remaining tile depicts a resource-void desert. The sea frame pieces each have one or two sea ports, which are essential to maritime trade, an important game activity, adding up to a total of nine ports, four of which are of a generic type and the rest of which are of a particular resource type, granting a better trade ratio due to their specificity (note that the port locations are fixed but the port types can be shuffled using the extra port pieces included with the game). Augmenting the board at

various stages in the game are roads, settlements, and cities, which can be placed by players during their turn. Enabling the placement of these game pieces are ninety-five resource cards, nineteen of each of the five types, which are collected by players during the normal course of gameplay. The acquisition of resource cards by players is controlled by eighteen dice roll chits, which are distributed across the eighteen resource tiles on the board, and by a pair of standard six-sided dice that are rolled during a normal turn of gameplay. Resource distribution can be curtailed by the placement of the robber piece, which initially starts out on the desert tile but can be moved when certain events occur during gameplay. In addition to being used to purchase roads, settlements, and cities, resource cards may also be used to purchase one of twenty-five development cards, which are shuffled in a deck and placed face down at the beginning of the game. Each development card is imbued with a special property that can affect the course of gameplay, and a small subset can even contribute to a player's victory. Finally, two special cards, Longest Road and Largest Army, can be granted to the player who meets certain special criteria, as alluded to by the titles of the cards.

3.2. Game Setup

The initial setup stage of the game proceeds according to a well-defined workflow. The first step is putting the board pieces together as described in significant detail in the game manual, whether following the prescribed static layout for beginners or using some degree of randomization. In the most randomized setup procedure that is mentioned in the rulebook, the hexagonal tiles, dice roll chits, and port types are shuffled before being used to construct the board. The hexagonal tiles are placed without further restriction after being shuffled, although in the physical version of the game players may opt to avoid excessive resource clustering. The dice roll chits are almost fully random, with the only stipulation being that the numbers six and eight may not be on adjacent hexes, due to their high chances of rolling (14%) relative to the other possible numbers. Finally, the available port locations do not change in this setup, but the port types are distributed randomly across all the possible locations.

The next step is determining the order in which players will take their turns, and there are a few potential mechanisms for doing so. The simplest is for the oldest player to go first and then the

remaining players to follow in descending order by age. Another option is for all players to roll the dice, where the player with the highest roll goes first and the remaining players follow in descending order by dice roll, and any dice roll ties are resolved by one or more rerolls. A third option is to alter either the age or dice roll mechanism such that the second, third, and potentially fourth players are ordered according to their seating location, either clockwise or counterclockwise. No matter which mechanism is chosen, the generated turn order remains the same throughout the game.

In the final step in the game setup process, each player is given the opportunity to place two settlements and two roads on the board. First, players take turns placing one settlement/road pair, such that settlements are placed on hex tile vertices, which are called "intersections", and roads are placed upon the edges connecting these vertices, with the stipulation that roads placed on the board must be connected to a settlement owned by the same player. Once each player has placed one settlement/road pair, the turn order reverses and the last player is the first to place the second pair, with the restrictions that the second settlement must not be an immediate neighbor of the first and that the second road must be attached to the second settlement. After each player has placed two settlements and roads, everyone collects one resource card of the same type as each hex tile adjacent to their second settlement. Finally, the turn order reverts to normal and the game proper can begin.

3.3. Victory Conditions

Victory in the three to four player version of Catan is achieved by the player who is the first to earn ten Victory Points, which are rewarded by the game when certain actions are completed. The simplest way to earn a Victory Point (VP) is to build a settlement, and as such all players start with two "free" points since each starting settlement counts for one point. Another way for players to gain VPs is to upgrade an existing settlement to a city, which grants an additional VP per city built. A third source of VPs is the development card deck, which primarily provides special gameplay advantages to players but also contains five cards that immediately grant one VP each to the player who acquires them. Respectively, these special cards are titled Chapel, Library, Market, Palace, and University. Finally, there exist two special cards that can be granted to players who satisfy their conditions. The first of these is Longest Road, which is valued at two VPs and is given to the

player who has the longest connected road of length at least equal to five. Equivalent in VP value to Longest Road, the Largest Army card is given to the player who has played the largest number of Knight development cards, with a minimum of three played Knights necessary to acquire the card in the first place. Both of these cards can change ownership throughout the game, an event that happens whenever another player becomes the superlative road builder or army commander, respectively.

3.4. Typical Turn Workflow

3.4.1. Rolling the Dice The first step in any player's turn is to roll the two standard six-sided dice included with the game in order to determine resource production. For each hex tile whose dice roll chit displays the rolled number, all players who own a settlement or city at one of the tile's vertices receive one or two resources of the hex's type, respectively. If there are not enough resource cards to be distributed according to this rule, then no player receives any resources of the type or types lacking sufficient cards in the resource decks. A special rule, henceforth referred to as the robber's rule, applies to rolls where the dice total seven in any configuration. The robber's rule dictates that the player who rolled the dice must move the robber object from its current position to a different tile of that player's choosing. If the robber is moved to a tile where other players own settlements or cities, the current player chooses one of these players and "steals" one random resource card from that player's hand. For the duration of the robber's residence on the new tile, no resources are allocated for that tile when the dice are rolled. One other aspect of the robber's rule is that all players who possess more than seven resource cards must choose half of them to discard and return to the resource decks, where fractional numbers of cards are rounded down to the nearest integer. It is important to note that players cannot perform any other actions during their turn before rolling the dice, for example in an attempt to spend resource cards for fear of rolling a seven.

3.4.2. Engaging in Trade After resources are allocated, a player may engage in either maritime trade or trade with other players, where "maritime trade" refers to the ability to swap multiple resource cards of the same type for one card of a different type without interacting with other players. The default maritime trade ratio is four to one, but this can be augmented by the ownership

of settlements or cities on an intersection adjacent to a port along the edges of the Catan island. Specifically, generic ports grant a ratio of three to one for any kind of resource, while resource ports grant a two-to-one ratio only for a particular kind of resource, for example enabling a trade of two Lumber cards for any other resource card. For player-to-player resource trades, any combination of resource cards can be traded provided that both players agree to the trade, with the caveat that only the player whose turn it is may initiate player-to-player trades.

3.4.3. Building Roads, Settlements, Cities, and Development Cards The primary activity undertaken during a player's turn is that of building objects, in accordance with the resource build costs printed on the four identical informational cards that are given to each player at the start of the game. Roads cost one Lumber card and one Brick card and may only be built adjacent to existing roads and/or existing settlements/cities owned by the same player. Furthermore, each hex tile edge permits the construction of only one road at a time, and roads may not be destroyed by any means during the course of the game. Finally, each player is allowed to build a maximum of fifteen roads, a restriction that grants paramount importance to careful road placement, particularly given the fact that roads cannot be reclaimed once placed.

Settlements cost one of each distinct resource, with the exception of Ore, and may only be placed on hex tile vertices that do not already house a building of any type. Aside from the initial two settlements placed in the game setup stage, settlements must be built adjacent to an existing road owned by the same player. Furthermore, settlements must be separated by at least two hex tile edges, whether or not they are owned by the same player. Like roads, settlements cannot be destroyed, but players can replace their own settlements with cities provided that they have sufficient resources. Each player may build up to five settlements, but they can be reclaimed by city upgrades. Finally, players receive one Victory Point for the construction of each settlement, as described in the Victory Conditions section above.

Cities cost two Wheat cards and three Ore cards and may only be placed on hex tile vertices with settlements owned by the same player. Like roads and settlements, cities cannot be destroyed by any means. Each player may build up to four cities on the board, and unlike settlements there is no

way to reclaim cities once they are placed. Cities are considerably more valuable than settlements in terms of resource production capabilities, producing twice as many resources as settlements as described in the Rolling the Dice section above. They are also the main vehicle for the pursuit of victory, granting two VPs each, and in fact it is virtually impossible to win the game without the strategic deployment of cities.

Development cards each cost one Wool, one Wheat, and one Ore, and they can be kept until their owner deems it appropriate to enter them into play. However, it is important to note that only one development card may be played during a player's turn, and furthermore a card cannot be played during the same turn in which it is acquired. The most common type of development card is the Knight, which constitutes a majority of fourteen out of twenty-five total development cards. When played, the Knight card allows a player to move the robber and steal a resource card from a player, as described above in the discussion of the robber's rule, but unlike the event where a seven is rolled, no players are required to discard half of their resource cards if they possess more than seven cards. Furthermore, Knights contribute to players' eligibility to receive the Largest Army card, with three Knight cards being the minimum necessary to acquire the card in the first place. It is important to reiterate that the Largest Army card only transfers ownership once another player's collection of played Knight cards exceeds the size of the next largest player-owned army.

The next most common development card type is that which immediately grants one Victory Point to the player who acquires it. There are five distinct cards in this category, as enumerated above in the Game Components section: Chapel, Library, Market, Palace, and University. Unlike other mechanisms for acquiring VPs, ownership of these cards is not made public to other players until their owner chooses to do so, which generally occurs when a player has enough VPs to win the game. Because of this property of VP cards, they are highly coveted and can be used strategically by clever players. In fact, the existence of VP cards is an important motivation for players to spend the resources necessary to acquire development cards, because they can be used to take other players by surprise and surge ahead to a leadership position.

The final category of development cards that can be built during a player's turn is that of progress

cards. Conferring a specific gameplay advantage to the player who deploys them, there are two of each of three different types of progress cards available in the game. The Road Building card allows players to build two roads according to ordinary road-building procedures, at no additional cost beyond that which was required to acquire the card. This card cannot, however, grant players additional roads beyond the physical per-player limit of fifteen roads. The Monopoly card allows its owner to name one specific resource type and collect all cards of that type in the possession of all other players. Finally, the Year of Plenty card allows its owner to collect any two resource cards from the remaining pool of cards in the resource decks.

4. Implementation Details [2]

In the following sections, I will take the reader on a journey through the workflow of my Catan implementation, from the dynamic generation of the game board to the functioning of the game as it is played. The goal of the discussion is to show the reader how the program actually works rather than to dissect its components in a sterile clean-room. As such, the progression of the discussion will follow the rough outline of the program's operation, introducing concepts and data structures when the necessity arises and explaining important design decisions along the way. Despite this, many references to variables and methods in the code are made throughout the paper, so it is highly recommended for the reader to have a copy of the code close at hand. Additional discussion of design challenges will take place in the Challenges Encountered section of this paper.

4.1. Insertion Point

As for all computer programs, execution begins with the `main` method, which in the case of the Catan implementation currently being discussed is in the `Game` class. The program is executed by typing `java Game NUM_PLAYERS BOARD_RADIUS BOARD_DIMENSIONS` at the command line, where `NUM_PLAYERS` is the number of players desired, `BOARD_RADIUS` indirectly describes the logical size of the board desired (note: a more detailed explanation of board radius can be found in the next section), and `BOARD_DIMENSIONS` lets the `BoardDraw` class know how large the rendered board should be in pixels (the graphical output is square, so only one parameter

is needed). If no values are entered or the values entered cannot be parsed as integers, the program lets the user know that the default values (`NUM_PLAYERS = 3`, `BOARD_RADIUS = 3`, and `BOARD_DIMENSIONS = 700`) will be used. Whether the user's input is taken into consideration or the default values are used, the program proceeds to create a new `Game` object, which sets into motion the constructor of that class. The constructor calls the `startGame` method, whose first line calls the `init` method of `Rules`, the main subject of the next section.

4.2. The `Rules` Class

The purpose of the `Rules` class is to encapsulate as many of the rules of Settlers of Catan as possible. It contains a number of constants that describe immutable (but customizable) features of the game, such as the fixed set of 19 hex tiles that are used to create a board of standard size, the 18 dice roll chits that are to be placed on each tile except for the desert, and the cost of building roads, settlements, cities, and development cards. All of the remaining data structures in this class, including the logical structure of the game board, are generated dynamically from a single seed, the `radius` value entered by the user as described above. As its name suggests, this user-provided parameter measures the distance in hexes from the center of the board to the edge of the ocean surrounding it. A more useful definition of the radius, however, is the number of hex "rings" that make up the board. The center hex comprises the first ring, and the remaining rings grow outward in a concentric fashion. The default value for the radius is 3, as mentioned above, which produces an ordinary Catan game board. Since the radius can take values higher than 3, it is possible to play Catan with larger boards of arbitrary size. The manner in which the board is generated, a task split between the `Rules` and `Board` classes, is the subject of the next section.

4.3. Dynamic Board Generation

4.3.1. Numbering Conventions When playing Settlers of Catan using a physical board, it is easy to refer to a desired hex or intersection simply by pointing to it with a finger. However, in order to interact with a computer-based representation of a Catan board, a unique identifier for each hex and intersection is required. The simplest and most obvious option is to use integer IDs for this task, but

the immediate and inevitable issue that arises is the manner in which these IDs are assigned. It turns out that, for both hexes and intersections, the ring paradigm mentioned above makes it possible to perform this assignment in a logical and scalable way. For hexes, the numbering begins at 0, with the solitary hex in ring 0, and proceeds outward to the right and then in a counterclockwise direction. Thus, the hex immediately to the right of hex 0 is assigned ID 1, and then the hex up and to the left from hex 1 is assigned ID 2. The three rings of a standard board that result from this numbering scheme contain the following ID ranges, in ascending order: 0, 1-6, and 7-18. For intersections, a similar assignment mechanism can be followed, with each ring beginning at the bottommost intersection of the rightmost hex in the corresponding hex ring. The three rings that result contain the following ID ranges, in ascending order: 0-5, 6-23, and 24-53. This numbering paradigm lends itself to relatively straightforward implementation in code, as explained in the following section.

4.3.2. Rings of Catan The first step in the board generation process is the initialization of two data structures, `hRings` and `iRings`, which store the IDs for hexes and intersections, respectively. Each is configured as a two-dimensional `ArrayList`, where rows index the rings and columns index the items within each ring. For hexes, the size of each ring `i` (except for ring 0) is determined by `i * HexShape.NUM_SIDES`, where `HexShape.NUM_SIDES = 6`. For intersections, the size of each ring `i` is determined by `((2 * i) + 1) * HexShape.NUM_SIDES`. Each entry in `hRings` and `iRings` stores the actual ID of a hex or intersection, so the net result of using these two-dimensional arrays is to allow IDs to be accessed using the row and column indices. This is immensely useful because, as the next two sections illustrate, there exist exploitable relationships between the column indices of adjacent rings.

4.3.3. Hex Graph The next step is to generate the hex graph, which is called `hGraph` and is stored as an adjacency list in a **`boolean`** two-dimensional array, where **`true`** indicates that two hexes (indexed by their integer IDs, as described above) are adjacent to each other. Having this data structure available and initialized is of crucial importance when attempting to randomize the dice roll chits, because the rules of Catan, enumerated in great detail above, stipulate that dice roll chits 6 and 8 cannot be adjacent to each other. In fact, in the absence of a data structure that stores

information about which hexes are adjacent to each other, this rule cannot be enforced at all.

The generation of `hGraph` involves the use of `hRings` and two loops that iterate through the rows and columns of that data structure, such that all hexes are traversed in the same order in which their IDs were initially assigned, as described above. For each hex, the algorithm first looks at the next hex in the same ring, and then at the adjacent hexes in the next ring. Looking ahead within the same ring is a straightforward operation in that `j`, the column index, is simply incremented by 1 at each step, with the caveat that modulus arithmetic must be used so that the last hex in a ring can "wrap around" to the first hex in that ring. However, determining which hexes in the next ring (if it exists, i.e. the current ring is not the last one) are to be examined and marked as adjacent to the current hex is a bit trickier and involves the introduction of a couple of simple concepts.

First, it is necessary to categorize hexes as either corners or not corners. A "corner" is defined in the following way: a hex is a corner if and only if its column index (its index inside the ring it is in) can be evenly divided by its row index (the index of the ring it is in). Visually speaking, the centers of all corner hexes lie on one of three axes that are perpendicular to the six sides of the center hex and pass through its center. Any hex that does not fit this criteria is not a corner. This distinction is important because corners are adjacent to three hexes in the next ring, while non-corners are only adjacent to two hexes in the next ring.

Second, the board needs to be divided into sextants, much in the same way as the two-dimensional Cartesian plane is divided into quadrants. As the name suggests, a "sextant" is one of six equal-sized slices into which a regular hexagonal Catan board (i.e. a board that has symmetry along the three corner axes) can be divided. Due to the nature of the ring indexing paradigm, it is easy to determine which sextant a given hex falls into simply by looking at its row and column indices, `i` and `j`, respectively: `sextant = j / i`. Note that, since `i` and `j` are both integers, integer division is performed and the remainder is ignored (although, as above, the existence of a remainder determines whether or not a hex is a corner). Also, observe that `sextant` can take values in the range 0-5.

Having introduced the corner and sextant concepts, it is now possible to explain the manner in which the hex graph algorithm determines, for a given hex, to which hexes in the next ring that

hex is adjacent. A typical hex located at row index `i` and column index `j` in `hRings` is adjacent to the two hexes in the next ring (row index `i + 1`) whose column indices are `j + sextant` and `j + 1 + sextant`. The role of the `sextant` value here is to account for the fact that each ring contains 6 more hexes than the previous ring, distributed evenly across the 6 sextants. Thus, sextant 1 has to account for the extra hex contributed by sextant 0, sextant 2 has to account for the extra hexes contributed by sextants 0 and 1, and so on. If the current hex is not a corner, the story ends here. However, if it is a corner, then there is an extra hex in the next ring that must be considered, which is indexed by `j - 1 + sextant`. An extra wrinkle is that it is necessary to account for wrap-around by using modulus arithmetic, since `j - 1 + sextant` can become negative if `j` and `sextant` are both 0.

4.3.4. Intersection Graph Following the initialization of the hex graph, the next step is to generate the intersection graph, which is important because the edges between intersections constitute the only valid locations for the placement of roads during a game of Catan. Similar to the hex graph, the intersection graph is titled `iGraph` and is stored as an adjacency list in a **`boolean`** two-dimensional array, where **`true`** indicates that two intersections (indexed by their integer IDs, as described above) are adjacent to each other. Note that, although hexes and intersections are inextricably linked on a physical board by virtue of the board being flat, in the implementation presently being discussed hexes and intersections are logically separate entities (that said, the next section describes the initialization of the logical bridges between hexes and intersections).

The generation of `iGraph`, like that of `hGraph`, involves the use of two loops to traverse `iRings` in order of ascending intersection ID. However, a key difference is that the `j` index of the inner loop begins at 1 and ends at 0, using modulus arithmetic to wrap around to the beginning of the current ring when the end is reached. The reason for this will be explained in more detail shortly, but the basic point is that starting at 1 rather than 0 eliminates the need to have a special case for the first ring. For each intersection traversed, the algorithm first looks at the next intersection in the same ring, and then at the solitary adjacent intersection in the next ring, if one exists. Looking ahead within the same ring is a straightforward operation in that `j` is simply incremented by 1 at

each step, but it takes a bit more work to determine when a link to the next ring needs to be created.

A basic observation is that every intersection in a ring contains 3 links to other intersections, 2 of which are in the same ring (pointing forward and backward) and one of which points to either the previous ring or the next ring. Since the algorithm only moves forward, it is only necessary to consider one of the links within the same ring, as described above. However, the third link alternates between pointing forward and backward, a fact that requires the use of a **`boolean`** called `hasNextLink`, which flips back and forth between **`true`** and **`false`**. The other challenge posed by the third link is determining the column index of the intersection in the next ring. Since the algorithm starts at `j = 1`, the very first intersection examined for any ring always has a link to the next ring, specifically to column index 0. Thus, `hasNextLink` is initialized to **`true`** and `ringIndexDiff` to -1. Recall from the Rings of Catan section above that each ring has 12 more intersections than the previous ring, a difference which can be distributed across the 6 corners of each ring such that each corner contributes 2 extra intersections. This can be verified by visually comparing corner and non-corner hexes: for a corner hex, 2 of its 6 vertices are in one intersection ring and the remaining 4 in the other, whereas for a non-corner hex the split is 3 and 3. Now consider the fact that for each hex, corner or non-corner, 2 of the 6 sides connect two intersection rings. Since the algorithm only moves forward in a ring, only one of these sides is relevant, so hexes can be counted by the number of intersections in one ring that connect to the next ring. For any given hex ring `i`, the number of non-corner hexes in a sextant is equal to `i - 1`. Because extra intersections appear only at corner hexes, `ringIndexDiff` only gets incremented by 2 at a corner hex after one of the 2 corner hex intersections in the first ring is traversed. Thus, a counter variable `waitTillNextInc` can be set to `i`, since the corner hex gets counted in addition to the non-corner hexes. For each sextant, this variable is initialized and then decremented at every other intersection, since as established above only every other intersection has a link to the next ring. Using `hasNextLink`, `ringIndexDiff`, `waitTillNextInc`, and the logic described above, it is thus possible to determine when a link to the next ring needs to be created and also to compute the value of the column index of the intersection in the next ring to be linked to, the value of which

is simply `j + ringIndexDiff`.

Having explained how the algorithm functions, it is now possible to return to the minor detail of beginning the algorithm at column index 1 instead of 0 and explain this design choice more fully. The reason for this design decision is that the link between column indices 0 and 15 of rings 0 and 1, respectively, would be difficult to capture without resorting to the use of special cases, either having a special case for the entire first ring or just for that troublesome link. By starting at column index 1, it is possible to initialize `ringIndexDiff = -1` as above and allow the algorithm to increment this variable until it reaches 15 once column index 0 in ring 0 is finally traversed.

4.3.5. Hex to Intersection Mappings As mentioned above, hexes and intersections are logically separate entities in the Catan implementation presently being discussed, but it is now necessary to create logical bridges between them, referred to henceforth as "mappings". These mappings occur in two directions: from hexes to intersections and from intersections to hexes, titled `hIMapping` and `iHMapping`, respectively. Both of these data structures are two-dimensional `ArrayLists` of integers, storing only intersection or hex IDs. `hIMapping` is set up such that the row indices refer to hex IDs and the columns store the 6 intersection IDs associated with each hex (the ordering and indices of the columns is irrelevant). `iHMapping` is set up in a similar manner, such that the row indices refer to intersection IDs and the columns store the 3 hex IDs associated with each intersection (again, the ordering and indices of the columns is irrelevant). `hIMapping` is needed for the scenario where the robber is moved to a new hex and the game has to know which players are eligible to have a resource card stolen by the player who moved the robber, by proxy of the player-owned settlements or cities that border that hex. `iHMapping` is needed to initialize the `Intersection` data structure (discussed in more detail later), so that when the time comes for players to collect resources after the dice are rolled, each `Intersection` owned by a player can know which hexes it borders and thus whether its player owner should be given one or more resource cards, depending on whether a settlement or city was built there.

The algorithm used to initialize these two mappings is reasonably straightforward, but there are several moving parts that require some explanation. Two loops are used to iterate through `hRings`,

such that `j` indexes the current hex ID in `hRings`, as in the Hex Graph section above. However, for each hex, intersections are also iterated in parallel, and in fact the row index `i` serves the dual function of indexing rings for both `hRings` and `iRings`, since the number of rings for each is equal to `radius`. For each hex traversed, both the current intersection ring (at index `i` in `iRings`) and the previous ring (at index `i - 1`) are visited, with `curIIndex` and `prevIIndex` being used to keep track of intersection column indices for the current and previous rings, respectively. The concept of corner hexes is again used here to determine what the split of each hex's intersections is with regard to the current and previous intersection rings. Recall from the previous section that the split for corner hexes is 4 and 2 for the current and previous intersection rings and 3 and 3 for non-corner hexes. This information is used, for each hex, to traverse the intersections in the current intersection row, add an appropriate entry to each mapping (an intersection entry for `hIMapping` and a hex entry for `iHMapping`), and then do the same for the previous intersection row. Once all hexes are traversed, the algorithm finishes and both mappings are fully initialized.

4.3.6. Resources The official Catan rules specify exactly how many resource tiles of each type (Wool, Grain, Lumber, Brick, Ore, and also Desert) are to be represented when putting together the game board, and for the physical game these are fixed by the printed hex tiles provided in the box. However, since the implementation presently being discussed allows for the use of board sizes larger than that produced using the standard radius parameter of 3, it is necessary to implement a method that decides how many additional tiles of each type are to be added to the board once the fixed 19 are exhausted. Because this decision has to be made outside of the official rules, which do not provide any information about tile type distribution beyond the 19 given tiles, I used my discretion to choose a scalable and fair method for generating additional tiles.

Before the simple decision algorithm is explained, however, a brief discussion of the basic `Resource` data type is required. The `Resource` class is essentially a wrapper for integer constants used to denote each different resource type, from `DESERT = -1` to `ORE = 4`. The ordering of these integer values reflects the relative worth of each resource type, with `DESERT` being the least valuable since no resource cards of that type can be collected (and even if such cards

existed, the standard build costs would still not include Desert). Other elements of `Resource` that are worth noting are the `Color` and `String` constants, for use by `BoardDraw` and for naming the resources, respectively. There also exists a **`static`** method called `getResourceType` (which overloads the `getResourceType` method that takes no arguments) for finding the integer value that corresponds to the given `String` parameter `sResource` containing the name of a resource. This method can be seen as the counterpart (in reverse) to the overridden `toString` method that gives the `String` name of a `Resource` instance. It is worth noting that `Resource` is used both for initializing the `Hex` data type (discussed in the Hexes section) and also as a resource card data type for initializing the resource deck. This dual use of `Hex` is the reason why `DESERT` is set to -1 rather than 0. Because the resource deck does not contain any Desert cards (as mentioned above) and is crucial to the functioning of the game (building anything costs resources), the latter use case is prioritized, so that `NUM_TYPES` can correspond to 1 greater than the last resource value (`ORE`) in order to allow these integer constants to be appropriated as array indices for the `ResourceBundle` data type.

The hex tile generation algorithm begins by adding the standard 19 hex tiles, which are stored in `DEFAULT_HEXES` as an array of frequencies, to `hexTiles`. A slight detail to take note of in the loop that implements this operation is that the indices in `DEFAULT_HEXES` are shifted by one, since `DESERT = -1` and thus cannot be used unmodified as an array index. If the user-provided radius parameter is equal to the default value of 3, then the story ends here. For larger boards, the algorithm continues by iterating through `hRings` and adding equal amounts, determined by `curRingSize / Resource.NUM_TYPES`, of each non-Desert resource type to `hexTiles`. If the current ring happens to be evenly divisible by 5 (the value of `Resource.NUM_TYPES`), then the algorithm continues on to the next ring. Otherwise, a **`switch`** block decides what resource types to assign to the remaining hexes that are unaccounted for. I chose to implement this decision such that extra Desert tiles are added for all nonzero remainders except 3, in which case all non-Desert resources except Brick and Ore are added. The reasoning behind this is twofold: first, a huge Catan island would look odd with only one Desert tile on it, and second, the relative abundance of

Wool, Grain, and Lumber on the standard Catan board is higher than that of Brick and Ore, so it makes sense to maintain this property for larger boards when possible.

4.3.7. Dice Roll Chits Since every hex tile in Catan must have a dice roll chit associated with it (except for Desert tiles, but those can be dealt with by simply assigning them chits of value 7), it is necessary to generate a number of dice roll chits equal to the number of hex tiles in `hRings`. For radius equal to 3, the default dice roll chit frequency distribution is used, which is stored in `DEFAULT_DICE_ROLLS`. Because dice roll values are in the range 2-12 but the indices of `DEFAULT_DICE_ROLLS` are in the range 0-10, an offset of 2 must be used to transform loop indices `i` into actual rolls, as in the following code fragment: `diceRolls.add(i + 2)`.

For radius larger than 3, the default chits are added to `diceRolls` as in the standard case, but then an additional step of generating new chits for each hex ring is performed. It is essential to ensure that the chits generated here match up with the hex tiles already initialized, such that the number of dice rolls equal to 7 is equal to the number of Desert tiles and the number of dice rolls not equal to 7 is equal to the number of non-Desert tiles. A helpful observation here is that the number of non-Desert resources (5) happens to divide the number of non-Desert dice roll chits (10) evenly, so the assignment task is reduced to matching up the cases for nonzero values of `curRingSize % (DEFAULT_DICE_ROLLS.length - 1)` in `initDiceRolls`, the method presently being discussed, to the cases for nonzero values of `curRingSize % Resource.NUM_TYPES` in `initHexTiles` (an in-depth discussion of which can be found in the previous section).

While `curRingSize % (DEFAULT_DICE_ROLLS.length - 1)` (range 0-9) has twice as many cases as `curRingSize % Resource.NUM_TYPES` (range 0-4), in fact the odd cases (1, 3, 5, and 7) can be discarded since all hex rings except the first one (already dealt with in the default chit initialization step) contain an even number of hexes. The remaining cases match up in the following way: values less than 5 correspond identically (0 to 0, 2 to 2, and 4 to 4), and values greater than 5 correspond to the result of taking their modulo for 5 (6 to 1 and 8 to 3). Recall that, in `initHexTiles`, all cases except case 3 add Desert tiles, which means that cases 2, 4, and

6 in `initDiceRolls` must add the appropriate number of chits equal to 7. Cases 2 and 4 are straightforward, adding 2 and 4 chits equal to 7, respectively, but case 6 is a bit trickier because 5 chits not equal to 7 must be added in addition to the one chit equal to 7. The 5 most common dice rolls are chosen, with ties being broken in favor of larger values (an arbitrary choice to satisfy an unavoidable decision). In a similar way, case 8 adds one of each dice roll, except for 2, 7, and 12.

4.3.8. Ports The components of board generation discussed thus far have all been located in `Rules`, since that class deals with as many unchanging rules for a given board radius as possible. The last board generation component that fits this criteria is the task of determining how many maritime ports are to be added and what their types should be. For a standard board of radius 3, these details are specified by the rules of Catan, but again it is necessary to attempt to extrapolate from the rules in a reasonable way so that larger boards can be generated appropriately. Before the algorithm that does this is discussed, however, the basic `Port` data type must first be briefly explained.

There are three basic port types represented in `Port`, which are stored as integer constants: `INLAND`, `GENERIC`, and `SPECIFIC`. `INLAND` is not technically a maritime port type, but it is used as a dummy port for intersections that are not associated with a real maritime port, thereby enforcing the game rule which states that, no matter where settlements and cities are built, players are allowed to trade 4 resources of one type for another resource of a different type. `GENERIC`, as its name suggests, corresponds to the generic port type discussed in the Engaging in Trade section above, which enables players to use a 3:1 trade ratio for any resource type. `SPECIFIC` corresponds to the resource port type discussed in the same section, enabling players to use a 2:1 trade ratio when trading a specific resource type for any other type.

The methods in `Port` implement some basic functionality for interacting with ports, including retrieval methods for the fields of the class and an overridden `toString` method. The most interesting method in the class is titled `compareRatio` and takes a (different) `Port` instance and a `Resource` instance as parameters. The task of `compareRatio` is to determine whether **`this`** or `p` (the other port) has a better (lower) trade ratio for the given resource parameter `r`. When a port is of the `SPECIFIC` type but for the wrong resource (i.e. a resource type different from that of `r`),

it is treated as functionally equivalent to an `INLAND` port. The output of `compareRatio` is an integer that follows the conventions of the `compare` method required for classes that implement the Java `Comparable` interface, which is to say that the return value is negative if `p` has the better ratio, 0 if the ratios are equal, and positive if **`this`** has the better ratio.

Now that the `Port` data type has been introduced, the discussion of the algorithm for generating the proper number of ports of each type can begin. The first step is to determine how many total maritime ports are needed for a given board radius, such that not too few and not too many ports are distributed around the edge of the island of Catan. Recall from the Game Components section above that a standard Catan board has 9 ports, 4 of which are generic and 5 of which are specific. A constant `PORT_RATIO = 0.6` attempts to extrapolate this rule to larger board sizes while still satisfying the initial rule constraint. Consider how the constant's value of 0.6 applies to a standard board: 60 percent of the 30 intersections in the last ring in `iRings` should be adjacent to a maritime port, a calculation that gives the result of 18, twice the standard number of ports specified in the official game rules. Since each maritime port in Catan is attached to two intersections, it is clear that the ratio of 0.6 satisfies the rules of Catan. For larger boards, multiplying by the port ratio can produce a result that is odd, which is undesirable but easily fixed by subtracting any remainder given by `numPortsI % 2`, where `numPortsI` is the number of intersections that need to be attached to a maritime port.

In order to follow the standard split of generic and specific ports as closely as possible, the algorithm tries to assign the `SPECIFIC` type to about half of the needed ports and the `GENERIC` type to the ports that remain. However, in order to maintain fairness and balance, the number of specific ports needs to be divisible by the number of resource types (5) so that no resource is given preference over the others in terms of trading advantage. To achieve this, it is necessary to first compute `(numPortsL / 2) % Resource.NUM_TYPES` and subtract the result from `Resource.NUM_TYPES`, where `numPortsL` is the number of logical ports (equal to one half of `numPortsI`). The purpose of this calculation is to determine how many ports need to be added to the computed `numPortsL / 2` value so that the number of logical ports is divisible by 5. Once

`numSpecificL` and `numGenericL` are computed, all that remains is to initialize the appropriate number of specific and generic ports using loops.

4.3.9. The `Board` Data Type The remaining steps in the dynamic board generation process involve some degree of randomness to ensure that every new game of Catan is different and exciting, and as such these steps cannot be implemented in the `Rules` class. The `Board` class exists for the purpose of picking up where `Rules` leaves off, taking advantage of the retrieval methods that provide access to the **`static`** fields of `Rules`. It stores the remaining information about the game board, including the data structures containing the game hexes, intersections, and roads. The manner in which these data structures are initialized is the subject of the next few sections.

4.3.10. Hexes The game's hex tiles are stored in the `hexes` field (in `Board`), which is an array of type `Hex`, a data type acting as a wrapper for important information about hexes. `Hex` contains only 4 fields (and includes retrieval methods for accessing them): `id`, `resource`, `diceRoll`, and `hasRobber`, where `id` is the unique integer hex ID (discussed in the Numbering Conventions section) and the remaining 3 are self-explanatory. `Hex` also includes methods for updating `hasRobber`, which are intuitively titled `placeRobber` and `removeRobber`. Finally, it implements an overridden `toString` method, which is not currently called in the course of a game but was useful for debugging purposes.

The initialization of `hexes` begins with the retrieval of the hex tiles generated by the `Rules` class, which are then copied into a new `ArrayList` (in order to avoid modification of the `hexTiles` field in `Rules`) titled `shuffledLand` and randomly reordered with `Collections.shuffle`, a standard Java method. This array is then searched for Desert tiles, and the indices of these tiles are added to `desertIndices`. The search is done with the help of `ArrayList`'s built-in methods, specifically `indexOf`, `lastIndexOf`, and `subList`, and also a temporary data structure titled `partialLand`. The algorithm for searching works by finding the last index where a `DESERT` value occurs, storing the index in `desertIndices`, and then taking a sub-array (starting from 0) which excludes that index and searching that sub-array (stored in `partialLand`). The process continues until all desert indices have been saved (i.e. until `indexOf` returns -1). This step of

finding desert indices is necessary because, as will be seen shortly, the desert indices must be known so that the dice roll values equal to 7 can be placed properly.

The next step is to generate the shuffled dice rolls, a task performed by the `getDiceRolls` method, which takes `desertIndices` as an argument. Since the dice rolls have already been initialized, as discussed in the Dice Roll Chits section above, all that needs to be done here is to shuffle them. However, there are a couple of issues: for the standard board size, the Catan rules stipulate that 6s and 8s cannot be adjacent to each other, and in general the indices of the dice rolls equal to 7 have to match up with the indices of the desert tiles. The resolution of the latter issue is delegated to the `randomizeDiceRolls` method, a helper to `getDiceRolls`, which uses `Collections.shuffle` to randomly (and non-destructively) reorder the dice roll array acquired from `Rules.getDiceRolls`, removes the dice rolls equal to 7, and then re-adds the 7s at the indices specified by `desertIndices`.

The former issue is dealt with in `getDiceRolls`, which verifies whether or not the shuffled dice rolls returned by `randomizeDiceRolls` respect the Catan rules. The test is performed using the hex graph array retrieved with `Rules.getHGraph`, where all possible adjacencies are checked to make sure 6s and 8s are not next to each other. The outer loop continues to re-generate the shuffled dice roll array with each iteration until there is no conflict, at which point `getDiceRolls` is finished and returns the shuffled dice roll array. Once the shuffled hex tiles and shuffled dice rolls have been generated, the final step is to iterate through each hex (the total number of hexes is retrieved using `Rules.getNumHexes`) and initialize a new `Hex` data structure with the proper ID, `Resource`, and dice roll. Finally, the robber is placed at a random desert index using `Math.random` and the `placeRobber` method of the `Hex` instance located at that index.

4.3.11. Intersections Following the initialization of `hexes`, the next step in the board generation process is to initialize `intersections`, an array of type `Intersection`. As with the `Hex` type above, `Intersection` is used to wrap around some basic pieces of information about each intersection and expose some methods for manipulating intersections. There are 5 fields in this data type: `id`, `player`, `building`, `port`, and `hexes`, where `id` is the integer ID

assigned as explained in the Numbering Conventions section above, `player` is a reference to the `Player` instance that "owns" the intersection (the `Player` type is discussed in more detail later), `building` is a reference to the `Building` that resides at the intersection, `port` is a reference to the `Port` adjacent to the intersection (either a true maritime port or a dummy port), and `hexes` contains a list of `Hex`es that border the intersection.

Aside from the standard field retrieval methods, `Intersection` contains an overridden `toString` method (used for debugging purposes) and some convenience methods for manipulating the underlying `Building` type, which are used by the `Player` class when building settlements and cities. The `Building` type wraps around the integer building types `OPEN` (i.e. nothing built), `SETTLEMENT`, and `CITY`, implements a simple `toString` method, and provides two methods, titled `canUpgrade` and `upgrade`, for manipulating the building (i.e. building a settlement or city by incrementing `buildingType`). The convenience methods that `Intersection` exposes for the purpose of manipulating its `building` field are `canBuild`, `canBuildSettlement`, `canBuildCity`, and `upgrade`, all of which depend on `Building`'s `canUpgrade` and `upgrade` methods to actually update the building. Note that `Building` and `Intersection` do not enforce the rules concerning building limits and costs, a task that is performed higher up in the class hierarchy (discussed in a later section).

Having explored the `Intersection` and `Building` data types, the discussion now turns to the initialization of `intersections`. An important prerequisite for this initialization is the randomization of maritime port locations using the ports that have already been generated in the `Rules` class (see the Ports section above for more details), a task performed by the `randomizePorts` helper method. This method begins by retrieving the list of ports using `Rules.getPorts`, randomly shuffling this list, and reinitializing some temporary variables for the sake of code readability (since it would be overkill to create retrieval methods for these in `Rules`). Then, a single dummy `Port` instance of type `INLAND` is created (titled `inland`) and `numIntersections - numI` references to this instance are added to `portsI` (the list of port references for all intersections) so that all intersections not in the last ring of `iRings` are accounted for (`numI` is simply the size of

the last ring). The last intersection ring is where the actual maritime ports are located in a game of Catan, so this is the point at which things get more interesting.

Recall the small note in the Game Components section that, for the physical version of Catan, port locations around the edge of the island are fixed but the port types can be shuffled around. In the Catan implementation presently being discussed, however, both port types and port locations are randomized. Having already explained how the port types are chosen and randomized, the conversation now turns to how the ports are distributed around the island. The first consideration is minimum spacing: since the official Catan board refrains from placing ports immediately next to each other, it seems reasonable to do the same in the computer version of the game. Thus, the first "pass" through the ports adds a reference to the dummy `inland` port instance between every two port instances (since each logical port is attached to two intersections) that are added to `portsI`, such that no sequences of 4 consecutive intersections with maritime port references exist along the last intersection ring.

In order to (slightly) randomize the port locations, the algorithm then loops through `portsI` and, at each step, randomly decides whether or not to add a reference to `inland`. This continues until the size of `portsI` becomes equivalent to the size of the last intersection ring, so multiple passes through `portsI` may occur. Since the size of `portsI` increases every time another reference to `inland` is added, it is necessary to keep track of its size separately, using `portsISize` and incrementing it appropriately. After `portsI` is fully initialized, the last step in the `randomizePorts` method is to initialize `Board`'s `portLocations` field, which will be used by `BoardDraw` to render the ports in the correct locations on the board. This is done simply by iterating through `portsI` and adding the indices of all non-`INLAND` ports to `portLocations`. Once the ports are returned by `randomizePorts`, it is finally possible to loop through and initialize the intersections by iterating over index `i` from 0 to `Rules.getNumIntersections`, where each intersection is created using `i` (the integer ID), the proper port, and an array containing the hexes bordering the intersection (retrieved using `Rules.getIHMapping`).

4.3.12. Roads In the Catan implementation presently being discussed, roads are stored in a two-dimensional adjacency list array titled `roads`, similar to `iGraph` (discussed in the Intersection Graph section above). However, instead of having **`true`** values denoting the adjacency of intersections and **`false`** denoting non-adjacency, `roads` has `Road` objects where roads can be built and **`null`** where they cannot. The `Road` data type is very simple, containing only a pair of intersection IDs (`iOne` and `iTwo`) and a reference to a `Player` owner. However, it does contain a few useful methods to be used for interacting with roads: `other`, `common`, `both`, `isNeighbor`, `canBuild`, and `build`, where the first 4 deal with interacting with intersection IDs and the last 2 deal with the player owner of the road. `other` and `common` are closely related: the former takes an ID and returns the other ID (or `Constants.INVALID`, which is equal to -10), while the latter takes a `Road` reference `r` and returns the ID that **`this`** and `r` have in common, if any (or `Constants.INVALID` if there is no shared intersection). `isNeighbor` builds on `common`, returning **`true`** if there is a common intersection and **`false`** otherwise, and `both` simply returns both intersection IDs (it is worth noting that these are always sorted in ascending order due to how the `Road` constructor behaves).

The remaining methods in `Road` (aside from `getPlayer`, a retrieval method for the `player` field, and a debugging `toString` method) are used to assign an owner to the road, since `Roads` are initialized with a **`null`** owner: `canBuild` verifies that the `Road` is unowned, and `build` changes the owner to the one provided as a parameter. Note that `Road` does not check whether players have enough resources to build roads, since this verification is performed higher up in the class hierarchy (discussed in a later section). Having discussed the `Road` data type, it is now possible to explain the initialization of `Board`'s `roads` field, which is much easier than that of hexes and intersections (discussed in great detail above). Since `roads` is parallel to `iGraph`, which has already been initialized in `Rules`, all that needs to be done is to create a new `Road` instance with row index `i` and column index `j` wherever there exists a **`true`** value in `iGraph`. Once the last `Road` is created, the game board is fully initialized and is ready to be rendered by `BoardDraw`, the subject of the next section.

4.4. Rendering the Board with `StdDraw` [11]

The `BoardDraw` class is responsible for graphically rendering the game board, using the `StdDraw` library (part of the Princeton IntroCS package of standard libraries) to create a new window with a blank canvas and then draw each board element at the proper coordinates. The minutiae of performing the actual drawing operations are dealt with by `StdDraw`, so the main task of `BoardDraw` is to calculate the proper coordinates for the various Catan board elements and call the appropriate `StdDraw` methods. To initialize a `BoardDraw` object, all that is needed is a reference to a `Board` and, optionally, a `dim` parameter that determines the dimensions of the rendered window in pixels (the window is square, so only one value is necessary). If no dimension parameter is provided, the `DEFAULT_DIM = 700` value is used instead, which works well for a standard size board but is too small for larger boards. The `dim` field is used to dynamically compute the dimensions of the game elements to be drawn, such that the components of a rendered board of any size are approximately proportionate and visually pleasing. These computed values will be discussed in more detail in the relevant sections on drawing different board elements, but it is worth noting here that the values are all calculated in the `initCoords` and `initCanvas` methods, such that the `BoardDraw` instance is ready to draw the board using the `draw` method once the class constructor returns.

4.4.1. The `HexShape` Data Type Central to `BoardDraw`'s dimension and coordinate calculations is the `HexShape` data type, whose main task is to compute the x and y coordinates of the 6 vertices of a hex using the coordinates of the center point of the hex and the size of one of 3 possible dimensions of the hex. In order to explain the inner workings of `HexShape`, it is first necessary to introduce some terms and concepts involved in the geometry of a regular hexagon (i.e. one with sides and angles of equal size). First and simplest is the rotation of a hex, which can either be `FLAT` or `BALANCE`. The flat orientation is such that a hex has two of its sides positioned parallel to the x axis. Conversely, the balanced orientation is such that a hex has two of its sides positioned parallel to the y axis. The names of these orientations refer to how a hexagon would behave if placed on an infinite line in a two-dimensional world where gravity acts downward (i.e. in the negative y

direction): a `FLAT` hexagon would remain stationary, while a `BALANCE` hexagon would "roll" in the negative or positive x direction unless perfectly balanced on its bottommost vertex.

The next detail to be considered is the geometry involved in describing a regular hexagon. The first and most obvious dimension is the length of a side, the `HexShape` field for which is aptly titled `side`. Another hexagon dimension is defined by the perpendicular distance between two parallel sides, which is titled `flatHeight` for the `FLAT` orientation and `balanceWidth` for the `BALANCE` orientation. The third and final dimension is the distance between the farthest 2 vertices, which is titled `flatWidth` and `balanceHeight`. These dimensions can also be defined in a different way. Consider an arbitrary vertex indexed by v, where any of its neighbors is $v+1$, the next neighbor is $v+2$, and so on. The distance between v and $v+1$ is `side`, the distance between v and $v+2$ is `flatHeight`, and, finally, the distance between v and $v+3$ is `flatWidth`.

In order to initialize a `HexShape` instance, 5 parameters are needed: `xCenter`, `yCenter`, `rotation`, `param`, and `selector`, where `xCenter` and `yCenter` are the coordinates of the center of the hexagon, `rotation` describes the orientation, `param` is the value of one of the 3 dimensions defined above, and `selector` specifies which of the dimensions is provided by `param`. Since only one of the 3 dimensions defined above is needed to compute the other two, `param` is the only argument that needs to be passed to one of the 5 `given*` methods (where `*` denotes any of the 5 names for the 3 dimensions), all of which use the geometric relationship among the dimensions: $side = x$, $flatHeight = \sqrt{3}x$, and $flatWidth = 2x$.

Once the 5 dimension aliases are calculated, the final step is to compute the coordinates of the 6 vertices, which is done in the `computeCoords` method. This method takes advantage of the fact that all 6 vertices are equidistant from the center of a regular hex, where the distance is equal to one half of a side. Thus, the 6 coordinate pairs are calculated once, and then `rotation` is used to determine which coordinate array corresponds to `xCoords` and which corresponds to `yCoords`. The resulting order of coordinates is as follows: in the `FLAT` case, the vertex indices ascend clockwise from the vertex with the lowest x coordinate, and in the `BALANCE` case, the indices ascend counterclockwise from the vertex with the lowest y coordinate.

4.4.2. `BoardDraw` Preliminaries Having explored the important `HexShape` data type, the discussion can now turn to the `BoardDraw` class proper. When its constructor is called, `BoardDraw` first stores references to the relevant fields of the provided `board` parameter and also some **`static`** fields of `Rules`, particularly `hRings` and `iRings`. Then, it computes `xCenter` and `yCenter` (using `dim`) and initializes a large flat `HexShape` to be used as the "ocean" in which the island of Catan will reside. Other important computed values include `w`, `h`, and `s`, which are the `balanceWidth`, `balanceHeight`, and `side` dimensions of a typical hex. `w` is calculated so that the entire width of a Catan board can fit in the ocean hex and still have room for a one hex padding on each side, where the width of the board in hexes is `(2 * radius) - 1`. `h` and `s` are calculated from `w` when a `HexShape` is created to represent the center hex (located at row and column indices 0 in `hRings`).

4.4.3. The `getNextX` and `getNextY` Methods Somewhat surprisingly, two humble and simple methods underpin the coordinate calculation algorithms for hexes, intersections, and ports. As their names suggest, these methods compute the "next" coordinate pair given the "current" coordinate pair and the relevant `HexShape` dimensions, where "current" and "next" refer to indices in a loop (since these methods are meant to be loop helpers). The concrete meanings of "current" and "next" are context-specific, but the general idea is that there is a movement from one point to another in one of 6 possible cardinal directions. These directions are best understood as lines of "outward" motion that are perpendicular to the 6 sides of a reference `BALANCE` hex, where "outward" refers to increasing distance from the center of the hex. The cardinal directions represented are northwest, west, southwest, southeast, east, and northeast, which are stored as constants in `HexShape` and referred to by their abbreviations (`NW`, `W`, `SW`, `SE`, `E`, and `NE`). The numeric values of these constants were purposely chosen to facilitate the computation of hex coordinates (the subject of the next section), but they also turned out to be useful for the computation of intersection coordinates (together with the application of some cleverness).

4.4.4. Hex Coordinates Since the `HexShape` data type performs the actual hex coordinate calculation, all that `BoardDraw` needs to do in order to compute the hex coordinates is to find all the

hex centers and then initialize an instance of `HexShape` for each hex. Using `hRings` (introduced in the Rings of Catan section), sextants (introduced in the Hex Graph section), and the `getNextX` and `getNextY` methods (introduced in the previous section), it is actually quite simple to find the hex center coordinates. The algorithm loops through each ring in `hRings` (except for the first one, which is just the solitary center hex and can be initialized separately), first calculating the coordinates of the hex at column index 0 (i.e. the initial `curHex`), where `hexYCenters[curHex]` is always equal to `yCenter` and `hexXCenters[curHex]` is simply `i` hex widths to the right of `xCenter`, expressed in code as `xCenter + (i * w)`. The "magical" step in the algorithm occurs when transitioning from a given `curHex` to `nextHex`: the `getNextX` and `getNextY` methods are called with parameters specifying the current coordinate, the appropriate combination of `w`, `h`, and `s`, the sextant (which is equal to `j / i`). The crucial observation here is that the sextant number (in the range 0-5) neatly coincides with the `HexShape` cardinal directions described in the previous section, a correspondence purposely established by design.

4.4.5. Intersection Coordinates The intersection coordinate calculation algorithm is somewhat more complicated than that for computing hex coordinates, but with some creative effort the same underlying mechanism of relying on the `getNextX` and `getNextY` methods can be applied. The clever insight here is that an intersection ring can be separated into two halves, one for intersections with odd column indices and one for even-indexed intersections, and these halves can be traversed simultaneously using two special indices, unimaginatively titled `iOdd` and `iEven`. Recall from The `HexShape` Data Type that the distance between a vertex and its second-degree neighbor is simply `balanceWidth`, which is also the distance between any two hex centers. Not only are these distances the same, they are defined by the same 6 possible distance vectors (i.e. line segments of length `balanceWidth` oriented in the 6 `HexShape` cardinal directions). Thus, the same `getNextX` and `getNextY` methods that were used to compute the hex centers can be called upon without modification, so the only task left is to determine which cardinal directions to use and when.

The overall code in the `initInterCords` method utilizes a 3-loop structure, where the outer loop iterates over `i` in `iRings`, the middle loop iterates over `j` for each `HexShape` cardinal

direction, and the inner loops iterate over `k` for values `oddInc` and `evenInc`. These values flip back and forth between `i + 1` and `i`, which count the number of hops from second-degree neighbor intersection to second-degree neighbor in the same direction before the direction has to change to a different cardinal orientation, for odd and even intersections. In programmatic terms, the flip is accomplished by `((*Inc - i + 1) % 2) + i`, where `*Inc` represents `oddInc` and `evenInc`. This may look complicated, but all that is happening here is that `i` is subtracted out, odd and even get swapped by taking the modulo 2, and then `i` is added back. The `j` and `k` loops taken together create a situation where, for each cardinal direction (`j`), `k` new intersections are traversed by following the current cardinal vector (of magnitude `balanceWidth`). When these two loops finish, the result is that an entire intersection ring has been traversed, the size of which is equal to the sum of the sizes of the two halves indexed by `iOdd` and `iEven`. The role of `getNextX` and `getNextY` in this algorithm is to calculate the next intersection coordinate for both the `oddInc` and `evenInc` `k` loops. Thus, the role of the `i`, `j`, and `k` loops is really just to iterate appropriately so that the `getNextX` and `getNextY` workhorse methods can perform their essential task of calculating intersection coordinates.

4.4.6. Port Coordinates The last coordinates that need to be calculated before the board can be rendered are those of the maritime ports scattered around the edge of the island of Catan. The only prerequisite needed to begin this computation is the knowledge of the coordinates of the intersections that are attached to ports, and this information is acquired by accessing `interXCoords` and `interYCoords` using the intersection indices stored in `portLocations`, which (as discussed in the Intersections section) is initialized by the `randomizePorts` method in the `Board` class and can be accessed using the `getPortLocations` retrieval method. The algorithm loops through `portLocations`, incrementing `i` by 2 after each iteration because the number of actual ports is half that of intersections attached to ports. Since the coordinates of these intersections have already been computed in `initInterCoords`, the only thing left to do is to travel an appropriate distance outward from the port intersections and record the resulting coordinates.

The distance and direction to be traveled is chosen such that each port coordinate lies at precisely

the location where a hex center would be found if the board had at least one more hex ring. It is easier to start at the port intersections rather than at the centers of the hexes in the last hex ring, so the distance is reduced by a factor of 2, since the port intersections are positioned exactly midway between the existing hex centers and the desired port centers. This means that, when calling the `getNextX` and `getNextY` methods, it is merely necessary to provide half of the `w`, `h`, and `s` fields in `BoardDraw`, along with the appropriate `HexShape` cardinal direction and the coordinates of the closest point that is equidistant from the intersections in each pair. These directions are determined by calculating the difference between the *x* and *y* coordinates of each intersection pair adjacent to a port and checking which case the differences fall under (note that it is necessary to round the differences to ensure that the cases which check for zero differences are properly detected). These cases are chosen empirically with regard to the geometry of the game board, specifically the direction that the imaginary line perpendicular to the line segment connecting each intersection pair must follow in order to move away from the board. Finally, the coordinates computed by `getNextX` and `getNextY` are stored in `portXCoords` and `portYCoords` at index `i / 2`, undoing the doubling of `i` caused by traversing `portLocations` by twos.

4.4.7. Drawing Hexes Now that the hex coordinates are known, it is time to actually draw the hexes using `StdDraw` method calls. The first step is to draw a filled hexagon of the appropriate color for each hex in `hexes`, which is done by calling `StdDraw.setPenColor` to change the color and then `filledPolygon` to perform the drawing. The proper color for each `Resource` type was determined empirically by sampling from the hex tile graphics in the official game rules, with the resulting correspondences of `0xFFFF99` for `DESERT`, `0x00CC00` for `WOOL`, `0xFFCC00` for `GRAIN`, `0x006600` for `LUMBER`, `0xCC6600` for `BRICK`, and `0x666666` for `ORE`, where all colors are in the RGB (red-green-blue) colorspace. Since `StdDraw` does not support drawing a border of a different color on a drawn polygon, it is necessary to simulate borders on each hex by setting the pen color to `StdDraw.BLACK` (shorthand for Java's `Color.BLACK`) and then calling `StdDraw.polygon` to draw just the border of a black hexagon, for the sake of game board readability (it is easier to see contrast between adjacent hexes with the use of black borders).

As with a physical Catan board, the dice roll chits need to be placed on top of the hexes, a task performed by the `drawChits` method. A white filled circle and a black unfilled circle are drawn for each hex, using the empirically chosen `chitRadius` (equal to `w / 5`) initialized in `initCanvas` to determine the size of these circles. Then, bold Arial text for the dice roll number is drawn in the center of the chit (centered in the hex), where the font size is chosen as `w / 5`. Finally, smaller text of the same font face but size `w / 11` is drawn three-quarters of the way down from the hex center, where the text describes the integer ID of the hex being drawn. This is needed on the board because the text-based interface (the subject of a later section) requires that players enter a hex ID when moving the robber piece, an event that takes place whenever a 7 is rolled or a Knight development card is played. One last note about dice roll chits is that the chit on which the robber is placed is denoted by a background colored as `StdDraw.DARK_GRAY` (instead of white) and the dice roll number is (temporarily) replaced with a bold white "R".

4.4.8. Drawing Intersections There are three possibilities for what is drawn at an intersection, determined by the `OPEN`, `SETTLEMENT`, and `CITY` constants in the `Building` data type (recall from the Intersections section that each intersection stores a `Building` instance). In the case where nothing has been built yet, a small white circle with a black border is drawn (again, by drawing a filled white circle and then an unfilled black circle), together with the integer ID of the intersection in small font (the same as that used to write the hex ID above). The ID is included so that players can specify where settlements are to be built when interacting with the text-based interface. For intersections where a settlement has been built, a slightly larger filled circle of the player's color (stored in `Player.COLORS` and indexed by the player's ID) with a black border is drawn. Finally, in the case where a city has been built, the same process as for a settlement is followed, with the addition of a black inner circle drawn on top of the settlement (of a radius which is two-thirds of that of a settlement). Once again, the specific values chosen for `intersectionRadius`, `buildingRadius`, and `innerCityRadius` were determined empirically by drawing boards of different sizes and attempting to scale the dimensions so that intersections look good for any board size (to the extent possible for the given `dim` parameter). It is worth noting here that the

appearances chosen for settlements and cities are arbitrary, selected primarily for ease of drawing (since a circle is only determined by its center and radius).

4.4.9. Drawing Ports Drawing ports is a simple affair once the coordinates have been calculated. Ports are represented by a black-bordered filled circle of a color denoting the port's resource for the `SPECIFIC` type, or simply black for the `GENERIC` type. The port's trade ratio is rendered as white bold text on top of the port's circle, which is 3:1 for `GENERIC` ports and 2:1 for `SPECIFIC` ports, as specified in the Engaging in Trade section. The final step in the port drawing process is to draw thin black lines from each port to the intersections that are attached to it, so that players can know which intersections yield the improved port trade rates and make settlement building decisions accordingly. It is worth noting here that `INLAND` ports are never rendered since they are not "real" ports but rather a convenience for implementing the default 4:1 resource trade ratio.

4.4.10. Drawing Roads The final step in the game board rendering process is to draw the roads built by players during the game. Note that, although `Road` instances are generated together with the board, no roads are drawn until their player ownership has been changed from **`null`** to an actual `Player` instance using the `Road.build` method. Each road is rendered by drawing a line between the two intersections that are connected by the road (retrieved using the `Road.both` method), where these lines show the player owner's color and also a black border. To achieve the black border effect, a thicker black line is drawn first and then a thinner colored line is drawn on top, where the thicknesses of the lines are set using `StdDraw.setPenRadius` and the empirically generated `roadInnerRadius` and `roadOuterRadius` values.

4.5. Resource and Development Cards

The discussion thus far has covered the operation of the program from the terminal command prompt through the `Game` constructor and into the `startGame` method, including the calls to `Rules.init` and `setUpBoard`, which creates a new `Board` and then a `BoardDraw` instance tied to that board. The next step in `startGame` is a call to `setUpDecks`, which, as its name suggests, is tasked with initializing the data structures used to represent the resource and development cards to be used in the game. These data structures and their initialization are considerably simpler

than those required for the game board, so the next two sections which cover them in detail are accordingly much shorter in length.

4.5.1. The `Resource` and `ResourceBundle` Data Types The `Resource` data type has already been covered in the Resources section, but it is worth repeating here that it serves the dual purpose of representing hex tile resources and the resource cards distributed to players as specified in the game rules (see the Game Setup and Rolling the Dice sections for more information on resource card allocation). In order to make it easier to work with bundles of cards (i.e. players' hands, the resource deck, and the cards used in trading and to pay for building game objects), the `ResourceBundle` data type is used, which at its core is simply a two-dimensional `ArrayList` of `Resource` instances. This data type exposes a number of convenience methods for manipulating the underlying array structure, following standard Java nomenclature whenever possible (e.g. for methods such as `add`, `remove`, `isEmpty`, and `size`). Most of these methods exist in multiple flavors (using argument overloading) so that the client classes calling them can worry less about dealing with implementation minutiae and more about performing operations relevant to the game.

The general pattern for these overloaded methods is to have one version for interacting with a single `Resource` or integer resource type and another version for interacting with multiple resources. The `add` methods take either a single `Resource` instance or another `ResourceBundle` instance that is to be merged into **`this`**. Similarly, `canRemove` and `remove` are called either with an integer resource type or an array of size `Resource.NUM_TYPES` specifying the number of each type of resource to be removed (particularly useful when enforcing the resource costs for players building various game objects). The final overloaded method pairs are `isEmpty` and `size` and take either no argument or an integer resource type, where the result is to give information about the size of either the overall data structure or a specific resource "bucket".

There are two more methods in `ResourceBundle` worth mentioning: `removeRandom` and `toString`. The former is used to remove a random resource card from a resource bundle (as the name suggests), which is useful when simulating resource card stealing (recall from the Rolling the Dice section that whenever a player rolls a 7 s/he is entitled to move the robber and steal

a resource card from another player). The latter method is used to print the contents of the resource decks and players' hands when it is their turn to play, since there is no graphical inferface for displaying the decks and players must know which resources are available in order to make gameplay decisions. The contents of a `ResourceBundle` are printed according to the following format: `Wool: x*; Grain: x*; Lumber: x*; Brick: x*; Ore: x*;`, where `*` is a placeholder for the actual number of cards of each type (note that any zero counts are not printed).

4.5.2. The `DevCard` and `DevCardBundle` Data Types The design of the data structures for development cards parallels that of the data structures for resource cards, where `DevCard` wraps around the basic integer types used to denote the different development cards (and also the `String` names for the cards) and `DevCardBundle` provides convenience methods for dealing with groups of cards (i.e. the main game deck and the cards in each player's possession). Like `ResourceCardBundle`, the `DevCardBundle` class uses a two-dimensional `ArrayList` as the underlying data structure for storing development cards, and in fact the method names and functionalities are almost identical between the two classes. However, there do exist a few differences that merit mention and brief explanation.

The `DevCardBundle` class enforces an additional check on the maximum size of a bundle, using `Rules.MAX_DEV_CARDS` to ensure that no more cards of any given type than the allowed maximum can be added to a bundle (note that the maximums do not scale up with the board size, unlike the many other board features already discussed). The remaining differences between `ResourceBundle` and `DevCardBundle` are omissions of methods that are useful for manipulating resources but not development cards. Specifically, the versions of `canRemove` and `remove` that take an array of card counts, which are present in `ResourceCardBundle`, are absent from `DevCardBundle` because there is never a need for removing multiple cards from a development card bundle at any point in the game (whereas resources are spent to build game objects, development cards are only played or drawn from the deck one at a time).

4.6. The `Player` Data Type

Following the initialization of the resource and development card decks in the `setUpDecks` method, the next step in `Game` is to instantiate the players. The `setUpPlayers` method itself is very simple, creating a sufficient number of `Player` instances, as specified by the `numPlayers` field, and then shuffling them to determine a random play order (note that actually rolling dice as in the official Catan rules is unnecessary, although adding the visual feedback for such a process would perhaps improve the user experience). However, the `Player` data type is fairly complex, storing a nontrivial amount of information about the logical possessions of each of the game's players and also implementing a substantial amount of functionality for manipulating player status and enforcing game rules (such as the build costs stored as constants in `Rules`).

4.6.1. Relevant Player Information The most basic detail concerning a `Player` instance is its integer ID, stored in `id` and associated with the 8 player colors and `String` names (which are constants). In addition to the standard 4 player colors of blue, orange, red, and white, the implementation presently being discussed includes 4 additional colors (green, light gray, magenta, and pink) so that games played on larger boards can be more interesting. Although additional players beyond these 8 are technically possible, `BoardDraw` treats all players with higher IDs in the same way, drawing their roads, settlements, and cities in black, so in practical terms it is infeasible to play with more than 9 players.

The next category of relevant player information is that of board objects owned by players, specifically roads, settlements, and cities. Although player properties built on the game board can be directly retrieved from `Board` and its underlying data structures, it is more convenient to keep track of each player's roads and buildings directly in `Player`. The `roads`, `settlements`, and `cities` fields are fairly self-explanatory, storing lists of references to each board object owned by a given player. Note that the latter two refer to `Intersection` objects rather than `Building` objects, since `Intersection` contains not only a `Building` instance but also other useful information (consult the Intersections section for more details). In addition to the board objects themselves, there also exist 3 integer counts that keep track of how many cost-free structures can be

built: `freeRoads`, `freeSettlements`, and `freeCities`. Although the latter is used strictly for debugging, the former two are needed for the special first two turns of the game in which each player is permitted to place 2 settlements and roads free of charge. Appropriate constants exist in `Rules`, but it is also necessary to keep track of when these "freebies" are exhausted so that players can start paying resource cards for any further construction.

The final category of `Player` fields is that of cards and victory points (VPs). The cards, implemented as `ResourceBundle` and `DevCardBundle` objects, are stored in `resourceCards`, `devCards`, and `playedDevCards`. While there is presently no significant difference between the latter two fields (aside from the role of `playedDevCards` in keeping track of played Knights for the largest army calculation), they exist as separate entities in anticipation of the implementation of a proper graphical interface, since in the physical version of Catan all players can see the development cards that have already been played but not those that have not yet been played. A similar distinction exists with `publicVP` and `privateVP`, where the former incorporates VP development cards that have been played and other VP sources while the latter only includes VP development cards that have not been played. Finally, `hasLongestRoad` and `hasLargestArmy` represent the large cardboard cards that are given to players who satisfy the eponymous conditions in the physical game, each worth 2 VP as stated in the official rules (`LONGEST_ROAD_VP` and `LARGEST_ARMY_VP` in `Rules`).

4.6.2. Enforcing Building Rules Recall from the Intersections and Roads sections that the `Building`, `Intersection`, and `Road` data types perform no enforcement of build costs. This task is delegated to the `Player` data type, which is adequately equipped to engage in such verification since it has knowledge of the resource cards in a player's hand and, given the resource deck, can pay the appropriate cards for each structure that is built. The `canBuild`$\star$ family of methods (where $\star$ is a wildcard) performs the appropriate checks before any structures can be built, where the flavors taking no arguments verify a player's ability to satisfy build costs and restrictions and the versions that take an appropriate reference to a board data structure check the attempted placement against any possible adjacency requirements and limitations.

Specifically, the `canBuild*` methods without arguments check whether the appropriate number of resource cards can be removed from a player's hand (the `resourceCards` field) using `ResourceBundle.canRemove` in conjunction with the build cost constants in `Rules` (unless the relevant `free*` fields in `Player` are nonzero, denoting cost-free structures are available) and also whether the player has any structures left to build (recall from the Building Roads, Settlements, Cities, and Development Cards section that each player may build up to 15 roads, 5 settlements, and 4 cities for the standard board size). The `canBuild*` methods taking specific instances of game objects call upon the argument-free `canBuild*` methods for initial verification and then perform additional verification as dictated by the rules concerning the specific structure type. Roads must generally be placed adjacent to other roads (checked using `Road.isNeighbor`), except for the first two roads placed at the beginning of the game, each of which must be adjacent to the settlement placed in the same turn (checked using `Road.other` and the settlement ID). Settlements must generally be placed adjacent to a road, except for the first two settlements (which can be placed anywhere), and cities must always be placed on an intersection where an existing settlement lies. All of these checks and verifications are called by the `build*` methods, and if the results are affirmative then the relevant structures are built (note that methods for development cards exist as well, in addition to those for roads, settlements, and cities).

4.6.3. Player Card Operations Aside from the resource card debits that occur when the `build*` methods are called and return successfully, other card operations are also supported by the `Player` class. Resource acquisition is implemented by the two `collectResources` flavors (one for the initial resources given out at the start of the game and the other for when the dice are rolled) and the `stealResource` method variant that receives a stolen resource from another player (i.e. when the robber is moved). On the flip side, resource removal is implemented by the two `discard` variants (which are not actually used in the game) and the `stealResource` version that takes another `Player` instance as a parameter. Combining resource credits and debits, the `doPortTrade` method implements trading resources at ports (refer to the Engaging in Trade section for more details), using the `findBestRatio` method to search the `Port` references

stored at each player-owned `Intersection` for the best trade ratio for a given resource. The final card operation implemented in `Player` is performed by the `playDevCard` method, which simply moves a card of the specified type from `devCards` to `playedDevCards` and adjusts the VP values if a VP card is played, decrementing `privateVP` and incrementing `publicVP` (see the Victory Conditions section for a complete listing of the development cards that grant VP).

4.7. The `UserInput` Class

Since the implementation presently being discussed does not utilize a graphical interface (aside from the dynamic board image covered in the Rendering the Board with `StdDraw` section), it is necessary to collect user input via the terminal command prompt instead. The role of the `UserInput` class is to provide basic text input functionality and to store important messages that are to be displayed to the user at various points in the game. It uses a standard Java `Scanner` instance wrapped around `System.in` to collect the actual input and then implements some basic verification methods to ensure that the collected input satisfies certain criteria, depending on which method is called at a given point. Aside from these, the only other method implemented is `doPrivacy`, which simply prints 100 blank lines whenever control switches from one player to another (at the end of a turn) so that the next player cannot "peek" at the previous player's sensitive information. Finally, numerous `Strings` are stored as constants, containing various prompts and informational tidbits that are to be displayed to the current player, contingent on different actions that the player takes (the actual `System.out.println` calls are in `Game`, but `UserInput` stores the messages).

4.8. Putting It All Together

In the preceding sections, a number of important data types and components of game functionality were covered in great detail, and now that all the relevant prerequisites have been explored it is possible to finally finish painting the full picture of how a typical game of Catan works in the implementation presently being discussed. Recall that the conversation thus far has followed the initialization of `Game` into the `startGame` method and through the call to the `setUpPlayers` method. The last steps in `startGame` are to initialize `UserInput`, prompt the players to place

their first two settlements and roads using `firstMoves`, allocate the starting resources, and then finally call `gameLoop`, where all remaining game activity takes place.

4.8.1. First Moves Recall from the Game Setup section that, once the turn order is determined (implemented in `setUpPlayers`), the 4 (or more, as the case may be, up to 8 or 9 players in this implementation) players take turns placing two settlements and roads, with the first "round" taking place in the normal order and the second round taking place in the reverse order. These special player turns are implemented in the `doInitialTurn` method, which calls `getSettlement` and `getRoad` to actually prompt the players to enter appropriate IDs (rendered on the board in small print for reference) and perform additional verification. Aside from the input verification performed by `UserInput`, these methods also check whether players are permitted to build at the chosen locations (whether there exists a road between the given integer IDs, whether settlements are spaced too closely together, et cetera). Once each player has placed two settlements and roads, initial resources are distributed according to the hexes adjacent to each player's second settlement, using `Intersection.getHexes` to determine the resource counts (one for each of the 3 hexes) and then `ResourceBundle.remove` and `ResourceBundle.add` to remove the resource cards from `resDeck` and add them to each player's `resourceCards`.

4.8.2. The Game Loop After the initial game setup is complete, the `gameLoop` method is entered and players take turns interacting with the game in the order randomly determined in `setUpPlayers`. Each iteration of the loop (i.e. a player's turn) closely follows the steps outlined in the Typical Turn Workflow section: the "dice" are rolled (i.e. `Math.random` is called twice), the current player (stored in the `pCurrent` field) is prompted to enter turn commands in a loop in `doTurn` until the `UserInput.END_TURN` command is entered, and then control switches over to the next player and the main game loop enters its next iteration. Inside this simple structure is encapsulated a copious amount of game logic for dealing with player input, accepting or rejecting it and then reacting to it, and enforcing the various game rules. Before continuing into more detail, it is worth noting that all major Catan features are implemented except player-to-player trading (and, as a result, players do not discard resources when a 7 is rolled), and only a few minor game features

that require user input are altered for convenience and make random decisions for the player instead.

All user input that takes place once `gameLoop` is entered takes place in the `doTurn` method, except for the prompt to move the robber when a 7 is rolled. At its core, `doTurn` is just a large **`switch`** statement that takes different actions based on the command entered by the current player, which is first verified by `UserInput` to ensure that it falls within one of the cases in the **`switch`** statement. The commands that involve building game objects are handled by the `getRoad`, `getSettlement`, `getCity`, and `getDevCard` methods, each of which performs a number of relevant additional checks on the user input such that the game rules are not violated (refer to the code for more details, but be advised that the implementation of these methods is largely procedural and not particularly interesting).

The other possible commands (aside from `TRADE_PLAYER`, which as mentioned above is not implemented) are `TRADE_PORT` and `PLAY_DEV_CARD`. The former command simply defers to `Player.doPortTrade`, where `pCurrent` is the relevant `Player` instance. The latter command calls the `playDevCard` method, which uses a **`switch`** statement to determine what action needs to be taken depending on which development card is chosen by the player (the code for this is also somewhat procedural and boring). An important thing to note here is that there exists a field in `Game` that is titled `newDevCards` and keeps track of the development cards built during the current turn so that players cannot buy these cards and then use them right away (since doing so would violate the game rules). Whenever the `BUILD_DEV_CARD` command is used, the card returned by `pCurrent.buildDevCard` is added to `newDevCards`, and then when the `playDevCard` method is called this field is accessed and compared with the development card counts in the `DevCardBundle` returned by `pCurrent.getDevCards`. Finally, at the end of each player's turn, `newDevCards` is emptied using the standard `ArrayList.clear` method.

The last noteworthy aspect of the game loop is the calculation of player victory points (VPs). Recall from the Relevant Player Information section that the `Player` class keeps track of each player's `publicVP` and `privateVP`, which can be collectively accessed using `Player.getVP`. The division of labor is such that the `Player` data type deals with the VPs gained by building

settlements, cities, and VP development cards and the `Game` class handles the VPs granted by the Longest Road and Largest Army cards (recall from the Victory Conditions section that 5 roads and 3 Knights are the minimum requirements to earn these cards and that each card grants 2 VPs). In order for `Game` to update players' VP when Longest Road and Largest Army are assigned, the appropriate `Player` methods are used: `giveLongestRoad`, `takeLongestRoad`, `giveLargestArmy`, and `takeLargestArmy`. This updating takes place whenever an action is taken by a player that could potentially influence the longest roads and largest armies in play. For the longest road calculation, this includes both the building of roads and the building of settlements, since the rules of Catan specify that a player's longest road is broken if an enemy player builds a settlement on an unoccupied intersection along the road's length. For the largest army calculation, the only action that needs to be accounted for is the playing of a Knight card.

In order for the owners of the Longest Road and Largest Army cards to be properly updated, it is necessary to compute the longest roads and largest armies of each player, performed by the `findLongestRoad` and `findLargestArmy` methods. The largest army calculation is trivial, since the `Player` class keeps track of the development cards that have been played and exposes a `getLargestArmy` method that simply counts the number of Knight cards in `playedDevCards`. The longest road calculation, however, is considerably more involved and requires the use of a non-recursive/recursive method pair, both titled `findLongestRoad` but distinguished from each other and the argument-free variant through Java's automatic argument overloading. The non-recursive method uses set operations (implemented in Java's standard `TreeSet` data type) and the various `Road` helper methods to create a set of road termini, which are simply intersections where a road ends, and also a temporary `iGraph` for just the player whose roads are being traversed. Then, for each road terminus intersection, the recursive method is called, and then the longest road produced by all of these calls is returned as the definitive longest road for the given player.

The recursive method has three base cases that cause it to return without making any additional recursive calls: either another terminus is reached, a loop is detected, or an enemy intersection is

found to be blocking the traversal path. The algorithm keeps track of the intersections it visits, which allows it to handle graph loops and branches without looping back on itself. An important detail to take note of here is that the loop base case is different from the other two, returning `idsVisited.size` instead of `idsVisited.size() - 1` since the extra link back to the loop is not traversed but still needs to be counted (recall from basic graph theory that the number of edges in a line graph is one less than the number of vertices). Once all recursive frames are terminated, the initial call to the recursive method returns and shortly thereafter the non-recursive `findLongestRoad` returns as well with the final result for the given player.

The game loop checks the victory condition at the end of every iteration using the `getWinner` method, which returns **`null`** while there is no victor and a `Player` instance when the victory condition is met. The `getWinner` method iterates through `players`, calls each player's `getVP` method, and compares the result against the value returned by `Rules.getMaxVP`. When a winner is finally determined, a victory message is printed to `System.out`, then `StdDraw.save` is called to save a .png file depicting the final state of the game board, and finally `System.exit` is called (since `StdDraw` would keep its window open indefinitely otherwise).

5. Challenges Encountered

6. Future Features

7. Conclusion

References

[1] R. Clobus *et al.*, "Pioneers - The Settlers of Catan," http://sourceforge.net/projects/pio/.

[2] V. Costescu, "COS 984 Senior Thesis 2014," https://github.com/lbds137/cos984-senior-thesis-2014.

[3] S. De Toni, "Solitaire Settlers of Catan ComputerGame," http://sourceforge.net/projects/solitairecatan/.

[4] B. Eskin, "Like Monopoly in the Depression, Settlers of Catan is the board game of our time," http://www.washingtonpost.com/wp-dyn/content/article/2010/11/24/AR2010112404140.html, Nov. 2010.

[5] J. Fugate, "Settlers3D," http://sourceforge.net/projects/settlers3d/.

[6] K. Law, "Settlers of Catan: Monopoly Killer?" http://mentalfloss.com/article/26416/settlers-catan-monopoly-killer, Nov. 2010.

[7] S. Maddock, "HTML5 Settlers of Catan," https://github.com/samuelmaddock/html5-settlers-of-catan.

[8] M. Masnick, "How Lawyers For Settlers Of Catan Abuse IP Law To Take Down Perfectly Legal Competitors," http://www.techdirt.com/blog/wireless/articles/20110211/21200213066/how-lawyers-settlers-catan-abuse-ip-law-to-take-down-perfectly-legal-competitors.shtml, Feb. 2011.

[9] J. Monin, "JSettlers2 - Java Settlers of Catan," http://sourceforge.net/projects/jsettlers2/.

[10] A. Raphel, "The Man Who Built Catan," http://www.newyorker.com/online/blogs/currency/2014/02/klaus-teuber-the-settlers-of-catan.html, Feb. 2014.
[11] R. Sedgewick and K. Wayne, "Standard Libraries," http://introcs.cs.princeton.edu/java/stdlib/.
[12] K. Teuber, "The Official Website for Settlers of Catan - About Us," http://www.catan.com/about-us/klaus-teuber.
[13] K. Teuber, "The Official Website for Settlers of Catan - Electronic Games," http://www.catan.com/electronic-games.
[14] K. Teuber, "The Settlers of Catan Game Rules & Almanac," http://www.catan.com/files/downloads/soc_rv_rules_091907.pdf, Sep. 2007.
[15] J. Wingrove, "Settlers of Catan: A low-tech island retreat for a plugged-in generation," http://www.theglobeandmail.com/technology/settlers-of-catan-a-low-tech-island-retreat-for-a-plugged-in-generation/article583018/, Jun. 2011.