Generating Solitaire Games

A Major Qualifying Project report:

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the degree of

Bachelor of Science

March 3, 2023

Submitted by:

Patrick Eaton

Benjamin Gilchrist

Alex Martinho

Kyle McFatter

Jason Odell

Advisor:

Professor George T. Heineman

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see http://www.wpi.edu/Academics/Projects.

Abstract

This project represents the culmination of a series of MQP projects advised by Professor George Heineman, with the addition of functionalities that further enhance the projects overall scope and capabilities. It focuses on developing a Solitaire code generation framework that utilizes Combinatory Logic Synthesis (CLS) to automate the creation of Java and Python code for various Solitaire games.

Table of Contents

Abstract 2

Table of Contents 3

1 Introduction 4

2 Background 7

3 Goals 9

4 CLS Solitaire Code Generation 12

4.1 Solitaire Model 12

4.2 Structure 13

4.3 Layout 14

4.4 Deal 15

4.5 Specialized Elements 17

4.6 Moves 18

4.7 Logic 19

4.8 Customized Setups 20

4.9 Completing the Model 21

5 CLS Solitaire Generator in Java 21

5.1 Adding Combinators 24

5.2 Abstraction 26

6 CLS Code Generation in a Different Domain 30

6.1 ChatGPT comparison 37

7 PySolFC Code Generation 38

7.1 Adding Simple Simon 39

7.2 Adding Several Variants 40

7.3 Adding Spider 43

8 Travis 44

8.1 Testing Code Generation 45

8.2 Common issues: 46

8.3 Testing Generated Variants 46

8.4 Exporting Generated Variants 51

9 Solitaire Player 53

10 Conclusion 55

Appendix 57

A Discord bot event list: ... 57

B ChatGPT output ... 62

C Pysolfc Code ... 63

1 Introduction

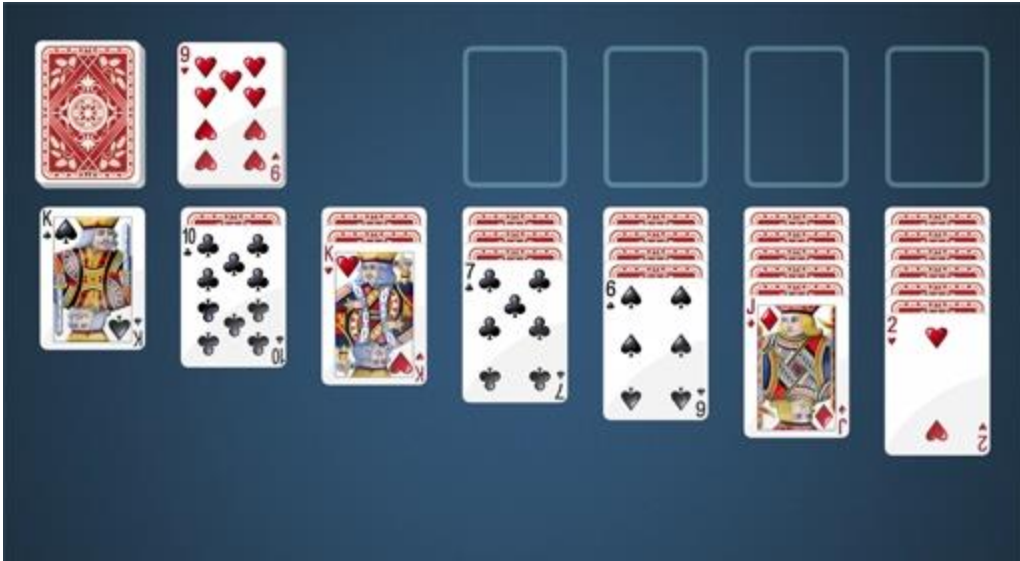
This project aims to finalize the work produced by a series of Major Qualifying Projects (MQPs) advised by George Heineman over the past few years.

- Synthesizing Mobile Games with Combinators by Joseph Blackman, Dean Kiourtsis, and Bailey Sheridan March 17, 2017 (E-project-031717-150219, https://digital.wpi.edu/show/41687j925)
    - This team set out to reduce the amount of time and effort required to use the Ionic framework to create mobile puzzle games. They used combinators to synthesize several games and produced a library of Ionic components to expedite the Ionic mobile game creation process.
- Applying Combinatory Logic Synthesis to work with Existing Software Frameworks by Jacob Bortell October 17, 2017 (E-project-101617-214733, https://digital.wpi.edu/show/bk128c69g)
    - This project demonstrates an application of combinatory logic synthesis to create the solitaire variation Archway. The concept, design and development of the program are explained as well as how to apply this to create multiple different variations.
- Generating Solitaire Games by Drew Ciccarelli, Ian MacGregor, and Simon Redding March 1, 2019 (E-project-030119-174649, https://digital.wpi.edu/concern/student_works/k0698907m)

    - This project is an exploration of implementing CLS in the existing software platform Next Gen Solitaire created by George Heineman. The team modeled thirty-four solitaire games from which they generated over 22,000 lines of working Java code.
- Next-Gen Solitaire Tutorials by Daniel Duff and Andrew Levy March 24, 2020 (E-project-032420-185648, https://digital.wpi.edu/show/zp38wg01d)
    - This project gives a complete tutorial on how to create and test a new Solitaire variation using the Next Gen Solitaire framework.

George Heineman, Jan Bessais, and Boris Düdder developed the Next-Gen Solitaire code generation framework in 2019. This framework utilizes Combinatory Logic Synthesis to generate Java or Python code for a solitaire game based on a Scala model that represents its design and rules. The framework includes numerous pre-built solitaire components and has been in development for several years. Moreover, developers can create customized logic to handle game rules or elements beyond the scope of the framework.

Solitaire is a card game typically played by one person, hence the name "solitaire." These games use a standard deck of cards, and the objective is to sort the cards into a specific order or pattern according to predefined rules. There are many variations of solitaire, but the most common and well-known version is known as "Klondike" or "Patience." A standard setup of this version is shown below.



In Klondike, the player starts with a shuffled deck of 52 cards and attempts to move all of them into four foundation piles, one for each suit, in ascending order from Ace to King. The stock pile contains the deck of cards that have yet to be played, and the waste pile is where cards from the stock pile are dealt and are available for play. The tableau is where most of the action takes place and where the player can move cards around to achieve the goal of the game. The tableau consists of several piles of cards, each with the top card face-up and the rest face-down. The player can move cards between the piles according to specific rules, such as only being able to move cards of alternating colors and in descending order. Empty tableau piles can be filled with a King or a stack of cards that starts with a King. The game is won when all cards have been moved to the foundation piles in the correct order. If the player cannot make any more moves, they can draw cards from the stock pile and continue playing. If the stock pile runs out, the game is over, and the player loses.

Each game of solitaire has its own specific rules and objectives, but the general goal is to move all the cards to the foundation, following the specific rules of the game. Winning the game requires strategy, skill, and a bit of luck.

In this project., we identified the following goals:

1. Complete the Solitaire code generation framework with a collection of solitaire games generated in Java and a few in Python
2. Incorporate testing strategies to generate test cases and automate their execution
3. Integrate continuous integration/continuous development into project using Travis[1]
4. Demonstrate the capability of the underlying approach within a different domain, specifically Discord[2] bots.

To evaluate the success of our efforts we expect to have the following deliverables at the end of our time with this project:

- A fully playable Solitaire application with all the working generated Solitaire variants.
- Tests cases with eighty percent code coverage
- Continuous Integration and Deployment through the Travis CI/CD framework
- Chat bots powered by Discord for the player to use.

2 Background

This project was in part a continuation of previous efforts by students to create a Combinatory Logic System (CLS) while also contributing new avenues in which to further explore code generation. Combinatory Logic is a formal system of mathematical logic that studies the properties of functions and their composition (Bessai et al., 2014). It's a type of formal system that doesn't rely on variables, but instead, relies solely on the composition of functions to build new functions. In other words, it's a system that allows you to build complex expressions by combining simpler expressions in specific ways. The basic idea is to use these

[1] https://travis-ci.com

[2] https://discord.com

small set of basic functions, called *combinators*, to define all other functions in the system. The combinators are simple functions that take one or more arguments and return a result. These can be combined in several ways to form more complex functions. The beauty of combinatory logic is that, because it only uses functions and their composition, it is completely free of any variables. This makes the system much simpler and more elegant than other formal systems that rely on variables, such as first-order logic.

The current code exists in an IntelliJ source project built using the SBT build tool. The project also incorporates the TWIRL[3] play framework that uses templates to synthesize code more productively. The Next Gen Solitaire repository includes three main parts being the src or source folder, the test folder and the generated folder. From within the src/main/scala/org.combinators/solitaire folder you can find all the source code for generating all the different variations for the game. To generate a variation, go to the folder and find the .scala file with the name of the variation. In this file you can then run one of the main objects which will generate and store code in the generated folder. From there you can run the generated java file to play the game. The standalone.jar file is a self-contained Java executable that contains all the code necessary to run a Java-based solitaire game. It includes the game logic, user interface, and all necessary libraries and resources, so it can be run on any system with a compatible version of Java installed. When the standalone JAR file is executed, it initializes the game logic and user interface, and presents the user with a playable version of the solitaire game. The user can then interact with the game using the mouse and keyboard, and the game state is updated and displayed in real-time. Once the game is completed, the user can start a new game or exit the program.

---

[3] https://github.com/playframework/twirl

This project also aimed to expand the code base using the PySolFC library. PySolFC is a free, open-source collection of more than 1000 solitaire games, implemented in Python. It includes a graphical user interface that allows users to play the games and customize various aspects of the game's appearance and behavior. PySolFC uses Python 2.7 code, which is an older version of Python that is no longer supported by the Python Software Foundation. However, the PySolFC developers have created a library for Python 3 that provides compatibility with many of the game variations.

In the first weeks of the project, our team assessed the state of the system and identified problems within the existing code repository, as hosted by https://github.com/combinators/nextgen-solitaire. Of the 45 total solitaire variations we were tasked to complete, 9 did not compile and/or run properly while a further 13 variations had defective code due to rule implementations or modeling that did not affect the compilation of the code but did affect the functionality of the resulting game. Over all we found close to 49% of the variants that needed additional work in order to bring the project to a state of completion. More problems were found during continued progress that required attention. Very few test cases were implemented, especially few for those implemented correctly, with a low percentage of code coverage. The project did not have a working continuous integration and continuous deployment system in place which became a goal of ours to add. A solitaire player was previously created for the project however it had not been tested or used in a significant amount of time.

3 Goals

After identifying the state of the system and assessing problems that needed to be addressed, large goals were set to focus our efforts and identify what we would like to accomplish within this project. Our goals were broken down into four main avenues that were each to be worked on concurrently. Our first avenue in which we chose to measure our success

was completion of the code base, taking previous students and professors' efforts and advancing them to a more finalized state. Second, we wished to implement a CICD framework, Travis was chosen for this avenue and significant work would need to be done to implement this system. Thirdly we needed to ensure the games could be easily played by a less code savvy user through fixing and finalizing the solitaire player application. Finally we chose to expand the project into a new realm by adding the functionality to create discord bots in python, utilizing the same CLS frameworks present within the solitaire project.

To achieve our goal of completing the previous project there were several tasks that needed to be completed. Our first task was to complete all the previously implemented variants, ensuring every variation compiles and runs a working version of solitaire designated by the rules of that variant. This would show to be the most challenging portion of our goals, occupying the majority of time due to the steep learning curve of understanding the system and how combinatory logic system's work in general. Our second task was for each member of the team to implement a new variation of solitaire each. This task would not only expand the existing project but would also show our teams' understanding of the system. We also wished to implement extensive testing of each variation to ensure the final games ran correctly and any changes could be tested and verified. We planned on getting one test per move for every variation, with the stretch goal of getting 80% code coverage. To complete our second goal, we had to implement a CICD framework. A CICD framework is a system that allows for continuous integration, development and deployment automatically, allowing for a streamlined process from implementing changes to deploying solutions. Our system would work by using Travis CI, a system designed to allow for this idea of CICD by executing commands on any changes made to the open-source repository. In practice we hope to implement a system that will compile, run, and test all variations whenever a change is pushed to the repository. The system will then

identify any problems within the system, whether that is a problem with the compilation of the code, any faults during runtime, or any failed test cases and email the problems to an inbox of our choosing.

To make this project more user friendly, we added a goal to improve and fix the solitaire player. The solitaire player is intended to be a stand-alone application for running any solitaire variation without the use of code editors or terminal commands. The solitaire player had been depreciating over time as efforts were put into improving the main repository while the solitaire player was untouched for many years. Through this goal we wished to fix any depreciated problems within the solitaire player and ensure its continued use and functionality as we look at completing our own project.

Our final goal of the project was to expand the scope of the combinatory logic system into a new environment of our choosing thereby better understanding the system and contributing to the overall usability and value of the system. Through expanding the project into a new scope we are able to use our findings to form a case study on how to take existing code and turn it into combinatory log systems. In exploring options to expand the scope we focused on identifying real world systems that people use that include many commonalities and portions of code that could be abstracted out into templates and modeling options. We came up with four options for avenues of exploration: Amazon Web Services (AWS), chat bots used on sites such as discord and twitch, website templates, and database systems. In narrowing down our options we eventually chose discord chat bots for our final expansion of the project. This was chosen due to the repetition of code between various different systems as well as the language availability of python, a language in which the project had already implemented some solitaire variations.

4 CLS Solitaire Code Generation

In this section we describe the various different parts that create the model for each solitaire variation. Deciding what goes into a model is a crucial step in creating a CLS. Within our model we outline various parts of a solitaire game that can be declared, modified, or dropped entirely. For this section we will use the solitaire variant "Narcotic" as an example to highlight how these parts come together to create a working variant.

4.1 Solitaire Model

A Solitaire Model is native Scala code that contains the necessary information to fully represent the structure and layout of a solitaire variation together with its rules that govern valid moves in the game. The structure of a solitaire model is shown below.

```
val variation:Solitaire = {

    Solitaire(name="…",
      structure = …,
      layout = …,
      deal = …,
      specializedElements = …,
      moves = …,
      logic = …,
      customizedSetup = …,
      autoMoves = …,    // OPTIONAL
      solvable = …      // OPTIONAL

    )
  }
```

Each of the designated attributes store the information needed for CLS to generate the required files to initialize endless solitaire variants with various different rules in a standardized way. Some of these variables are self-explanatory such as name being a simple string that refers to the name of such solitaire variant, solvable is a Boolean that when set to true will add a solve button that attempts to auto-solve the game based on the system's pre-defined solving algorithm. We now describe the remaining attributes in the upcoming sub-sections.

4.2 Structure

Structure declares the solitaire elements required for a specific variant. A typical solitaire game will include a Tableau and a Foundation. If all cards are dealt before the beginning of play, then there is no visible Stock; however, if cards are not fully dealt then the stock container will contain the remaining cards. An example of a structure map from the Narcotic variant is shown below.

```
val numTableau:Int = 4

val map:Map[ContainerType,Seq[Element]] = Map(
  Tableau -> Seq.fill[Element](numTableau)(Pile),
  StockContainer -> Seq(Stock())
)
```

Within this structure map we can see multiple ways of mapping elements to the structure of this variant. Within the first row we initialize the Tableau to be a sequence of Pile objects (in this case 4 of them base on numTableau. The second row declares there is a stock that contains a single deck of cards.

There is a rich collection of Element classes that one can select to define the structure of a variation:

- Pile – a collection of cards of which only the topmost card is visible.
- Column – a collection of cards which overlap to be able to see multiple cards at the same time.
- BuildablePile – a collection of cards which may initially have a number of cards face down (with a small overlap to be able to see the total number) on top of which a sequence of face up cards are visible with an overlap to be able to see multiple cards at once.
- Row – a collection of cards which would present horizontally and which overlap to be able to see multiple cards at the same time.

The system supports the ability to define new elements as needed for different variations, such as a kind of Pile on which the cards would be built from King down to Ace.

4.3 Layout

Layout declares how the structure will be placed within the graphical interface. Any element listed within the structure Map must have a corresponding placement within the layout map. An example of a layout map is shown below.

```
def stockTableauPileLayout(numTableau:Int):Layout = {
  Layout(Map(
    StockContainer ->
        horizontalPlacement(15, 20, 1, card_height),
    Tableau ->
        horizontalPlacement(120, 20, numTableau, card_height)))
     }
```

This is an example of a *stock tableau layout* meaning it is the base layout for many solitaire variations. Within this layout map we can see initializing locations of elements with the use of the horizontalPlacement function. The horizontalPlacement function takes 4 parameters that specify the x and y coordinates that will correspond to the top left corner of said element within the graphical user interface. The next variable is the number of elements to initialize which should correspond to the number of elements initialized within the structureMap. The Narcotic variant uses this stock layout in the initialization of its layout by setting the value of layout=stockTableauPileLayout(4).

There are different Placement alternatives one can choose:

- VerticalPlacement – orients a collection of Elements vertically rather than horizontally.
- CalculatedPlacement – occasionally the layout specification for a solitaire variation is so unique one has to resort to individually placing different elements on a virtual table. In this strategy, a sequence of (x, y) coordinates specifies where individual elements are to be placed on the screen. For example, placing four Foundation piles in a vertical

arrangement with 100 pixel spacing between them, you could use

```
calculatedPlacement(Seq( (512,100), (512,200), (512,300), (512,400),
card_height)
```

To make it easy to work out pixel arrangements, a helper `expand()` function is provided that uses card width and height as relative dimensions to lay out cards. For example, the sequence `calculatedPlacement(expand(Seq( (2,23), (5,23), (2, 27), (5, 27))` would place for cards in a box-like layout, with the first card two card widths to the right and 23 card heights down (and so on). The layoutMap must ensure that none of the elements overlap in placement, otherwise the generated code will generate a runtime exception.

4.4 Deal

Each solitaire variation specifies the initial sequence of cards that are dealt before play begins. There is a distinction between the initial deal triggered once at the start of the game and a deal move which can be requested by the player during game play. This initial deal does not affect how cards are dealt during normal course of play but will instead set up the board during initial runtime. An example of a deal implementation is shown below.

A deal attribute defines a sequence of DealSteps to be performed to initialize the board state. A `DealStep` has an target element which specifies where to deal the cards and a Payload that can have the option parameters `faceup` which if set to false (i.e., deals the cards face down so the type of card is unknown) or left as the default true value (i.e., cards are dealt face up), and `numCards` which specifies how many cards are added (the default is one). The element target can be simply a container element, such as Tableau, in which case the specification is that the payload is dealt to each element in the Tableau. For example, if the Tableau contained four Pile elements, then the following will deal one card to each of the four Piles in the Tableau.

```
Seq(DealStep(ContainerTarget(Tableau)))
```

The sequence of DealStep can contain various more fine-grained specifications:

- MapStep – deal cards to a target contained by mapping either the card's suit or rank to the target. For example, the **archway** solitaire variation uses two decks and has eight Foundation Piles – four that are built Ace up to King and four that are built King down to Ace – four Tableau piles, and thirteen reserve Piles, into which cards of specific ranks are dealt. In the initial deal, four aces are dealt to the Aces Up piles, and four kings are dealt to the Kings Down piles. 48 cards are then dealt to the Tableau piles (12 to each). Finally, the remaining 104 – 4 – 48 = 52 cards are distributed to the thirteen reserve piles, with all the Aces going to Pile 1, the Twos going to Pile 2 and so un until the remaining Kings are dealt to Pile3. This distribution of cards to these reserve Piles is captured by the specification `MapStep(Reserve, Payload(numCards=52), MapByRank)`
- FilterStep – extract cards from the deck that need to be dealt first to specific elements on the board. For example, to deal the four Aces to the Foundation, add the following step to your deal sequence – `FilterStep(IsRank(DealComponents, 1), 1)` – which will move the Aces to the top of the stock; then follow with a standard `DealStep(ContainerTarget(Foundation))` that sends one card each to the Foundation piles

The following specification contains the dealing sequence for FreeCell, which must deal 8 cards to the first four columns and 7 cards to the remaining four columns.

```
def getDeal: Seq[DealStep] = {
  var deal:Seq[DealStep] = Seq()
  var colNum = 0
     // Deal cards to all columns (from left to right) until
     // none left, in which case only first four have extra cards
  var numDealt = 0
  while (numDealt < 52*numDecks) {
    deal = deal :+ DealStep(ElementTarget(Tableau, colNum), Payload())
    colNum += 1
    numDealt += 1
```

```
      if (colNum > 7) { colNum = 0}
    }
    deal
  }
```

The **narcotic** variant has simple deal logic as it only deals out the cards to the four tableaus using `Seq(DealStep(ContainerTarget(Tableau)))`.

4.5 Specialized Elements

One can add variant-specific elements that are not present within the default generation. Most variants initialize this value to Seq.empty(), specifying that no specialized elements are present while many other variations specify elements to be added. The **narcotic** variant we have been analyzing has no specialized elements and therefore sets the value to Seq.empty(). One variant that does use specialized elements is the **freecell** variant. For this variant, the specialized element has to do with the types of piles on the tableaus. A FreeCellPile is one where you can only view one card at a time, and these will contain reserve cards set aside during game play. These must be distinguished from the regular Pile elements which belong to the Foundation, where cards are stacked from Ace up to King.

```
specializedElements = Seq(FreeCellPile),
```

Here a FreeCellPile extends the Element class shown below. This can be used to create variation specific element sub types that determine the view of the cards.

```
abstract class Element(
            val viewOneAtATime: Boolean,
            val verticalOrientation:Boolean = true,
            val modelMethods:Seq[BodyDeclaration[_]] = Seq.empty,
            val viewMethods:Seq[BodyDeclaration[_]] = Seq.empty,
            val modelImports:Seq[ImportDeclaration] = Seq.empty,
            val viewImports:Seq[ImportDeclaration] = Seq.empty)
```

Here a FreeCellPile extends the Element class and can then have specialized logic attached to it, using Combinator Logic Synthesis.

4.6 Moves

The most complicated part of the solitaire model are the moves that specify every way in which the player can move cards while playing the game. The moves variable is defined as a sequence of moves. Moves can be found within the org.combinators.solitaire.domain package file with the most common being SingleCardMove, MultipleCardsMove, and DealDeckMove.

Each Move is constructed with four parameters:

- Name – defines the type of move
- Gesture – defines the user interaction that realizes the move during game play, which is either `Press`, `Drag` or `Click`
- Source – defines the source element for the move and the logical condition under which the move is allowed to begin
- Target – defines the target element for the move and the logical condition under which the move is allowed to complete

The most common move type in Solitaire is a Drag move which is initiated by a mouse press on source element, followed by dragging events, to be completed by a mouse release on a target element. The logical condition is specified using Constraints, a powerful feature of nextgen-solitaire where a registry of built-in constraints (and user defined ones can be added as well) can be combined using logical operators (such as **if**, **and**, **or**, and **not**) to specify the proper logical condition.

An example of the moves available in the Narcotic variant is shown below.

```
val sameRank =  SameRank(MovingCard, TopCardOf(Destination))
val toLeftOf =  ToLeftOf(Destination, Source)
val tt_move = AndConstraint(sameRank, toLeftOf)
val tableauToTableauMove =
   SingleCardMove("MoveCard", Drag,
     source=(Tableau,Truth), target=Some((Tableau, tt_move)))

val tableauRemove=
```

```
    RemoveMultipleCardsMove("RemoveAllCards", Press,
      source=(Tableau, AllSameRank(Tableau)), target=None)

  val deck_move = NotConstraint(IsEmpty(Source))
  val deckDealMove =
    DealDeckMove("DealDeck", 1,
   source=(StockContainer, deck_move),
   target=Some((Tableau, Truth)))

  val deckReset = ResetDeckMove("ResetDeck",
    source=(StockContainer,IsEmpty(Source)),
    target=Some((Tableau, Truth)))
```

This move shows the implementation of a SinglecardMove in which the action is triggered on a Drag with the source of a Tableau and target of another Tableau with the constraints listed within allowed. It also shows an example of RemoveMultipleCardsMove, a DealDeackMove, and a ResetDeckMove. Moves can be initiated by Drag, Click, or Press. Targets and sources can be any element within the model. Constraints can be any type of restriction making use of the AndConstraint and OrConstraint composition functions, Destination variable, Source variable, MovingCards variable and any number of other variables and functions available within the system.

4.7 Logic

Logic defines the BoardState that will correspond with the conditions for victory. Some examples of this include logic = BoardState(Map(Tableau -> 0)) which specifies that all cards must be removed from the Tableau in order to win the game. Another example is BoardState(Map(Foundation -> 104)) which tells the system that the game is won when all cards within a two deck game are moved to the Foundation. In the Narcotic variant the logic is set in a similar way using logic = BoardState(Map(Tableau -> 0, StockContainer -> 0)) which states the game is won when both the tableau and the stock container are empty.

4.8 Customized Setups

The variable `customizedSetup` is declared as a sequence of Setup objects that can be initialized to create test cases. Test cases are an optional recommended variable within models that can be initialized to Seq.empty() if no test cases are present but should always have at least one test per move. The Narcotic variant does not have a customized setup and therefore sets the value of customizedSetup = Seq.empty(). If we look at the **fan** variation, we can see the value of customizedSetup = Seq(TableauToEmptyFoundation, TableauToNextFoundation, TableauToEmptyTableau, TableauToNextTableau). The values set in this sequence can be found in the file variationPoints.scala. The example of TableauToEmptyFoundation is shown below.

```
case object TableauToEmptyFoundation extends Setup {
  val sourceElement = ElementInContainer(Tableau, 1)
  val targetElement = Some(ElementInContainer(Foundation, 2))

  val setup:Seq[SetupStep] = Seq(
    RemoveStep(sourceElement),
    RemoveStep(targetElement.get),
    MovingCardStep(CardCreate(Clubs, Ace))
  )
}
```

This example specifies that moving an Ace of Clubs from the Tableau[1] to the Foundation[2] is valid under the condition specified in setup. Specifically, given an initial deal of the game, this setup clears the source element and the target element (which must be accessed by targetElement.get since it is an optional attribute) before adding an Ace of clubs to be the moving card from sourceElement to targetElement.

Various step mechanics can be used within the test, RemoveStep removes cards from a given location, initializeStep will add a card to a given location, MovingCardsStep moves a sequence of cards from the variable sourceElement to the variable targetElement. By combining these steps any number of moves can be tested for functionality.

4.9 Completing the Model

These elements come together to form the basis for our model within the package.scala file of our variation. The value *narcotic* is shown below.

```
val narcotic:Solitaire = {

  Solitaire( name="Narcotic",
    structure = map,
    layout=stockTableauPileLayout(4),
    deal = Seq(DealStep(ContainerTarget(Tableau))),
    specializedElements = Seq.empty,
    moves = Seq(tableauToTableauMove,tableauRemove,deckDealMove,deckReset),
    logic = BoardState(Map(Tableau -> 0, StockContainer -> 0)),
    customizedSetup = Seq.empty
  )
}
```

Once all the elements are declared, the Combinatory Logic Synthesis will generate the solitaire variation in Java from this specification.

5 CLS Solitaire Generator in Java

The internal mechanism of nextgen-solitaire takes a Solitaire specification as described in the previous chapter and generates all the code necessary for a full Java implementation using a Java Solitaire framework provided by Professor Heineman. This robust framework was developed over a decade of use within a software engineering class and has a well-defined API and modeling classes.

To initiate the generation of the **narcotic** variation, one starts with an object that extends DefaultMain and a special trait that knows how to construct the Gamma repository, which is the starting point for synthesis. While the code below is complicated boilerplate, it only needs to be written once for every solitaire family. The key idea is that from the repository, CLS is able to process queries that seek specific targets.

```
trait NarcoticT extends SolitaireSolution {
  lazy val repository = new gameDomain(solitaire) with controllers {}
  lazy val Gamma = repository.init(ReflectedRepository(repository,
          classLoader = this.getClass.getClassLoader), solitaire)
```

```
  lazy val combinatorComponents = Gamma.combinatorComponents
  lazy val targets: Seq[Constructor] = Synthesizer.allTargets(solitaire)

  lazy val results: Results =
    EmptyInhabitationBatchJobResults(Gamma)
        .addJobs[CompilationUnit](targets).compute()
}

object NarcoticMain extends DefaultMain with NarcoticT {
  override lazy val solitaire = narcotic
}
```

These targets are defined by the `Synthesizer` object through its `allTargets(s)` method that computes the desired targets to hand off to the CLS engine. The definition of this function appears below:

```
def allTargets(model:Solitaire): Seq[Constructor] = {
      Seq(game(complete), constraints(complete)) ++
      computeControllersFromDomain(model) ++
      computeSpecialClasses(model) ++
      computeMovesFromDomain(model) ++
      generateTestCases(model)
        .distinct
  }
```

This function returns a sequence of design targets, each of which is required for the final result. In a way, this function reads like a specification of the final project. The first Constructors in the sequence are `game(complete)` and `constraints(complete)`. These two terms match existing combinator in the Scala code base. The CLS will attempt to locate combinators that match these terms (and all the others as listed in `allTargets`). The combinator that matches game(complete) is shown below. Each combinator has two parts. First it has a def apply(…) method which will produce the actual code artifact – in the example below, it is a full `CompilationUnit` which represents a Java class. The second part of the combinator is its `semanticType`. These two sections parallel each other, with regards to the number of parameters in the apply() method and the number of chained semantic types (using the =>: Scala operator).

```
class MainGame(sol:Solitaire) {
```

```
  def apply(rootPackage: Name, nameParameter: SimpleName,
            extraImports: Seq[ImportDeclaration],
            extraFields: Seq[FieldDeclaration],
            extraMethods: Seq[MethodDeclaration],
            initializeSteps: Seq[Statement],
            winParameter: Seq[Statement]): CompilationUnit = {

    val comp = shared.java.GameTemplate
      .render(rootPackage = rootPackage,
        nameParameter = nameParameter,
        winParameter = winParameter,
        initializeSteps = initializeSteps)
      .compilationUnit()

    // add extra imports, fields and methods
    val clazz:TypeDeclaration[_] = comp.getTypes.get(0)

    extraImports.foreach { i => comp.addImport(i) }
    extraMethods.foreach { m => clazz.addMember(m) }
    extraFields.foreach { f => clazz.addMember(f) }

    comp
  }

  val semanticType: Type =
    packageName =>: variationName =>:
      game(game.imports) =>:
      game(game.fields) =>:
      game(game.methods)=>:
      game(initialized) =>:
      game(game.winCondition) =>:
      game(complete)
}
```

In essence, when CLS is queried to locate the semantic type `game(complete)`, it finds this combinator and issues a number of sub-queries based on these chained semanticTypes. For example, the `game(game.winCondition)` must be generated first, and it will correspond to a sequence of Java statements.

The combinators in this code base form the basis for generating all of the necessary classes to conform to the standalone.jar Java API. These combinators were developed over the course of many months and they encapsulate all of the design knowledge needed for writing Java solitaire games using the standalone API. The benefit of this approach is that once written, the

user only needs to provide a different Solitaire model, as described in the prior chapter, and the engine will generate the necessary Java classes to implement the variation.

5.1 Adding Combinators

Through the use of supplemental code we are able to change the generated code by directly rewriting what the model is meant to output in order to utilize a new feature or process that we wish to implement.

One method for adding supplemental code is to modify controllers through the use of supplemental combinators. This process works by creating combinators that will work similar to typical combinators within the CLS however instead of generating new code, these combinators work by overriding code that exists due to other combinators with the supplemental code that the user specifies. The process works by first modifying the generated code to create the feature you wish to implement then creating a class within the controller containing all the new code you wish to include as well as any code from the model you wish to remain unchanged. We will look at how the RemoveAllCards move was implemented to occur directly after another column move is completed which is not typical for our current controller setup.

Our first Goal was to modify a Java controller as it was generated in order to create our desired outcome. Our move affected the buildPile controller, when looking at the generated output it is noted that the current controller is hard coded to only affect one move at a time.

```
Move m = new MoveColumn(sourceEntity, movingElement, toElement);
if (m.valid(theGame)) {
      m.doMove(theGame);
      theGame.pushMove(m);
}
```

After looking at the code we can modify it in order to add a check after a move is done to see if our RemoveAllCards move can be initiated.

```
Move m = new MoveColumn(sourceEntity, movingElement, toElement);
Move rm = new RemoveAllCards(toElement);
if (m.valid(theGame)) {
```

```
        m.doMove(theGame);
        theGame.pushMove(m);
        if(rm.valid(theGame)) {
              rm.doMove(theGame);
              theGame.pushMove(rm);
        }
}
```

This modified code above gives us the desired outcome of removing all cards from a stack if the stack meets all the RemoveAllCards criteria. Now that we have a working controller that follows our desired outcome, we can work to modify our controller combinator to generate this code without the need of human intervention after the fact. In order to add this additional combination into our model a class must be created within the controller.scala file. This class must contain a function definition named apply() which contains a sequence of statements that resemble our outputted Java code. We also require a value named semanticType which specifies where our code should be deposited.

```
class BuildablePileRelease() {
new*
def apply(): Seq[Statement] =
Java(
  s”””|{
        |Column movingElement = (Column) m.getModelELemento:
        |BuildablePile toElement = (BuildablePile) src.getModelElement():
        |// Get sourceWidget for card being dragged
        |Widget sourceWidget - theGame.getContainer().getDragSourceO:
        |// Identify the source
        |BuildablePile sourceEntity = =
              (BuildablePile)sourceWidget.getHodelElementO:
        |// this is the actual move
        |Move m = new MoveColumn(sourceEntity, movingElement, toElement):
        |Move rm = new RemoveAllCards(toElement);
        |if (m.valid(theGame)) {
        |     m.doMove(theGame);
        |     theGame.pushMove(m):
        |     if(rm.valid(theGame)) {
              Rm.doMove(theGame);
              theGame,pushMove(rm);
        |    }
        |}
        |}”””.stripMargin
).statements()
}

val semanticType: Type = controller(buildablePile, controller.released)
```

```
}
```

The above code snippet shows the class BuildablePileRelease which results in the desired outcome discussed previously. We can see our new code snippet within the Java() function with the addition of some setup code taken from the model that otherwise would not be generated. The semanticType value is set to direct the combinator where to input the new supplemental code. In this case the code is set to be deposited within a controller, specifically the controller buildablePile and that this code will affect controller.released specifically. By setting the semantic type to these values we are able to overwrite code that is within the buildable pile controller and create new functionality when a buildable pile is released. The last step is to add the combinator to our model within the controller class.

```
if(s.name.equalsIgnoreCase("Baby")){
       updated = updated
              .addCombinator(new BuildablePileRelease())
}
```

We can use s.name to add the functionality to a specific model if we are working within a family that contains multiple solitaire variations. After the optional step of specifying which variant we are working with, the new combinator is then added through updated = updated.addCombinator(new SupplementalClass()). If all these steps are done properly then the combinator will be added to the model and all of our supplemental code found within the supplemental class' apply function will be directed into our generation, creating the desired functionality in the generated variation.

5.2 Abstraction

Abstraction is one of the key concepts that makes Combinatory Logic Systems (CLS) work. The idea behind CLS is to abstract common sections of code into the generator to reduce the amount of duplicate code fragments that have to be handwritten by human software engineers. Within this section we will take a look at how moves are abstracted within Solitaire to

create abstractions that can then be used by all, some, or future solitaire variants based on the need of specific variants as well as the process of creating new moves. By using languages such as Scala that can generate java code and seamlessly switch between Java code and Scala code functions, methods, and even whole classes can be abstracted away from the user providing access to functionalities without the need to write any significant code. The Combinator Logic System within our Solitaire application generates fully working Java code through the use of template files that are then filled in later by the Combinators. For example, by looking at the moves temple (moves.scala.java) we can start to understand how classes are generated by the CBL. The template has mostly standard looking Java code with the exception of @Java dispersed throughout the template. The @Java followed by a variable name allows the Combinatory Logic System to place unique variables at these locations. This allows the template to be used by multiple systems regardless of the required class name, function body, or additional methods required.

```
public class @Java(MoveName) extends Move {
      protected Stack destination;
      protected Stack source;
```

In this listing we can see that the class name will be replaced during code generation by the MoveName variable however the remainder of code will remain unchanged. This provides the functionality to create various move classes, all with different class names while still using the same template. The template file also adds the functionality of implementing new helper methods and/or constructors by including the line @Java(Helper). By allowing for helper methods to be imported, the system can be used in far greater ways, creating new functionalities within the model directly rather than going through the process of updating the system to create new templates and generators.

Generators are able to fill in Templates with the desired code during compilation in order to generate unique files with unique functionalities based on the model. Through investigating how a single card move is generated we can see how a template is turned into a full java move class. The move is first declared within solitaire.scala (case object SingleCard extends MoveType) followed by declaration of various functionalities within constraints.scala.

```
case SingleCard =>
      Java(s"destination.add(movingCard);").statements()
```

Within our doGenerator we can see a case that deals with the SingleCard move. This Java snippet of Java code written in scala can then be used by the doGenerator to replace the @Java(Do) section of our template in order to complete the method doMove within the generated class associated with a single card move.

```
public boolean doMove(Solitaire game) {
      if (!valid (game)) { return false; }

      @Java(Do)
      return true;
}
```

We can see in the code above how all of the code from the template was added to the generated file with @Java(Do) being replaced by the code we provided within the doGenerator. The same process is then repeated for the undo variable within the undoGenerator. Helper methods are also able to be added within the helperGenerator with the caveat being that the entire method must be modeled within the generator. Again, we can see this in action within the SingleCard move.

```
case SingleCard =>
      val name = move.name
      Java(s"""  |Card movingCard;
                 |public $name(Stack from, Card card, Stack to) {
                 |      this(from, to);
                 |      this.movingCard = card;
            |}""".stripMargin).classBodyDeclarations().mkString("\n")
```

In the code above we can see a new constructor being added to SingleCard moves in which a movingCard variable is declared as a global variable followed by a unique constructor that differs from the template. By adding functionality to include helper methods, the functionality of our templates is greatly expanded allowing for unique methods, constructors with differing numbers of parameters and additional variables unique to specific moves.

```
// Extra fields methods and constructors brought in here
@Java(Helper)
// Extra fields, methods and constructors brought in here
Card movingCard;

public MoveCard(Stack from, Card card, Stack to) {
	this(from, to);
	this.movingCard = card;
}
```

The code above shows how additional variables and constructors can be added through the addition of the line @Java(Helper) within our template.

The final requirement to ensure our move is available to the Combinatory logic system is to declare the move within domain/package.scala, making the move available to all solitaire variants. A simple declaration of parameters and declaring what type of move you wish to generate allows for the move to then be generated by any model that uses the move.

```
def SingleCardMove(name:String, gesture:GestureType,
                   source:(ContainerType,Constraint),
                   target:Option[(ContainerType,Constraint)]):Move ={
    Move(name, SingleCard, gesture, movableElement=Card,
      source=source, target=target)
  }
```

The code above shows the declaration of the SingleCardMove, with all required parameters such as name,gesture,source,target and finally the move that we expect to be generated. Notice how the name variable is used within our helperGenerator and SingleCard is the case used within the generators to identify SingleCardMoves as the moves requiring such code to be generated.

```
val buildFoundationFromWaste:Move =
      SingleCardMove("BuildFoundationFromWaste", Drag,
      source=(Waste, Truth), target=Some((Foundation, wf_tgt)))
```

The code above shows how within the modeling of a solitaire variation a SingleCardMove can be easily added without the need to write significant code. Through the use of this abstracting out similarities we are able to make multiple moves based on a single template, and use those moves across a variety of solitaire variants.

Abstraction is a key concept within Combinatory Logic Systems, allowing for common functions, classes, methods, variables and more to be reused and recycled throughout various locations. By abstracting away nearly everything needed for the desired system (either Solitaire or Discord in our case) models are able to quickly be generated, creating new code bases far quicker and less prone to errors due to the limited code that a user has to create. Through our look at how singleCardMove works we can see how new moves can be added and a wide range of functionalities can be easily added without having to write sizable amounts of code. After a moderate amount of work is completed creating new functionalities, these functionalities can then be used within any number of models without adding any additional work.

6 CLS Code Generation in a Different Domain

Our team was responsible for selecting a different domain to explore for code generation. We chose three target domains to evaluate -- HTML-based websites, Lambda functions in Amazon Web Services (AWS), and chat bots on platforms such as discord and slack. We settled on exploring chat bots on discord because the team felt that a chat bot with a simple command and response system would be ideal for CLS modeling and the team’s experience with discord as a communication platform. A chat bot is a simple program that responds to user messages with pre-programmed responses based on the message content. Discord is an online chat platform originally designed to connect gamers to one another and offers features such as voice, text and

video calls as well as centralized “servers” which serve as hubs for users based on shared interests. Discord chat bots or simply Discord bots support different features as specified by the Discord Application Programming Interface (API). (https://discord.com/developers/docs/topics/community-resources) Some features include being able to log information such as message edits, permission changes and joining voice channels, being able to execute commands and playing audio in a voice channel. There are freely available coding libraries for creating discord bots in different programming languages. Examples of these libraries include: DSharpPlus for C#, DiscordGo for Go, Javacord for Java, Discordia for Lua, DiscordPHP for PHP, Discord.py for Python, discord.js for JavaScript, discordrb for Ruby and Serenity for Rust.[4] After evaluating the available options, the team opted to use Discord.py because it is one of the most widely used libraries and there was pre-existing structure in the model for generating Python programs for PySol FC.

Discord bots written using Discord.py react to events. The bot defines the events it listens for and respond to. Events include actions such as messages being sent, creation of new roles, and servers being available and unavailable. Using those events as triggers, the bot reacts based on what event occurred. For example, when waiting for a message to be sent, the bot can wait for specific trigger phrases so it can react differently based on what is being said by a user. One common trait of discord bots is to have a pre-defined prefix to which the bot responds, rather than responding randomly to messages that just happen to include a trigger phrase.

The initial process of exploring discord bots was difficult because the team lacked a proper vision for what the bot should do. Eventually the team settled on exploring basic features such as logging information, basic response commands, and creating embeds. Additional features

[4] https://discord.com/developers/docs/topics/community-resources

that were discussed to be explored included commands which allow the bot to play files through the voice channel or interacting with an external API.

The Discord.py library allows for direct definition of bot commands instead of relying on message sent events to check if the message matches a command. This change allowed us to better decompose our commands and fit a better structure. It also meant we would not need to generate one large function and could instead generate an individual function for each command. We also found some drawbacks of Discord.py, for example, there is no command that allows the bot to look up a specific text channel with just the ID, instead you need to hardcode the text channel by finding it through the guild object (Servers are labeled as guilds in the API). This makes certain things like logging a bit more difficult since you want a designated channel for logging information. To counteract this, the team created a command for designating a logging channel. Commands like this will likely be generated by default if the model includes any function that involves logging. As the team continues to research and develop discord bots, certain commands designated as "default" commands should be generated regardless of what is outlined in the model. Examples of this include commands for changing the command prefix, changing where the bot logs information (if logging is modeled) and functions for updating the bot or user's permissions with commands.

Once discord bots were established as the domain we wanted to explore and our initial exploration of discord bots was completed, the team moved to applying code generation to the domain. The process for building a code generation system for discord bots followed an iterative process decomposed into three steps: Distillation, Modeling, and Synthesis.

Distillation is segmenting the bot into specific actions such as a command or an event. Modeling is modeling the distilled segment so that CLS can define and recreate the code from the model. Synthesis is creating the CLS fragment that recreates the distilled event from the

model. This process is repeated until one can fully reproduce a working discord bot through a model.

To begin the distillation process, the team focused on basic single events and commands and worked to model them and reproduce them. The team initially looked at implementing two very basic commands as shown below.

```
@bot.command()
async def echo(ctx, *, message: str):
       await ctx.send(message)

@bot.command()
async def hello(ctx):
       await ctx.send('Hello!')
```

The first command is a simple echo command, the user invokes the command from within discord with a phrase and the bot repeats the phrase back to the user. The second command is a simple hello command where the user says hello and the bot replies with hello.

During modeling, any number of predefined coding fragments are identified. In this example, there are a few:

- Imports: Discord.py & commands
- Intents
- bot = commands.Bot()
- bot.run()

Things that need to be modeled:

- Basic information about the bot: command prefix, description
- Commands, what the command does, trigger for it etc
- Events: what event is being called
- The Bot's ID

To properly generate events and commands, the team needed to consider how to generate multiple of the same type of object from a model. The first implementation of this used a set of two arrays, one holding the event name and another holding the events contents. The program would loop through each array and generate an event based on the given event name and fill the generated event with the content provided.

The team selected the following five events as their initial targets for generation:

- on_ready(): Called when the discord bot successfully launches
- on_message(message): Called when a user sends a message
- on_message_edit(before, after) called when a user edits a message
- on_message_delete(message) called when a user deletes a message
- and on_voice_state_update(member, before, after) called when a user changes voice states (joins/leaves a voice channel).

These events were selected as the initial targets for generation because they are some of the most common events used with discord bots and would be the easiest to test if they generated properly.

To get the code to properly generate events, the team needed to implement Scala match cases in a similar use case to switch cases. The team is also using a similar system to implement command generation however without the need for a match case since commands do not need to fit specific definitions.

To evaluate the model and generation system, the team wanted to use pre-existing discord bots as a base for generation. The first discord bot the team attempted to replicate was a logging

bot written for the WPI discord server by Samuel Currid.[5] The team would use this bot as a guide to determine what events to generate next.

Once the team began examining the code for replication, the team realized there were several key factors that still needed to be modeled. These included adding the ability for the model to support general utility functions that are not commands or events, permission checks, and the ability to import other libraries.

One factor the team chose not to implement is *cogs*. Cogs are Python subclasses which help to organize command listeners. Cogs are an important structural tool for discord bots. They allow for bots to dynamically alter their functionality by adding and removing commands during run-time. When you unload a cog it unloads the file, if you want to make a change to the code, you can unload the cog, make changes to that cog and reload it to the bot without shutting the bot down. This would also allow moderators to disable certain groups of commands when needed. Implementing cog generation would require a separate model from the main one due to their structural difference from a standard discord bot. Allowing for a bot to load pre-existing cogs using the main model would only require minor alterations. Cogs similar to how the model was designed are a form of code distillation, they allow for each command group to be compartmentalized and focused on one specific aspect of the bot's functionality. They also allow for code mobility since cogs can easily be loaded into different bots which allow for modular code design.

The first simple discord bot the team had selected was technically able to be replicated using the current system, however upon replication, the team realized that the bot was old and used deprecated commands which Discord.py no longer recognized. Moving from this, the team located another bot, this time a very basic bot template written earlier this year. Reading over the

[5] https://github.com/SamuelCurrid/Gompei-Bot/blob/master/cogs/Logging.py

code, the team was still missing a few things that the bot wanted to have generated including subcommands and the ability to change what the bot was named. After making minor adjustments to the generated code however, the bot ran mostly correctly. The only error was caused by one command which requested image information from a website resulting in a JSON error. Ignoring this, the bot ran as intended and the only change needed to be made to the code was altering some variable names. [6]

Following this initial generation, the team decided it would be easiest to generate a bot written during the initial discord.py exploration. The bot was very simple but included enough commands to test all aspects of the model including importing new libraries, multiple events and commands and a proper description. Once the bot was properly modeled and the code was generated, the only difference between it and the original code was a difference in formatting. The bot was able to maintain all of its functionality after being modeled and generated.

Following this successful generation, the team felt it was important to generate another bot and continued searching for a bot that would be more amenable to modeling.

This bot was also able to be fully modeled with very minor alterations to the code. These were primarily changes to how the commands were defined, the original code has more in-depth command definitions which would allow the commands to have a different name than the one defined in the Python function. The team chose to not include this functionality in the model to keep the model simple.

From these generations the team was able to gain a great deal of insight into creating models for code generation as well as insight into discord bots' structure and functionality.

[6] https://github.com/aadityabhoyar/Python-Discord-Bot-Template

The team used these generations to test the model and understand what improvements could be made down the line. Currently the bot is able to generate: 21 events, chat commands, helper functions and a basic discord bot outline These improvements include: supporting cogs (could be through creating a cog specific model or allowing multiple files to be generated at once), increasing event coverage, Discord.py has over 80 events and will likely add more as discord updates their API and improving ease of use for the model by making it easier to use and understand.

[7]The core takeaway from creating this model was understanding the need to create a balance between functionality and simplicity. The model needs to be functional enough to be able to properly generate the code you need to create while simple enough for someone to easily pick up the model and use it for their own purposes. In the discord model created an example of this would be how commands are modeled. As outlined previously, discord.py allows for more complex definitions of commands including aliases and use restrictions. The team chose to omit these features from the first iteration of the model because it added unnecessary complexity to the model.

6.1 ChatGPT comparison

ChatGPT is an online artificial intelligence which can translate chat requests into outputs. The team used this platform to see the difference in code generation and compare the output of a similar bot generated previously by the team. ChatGPT was given the following prompt:

"Generate a discord bot using discord.py that has the following functionalities: Its default command prefix is ? but it has a command which allows the user to change the prefix A hello command that simply responds to the user saying Hello with a Hello back, Triggers on message

[7] https://github.com/HarshitSati/BasicDiscordBot

edits, sends the new and old message in the same channel the message was edited. Triggers on message deletion, sends the deleted message in the same channel that message was deleted.

The AI output (See Appendix B) was a mostly correct Discord.py file. However, without any alterations to the code, the bot did not run. Upon examination, the code generated was based on older versions of Discord.py which did not require intent decorations in bot definitions. Additionally, the code does not use the commands library and instead uses the on-message command to track user messages. While still functional, the command library helps separate into individual function declaration instead of being completely contained within the onMessage event.

In terms of comparing this generation to our own generation, the process was obviously much easier since no actual Python knowledge was needed to write this bot and instead the prompt only needed to know what library to use. Additionally for events like message edits, the proper name of the event was not needed since the AI understood which event was being referred to. The team thinks that using ChatGPT for individual command and event definitions is probably much better than using it for a whole discord bot since it is missing some of the needed structure. Additionally, using ChatGPT in tandem with the model would mean the structure would be generated and the content of the events and commands could be generated by chatGPT.

Overall, while ChatGPT is more powerful and easier to use than our model but may not be as reliable since it is based on reading many different discord bots which may look different due to discord.py changing over time.

7 PySolFC Code Generation

PySolFC is a collection of more than 1,000 solitaire games written in Python. It was originally developed by Markus F.X.J. Oberhumer and has since been maintained by an open-

source development community. PySolFC offers a wide range of solitaire games, including many familiar variants such as FreeCell and Klondike, as well as less well-known games like Bakers Dozen and Spider. The program is free and open-source, and is available for Linux, macOS, and Windows.

The nextgen-solitaire codebase contains several specialized generators that run as plug-ins for the PySolFC hub software. For B term 2022, we started our attempt to expand the codebase and have a proof-of-concept additional variant. This ended up being more troublesome than predicted, as several ambiguous errors came about. With time, we were able to narrow down the problem to solve the error.

This section of the codebase works by taking the same Scala solitaire domain object used to model for the Java code generation, and drawing from it to generate Python code. Internally, this was accomplished because of a Python generating engine provided by the original developers of nextgen-solitaire.

7.1 Adding Simple Simon

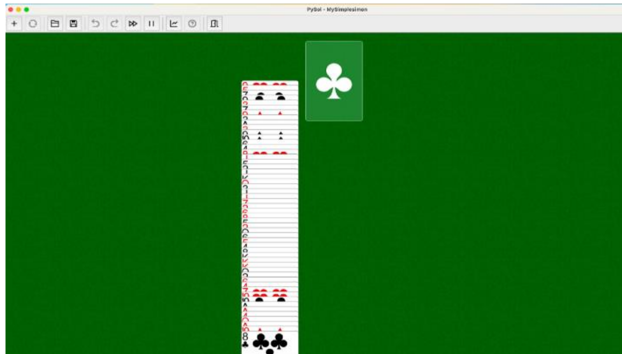
In an attempt to branch out the codebase of PySolfFC code generators, the team developed a new generator for the "simple" variant. This variant is a baseline proof-of-concept meant to simply make a functional variant. This variant has a tableau and a deck, and the player wins once they move all cards from the deck to the tableau, with no move stipulations. To branch out, the team used the "fan" PySolFC as a starting point to get better acquainted with the PySolFC system.

This process did take longer than intended, due to a deeply rooted InvocationError. After heavy digging, the core of the problem came from missing and commented out constraints in the Java model used to generate this Python code. For example, the BottomCardOf constraint was commented out in the former state of the codebase. Through this, the generator would fail as it

would try to draw non-existent logic from the Java model.

```
37  case class BottomCardOf(moveInfo:MoveInformation) extends MoveInformation {
38    override val isSingleCard:Boolean = true
39  }
```

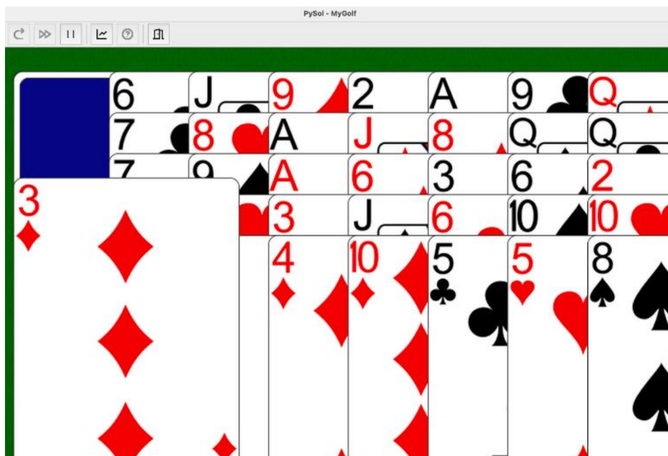
The resolution of this issue led to the first addition of this iteration to the PySol codebase. (Appendix C).



This addition to the codebase has added the variant "Simple Simon". In this variant, the player must move the cards from one pile over to another in order to win. It is a simplistic game that set the pace for development, established a proof of concept, and got our team acquainted with the PySolFC framework. From there, we could continue to expand.

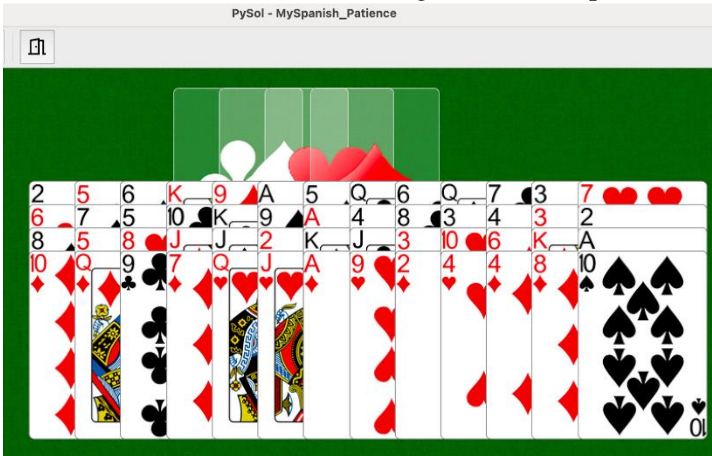
7.2 Adding Several Variants

From this spot, we added the next variant, Golf. This was again done by taking reference from the Simple Simon variant, and then adjusting it accordingly by giving the variant its own game ID, and other small adjustments.

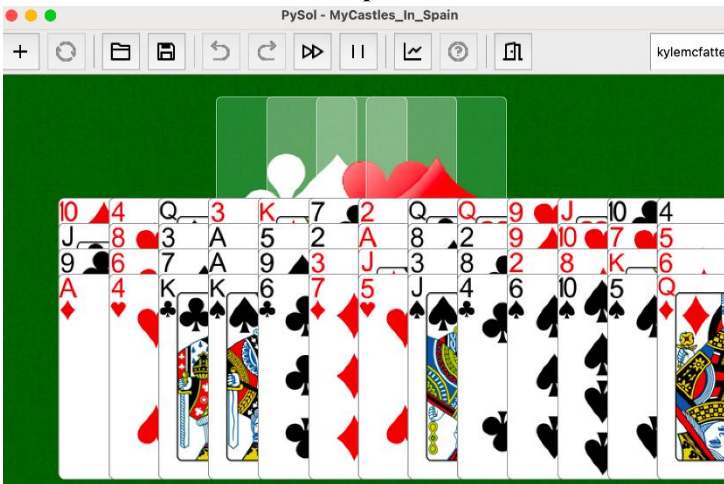


Through a similar process, we added Baker's Dozen.

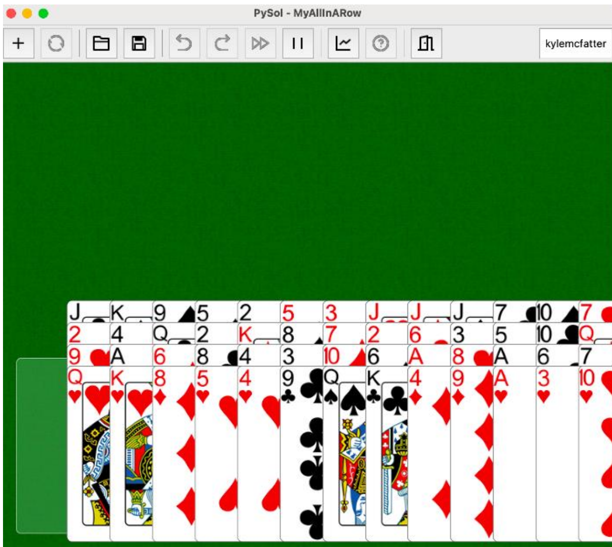
Once Baker's Dozen was working, we added Spanish Patience.



Then we added Castles in Spain.



Then All in a Row.



Then Flake.

Then Robert.

7.3 Adding Spider

We did run into a few issues through this process, for example, the Spider variant did not work due to a JavaInvocationError. This was due to a missing constraint in the generator library. The Scala object this was drawing from was attempting to use a constraint called "emptyPiles" to check if the piles in the game had no cards left in them. However, this constraint had not yet been added to the codebase. To solve this, we figured out what this constraint should entail in the game logic, and added it to the codebase.

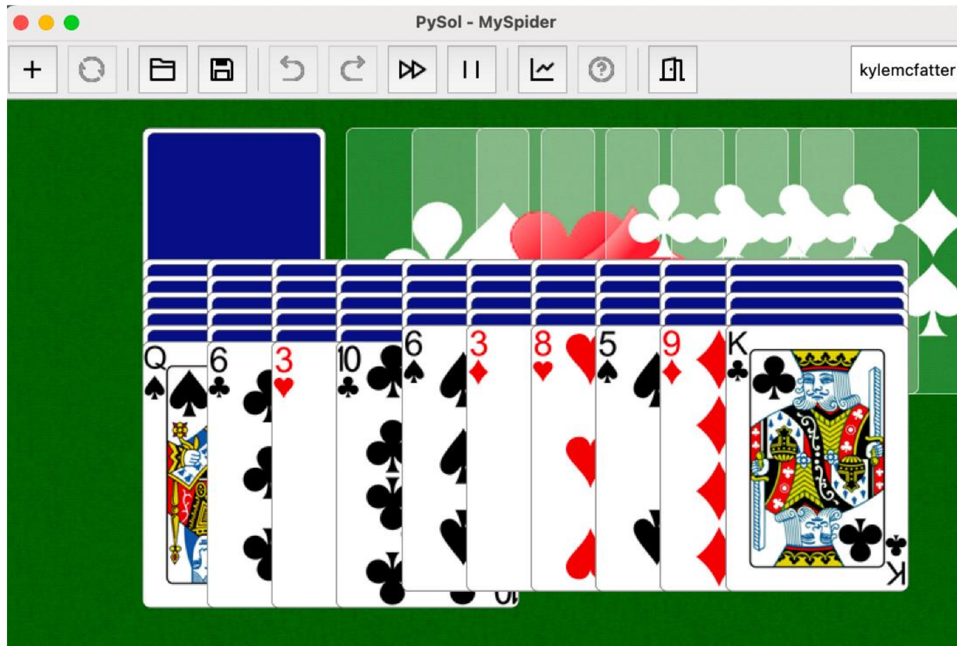
```
@combinator object HelperMethodsSpider {
  def apply: Python = {
    val helpers: Seq[Python] = Seq(generateHelper.tableau(),

      Python(
        s"""
           |def emptyPiles():
           |    for t in tableau():
           |        if len(t.cards) != 0:
           |            return False
           |    return True
           |""".stripMargin)
    )

    Python(helpers.mkString("\n"))
  }

  val semanticType: Type = constraints(constraints.methods)
}
```

With that, we had now added the Spider variant to the codebase.



8 Our team was able to add several new variants to the PySolFC section of the code base, and also catch errors in the existing codebase, such as missing, problematic, or commented-out constraints. By the end of this project, the variants available for PySolFC had more than doubled.

Travis

Travis [8] was added to the project to facilitate continuous integration and deployment. Continuous integration and deployment (CI/CD) is a process that allows for automated testing and deploying of final products anytime changes are made to the code base. In practice, CI/CD allows for changes to be made to the code base and those changes will immediately be followed by a report generated by the CI/CD system that runs all tests and deploys the changes if all testing parameters are passed. This change should streamline the development process by reducing the need for developer driven testing and allow engineers to spend more time improving the product as tests are run without need for human intervention.

[8] Travis CI: https://docs.travis-ci.com/

8.1 Testing Code Generation

To properly implement continuous integration and deployment we had to learn how to work with Travis. The first challenge to overcome was fixing the Travis files to allow for code generation within the newly minted system. Travis was previously operating in some capacity within an old system that generated the code through string concatenation, since the system has changed to use templates and Twirl the Travis files had deprecated and needed updating. After trying many different avenues for generating variants, three different approaches were found and considered. Calling the main methods directly by using direct references was ruled out due to the complex nature of using direct references in favor of a more user-friendly approach such as the two we will discuss later. This method could be improved by looping through the Scala classes folder to find all main methods, however due to this directory being located within the target folder this modification would require including at least part of the target folder within GitHub which was ruled out as an impractical method within this project. A second approach under consideration was to create a separate Scala file where all the main methods could be called with only one Travis command. This method was also ruled out due to the lack of data collection that Travis could provide with a single Travis command call. To find the correct process we would use to test our variants, we needed to find a system that would allow us to test every variant in a user-friendly way while still reporting back data on which variants specifically needed more work before deployment. For those reasons a simple variable was created to simplify the direct pathing problem by allowing users to call the variable $Generate with the corresponding family and variant names. This method allowed for a simple readable command structure that would return back direct informative data via an email report. In combination with a travis.txt file that lists all family/variant names, this option proved to be the most modular and user-friendly method.

8.2 Common issues:

Testing generated files comes with many problems with testing that is not present in static repositories. In order to run automated testing through a CI/CD pipeline the files must first be found, then moved to a testing directory one by one before tests can be executed to determine whether errors are present within the code base. Common problems that were run into include compilation errors within individual variants can cause other variants to also fail resulting in negatively skewed results. Another problem encountered within the process of setting up automatic CI/CD was that the generated variants originally presented themselves within a target folder that is part of the git ignore configurations and thus the files are invisible to Travis. A final problem is that Travis and many other continuous integration platforms run on a headless version of Java, meaning graphical interfaces will not work properly within Travis unless xvfb[9] service is added to Travis.

8.3 Testing Generated Variants

This section will discuss setting up options and configuration files pertaining to Travis CI, for information on setting up Travis itself, see the quick start guide at docs.travis-ci.com/user/tutorial/. The first step to setting up automated testing for a combinatory logic system is to ensure that you can generate your variant through commands specifically sbt for this project and that all relevant files will be placed within a directory included in the project structure. Within the solitaire project the command

```
sbt runMain org.combinator.solitaire.FAMILY.VARIANT
```

[9] Travis xvfb: https://docs.travis-ci.com/user/gui-and-headless-browsers/

will generate the code that corresponds to the solitaire variation with the name (VARIANT) and family (FAMILY), note that with this command and many others within Travis that quotation marks must be placed around all options for commands:

```
sbt “runMain org.combinator.solitaire.FAMILY.VARIANT”
```

This command can be made more readable by adding a bash variable that can be interpreted by Travis resulting in the same functionality. The variable GENERATE was created to facilitate this process and was placed as an environmental variable within Travis.

```
env:
    global:
                - GENERATE = “runmain org.combinators.solitaire”
```

Adding this environment variable to Travis allows the code to be more uniform and readable by changing our previous command into:

```
sbt “$GENERATE.FAMILY.VARIANT”
```

This command will generate the solitaire variation (VARIANT) of the family (FAMILY). We will use this command along with a bash for loop to generate all variants within the final Travis configuration. The next step to set up automated testing is to ensure that the generated variant can be tested through commands. This project is set up as a sbt project so we will use the sbt test command to facilitate our variation testing. First, we must move the variant into the src/test/java folder in order to present the files for sbt test to run. One problem that could occur to note is that the files must retain their package directories, in this case the files are automatically generated within directories org/combinators/solitaire so we must create those directories within the testing folder. This leads to our final path being src/test/java/org/combinators/solitaire and thus we will move our files from generated to this folder.

```
mv -f generated/src/main/java/org/combinators/solitaire/*
src/test/java/org/combinators/solitaire/;
```

This command will move all files from the generated/……/solitaire directory into the testing directory. By generating the variants individually and testing them individually we can move all files from our generated directory by using the * symbol. After this move is complete, we can simply run:

```
Sbt test
```

This command will run all java Junit tests and report out whether any tests failed or if all tests passed. We can use this command in combination with a bash if statement to move the directories into folders designated pass and fail based on whether or not any Junit tests failed.

```
If sbt test; then
mv src/test/java/org/combinators/solitaire/* generated/pass; else
mv src/test/java/org/combinators/solitaire/* generated/fail;
fi;
```

These commands will allow our generated variant that has been moved into the testing directory to run all Junit tests and then be placed within the proper directories allowing for the variant to be generated without any interference from compilation errors of previous variations. After the tests have been run, we can navigate to the corresponding folder and use the ls command to list out which variants passed all tests and which variants require more work.

- `cd generated/pass`
- `ls`
- `cd ../fail`
- `ls`

These commands will list out all files and directories within the pass and fail directories which should include the variants we have just tested and moved. One common problem to note is that the directories that pass and fail must be added to GitHub before the commands can be successfully run. This can be done by adding a temporary file such as temp.txt within each directory and pushing both files to GitHub, ensuring the directories exist within GitHub. The only problem left to tackle is to get all family names and variant names that can then be looped through, generated, tested, and report generated. We will discuss two options that were

considered and tested for this step. The first option was an automated version that would find all variants regardless of human input and run the corresponding tests. This option was far superior in ensuring no variants were missing from tests, however it came with a large storage cost as part of the target folder would need to be added to GitHub to ensure that Travis had access to such folders. This method works by going to the target/scala-2.12/classes/ and looping through all the class files to find methods that match the method signatures of variants.

```
cd target/scala-2.12/classes/
for i in `find org -name "*.class"`; do
        javap $i | grep DefaultMain >/dev/null;
        if [$? -eq 0]; then
        remover="org/combinators/solitaire/";
        i_without=${i#$remover};
        i_cleaned=${i_without%\$*};
        variant=${i_cleaned////.};
        echo $variant;
        cd ../../..;
        sbt "$GENERATE.$variant";
```

These commands loop through all class files and find all methods that extend Default Main and pass the test [$? -eq 0] this will leave us with a list of all methods that generate variants within our project. These variants will need some processing before our commands can use them reliably. The outputted classes will appear in the following format:

```
org/combinators/solitaire/castle/CastleMain$.class
```

To take this string and convert it to the format we need (FAMILY.VARIANT) we first must remove org/combinators/solitaire/ from the string. Note that this is not necessary if we modify our generate variable, however for simplicity's sake, we will process the string anyways.

```
i_without=${i$remover};
```

This command will remove the first part of the string leaving us with

```
castle/CastleMain$.class
```

Next, we will use bash to remove everything including and after the $ symbol. The bash command %/ will operate in this way:

```
i_cleaned = ${i_without%/$}
```

This will leave us with a string very close to the format we are looking for.

```
castle/CastleMain
```

Finally, we must change all "/" to "." in order to convert the string from a directory format to the format that sbt recognizes. By using the ////. Bash script we are able to replace all (//) slashes (/) with (/) periods (.). This will result in all the class strings being processed into a format that we can use and that we can loop through. Putting this all together we get our final script that loops through classes within the target folder, generates the variant, moves the variant, runs the test, and moves the variant to the corresponding folder.

The other method that was eventually implemented works in a simpler manner that is far less automated and also requires less storage. In this method a Travis.txt file was created that contains a list of all variants in the format we need. In order to implement this, we change the for loop into a while loop and read from the text file.

```
while read i; do
…
done < travis.txt
```

The only pre-processing that needs to be done with this method is to remove trailing newline characters from the strings.

```
variant = $(echo "$i" | tr -d '\n');
```

This line will remove extra new line characters and allow our configuration file to run all its necessary commands.

```
sbt "$GENERATE.$variant";
```

This command will now generate all the variants within the while loop. Allowing for a complete Travis configuration and producing a CI/CD pipeline that would run and test all variants as well as produce a report that is automatically emailed to the owner of the Travis CI account that lists which variants pass and which variants require more work. Additional

functionality can be added including making the pass folder a GitHub subdirectory and having Travis push to that directory after all tests are run, maintaining an automatic repository of valid variants that can then be run and used by other users or a solitaire player program which we will explore in the next section.

This travis.yml file can be generated by running the commands located within generatedTravis.txt. The first command is to compile the project so that the scala classes we wish to run have associated class files that we can query and eventually run.

```
Sbt compile
```

Next, we want to clear the travis.txt file so that we may repopulate the results:

```
> travis.txt
```

After the file is cleared and the classes are compile, we need to move our working directory to the location of the compiled class files.

```
Cd target/scala-2.12/classes
```

Once we are in the proper directory, we can run our for loop that will get all the family and variant names with the only change being that we will echo to travis.txt instead of outputting to std out. See above for full explanation of for loop.

```
Echo $variant >> travis.txt
```

At this point our file is populated with all variants that exist and should be generated then tested and we can move our working directory back to the project root and continue to run Travis or continue working.

8.4 Exporting Generated Variants

Once each variant has been generated, tested, and classified into valid and invalid directories the next step will be to export the generated code so that it can be used and run by external users and the solitaire player. Within this project we set up a separate GitHub repository

that will contain the generated java code separated by variant and sorted by Travis testing results. This is done through the commands in travis.yml:

Before script:

```
git fetch –unshallow
```

This command with fetch the repository from GitHub and ensure we are not in a shallow repository which could prevent some further functionality.

```
Git config –global user.name “USERNAME”;
```

This command will allow Travis to use a specified GitHub account to perform commands and contribute to repositories.

```
Git add Valid/ Invalid/;
Git commit -m”MESSAGE”;
```

This command will add the valid and invalid folders that Travis has populated before committing those folders to GitHub with a specified message.

```
git push - -force “https://${GH_TOKEN} @ LINKTOREPO”HEAD:main;
```

This command will push the commit to a specified repository provided by LINKTOREPO on the main branch, forcing all changes to go through by overriding any merge conflicted data with the data we provide. GH_TOKEN must be set to a GitHub personal access token; it is recommended that this is stored as an environment variable within the Travis website so that the key is not visible to the public. This can be set up by going to Travis, clicking the repository that you wish to include an access token, clicking more options, settings, and adding the environmental variable to this repository. This type of access token gives anyone who obtains the token full access to their GitHub account.

Within Travis you should ensure that the external account is not visible by Travis so that if a travis.yml file is ever pushed to the external account, Travis does not run this build which would result in an infinite loop of Travis testing. It also must be guaranteed that the GitHub

account associated with the command git config global user.name has access to both repositories and their personal access token is properly configured to give Travis access to the commands, git add, git commit, and git push. Within the separate repository you want to ensure that this repository exists, is empty, and has the specified “LINKTOREPO”.

9 Solitaire Player

The Next Gen Solitaire Player is an application written in Java that allows the user to download and play our generated solitaire variations. The user is able to fetch all of the available variants from GitHub, allowing us to continue to update the model with fixes and new variants. Each of the local variations appears in a scrollable list, from which a variation can be selected. Clicking the play game button will launch the highlighted variation in a separate window, ready to play.

The previous iteration of the solitaire player was built on JavaFX and was able to launch locally stored .jar files to play solitaire variations. Since the inception of this project, official support for JavaFX has been occluded by Swing, a similar UI creation tool that is now included in development environments such as IntelliJ. The new solitaire player is built in Swing and designed to meet three use cases; downloading a package of solitaire variants from GitHub, selecting a variation from a list, and launching the selected variant in another window. In order to meet these requirements, the player includes two buttons, a list, and a scrollable view panel. The list, formally a JList, is embedded in the scrollable view, or JPanelScrollable. While some of the attributes within the Swing UI are difficult to interact with, the JList object can easily be manipulated by referencing them in various methods, allowing the model to follow which list items are being interacted with.

To use the player, users first need to download the playable solitaire variants from GitHub. After Travis generates and validates all of the variants, they are hosted on a GitHub page. Clicking the “download from git” button in the player interface grabs a .zip file containing all of the variations, and places it in a subdirectory of the player. Then the file is unzipped using a ZipInputStream and iterating over each ZipEntry until all files in all subdirectories are extracted. This step is important, as the file structure needs to be preserved during extraction. All of the downloaded variants are then displayed to users as elements of the scrollable JList. Users can click on individual elements of the list, which will update the model to reflect which item is selected. Then, when the “launch game” button is pressed, the controller gets the selected variation from the model and finds the main function that corresponds to that variation. Using the ProcessBuilder class, the main function for the solitaire variant is run, opening the game in a separate window. The user can play the game for as long as they want, interacting only with the variant itself. Once they close the solitaire game, they will be returned to the solitaire player, where they can choose another variant to launch or close the player.

Downloading, unzipping, and running java classes from the internet is inherently an insecure operation to carry out. Early on in the process, we considered the use of a Java security manager. This is something that had been used in the older versions of the solitaire player, but unfortunately almost all recent versions of the Java Development Kit explicitly prevent the use of the security manager. Oracle itself found that there was no eloquent way to include the security manager in Java and disallowed its use many versions ago. Going all the way back to JDK 1.7 did allow us to use the security manager alongside the process builder to prevent modified classes from running--for example, if a malicious actor included a class that did not share the same overall structure of a solitaire variation class, it would be stopped from running. However, this solution was not only very clunky and limited, but it also required JDK 1.7. Upgrading even

to JDK 1.8 would not allow further use of the security manager. As a result, we decided to treat the solitaire player like we would a custom mod to Minecraft, a popular video game with a very strong community of custom mods and add-ons for the base game. Users who would use the solitaire player would need a certain level of existing knowledge to use it, similar to those who play Minecraft with mods, and in turn would take responsibility for the security of their system. Just like Minecraft mods, our variants are downloaded and executed on the user's system without the security checks that would be present on an officially supported game. This allowed us to make the solitaire player easier for the user to interact with, since both the downloading and executing of each variant is handled by the player, instead of requiring outside action by the user--similar to the TModLoader that can be used to download and manage custom mods for Terraria, another popular game with a healthy modding community. This also allowed us to make the solitaire player in base Java, without needing to use outdated JDKs or externally developed packages. In the end, the solitaire player only uses built-in packages developed by Oracle, such as the Swing UI editor, Java IO and NIO, and Java Util.

10 Conclusion

This project represents the culmination of several years of work by various Major Qualifying Project teams, all advised by Professor George Heineman, to develop and enhance the Next-Gen Solitaire code generation framework. The project aimed to complete the framework by generating Java and Python code for a collection of solitaire games and incorporating test cases with eighty percent code coverage. Continuous integration and development were added using the Travis CI/CD framework. This would allow for any changes to the code base to be tested and deployed automatically if the test cases pass. The project also demonstrated the capability of CLS code generation in a different domain by implementing

Discord bots. The team used Discord.py library and followed an iterative process of distillation, modeling, and synthesis to develop a code generation system for Discord bots. One deliverable from this project is a fully functional Solitaire player that has all the generated code stored in it to allow the user to have access to all the variations without going through the process of generating the code.

Combinatory Logic Synthesis for code generation has potential applications beyond Next-Gen Solitaire. It can be utilized in situations where multiple models have similar structures and functions but differ slightly, especially when memory efficiency is important. This approach is not recommended for one-time programs with simple objectives or non-hierarchical objects. For instance, game engines can benefit from this approach as they often have various levels, AI players, and items that share similarities but also have differences. Loading only the necessary elements for a particular level can significantly reduce load time and enhance performance. Moreover, code generation facilitates the creation of games in different programming languages, making it easier to port them to diverse platforms. We encourage others to explore the potential of CLS and Code Generation in their future projects.

Appendix

A Discord bot event list:

App Commands:

1. on_raw_app_command_permissions_update(payload)
   Called when application command permissions are updated.
2. on_app_command_completion(interaction, command)
   Called when a app_commands.Command or app_commands.ContextMenu has successfully completed without error.

AutoMod

3. on_automod_rule_create(rule)
   Called when a AutoModRule is created
4. on_automod_rule_update(rule)
   Called when a AutoModRule is updated.
5. on_automod_rule_delete(rule)
   Called when a AutoModRule is deleted.
6. on_automod_action(execution)
   Called when a AutoModAction is created/performed.

Channels:

7. discord.on_guild_channel_delete(channel)
8. discord.on_guild_channel_create(channel)
   Called whenever a guild channel is deleted or created.
9. on_guild_channel_update(before, after)
   Called whenever a guild channel is updated. e.g. changed name, topic, permissions.
10. on_guild_channel_pins_update(channel, last_pin)
    Called whenever a message is pinned or unpinned from a guild channel.
11. on_private_channel_update(before, after)
    Called whenever a private group DM is updated. e.g. changed name or topic.
12. on_private_channel_pins_update(channel, last_pin)
    Called whenever a message is pinned or unpinned from a private channel.
13. on_typing(channel, user, when)
    Called when someone begins typing a message.
14. on_raw_typing(payload)
    Called when someone begins typing a message. Unlike on_typing() this is called regardless of the channel and user being in the internal cache.

Connection:

15. on_connect()
    Called when the client has successfully connected to Discord. This is not the same as the client being fully prepared, see on_ready() for that.
16. on_disconnect()
    Called when the client has disconnected from Discord, or a connection attempt to Discord has failed. This could happen either through the internet being disconnected, explicit calls to close, or Discord terminating the connection one way or the other.
17. on_shard_connect(shard_id)|
    Similar to on_connect() except used by AutoShardedClient to denote when a particular shard ID has connected to Discord.

18. on_shard_disconnect(shard_id)
    Similar to on_disconnect() except used by AutoShardedClient to denote when a particular shard ID has disconnected from Discord.
    Debug:
19. on_error(event, *args, **kwargs)
    Usually when an event raises an uncaught exception, a traceback is logged to stderr and the exception is ignored. If you want to change this behaviour and handle the exception for whatever reason yourself, this event can be overridden. Which, when done, will suppress the default action of printing the traceback.
20. on_socket_event_type(event_type)
    Called whenever a websocket event is received from the WebSocket.
    This is mainly useful for logging how many events you are receiving from the Discord gateway.
21. on_socket_raw_receive(msg)
    Called whenever a message is completely received from the WebSocket, before it's processed and parsed. This event is always dispatched when a complete message is received and the passed data is not parsed in any way.
22. on_socket_raw_send(payload)
    Called whenever a send operation is done on the WebSocket before the message is sent. The passed parameter is the message that is being sent to the WebSocket.
    Gateway:
23. on_ready()
    Called when the client is done preparing the data received from Discord. Usually after login is successful and the Client.guilds and co. are filled up.
24. on_resumed()
    Called when the client has resumed a session.
25. on_shard_ready(shard_id)
    Similar to on_ready() except used by AutoShardedClient to denote when a particular shard ID has become ready.
26. on_shard_resumed(shard_id)
    Similar to on_resumed() except used by AutoShardedClient to denote when a particular shard ID has resumed a session.
    Guilds:

27. on_guild_available(guild)
28. on_guild_unavailable(guild)
    Called when a guild becomes available or unavailable. The guild must have existed in the Client.guilds cache.
29. on_guild_join(guild)
    Called when a Guild is either created by the Client or when the Client joins a guild.
30. on_guild_remove(guild)
    Called when a Guild is removed from the Client.
31. on_guild_update(before, after)
    Called when a Guild updates, for example:
32. on_guild_emojis_update(guild, before, after)
    Called when a Guild adds or removes Emoji.
33. on_guild_stickers_update(guild, before, after)
    Called when a Guild updates its stickers.

34. on_invite_create(invite)
    Called when an Invite is created. You must have manage_channels to receive this.
35. on_invite_delete(invite)
    Called when an Invite is created. You must have manage_channels to receive this.
    Integrations:
36. on_integration_create(integration)
    Called when an integration is created.
37. on_integration_update(integration)
    Called when an integration is updated.
38. on_guild_integrations_update(guild)
    Called whenever an integration is created, modified, or removed from a guild.
39. on_webhooks_update(channel)
    Called whenever a webhook is created, modified, or removed from a guild channel.
40. on_raw_integration_delete(payload)
    Called when an integration is deleted.
    Interactions:
41. on_interaction(interaction)
    Called when an interaction happened.
    This currently happens due to slash command invocations or components being used
    Members:
42. on_member_join(member)
    Called when a Member joins a Guild.
43. on_member_remove(member)
    Called when a Member leaves a Guild.
44. on_raw_member_remove(payload)
    Called when a Member leaves a Guild.
    Unlike on_member_remove() this is called regardless of the guild or member being in the internal cache.
45. on_member_update(before, after)
    Called when a Member updates their profile.
46. on_user_update(before, after)
    Called when a User updates their profile.
47. on_member_ban(guild, user)
    Called when user gets banned from a Guild.
48. on_member_unban(guild, user)
    Called when a User gets unbanned from a Guild.
49. on_presence_update(before, after)
    Called when a Member updates their presence.
    Messages:
50. on_message(message)
    Called when a Message is created and sent.
51. on_message_edit(before, after)
    Called when a Message receives an update event. If the message is not found in the internal message cache, then these events will not be called. Messages might not be in cache if the message is too old or the client is participating in high traffic guilds.
52. on_message_delete(message)
    Called when a message is deleted. If the message is not found in the internal message

cache, then this event will not be called. Messages might not be in cache if the message is too old or the client is participating in high traffic guilds.

53. on_bulk_message_delete(messages)
    Called when messages are bulk deleted. If none of the messages deleted are found in the internal message cache, then this event will not be called. If individual messages were not found in the internal message cache, this event will still be called, but the messages not found will not be included in the messages list.
54. on_raw_message_edit(payload)
    Called when a message is edited. Unlike on_message_edit(), this is called regardless of the state of the internal message cache.
55. on_raw_message_delete(payload)
    Called when a message is deleted. Unlike on_message_delete(), this is called regardless of the message being in the internal message cache or not.
56. on_raw_bulk_message_delete(payload)
    Called when a bulk delete is triggered. Unlike on_bulk_message_delete(), this is called regardless of the messages being in the internal message cache or not.
    Reactions:
57. on_reaction_add(reaction, user)
    Called when a message has a reaction added to it.
58. on_reaction_remove(reaction, user)
    Called when a message has a reaction removed from it.
59. on_reaction_clear(message, reactions)
    Called when a message has all its reactions removed from it.
60. on_reaction_clear_emoji(reaction)
    Called when a message has a specific reaction removed from it.
61. on_raw_reaction_add(payload)
    Called when a message has a reaction added. Unlike on_reaction_add(), this is called regardless of the state of the internal message cache.
62. on_raw_reaction_remove(payload)
    Called when a message has a reaction removed. Unlike on_reaction_remove(), this is called regardless of the state of the internal message cache.
63. on_raw_reaction_clear(payload)
    Called when a message has all its reactions removed. Unlike on_reaction_clear(), this is called regardless of the state of the internal message cache.
64. on_raw_reaction_clear_emoji(payload)
    Called when a message has a specific reaction removed from it. Unlike on_reaction_clear_emoji() this is called regardless of the state of the internal message cache.
    Roles
65. on_guild_role_create(role)
66. on_guild_role_delete(role)
    Called when a Guild creates or deletes a new Role
67. on_guild_role_update(before, after)
    Called when a Role is changed guild-wide.
    Scheduled Events
68. on_scheduled_event_create(event)
69. on_scheduled_event_delete(event)
    Called when a ScheduledEvent is created or deleted.

70. on_scheduled_event_update(before, after)
    Called when a ScheduledEvent is updated.
71. on_scheduled_event_user_add(event, user)
72. on_scheduled_event_user_remove(event, user)
    Called when a user is added or removed from a ScheduledEvent.
    Stages:
73. on_stage_instance_create(stage_instance)
74. on_stage_instance_delete(stage_instance)
    Called when a StageInstance is created or deleted for a StageChannel.
75. on_stage_instance_update(before, after)
    Called when a StageInstance is updated.
    Threads:
76. on_thread_create(thread)
    Called whenever a thread is created.
77. on_thread_join(thread)
    Called whenever a thread is joined.
78. on_thread_update(before, after)
    Called whenever a thread is updated. If the thread could not be found in the internal cache this event will not be called. Threads will not be in the cache if they are archived.
79. on_thread_remove(thread)
    Called whenever a thread is removed. This is different from a thread being deleted.
80. on_thread_delete(thread)
    Called whenever a thread is deleted. If the thread could not be found in the internal cache this event will not be called. Threads will not be in the cache if they are archived.
81. on_raw_thread_update(payload)
    Called whenever a thread is updated. Unlike on_thread_update() this is called regardless of the thread being in the internal thread cache or not.
82. on_raw_thread_delete(payload)
    Called whenever a thread is deleted. Unlike on_thread_delete() this is called regardless of the thread being in the internal thread cache or not.
83. on_thread_member_join(member)
84. on_thread_member_remove(member
    Called when a ThreadMember leaves or joins a Thread.
85. on_raw_thread_member_remove(payload)
    Called when a ThreadMember leaves a Thread. Unlike on_thread_member_remove() this is called regardless of the member being in the internal thread's members cache or not.
    Voice:
86. on_voice_state_update(member, before, after)
    Called when a Member changes their VoiceState.
    The following, but not limited to, examples illustrate when this event is called:
    A member joins a voice or stage channel.
    A member leaves a voice or stage channel.
    A member is muted or deafened by their own accord.
    A member is muted or deafened by a guild administrator.

B ChatGPT output

```
import discord

client = discord.Client()
prefix = "?"

@client.event
async def on_ready():
    print(f'{client.user} has connected to Discord!')

@client.event
async def on_message(message):
    global prefix
    if message.content.startswith(prefix + 'changeprefix'):
        new_prefix = message.content.split()[1]
        prefix = new_prefix
        await message.channel.send(f'Prefix changed to
{new_prefix}')
    elif message.content.startswith(prefix + 'hello'):
        await message.channel.send("Hello back!")

@client.event
async def on_message_edit(before, after):
    if before.content != after.content:
        await after.channel.send(f'Message edited:
{before.content} -> {after.content}')

@client.event
async def on_message_delete(message):
    await message.channel.send(f'Deleted message:
{message.content}')

client.run('INSERT TOKEN HERE')
```

C Pysolfc Code

pysolfc/simplesimon/controllers.scala

```
package pysolfc.simplesimon

import org.combinators.cls.interpreter.ReflectedRepository
import org.combinators.solitaire.domain.Solitaire
import org.combinators.solitaire.shared.SolitaireDomain
import org.combinators.solitaire.shared.python.PythonSemanticTypes
import pysolfc.shared.GameTemplate

trait controllers extends GameTemplate  with PythonSemanticTypes  {
   // dynamic combinators added as needed
   override def init[G <: SolitaireDomain](gamma :
ReflectedRepository[G], s:Solitaire) :  ReflectedRepository[G] = {
     var updated = super.init(gamma, s)
     println(">>> Simplesimon Controller dynamic combinators.")
     updated = updated
       .addCombinator (new ProcessView(s))
     updated
   }
}
```

pysolfc/simplesimon/SimpleDomain.scala

```
package pysolfc.simplesimon

import org.combinators.cls.interpreter.combinator
import org.combinators.cls.types.Type
import org.combinators.solitaire.domain._
import org.combinators.solitaire.shared.SolitaireDomain
import
org.combinators.solitaire.shared.compilation.CodeGeneratorRegistry
import org.combinators.solitaire.shared.python.{PythonSemanticTypes,
constraintCodeGenerators}
import org.combinators.templating.twirl.Python
import pysolfc.shared.GameTemplate

/**
 * @param solitaire    Application domain object with details about
solitaire variation.
 */
class SimpleDomain(override val solitaire:Solitaire) extends
SolitaireDomain(solitaire) with GameTemplate with PythonSemanticTypes {

 /**
   * Convert ID into string. Each different variation adds a unique ID
to the pygames grouping
   */
 @combinator object simplesimonID extends
IdForGame(pygames.simplesimon)
```

```
 @combinator object OutputFile {
   def apply: String = "simplesimon"

   val semanticType: Type = game(pysol.fileName)
 }
 /**
   * NO special constraints just yet
   */
 @combinator object DefaultGenerator {
   def apply: CodeGeneratorRegistry[Python] =
constraintCodeGenerators.generators

   val semanticType: Type = constraints(constraints.generator)
 }

 /**
   * Deal may require additional generators.
   */
 @combinator object DefaultDealGenerator {
   def apply: CodeGeneratorRegistry[Python] =
constraintCodeGenerators.mapGenerators

   val semanticType: Type = constraints(constraints.map)
 }

 /**
   * Specialized methods to help out in processing constraints.
Specifically,
   * these are meant to be generic, things like getTableua,
getReserve()
   */
 @combinator object HelperMethodsCastle {
   def apply: Python = {
     val helpers:Seq[Python] = Seq(generateHelper.tableau(),
generateHelper.waste())

     Python(helpers.mkString("\n"))
   }


   val semanticType: Type = constraints(constraints.methods)
 }

 /*
 /**
   * Specialized methods to help out in processing constraints.
Specifically,
   * these are meant to be generic, things like getTableua,
getReserve()
   */
```

```
@combinator object HelperMethodsCastle {
  def apply: Python = {
    val helpers:Seq[Python] = Seq(generateHelper.tableau(),


      Python(s"""
                |def toLeftOf (targetCards, source):
                |    for t in tableau():
                |        if t == source:
                |            return False
                |        if t.cards == targetCards:
                |            return True
                |    return False
                |
                |def allSameRank():
                |    top = tableau()[0].cards
                |    if len(top) == 0:
                |      return False
                |    for t in tableau():
                |        next = t.cards[-1]
                |        if next.rank != top[-1].rank:
                |            return False
                |    return True
                |""".stripMargin)
    )


    Python(helpers.mkString("\n"))
  }


  val semanticType: Type = constraints(constraints.methods)
}
//
//  @combinator object InitView {
//    def apply(): Python = {
//
//      val tableau =
solitaire.containers.get(SolitaireContainerTypes.Tableau)
//      val stock =
solitaire.containers.get(SolitaireContainerTypes.Stock)
//
//      val sw:Python = layout_place_stock(solitaire, stock)
//
//      // when placing a single element in Layout, use this API
//      val cs:Python = layout_place_tableau(solitaire, tableau)
//
//      // Need way to simply concatenate Python blocks
//      val comb = Python(sw.getCode.toString ++ cs.getCode.toString)
//      comb
```

```
 //    }
 //
 //    val
 //
 //    semanticType: Type = game(game.view)
 //  }
*/
}
```

pysolfc/simplesimon/Simple.scala

```
package pysolfc.simplesimonAlsoBad

import org.combinators.cls.git.{EmptyResults, Results}
import org.combinators.cls.interpreter.ReflectedRepository
import org.combinators.cls.types.Constructor
import org.combinators.solitaire.shared.compilation.{DefaultMain,
SolitaireSolution}
import org.combinators.templating.persistable.PythonWithPath
import
org.combinators.templating.persistable.PythonWithPathPersistable._

trait PythonSimplesimonT extends SolitaireSolution {

 lazy val repository = new SimpleDomain(solitaire) with controllers {}
 import repository._
 lazy val Gamma = repository.init(ReflectedRepository(repository,
classLoader = this.getClass.getClassLoader), solitaire)

 lazy val combinatorComponents = Gamma.combinatorComponents
 lazy val targets: Seq[Constructor] = Seq(game(complete))
 lazy val jobs =
   Gamma.InhabitationBatchJob[PythonWithPath](targets.head)    // Why
just singular target here?

 lazy val results:Results = EmptyResults().addAll(jobs.run())
}

// Match the Trait with multi card moves with the model that defines
multi card moves
object PythonSimpleMain extends DefaultMain with PythonSimplesimonT {
 override lazy val solitaire =
org.combinators.solitaire.simplesimon.simplesimon
}
```

We also ensured to give the game its own ID to work with the PySolFC hub software

```
object pygames {
 val castle:Int = 99000
 val klondike:Int = 99001
 val narcotic:Int = 99002
```

```
 val archway:Int = 99003
 val freecell:Int = 99004
 val fan:Int = 99005
 val minimal:Int = 99999
 val simplesimon:Int = 99006
}
```

D Discord Modeling Tutorial

Part 1:

Overview of the model:

Name: The name of the bot & the name of the output file

Libraries: what external libraries the bot uses (include import discord)

Description: the bot's description & the bot's about me status when the bot is launched

Prefix:The prefix used when calling commands ex: ?hello

commName: An array of the names for all bot commands

commArgs: An array of all the arguments each command takes (needs to be in the same order as the name)

commContent: An array of the contents of all the commands, must include proper indentation and new line characters after the first line

eventName: An array of the events triggered by the bot (must properly match the one used in discord.py)

eventContent: An array of the contents of the events triggered by the bot, must include proper indentation and new line characters after the first line.

Part 2:

Converting a bot to the model:

To examine this, we'll be taking a look at a very basic example bot and separating it into different pieces to understand how to convert it to the model. It is important to note, discord bots are generally flexible in how they can be set up so these examples may not exactly reflect how your bot is set up. This does assume the reader has some understanding of discord bots.

Example import statements:

```
import random

import discord
from discord.ext import commands
```

What this looks like in the model:

```
libraries = "import random \n import discord \n from discord.ext import commands",
```

To convert import statements to the model you basically convert it to a string, make sure you keep any indentations and new lines in the model.

Example event:

```
@bot.event
async def on_message_delete(message):
    embed = discord.Embed(title="Message deleted in " + "#" + message.channel.name, colour=discord.Colour(0xbe4041),
                          description=message.content)
    await message.channel.send('', embed=embed)
```

Example definition:

```
bot = commands.Bot(command_prefix='?', description='Testing for MQP!', intents=intents)
```

```
description = "An example bot for MQP",
prefix = "?",
```

The model automatically generates the intents section so all you have to model is what you want to use as your command prefix and what you want the description of your bot to be.

Example command:

```
@bot.command()
async def add(ctx, left: int, right: int):
    """Adds two numbers together."""
    await ctx.send(left + right)
```

This is a basic addition command, it just returns the sum of the imputed 2 numbers.

To model a command: The important pieces you would take from this is:

The command name: Add

The arguments of the command: ctx, left, right

And the contents of the command: await ctx.send(left + right)

This command implemented into the model would look like this:

```
commName = Array("add"),
  commArgs = Array("ctx, left, right"),
  commContent = Array("await ctx.send(left + right)"),
```

If you were to add additionally commands, you would separate it by a comma as a new string within the array definition

```
commName = Array("add", "command2"),
  commArgs = Array("ctx, left, right", "ctx"),
  commContent = Array("await ctx.send(left + right)","await ctx.send(1)"),
```

Example event:

```
@bot.event
async def on_message_delete(message):
    embed = discord.Embed(title="Message deleted in " + "#" + message.channel.name, colour=discord.Colour(0xbe4041), description=message.content)
    await message.channel.send('', embed=embed)
```

Converting an event to the model is simpler than converting a command since events have predefined arguments by the API so all you need is the event name and the contents of the event.

```
eventNames = Array("on_message_delete"),
eventContents = Array(" embed = discord.Embed(title=\"Message deleted in \" + \"#\" + message.channel.name, colour=discord.Colour(0xbe4041), " +
  "description=message.content)\n    await message.channel.send('', embed=embed)"),
```

Part 3:

Creating a new bot with the model

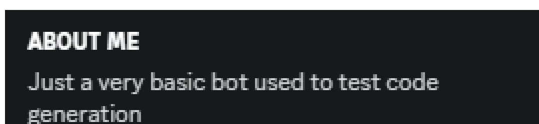
```
package object hello {
  val hello:Discord = {
    Discord(name = "",
      libraries = "",
      description = "",
      prefix = "",
    commName = Array(""),
      commArgs = Array(""),
      commContent = Array(""),
      eventNames = Array(""),
      eventContents = Array("")
    )
  }
}
```

Here is what an empty model looks like

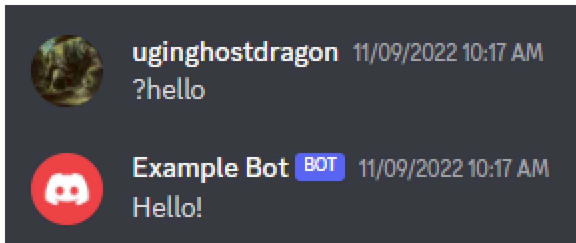
Step 1: Decide what the bot name is, the bot name will determine what the file name the generator outputs to.

Step 2: Discord bots need to import discord.py to function so the model needs at least "import discord" in the libraries section in addition to any other external libraries the bot needs to function.

Step 3: Writing the description for your bot, it is used as the about me section when the bot is activated on discord



Step 4: prefix: the prefix is the character(s) used so that the bot is able to recognize that a command is being called.



In this example "?" is the prefix

The next two parts of the model are commands and events.

Each consists of multiple parts in the model. Events are more restrictive since they rely on predefined functions in discord.py.

Commands consist of 3 parts

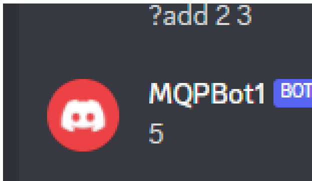
Name: What the command is named and what the user needs to call for the bot to register the command.

Arguments: What arguments the command takes from the user to function, at minimum commands need to take some form of a context argument such as ctx so the bot is able to get basic information like where the command was sent and by who.

Contents: What the bot does once it receives the command.

For example, this add command has the name add, it takes the arguments of ctx (context) & a left and a right integer. Once called, it will send a response message adding the two numbers together.

```python
@bot.command()
async def add(ctx, left: int, right: int):
    await ctx.send(left + right)
```



So to model a command, you need to think about what you want to call it, what the command needs to take as inputs and how the bot responds to the inputs.

Events:

Events are much more rigid in their definition.

Step 1 Decide which event fits your bot and what you need to use in this situation.

Step 2: Determine the proper naming convention for that event (consult the documentation)

Step 3: Determine what the bot should do when that event is triggered

Step 4: Synthesize into the model

Part 4:

Augmenting the model

The two core areas where the model can be augmented is which events the model supports and what the model generates by default.

To add new events from the discord.py API:

In the DiscordTemplate.scala file there is a method called genEvent

Within that method is a switch case based on the Event name, to add a new event you simply need to add a new case based on the new event you want to implement.

Events have predefined arguments so simply consult either the discord.py API page or the included event list in the appendix (insert appendix #)

Editing the predefined section:

In the apply method in the DiscordTemplate.scala file, there is a variable called code which contains a series of python code and variables which is what the generator outputs for the final product. You can make changes to the code as you see fit whether it is introducing default helper functions, comments etc.