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 1, 2019

Submitted by:

Drew Ciccarelli

Ian MacGregor

Simon Redding

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

Applied combinatory logic synthesis has the potential to greatly improve the efficiency of software development. This project is an exploration of implementing CLS in an existing software platform, and an analysis of its impacts on the development process.

Table of Contents

Abstract 2

I Introduction 4

II Background 5

- Existing Paradigm Overview 5
- Solitaire Overview 5
- Generation - A new Paradigm 6
- Impacts of Generation 7
- Project History 7

III Design 8

- Design of the Synthesizer 8
- Design of the Solitaire Models 8
  - Abstraction for Solitaire Families 9
  - Java vs Scala Modeling 9
- Design of Generated Code 10

IV: Development 12

- Tools and IDEs 12
- Development Cycle - Creating New Variations 12
- Debugging and Testing 14
- Compiler, Client, and Generation Overview 14

IV Results 17

- Base Solitaire Games 17
- Families and Variations 20
- Development Time Across Game Types 24
- Life as a Modeler 25

V Conclusion & Recommendations 26

References 28

Appendix A: Solitaire Creation Tutorial 29

Appendix B: Modeling Needed for Golf 42

I Introduction

Software developers attempting to work with complex frameworks are faced with a steep learning curve (Fayad and Schmidt, 1997). This is especially prominent when onboarding new developers in an unfamiliar framework and when expanding an established framework. Most research on software development processes have focused on the generic problems faced by software engineers over the entire lifecycle of software development. We are interested in the specific challenges faced by developers when using third party software frameworks.

Large frameworks are created by many developers over a long period of time, growing increasingly complex throughout the process (Butler, 2002). As a result, new developers working in these frameworks often face a steep learning curve (Yates 2004) and, while catching up, spend their time writing suboptimal code. While onboarding, training, and development techniques can speed up this process somewhat, in the end a new developer simply needs to work with the framework long enough to build up a familiarity with its components. In addition to this problem, developing new code for these frameworks is complicated because new framework developers are often not familiar with the entire framework, resulting in suboptimal code.

Generating code helps alleviate these problems by minimizing the framework knowledge needed by new users, reducing code rewriting, and standardizing the structure of code. In a generation system, users only need to understand the abstractions of the framework and have some basic experience with modeling to begin working on a system. A generation framework can automatically include commonly used functionality, decreasing the amount of repeated code. Finally, since the generation process pulls this code from a single source, the resulting code should look exactly the same each time.

Next-Gen Solitaire is a code generation framework designed by George Heineman, Jan Bessais, and Boris Düdder (Heineman et al., 2019). Given a Scala model representing the rules and design of a solitaire game, it uses Combinatory Logic Synthesis (Bessai et al., 2014) to generate Java or Python code for that game. The framework has been under development for several years and includes a wide variety of pre-built solitaire elements. Additionally, developers are able to create their own specialized logic to deal with any game rules or elements that are outside the scope of the base framework.

Our purpose in this project was to act as developers for this code generation framework and evaluate its limitations. At first, we simply built individual solitaire games, but as we became more comfortable with the framework, we were able to implement complicated specialized logic and even improve the code generation framework itself. Ultimately, our own development experience stands as a testament to the effectiveness of this development technique, and the success of the framework itself.

II Background

Existing Paradigm Overview

Software Engineers use a variety of programming languages, each with their own strengths and weaknesses. These languages can be classified into a taxonomy based on a concept called a “programming paradigm”. The three most commonly used modern paradigms are *procedural*, *functional*, and *object-oriented*. Many programming languages are created with a certain paradigm in mind and encourage developers to use it as well. For example, C is based on procedural, Java on object-oriented, and Lisp on functional.

Procedural programming is the oldest paradigm still used in modern computing. The basic idea is to write and run a list of instructions for the computer. These instructions are stored in procedures (or functions), which can be called by other procedures or by the user when running a program.

Functional programming is a paradigm where programmers treat computation as the evaluation of mathematical functions and seek to avoid mutable data structures. Often relying on recursion, these programs are smaller and easier to understand, though they often are not as efficient to execute as programmers written in, say, procedural programming languages.

Object-Oriented programming is the newest and most widely used paradigm in modern software development. Its main principle is that code should be stored in objects as data entries known as *fields* and functions/procedures known as *methods*. Generally, an object is an abstract representation of a real-life concept. For example, an Airplane class may contain fields for its capacity, max speed, and airline and a method for how long it takes to travel a given distance. This paradigm results in code that is easier for a user to read, understand, and expand.

A code generation framework must decide whether to focus on a specific programming language or whether it should remain generic and be applicable to a wide range of programming languages. While Next-Gen Solitaire originally focused solely on Java, it is now generic; the framework can generate code in both Java and Python, and could be further expanded to include more languages.

Solitaire Overview

Dozens of different solitaire applications exist online. Many only offer users a small selection of the most popular variations of the game. However, there exist a few applications that allow players to experience dozens of variations. We will examine PySol (PySolFC, 2019) as an example of how most solitaire applications are written as it is open source, has hundreds of variations, and allows users to add their own game variations.

PySol uses the Object-Oriented Paradigm and relies heavily on a hierarchy of inheritance to manage its variations. There are *game* files which each contain a group of similar game classes that extend each other. The base framework of the game contains files for common elements of solitaire such as cards, stacks, and layouts. However, each of these classes contains extensions for every needed version of an element. For example, there are about 30 different types of stacks that are needed for different solitaire variations. Some of these stacks contain methods that most games do not use.

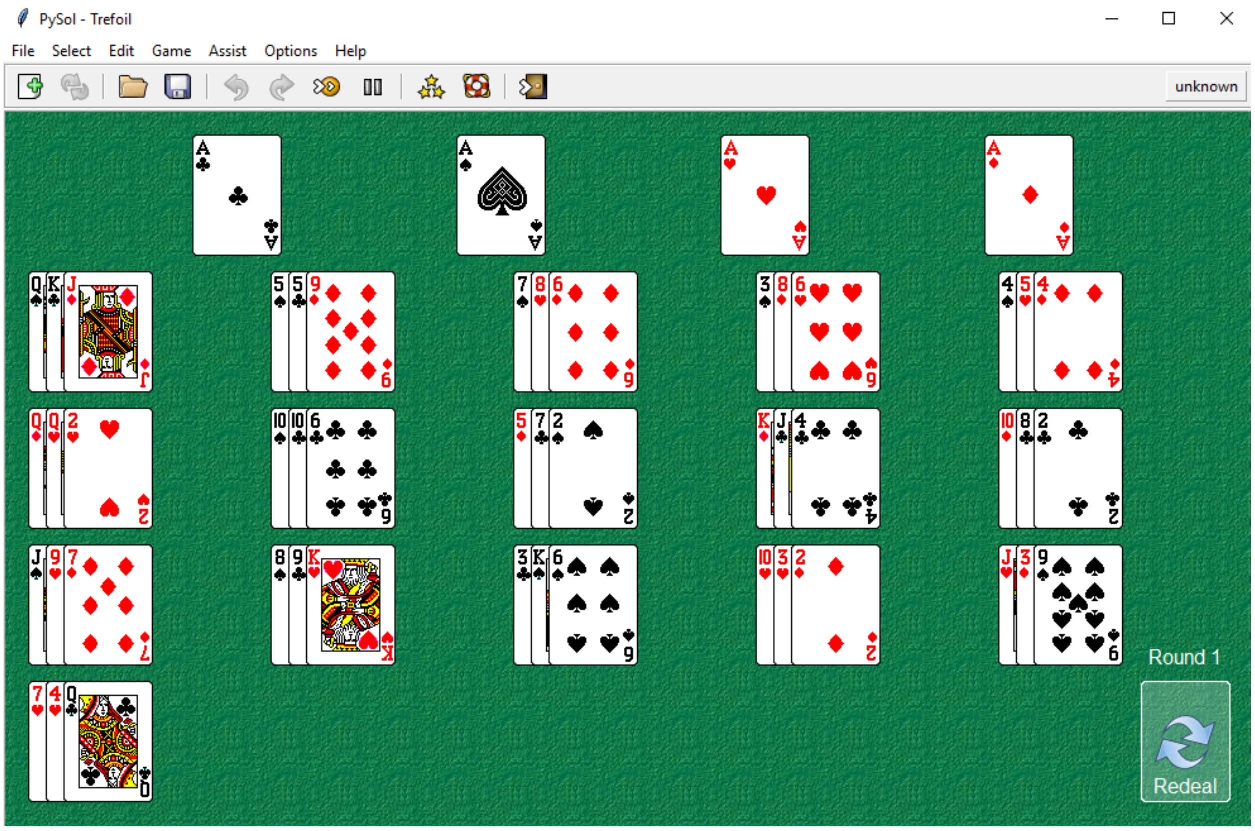


Figure 1: The PySol interface for Trefoil Solitaire

While PySol is predominantly a consumer application for playing existing variations of solitaire, it allows users to create new variations of solitaire by adding a file with code for a new variation in it. This is great for our system as it means that we do not need to develop our own framework to run Python-based solitaire games; we only need to generate the code for our own PySol variations.

Generation - A new Paradigm

The first key issue that comes with the purely object-oriented design of PySol is that there is a lot of code bloat in the basic framework of the game. For example, the shared class *Stack* in PySolFC contains nearly 3000 lines of Python code, with countless examples of code fragments necessary only for a specific variation. There are classes and methods that are not used in most games but are needed for a few and so they must be included in this common class. The second issue is code repetition. Some variations use similar code for small parts of their game but are not related enough for one to parent the other. In these cases code must be copied and repeated in different areas.

A new paradigm of generating code solves these issues. The basic idea of code generation is using models to produce individual variations (for example, a solitaire game) that each have only the code it needs to function. In the ideal case, a user only needs to create a new model and a generation framework will do the rest. In reality, many variations will require custom code added either just for the single variation, or for the framework as a whole.

Impacts of Generation

Generating code has the potential of saving programmers hours of work and tedious debug sessions. Once a framework for generating within a domain space is created, someone should only need to model a desired application and the framework will generate it. The process of creating the models and generating the code should be straightforward, allowing users with less programming experience to more easily develop applications. However, experienced programmers will still be needed to expand the framework if new functionality is desired.

In reality, our system is not at a stage where this is completely feasible because the generation process still requires several steps and any resulting errors would be meaningless to some who isn't well trained in the system and in programming. Therefore, programmers are still necessary for the system.

Project History

The Next Generation Solitaire project has a long history which began with an Object-Oriented solitaire app, KombatSolitaire, developed for an undergraduate course at WPI. This app was extended by adding new classes to an existing framework, similar to the PySol system but written in Java. Heineman created the LaunchPad plugin for Eclipse which integrated and expanded the FeatureIDE framework (Heineman et al., 2015); This allowed the team to create solitaire games by combining existing game features. This framework was used to create four different variations, but was not as scalable as desired.

The next framework wrote its combinators using Scala but retained Java modeling. Combinators are lambda functions with no free variables. In other words, they are nameless functions that do not bind any variables to locations in memory but instead only use their parameters to create an output. This system proved advantageous in scalability and provided the ability to make combinators to generate code in other languages, such as Python. While a great improvement, in this system it was difficult to add unique features that were not specified in the models (Heineman et al., 2018). Eventually, this system was modified to use Scala modeling which alleviated these issues.

III Design

Design of the Synthesizer

At a high level, the Next-Gen Solitaire framework contains several major parts. The first is a system of modeling solitaire structural elements: classes that define types of decks, card piles, or basic moves. Because these models exist solely to define structure, their logic does not need to be implemented until the variation is generated. Prior to that, all that is needed is knowledge of the relationship between these structures, and thus the classes or objects that define them are almost entirely empty. Some do contain basic functions or fields, but at this stage of the project they are used generally to allow a programmer to create an abstracted model of a variation, which will only later be given implementation-level logic through the synthesis process. For example, a move defined here, such as a *MultipleCardsMove*, needs only to contain fields that specify a source container and a target container. A source container could be the tableau of cards at play in a soltiaire variation, and the target container could be the foundation, where cards are placed in order from Ace to King, by suit. This provides the programmer with all they need to model the move, and the generator can take care of the rest.

The next major piece is constraints. Constraints are a way to model logic, as opposed to structure. They are used in the modeling process as a way to define the conditions required for moves to occur or for any change in game state, such as a victory. They generally take the form of Boolean functions that evaluate an aspect of the game or the states of various objects. These constraints can then be chained together in a functional style. For example, if a move requires that the moving card be placed on top of a card one rank above and of the opposite color, then the programmer models this with a constraint, such as *AndConstraint(isRank(...), OppositeColor(...)).* Like the structural elements, at this stage constraints represent a model of logic that will be implemented in the future making their definitions simple and short. At this stage, a move constraint does not need to provide the actual logic that limits the operation of its move; this is the job of the synthesizer. Instead, these constraints simply need to contain the information that the synthesizer will need in the future. While, ultimately, one goal of this project is to minimize the amount of logic coded by the programmer, some will always be required. In our case, during the modeling process, the programmer must thoroughly describe the game's rules using this system, but is generally spared from delving further into the solitaire framework.

These pieces provide the generator with all it needs to synthesize a solitaire variation, and thus the synthesizer itself comprises the final piece. Once the previous steps are complete, the synthesizer fills in the rest of the code and the solitaire game is complete.

Design of the Solitaire Models

Each individual solitaire variation has its own model. These models seek to define the variation's structure and rules in their most minimal form, allowing the synthesizer to create the rest. To do this requires several steps. An example of a fully-realized model can be found in Appendix B.

First, the model itself is specified as a Scala object containing at minimum a structure definition (a list of structural pieces, such as the Foundation or Tableau), a layout map of where each structural element is placed on the game board, a deal, a list of specialized elements (if any), a list of moves, and a win condition. Each of these pieces can be built within this class using instances of model elements: container types, move types, etc. The logic required for each move or win condition can be defined with the use of constraints. This is where the majority of the work for the modeler takes place, even though the resulting code is generally very short.

Next, each variation must be registered in a *routes* file that records the location of the model, allowing the synthesizer to find it during generation at a later time. Each variation needs a definition class, which contains some setup and naming information to be used by the synthesizer. Additionally, for each family a domain class must be created. These are very much the same between variations, but allow the programmer to specify more meta-information, such as requesting additional imports to the generated code, or adding helper functions directly.

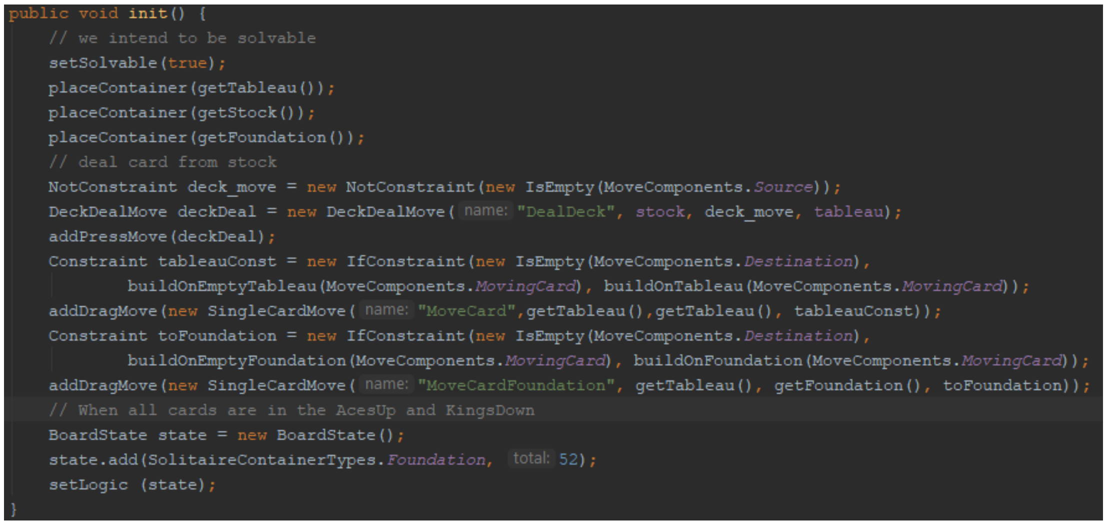
Finally, each variation needs a controllers trait. This consists of a Scala trait that contains some minimal information about generation, but mainly serves as a place to specify event listeners required by the game's moves. For example, if a variation allows cards to be moved from the Foundation to the Tableau, the Foundation must require, at minimum, a mouse press handler, and the tableau must require, at minimum, a mouse released handler. The controllers also provide a place to specify logic for custom moves, such as dealing to a custom pile.

Abstraction for Solitaire Families

The abstraction process is very simple, and only a few things need to be done. Essentially, the building blocks of the model of the family's base variation are simply moved into a Scala trait, allowing each variation to extend the trait and change the parts they need. As more variations are added to a family, rules, structures, or helpful methods that change between each can be left in this base trait, which we generally refer to as *variationPoints*. Each new variation can override fields from *variationPoints*, and further development and expansion follows the traditional hierarchical object-oriented model. Controller information can be shared between families, so only one class is required. However, switch or conditional statements may be required to handle special cases among variations. All that is left is to register each new variation in the routes file.

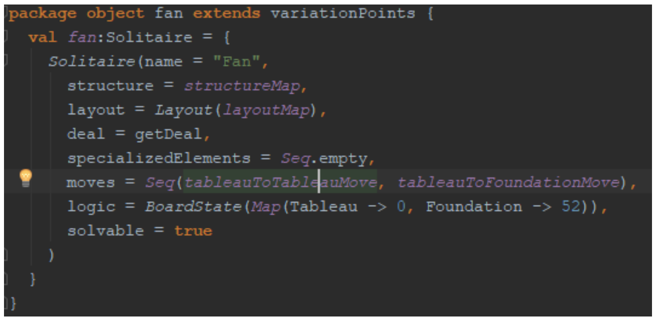
Java vs Scala Modeling

When we first started this project, the framework used Java to model the soltiaire variation, while the synthesizer was written in Scala. One advantage of this was Java is a distinctly familiar language to not only our team, but to a large portion of developers. However, partway through the project we switched to a new capability that used Scala to create the models, which provided a few advantages over Java. The Scala compiler can detect errors in the model long before any attempt was made to generate the code from it. The Scala modeling also shortened the amount of code we needed to write for a given variation. For example, the modeling code needed for Fan in Java was about 175 lines long while in Scala it was about 90 lines long. The area Scala cleaned up the most was the initialization of a game shown in Figure 2 and 3, which compares the Java and Scala for Fan's initialization.



```java
public void init() {
    // we intend to be solvable
    setSolvable(true);
    placeContainer(getTableau());
    placeContainer(getStock());
    placeContainer(getFoundation());
    // deal card from stock
    NotConstraint deck_move = new NotConstraint(new IsEmpty(MoveComponents.Source));
    DeckDealMove deckDeal = new DeckDealMove( name: "DealDeck", stock, deck_move, tableau);
    addPressMove(deckDeal);
    Constraint tableauConst = new IfConstraint(new IsEmpty(MoveComponents.Destination),
            buildOnEmptyTableau(MoveComponents.MovingCard), buildOnTableau(MoveComponents.MovingCard));
    addDragMove(new SingleCardMove( name: "MoveCard",getTableau(),getTableau(), tableauConst));
    Constraint toFoundation = new IfConstraint(new IsEmpty(MoveComponents.Destination),
            buildOnEmptyFoundation(MoveComponents.MovingCard), buildOnFoundation(MoveComponents.MovingCard));
    addDragMove(new SingleCardMove( name: "MoveCardFoundation", getTableau(), getFoundation(), toFoundation));
    // When all cards are in the AcesUp and KingsDown
    BoardState state = new BoardState();
    state.add(SolitaireContainerTypes.Foundation,  total: 52);
    setLogic (state);
}
```

Figure 2: Fan Initialization via Java Modeling



```scala
package object fan extends variationPoints {
  val fan:Solitaire = {
    Solitaire(name = "Fan",
      structure = structureMap,
      layout = Layout(layoutMap),
      deal = getDeal,
      specializedElements = Seq.empty,
      moves = Seq(tableauToTableauMove, tableauToFoundationMove),
      logic = BoardState(Map(Tableau -> 0, Foundation -> 52)),
      solvable = true
    )
  }
}
```

Figure 3: Fan Initialization via Scala Modeling

Design of Generated Code

The code generated by the synthesizer follows an object-oriented, Entity-Boundary-Controller model as required by the underlying KombatSolitaire framework. The Java solitaire variations use the KombatSolitaire framework (developed by Professor Heineman for an undergraduate class) as is without changes. The entity is comprised of structural and move information: classes that outline the structure of each solitaire game. These classes define container types, decks, cards, moves, and any other such elements. The boundary, on the other hand, focuses more on the environment that each variation is created for. It contains primarily graphical information, GUI controls, and images. Finally, the controllers contain handle all complex interactions with the user: mouse clicks and drags from pile to pile, for example. These controllers are responsible for directing the appropriate moves as the user interacts with the game. Because much of these sections remain the same across all variations, a large part of the code kept compiled and included as a dependency when running a generated game. This allows each variation to be synthesized without needing to regenerate the same code, particularly graphical and high-level structural information, every time, and thus greatly speeds up the generation process.

It is important to note that we were not generating solitaire variations “from scratch” but rather were generating code that depends upon the KombatSolitaire framework. In this way, programmers would no longer be required to natively understand the interface to the KombatSolitaire Framework, but would rather only need to understand how to model the variations using the abstractions provided by the Next-Gen code generation framework.

IV: Development

Tools and IDEs

Throughout the development of this project, we used few external tools or resources. Instead, our development environment consisted solely of the IDEs we were working with. In this case, we used both IntelliJ and Eclipse; the former for writing Scala models and working with the synthesizer, and the latter for testing and running generated solitaire games. The use of each IDE is generally preferential, and has little bearing on the development process itself. Additionally, we used a Github repository for source control, which can be found at https://github.com/combinators.

Development Cycle - Creating New Variations

Although our methods changed slightly throughout the course of the project, we ultimately settled on a single general development approach. What follows is a summary of the implementation of a new solitaire variation family from scratch. A far more detailed tutorial can be found in Appendix A.

The first step is to start creating all the files that will be required for a new variation. To do this, first create a new package in `src/main/scala/org/combinators/solitaire/` with the same name as the variation. Once done, you will need to create four files: a SolitaireController named controllers, a SolitaireTarget with the same name as the solitaire family, a SolitaireGameDomain named following the camelCase convention: “spiderDomain”, and a SolitairePackageObject with the name of the specific variation. For each of these files we have created a generic template that can be modified to create specific variations. If you are adding a variation to an existing family, you need only create a new SolitairePackageObject and extend that family’s variationPoints trait.

Next, the modeling begins. Each solitaire object needs several things: a structure map that defines what major structural elements the game needs and what containers belong in each one, a layout map that defines the positioning of each container, a starting deal, a list of any specialized elements used, a list of moves, and a winning condition.

The structure map is generally the first thing created. Most other parts of the game will need to reference existing structures (tableau, foundation, etc) in some way. This is created as a map of structural elements to the containers that populate them. For example, a Spider game will require a Tableau with ten BuildablePiles, a Foundation with 8 Piles, and a StockContainer with one Stock containing two decks.

After this, the layout map is arranged. In this step, the developer must create a map that pairs each structure with a layout rule. A layout rule specifies where on the board each of the structure’s piles will be placed and how they are oriented. A common example of this would be a HorizontalPlacement, which specifies an alignment where each pile of cards in a container should be placed adjacent to each other horizontally.

The next step is creating a Deal, which defines the initial dealing of cards. This could involve dealing different numbers of cards to different piles, placing some face up, some face down, etc. It is constructed of a sequence of DealSteps, which each provide one dealing rule and can be chained together to create more complex initial deals.

Before any moves can be created, constraints for each move must be defined. These constraints define the conditions where a move can be allowed. For example, in a game where you can move a card on the tableau to any other column on the tableau provided that the target column ends with a card that is one higher and of the opposite color than the moving card (the classic solitaire move condition), you may define a constraint like this:

```
def buildOnTableau(cards: MovingCards.type): Constraint = {
 val topDestination = TopCardOf(Destination)
 val bottomMoving = BottomCardOf(cards)
 val isEmpty = IsEmpty(Destination)
 val descend = Descending(cards)
 val suit = AlternatingColors(cards)

 OrConstraint(AndConstraint(isEmpty, descend, suit),
             AndConstraint(descend, suit, OppositeColor(topDestination, bottomMoving)))
}
```

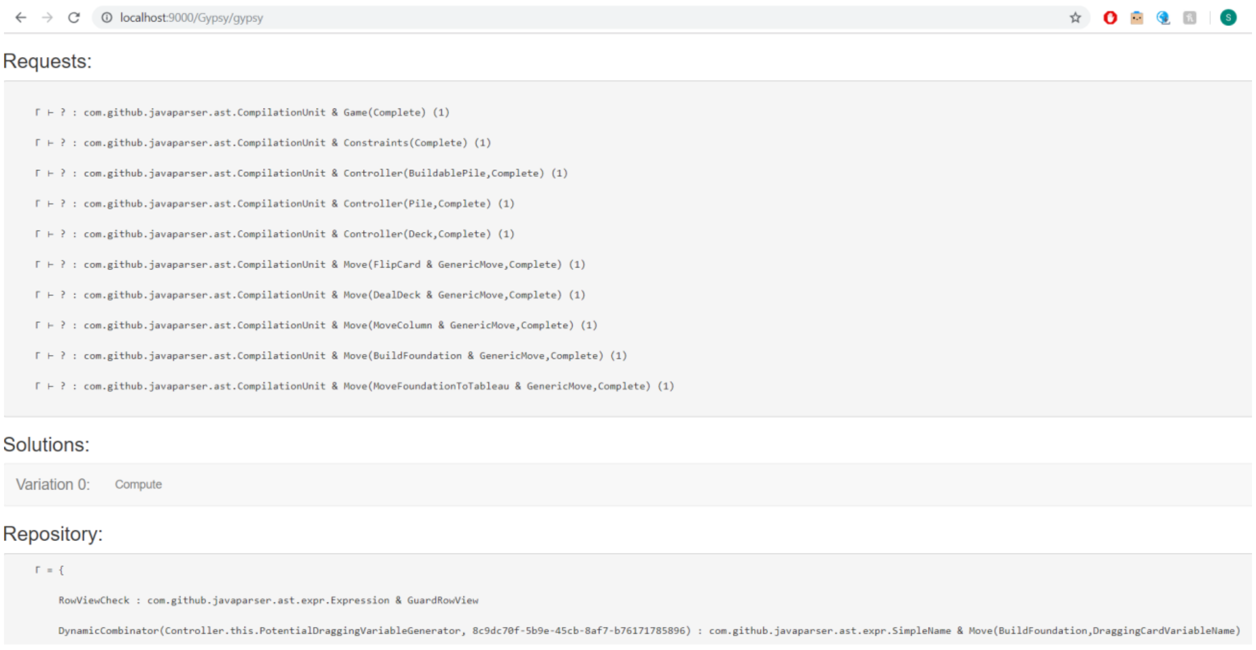
Figure 4: The buildOnTableau constraint for Gypsy

Next, moves can be created simply by specifying the type of move (such as SingleCardMove move or DealDeckMove), source pile (if applicable), destination pile (if applicable), and required constraints. The winning condition is similarly declared, generally using a constraint that checks for a certain board state, such as all cards inside the foundation.

Next, controllers must be specified. Most of this will likely be handled by code in the controller's template and may not need to be changed much from variation to variation, but special controllers may need to be added to help clarify certain rules. For example, if a BuildablePile offers both DragMoves and FlipMoves, a controller must be made to determine which action should be taken when the pile is pressed.

Finally, the game controller must be registered in a special routes file, which tells the synthesizer where it can find each part of the model. After this, and some minor tweaks to the other files, the game is ready for generation.

To generate a game, one needs to run "run service" in IntelliJ, then open a web browser. In the browser, they need to navigate to http://localhost:9000/<FamilyName>/<VariationName> (e.g. http://localhost:9000/Gypsy/gypsy). This will open what we call the *inhabitation page*, which contains several features. First, it contains a list of compilation units: the Game object itself, controllers, constraints, and moves, as well as some rudimentary success/failure information about each one (1 for success, 0 for failure, higher numbers for multiple available interpretations). Below that is a compute button, which, if the compilation units are valid, will create a git repository that can be pulled with the created solitaire project. At the bottom of the page is a (combinator) repository section that contains a list of all the combinators used in the model, which can be helpful for diagnosing certain errors. Once the project is generated, one just needs to pull from the git repository and execute the code with the standalone.jar included as a dependency.



Requests:

```
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Game(Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Constraints(Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Controller(BuildablePile,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Controller(Pile,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Controller(Deck,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Move(FlipCard & GenericMove,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Move(DealDeck & GenericMove,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Move(MoveColumn & GenericMove,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Move(BuildFoundation & GenericMove,Complete) (1)
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Move(MoveFoundationToTableau & GenericMove,Complete) (1)
```

Solutions:

Variation 0: Compute

Repository:

```
Γ = {
    RowViewCheck : com.github.javaparser.ast.expr.Expression & GuardRowView
    DynamicCombinator(Controller.this.PotentialDraggingVariableGenerator, 8c9dc70f-5b9e-45cb-8af7-b76171785896) : com.github.javaparser.ast.expr.SimpleName & Move(BuildFoundation,DraggingCardVariableName)
```

Figure 5: The Inhabitation page for Gypsy Solitaire

Debugging and Testing

The process of debugging in this project at first was very difficult, but quickly became easier as we gained familiarity with the Next-Gen Solitaire framework. The main challenge was learning to use the error reporting system and understanding its output. If the synthesizer recognizes a problem while generating a new variation, it may do one of a few things. If, for example, the model defines a rule ambiguously, the synthesizer will recognize that there are multiple ways to interpret the model and generate all of them. This allows the developer to choose from all possible permutations, even if some of those create games with unintended rules. If, however, the synthesizer detects a problem that prevents it from generating code, it may flag that part as incomplete or incorrect. Unfortunately, this does little more than hint at where the problem may be occuring, and does not say much about what the problem could actually be. Sometimes errors would not be reported at all, and would only be discovered at compile or run-time exceptions in eclipse. Other times the synthesizer itself would throw an error, leaving the developer with much exploring to do. Because of this error reporting system, the brunt of the debugging process must be shouldered by the developer. Although this can lead to frustration at first, the process quickly becomes easier as the developer gains experience with the solitaire framework.

Compiler, Client, and Generation Overview

While the instructions above highlight our general development process, throughout the project we created two clients that allow for alternative steps. The first was a custom client-side application capable of loading and running compiled solitaire games. This allows a user to store jar files from compiled solitaire variations and run them at will in a GUI-based, easy-to-use client app, and provides added security by means of a SecurityManager that prevents any solitaire games from making unwanted changes to the client's computer. The goal is that one day this app may be used by end-users to play the solitaire variations created on the Next-Gen Solitaire platform.

The other step we added was a custom compilation tool. We did not want to require a user or developer to have a certain IDE or program installed just to compile our variations, so instead we built another client-side app that can do this programmatically. Additionally, this standardizes the compilation of each variation, allowing us to remove common code (the KombatSolitaire framework, for example) from each variation, which is later included as a dependency when run using the client application. This process allows us to reduce the size of generated jar files, in addition to streamlining the compilation process.

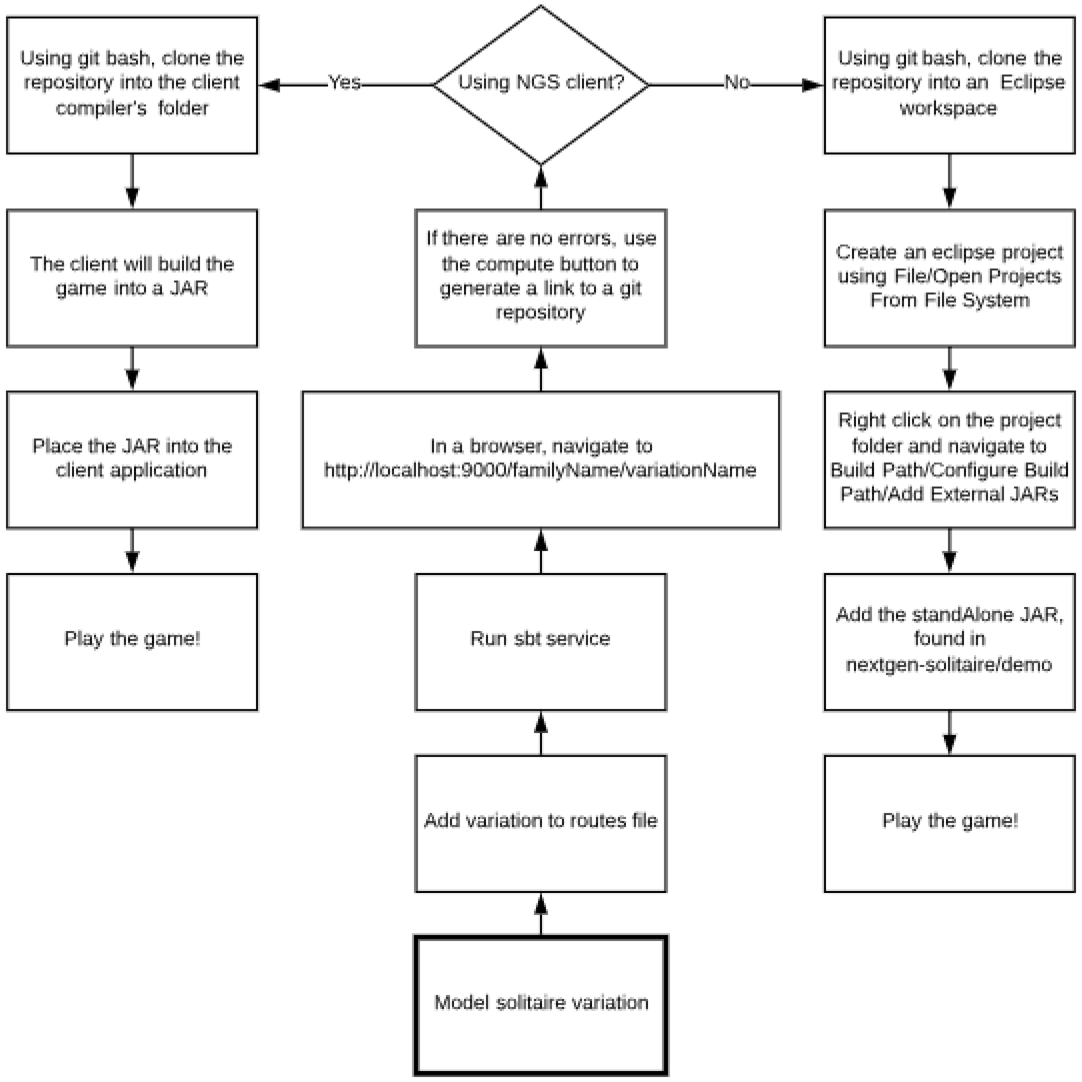

Using git bash, clone the repository into the client compiler's folder
Yes
Using NGS client?
No
Using git bash, clone the repository into an Eclipse workspace
The client will build the game into a JAR
If there are no errors, use the compute button to generate a link to a git repository
Create an eclipse project using File/Open Projects From File System
Place the JAR into the client application
In a browser, navigate to http://localhost:9000/familyName/variationName
Right click on the project folder and navigate to Build Path/Configure Build Path/Add External JARs
Play the game!
Run sbt service
Add the standAlone JAR, found in nextgen-solitaire/demo
Add variation to routes file
Play the game!
Model solitaire variation

Figure 6: Walkthrough of the two different methods of running generated code

Ultimately, the final process for creating a solitaire game start-to-finish once the modeling is complete, is to first run the service in IntelliJ, compute the model on the inhabitation page, clone the git repository into the compiler, compile the project, and, finally, move the generated jar file to the client application. Once done, the game is ready to run. The cloned repository can alternatively be opened as an Eclipse project to allow for further development or testing.

IV Results

In this section, we present the results we have produced throughout the project and discuss the challenges associated with each of them. Going into this project we had no experience with the specific framework, code generation, or Scala. After a short toy example, our work focused on four solitaire families: Fan, Spider, Golf, and Gypsy, as well as Demon, which we experimented with briefly. Each of these families required the exploration of new techniques or additions to the framework.

Base Solitaire Games

At the start of the project, we were given a demonstration of the basics of the framework through Alpha, a toy example in which the player simply deals cards from the stock to the tableau one at a time, winning when the stock is empty. Using the models for Alpha and existing games such as Klondike as reference, we created a toy example of our own that further explored the layout. This example still did not feature any moves, constraints, custom elements, or a starting deal, but allowed us to begin working with the basics of the framework. From there, we started development on our first solitaire families, with each team member developing one family.

Demon (also called Canfield), is a notoriously difficult variation of solitaire with purported origins in American casinos. It quickly presented itself as an equally challenging variation to model. While the framework has a high degree of customizability, some rules that seem logically reasonable can be very difficult to implement. Demon requires a special “demon” pile that had to be modeled as a BuildablePile but function differently in gameplay than the other BuildablePiles that make up the tableau. While future variations included custom piles that dealt with similar issues, this proved far too difficult at the time, and we decided to stop development on Demon.

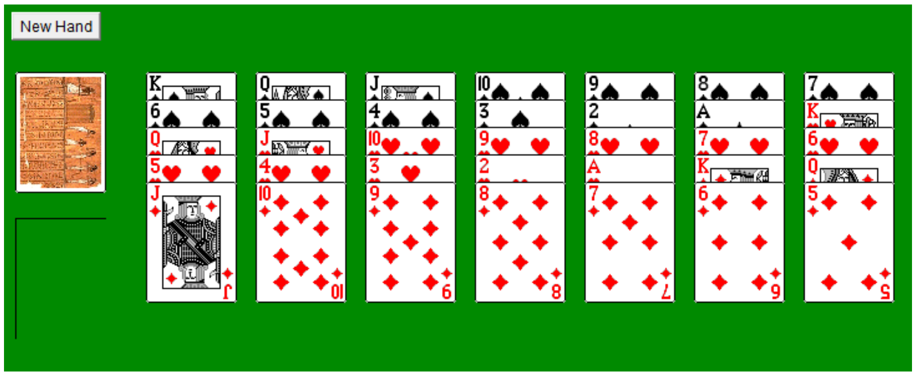


Figure 7: The starting layout of Golf Solitaire

Golf is a relatively easy and fast-paced solitaire game where a player builds a single waste pile from the tableau, adding cards that are either one rank above or below the top card of the pile. With no need for complicated moves or specialized logic, the base version of Golf was very straightforward to model, a welcome change from Demon.

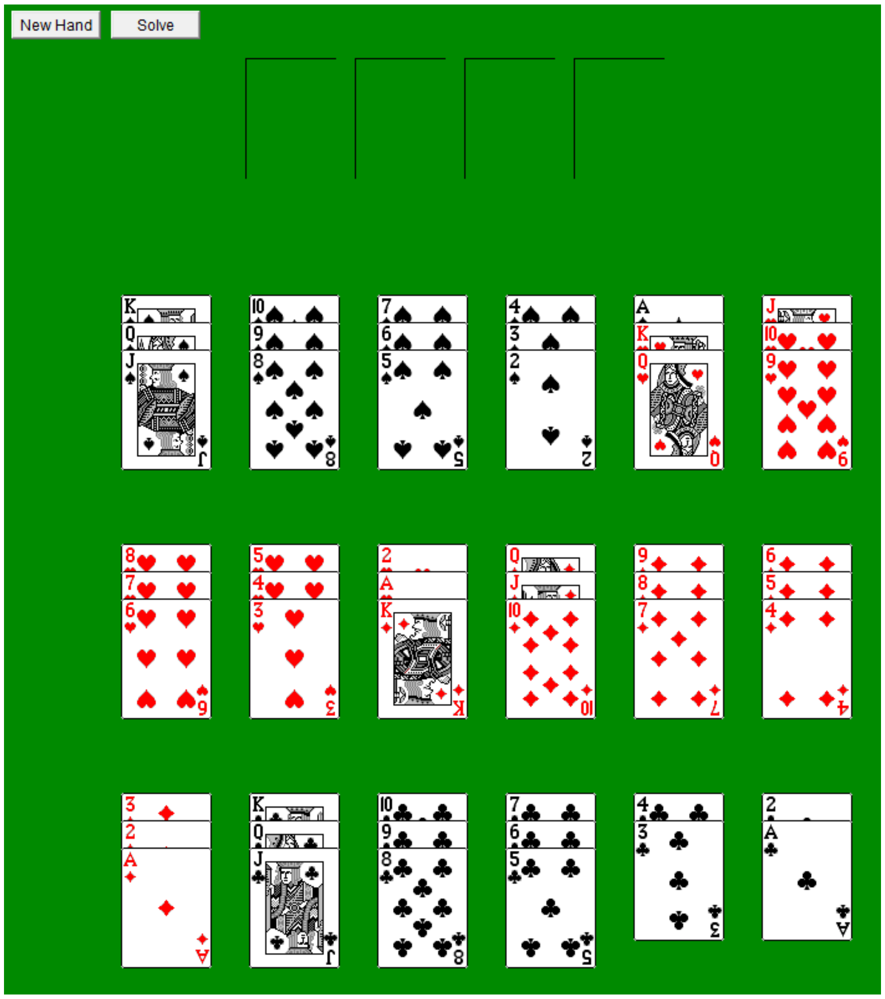


Figure 8: The starting layout for Fan Solitaire

Fan solitaire once again has simple rules but a unique layout. There are 18 tableau piles, each starting with 3 face-up cards except for the last two, which have 2. Players can build down by suit from King on these piles, the goal being to build up from aces by suit on each of the four foundation piles. From a modeling perspective, the biggest challenge with the base version of Fan was creating the deal and layout. The framework has built-in options for creating layouts with one or two rows, but for Fan's tableau each of the 18 piles had to be given custom coordinates, and the starting deal had to be modified to account for this.

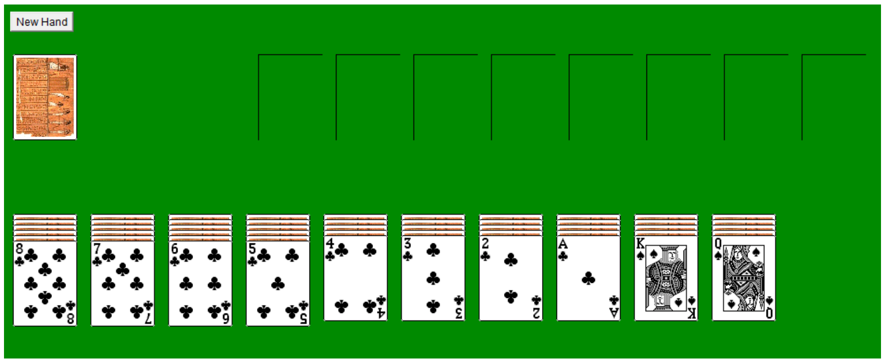


Figure 9: The starting layout of Spider Solitaire

Spider solitaire plays fairly similar to Klondike. Using two decks, 44 cards are dealt across 10 tableau piles. All but the top card of each pile starts face down but are flipped and revealed when they become the top card of their pile. The goal is to build down by suit from king to ace on the tableau before moving the whole sequence to one of the 8 foundation piles. Cards on the tableau can be placed on any other card on the tableau that is one rank higher. A sequence of cards that are in descending rank and have the same suit can be moved as a group. Spider required the usage of Klondike’s custom CP2 controller to handle flip moves as well as the implementation of our first custom constraint, AllSameSuit.

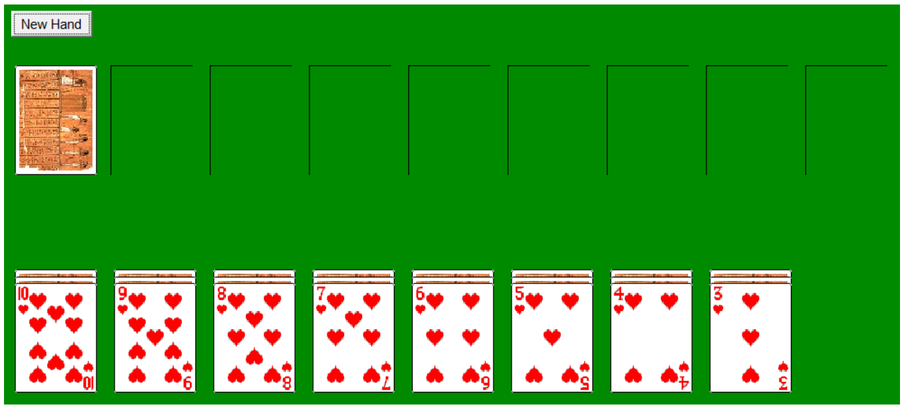


Figure 10: The starting layout of Gypsy Solitaire

Gypsy was the last base solitaire game we developed. Its rules are rather similar to Klondike and Spider in that it also uses face down cards and allows for sequences of cards to be moved as a group. The starting deal uses two decks to create 8 tableau piles of 3 cards with all but the top card of each being face down. You can build down on the tableau in alternating colors. Additionally, you can move any sequence of descending cards in alternating colors. The goal is to build up each of the 8 foundation piles by suit, starting at aces. Unlike many other solitaire games, you can move cards from the foundation back to the tableau. While the base game didn’t require any specialized logic, it does use five moves, which is relatively complicated compared to other base games.

Families and Variations

Throughout their long history, solitaire games have organized themselves into ‘families’, consisting of a base game and variants resulting from tweaks to that game’s rules. For example, there is an easier variation of Spider called Open Spider that starts the game with all cards face up, or a variation of Fan called Shamrocks that relaxes the restrictions on tableau building but places a 3 card limit on each pile. The framework excels at handling these logical tweaks. We were able to implement families by creating a scala trait called variationPoints containing the rules and modeling for the base game of each family. New variations in a family inherit from its variationPoints and override the aspects of the model that they change.

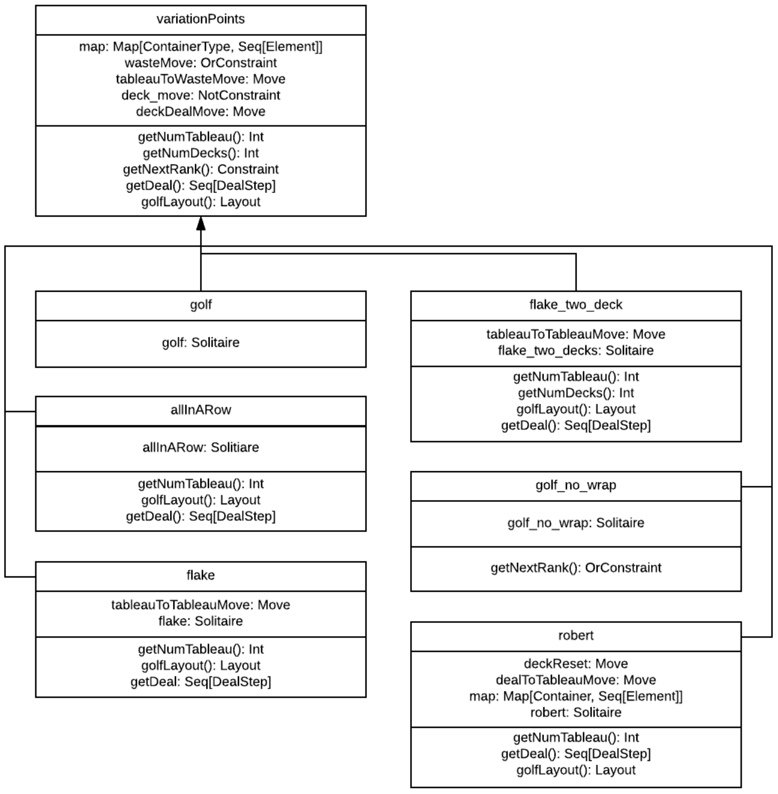
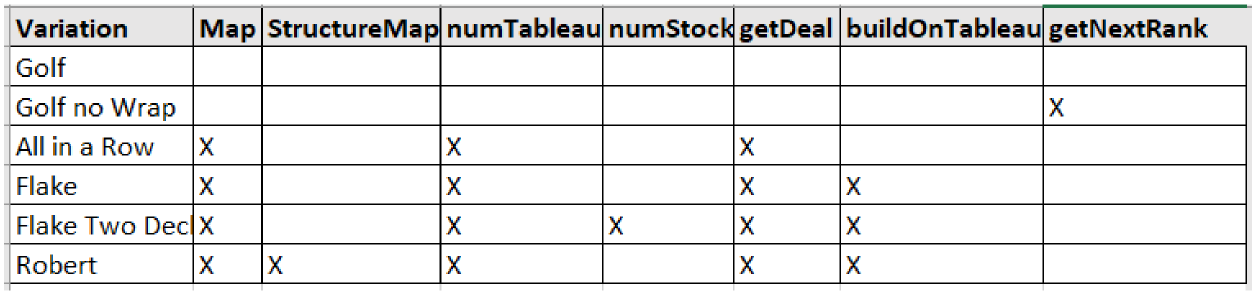


Figure 11: Hierarchy of Golf Variations



Variation	Map	StructureMap	numTableau	numStock	getDeal	buildOnTableau	getNextRank
Golf							
Golf no Wrap							X
All in a Row	X		X		X		
Flake	X		X		X	X	
Flake Two Dec	X		X	X	X	X	
Robert	X	X	X		X	X	

Figure 12: Variation Points and Custom Logic of Golf Variations

For Golf, we created five variations: Flake, Flake Two Decks, All in a Row, Golf no Wrap, and Robert. These variations were relatively simple and easy to implement, though some did require a specialized waste pile. Other than that, variations required only small adjustments to the base game.

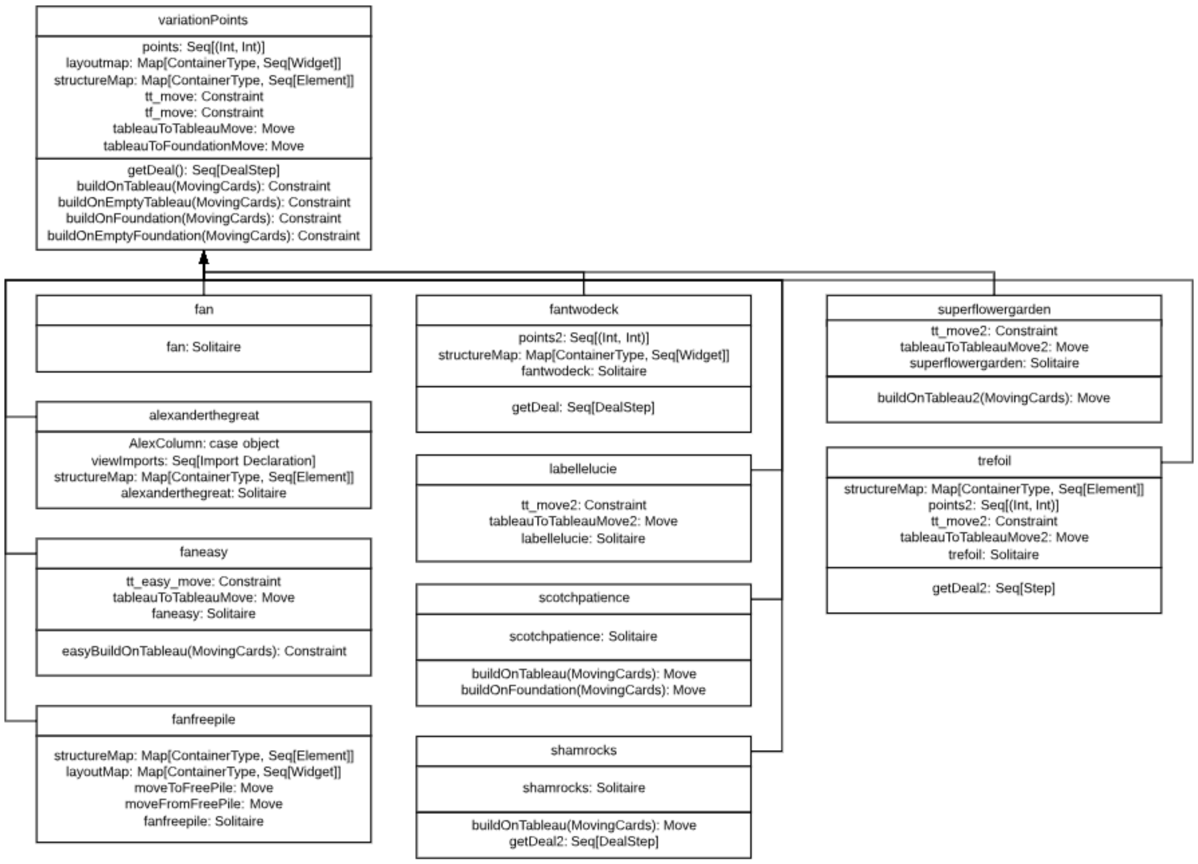
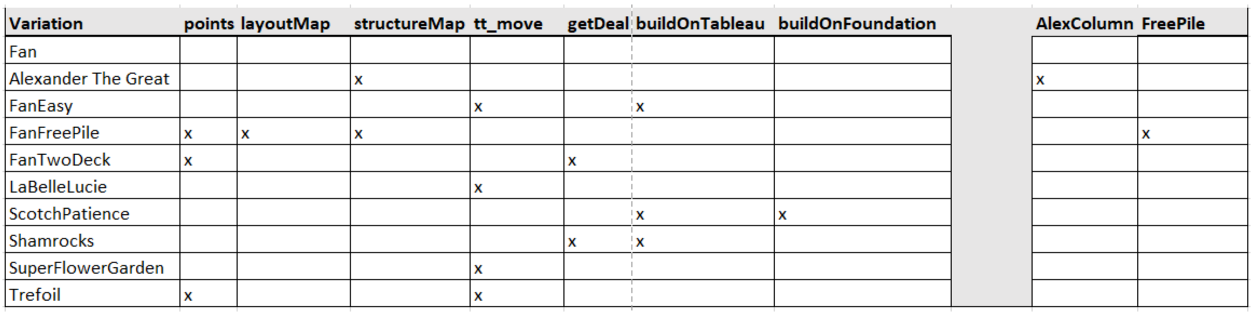

variationPoints
points: Seq[(Int, Int)]
layoutmap: Map[ContainerType, Seq[Widget]]
structureMap: Map[ContainerType, Seq[Element]]
tt_move: Constraint
tf_move: Constraint
tableauToTableauMove: Move
tableauToFoundationMove: Move
getDeal(): Seq[DealStep]
buildOnTableau(MovingCards): Constraint
buildOnEmptyTableau(MovingCards): Constraint
buildOnFoundation(MovingCards): Constraint
buildOnEmptyFoundation(MovingCards): Constraint
fan
fan: Solitaire
fantwodeck
points2: Seq[(Int, Int)]
structureMap: Map[ContainerType, Seq[Widget]]
fantwodeck: Solitaire
getDeal: Seq[DealStep]
superflowergarden
tt_move2: Constraint
tableauToTableauMove2: Move
superflowergarden: Solitaire
buildOnTableau2(MovingCards): Move
alexanderthegreat
AlexColumn: case object
viewImports: Seq[Import Declaration]
structureMap: Map[ContainerType, Seq[Element]]
alexanderthegreat: Solitaire
labellelucie
tt_move2: Constraint
tableauToTableauMove2: Move
labellelucie: Solitaire
trefoil
structureMap: Map[ContainerType, Seq[Element]]
points2: Seq[(Int, Int)]
tt_move2: Constraint
tableauToTableauMove2: Move
trefoil: Solitaire
getDeal2: Seq[Step]
faneasy
tt_easy_move: Constraint
tableauToTableauMove: Move
faneasy: Solitaire
easyBuildOnTableau(MovingCards): Constraint
scotchpatience
scotchpatience: Solitaire
buildOnTableau(MovingCards): Move
buildOnFoundation(MovingCards): Move
fanfreepile
structureMap: Map[ContainerType, Seq[Element]]
layoutMap: Map[ContainerType, Seq[Widget]]
moveToFreePile: Move
moveFromFreePile: Move
fanfreepile: Solitaire
shamrocks
shamrocks: Solitaire
buildOnTableau(MovingCards): Move
getDeal2: Seq[DealStep]

Figure 13: Hierarchy of Fan Variations



Variation	points	layoutMap	structureMap	tt_move	getDeal	buildOnTableau	buildOnFoundation		AlexColumn	FreePile
Fan										
Alexander The Great			x						x	
FanEasy				x		x				
FanFreePile	x	x	x							x
FanTwoDeck	x				x					
LaBelleLucie				x						
ScotchPatience						x	x			
Shamrocks					x	x				
SuperFlowerGarden				x						
Trefoil	x			x						

Figure 14: Variation Points and Custom Logic of Fan Variations

We implemented nine additional variations for Fan: Alexander the Great, Fan Easy, Fan Free Pile, Fan Two Deck, La Belle Lucie, Scotch Patience, Shamrocks, Super Flower Garden, and Trefoil. Alexander is most noteworthy among these. It uses a large amount of specialized logic to handle a move that, once per game, allows a player to move a card from the middle of a pile as if it were on top. Neither the ability to select an individual card from the middle of a pile nor the idea of an action that can only be taken once per game are present in other games we have, so implementing this move took considerable effort and experimentation. With this new ground broken, we now have a reference point for similar moves in the future.

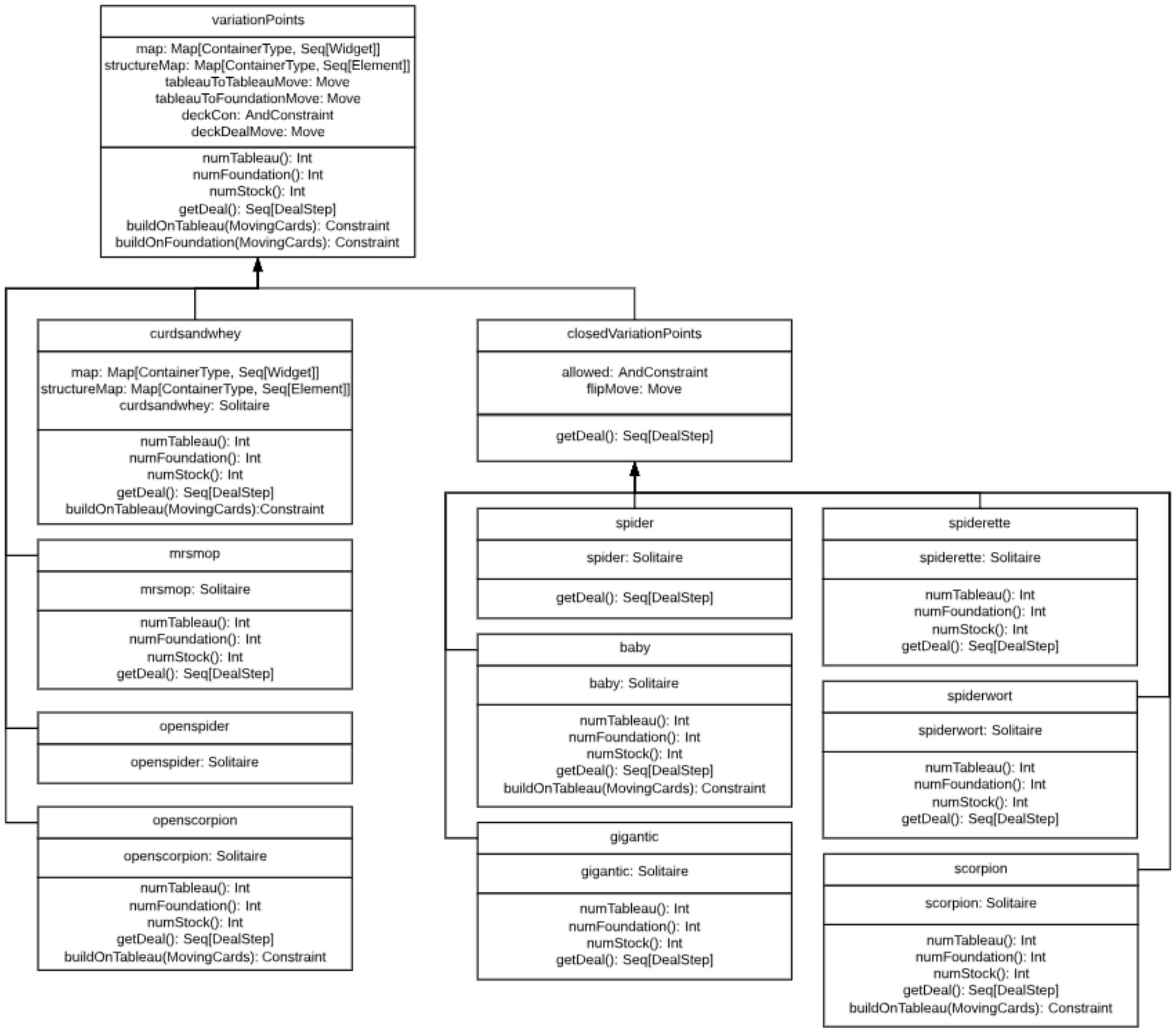
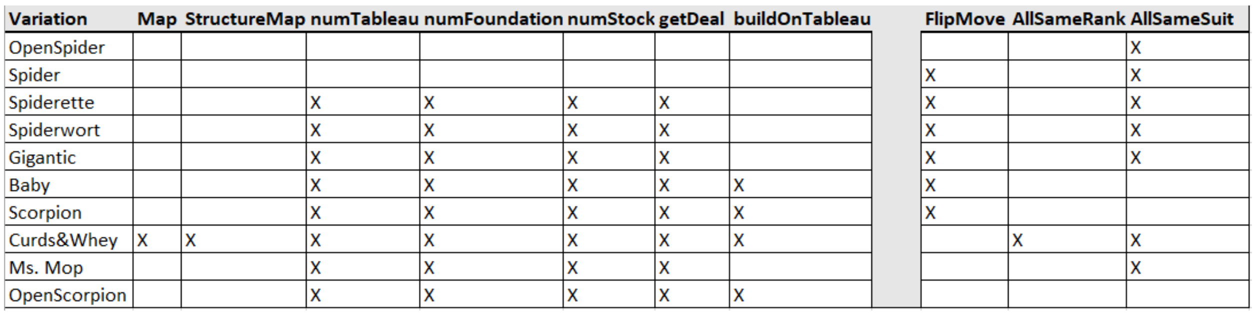


Figure 15: Hierarchy of Spider Variations



Variation	Map	StructureMap	numTableau	numFoundation	numStock	getDeal	buildOnTableau		FlipMove	AllSameRank	AllSameSuit
OpenSpider											X
Spider									X		X
Spiderette			X	X	X	X			X		X
Spiderwort			X	X	X	X			X		X
Gigantic			X	X	X	X			X		X
Baby			X	X	X	X	X		X		
Scorpion			X	X	X	X	X		X		
Curds&Whey	X	X	X	X	X	X	X			X	X
Ms. Mop			X	X	X	X					X
OpenScorpion			X	X	X	X	X				

Figure 16: Variation Points and Custom Logic of Spider Variations

For Spider we implemented nine variations: Spiderette, Spiderwort, Gigantic, Baby, Scorpion, Open Spider, Open Scorpion, Mrs. Mop, and Curds & Whey. These variations each further explored the modeling techniques used in the base Spider, but beyond that there weren't any variations that required extensive custom development. However, Spider did prompt us to explore the possibility of subfamilies to separate the variations that used face down cards from those that didn't. In this case, a second Scala trait, closedVariationPoints, inherited from Spider's variationPoints and implemented the logical tweaks that come with face down cards. Variations that used face down cards then inherited from this trait. In this manner, one could create build a hierarchy of subfamilies off a base game to further separate variations.

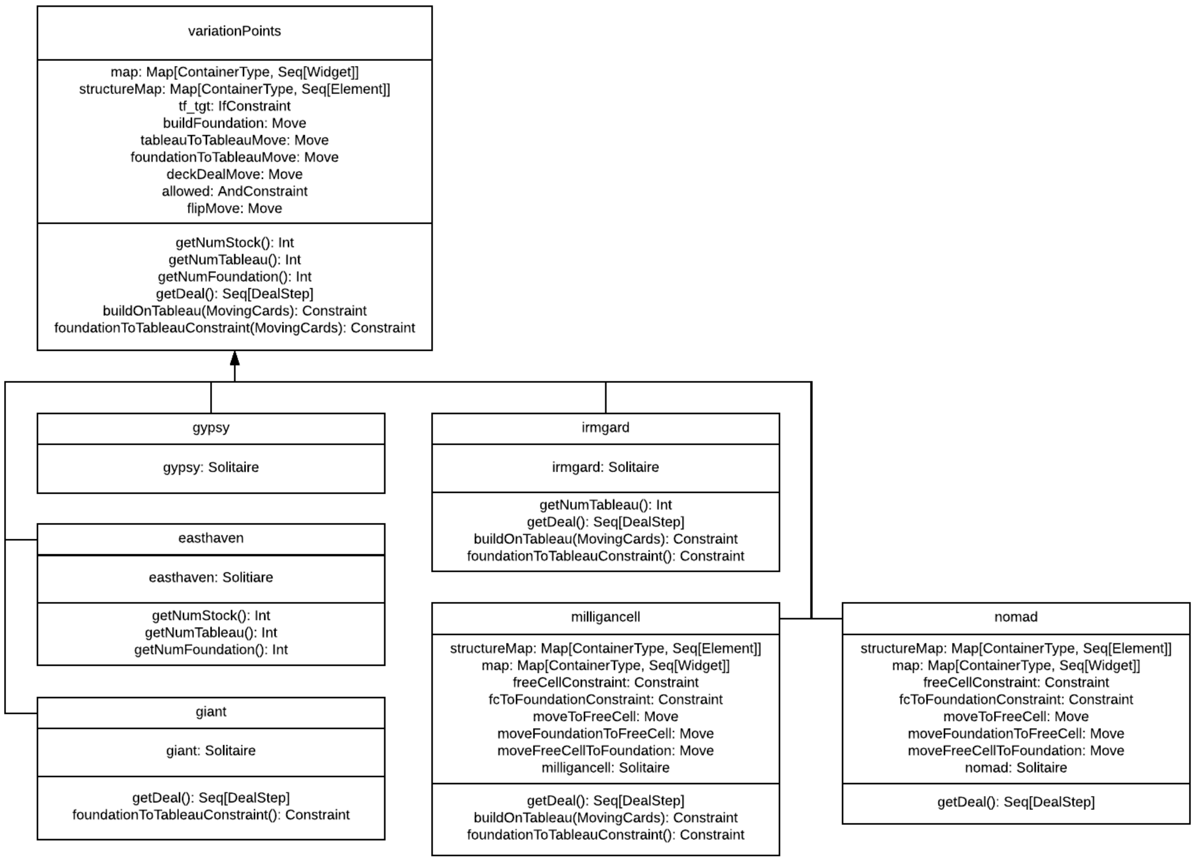
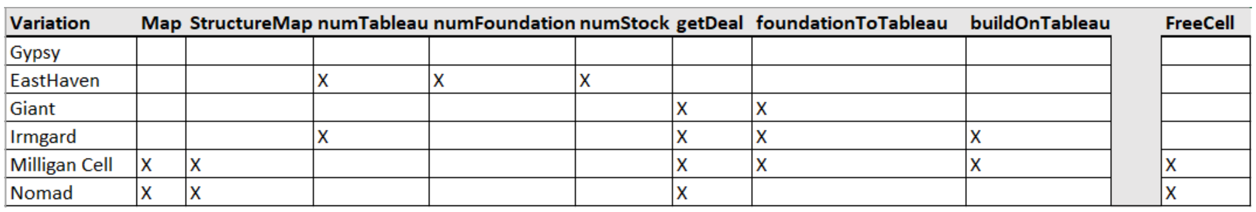

variationPoints
map: Map[ContainerType, Seq[Widget]]
structureMap: Map[ContainerType, Seq[Element]]
tf_tgt: IfConstraint
buildFoundation: Move
tableauToTableauMove: Move
foundationToTableauMove: Move
deckDealMove: Move
allowed: AndConstraint
flipMove: Move
getNumStock(): Int
getNumTableau(): Int
getNumFoundation(): Int
getDeal(): Seq[DealStep]
buildOnTableau(MovingCards): Constraint
foundationToTableauConstraint(MovingCards): Constraint
gypsy
gypsy: Solitaire
irmgard
irmgard: Solitaire
getNumTableau(): Int
getDeal(): Seq[DealStep]
buildOnTableau(MovingCards): Constraint
foundationToTableauConstraint(): Constraint
easthaven
easthaven: Solitaire
getNumStock(): Int
getNumTableau(): Int
getNumFoundation(): Int
giant
giant: Solitaire
getDeal(): Seq[DealStep]
foundationToTableauConstraint(): Constraint
milligancell
structureMap: Map[ContainerType, Seq[Element]]
map: Map[ContainerType, Seq[Widget]]
freeCellConstraint: Constraint
fcToFoundationConstraint: Constraint
moveToFreeCell: Move
moveFoundationToFreeCell: Move
moveFreeCellToFoundation: Move
milligancell: Solitaire
getDeal(): Seq[DealStep]
buildOnTableau(MovingCards): Constraint
foundationToTableauConstraint(): Constraint
nomad
structureMap: Map[ContainerType, Seq[Element]]
map: Map[ContainerType, Seq[Widget]]
freeCellConstraint: Constraint
fcToFoundationConstraint: Constraint
moveToFreeCell: Move
moveFoundationToFreeCell: Move
moveFreeCellToFoundation: Move
nomad: Solitaire
getDeal(): Seq[DealStep]

Figure 17: Hierarchy of Gypsy Variations



Variation	Map	StructureMap	numTableau	numFoundation	numStock	getDeal	foundationToTableau	buildOnTableau		FreeCell
Gypsy										
EastHaven			X	X	X					
Giant						X	X			
Irmgard			X			X	X	X		
Milligan Cell	X	X				X	X	X		X
Nomad	X	X				X				X

Figure 18: Variation Points and Custom Logic of Gypsy Variations

We created five variations for gypsy: Easthaven, Giant, Irmgard, Milligan Cell, and Nomad. As Gypsy's base game is relatively complicated to begin with, most of the variations don't make significant changes. Milligan Cell and Nomad, however, require the implementation of a free pile and several additional moves to interact with it. With that in mind, these variations have the most moves of any game we've implemented.

We immediately felt the benefits of designing around families. In addition to categorizing the different versions of solitaire, inheriting from a base's Scala trait allows a modeler to easily implement variations. Many solitaire variations result from minor changes to a small number of rules or the initial deal, so overriding variables as needed is the perfect solution. Furthermore, a modeler could invent their own variations by viewing variationPoints as an overview of the different game elements and overriding one.

Development Time Across Game Types

One of the major initial goals of the framework was to reduce the amount of duplicate code that has to be rewritten when implementing similar programs. In the scope of our project, there are three types of games: a standalone game, a game that is going to start a family of variations, and a variation built onto an existing family. Once the framework has been set up, a standalone game that doesn't require complicated specialized logic can be implemented in as little as 30 minutes by an experienced modeler. One that starts, a family requires the modeler to consider future variations in designing variationPoints and as such may require slightly more time, around 40 minutes. In both cases, the modeler must create and set up the family Scala file, the game domain file, and the controllers file for the game. Variations can be built off an existing family in as little as a few minutes, depending on how extensive the changes are. Gigantic, for example, a spider variation that simply doubles the size of the deck and changes the layout accordingly, took only 5 minutes to model.

Games or variations with specialized logic will require additional development time, depending on the complexity of the logic and how much it deviates from ordinary solitaire rules. These can take anywhere from half an hour to several hours to develop. For an easy example, we have AllSameSuit from the spider family, which takes in a Stack of cards and returns true if they all share the same suit. As it is relatively simple to iterate through a list of cards and check their suits, it only took around 20 minutes to implement. On the other hand, the special move in Alexander, as described above, took several hours to perfect. Each new piece of logic provides precedence for future, similar, variations, so specialized logic development in general should become easier over time.

Generation time among solitaire games varied slightly, but doesn't appear to be tied to model complexity, family, or specialized logic; it is likely just natural fluctuations. The generation times observed for all variations are less than 0.1 seconds, so despite these fluctuations they are all generated rather quickly. The Fan variations, for example, have times ranging from 0.015s to 0.032s, with the average being 0.026s. In practice, these times are bookended by actions in the Inhabitation page and git bash, but overall this is negligible compared to the amount of time the framework saves in other ways.

Life as a Modeler

As this was our first time working in a code generation framework, working as a modeler as opposed to a programmer was a new experience for us. It was an easy transition, however, as both use very similar skills. For example, the general debugging process when a game didn't generate correctly was very much the same as in ordinary programming. Additionally, non-modeling aspects of the work, such as implementing specialized logic and writing test cases, were the same as normal.

Working as a modeler involved combining the strengths of multiple programming languages. To begin with, learning to design the models was rather similar to learning a new language. Models have their own syntax and requirements for proper generation, and designing them follows the same workflow as normal programming. Furthermore, the inhabitation and generation steps of the framework could be viewed as a form of compiler. With the framework configured toward producing solitaire games as far as the scope of our project is concerned, implementing the models felt like working in a high-level, solitaire-specific language, albeit one with scala syntax. In addition to the pseudo-language of the models, we worked with the Scala code of the framework itself, the Java code of the specialized logic and generated code, and small amounts of Python code during experiments with PySol.

Our close collaboration with Professor Heineman was an integral part of our experience, and a unique one as well. While we can work through problems involving Scala or Java with research as one might in normal programming, there aren't at the time any outside resources on the framework. As a result, when it came to these framework-specific issues we were mostly left to work through them as a team. Professor Heineman, as the chief developer of the framework, proved an invaluable resource when we were personally breaking new ground. His familiarity with the framework allowed him to make tweaks to the more obscure parts of the framework. The back and forth between the professor and our team, passing feedback and requests for the framework and models in both directions, proved an interesting experience. The feeling that we were relatively on our own and breaking new ground was unique, and it was great to be able to work so closely with the creator of the framework.

V Conclusion & Recommendations

Our project aimed to determine and evaluate the effectiveness of code generation via combinatory logic synthesis (CLS) as a software development technique. To that end, we became developers of a CLS platform and monitored the evolution of our coding process. By the end of the project, we were not only comfortable enough with the environment to contribute to the development of the framework itself. Along the way we modeled 34 solitaire games, from which we generated over 22,000 lines of working Java code. Compared to the often long learning period for new developers in a production environment, the few months we spent on this project showed significant promise. Although the solitaire framework still has a long way to go, our project serves as a testament to the success of its goal: demonstrating the effectiveness of CLS in software development.

Recommendations for Next-Gen Solitaire

Over the course of our project work we encountered several issues with the Next-Gen Solitaire framework. While we worked through many with Professor Heineman and were able to deal with more on our own as we gained experience, there are still several changes that could be made that would benefit framework developers:

- The framework could infer which controllers and handlers are necessary based on the layout and moves of a game's model. Alternatively, the framework could have default handlers. Currently, developers need to manually set event handlers for piles even in the cases where moves do not interact with them.
- The Inhabitation page, where the controllers are organized before the code is generated, could detect a wider variety of errors and provide more specific feedback. Currently, for each element of the game, users are warned if their controllers are ambiguous or incomprehensive. If that is the case, the page should provide more information on the exact cause of the error. Furthermore, the page should display errors that arise during the computation. Both of these changes would help make debugging more straightforward.
- Compile time could be decreased. Currently some variations require a minute or more of load time before the Inhabitation page is displayed. A large contributing factor is that the page generates all games in the routes file before displaying.
- Some built-in logic could be made more flexible. One example being: moves that deal cards from the deck deal a fixed, specified amount, which often is acceptable but causes errors if there aren't that many cards in the stock. Dealing *up to* that many cards instead would prevent these issues.
- A more straightforward way to implement specialized logic. The ability to write and include specialized logic is one of the biggest strengths of Next-Gen Solitaire. Currently, you have to write code within a string, which the framework will include as Java code post-generation. This prevents developers from utilizing the benefits of their IDE and increases the likelihood that syntax errors or typos will go unnoticed.

- Changing the Java code for the containers to allow direct access to their fields would also help specialized logic development. Currently, if you want to get information from a pile or column, like how many cards it contains, you need a helper function to fetch that info.
- The ability to create dynamic or relative layouts, rather than specifying coordinates or using pre-made formations, for the solitaire games.
- Triggers or state-based actions with conditions, such as a face down card flipping face up when it becomes the top card of a pile.
- Documentation on the variables and functions of the game elements, such as the Cards, in their generated Java form would help in the development of specialized logic.

These small shortcomings do not change the fact that Next-Gen Solitaire is an excellent demonstration of Code Generation and CLS. For the most part, these issues are quality of life changes that would ease the framework's learning curve or make development of new variations and specialized logic smoother. While we were unable to tackle these issue during our project work, we believe that they would be worth looking into for anyone interested in improving the framework.

Recommendations for Generation and CLS Usage

Generation via Combinatory Logic Synthesis has a variety of potential uses outside of Next-Gen Solitaire. It is appropriate to be used anywhere that has many different models with similar structure and functions but some slight differences. This is especially useful when you only want some of the code loaded into memory at a given time. This paradigm should not be used for one and done programs where the goal is straightforward or objects have little to no natural hierarchical structure.

For example, game engines are a great place to use this paradigm. Games often have different levels, AI players, and items which all are similar but feature differences. Games will often only load a level and the elements needed for that level, so having each one generated separately could greatly decrease load time and boost performance. In addition, generation means that the game could be easily created in different programming languages which are often required to port a game to different platforms. We strongly encourage others to further explore CLS and Code Generation in their own future projects and endeavors.

References

Bessai, J., Dudenhefner, A., Düdder, B., Martens, M., and Rehof, J. (2014), “Combinatory Logic Synthesis,” 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA), LNCS, vol. 8802, Springer, pp. 26–40.

Butler, G. (2002), “OO Frameworks”, 18th European Conference on Object-Oriented Programming (ECOOP), Tutorial.

Fayad, M. and Schmidt, D. (1997), “Object-Oriented Frameworks”, Guest editorial, Communications of the ACM, special issue on Object-oriented Frameworks, 40(10).

Heineman, G., Hoxha, A., Düdder, B., and Rehof, J. (2015). “Towards migrating object-oriented frameworks to enable synthesis of product line members”. In Proceedings of the 19th International Conference on Software Product Line (SPLC '15). pp. 56-60.

Heineman, G., Bessai, J., Düdder, B., Rehof, J. (2018), “Towards Language-independent Code Synthesis”, ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM), Poster.

Heineman, G., Bessai, J., Düdder, B., “Next-Gen Solitaire Code Generation Framework”, https://github.com/combinators/nextgen-solitaire, March 2019.

Pysol FC (2019), “PySol Fan Club”, https://pysolfc.sourceforge.io

Yates, Rebecca Yolande (2014), “Onboarding in software engineering”. Retrieved from http://hdl.handle.net/10344/4272

Appendix A: Solitaire Creation Tutorial

I. Getting started (Setup work environment)
   A. Install IDE
      1. Install IntelliJ making sure to add the Scala Tools when prompted
      2. Install latest Java eclipse
   B. Configure IntelliJ
      1. In the top right corner click the drop down and go to edit configurations
      2. Choose to add a new SBT Task
      3. Change the tasks field to be just “run”
      4. Rename the command if desired and save and close configuration
      5. Got to File>install settings and select the IntelliJSettings.jar in the demo folder
   C. Test/Run Service
      1. Run the service you created in B
      2. In a web browser go to http://localhost:9000/narcotic (you can replace narcotic with any existing variations, note variations in families include their family name, such as http://localhost:9000/fan/shamrocks
      3. Wait for the page to load (may take up to a minute)
      4. Press the compute button on the web page
      5. Copy the git clone line from the git tab that appears when you compute
      6. Open a terminal and navigate to a folder where you want to store your variations
      7. Run the git clone command
      8. Open Eclipse and import or link the generated code. If you are linking make sure to link git\VARIATION-NAME\src\main\java
      9. Link the standalone.jar found in demo/ folder of the generating-solitaire project
      10. The code should be runnable and allow you to run any downloaded variations

II. Example: Modeling/Starting a Family - Spider
   A. Rules
      1. This example will walk you through starting a spider solitaire “family” of variations, from rules to modeling.
      2. Though this will discuss starting a solitaire family, making a singular/standalone game only requires minor changes in these steps.
      3. The rules we will be using for spider are:
         a) Spider uses two decks of cards in a single stock
         b) The initial deal consists of dealing 5 face-down cards to the first four piles of the tableau, 4 face-down cards to the remaining 6, and finally one face-up card to each pile.
         c) There is a foundation consisting of 8 piles

        d) The goal is to create sequences of the same suit and descending rank, which are then moved to the foundation. You win when you've done this eight times and filled the foundation.
        e) You can build on the tableau by moving singular cards, or by moving a stack of 2+ cards that are in descending rank and have the same suit.
        f) As long as you are following the above rule, you may place card(s) onto an empty tableau space or another card with rank 1 higher than the bottom-most moving card.
        g) While there are cards left in the stock, you may deal one card to each tableau pile if you get stuck

B. Setup
    1. Begin by getting the files you need in order.
    2. In src/main/scala/org.combinators/solitaire/ create a folder named "spider"
    3. Right click on that folder and use the templates in New to create one each of SolitaireController, SolitaireGameDomain, SolitairePackageObject, and SolitaireTargetFamily
    4. In preparation for making it a family of games, also make a Scala trait class called variationPoints.
        a) This is where the most common or "default" settings for the family will be. As the founder of the family, most of the spider modeling will be done here.
        b) You'll need: `import org.combinators.solitaire.domain._`

C. Modeling in variationPoints
    1. Declare custom constraints
        a) `case class AllSameSuit(movingCards: MoveInformation) extends Constraint`
        b) Spider uses the custom constraint AllSameSuit. We'll cover implementing that later, but first you'll need to declare it here at the top of the model.
    2. Structure Map
        a)
```
val structureMap:Map[ContainerType,Seq[Element]] = Map(
 Tableau -> Seq.fill[Element](10)(BuildablePile),
 Foundation -> Seq.fill[Element](8)(Pile),
 StockContainer -> Seq(Stock(numStock))
)
```
        b) The first of two maps that determine our setup and board. For each type of container your game will use, you need to include, it, specify how many there are, and what type they are.
        c) Spider requires a Tableau to build on, a Foundation to move completed stacks to, and a StockContainer to deal extra cards from.
        d) Spider has 10 piles in its Tableau and 8 in its Foundation. Additionally, numStock (declared outside of this screenshot) = 2 since we have two decks.

e) BuildablePiles look differently from Piles, and allow us to see cards beyond the one on top of the stack assuming they're face-up.

f) Using different container types for the Tableau and Foundation also makes it easier to specify differences in how we are able to interact with them.

3. Layout Map

a)
```
val map:Map[ContainerType, Seq[Widget]] = Map (
 Tableau -> horizontalPlacement(15, 200, 10, 13*card_height),
 StockContainer -> horizontalPlacement(15, 20, 1, card_height),
 Foundation -> horizontalPlacement(293, 20, 8, card_height)
)
```

b) The second of our maps. For each container specified in the structure map that you want to be visible while playing the game, you need to give it a placement.

c) In Spider, we want every container to be visible, so we will have to define placements for each of them

d) The traditional layout is a row for the Tableau, a row for the Foundation, and the Stock in the top left. We can use horizontalPlacement to achieve that.

e) Card_height allows us to make sure that the containers are large enough to display full cards

4. Deal

a)
```
def getDeal: Seq[DealStep] = {
 var colNum:Int = 0
 var dealSeq:Seq[DealStep] = Seq()// doesn't like me declaring it without
initializing
 //first four piles get 5 face down cards
 for (colNum <- 0 to 3) {
   dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum),
Payload(faceUp = false, numCards = 5))
 }
 //the rest get 4 face down cards
 for (colNum <- 4 to 9) {
   dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum),
Payload(faceUp = false, numCards = 4))
 }
 //each pile gets a face up card
 colNum = 0
 for (colNum <- 0 to 9) {
   dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum),
Payload(faceUp = true, numCards = 1))
 }
 dealSeq
}
```

b) We need to specify a deal to start our game off with. While some games allow you to uniformly deal out cards, many have irregularities that necessitate individually targeting containers in the Tableau.

c) It's fairly straightforward what's going on here. In spider we first deal 5 face-down cards to the first 4 piles. Then we deal 4 face-down cards to the remaining 6. Finally, we deal a single face-up card to each pile.

5. Constraints

```
def buildOnTableau(cards: MovingCards.type): Constraint = {
   val topDestination = TopCardOf(Destination)
   val bottomMoving = BottomCardOf(cards)
   val isEmpty = IsEmpty(Destination)
   val descend = Descending(cards)
   val suit = AllSameSuit(cards)
   AndConstraint( AndConstraint(descend, suit), OrConstraint(isEmpty,
NextRank(topDestination, bottomMoving, true)) )
}
```

a) This is our first noticeable difference between implementing a standalone game and a variation family.
b) Normally you don't need constraints to be produced by functions and can just code them as variables/values, but since we will be dealing with inheritance when we add to the family, having them produced by functions makes them easier to override.
c) This constraint will be used to govern what we're allowed to do when moving cards from one tableau pile to another.
d) Returning to our above rules:
   (1) We can place any card on an empty space
   (2) We can move 2+ cards as a group if they are all the same rank and are in descending order
   (3) We can place card(s) onto an occupied pile if the bottom-most moving card is one rank below the pile's top-most card
e) The first half of the AndConstraint is yet another, nested, AndConstraint which checks that our card(s) are in descending order and are all the same suit (a single card returns true for both)
f) The second half is an OrConstraint that confirms that either our target pile is empty, or its top card has an acceptable rank
g)
```
def buildOnFoundation(cards: MovingCards.type): Constraint = {
   val topMoving = TopCardOf(cards)
   val bottomMoving = BottomCardOf(cards)
   val descend = Descending(cards)
   val suit = AllSameSuit(cards)
   AndConstraint( AndConstraint(descend, suit),
AndConstraint(IsAce(topMoving), IsKing(bottomMoving)) )
}
```
h) We have a similar setup for the constraint that governs when we can move to the foundation.
i) Recalling the above rules, we need to have a full stack of King->Ace in descending rank and matching suit.
j) The first half of the AndConstraint is another And that checks once again that we are in descending rank and have the same suit for our moving cards.
k) The second half is yet another And, this time checking that the top card of our moving cards is an Ace and the bottom card is a King. If we are in descending order and the same suit, this is only possible if we have a proper full stack.

6. Moves
   a)
```
val tableauToTableauMove:Move = MultipleCardsMove("MoveColumn", Drag,
 source=(Tableau,Truth), target=Some((Tableau, buildOnTableau(MovingCards))))
```

```
val tableauToFoundationMove:Move = MultipleCardsMove("MoveCardFoundation", Drag,
 source=(Tableau,Truth), target=Some((Foundation, AndConstraint(
IsEmpty(Destination), buildOnFoundation(MovingCards)))))
val deckDealMove:Move = DealDeckMove("DealDeck", 1,
 source=(StockContainer, NotConstraint(IsEmpty(Source))), target=Some((Tableau,
Truth)))
val allowed = AndConstraint(NotConstraint(IsEmpty(Source)),
NotConstraint(IsFaceUp(TopCardOf(Source))))
val flipMove:Move = FlipCardMove("FlipCard", Press, source = (Tableau, allowed))
```

b) Now that we have our constraints, all that's left to do is put them into moves.
c) Since we can move one or more cards at a time, our tableau and foundation moves are MultipleCardMoves and are Drag type.
d) For both, the source is the Tableau. "Truth" specifies that we are allowed to attempt to start a move no matter what. You can put conditions on it, but currently that can cause issues with MultipleCardMoves.
e) The target for the first is the Tableau, and it checks that our MovingCards follow the buildOnTableau constraint/function we declared earlier.
f) The target for the second is the Foundation. In addition to checking that it follows our buildOnFoundation constraint/function, we throw in one last And to make sure that the Foundation pile we're targeting is empty.
g) Finally, we have a DealDeckMove. This allows us to take leftover cards from the stock and deal them to the tableau.
h) We only want to deal 1 card to each pile
i) We also want to use a NotConstraint to make sure that we aren't trying to grab cards from an empty deck.
j) It's important that you name this move DealDeck, in order to make sure the generated Java code references it properly.

D. Modeling in Package

1.
```
package object spider extends variationPoints {
 val spider:Solitaire = {
   Solitaire(name = "Spider",
     structure = structureMap,
     layout = Layout(map),
     deal = getDeal,
     specializedElements = Seq.empty,
     moves = Seq(tableauToTableauMove, tableauToFoundationMove, deckDealMove,
flipMove),
     logic = BoardState(Map(Foundation -> 104)),
     solvable = false
   )
 }
}
```
2. As the "parent" of our family, we don't need to do much for spider now besides open up its package object and declare the variable, see above.
3. Make sure you extend variationPoints in each of the packages in the family

E. Controllers

1. For the most part, you won't need to edit the controllers much beyond the initial template.
2. However, spider requires an extra controller to handle the move that flips face-down cards.
    a) This is necessary due to the ambiguity associated with pressing on a pile. Without extra specifications, the framework isn't sure if you want to flip the card or start a drag move
3. These special controllers are very much a case-by-case basis, but the following extends Buildable Piles to be able to handle this special case.
    a)
```
object buildablePilePress {
 val buildablePile1:Type = 'BuildablePile1 //Changed from :Constructor to
:Type
 class CP2() {
   def apply(): (SimpleName, SimpleName) => Seq[Statement] = {
     (widget, ignore) =>
       Java(s"""|BuildablePile srcPile = (BuildablePile)
src.getModelElement();
                |
                |// Only apply if not empty AND if top card is face down
                |if (srcPile.count() != 0) {
                |  if (!srcPile.peek().isFaceUp()) {
                |    Move fm = new FlipCard(srcPile, srcPile);
                |    if (fm.doMove(theGame)) {
                |      theGame.pushMove(fm);
                |      c.repaint();
                |      return;
                |    }
                |  }
                |}""".stripMargin).statements()
   }
   val semanticType: Type =
     drag(drag.variable, drag.ignore) =>: controller (buildablePile1,
controller.pressed)
 }
 class ChainBuildablePileTogether extends
ParameterizedStatementCombiner[SimpleName, SimpleName](
   drag(drag.variable, drag.ignore) =>: controller(buildablePile1,
controller.pressed),
   drag(drag.variable, drag.ignore) =>: controller(buildablePile,
controller.dragStart),
   drag(drag.variable, drag.ignore) =>: controller(buildablePile,
controller.pressed))
}
```
    b) You add it similar to other controllers
    c)
```
Updated = updated.addCombinator (new buildablePilePress.CP2())
```

F. Custom Constraints
1. Depending on your solitaire game, you'll likely need some number of constraints that aren't included in the base framework.
2. For example, Spider and variants need to use AllSameSuit, a constraint that checks one or more moving cards and returns true if they all have the same suit.
3.
```
@combinator object ExtraMethods {
 def apply(): Seq[MethodDeclaration] = {
   Java(s"""|public boolean allSameSuit(Stack col) {
            | if(col.empty() || col.count() == 1) { return true; }
            | else{
            |   Card c1, c2;
            |   int size = col.count();
            |   for(int i = 1; i < size; i++){
            |    c1 = col.peek(i - 1);
```

```
                  |     c2 = col.peek(i);
                  |     if(c1.getSuit() != c2.getSuit()) { return false; }
                  |    }
                  |   return true;
                  | }
|}""".stripMargin).classBodyDeclarations().map(_.asInstanceOf[MethodDeclaration])
 }
 val semanticType: Type = game(game.methods)
}
```

4. This and any other custom constraints should be added along with the other combinators in the gameDomain file

G. Child Variations
   1. Variations that involve slight changes to the rules are incredibly easy to add once you have variationPoints, gameDomain, controllers, and the rest of the files set up.
   2. For this example, we'll make spiderette, a game identical to spider except using a klondike layout and deal.
   3. To do so, all you need to do is create another package object, same as you did for spider.
   4. Name this one spiderette, and in addition to definining a package object like you did for the original spider, override anything that needs to be different.
      a) In our case, we need to change the layout and structure maps to account for the klondike layout's different tableau, foundation, and stock number.
      b) Remember to extend it from variation points!
   5. Once that's done, you just need to add the controller to the Spider.scala file alongside the spider controller.
   6.
```
class SpideretteController @Inject()(webJars: WebJarsUtil, applicationLifecycle:
ApplicationLifecycle)
          extends SpiderVariationController(webJars, applicationLifecycle) {
          override lazy val variation = spiderette
          }
```
   7. Finally, remember to add SpideretteController to routes! Remember, when you generate games in a variation family, you need to use localhost:9000/familyname/variation name, ex: spider/spiderette

H. If you need any assistance throughout this process, take a look at the actual spider files! You can find them in src\main\scala\org\combinators\solitaire\spider

III. Reference Guide

A. Templates and set up
   1. Variation Folders go in src/scala/org.combinators/solitaire and should bear the name of
   2. There are 4 templates one needs for a variation/family
      a) SolitaireController (always name it "controllers")
      b) SolitaireTarget or SolitaireTargetFamily (if creating a family of variations) which should be named the name of the variation/family
      c) Solitaire package, always name it "package" (this may automatically change)

(1) If you are making a variation in a family check to make sure the package object name is the same as your variation

d) SolitaireGameDomain, name it your variation/family name and the word Domain ex. “narcoticDomain”

B. Structure

1. The structure determines which objects exist in plan (such as decks columns and piles) and where each exists
2. If you used the template there is a val that is the structure’s map. It maps the potential locations: Tableau, StockContainer(deck), Reserve, and Foundation to what they contain: stocks, piles, columns, etc.
3. Add the StockContainer (needed for all variations) like this: Container->Seq(Stock(1)). The 1 indicates to use a single deck.
4. Add all others that you need like so: Tableau -> Seq.fill[Element](numTableau)(Pile). Replace numTableau with the number of Piles you want in the Tableau for this example
5. The End product looks something like this:

```
val structureMap:Map[ContainerType,Seq[Element]] = Map(
 Tableau -> Seq.fill[Element](4)(Pile),
 StockContainer -> Seq(Stock(1))
}
```

C. Layout

1. Similar to the Structure the layout is usually stored as a map but you can also use a premade one if it matches what you need exactly. The Layout determines where you elements will be placed on the board.
2. A finished version of this may look like:

```
val layoutMap:Map[ContainerType, Seq[Widget]] = Map (
 Foundation -> horizontalPlacement(200, 10, 4, card_height),
 Tableau-> horizontalPlacement(200, 410, 4, card_height),
)
```

   This creates a row of 4 cards each placed the same distance apart at (200,10) and another row for the tableau under it
3. If you do not need a physical deck only add the the stock to the structure NOT the layout

D. Deal

1. The deal is how cards are initially dealt at the start of the game. It consists of a sequence of steps, primarily deal steps
2. A deal step specifies to where to deal and how many cards and lastly whether face up or down
3. For example here is declaring the sequence and dealing two cards to the 4th element in the tableau

```
var deal:Seq[Step] = Seq()
deal = deal :+ DealStep( ElementTarget(Tableau, 4), Payload(numCards =  2))
```

4. Using containertarget instead of element target, one can send cards to the container as a whole instead of individual elements

5. Filter steps can be added instead of dealSteps, these allow you to filter certain cards to the top of the deck. The below is
```
var deal:Seq[Step] = Seq (FilterStep(IsKing(DealComponents)))
```

E. Moves and Constraints

1. The moves that you'll need depend greatly on which Solitaire game you're playing. However, you'll very likely need **Deal**, **Tableau-to-Tableau**, and **Tableau-to-Foundation** moves.
2. Deal moves involve placing cards from the top of the deck onto the board, usually the tableau.
```
val deckDealMove:Move = DealDeckMove("DealDeck", 1,
 source=(StockContainer, NotConstraint(IsEmpty(Source))), target=Some((Tableau,
Truth)))
```
This move, for example, deals a single face up card to each container of the tableau, as long as the deck isn't empty.
3. Each move has a **source** and a **destination**, with potential **constraints** on each. In the above example, the source is the StockContainer (where the deck is held) and the destination is Some((Tableau, Truth)), meaning any/all tableau container.
4. The NotConstraint in the source is the conditions that must be met for the move to begin. In this example, you can't even start the deal move if the deck is empty, sensibly.
5. The Truth in the target is the conditions that must be met for the move to finish. In this case, assuming you can start the move, nothing will stop you from dealing. Setting it as Truth allows it in all cases.
6. Moves also have names, in this case, "DealDeck". It's best to choose something straightforward.
7. DeckDealMoves have an additional int parameter to indicate the number of cards to deal. Here it's set to 1, but if it were 3 then it would deal 3 cards to each target.
8. Tableau-to-Tableau moves are those that move one or more cards from one container on the tableau to another. They're often the bulk of moves made in a solitaire game.
```
val tableauToTableauMove:Move = MultipleCardsMove("MoveColumn", Drag,
 source=(Tableau,Truth), target=Some((Tableau, Descending(MovingCards))))
```
This move allows you to drag-move a stack of one or more cards from one tableau container to another, as long as their ranks are in descending order. The syntax for moving a single card is the same, but the type is SingleCardMove
9. Once again, we have a source and target with constraints as well as a name. However, we also specify here that this is a Drag move, as opposed to a click or press move. The Truth constraint on the source means that you can always start this drag move. However, Descending(MovingCards) on the target makes it so that you can only complete the move if you finish it with a stack of cards whose ranks are in

descending order. If you try to perform the move without meeting the target’s constraints, the cards will return to where they started.

10. Tableau-to-Foundation moves function identically, though instead of the target being Some(Tableau) it would be Some(Foundation)

F. Constraints (cont.)

1. You can combine constraints to implement complex move requirements, just like real solitaire games!
2. This is mainly achieved through **AndConstraint**, **OrConstraint**, **IfConstraint**, and **NotConstraint**.
3. AndConstraint allows you to AND two constraints. AndConstraint(A, B) creates a new constraint that requires that both A and B to be satisfied. You can stack AndConstraints as well, such as AndConstraint(A, AndConstraint(B, C)).
4. OrConstraint functions the same as AndConstraint but allows you to OR two constraints.
5. IfConstraint functions like an if, it checks a condition (first parameter) and applies its second parameter (a constraint) if true and the optional third constraint if the condition is false
6. NotConstraint takes in another constraint and creates a new constraint that requires the opposite. You can see it in the DeckDealMove example above, being used with IsEmpty to make sure the stock has cards to deal.
7. A wide variety of basic constraints are included with the framework. They can be combined using And/Or/If/Not to emulate a great portion of solitaire logic! Additionally, you can write custom constraints to accommodate more game-specific rules, (See V. Special Constraints).

G. Logic

1. This refers to the win logic, it is a state of the board
2. The most common logic for victory is all cards in foundation, shown below for a single stock game

```
Logic = BoardState(Map(Tableau -> 0, Foundation -> 52))
```

H. Routes

1. The routes file (/src/main.resources/routes) and is where you add your route (example below). Just replace napolean.Napolean with your variation name (the package then Scala file name)

```
->    /               org.combinators.solitaire.napoleon.Napoleon
```

I. Specialized elements

1. Specialized Elements require changes in multiple files you must first declare the object in the package object you are using it in:

```
case object FreePile extends Element(true)
```

2. You can then add it to the structure like any other element

```
Reserve -> Seq.fill[Element](2)(FreePile)
```

3. Finally for the packaged object make sure to add it to the specialized elements Seq

```
specializedElements = Seq(FreePile),
```

4. Now go to the controllers file and add ignore handlers as needed

```
updated = updated.addCombinator(new IgnoreClickedHandler('FreePile))
```

5. If you need to only apply the specialized elements to certain variations of a family you can use a case

```
s match {
 case fanfreepile => {
   updated = updated.addCombinator(new IgnoreClickedHandler('FreePile))
 }
}
```

6. Finally in the domain you must override both the model and view names and match your special element to what it appears as (ex for column use “Column” and “ColumnView” instead of “Pile” and “PileView”

```
override def baseModelNameFromElement (e:Element): String = {
 e match {
   case FreePile => "Pile"
   case _ => super.baseModelNameFromElement(e)
 }
}

override def baseViewNameFromElement (e:Element): String = {
 e match {
   case FreePile => "PileView"
   case _ => super.baseViewNameFromElement(e)
 }
}
```

J. Special constraints/functions

1. Similar to Specialized elements first declare the constraint in your package object. It must be a case class

```
case class MaxSizeConstraint(movingCards: MoveInformation,
destination:MoveInformation, maxSize:Int) extends Constraint
```

2. Then use it like you would any other constraint
3. In the code generator of the domain file add the registration for it

```
object fanCodeGenerator {
 val generators:CodeGeneratorRegistry[Expression] =
CodeGeneratorRegistry.merge[Expression](
   CodeGeneratorRegistry[Expression, MaxSizeConstraint] {
     case (registry:CodeGeneratorRegistry[Expression], c:MaxSizeConstraint) =>
       val destination = registry(c.destination).get
       val moving = registry(c.movingCards).get
       val num = c.maxSize
       Java(s"""ConstraintHelper.maxSizeExceeded($moving, $destination,
$num)""").expression()
     },
 ).merge(constraintCodeGenerators.generators)
}
```

4. Finally in the helper methods for Java add the function to generate

```
@combinator object HelperMethodsFan {
 def apply(): Seq[BodyDeclaration[_]] = {
   val methods = generateHelper.helpers(solitaire)
   methods ++ Java(s"""
                      |public static boolean maxSizeExceeded(Card moving, Stack
destination, int max) {
                      | return destination.count() < max;
                |}""".stripMargin).methodDeclarations()
 }
 val semanticType: Type = constraints(constraints.methods)
}
```

K. Solvable

1. Solvable is a type of special function.
2. First set the optional solvable parameter for your variation to true

3. Then implement the available moves function in the extra methods combinator in the domain file. Note the potential move are just the names you gave to each of your moves with the word potential before them

```
@combinator object ExtraMethods {
 def apply(): Seq[MethodDeclaration] =

   Java(s"""public java.util.Enumeration<Move> availableMoves() {
           |  java.util.Vector<Move> v = new java.util.Vector<Move>();
           |        for (Column c : tableau) {
           |            for (Pile p : foundation) {
           |                PotentialMoveCardFoundation pfm = new
PotentialMoveCardFoundation(c, p);
           |                if (pfm.valid(this)) {
           |                    v.add(pfm);
           |                }
           |            }
           |        }
           |        if (v.isEmpty()) {
           |            for (Column c : tableau) {
           |
           |            for (Column c2 : tableau) {
           |                PotentialMoveCard pm = new PotentialMoveCard(c, c2);
           |                if (pm.valid(this)) {
           |                    v.add(pm);
           |                }
           |            }
           |           }
           |        }
           |        return v.elements();
           |}
    """.stripMargin).methodDeclarations()

 val semanticType: Type = game(game.methods :&: game.availableMoves)
}
```

L. Variation Points

1. When making a family of variations place all the code from the package object of the base variation, except the declaration of the variation, into a new file which is a trait that all variations in this family will extend. Below is an example of an empty variation points trait and the package object after code was moved to variation points

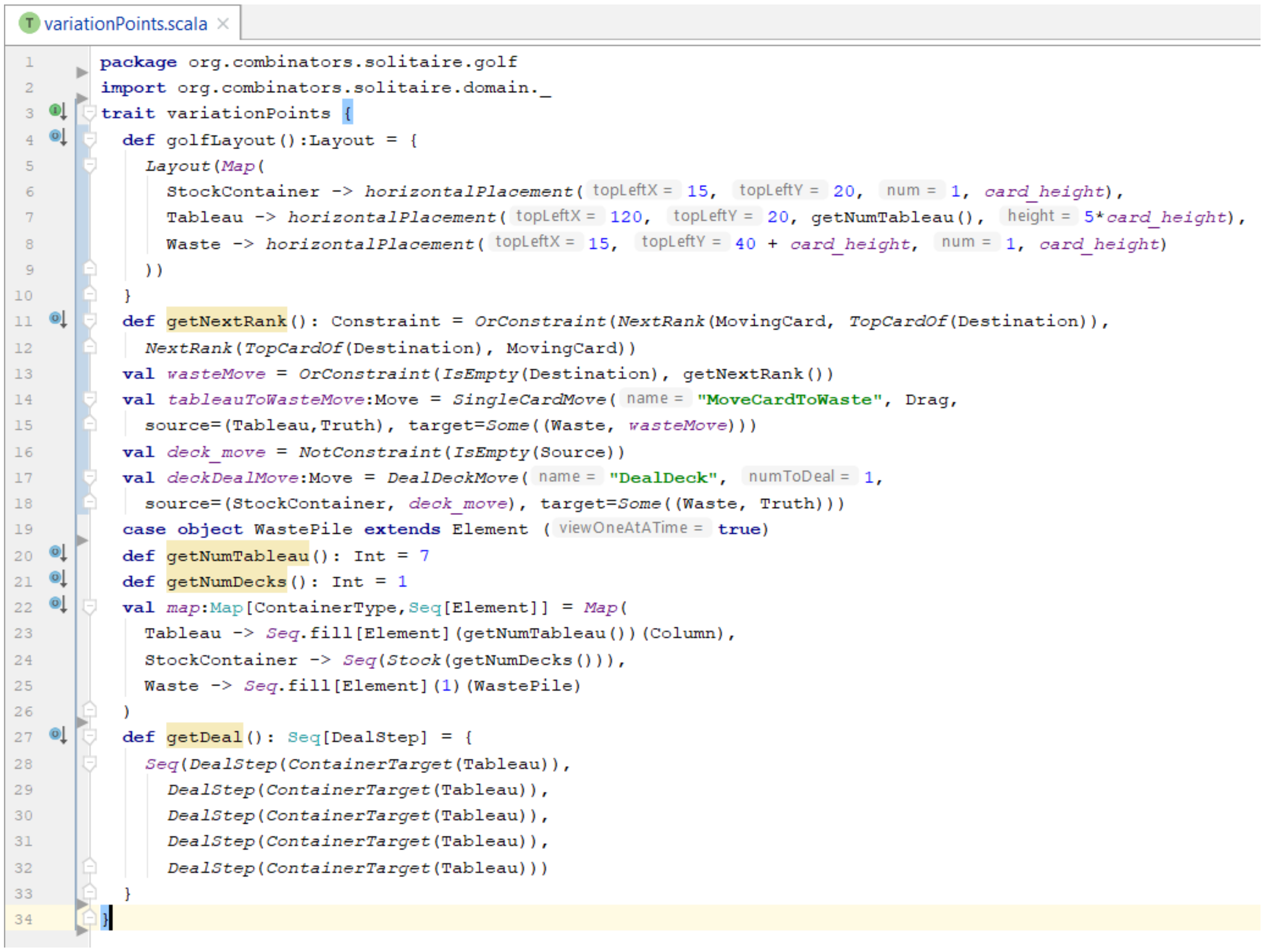
```
trait variationPoints {...}

package object fan extends variationPoints {
 val fan:Solitaire = {
   Solitaire(name = "Fan",
     structure = structureMap,
     layout = Layout(layoutMap),
     deal = getDeal,
     specializedElements = Seq.empty,
     moves = Seq(tableauToTableauMove, tableauToFoundationMove),
     logic = BoardState(Map(Tableau -> 0, Foundation -> 52))
   )
 }
}
```

2. You may also want to make a constants trait which contains everything that does not actually vary if there are significant characteristics of your family that are the same for each variation

IV. Known Bugs

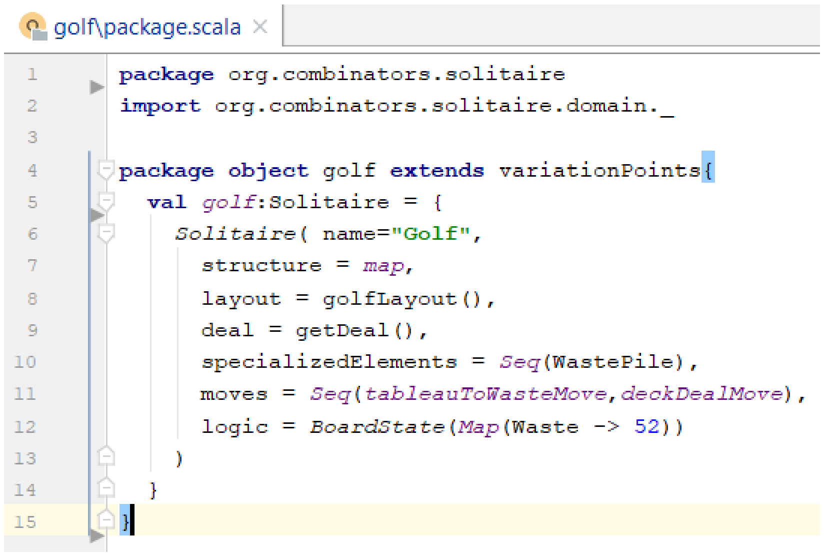
A. If a card is dragged into a container that there is no handler for, it disappears

Appendix B: Modeling Needed for Golf



```
package org.combinators.solitaire.golf
import org.combinators.solitaire.domain._
trait variationPoints {
  def golfLayout():Layout = {
    Layout(Map(
      StockContainer -> horizontalPlacement( topLeftX = 15,  topLeftY = 20,  num = 1, card_height),
      Tableau -> horizontalPlacement( topLeftX = 120,  topLeftY = 20, getNumTableau(),  height = 5*card_height),
      Waste -> horizontalPlacement( topLeftX = 15,  topLeftY = 40 + card_height,  num = 1, card_height)
    ))
  }
  def getNextRank(): Constraint = OrConstraint(NextRank(MovingCard, TopCardOf(Destination)),
    NextRank(TopCardOf(Destination), MovingCard))
  val wasteMove = OrConstraint(IsEmpty(Destination), getNextRank())
  val tableauToWasteMove:Move = SingleCardMove( name = "MoveCardToWaste", Drag,
    source=(Tableau,Truth), target=Some((Waste, wasteMove)))
  val deck_move = NotConstraint(IsEmpty(Source))
  val deckDealMove:Move = DealDeckMove( name = "DealDeck",  numToDeal = 1,
    source=(StockContainer, deck_move), target=Some((Waste, Truth)))
  case object WastePile extends Element ( viewOneAtATime = true)
  def getNumTableau(): Int = 7
  def getNumDecks(): Int = 1
  val map:Map[ContainerType,Seq[Element]] = Map(
    Tableau -> Seq.fill[Element](getNumTableau())(Column),
    StockContainer -> Seq(Stock(getNumDecks())),
    Waste -> Seq.fill[Element](1)(WastePile)
  )
  def getDeal(): Seq[DealStep] = {
    Seq(DealStep(ContainerTarget(Tableau)),
      DealStep(ContainerTarget(Tableau)),
      DealStep(ContainerTarget(Tableau)),
      DealStep(ContainerTarget(Tableau)),
      DealStep(ContainerTarget(Tableau)))
  }
}
```

Variation Points for Golf



```
package org.combinators.solitaire
import org.combinators.solitaire.domain._

package object golf extends variationPoints{
  val golf:Solitaire = {
    Solitaire( name="Golf",
      structure = map,
      layout = golfLayout(),
      deal = getDeal(),
      specializedElements = Seq(WastePile),
      moves = Seq(tableauToWasteMove,deckDealMove),
      logic = BoardState(Map(Waste -> 52))
    )
  }
}
```

Model for Golf