Generating Solitaire Games

A Major Qualifying Project Report
submitted to the Faculty of:

WORESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the
Degree of Bachelor of Science.

March 11, 2020

SUBMITTED BY:
Daniel Duff, Andrew Levy

ADVISOR:
George T. Heineman

Abstract

Software developers face a steep learning curve when trying to work with complex object-oriented frameworks. This reduces productivity since it takes a long time for a developer to learn how to effectively use the complex frameworks. Combinatory Logic Synthesis (CLS) can be used to generate code for arbitrarily complicated object-oriented frameworks given a simple domain model. This automation can vastly improve the productivity of development in a new framework by generating code that involves arbitrarily complicated boilerplate code necessary to use the framework. This project looks to upgrade and complete a solitaire domain modeling effort in an ongoing research project, Next-Gen Solitaire. A successful completion of this effort will allow us to model a wide range of solitaire families, from which hundreds of variations can be generated. We evaluate the project by the number of variations that are generated, ease of usability, and making it possible to generate unit test cases directly from the domain models.

Table of Contents

Table of Figures

1 Introduction

Software frameworks are created and implemented to simplify the development environment for programmers. While a developer that is experienced in the framework will be able to use this simplicity to their advantage, inexperienced developers have to learn the framework from the ground up. It can be time-consuming to grasp a new framework with all of its intricacies. During this learning process, newcomers may have a challenging time working with the unfamiliar framework that results in suboptimal code.

Of course, the goal of a framework is to improve the efficiency of coding rather than hinder it, but the learning curve can certainly be a burden. As time working with the framework increases, so does the efficiency and eventually, a developer can master and implement with ease. One of the significant goals of this project was to create a framework that could be learned much faster, allowing any user to design and create without the challenge of understanding a foreign schema. A key component in the creation of this new framework is code generation.

Code generation is a process in which the compiler will use a code generator to transform a representation of source code into a machine-readable form that then can be run on its own. This process is an extremely useful tool for alleviating difficulty in learning a new framework because the programmer will not need to learn certain concepts within the framework if the program is generating that code on its own. When used properly, code generation can remove a lot of the heavy lifting for the programmer, which will drastically increase the efficiency of the process and cut down on the time it takes to learn the framework.

A further expansion of code generation is the automatic generation of test cases. Testing is an important part of the development of any software as it ensures the user that their code is completing tasks properly and in full. It is essential for any project to be tested, but sometimes this may be a time-consuming task for the user. Testing all the edge cases for each action may result in more complete and secure code, but it comes at the cost of time and efficiency. Generating these test cases is a way to limit that time since the program will be able to test the code for the user. Assuming the test generation method is accurate and thorough, the program will be able to test all edge cases in a smaller amount of time. Depending on the program, the method of generating test cases will vary, as the test generation framework will need to know how to test each method. Test case generation can be an extremely useful feature to improve efficiency when implemented properly.

As described in Drew Ciccarelli, Ian MacGregor, and Simon Redding's *Generating Solitaire Games* Major Qualifying Project: "Next-Gen Solitaire is a code generation framework designed by George Heineman, Jan Bessais, and Boris Düdder. Given a Scala model representing the rules and design of a solitaire game, it uses Combinatory Logic Synthesis (CLS) (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. Using the existing domain logic, developers can quickly create their own specialized logic to deal with game rules or elements that are outside the scope of the base framework." (Ciccarelli)

The newest iteration of this project continues upon the development of the Next-Gen Solitaire framework. The first steps of this iteration were to learn the Next-Gen Solitaire framework and how it uses code generation by creating new variations. This also included testing all of the current variations as to whether or not they can be compiled and run. Once the team learned the framework, the next step was to implement automatic test case generation for Solitaire variations. Once the new variations and test case generation were completed, the last undertaking was to polish the project so that it could be accessed by any user who had an interest. This includes playing any of the generated Solitaire variations or creating their own. Several tutorials were also created to allow users to follow along and learn how to create a variation, which will decrease the learning curve of the Next-Gen Solitaire framework.

The changes in this newest iteration of the project were made with two objectives in mind. The first was to improve the efficiency of working with the framework. While many strides had been made previously in this regard, test case generation was a brand-new feature that was implemented to improve efficiency. Users can test their code in a quick manner that also ensures the code is working properly and that the variations are following the proper rulesets. The second was to make the project more user-friendly. The intent of the framework was to allow users to make their own Solitaire variations by creating their own models. However, before this iteration, there was no simplistic way for the user to get access to the code or to add to it. With the addition of the new tutorials, the ability for anyone to push variations to Git, as well as the interface, users have a much more streamlined process to make and play variations.

2 Background

2.1 Solitaire

The game of Patience (Solitaire) dates back to the mid-18th century. The game was spread all across Europe and while it was not the most popular game at the time, many historical figures enjoyed the game, such as Napoleon, Prince Albert, and Charles Dickens (Tung, 2015). All across the world, different countries and cultures put their own twist on the game with different names and rules. This was the start of the creation of new solitaire variations which paved the way for the countless number of solitaire variations that we see today. Over time, the game continued to grow but in the 1980s Solitaire took off at a rapid rate due to the rise in personal computers. Since these new computers could shuffle cards and moderate rules, this made the process of playing a game of solitaire much simpler and more effortless. The release of portable computers also led to a tremendous increase in the popularity of Solitaire, due to the convenience and accessibility of being able to run Solitaire programs in any location.

Many people have taken the base solitaire game and modified it by adding their own rules, moves, and structure. These modifications create entirely new adaptations of solitaire called "variations". The possibilities of variations of solitaire are seemingly endless. Due to this, variations can be quite different from one another, which allows the player to get a new and interesting experience with every game. However, not all variations are completely unique as many tend to be slight changes of another variation. These similar variations are organized together in groups called "families". Families are named after the variation that is the origin of that family. Each variation in a family usually has its own name, but also falls under the name of a family as well. For example, a commonly known solitaire variation is Scorpion Solitaire. Scorpion falls under the Spider Solitaire family since Scorpion originated from the Spider Solitaire variation. The number of families and variations is large and continually growing.

Solitaire was a natural fit for programmers during the period of the first personal computers. Patience was simple and could be represented easily via text-mode (SolitaireCentral, 2012). It also allowed for a great deal of creativity as programmers could implement whichever variations they enjoyed and could even create their own. The earliest programs allowed for only one of these text-based games, but as computers continued to advance in graphics and memory capacity, Solitaire programs progressed to include multiple variations with better graphics that are still improving to this day. In 1987, the first commercial Solitaire collection, "Solitaire Royale" was released. This collection contained eight playable variations. Five years later a new collection, "Solitaire's Journey" was released. This collection featured 105 different variations and also included comprehensive user statistics which allowed players to work toward improving their scores and earning rewards. The release of Microsoft's Windows Solitaire contributed to a major spike in the increase in the popularity of Solitaire. Microsoft offered a wide variety of Solitaire variations, which led to the popularity of games such as Klondike, FreeCell (https://en.wikipedia.org/wiki/FreeCell), and Spider. Within the past 20 years, the number of Solitaire programs has exploded, allowing users to play any Solitaire variation they want on almost any computing device.

An example of a popular open-source collection of Solitaire games is PySol. PySol was a cross-platform collection of over 1,000 different solitaire variations. A feature within PySol allowed users to include their own variations by adding a file with python code with the new variation on it. The original framework came to a halt in 2004 but was continued as the Fan Club edition and is still running today. PySol was the inspiration for many code generation projects including the Next-Gen Solitaire framework, which utilizes its combinators to generate Solitaire games in Python code.

In 2003, Professor George Heineman developed KombatSolitaire for an undergraduate software engineering course. Developed in Java, this project allowed students individually to write solitaire variations that conformed to the framework. In 2014, Professor Heineman collaborated with Jan Bessai and Boris Düdder to create LaunchPad v0.6 [https://github.com/combinators/nextgen-solitaire/wiki/V-0.6-solitaire], the first working version of a code generation framework using CLS that could be extended to create new Solitaire variations by combining existing game features. This tool was included as part of Boris Düdder's Ph.D. entitled "Automatic Synthesis of Component & Connector-Software Architectures with Bounded Combinatory Logic"

The first major release, V1.0 [https://github.com/combinators/nextgen-solitaire/wiki/V-1.0-solitaire] was described in a research publication that demonstrated how to automatically synthesize five solitaire variations as part of a product line of solitaire variations (Heineman et al., 2015).

The project later evolved in 2017 to the current project seen today, which utilizes combinators in Scala [https://github.com/combinators/nextgen-solitaire/wiki/V-2.0-solitaire]. Developers model solitaire variations in Scala, which are provided as input to the code generator which generates the resulting Java code. This new variation allowed for a significant increase in scalability and since this process was language-independent, the combinators could also be used to generate code in other languages such as Python. The current iteration of the project included the generation of working test cases as well as the combination of the Next-Gen solitaire framework with the UI which allows users to run all of the working variations of the project.

2.2 Combinatory Logic Synthesis

Combinatory logic synthesis (CLS) utilizes combinators to synthesize code using combinators. Combinatory logic synthesis (Bessai et al., 2014) [ETAPS2014] is a type-based approach to component-oriented synthesis using types as interface specifications (Rehof & Vardi, 2014). CLS automates the composition of components from a repository using combinatory logic. CLS repository is modeled as a finite combinatory type environment Γ with type assumptions $x : \tau$, where x is a combinator symbol and τ is its implementation type. The logical foundation of this idea is to consider the inhabitation relation in combinatory logic: Given an environment Γ and a type τ , does there exist a combinatory expression M such that $\Gamma \vdash M : \tau$? An algorithm that solves inhabitation problems can compute (or enumerate) expressions M, referred to as inhabitants of the type τ. The basic process of CLS is to construct a repository of combinators and then request the inhabitants for a given collection of types $\tau_1, \tau_2, \ldots \tau_n$.

To increase the flexibility of CLS, staged composition synthesis (SCS) (Düdder et al., 2014) introduces a functional meta-language (referred to as L2) in which component implementation language-code (referred to as L1) can be manipulated. The metalanguage is a restricted form of the $\lambda^{\square}$-calculus of Davies and Pfenning. In $\lambda^{\square}$, a modal type operator $\square$ ("box") is used to inject L1-types into the type-language of L2. Type $\square$int can be read as code of a program (L2) when compiled and executed yielding a value of type *int* (L1). In SCS, inhabitation treats L1/L2 combinators as atomic building blocks additionally described by a stratified layer of semantic types, attached by the intersection type operator "∩". Semantic specifications are used to guide program synthesis

The best way to explain how CLS works is to use a small example. The current toolset uses Scala as the L2 language and -- for the Solitaire example -- Java is the L1 language.

```
@combinator
object Earth {
  def apply:Expression = {
    Java(s""""Earth"""").expression()
  }
  val semanticType: Type = 'Planet
}

@combinator
object Mars {
  def apply:Expression = {
    Java(s""""Mars"""").expression()
  }
  val semanticType: Type = 'Planet
}
```

Figure 1: Simple Java Expression Combinators

These two combinators, respectfully, represent Java expressions. Combinator Earth represents the Java string `"Earth"` while Mars represents `"Mars"`. These two atomic building blocks can be used to assemble larger programs. The `@combinator` annotation tells the Scala system to treat this object as a combinator for CLS. Each combinator has a type specification as shown by the `semanticType` **`val`** associated with the object. Each `semanticType` is a `Type`, which here is a simple terminal symbol `'Planet`. The application of a combinator (the apply method) results in code artifacts in Java, in this case an Expression. Java is a convenient language to use as L1 because of the extensive Abstract Syntax Tree (AST) tools available. In our case, we use the powerful opensource javaparser [https://github.com/javaparser/javaparser]. Ultimately, each combinator results in a node in an AST.

```
@combinator
  object PlanetExploration {
    def apply(expr:Expression) : CompilationUnit = {
      Java(
        s"""
           |package something;
           |
           |public class NewClass {
           |  public static void main (String[] args) {
           |    System.out.println($expr);
           |  }
           |}
           |""".stripMargin).compilationUnit()
    }

    val semanticType: Type = 'Planet =>: 'Complete
  }
```

Figure 2: CompilationUnit Combinator Example

The PlanetExploration combinator combines all elements and shows the power of CLS. First, observe the `semanticType` which is of the form A → B. This is a standard abstraction which can be viewed as a function f(A) which results in an object of type B. Note that A → B → C can be interpreted as f(A, B) which results in an object of type C. In the case of PlanetExploration, the result will be a `CompilationUnit`, (that is an entire Java class) after it consumes an input of type `'Planet`. From the earlier example, there are two combinators of this type. PlanetExploration will consume either one (which produces an `Expression`) and use that Expression inside the Java class as the argument to `System.out.println`. Observe how this combinator definition makes extensive use of the String interpolation capability of Scala to construct arbitrary strings, which are then parsed for syntax correctness to produce a Java class.

The inhabitation request of `'Complete` would result in two inhabitants found, one would be a Java class that prints out "Earth" – let's call this `PlanetExploration(Earth)` – and the other `PlanetExploration(Mars)` would print out "Mars."

To make it easier to develop complicated combinators, we take advantage of a template mechanism to simplify their definition.

```
@combinator
  object TemplateLoad {
    def apply(expr:Expression) :CompilationUnit = {
      shared.java.SampleTemplate.render(expr).compilationUnit()
    }
    val semanticType: Type = 'Planet =>: 'Complete
  }
```

Figure 3: Expression Template Loader

The above combinator has the same overall type as PlanetExploration, but TemplateLoad instead loads up the definition of the Java class file from a separate template file whose contents appear in Figure 4. This stand-alone file is defined using Play [https://www.playframework.com/documentation/2.8.x/ScalaTemplates] and supports passing in arguments -- in this case, the consumed Expression object:

```
@(planet:Expression)

package anotherOne;
public class SecondClass {
     public static void main (String[] args) {
         System.out.println(@Java(planet));
     }
}
```

Figure 4: Generated Expression from Template

This template specifies in its first line that it takes a single argument, planet, of type `Expression`. Similarly, the `Expression` value of `planet` is injected into the template.

Now assume there is a Γ repository consisting of { `Earth, Mars, PlanetExploration, TemplateLoad` }. From this repository, the inhabitation request Γ ⊢ M : `'Complete`? will produce four inhabitants:

- PlanetExploration(Earth)
- PlanetExploration(Mars)
- TemplateLoad(Earth)
- TemplateLoad(Mars)

The existing toolset is supported by the Play framework to generate each inhabitant upon request using a web server. From within the **sbt** framework, execute the command

```
nextgen-solitaire/run
```

and a web server is launched to process HTTP inhabitation requests. This populates the Gamma repository with its included combinators.

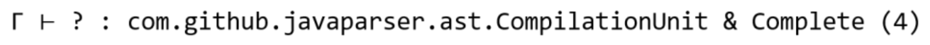
```
Γ = {
  Earth : com.github.javaparser.ast.expr.Expression & Planet
  Mars : com.github.javaparser.ast.expr.Expression & Planet

  TemplateLoad : (com.github.javaparser.ast.expr.Expression ->
com.github.javaparser.ast.CompilationUnit) & (Planet -> NewCode(Begin))

  PlanetExploration : (com.github.javaparser.ast.expr.Expression ->
com.github.javaparser.ast.CompilationUnit) & (Planet -> NewCode(Begin))
}
```

Figure 5: Gamma Repository Combinators

Based on the following request:



```
Γ ⊢ ? : com.github.javaparser.ast.CompilationUnit & Complete (4)
```

Figure 6: Gamma Repository Request

There are four inhabitants found. The web site produces four variations that can be generated and the user can choose which one to produce. After selecting a variation, a git repository is created which contains the generated code, and the user can simply check it out using git. There is no need to show the actual generated code since the above examples are so small.

Solutions:

Variation 0:

- Compute

Variation 1:

- Compute

Variation 2:

- Compute

Variation 3:

- Compute

Figure 7: Generated Variations

2.3 Code Generation

Code generation is a process in which the compiler will use a code generator to transform a representation of source code into a machine-readable form that then can be run on its own. Since the early stages of code generation, programmers have been attempting to generate code for many various uses. The most prominent use is to generate repetitive and boilerplate code that can be written once and utilized repeatedly throughout a program. Other uses include generating code from information gathered from documents during runtime and generating a skeleton from defined models (utilized within Next-Gen Solitaire). The code that is generated by a program can be as small as a simple function or as large as an entire application.

The four main reasons why programmers choose to generate code are to increase productivity, consistency, portability, and simplicity (Tomassetti, 2018). When the user generates code, they do not have to continue to rewrite duplicate code snippets within the same program. When the program is generating repeated code on its own, the user can save time and be more productive. Since the same code is being generated every time it is needed, it removes the chance for variability within the code. Each variable, method name, and all other facets of the code will be exactly the same. This will make it easier as everything remains consistent and compatible when different developers with different styles may be working on the same project. The portability of a project is increased with code generation because of the ability to generate that code into different languages or frameworks. In Next-Gen Solitaire, the project has utilized generated Python and Java code, which demonstrates the significance of code generation's portability. The last useful reason for code generation is simplicity. When code is being generated from some abstract description, that description defines what will be generated and run in the final product. All the user needs to work with is that description. In Next-Gen Solitaire, this description is a model, which defines all the concepts for each variation of solitaire

While code generation has many advantages, there are also reasons why it is not always the best option for every program. When you create a code generator, the code that is generated will always be the same, so while the rest of the program is developed and advances, the generator needs to be maintained to stay up to date. This can become a major problem if the right knowledge or resources become unavailable to continue maintaining the generator. Another complication with code generation is that creating the code generator can sometimes be a complex process. While working with a complex generator to improve simplicity in the future can be extremely useful, sometimes the risk of overcomplicating the process does not outweigh the reward. Finding a productive and worthwhile code generation process can drastically improve a project, but an overly intricate process may make it harder for the developers in the long run.

2.4 Test Case Generation

While base code can be generated for an application, test cases can be automatically generated as well. One of the biggest issues with generating test cases is creating meaningful tests that are checking for accuracy instead of just for the ability to run without failure (Hicken, 2018). Generating assertions to check for success will ensure that the accuracy is being tested as well as the program's ability to run. It can be simple at first to increase code coverage by testing commonly used methods, but generating code for edge cases can be more difficult. Due to the intricacies of some programs, it may be hard to create a generator that tests each program in its entirety. Also, as the code changes, the tests will need to be regenerated, so any good test suite generator needs to be able to be regenerated. While it may be difficult to produce complete coverage with meaningful tests, when implemented properly, test case generation can be an efficient tool for ensuring that a program works properly. A major advantage of generating tests over writing them by hand is the increase in productivity. The developers do not need to spend time writing test cases by hand when they can simply have the test cases generated for them.

3 Design

3.1 Scala Solitaire Generation Framework

The Scala solitaire generation framework is the result of refactoring the first initial release of LaunchPad (Heineman et al., 2015). The inherent technical challenge with solitaire as a feature-based product line is that the variation points do not align well with the composition tools provided by FeatureIDE , the technology on which LaunchPad was built. Consider the feature model from this paper:

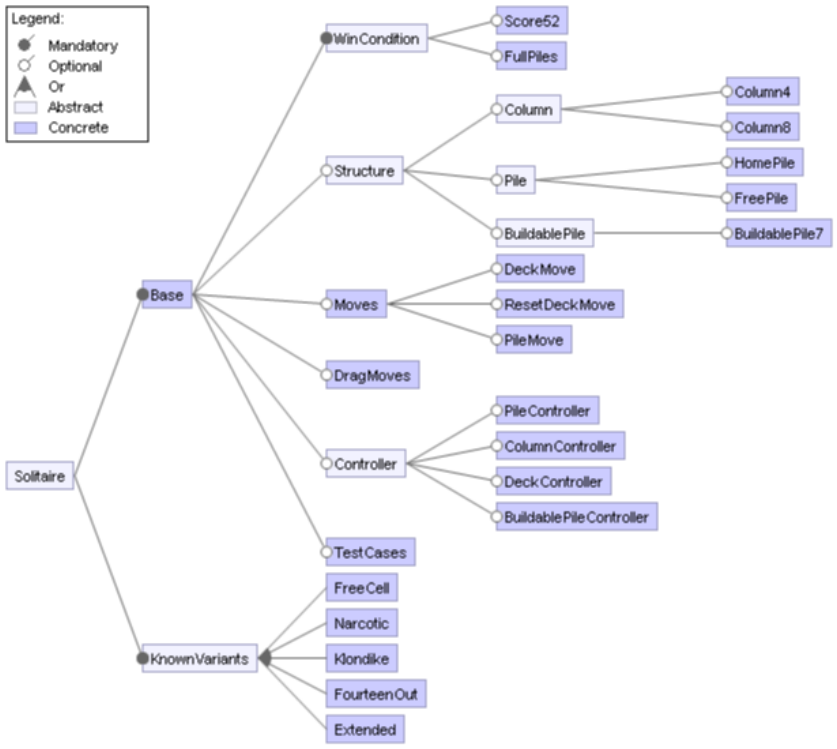


Figure 8: LaunchPad Feature Model

This feature model contains the elements one would typically find in a feature diagram. However, the coarse-grained nature of feature modeling means that the subdivisions do not align well with the numerous variation points one typically finds in solitaire games. For example, in the game of Klondike, one can either deal one card at a time from the stock to the waste pile or three cards at a time. In the feature diagram above, **DeckMove** is the only feature one can select. Under the hood, that is, invisible to the feature diagram, the **Klondike** feature has code artifacts that determine the number of cards to deal.

Essentially, there is an *impedance mismatch* of sorts, where the feature units that are visible in the diagram are much too coarse-grained to be of use when modeling dozens or even hundreds of solitaire variations. In many ways, this is akin to the challenge of using straight object-oriented programming to model all the various solitaire variations using subclasses. The composition tools available to object-oriented programmers or feature-oriented software engineers are not well suited to this application domain.

We assessed that LaunchPad was successful in generating a small number of variations but that it would not be able to scale to support families of solitaire variations, where rather small changes separate two different variations. Professor Heineman conducted an extensive refactoring process that resulted in the Scala code generation framework, currently available in the **nextgen-solitaire** GitHub repository on https://combinators.org.

The essence of the generation framework is that a solitaire variation is composed using CLS from a repository of combinators, thus the problem reduces to determining how to create the necessary combinators. The examples so far use static combinators as tagged with the **`@combinator`** annotation. What is needed is a more flexible arrangement, which allows for the programmatic construction of combinators. To give a specific example, consider the following Scala class definition:

```
class NameDef(cons: Constructor, value: String) {
    def apply(): SimpleName = Java(value).simpleName()
    val semanticType: Type = cons
}
```

Figure 9: Parameterized Combinator

This looks very similar to the combinators we have seen thus far. The primary difference is that it is parameterizable. Since we need to generate a number of Java classes, methods, and fields, each one needs to have a proper name, which is the `value` parameter above which is converted to a Java `SimpleName`. The semantic type is determined by the `cons` parameter. Thus instead of hard-coding all information in a static combinator, from this basic template, many specialized subclasses can be defined:

```
class ClassNameDef(moveNameType: Type, value: String) extends
NameDef(widget(moveNameType, className), value)

class MovableElementNameDef(moveNameType: Type, value: String) extends
NameDef(widget(moveNameType, widget.movable), value)
class SourceWidgetNameDef(moveNameType: Type, value: String) extends
NameDef(widget(moveNameType, widget.source), value)
class TargetWidgetNameDef(moveNameType: Type, value: String) extends
NameDef(widget(moveNameType, widget.target), value)
```

Figure 10: Specialized Subclasses

Each of these is specialized to name elements that are properly typed. To see these combinators in action, consider how moves are represented in the Java solitaire framework. As you may recall, each move has its own class, and in a move class, there must be logic to execute a move, check if it is valid, and undo the impact of a move. Since each of these capabilities is generated from a constructor, we need to populate the gamma repository with the relevant combinators. The following fragment shows how move logic, stored in the solitaire variation **`s`** as a collection of *moves*, causes several different combinators to be added to the gamma repository, using the `addCombinator` method.

```
for (m <- s.moves) {
      val moveSymbol = Symbol(m.name)

      updated = updated
        .addCombinator(new ClassNameDef(moveSymbol, m.name))
        .addCombinator(new ClassNameGenerator(moveSymbol, m.name))
        .addCombinator(new UndoGenerator(m, moveSymbol))
        .addCombinator(new DoGenerator(m, moveSymbol))
        .addCombinator(new MoveHelper(m, moveSymbol))
}
```

Figure 11: Adding Combinators to the Gamma Repository

The solitaire generation framework makes it possible to provide boilerplate strategies for constructing the repository. Here is the initial specification from the Klondike variation which takes a solitaire object (whose specification is described in Chapter 3 of this report).

```
// dynamic combinators added as needed
override def init[G <: SolitaireDomain](gamma : ReflectedRepository[G], s:Solitaire)
:  ReflectedRepository[G] = {
  var updated = super.init(gamma, s)

  updated = createMoveClasses(updated, s)
  updated = createDragLogic(updated, s)
  updated = generateMoveLogic(updated, s)
  updated = generateExtendedClasses(updated, s)

  // these all have to do with GUI commands being ignored.
  s.structure.foreach(ctPair => {
    updated = updated.addCombinator(new
IgnoreClickedHandler(Constructor(ctPair._2.head.name)))

    // In Klondike, it is never possible to release on the Deck
           // or the wastepiles. Can't release on the Deck. And can’t initiate
    // from Foundation.
      ctPair._1 match {
        case Waste => updated = updated.addCombinator(new
IgnoreClickedHandler(Constructor(ctPair._2.head.name)))
        case StockContainer => updated = updated.addCombinator(new
IgnoreReleasedHandler(Constructor(ctPair._2.head.name)))
        case Foundation => updated = updated.addCombinator(new
IgnorePressedHandler(Constructor(ctPair._2.head.name)))
        case _ =>
      }
    })
    // these clarify the allowed moves
    updated = updated
      .addCombinator (new deckPress.DealToTableauHandlerLocal())
      .addCombinator (new SingleCardMoveHandler(wastePile))
      .addCombinator (new SingleCardMoveHandler(fanPile))    // Variation
      .addCombinator (new buildablePilePress.CP2())

    updated = updated.addCombinator (new deckPress.ResetDeckLocal())
    updated = createWinLogic(updated, s)

    updated = updated
      .addCombinator (new DefineRootPackage(s))
      .addCombinator (new DefineNameOfTheGame(s))
      .addCombinator (new ProcessModel(s))
      .addCombinator (new ProcessView(s))
      .addCombinator (new ProcessControl(s))
      .addCombinator (new ProcessFields(s))

    updated
  }
```

Figure 12: Initial Klondike Variation Specification

While this appears complicated, it streamlines the construction of the **Γ** repository. Once the repository is constructed, then a given set of targets is requested for inhabitation. The standard set is:

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

Figure 13: Inhabitation Target Set

In plain English, this requests that the complete game be generated, together with the complete set of constraints. Then all mouse controllers are generated, followed by special classes, move classes for the solitaire domain, and finally all JUnit test cases.

3.2 Code Registries

Professor Heineman developed several code recipes to improve the extensibility of the Scala code generation framework. In particular, the code infrastructure supports the principle that solitaire variations should be able to define their own custom move logic separate from the common code provided by the infrastructure. This most commonly presents itself with the ability to specify additional constraints that are relevant only for the individual variation. For example, in the solitaire game N**arcotic** one of the moves is to remove all four cards from the tableau if they all have the same rank. Because this move is so unique to the **Narcotic** variation, the framework does not provide a built-in constraint which could allow this type of behavior. As such, it must be defined explicitly. Describing how Nextgen-Solitaire supports this capability will demonstrate its flexibility and extensibility.

The first challenge is to describe the constraint.

```
case class AllSameRank(src:MoveInformation) extends Constraint
```

This Scala construct represents a new kind of constraint for the solitaire domain which needs to be integrated into the existing framework. This is done at run-time using the following structure:

```
object narcoticCodeGenerator {
    val generators:CodeGeneratorRegistry[Expression] =
CodeGeneratorRegistry.merge[Expression](

      CodeGeneratorRegistry[Expression, AllSameRank] {
        case (registry:CodeGeneratorRegistry[Expression], all:AllSameRank) =>
          val inner = registry(all.src).get
          Java(s"""((Narcotic)game).allSameRank($inner)""").expression()
      }
    ).merge(constraintCodeGenerators.generators)
  }
```

Figure 14: Narcotic Code Generator

Briefly, this object takes the existing `constraintCodeGenerators.generators` supported by the code generation framework and merges in a new case to be handled, namely `AllSameRank`. The merged generator is integrated using the following combinator, which ensures this object is used whenever the inhabitation algorithm requests constraints`(constraints.generator)`.

```
@combinator object NarcoticGenerator {
   def apply: CodeGeneratorRegistry[Expression] = {
      narcoticCodeGenerator.generators
    }
   val semanticType: Type = constraints(constraints.generator)
}
```

Figure 15: Narcotic Combinator

While this represents some of the combinatory logic, it does not provide the definition for any individual constraints. The actual logic that implements this constraint is defined inside an `allSameRank` method provided by the following `ExtraMethods` combinator.

```
@combinator object ExtraMethods {
  def apply(): Seq[MethodDeclaration] = {
      Java(s"""|public boolean allSameRank(Stack[] group) {
               |   if (group[0].empty()) { return false; }
               |   // Check whether tops of all piles are same rank
               |   for (int i = 1; i < group.length; i++) {
               |      if (group[i].empty()) { return false; }
               |      if (group[i].rank() != group[i-1].rank()) {
               |        return false;
               |      }
               |   }
               |  // looks good
               |  return true;
               |}""".stripMargin).classBodyDeclarations()
                        .map(_.asInstanceOf[MethodDeclaration])
    }

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

Figure 16: Extra Methods Combinator

This combinator will return a collection of new methods to be injected into the solitaire game variation. The `allSameRank` method ensures the array of stack objects all have the same rank on top. The end result of these combinators is that existing infrastructure capabilities (in this case, the mapping of a `Constraint` to a Java `Expression` that implements that `Constraint`) can be easily extended to support additional user-specified extensions. In addition, if a solitaire variation wishes to “override” an existing mapping, then this same approach can be used to replace a mapping with a new version.

With this new constraint integrated into the solitaire generation framework, existing logic that generates valid Move classes will simply inject the constraint logic as necessary without requiring any further modifications to the framework; this is the very definition of extensibility.

This specific recipe is repeated a number of times:

1. `constraintCodeGenerators.generators` handles the mapping of Constraint to Expression.
2. `constraintCodeGenerators.doGenerators` handles the mapping of `MoveTypes` into `Seq[Statement]` that describe the logic for carrying out a move in Solitaire.
3. `constraintCodeGenerators.helperGenerators` handles the mapping of Move Types into Strings that contain class fields and/or additional method declarations to be inserted into a Move class.
4. `constraintCodeGenerators.undoGenerators` handles the mapping of `MoveTypes` into `Seq[Statement]` that describe the logic for undoing a Solitaire move.
5. `constraintCodeGenerators.mapGenerators` handles the mapping of `MapTypes` into `Seq[Statement]` that describes how cards can be dealt to existing Tableau elements based on specific criteria (i.e., by Rank or by Suit).

In all of these cases, individual variations can extend the capabilities of the Solitaire code generation framework.

3.3 Technical Solitaire Terminology

Before analyzing the structure of a given variation's domain within the framework, it is important to understand basic solitaire terminology. The following terms are used throughout all variations and are used to describe both physical and constraint elements.

Card	A regular playing card has a **suit** and a **rank**. It may be face up (in which case this information is visible) or face down (which hides this information).
Column	A stack of cards that are offset to allow the user to see all cards within the stack
Deal	Adding cards to a **layout** by taking them from the **stock** at the start of the game. Dealing is different in every solitaire variation. In some games, there is no **stock** and therefore no dealing at all. In other variations, cards may even return to the **stock**.
Foundation	A designation **pile** for the cards. The goal of many solitaire games is to eventually move all of the cards to foundation piles. Often, the foundations are empty at the start of a game, but in some games, they may begin with a starter **card** that determines which initial cards can be placed.
Layout	The pattern of cards on the table. The initial layout is the set of cards created at the start of the game.
Pile	A stack of cards of which only the topmost card is visible.
Rank	The number value of a card. In solitaire, the Ace can count as either the lowest value card (one) or the highest value card (fourteen). The **Jack**, **Queen**, and **King** count as eleven, twelve, and thirteen respectively.
Stock	A kind of locational pile. Usually a single pile of cards that can be drawn upon, one card at a time, during the game. In many variations, this is typically the deck of remaining cards.
Suit	The suits are **Hearts** ♥, **Spades** ♠, **Diamonds** ♦, and **Clubs** ♣. Hearts and Diamonds are colored red, while Spades and Clubs are colored black.
Tableau	These **columns** and **piles** are typically the "workspace" on the board. The player can typically move individual (or stacks of cards) between tableau elements during game play.

Figure 17: Technical Solitaire Terminology

3.4 Scala Domain Modeling

With the framework in place, working code for Solitaire variations can be generated without users having to write out all of the code themselves. However, a model still must be defined so that the framework knows what structure and rules the generated game will follow. These models appear in each variation's respective Scala package object.

Each model contains a set of concepts defined by the user when creating a variation, and used by the framework for various purposes to generate the proper solitaire game. For example, one of the defined concepts is the set of viable moves called `moves`. This set contains every move within the variation that could be completed within the rules of the specific variation. These moves are defined within the `variationPoints.scala` file for each family and can also be made variation specific within the package object for that variation.

```
val simplesimon:Solitaire = {
    Solitaire(name = "Simplesimon",
      structure = structureMap,
      layout = Layout(map),
      deal = getDeal,
      specializedElements = Seq.empty,
      moves = Seq(tableauToTableauMove, tableauToFoundationMove),
      logic = BoardState(Map(Foundation -> 52)),
      solvable = false,
      testSetup = Seq(),
    )
  }
```

Figure 18: Simplesimon Scala Domain

The above code snippet is an example of a Scala model from the **Simplesimon** variation (derived from the **Simple Simon** Solitaire variation). This model is defined within the package object for each variation within a family. The concepts of a particular game are derived from the `Solitaire` class, which consists of multiple optional and non-optional concepts.

3.4.1 *name* (name = "Simplesimon")

Within any given solitaire variation model are concepts defined to create the components of a game. The first concept shown is `name` which is simply the name of the variation and is defined as a String variable. This String should be unique among all produced variations, as it is used as a unique identifier.

3.4.2 *structure* (structure = structureMap)

The concept `structure` defines the type and quantity of card containers that will be used within the variation. The `structureMap` is defined within the `variationPoints.scala` file that is shared by all variations within the family (or placed in the base package if there is no `variationPoints.scala` defined). This allows for other variations within a given family to override or replace the base map. The variable type for this component is a Map of `ContainerTypes` (such as *Foundation*, *Stock*, or *Tableau*) and element sequences (such as a sequence of 10 *Columns*).
Below is the collection of generic `ContainerType`*:*

```
case object Tableau extends ContainerType
case object Foundation extends ContainerType
case object StockContainer extends ContainerType
case object Reserve extends ContainerType
case object Waste extends ContainerType
```

Figure 19: Generic ContainerTypes

A `structureMap` example is shown below, creating 10 *Column* tableaus, 4 *Pile* foundations, and a single stock.

```
val structureMap:Map[ContainerType,Seq[Element]] = Map(
    Tableau -> Seq.fill[Element](10)(Column),
    Foundation -> Seq.fill[Element](4)(Pile),
    StockContainer -> Seq(Stock(1))
  )
```

Figure 20: Example StructureMap

3.4.3 *layout* (layout = Layout(map))

The layout map (usually called `layoutMap` or `map` in `variationPoints.scala`) defines the location and size of the containers defined in the structure map. The container types are arranged based on a placement function. Because the variable type of a layout is a map of `ContainerTypes` (similar to those used in the `structureMap`*)*, and sequence of widgets, the `horizontalPlacement` function returns a collection of produced widgets. In the below example, the layout map uses the `horizontalPlacement` function to define the x and y position of the container, how many of that container will be placed (for example, 10 *Tableaus*), and the size of the container (which can utilize the global `card_height` variable which represents the size of one card). The number of container elements should match the amount defined in the `structureMap`. As a result, it is recommended that variables be used in place of definitive integers.

```
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, 4, card_height)
  )
```

Figure 21: Variation Layout

3.4.4 *deal* (deal = getDeal)

After the layout is set, the cards must be dealt into the proper containers at the start of the game. This is defined within the *deal* concept. The deal is also defined within `variationPoints.scala` and is shared among all variations within a family. The `deal` concept consists of a sequence of `DealSteps`, as defined below:

```
case class DealStep(target:Target, payload:Payload = Payload()) extends Step
```

The `DealStep` class consists of a target and a payload. In the context of the *deal* concept, tableau elements associated with the given column number (`colNum` in the example below via the `ElementTarget` association method) are the target or receiver of the deal. The *Payload* is simply a collection of cards to be moved, with parameters determining if they should be face-up, as well as the quantity of the cards dealt. These should all be compounded into a single sequence to be returned. In the example below, the 0th to 3rd tableaus are dealt 6 cards, while the remaining tableau columns are dealt 5 cards. Any cards not dealt at the start of the game remain in the stock.

```
def getDeal: Seq[DealStep] = {
    var colNum:Int = 0
    var dealSeq:Seq[DealStep] = Seq()
    for (colNum <- 0 to 3) {
      dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum), Payload(faceUp =
true, numCards = 6))
    }
    for (colNum <- 4 to 7) {
      dealSeq = dealSeq :+ DealStep(ElementTarget(Tableau, colNum), Payload(faceUp =
true, numCards = 5))
    }
    dealSeq
  }
```

Figure 22: Deal Concept Definition

3.4.5 *specializedElements* (specializedElements = Seq.empty)

`specializedElements` is a part of the model that is used to set special elements on the board display. Elements that are "specialized" are any elements that are not contained within the standard board components of decks, tableaus, and foundations. For example, the **bigforty** variation involves the usage of a waste pile, which is added to when the stock is clicked. Only the top card of this waste pile can be viewed and moved as well, adding an additional interaction that sets it apart from a standard foundation or tableau element. An example of the waste pile from the **bigforty** variation package is shown below:

```
case object WastePile extends Element (true)

val map:Map[ContainerType,Seq[Element]] = Map(
    Tableau -> Seq.fill[Element](numTableau)(Column),
    Foundation -> Seq.fill[Element](numFoundation)(Pile),
    Waste -> Seq.fill[Element](1)(WastePile),
    StockContainer -> Seq(Stock())
  )
```

Figure 23: Specialized Waste Element

The interactions which are involved with the Waste Pile are defined in the variation's controller definition, as the pile is added to, and removed from via the usage of the stock, or the waste pile itself. If there are no special elements to be defined, the concept can be set to an empty sequence.

3.4.6 *moves* (moves = Seq(tableauToTableauMove, tableauToFoundationMove))

Next, the set of moves that a user can attempt to make in a given game needs to be defined. The *moves* concept is set as a sequence of moves, where each move is defined within `variationPoints.scala` (or in the variation package file if no `variationPoints.scala` is defined). Move types are arguably the most complex element of a variation definition, as they consist of multiple characteristics and constraints. Below is the class definition for the *move* type:

```
case class Move
(
  name:String,
  moveType:MoveType,
  gesture:GestureType,
  movableElement:Element,
  source:(ContainerType,Constraint),
  target:Option[(ContainerType,Constraint)],
  isSingleDestination:Boolean = true,
)
```

Figure 24: Move Class Definition

The `name` is a String that uniquely identifies the move itself.
The `moveType` is the type of movement for which a given move corresponds to. The generic `moveTypes` are listed below:

```
sealed trait MoveType

case class DealDeck(numCards:Int) extends MoveType
case object ResetDeck extends MoveType
case object MultipleCards extends MoveType
case object FlipCard extends MoveType
case object SingleCard extends MoveType
case object RemoveSingleCard extends MoveType
case object RemoveMultipleCards extends MoveType
```

Figure 25: MoveType Variations

`DealDeck(numCards:Int)`: Deals a specified number of cards to the specified *source* of the overall move.
`ResetDeck`: Resets the deck via the given move.
`MultipleCards`: Movement involving multiple cards.
`FlipCard`: Flips the selected card over (either face up or down).
`SingleCard`: Movement involving a single card.
`RemoveSingleCard`: Removes the single selected card.
`RemoveMultipleCards`: Removes the stack of selected cards.

In addition to having container elements respond appropriately after performing a given action, these types are also parsed in the variation's controller class, causing the visual components to be performed with the given `MoveType`.

A move's `gestureType` is even more closely related to the controller's front-end display of the move, with visuals appearing in concordance with the action of the move. The base gesture types are defined below:

```
sealed trait GestureType

case object Drag extends GestureType
case object Press extends GestureType
case object Click extends GestureType
```

Figure 26: Interaction GestureTypes

All of the `GestureTypes` correspond to the user's mouse actions, with `Click` being a `Press` and `Release` of the mouse. Most actions that correspond to physically moving cards are considered `Drag` gestures, whereas interacting with an element like the stock would be `Press` or `Click` gestures.

The `moveableElement` of a move is the type of element that is actually being moved when the user performs a given action. This could range from single cards (such as when clicking the stock) to whole columns (such as moving a stack of cards from tableau to tableau). The definition and types of `Elements` are shown below:

```
abstract class Element(
                        val viewOneAtATime: Boolean,
                        val verticalOrientation:Boolean = true,
                        val modelMethods:Seq[BodyDeclaration[_]] = Seq.empty,
                        val viewMethods:Seq[BodyDeclaration[_]] = Seq.empty,
                        val modelImports:Seq[ImportDeclaration] = Seq.empty,
                        val viewImports:Seq[ImportDeclaration] = Seq.empty)
                        {
  // case classes have $ in their name
  def name:String = getClass.getSimpleName.replace("$","")
}

case object Card extends Element(true)

case object Column extends Element(false)
case object Pile extends Element (true)
case object Row extends Element(false, true)
case object BuildablePile extends Element (false)
case class Stock(numDecks:Int = 1) extends Element(true)
```

Figure 27: Element and Card Definitions

`Column` and `Pile` elements are effectively the same, aside from `Piles` of cards only showing the top of the stack. `BuildablePile` elements are considered `Pile` elements, yet show

cards similarly to `Column` elements, and are typically meant for building sequences of cards in a particular order.
`Row` elements are similar to `Column` elements, besides vertical orientation.
`Stock` elements represent the game's stock container, with only the top card being shown.

The final main elements of a move, the `Source` and `Target` concepts, have two components: `ContainerType` and `Constraint`. `ContainerType` corresponds to the type of container element involved in either the source or destination of the move. `Constraint` components are the main facet of the validity of a given move. A move will only succeed if the constraints for the `Source` and `Target` destinations are met. The amount of basic constraints is large, and listed below:

```
trait Constraint

case class AlternatingColors(on: MoveInformation) extends Constraint
case class Descending (on: MoveInformation) extends Constraint
case class HigherRank(higher: MoveInformation, lower:MoveInformation) extends
Constraint
case class IsAce(on: MoveInformation) extends Constraint
case class IsEmpty(on: MoveInformation) extends Constraint
case class IsFaceUp(on: MoveInformation) extends Constraint
case class IsKing(on: MoveInformation) extends Constraint
case class IsRank(on: MoveInformation, rank: Rank) extends Constraint
case class IsSingle(on: MoveInformation) extends Constraint
case class IsSuit(on: MoveInformation, suit:Suit) extends Constraint

case class NextRank(higher: MoveInformation, lower:MoveInformation, wrapAround:Boolean
= false) extends Constraint
case class OppositeColor(on: MoveInformation, other:MoveInformation) extends
Constraint
case class SameColor(on: MoveInformation, other:MoveInformation) extends Constraint
case class SameRank(on: MoveInformation, other:MoveInformation) extends Constraint
case class SameSuit(on: MoveInformation, other:MoveInformation) extends Constraint

case object Truth extends Constraint
case object Falsehood extends Constraint

case class IfConstraint(guard: Constraint, trueBranch:Constraint = Truth,
falseBranch:Constraint = Falsehood) extends Constraint
case class NotConstraint(inner: Constraint) extends Constraint
case class OrConstraint(args: Constraint*) extends Constraint
case class AndConstraint(args: Constraint*) extends Constraint
```

Figure 28: Basic Constraint Types

While going into depth about the specifics of each constraint is outside the immediate scope of this paper, additional examples of their usage and explanation are within the source code. Many constraints are fairly self-explanatory, however. For example, the `AllSameSuit` constraint returns true if a set of cards is all the same suit. Because constraints can be placed on both the source (the cards being moved) and the target (the location they are being moved to), both are checked when observing whether an action should succeed. To allow for multiple constraints to be applied, they can be chained together using the `AndConstraints` or `OrConstraints`, which are constraints themselves. These `OrConstraints` and `AndConstraints` simply determine the Boolean logic of all input constraints, returning the “Or” or “And” results respectively. An example of multiple constraints being compounded is shown below (from the Simplesimon variation):

```
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)) )
  }
```

Figure 29: buildOnFoundation Constraint Definition

This `buildOnFoundation` constraint is the logic for determining whether a given stack of cards can be moved to the foundation. Only a set of cards that are descending, all the same suit, and range from King to Ace can be placed onto the foundation. If no constraints are required for making a given move, this can be determined by the `Truth` constraint.

The last component for a move, `isSingleDestination`, is a simple Boolean flag that determines whether the destination for the moving components is a single source or multiple sources. This flag is set to true by default, however, if a move distributes cards to multiple tableaus (such as dealing from the stock deck), then this flag should be set appropriately. There are many components to a single `move` action, however many of these actions are similar across variations. Usually, it is the number and constraints of the `moves` that determine the variability of the rules between variations, even those within the same family. An example `Move` that could be added to the sequence of possible actions is shown below (using the `buildOnTableu` constraint from the previous subsection:

```
val tableauToFoundationMove:Move =
  Move("MoveCardFoundation",
    MultipleCards,
    Drag,
    movableElement=Column,
    source=(Tableau,Truth),
    target=Some((Foundation, AndConstraint(
      IsEmpty(Destination), buildOnFoundation(MovingCards)))))
```

Figure 30: Tableau to Foundation Move Definition

To summarize, this `tableuToFoundationMove` determines the action of moving a number of cards from the tableau to the foundation. This move allows for multiple cards to be dragged as a column. When coming from the tableau, there are no constraints, however when placing them into the foundation, the constraints `IsEmpty` and `buildOnFoundation` need to be met. Only when these constraints are satisfied, will the move be valid and proceed.

3.4.7 *logic* (logic = BoardState(Map(Foundation -> 52)))

Once the moves of a given variation have been determined, the final board state logic needs to be determined. This refers to how the user can win the game of solitaire. In the **Simplesimon** variation example, the user wins when the game is in a board state where all 52 cards are all in the foundation (shown below):

```
logic = BoardState(Map(Foundation -> 52))
```

All forms of the logic are in the form of a `BoardState` object, with an input `Map` applying the conditions to the listed elements. Another example is the Narcotic variation, wherein the user wins by emptying the stock and all tableaus:

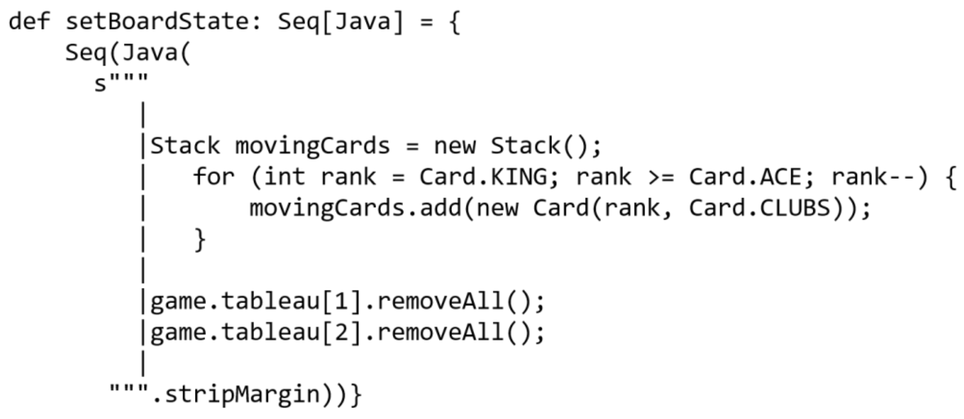
```
logic = BoardState(Map(Tableau -> 0, StockContainer -> 0))
```

3.4.8 *solvable* (solvable = false)

The next concept in the model is solvable. This is an optional value as when it is set to false then it will not be utilized in the variation. If the value is set to true, then an additional button is available to the user upon launching a variation: *solve*. This button then iterates through all possible potential moves for a given board state, effectively brute-forcing all available options until the most progress is made. The iteration of these moves is variation-specific and is defined within the `ExtraMethods` combinator in the variation family's domain. Even with a method of iterating through possible moves defined, this does not guarantee that any given board state is solvable. Many board states, depending on the variation and constraints for card movements, are not actually solvable: an intended facet of many types of real-life solitaire games.

3.4.9 *testSetup* (testSetup = Seq())

The final concept defined within the model is the optional `testSetup` of the variation. When set to an empty sequence, `testSetup` does nothing. However, `testSetup` can be set as a board state (in the form of a Java sequence) that defines a situation in which all possible moves defined by the `moves` trait can be tested. This is discussed further in the following section on testing, but below is how it looks within the model (this example comes from the **Simplevar** variation within the **Simplesimon** variation).



```
def setBoardState: Seq[Java] = {
    Seq(Java(
      s"""
         |
         |Stack movingCards = new Stack();
         |   for (int rank = Card.KING; rank >= Card.ACE; rank--) {
         |       movingCards.add(new Card(rank, Card.CLUBS));
         |   }
         |
         |game.tableau[1].removeAll();
         |game.tableau[2].removeAll();
         |
       """.stripMargin))}
```

Figure 31: TestSetup Java Board State

The intention of this design component was to create a simpler and more efficient way for users to create their own Solitaire variation with the framework. Instead of having to write out long sections of code to define various concepts for a variation, all a user needs to do is define a model with already built-in constraints and methods. Of course, anyone looking to pick up this framework to create a variation would still meet a learning curve and would need to take time to learn how to create a model and which constraints/methods are available to be used. The simplistic nature of the modeling should allow users to define all of the concepts that will generate a working variation faster than it would be to write out the entire variation by hand. Each concept within the model can be defined easily and picked up quickly by new users following through the tutorial discussing each concept. Even if users finds themselves confused when learning how to utilize a concept, the likeness between each variation within the entire Next-Gen Solitaire program allows users to look at how other variations define each concept and copy or modify similar concepts across the variations.

3.5 Testing

Originally, the unit testing capabilities of the framework only consisted of production element accountability. The tests mostly consisted of checking whether a certain element or constraint was generated properly within the Java code, confirming that the combinator components were produced correctly. This did leave an absence in logic testing however. Although it could be checked whether a certain Java component was produced, there was no way of ensuring that the intended logic of card movements, winning and move validity was correct. To remedy this missing component within the project and to improve scalability, a `TestSynthesis` combinator was added to the framework, allowing the option to generate JUnit 4 move test cases to ensure that the variation logic is being generated as intended. The user can choose to generate test cases by defining a `testSetup` concept in the model.

In order for the program to actually test solitaire moves, there must be a board state set where all of the cards are placed into an orientation that allows for the moves to be tested. When the user creates `testSetup`, they must define it as the board state that allows all of the moves to be tested and succeed. This means that the starting board setup should contain the state for performing a valid move, as the process involves negating a particular constraint to be tested individually. The manual construction of the board state is necessary so as to be able to test both user-defined and generic constraints. If there is no `testSetup` defined, or defined as an empty sequence, then no test cases will be generated and the combinator will not be added to the code synthesis.

An example board state for the Simplevar variation:

```
def setBoardState: Seq[Java] = {
    Seq(Java(
      s"""
         |
         |Stack movingCards = new Stack();
         |   for (int rank = Card.KING; rank >= Card.ACE; rank--) {
         |       movingCards.add(new Card(rank, Card.CLUBS));
         |   }
         |
         |game.tableau[1].removeAll();
         |game.tableau[2].removeAll();
         |
       """.stripMargin))}
```

Figure 32: Simplevar Test Setup

The test cases generated test each constraint for all possible moves and should succeed when the constraints are met and fail if any of the constraints are negated. Individual test cases are generated based on the variation's collections of move types defined in the `moves` concept, such as movements to piles or foundations. Based on the constraints defined for each type of move, a negating version of the constraint will be applied to the board state, innately negating a particular constraint. This allows the user to test for false validity after the negated version of a constraint is applied to a valid board state. By negating only a single constraint in each test, each constraint is tested separately. In parsing a given variation's moves, the validity of the board state is negated using a drop-in Java function, such as the example below:

```
def allSameSuitNegative(constraint: Constraint) : Seq[Statement] = {
      Java(
        s"""
           |movingCards = new Stack();
           |movingCards.add(new Card(Card.ACE, Card.CLUBS));
           |movingCards.add(new Card(Card.ACE, Card.HEARTS));
           |movingCards.add(new Card(Card.ACE, Card.SPADES));
           |""".stripMargin).statements()
    }
```

Figure 33: AllSameSuit Constraint Negation Function

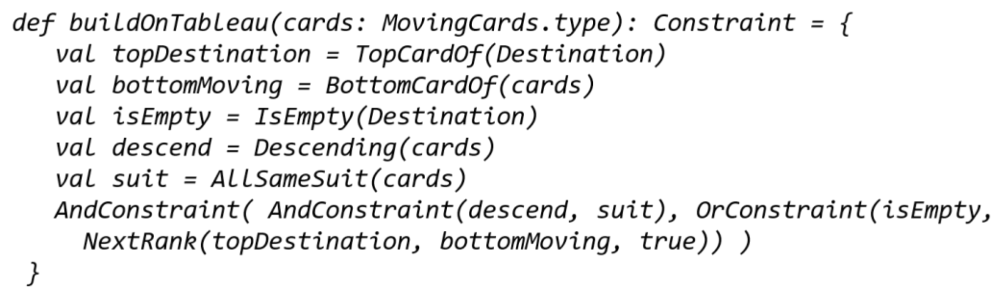
In this example, the given Java code is inserted into the variation if a variation's move contained the `allSameSuitNegative` constraint.

While this system is valid for predefined constraints, it does not cover user-defined constraints. To test a user-defined constraint, the test simply performs the move in which the constraint was derived from, without falsifying any constraints. If the user-defined function was created correctly, then it should already be reflected in the board state (as the state should be an environment that allows for any legitimate move for that variation). If the test for the user-defined constraint fails, then it can be reasonably assumed that there is either a problem in the constraint or a discrepancy in the assumedly-valid board state. In either case, there is an error in the model's logic that should be addressed.

Overall, the generation of test cases was designed to allow users to be able to test their variations easily. Similarly to the benefits of the model, the testing allows the user to save time and create tests in a much simpler manner than writing all of the code out by hand. Once the user learns the basics of setting up a test board state, they will be able to create test cases for any variation at a more efficient pace. Also, by creating a dynamic combinator that applies tests for specific moves, the scalability of testing increases with variation size and diversity.

3.6 Code Consolidation

Much of the project was designed with code consolidation in mind. When making solitaire variations, a lot of the same structure, rules, and concepts are similar between variations. A big emphasis on the design of the framework was to allow for the combination of code that is duplicated over many of the variations and families. Instead of forcing the user to have to define parts of the solitaire game that are shared between multiple families, this framework defines all of those aspects itself and generates the game code without any work by the user. Many methods that are utilized across families generate parts of the game such as the structure of the board, the orientation of the cards, and the deal. The user will never have to spend their own time writing out this method by hand, since it will be called within every newly-created variation. An example of this is constraints. Constraints (as defined earlier in this section) are small tests for the validity of a card or move. Throughout the entirety of the project, all moves in all variations can utilize the same list of constraints to test moves.



```
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)) )
 }
```

Figure 34: Combination of Constraints

The figure above displays the method that defines an acceptable move when placing a card onto a tableau in Simplesimon. `TopCardOf()`, `BottomCardOf()`, `Descending()`, `AllSameSuit()`, `AndConstraint()`, `OrConstraint()`, and `NextRank()` are all constraints that are not defined within `variationPoints.scala` for the family, but are still available to be used. Constraints are defined in the `constraints.scala` file and shared across all variations.

The only details that the user needs to define are the characteristics of each game that make it unique among all other variations. The model for each variation is where these differences are determined. Each separate concept is applied to the methods shared by each variation, in turn generating the game structure, rules, etc.

In addition to the consolidation of methods that are used between all families, the models also utilize shared variables within a family. Many of the concepts that the model defines can have attributes that could also be used by a different variation for that same concept. For example, more than one variation can have a move that utilizes the same deal or the same constraints on a move. Variations within the same family consolidate code by using the same method for many of their concepts within `variationPoints.scala`. Each variation within the family can use the variables defined there for their own model.

```
val structureMap:Map[ContainerType,Seq[Element]] = Map(
    Tableau -> Seq.fill[Element](numTableau())(Column),
    Foundation -> Seq.fill[Element](numFoundation())(Pile),
    StockContainer -> Seq(Stock(numStock()))
  )

  val map:Map[ContainerType, Seq[Widget]] = Map (
    Tableau -> horizontalPlacement(15, 200, numTableau(), 13*card_height),
    StockContainer -> horizontalPlacement(15, 20, numStock(), card_height),
    Foundation -> horizontalPlacement(293, 20, numFoundation(), card_height)
  )
```

Figure 35: Simplesimon Layout and Structure Maps

The two figures above are the structure and layout maps for the **Simplesimon** family. These maps are available to be utilized by any variation within the family. This is extremely useful when creating a new variation because the user does not have to rewrite new maps for each of the variations, they can just call upon the already existing ones within `variationPoints.scala`. Many more shared concepts can be defined, including the number of tableaus, stocks, foundations, deals, moves, etc. The models also offer the option to override anything in `variationPoints.scala` to allow for flexibility if another variation within the family needs to adjust any of the shared concepts. An example of this is shown in the figure below where **Simplevar** overrides the number of tableaus because it uses 8 instead of the 10 used in most variations of the **Simplesimon** family.

```
override def numTableau(): Int = {
    8
  }
```

Figure 36: Overriding the Number of Tableaus in a Family

Code consolidation within this project allows for the user to spend less time writing code while the framework generates repeated code and utilizes shared methods instead. This allows for an increase in productivity as there is no time wasted on rewriting much of the code that every game needs and shares.

4 Development/Evaluation

4.1 Start of Current Project Iteration

This iteration picked up after several years of development on the framework. In prior iterations, solitaire variations in Java were generated using CLS but there was no separate domain modeling. This resulted in a fragile code repository due to all the specialized knowledge of the different variations being embedded within the individual combinators. Eventually Scala domain modeling was added to the project and the process of modeling variations began. The synthesizer had been almost completely finished and many Scala models had been created which could generate lots of solitaire variations and families. While many models had been created, not all of these models could compile and be run as a working solitaire game. This iteration of the project picked up with finding the faulty variations and debugging them before continuing to create new variations.

4.2 Tools and IDE

The IDE used for this project was IntelliJ. Creating Scala models, working with the synthesizer, and running the solitaire games were all done in the same IDE. In previous iterations of this project, the generated Java code was opened in Eclipse and then run from there, however in this iteration of the project, that process was moved over to IntelliJ as well, using the modules feature. The generated solitaire games were imported as a new module, given the standalone.jar, and then run from the same project that held the models and synthesizer. For version control, GitHub was used and the project can be found at https://github.com/combinators/cls-scala.

4.3 Creating Solitaire Variations

In this iteration, the **Simple Simon** and **Baker's Dozen** families were added to the project. **Simple Simon** included the **Simple Simon** and **Simplevar** variations. **Baker's Dozen** included the **Baker's Dozen**, **Castles in Spain**, and **Spanish Patience** variations.

Baker's Dozen is a variation of solitaire with thirteen columns in the tableau, four cards in each column, and no stock. In order to win the game, all of the cards must be placed in the foundation. Cards can be moved around to different piles in the tableau if the moving card is the rank below the card on the top of the destination pile. For the classic variation of **Baker's Dozen** the suit does not matter when moving cards on the tableau. If the destination column on the tableau is empty, then only a king can be moved to that column on the tableau. Cards in the tableau can also be moved to the foundation. There are four foundation piles, each one of which starts as an empty pile. Cards must be placed into the foundation in ascending order starting with Ace and ending with King. Aces can be placed in any empty pile in the foundation, but every other card must be placed on the card of the previous rank with the same suit. For example, the Ace of clubs can be placed into an empty pile, yet the two of clubs must be placed on the Ace of clubs. The variation of **Castles in Spain** follows the same ruleset as **Baker's Dozen**, however, the cards in the tableau can only be moved to other columns in the tableau if the top of the destination is the same suit as the moving card. In the variation **Spanish Patience**, any card can be placed on an empty tableau, instead of just kings.

Simple Simon is a variation of solitaire which has no standard stock, as all the cards are dealt among the 10 tableaus. Cards can only be moved from one tableau to another if the moving set contains descending values of the same suit, and if the top of the receiving pile is one rank higher than the top of the moving cards. Stacks of cards may only be moved to the foundation if they consist of a full King to Ace descending same-suit stack. The game ends once all the cards are placed in the foundation. The **Simplevar** variation of **Simple Simon** wherein 6 cards are dealt among the first 4 tableaus, and 5 cards dealt among the remaining 4. All 8 remaining cards reside in the newly added stock, which distributes a new card to each tableau when clicked.

The intention of generating Java code that runs Solitaire variations from Scala models is to allow users to create new variations more simply and efficiently than writing out the Java code by hand. To create a brand-new variation, all the user has to do is define various elements of the Scala model and the code generation will do the rest. This severely cuts down on the time it takes for a user to make a working solitaire variation. However, while modeling is a much shorter and simpler process, there are still several steps that need to be followed to actually create the model for a variation.

A new package can be created for the family within the solitaire folder. Several files will be located here that define and control the models for the family and allow the variations to run properly. There are built-in templates within the project that allow for easier creation of these files. The package object of the family is where the concepts of the solitaire game are set within the model. Some of these concepts include the name, moves, and deal, among others. Many of these attributes are more complex than just a string or boolean and therefore are defined within the `variationPoints.scala` file and called in the package. `variationPoints.scala` defines concepts such as the structure, layout, and move constraints. Each variation is different and therefore no model will be able to be defined in the same way. Due to this, each different variation in the model can override attributes that `variationPoints.scala` sets.

Once the user creates a model, they can generate the Java code for their variation by running the run service in IntelliJ and then opening a web browser to *http://localhost:9000/<FamilyName>/<VariationName>*. This will load the inhabitation page. Here, a git repository with the generated code will be made. This repository can be cloned into a folder on a computer and executed with the standalone.jar as a dependency. Running this generated code should compile and display your working variation which can be played exactly how it was defined within the model.

4.4 Functioning Variations

In addition to creating and testing new variations, we also evaluated the current condition of existing variations within the NextGen-Solitaire framework. The following is a record of all working variations at the time of this document being written (as well as the route to which they can be accessed if the local server is running):

Variation (*family*)	**Route**
Archway	`http://localhost:9000/Archway`

Fan	`http://localhost:9000/fan/fan`
Alexander The Great (Fan)	`http://localhost:9000/fan/alexanderthegreat`
Fan Easy (Fan)	`http://localhost:9000/fan/faneasy`
Fan Free Pile (Fan)	`http://localhost:9000/fan/fanfreepile`
Fan Two Deck (Fan)	`http://localhost:9000/fan/fantwodeck`
Labelle Lucie (Fan)	`http://localhost:9000/fan/labellelucie`
Scotch Patience (Fan)	`http://localhost:9000/fan/scotchpatience`
Shamrocks (Fan)	`http://localhost:9000/fan/shamrocks`
Super Flower Garden (Fan)	`http://localhost:9000/fan/superflowergarden`
Trefoil (Fan)	`http://localhost:9000/fan/trefoil`
Free Cell	`http://localhost:9000/freecell/freecell`
All in a Row (Golf)	`http://localhost:9000/golf/allinarow`
Flake (Golf)	`http://localhost:9000/golf/flake`
Flake Two Decks (Golf)	`http://localhost:9000/golf/flake_two_decks`
Golf	`http://localhost:9000/golf/golf`
Golf no Wrap (Golf)	`http://localhost:9000/golf/golf_no_wrap`
Robert (Golf)	`http://localhost:9000/golf/robert`
Narcotic	`http://localhost:9000/narcotic`
Spider	`http://localhost:9000/spider/spider`
Spiderette (Spider)	`http://localhost:9000/spider/spiderette`
Scorpion (Spider)	`http://localhost:9000/spider/scorpion`
Mrs. Mop (Spider)	`http://localhost:9000/spider/mrsmop`
Gigantic (Spider)	`http://localhost:9000/spider/gigantic`
Spiderwort (Spider)	`http://localhost:9000/spider/spiderwort`
Baby (Spider)	`http://localhost:9000/spider/baby`
OpenSpider (Spider)	`http://localhost:9000/spider/openspider`

OpenScorpion (Spider)	`http://localhost:9000/spider/openscorpion`
Curds and Whey	`http://localhost:9000/spider/curdsandwhey`
Simplesimon	`http://localhost:9000/simplesimon/simplesimon`
Simplevar (Simplesimon)	`http://localhost:9000/simplesimon/simplevar`
Bakersdozen	`http://localhost:9000/bakersdozen/bakersdozen`
Spanish Patience (Bakersdozen)	`http://localhost:9000/bakersdozen/spanish_patience`
Castles in Spain (Bakersdozen)	`http://localhost:9000/bakersdozen/castles_in_spain`

Figure 37: All Currently-Working Variations

4.5 Generating Test Cases

As well as the generation of solitaire games from Scala models, this iteration of the project also began the creation of generating unit tests. These tests were designed with the intent of validating the generated Java code to ensure that the models were creating a solitaire game that followed all of the constraints created by the designer of the model.

Originally, the design of the unit test generation was to create tests that took the Scala model and generate Java tests based on the map and constraints defined in that model. The solitaire variation **Simplevar** from the Simple Simon family was used as the sample variation to generate the first test cases of the project. **Simplevar** includes a stock, tableau, and four foundation piles. Valid moves in **Simplevar** include moving a stack of cards from the tableau to another tableau or from the tableau to a foundation. Tests were created to determine if these moves were working properly by attempting to move cards from tableau to tableau and from tableau to foundation and checking to see if these cards successfully moved when they should. The tests look at a list of constraints that must be met for each move to be valid. These constraints are also defined in the model for each move. If all constraints are true then the test should pass. Likewise, if even one constraint is false then the test should fail. To test these moves and all of the constraints, a set of tests were created with one test testing all of the constraints as true and the rest of the tests testing each constraint individually as false and asserting false.

Two main challenges were faced when creating generated test cases in this way. The first problem was that the Scala model defined what a proper move was, however, the model did not define any specific way to organize the cards before testing a proper move. When creating the first generated test cases for **Simplevar**, Java code was manually written for the tests to organize the cards in such a way that a valid move could be tested. While this method worked for creating the first set of generated tests, this was not a long-term solution since every single variation would need to have this code manually written for each move to pass all of the tests. This was resolved by adding a valid board state to the models. This state is user-defined when creating the model and is a representation of a valid state that can be used to test all possible moves.

The second challenge was dealing with user-defined constraints. Some constraints were methods that were user-defined in the domain of the families and therefore could not be easily generated and tested along with the other Scala constraints. In **Simplevar**, the `AllSameSuit` constraint checks all of the cards in a stack and determines if they were all the same suit. This specific method is user-defined in the Java code in the **Simple Simon** domain and therefore it does not need to be converted to Java since it already exists in that format. The original solution to this problem was to have users define their own tests for user-defined components, however, we felt that this solution required too much explicit programming on behalf of the user. Our alternative and simpler solution is to leave all user-defined constraints unalerted. This means that user-defined constraints were testing by simply performing an unalerted move, with no validity negation occurring. As a result, when run with all other constraints being met, the move should succeed and the test should pass. The logic behind this decision was matching the user-defined model and the user-defined constraint. If the model and its constraints are constructed as intended, then a valid stack defined in the variations test setup should be able to validate without fail. However, if this simple test failed, then it would show a discrepancy between the intended model and the resulting generated structure. This would mean that either the model constraint logic is incorrect, or the user-defined function is not working as intended. In either case, it would signal to the user where there could be issues in the variation.

After creating tests for **Simplevar**, another solitaire variation in another family was also tested to make sure that the test generation would create tests properly for an entirely new game. The **Shamrocks** variation of the **Fan** family was selected. The unit test generation had to be altered slightly to fit with this variation since moves in **Shamrocks** are for one single card at a time instead of a pile of cards. Since every variation is different, there may be slight adjustments that need to be made in future iterations of this project for unit test generation. However, every variation that uses the same moves and constraints as **Simplevar** and **Shamrocks** will be able to auto-generate test cases for each constraint that pass successfully when given a valid board state in the model. `UnitTestCaseGeneration.scala` can be found in the shared Scala solitaire folder alongside all of the Scala variation models.

All test cases were generated in `UnitTestCaseGeneration.scala`. The `apply` method is the main method that goes through all of the moves of a given solitaire variation and generates test cases for them. The constraint tree for each move is flattened, allowing each constraint to be tested individually. For each move's constraints, the moving stack of cards is invalidated based on each respective constraint, effectively falsifying a move that would normally be valid. This is then asserted as false, confirming that reversing a particular constraint would result in an invalid move. Some additional logic checks are also determined for the particular move (for example, checking if the move is a deal or if a moving item is a single card instead of a pile). `UnitTestCaseGeneration.scala` takes information from the variation's model and defines variables for several of each game's key components such as the stock and foundation. For each variation, a valid board state must be set up for each move. This is another domain parameter that uses user-defined Java code to set the cards into a valid placement so that when a move is tested it will be successful. This method (`setBoardState`) is created within the Scala code of each variation and is located in the model as the `testSetup` variable.

To summarize the series of operations, firstly the `apply` method in `UnitTestCaseGeneration` will generate the `testSetup` Java code and then attempt to make a successful move. Then the move is tested several more times, once for each constraint. Every one of these tests has one negated constraint. These should all be asserted as false since each test should fail if even one constraint is falsified. The one exception to this is user-defined constraints which are not negated but just tested as true to prove continuity between the desired model and hardcoded constraint check.

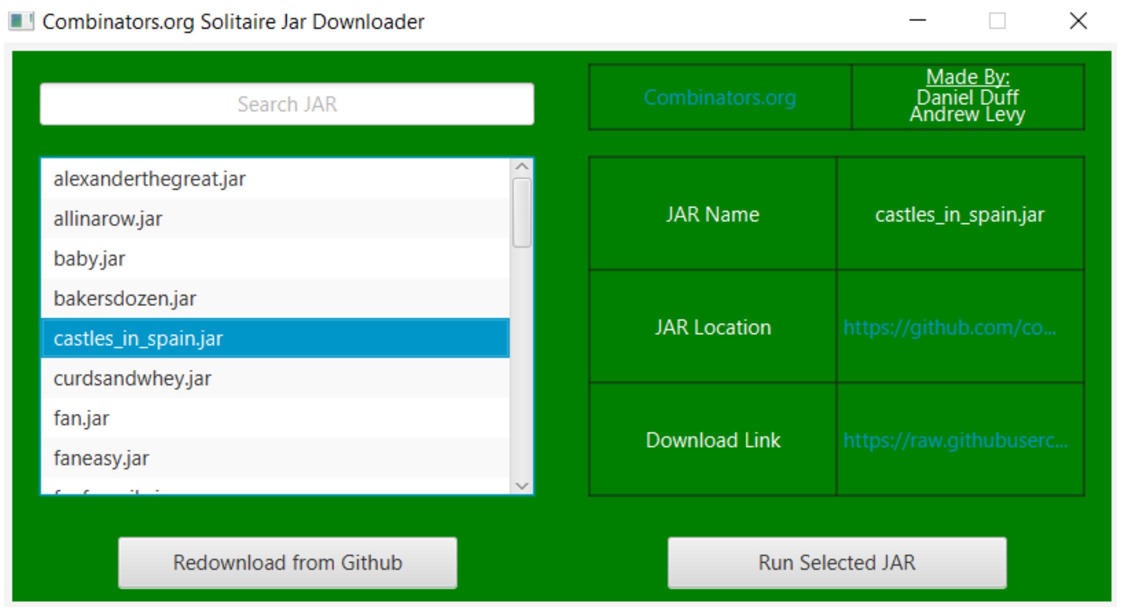
These steps are repeated for each move until all of the moves have successfully generated tests for each of their constraints. Tests will only be generated for a specific variation if an initial board state has been defined within the model, however. The `Controller.scala` file checks to see if the `testSetup` parameter is empty and if it is not then it creates a new `SolitaireTestSuite` for that variation.

4.6 JAR Collection & GUI Launcher

As the number of variations increased, more and more time was spent on configuring generated variations. While developing a framework model for a given variation was streamlined, the process of running the generated code in a development environment required additional steps to play the variation. As a result of this, all current variations were packaged in JAR form and stored in the Github Repository *Solitaire-Downloads*. (A link to this repo can be found here: https://github.com/combinators/solitaire-downloads). By creating a collection of working JAR variations, the process of configuring the generated code in an IDE was eliminated. If a user simply wanted to play a variation, they could download and run a given variation's packaged JAR file without any additional setup.

The online JAR collection also allows for the contributions of 3rd-party users. After developing a variation using the main Scala framework, a compiled JAR file can be submitted in the form of a pull request to the *solitaire-downloads* repository. This submitted JAR can then be tested and approved by any project administrator.

To further increase the usability and access of the JAR collection, a JavaFX-based GUI launcher was developed. While the main components of the Scala framework are publicly available, its complexity warranted a tool that 3rd-party users could easily access and understand.



JAR Name	castles_in_spain.jar
JAR Location	https://github.com/co...
Download Link	https://raw.githubuserc...

Figure 38: Solitaire-Player GUI JAR Launcher

The launcher, upon startup, collects the names, locations, and download links of all variation JARs. Using a simplified ListView JavaFX element, all variations uploaded to GitHub are listed to the user, which can be run by selecting the variation and clicking the *Run Selected JAR* button. The information can also be regathered from GitHub via the *Redownload from Github* button. Upon selecting the *Run Selected JAR* button, the selected JAR's download link is retrieved and a connection to the online resource is established. Once this process is completed, the JAR's Main class is loaded and run dynamically, starting the process as an extension of the currently-running launcher. Variations among the ListView can also be searched using an edit-distance fuzzy-search algorithm.

While the process of starting a JAR was simple, the closing of a launched application closes the entire Java Virtual Machine (JVM). As a result, closing the variation also closed the GUI Launcher. To alleviate this problem, an extension of the *SecurityManager* class structure was implemented to prevent the shutdown of the JVM through the closing of a variation.

The JAR launcher can be found at the Combinators *solitaire-player* repository (https://github.com/combinators/solitaire-player). In addition to having the source code for the launcher, it also holds a JAR that contains the program itself.

5 Conclusion

The current iteration of the project implemented useful features to improve the efficiency of the framework for new users. Coming into the project, there was little information regarding the creation of a variation, or the main components of the domain model. Also, the process of running a variation from the ground-up was tedious, especially with lacking and outdated documentation. Our goals coming into the project were to make the framework more approachable, as well as design a test suite combinator to validate solitaire moves defined in the domain. We believe we achieved our goals by improving the client framework, implementing a dynamic testing suite combinator, and developing a 3rd-party accessible JAR collection and GUI Launcher.

The improved client and tutorials allowed for users inexperienced with the framework to make and play their own solitaire variations as well as play the already existing variations. Test case generation was added to allow the user to be able to check the success of their models and the generated code, iterating through all of the domain's designated moves, and producing dynamically-appropriate JUnit 4 test cases. Among these larger changes, numerous bug fixes were remedied, and new variations added. A JAR database and launcher were also developed to streamline the ability to play and search approved variations. Third-party users are now capable of contributing to the system more easily than ever, with an online JAR database holding executable variations that can be approved via Github's pull request system.

While this iteration concluded most of the work from the previous iterations of Next-Gen Solitaire, there are still many more routes in which the project can be continued. The idea of utilizing code generation and combinatory logic synthesis to improve efficiency can be expanded much farther, not only in this project but also in other areas and programs in the future.

References

(Bessai et al., 2014) Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. 2014. Combinatory Logic Synthesizer. In ISOLA 2014 (LNCS), Vol. 8802. Springer, 26–40.

(Ciccarelli et al., 2019) Drew Ciccarelli, Ian MacGregor, Simon Redding, George Heineman. 2019. Generating Solitaire Games (https://web.wpi.edu/Pubs/E-project/Available/E-project-030119-174649/unrestricted/MQP_Report_Final.pdf)

(CleverMedia, 2020) Just Solitaire: The History of Solitaire [https://justsolitaire.com/history.html]

(Düdder et al., 2014) Boris Düdder, Moritz Martens, and Jakob Rehof. 2014. Staged Composition
Synthesis. In Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings. 67–86. [https://doi.org/10.1007/978-3-642-54833-8_5]

(Heineman et al. 2019), Jan Bessai, Boris Düdder, George Heineman, Jakob Rehof
Towards Language-independent Code Synthesis
ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2018.

(Heineman et al., 2015), George Heineman, Armend Hoxha, Boris Düdder and Jakob Rehof, Towards migrating object-oriented frameworks to enable synthesis of product line members. Software Product Line Conference (SPLC) [https://dl.acm.org/doi/10.1145/2791060.2791076]

(Hicken, 2018) Arthur Hicken Code Coverage and Automated JUnit Test Case Generation. Parasoft [https://blog.parasoft.com/code-coverage-and-automated-junit-test-case-generation]

(Pacheco & Ernst, 2005) Carlos Pacheco and Michael D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs [https://homes.cs.washington.edu/~mernst/pubs/classify-tests-ecoop2005.pdf]

(Randoop, 2020) Randoop Manual Version 4.2.2
[https://randoop.github.io/randoop/manual/index.html]

(Rehof, 2013). Jakob Rehof, Towards Combinatory Logic Synthesis
[https://pdfs.semanticscholar.org/8754/d21b46bf9b1a66b22936d1d71c377d3c1fdb.pdf?_ga=2.263244175.2001077669.1580746930-28288596.1580746930]

(SolitaireCentral, 2012) The History of Solitaire. The History of Computer Solitaire
[http://www.solitairecentral.com/history.html]

(Rehof & Vardi, 2014) Jakob Rehof and Moshe Y. Vardi. 2014. Design and Synthesis from

Components (Dagstuhl Seminar 14232). In Dagstuhl Reports, Vol. 4. Leibniz Zentrum für Informatik

(Tomassetti, 2018) Gabriele Tomassetti. A Guide to Code Generation [https://tomassetti.me/code-generation/]

(Tung, 2015) Angela Tung A brief history of Solitaire, Patience, and other card games for one. The Week [https://theweek.com/articles/558738/brief-history-solitaire-patience-other-card-games]