Release 3.1 (February 1998)

Writing Solitaire Games

Introduction

This document describes the tools provided by the Solitaire Kit for writing new solitaire-style card games. The kit consists of the program **Solitaire.app** (which provides dynamic loading services for game modules), the Solitaire framework, and a sample game module named **Template.solitaire** (from which all new games will evolve).

The restriction that new games be written within the same direcotry structure as the Solitiare source code has now been removed with the addition of the Solitaire framework. You'll likely want to copy the Template project and build from there, but as long as you've built and installed the framework you can build your new game wherever you'd like. Just make sure to set the Framework search path so Solitaire.framework will be found. The default is $(HOME)/Library/Frameworks.

See the section **Writing a New Solitaire Module** below for step-by-step details on beginning a new game project.

Solitaire.app

Solitaire.app and Solitaire.framework provides several classes and informal protocols for game writers:

- The **CardSet** classes, for displaying and manipulating piles of playing cards. These are contained in **CardSet.subproj**, with the header file **CardSet.h**. This subproject can be extracted from Solitaire for building card games outside the Solitaire framework. Full class documentation is located in **Solitaire/Documentation/CardSet**. Understanding these classes is essential for writing new games.

- An object of the Solitaire class (referred to as the **Solitaire object**) acts as the program controller, and vends access to general preferences. Call **[NSApp delegate]** to message the Solitaire object (this used to be through the S**olEngine(void)** function). The following messages can then be sent:

  ± (CardSize)**cardSize**
  ± (CardBack)**cardBack**
  ± (NXColor)**backgroundColor**

  These methods are rarely used in game module programming, as card size, card back, and background settings are handled automatically by the **GameModule** class.

  Solitaire includes the generic undo/redo code from **Draw.app**, and the Solitaire class is a subclass of **ChangeManager**, allowing future games to implement undo/redo.

- The **GameModule** class is the abstract superclass from which all game controller classes must inherit. It hides the details of game loading, CardSet preference setting, and provides methods for controlling generic aspects of game play such as starting a new game, displaying the rules, displaying the game inspector, remembering the location of game windows, and loading the game NIBS. Class documentation is provided in **Documentation/GameModule.rtf**. Use the code provided in **Template.[hm]** (in the Template project) as the starting point for writing a custom GameModule subclass.

- The **GamePref** class is the abstract superclass from which all game-specific preferences controllers must

inherit. It contains one method (± **registerPrefs**) which must be overridden if the game has preferences to register.

Template.solitaire and Game Modules

The **Template** project (which builds into **Template.solitaire**) provides a starting point for new games. The components of this (and every game) are:

- A GameModule subclass, whose class name matches the game name. A single instance of this class (referred to as the **game controller object**) is allocated the first time the game is selected in **Solitaire.app**; it is never freed. Access to the current game's controller object is always available through the **[*GameModuleClass* sharedInstance]** class method (as long as you define it... see any of the other game modules for an example) (This used to be accessed through the **SolGameController(void)** function). All messages sent from Solitaire.app to the game are sent to the game controller object.

  See the class documentation for **GameModule** for more details. Use the sample subclass **Template.[hm]** as the starting point for a new game.

- A GamePref subclass, in which the ± **registerPrefs** method has been overridden.

- A **localstrings.h** file in which localization of user-visible strings is performed. This should be compile into the file **Localizable.strings** using genstrings at the end of module development.

- **Engine.nib**, which is loaded once by (and owned by) the game controller object. This nib must contain a game inspector view, and an object of the custom GamePref subclass. **Delegates** of the card piles should also be in this nib file.

- **LargeGame.nib** and **SmallGame.nib** contain the "playing field" for the game, using the two different card

sizes. These nibs are loaded and freed as the active game and the card size preference is changed. They are owned by the game controller.

- **Rules.nib** contains a single panel in which the rules of the game are displayed.

Writing a New Solitaire Module

Preliminaries

First, select a name for the new game. It must be a single word (no spaces), and must be **UNIQUE AMONG ALL EXISTING SOLITAIRE MODULES**. Contact one of the game authors to informally *register* your name to prevent name clashes. For the purposes of this discussion, the name **MyGame** will be used.

Follow these (tedious) steps to create a new game project. Make sure you save all changes after each step. (Note that these instructions were not updated for PB 4.x)

- open the **PB.project** in the Template project, do a **make clean**, close the project
- make a **copy** of the Template project
- **rename** the **directory** to the name of your game (i.e. MyGame)
- rename **Template.h** and **Template.m** to **MyGame.h** and **MyGame.m**
- rename **TemplatePrefs.h** and **TemplatePrefs.m** to **MyGamePrefs.h** and **MyGamePrefs.m**

- in **MyGame**, open **PB.project**, go to **Attributes**, change the name from Template to MyGame
- go to **Files**, remove the **greyed out** class entries for Template.m and TemplatePrefs.m
- add **MyGame.m** and **MyGamePrefs.m**
- edit **MyGame.h**, change comments and name of class from **Template** to **MyGame**
- edit **MyGame.m**, change **#import "Template.h"** to **#import "MyGame.h"**; change classname to **MyGame**
- edit **MyGamePrefs.h**, change comments and name of class from **TemplatePrefs** to **MyGamePrefs**
- edit **MyGamePrefs.m**, change **#import "TemplatePrefs.h"** to **#import "MyGamePrefs.h"**; change classname to **MyGamePrefs**
- edit **localstrings.h**, change first string in **LOCALIZED_GAME_NAME** to **MyGame**; if you want the name displayed in the game selection list to be something different than the internal game name, replace the NULL with that string (this string can contain spaces, unlike the internal game name)
- in a Terminal, change into the English.lproj directory and run the following commands:

```
rm Localizable.strings
genstrings ../*.[hcm] > Localizable.strings
```

- open **Engine.nib** and make these changes:
  - drag **MyGame.h** and **MyGamePrefs.h** into the **classes suitcase**
  - go to **Objects**, change the class of **TemplatePrefs** object to **MyGamePrefs**; change the comment label
  - change the class of the **File's Owner** from **Template** to **MyGame**
  - go to **Classes**, delete the **Object/GameModule/Template** and **Object/GamePref/TemplatePrefs** classes
  - in the "**inspector**" panel, change the **comment label** to reflect the game name

- open **LargeGame.nib** and make these changes:
    - drag **MyGame.h** into the classes suitcase
    - go to **Objects**, change the class of **File's Owner** from **Template** to **MyGame**
    - go to **Classes**, delete the **Object/GameModule/Template** class
    - open the game window, change the **title** to match your game
- repeat the above steps for **SmallGame.nib** and **Rules.nib** (has a rules panel, not a game window)
- **build** the project
- if everything worked, you will have **MyGame.solitaire**; double-click it to try it out (assuming Solitaire.app is in your search path)

Coding a Game

The best way to understand how games work is to examine the source code for the existing games. An understanding of the **CardSet** classes and the **GameModule** class are crucial. In a nutshell, you lay out **CardPileViews** in **SmallGame.nib** and **LargeGame.nib**, and create **delegate** objects for these views. The delegates handle most aspects of game play.

Each game module is given a unique memory zone, so use the **allocFromZone:** style of memory allocation.

IMPORTANT: All class names, whether they are compiled into Solitaire.app, or loaded dynamically from a game module, must be unique. This includes the delegate classes created to handle game play. To reduce the chances of a name clash, preface all internal class names with the game name (or a reasonable subset). Neither Klondike.solitaire nor Pyramid.solitaire follow this rule(the luxury of being first).