Fashion Quest Documentation

Release Preview 2

Mike Cantelon

September 26, 2009

CONTENTS

CHAPTER

ONE

INTRODUCTION

Fashion Quest is an interactive fiction framework created to make text adventure games about fashion because we love fashion and we love text adventure games. Fashion Quest is written in the Ruby programming language and requires Shoes, a cross-platform GUI framework created by Why the Lucky Stiff.

Games are created in Fashion Quest by defining game elements using YAML and bits of Ruby. The framework includes two demonstration games. The game "Fashion Quest: Daydream" is very small and designed to demonstrate non-player character features. It lives in the *game* directory. The game "Pirate Adventure Knockoff" is a port of the 1978 text adventure "Pirate Adventure" by Scott and Alexis Adams and lives in the *pirate_adventure* directory.

To play either of these use Shoes to run *run.rb* and select which you'd like to play.

Thanks to Why the Lucky Stiff for creating Shoes and inspiring the creative use of computers!

1.1 Why Use a Framework?

Development of interactive fiction (IF) involves dealing with problems not inherent in many other realms of development, including parsing and game world simulation. Because of this a number of frameworks have been developed to deal with the IF domain.

Fashion Quest is a relative newcomer. Established frameworks include *Inform*, *Adrift*, and *TADS*.

ADRIFT is one of the most user friendly of the frameworks. It allows games to be created using a GUI. It is not, however, extensible, cross-platform, or open source.

Inform is one of the most elegant and established of the frameworks. It allows games to be developed either in natural language (Inform 7) or a specialize programming langague (Inform 6). It is extensible, cross-platform, open source, and supports automated game testing.

TADS is reputedly more powerful than Inform, but has a fairly steep learning curve.

When evaluating IF frameworks, an interesting site is Cloak of Darkness. This site links to implementations of the same simple interactive fiction game created using twenty different IF frameworks.

1.2 Design Goals of Fashion Quest

Fashion Quest was created with the goal of being somewhere between ADRIFT and Inform in ease of use.

The design goals are as follows:

- **Minimalist**

Fashion Quest was designed to be lightweight and easy to learn. The simulated world is as simple as possible. Game command syntax is defined using patterns rather than natural language rules.

- **Cross-platform**

  By leveraging Shoes, Fashion Quest is able to provide a consistant user experience across whether being used in Windows, Mac OS, or Linux.

- **Programmer-Friendly**

  Fashion Quest development is done using the Ruby programming language rather than a domain-specific programming language. This lessens the framework learning curve as there are many resources for those wishing to learn Ruby (and those who already know Ruby get a head start).

- **Extensible**

  Because Fashion Quest is open source, it is fully extensible. Default game engine behaviour can be overriden using monkey patching.

CHAPTER

TWO

OVERVIEW

2.1 Directory Structure

The *doc* directory contains Fashion Quest developer documentation.

The *engine* directory contains Fashion Quest application logic. Unless you want to play around with Fashion Quest's internals, you can ignore this directory.

The *standard_commands* directory contains standard game command definitions that can be shared between games.

Files pertaining used by specific games are put in *game directories*. These directories can be put into the same parent directory as the above directories and can be named anything. When Fashion quest starts, it will look through all directories at this level to see which ones contain config.yaml files (in which game name, etc., are stored). If only one game directory is found, Fashion Quest will automatically select it. Otherwise, a game selector will be presented to the user.

2.2 Game Elements

The elements that make up a game include the player, locations, props, non-player characters, doors, game state, commands, and transitions. All are defined using YAML with embedded Ruby.

Elements that the player may be able to carry are called *game components*. These include props, characters, and doors. Usually only props can be carried, but some games might require a character or door be carryable.

Game state allows ad-hoc game world conditions to be stored. In the demonstration game "Pirate Adventure Knock-off", for example, game state is used to record whether or not the tide is in.

Commands and transitions rely on Ruby logic to manipulate the other game elements. Commands are triggered by the user whereas transitions are triggered by the conditions of other game elements.

2.3 Game Directory Structure

The game directory contains a number of folders and files in which game configuration live.

- **Basic Configuration**

  Each game directory must contain a *config.yaml* file. This file contains high-level configuration parameters such as game title, window width/height, whether the game window should be resizable, startup message, and startup logic.

The basic look of a game can be tweaked using the startup logic. See the "Pirate Adventure Knockoff" demonstation game for an example.

- **Transitions**

  A game directory may contain a *transitions.yaml* file. This file defines any transitions.

- **Characters**

  A game directory may contain a *characters* directory in which files that define non-player characters are kept.

- **Commands**

  A game directory requires a *commands* directory in which files that define commands, or reference standard commands, are kept.

- **Doors**

  A game directory may contain a *doors* directory in which a file that defines doors is kept.

- **Locations**

  A game directory requires a *locations* directory in which files that define locations are kept.

- **Parsing**

  A game directory may contain a *parsing* directorry in which files related to the fine-tuning of command parsing are kept.

- **Player**

  A game directory requires a *player* directory in which a file that defines the player's attributes is kept.

- **Props**

  A game directory requires a *props* directory in which a file that defines game props is kept.

2.4 Built-in Commands

There are a number of built-in commands that don't appear in the command directories. These are: *restart*, *clear*, *load*, *save*, *load walkthrough*, *save walkthrough*, save *transcript*, and *compare to transcript*.

restart restarts the game. *clear* clears the command output. *load* and *save* allow the user to load or save their game progress to a file.

The other built-in commands are developer-oriented and discussed in the testing section.

CHAPTER

THREE

GAME WORLD ELEMENTS

Game world elements are defined using YAML, a human-readable standard used to describe data structures using text. Each game component must have a globally unique identifier.

3.1 Locations

Locations are places a player can visit during a game.

Each location is defined in its own YAML file within the 'locations' subdirectory of the game directory.

The example below, from the "Pirate Adventure Knockoff" demonstration game, defines a location with two exits: an exit to the north and an open window. The unique indentifier of the location is *alcove*.

3.2 Doors

Doors allow two or more locations to be connected. If a door connects more than two locations, when entering from one location you will end up at a random pick of the other locations.

Doors are defined in a file called *doors.yaml* within the *doors* subdirectory of the game directory.

The example below, from the "Fashion Quest: Daydream" demonstration game, defines a door that allows the player to travel between two locations. The door is locked by default, but may be opened using the *brass key* prop. The unique indentifier of the door is *door*.

3.3 Props

Props are items that players can interact with in the game. They may be portable items, such as a pack of cigarettes, or items that can't be carried, such as a dresser.

Props are defined in a file called *props.yaml* within the *props* subdirectory of the game directory.

The example below, from the "Fashion Quest: Daydream" demonstration game, defines a dresser located in a location with the unique identifier *bedroom*. The dresser can be opened by the player and contains another prop, a pack of *smokes*.

3.4 Characters

Characters are beings that players can interact with in the game.

Each character is defined in its own YAML file within the 'characters' subdirectory of the game directory.

The example below, from the "Pirate Adventure Knockoff" demonstration game, defines a character located in a location with the unique identifier *shack*. The pirate will accept the *rum* prop if the player gives it to him.

3.5 State

Game state is used to keep track of game conditions other than the state of other game elements. State can be referenced, or set, from logic within commands, transitions, and events.

One example from the *Pirate Adventure Knockoff* demonstration is tide state. Tide state is changed using transitions that set state using simple logic, such as the line shown below.

```
@state['tide'] = 'in'
```

CHAPTER

FOUR

GAME WORLD MANIPULATION

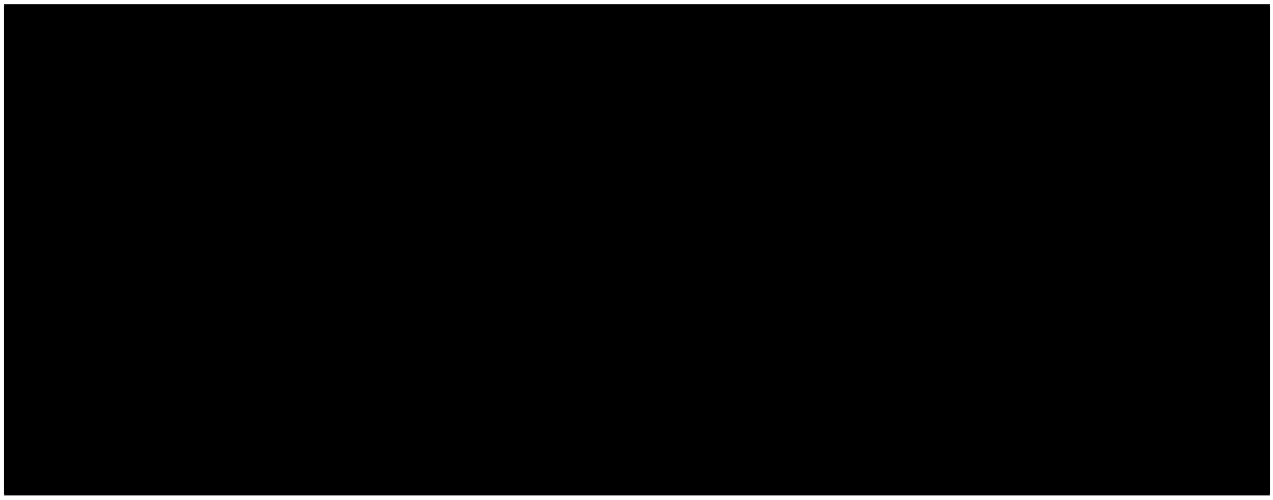
Game components are defined using YAML, a human-readable standard used to describe data structures using text. Each game component must have a globally unique identifier.

4.1 Commands

Each command is defined in its own YAML file within the 'commands' subdirectory of the game directory. If a command file within this directory exists, but is empty, the game engine will look for a command with the same filename in the *standard_commands* directory.

Note: Symbolic links can also be used, instead of empty command files, to point to standard commands but Windows doesn't support symbolic links so the game won't be cross platform.

The example below, from the "Pirate Adventure Knockoff" demonstration game, defines a command that enables the player to wake up the pirate character. If the character's *asleep* trait is *true* the *asleep* trait will be changed to *false* if the player enters the command *wake pirate*.



Commands are made up of syntax and logic.

TIP: Keep the idea of reusing commands between games in mind when creating commands. If logic is game-specific, try to use transitions instead of commands to implement the logic.

4.1.1 Syntax Forms

Command syntax can have multiple forms. For example, a command that allows the player to pick up a prop could have the form *get* <*prop*> or *take* <*prop*>.

Each syntax form is composed of keywords and parameters. Parameters are usually references to game elements. With the case of the above example *get* and *take* are the keywords and <*prop*> is the reference parameter.

Keywords are static words identifying an action: verbs. References refer to "things": nouns.

Four types of parameters can be used: prop references, character references, door references, and text.

Prop, character, and door references can refer to any prop, character, or door in the same location as the player. If a prop, character, or door is referenced, but doesn't have the same location as the player, an error will be output.

When defining syntax forms, parameters are enclosed in less-than and greater-than symbols. A reference paramter can be given the same name as its type or can be given a name. A syntax form containing the prop reference parameter <*prop*> would pass to the command a reference named *prop*. A syntax form <*prop:thing*> would pass to the command a reference named *arg['thing']*.

Text parameters are always named. A syntax form containing the ad-hoc reference <*colour*> would pass to the command the variable *arg['color']*.

Examples: - "<prop>" for unnamed prop reference - "<character>" for unnamed prop reference - "<prop:some name>" for a named prop reference - "<character:some other name>" for a named character reference - "<anything>" for an ad-hoc reference

4.1.2 Logic

Command logic is written in Ruby. References to props, characters, or doors can be passed in as specified by syntax forms.

In addition to data passed in via syntax forms, game elements can also be arbitrarily accessed.

@*game* provides access to game properties and methods.

@*player* provides access to player properties and methods.

@*props* provides access to the properties and methods of individual props.

@*characters* provides access to the properties and methods of individual characters.

The best way to understand how commands work is to check out the commands in the *standard_commands* directory.

4.2 Events

Events enable Ruby logic to be triggered by happenings in the game world. Characters, props, and doors can all have event outcome associated with them.

For example, the *cat* character in the "Fashion Quest: Daydream" demonstration game responds to two events: *on_attack* (when the cat is attacked) and *on_death* (when the cat is killed).

Commands can be used to trigger events. For example, the standard get command triggers the *on_get* event on a prop (and collects event output into the variable *on_get_output* by including the following line:

```
              = @game.            'on_get'
```

The "Pirate Adventure Knockoff" demonstration game uses the *on_get* event of the *book* prop to change the description of a room, revealing a secret passage, and return a hint to the player that something has changed.

If event YAML is set to be a list of event outcomes then an outcome will be randomly selected from the list when the event is triggered, as an example shows below.

```
- "The deadbeat squints at you and shuffles his feet before answering.\n"
- "The deadbeat tilts his head sceptically before answering.\n"
```

4.3 Transitions

Transitions enable Ruby logic to be triggered by happenings in the game world. Transitions are more versitile than events: any game condition(s) can be used to trigger the manipulation of any game element(s).

To add transitions to a game, create the file *transitions.yaml* in the appropriate game directory. Transitions are made up of one or more triggering conditions and one or more outcomes.

The example transition below, containing conditions and outcomes extracted from the "Pirate Adventure Knockoff" demonstration game, shows a transition that makes a pet leave if neither his master nor food are present.

If you want a transition output to not return output, end it with a line containing only “”.

CHAPTER

FIVE

FINE-TUNING

5.1 Parsing

Fashion Quest includes employs a crude, case-insensitive, parsing mechanism that converts player input into "lexemes": text elements (single words or game element identifiers such as "hat" or "blue hat") that can be compared to command syntax forms.

Here's the basic flow of parsing:

1. Abbreviated commands are expanded
2. Input is broken into lexemes (single words or game element indentiers)
3. Lexemes that match synonyms are replaced
4. Lexemes that match garbage words are deleted
5. Lexemes that are aliases to game element IDs are resolved

5.1.1 Abbreviated Commands

Abbreviated commands reduce the amount of typing the user must do. One popular convention in interactive fiction is allowing a user simply to enter the first letter of the direction (s)he'd like to go in: *n* for *go north*, for example.

The *command_abbreviations.yaml* file, in the *parsing* subdirectory of the game directory, allows a list of abbreviations for specific command instances to be defined using YAML. The file from the "Pirate Adventure Knockoff" demonstration game is shown below.

Note: The *command_abbreviations.yaml* file isn't the only place command abbreviation can be specified. Command syntax forms that don't contain parameters, like those of the *inventory* command, allow abbreviations to be stored in the command's syntax forms (*i* for inventory, for example).

5.1.2 Synonyms

- the global_synonyms.yaml file, in the game parsing path, allow a list of words that should be replaced with other words
- for example: "using" for "with"
- in the above example, this synonym would elimiate the need to make the syntax "attack <character> with <prop>" also work when the player issues the command like "attack bear using hat"

5.1.3 Garbage Words

- the garbage_words.yaml file, found in the game's parsing path, allows certain words to be discarded from player input
- these words should be words like "the" and "a" which have little semantic meaning
- this makes specifying command syntax easier

5.1.4 Aliases

Explain about aliases.

5.2 Testing

Testing interactive fiction games can be tedious. To make testing easier Fashion Quest provides a couple of simple tools in the form of built-in commands.

In addition to the built-in commands, the Shoes *alert* function is handy for confirming logic is being executed. *alert('Hello)*' will, for example, pop up a dialog box with the word "hello".

When there are syntax errors in game logic, or other errors that stop game execution, you can often get useful clues by pressing Alt-/ to view the Shoes debugging console.

5.2.1 Walkthroughs

When a player loads or saves a game, via the built-in *load* and *save* commands, all game element definitions are included in the game save. Because of this, these commands aren't very useful for testing.

Walkthroughs, on the other hand, can save a sequence of commands needed to arrive at a certain point in a game. This makes them useful for functional testing. Walkthrough files are simply a YAML list of commands.

To create a walkthrough, simply start you game and play it until the point at which you'd like your walkthrough to end. Entering the command *save walkthrough* will then allow you to save the walkthrough. When you wish to use a walkthrough, start or restart Fashion Quest and enter the command *load walkthrough*.

An example walkthrough is provides for the "Pirate Adventure Knockoff" demonstration game. It lives in the *pirate_adventure* directory and is named *complete_walkthrough*. It cycles through all the commands needed to win the game. Once the walkthrough has loaded, enter the command *score* and the win will be confirmed.

5.2.2 Transcripts

While walkthroughs are good for confirming nothing is broken, transcripts provide a way to confirm no output in a game has changed.

The built-in command *save transcript* will save the game output to a file. You can then make changes to your game, enter the commands needed to arrive at the point in the game where you originally saved the transcript, and use the built-in command *compare transcript* to compare the game output to the original transcript.

5.3 Naming

noun noun_cap

proper Noun Noun plural the nouns The nouns general a noun A noun countable water Water

CHAPTER

SIX

INDICES AND TABLES

- *Index*
- *Module Index*