# Fashion Quest Documentation

***Release Preview 2***

**Mike Cantelon**

September 20, 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 requires Shoes, a cross-platform GUI framework created by Why the Lucky Stiff.

Text adventure games involve issuing tiny action plans, like *go north* or *fight the police*.

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 Interactive Fiction Frameworks

Text adventure games are also known as interactive fiction (IF). Development of IF involves dealing with problems not inherent in many other realms of development, including parsing and simulation. Because of this a number of frameworks have been developed to deal with the IF domain. Major 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 extensible, cross-platform, or open source, however.

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.

## 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 lessen the framework learning curve for those who already known Ruby.

- **Extensible**

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

CHAPTER

TWO

# FRAMEWORK 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.

CHAPTER

THREE

# CREATING GAME COMPONENTS

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.

## 3.1 Creating Locations

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 Creating 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 Creating 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 Creating Characters

- characters are put in the 'characters' subdirectory
- characters are defined in YAML
- any props a character will accept being given are indicated by the "exchanges" property

## 3.5 Creating Commands

- commands are defined in YAML
- command files are put in the commands directory of the game's base directory
- within the commands directory, commands can be placed in subdirectories if desired
- commands are made up of syntax and logic
- command syntax can have multiple forms - each syntax form is composed of keywords and references - keywords are static words: verbs - references refer to "things": nouns
  - types of references: prop, character, door, ad-hoc
  - 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 doesn't have the same location as the player, an error will be returned
  - ad-hoc references can refer to anything... there is no checking before passing an ad-hoc reference to command logic

  - in syntax form, references enclosed in less-than and greater-than symbols
  - 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
- command logic is written in Ruby - variables are passed to the command logic - references are passed as variables
  - “<prop>” unnamed prop reference passed as “prop” variable
  - “<character>” unnamed character reference passed as “character” variable
  - “<prop:some name>” named prop reference passed as “arg[’some name’]” variable
  - “<character:some other name>” named character reference passed as “arg[’some other name’]” variable
  - “<anything>” ad-hoc refrence passed as “arg[anything]”
  - “game”, “player”, “characters”, “props” variables allow interaction with game engine and game data - “game” spec - “player” spec - “characters” spec - “props” spec
  - example commands
- commands can be shared between games - if you want to share a command between games, put the command in the standard_commands dirctory - to include a shared command in a game, put an empty file in your game’s command directory with the same filename

    as the shared command
  - alternately, if you’re not using Windows you can use symlinks

## 3.6 Events

- characters and props can have events associated with them
- built-in character events are: on_death, etc.
- props can also have events
- events can contain text and/or logic
- to trgger events, simply add the logic into a command
- **for example, the get command could trigger an “on_get” event in a prop:** output << game.event(props[prop], ‘on_get’)
- the above example would have the game check a certain prop for an on_get response
- **an example response could be:** @locations[’alcove’].add_to_description(“There is a bookcase with a secret passage beyond.n”); “There’s a strange sound.n”;
- the above example would add to a rooms description and give the player a hint that something in the game has changed

## 3.7 Transitions

- transitions allow manipulation of components to be triggered by game conditions, rather than user commands
- this allows you to avoid having to add the same logic in multiple commands
- transitions return output

- if you want a transition to be silent, end it with the line “”

## 3.8 State

- state can be used to keep track of global game conditions
- state can be referred to or manipulated in transitions or commands
- within transtions, use @state
- example: @state['tide'] = 'in'
- within commands, use game.state
- example: game.state['tide'] = 'out'

CHAPTER

FOUR

# FINE-TUNING

## 4.1 Style

- whenever possible, keep game-specific "rules" in transitions instead of commands
- keeping game-specific logic out of commands allows them to be reused in different games

## 4.2 Abbreviations, Synonyms, and Garbage Words

- the command_abbreviations.yaml file, in the game parsing path, allows a list of abbreviations for specific command instances to be defined
- for example: "n" for "go north"
- in the above example, "north" was a parameter to the "go" command
- in the case of abbreviations that don't need to specify a parameter, like "i" for "inventory", those should be included as a syntax of the command itself
- 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"
- 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

## 4.3 Testing

- when saving and loading games be mindful that prop, etc., definitions get saved as well so your game changes may not be reflected
- use "save walkthrough" command to save your previous commands
- use "load walkthrough" command to run through commands you've previously saved

- these commands can be used for testing
- walkthrough files are YAML, so easy to edit
- COMPARE, TRANSCRIPT?
- use the Shoes debugger... press alt-/ to see error messages
- alert('some message') is also handy to deduce the flow of logic
- no need to check for prop or character locations in commands because command parser will return error if prop or character referenced doesn't exist or isn't located near player

## 4.4 Naming

noun noun_cap

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

CHAPTER

FIVE

# INDICES AND TABLES

- *Index*
- *Module Index*