

---

# **Fashion Quest Documentation**

***Release Preview 2***

**Mike Cantelon**

September 20, 2009



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Interactive Fiction Frameworks . . . . .	1
1.3	Design Goals of Fashion Quest . . . . .	1
<b>2</b>	<b>Game Creation Overview</b>	<b>3</b>
2.1	Directory Structure . . . . .	3
2.2	Game Configuration . . . . .	3
2.3	Game Elements . . . . .	3
<b>3</b>	<b>Creating Game Components</b>	<b>5</b>
3.1	Creating Locations . . . . .	5
3.2	Creating Doors . . . . .	5
3.3	Creating Props . . . . .	5
3.4	Creating Characters . . . . .	5
3.5	Creating Commands . . . . .	5
3.6	Events . . . . .	6
3.7	Transitions . . . . .	7
3.8	State . . . . .	7
<b>4</b>	<b>Fine-Tuning</b>	<b>9</b>
4.1	Style . . . . .	9
4.2	Abbreviations, Synonyms, and Garbage Words . . . . .	9
4.3	Testing . . . . .	9
4.4	Naming . . . . .	10
<b>5</b>	<b>Indices and tables</b>	<b>11</b>



# 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 consistent 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 for those who already know Ruby.

- **Extensible**

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

# GAME CREATION OVERVIEW

## 2.1 Directory Structure

The *docs* 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 Configuration

Basic game configuration is done via a *config.yaml* file. Each game directory must have one.

Game configuration options include game title, window width/height, whether the 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” demonstration game for an example.

## 2.3 Game Elements

GAME STATE?

The elements that make up a game include the player, locations, props, non-player characters, doors, 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.

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





# CREATING GAME COMPONENTS

## 3.1 Creating Locations

- locations are put in the 'rooms' subdirectory
- exits can be directions "north" or arbitrary names "stairs"

## 3.2 Creating Doors

- doors are put in the 'doors' subdirectory
- doors connect two or more locations
- 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

## 3.3 Creating Props

- props are put in the 'props' subdirectory

## 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 character 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 reference 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 directory - 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 trigger 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 transitions, use `@state`
- example: `@state['tide'] = 'in'`
- within commands, use `game.state`
- example: `game.state['tide'] = 'out'`



# 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 eliminate 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

# INDICES AND TABLES

- *Index*
- *Module Index*