# Fashion Quest Documentation

***Release 1.0 Alpha***

**Mike Cantelon**

September 06, 2010

# 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. Fashion Quest has been tested with Shoes 2.

Games are created in Fashion Quest by defining game elements using YAML and bits of Ruby. The framework includes three 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. The game "Cloak of Darkness" is a port of a game that was created as a means of comparing different interaction fiction frameworks. It lives in the *cloak_of_darkness* directory. There is also a game skeleton exists primarily to serve as the basis of new games. It lives in the *new_game* 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 language (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.

The Cloak of Darkness site is useful for comparing interaction fiction frameworks as it contains 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 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 (a parrot, for example) or door (a teleportation device, for example) 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.

## 2.2 Framework Directory Structure

The *framework directory* is the directory containing *run.rb*. Its file structure is explained below. The *new_game* directory contains a skeleton you can use as the basis of a new game.

Table 2.1: Framework directory file structure

| **Directory/File** | **Description** |
|---|---|
| doc | Directory containing developer documentation |
| engine | Directory containing framework engine logic |
| standard_commands | Directory containing standard game command definitions that can be shared between games |

Files used to define games are put in *game directories*. These directories can be put inside the *framework directory*. Every *game directory* must contain a *config.yaml* file (in which basic game configuration is stored). When Fashion quest starts, it will look through all directories in the *framework directory* to see which ones contain a *config.yaml* file. If only one game directory is found, Fashion Quest will automatically start this game. Otherwise, a game selector will be presented to the user.

## 2.3 Game Directory Structure

The *game directory* file structure is explained below.

Table 2.2: Game directory file structure

| **Directory/File** | **Description** |
|---|---|
| config.yaml | File containing YAML basic game configuration |
| transitions.yaml | File containing YAML transition definitions |
| characters | Directory containing YAML character definition files |
| commands | Directory containing YAML command definition files |
| doors | Directory containing the YAML door definitions file |
| locations | Directory containing YAML location definition files |
| parsing | Directory containing parsing-related YAML configuration files |
| player | Directory containing the YAML player definition file |
| props | Directory containing the YAML props definition file |

## 2.4 Built-in Commands

There are a number of built-in game commands that don't appear in the *standard_commands* directory. 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

Each game component must have a globally unique identifier. There are two ways to define game components: either using YAML, a human-readable standard used to describe data structures using text, or using Ruby. This documentation focuses on the use of YAML, but if you're interested in using Ruby check out the *Cloak of Darkness* implementation for an example.

## 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 defines a location with a number of exits. The unique indentifier of the location is *entrance*. Each exit has a destination, which is the unique identifier of the location to which it leads. Note that the *stairs* exit has a description: "upstairs". This is used to describe travelling this way. For example a character taking this exit will be described using "the hobo goes upstairs" rather than "the hobo goes stairs".

```
entrance:
  exits:
    north:
      destination: yard
    south:
      destination: hallway
    up:
      destination: upstairs
    stairs:
      destination: upstairs
      description: upstairs

  description: |
    You are in the sunlit entranceway of a sizeable home.

    To the north is a door leading outside. Stairs lead upwards. A hallway leads south.
```

## 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 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*.

```
door:
  description: The door is made of dark-brown wood.
  locations:
  - hallway
  - bedroom
  traits:
    opened: false
    open_with:
    - brass key
```

## 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 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*.

```
dresser:
  description: The dresser looks like it has seen better days.
  location: bedroom
  traits:
    opened: false
    portable: false
    contains:
      - smokes
```

Props, as the example below shows, can have one or more aliases. The aliases can be used by players to refer to the prop.

```
safety sneakers:
  aliases:
  - sneakers
```

Props can have traits set that determine what can be done with them.

### 3.3.1 Portability

If a prop has its *portable* trait set to true, the player will be able to take it. When props are defined using *props.yaml* the *portable* trait gets automatically set to true if not otherwise specified.

### 3.3.2 Visibility

If props have their *visible* trait set to true, these props will be automatically shown when the player looks. When props are defined using *props.yaml* the *visible* trait gets automatically set to true if not otherwise specified.

### 3.3.3 Text

If a prop has its *text* trait set, the prop can be read. *text* may be set to text to be shown to the player or, if the first character is ">", a text file in the game folder.

```
leaflet:
  description: The leaflet is faded yellow and seems to warn against something.
  traits:
    text: "The leafet is about men's rights. It thinks the child support is wrong.\n"
```

### 3.3.4 Containers

If a prop has the *open* trait set to false it can contain other props. These props are specified using the *contains* trait. The props may require other props to open them, if the *open_with* trait is set.

```
box:
  description: The box is made of wood.
  location: field
  traits:
    opened: false
    open_with: hammer
    contains:
      - stamps
```

### 3.3.5 Size

The *size* trait can be used to prevent large props from passing through doors that have lesser *size* traits defined.

The "crack" door in the Pirate Adventure demo game, for example, has a size of 1.

```
---
crack:
  name: crack
  description: The crack is too narrow to bring large items through.
  locations:
  - top_of_hill
  - cavern
  traits:
    size: 1
    opened: true
```

The size of the crack prevents the player from entering it if carrying props, such as the book and the shovel, that have a size of 2.

### 3.3.6 Construction

A prop can be specified as being built from other props. This is done by setting the *build_with* trait to the component props. If any of the component props should be taken out of play, they should be includes in the *build_consumes* trait.

```
table:
  traits:
    build_with:
    - lumber
    - nails
    - saw
    - hammer
    build_consumes:
    - lumber
    - nails
```

### 3.3.7 Get With

If you need to have one or more props to get another (a bottle, for example, to get water), you can set the *get_with* trait of a prop.

```
water:
  plural: true
  location: ocean
  traits:
    get_with:
    - bottle
  events:
    on_get: |
      "The bottle holds the water.\n"
    on_drop: |
      "The bottle is now empty.\n"
```

### 3.3.8 Buried Props

If a prop has its *can_dig* trait set to true it can be used to dig. If a prop has its *buried* trait set to true it can be dug up. When a prop is dug up its *portable* and *visible* traits get set to true. Below is an example of a buried prop.

```
treasure:
  description: This treasure is really something else. It might be worth something, even!
  location: yard
  traits:
    visible: false
    buried: true
    portable: false
```

### 3.3.9 Wearables

If a prop can be worn by the player, set the *wearable* trait to true.

```
shirt:
  description: "The t-shirt evokes the desire to party."
  traits:
    wearable: true
```

### 3.3.10 Flammables

Locations can be set to be dark, in which case a player needs a source of illumination to see the description. If a prop has the trait *lit* set to false the player will be able to light on fire using a prop that has the *firestarter* trait set to true. If the prop has its *burn_turns* trait set to a number then it will only burn for that number of turns.

```
torch:
  location: attic
  traits:
    lit: false
    burn_turns: 150
```

### 3.3.11 Support

If a prop has the *supports* trait set to true, other props can be put on it. If the prop has the *supports_only* trait set to one or more props, only these props will be supported by it.

```
hook:
  description: "A hook on which to hang a garment."
  traits:
    visible: false
    portable: false
    supports: true
```

## 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 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.

```
---
pirate:
  description: The pirate has a wicked look.
  location: shack
```

### 3.4.1 Mobility

Characters will wander from location to location if their *mobility* is set. Mobility is the probability (in percentage) that the character will move each turn. The character example below will go to a new location each turn.

```
---
cat:
  location: bedroom
  mobility: 100
```

### 3.4.2 Aggression

Characters will be prone to attack the player if their *aggression* is set. Aggression is the probability (in percentage) that the character will start to attack each turn. A character's *strength* determines how much damage it can do each attack if they don't posess a weapon prop (the *default_attack* property determines how the weaponless attack will be described). If a character does have a weapon prop with a greater attack strength than their default, the character will automatically use it in attacks.

The character example below has a 5% chance of turning hostile and will do one or two hit points of damage each turn.

```
--
cat:
  location: bedroom
  mobility: 100
  description: The cat is small and agile.
  hp: 2
  strength: 1
  aggression: 5
```

A prop can serve as a weapon if the prop's *attack strength* is set. If a character possesses a weapon, this weapon will be used if its *attack strength* is greater than the character's *strength*. The prop example below is possessed by the "deadbeat" character and gives the character a strength of 7.

```
shiv:
  attack strength: 7
  description: The shiv looks sharp... useful.
  location: deadbeat
```

### 3.4.3 Communication

Characters can be asked questions about topics. Topics and responses are put into the *discusses* setting. The example below shows a character that, when asked about a "party", "parties", or "partying", responds with one of two opinions about the topic.

```
---
rick:
  discusses:
    ?
      - party
      - parties
      - partying
    :
      - >Parties are a real gas.
      - >I'd like to think that I'm in it for the party.
```

If the letter ">" is the first character of a response, double quotes will be put around the remaining characters of the response before outputting to the player.

Characters can also be made to occasionally mutter random things or be described as doing random things. The example below shows a character that has a 10% chance, each turn, of either being described as looking at the player or as saying something.

```
child:
  location: hallway
  description: "The child look sad. The child has no shoes."
  mutter_probability: 10
  mutters:
  - The child looks at you sadly.
  - >Why is the world against me?
```

### 3.4.4 Trade

Characters may be willing to accept props as gifts or for trade.

In the example below the character will accept the gift of rum.

```
---
pirate:
  description: The pirate has a wicked look.
  location: shack
  exchanges:
    rum: true
```

In the example below the character will give a pair of shoes and a shiv in exchange for smokes.

```
---
deadbeat:
  exchanges:
    smokes:
    - shoes
    - shiv
```

### 3.4.5 Portability

Characters can be set to allow the player to carry them, as with the example below.

```
---
parrot:
  description: The parrot looks great.
  location: shack
  traits:
    portable: true
```

### 3.4.6 Logic

Characters can execute custom Ruby logic each turn. In the example below the parrot character will, if in the same location as the player, occassionally eat a cracker if the player possesses crackers.

```
---
parrot:
  logic: |
    output = ''

    # Parrot interaction
    if @location == @player.location

      # The parrot occasionally eats a cracker if player has them
      case rand(3) + 1
        when 1:
          if @props['crackers'].location == 'player'
            output << "The parrot ate a cracker.\n"
          end
      end
    end
```

## 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 game, 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

Once game elements are defined, they can be manipulated in the game by commands, events, and transitions.

## 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*.

```
---
syntax:
- "wake <character>"
- "wake up <character>"

logic: |

  output = ''

  if character.traits['asleep'] == true
    character.traits['asleep'] = false
  else
    output << "You can't wake up what is not asleep.\n"
  end

  output
```

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.

### 4.1.3 Noun Grammar Contexts

Characters, doors, and props can be referred to, when outputting messages to the user from commands, using noun grammar context functions. Checking out how noun grammar context functions are used in the standard commands may help you understand them.

Noun grammar context functions work with three basic noun types: singular ("hammer", "cat", etc.), plural ("bottles", "bullets"), and proper nouns ("Rick"). Proper nouns will always be capitalized.

The four noun grammar context functions are explained below.

#### noun

*noun* is the most used noun context. It prefixes with "the " for non-proper nouns. If a noun is proper, it is capitalized. If a specific *name* setting has been assigned to the object this will be used, otherwise the object's unique ID will be used as a name.

Example: "You take #{prop.noun}."

#### noun_cap

*noun_cap* is generally used at the start of sentences. It works like *noun*, but capitalizes the first letter.

Example: "#{prop.noun_cap} contains something."

#### noun_direct

*noun_direct* works like *noun*, but prefixes with "a " (for singlular) or "some " for (plural) for non-proper nouns.

Example: "You find #{prop.noun_direct}."

#### noun_direct_cap

*noun_direct_cap* is gnerally used at the start of sentences. It works like *noun_direct*, but capitalizes the first letter.

Example: "#{prop.noun_direct_cap} falls from the sky."

### 4.1.4 Conditions

Command conditions are logic that determines if a command can be carried out by the player. Conditions can be defined for individudal commands and/or globally for any command.

Condition logic should return a hash with a "success" element (boolean to indicate whether or not the attempted command should be carried out) and, optionally, a "message" element that returns information to the player.

Command conditions for individual commands may be defined in a command's YAML file, as shown below.

```
condition: |
  if @props['crackers'].location == 'player'
    {'success' => true, 'message' => "Crackers give me power.\n"}
  else
    {'success' => false, 'message' => "I NEED CRACKERS.\n"}
  end
```

A global command condition can be specified in the *command_condition* element of a game's config.yaml file.

## 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 three events: *on_attack* (when the cat is attacked), *on_death* (when the cat is killed), and *on_pet* (when the player pets the cat).

```
---
cat:
  location: bedroom
  mobility: 100
  strength: 1
  description: The cat is small and agile.
  hp: 2
  aggression: 5
  default_attack: its claws
  events:
    on_attack:
    - "The cat yowls as he leaps at you.\n"
    on_death:
    - "The cat shrieks as it crumples to the ground.\n"
    on_pet:
    - "The cat purrs and rolls around.\n"
    - "The cat makes a strange cooing sound.\n"
    - "The cat stretches and purrs.\n"
```

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:

```
on_get_output = @game.event(prop, '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.

```
book:
  description: The book is large and blood-soaked.
  location: alcove
  events:
    on_get: |
      if not (@locations['alcove'].has_exit('passage'))
        @props['book'].traits['visible'] = true
        @locations['alcove'].add_to_description("You see a secret passage.\n")
        @locations['alcove'].set_exit('passage', 'passageway')
        "There's a strange sound.\n";
      end
```

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.

```
on_discuss:
- "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.

```
---
conditions:

  ?
    - @player.location == @characters['parrot'].location
      && @characters['parrot'].location != @characters['pirate'].location
      && @characters['parrot'].location != @props['crackers'].location
  :
    - parrot_flies_off
```

```
outcomes:

  parrot_flies_off: |
    if @props['crackers'].location != 'player'
      @characters['parrot'].location = @props['crackers'].location
      "The parrot flies off looking very unhappy...\n"
    end
```

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

CHAPTER

FIVE

# FINE-TUNING

Each game's *config.yaml* file allows fine-tuning of a number of elements.

Table 5.1: Game settings

| **Setting** | **Description** |
|---|---|
| title | title of the game (displayed in the window bar) |
| width | width of game window |
| image_height | maximum height of images |
| resizable | whether or not the player should be able to resize the game window |
| startup_message | text to display to the player upon startup |
| background | Background image for game |
| command_condition | Ruby logic that determines whether any command should execute |
| setup_logic | Ruby logic to set up game (see demo games for examples) |

Other elements that can be set in *config.yaml* are explained later in this section.

## 5.1 Scoring

If a game incorporates scoring, player actions can result in score increases that generally serve as a marker of game progress. The standard command *score* will let the player know of his or her current score.

Scoring items are configured in *config.yaml*. An example is shown below.

```
scoring:
  cloak_on_hook:
    points: 1

  message_read:
    points: 1
```

Each scoring item has a name and point value. Scoring is triggered using Ruby logic. The Ruby logic *@game.set_score('<name of scoring item>')* will trigger a scoring item.

## 5.2 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

### 5.2.1 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 each game's directory, allows a list of abbreviations for specific command instances to be defined using YAML. An example is shown below.

```
---
n: "go north"
s: "go south"
e: "go east"
w: "go west"
u: "go up"
d: "go down"
```

Command abbreviations can alternatively be specified in *config.yaml*. An example is shown below.

```
command_abbreviations:
  n: "go north"
  s: "go south"
  e: "go east"
  w: "go west"
  u: "go up"
  d: "go down"
```

**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).

### 5.2.2 Synonyms

Synonyms help reduce the number of command syntax forms needed to support command syntax variations. The *global_synonyms.yaml* file, in the game *parsing* subdirectory of each game's directory, can contain a list of word substitutions defined using YAML. These substitutions get applied to a user's command input.

For example, the word "using" can be replaced with the word "with". This synonym would make the command syntax form "attack <character> with <prop>" work when the player enters the command "attack bear using hat".

Example YAML is shown below.

```
---
using: "with"
examine: "look"
```

Synonyms can alternatively be specified in *config.yaml*. An example is shown below.

```
global_synonyms:
  using: "with"
  examine: "look"
```

### 5.2.3 Garbage Words

Garbage words are words with little or no semantic meaning (like “the” and “a). By removing them from the user’s command input we recude the number of command syntax forms needed to support command syntax variations. The *garbage_words.yaml* file, found in the *parsing* subdirectory of each game’s directory, can contain a list of words that should be discarded from player input.

Example YAML is shown below.

```
---
- "the"
- "that"
- "this"
- "at"
```

Garbage words can alternatively be specified in *config.yaml*. An example is shown below.

```
garbage_words:
- "the"
- "that"
- "this"
- "at"
```

## 5.3 Testing

Testing interactive fiction games can be tedious. To make testing easier Fashion Quest provides a couple of simple tools: walkthrough and transcripts (explained in detail later on).

In addition to these tools, 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”. The Shoes *error* function is also handy. *error(“Hello”)* will write the world “Hello” to the console.

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.

### 5.3.1 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.

### 5.3.2 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.