ResponsIF

User's Guide
Version 0.65

Table of Contents

Revision History

Version	Who	Date	Change
0.1	J. Nabonne	2015-05-27	Initial revision.
0.2	J. Nabonne	2016-03-03	Next noted rev. More added. More to come.
0.3	J. Nabonne	2016-03-27	Call markup and dynamic user interface responses.
0.4	J. Nabonne	2016-03-28	Added fuzzy logic section.
0.41	J. Nabonne	2016-03-29	Minor additions.
0.42	J. Nabonne	2016-04-03	Changed relative vs absolute attributes strategy.
0.45	J. Nabonne	2016-04-03	Fleshed out "Organizing" section.
0.46	J. Nabonne	2016-04-05	Minor fixes.
0.5	J. Nabonne	2016-04-14	JavaScript integration plus "uses first"
0.6	J. Nabonne	2016-05-21	Added start of more advanced topic handling
0.65	J. Nabonne	2016-06-23	Added .onto. Updated setup cases.

Introduction

"I offer this definition of interactivity: A cyclic process between two or more active agents in which each agent alternately listens, thinks, and speaks." (Chris Crawford, *Chris Crawford on Interactive Storytelling*)

"We design the rules that shape her experience, her choices, her performance. Rules are how we communicate. Verbs are the rules that allow her to communicate back. The game is a dialogue between game and player, and the rules we design are the vocabulary with which this conversation takes place." (Anna Anthropy, *A Game Design Vocabulary)*

"call-and-response: a statement quickly followed by an answering statement; *also*: a musical phrase in which the first and often solo part is answered by a second and often ensemble part" (Mirriam Webster Dictionary)

ResponseIf is a response-based system for creating Interactive Fiction, whether that means old-school, text adventure type games, "choose your own adventures", or even something that perhaps only you have so far dreamed up. It is designed to be flexible and general purpose. ResponsIF is written in JavaScript, and its native text format is HTML.

Boring Historical Stuff and Design Motivation

[This section is optional but might be interesting to some. If you want to get right into things, you can skip it.]

The history of ResponsIF begins with my discovery of Aaron Reed's game *Blue Lacuna*. I had been playing some text adventure games and trying to create my own, and his implementation of single keyword input in addition to the usual "verb noun" format of standard text adventure commands was inspirational for me. I was using Quest at the time – as opposed to Inform, which his extension was written for – and I began thinking of how to implement something like that myself, but for my own work. I was also helping out in the Quest forums, and I began to notice a pattern to the sorts of questions new authors were asking:

– How can I have characters with a variety of responses?

– I want my room descriptions to change depending on <xxx>.

– I want my object descriptions to vary depending on what the player does.

– How can I display occasional, random text after each turn?

– How can I change the help text?

– How can I implement hints?

What I noticed is that everyone wanted to vary the responses their games made in more or less the same way, but across all aspects of the game. Whereas things like room and object descriptions were often thought of as static, more enterprising game authors wanted them to be as dynamic as conversation might be with an in-game non-player character. And that's when I had my epiphany: as far as the interactive fiction I was seeing,

everything was a conversation.

Conversation concepts could be applied not only to characters but to rooms and objects and pretty much everything, including the help system or even in-game hints. In fact, the basic structure of player input followed by a game response could be perceived as the back-and-forth of conversation itself. When you were playing a text adventure game, you were, in a sense, having a conversation with the game.

With that in mind, I created a "response library", one which had a simple core engine that had one purpose: select and activate responses based on topical keyword input. The engine had no real knowledge about the domain of interactive fiction. It had no intrinsic notion of rooms or objects or taking things or dropping things or opening things. All it knew about were responses, and responses would be used to build everything, but in a general, flexible way.

That engine was used in *spondre*, my first completed IF game.

I was happy with the way the library worked – and the game proved it was reasonably viable – but I was dissatisfied by three things. First, I was using Quest objects to hold my response information, and the editor was not set up to input request information in an optimal way. It was necessary to edit the XML by hand, and editing XML was not something I would expect non-technically savvy authors to do. That was a huge barrier to anyone but me using it.

Second, I was beginning to run into performance problems in Quest, not so much in the response library but in other things I was trying to do. That didn't bode well for the future.

Third, there was no clear and straightforward way to convert Quest games into a more universally

deployable format (e.g. something that could be put on a mobile device), something I definitely wanted to do. *[These deficiencies could possibly be cured by Alex Warren's new QuestJS project.]*

I decided to start over and rewrite the "response library" as a standalone JavaScript engine. Along the way, it has picked up features not even dreamed of in the original library (e.g. built in support for animation). And who knows where it will go next?

The result is *ResponsIF*.

Some Terminology

There are some terms that will be used over and over in this document.

riff: Some IF is game-like (and therefore called a "game"). Some IF is more story-like (and therefore called a "story"). And some IF is something else entirely. Given that we need some way to talk about what can be created with ResponsIF, neither "game" nor "story" suffices as an umbrella term, and words like "work" and "piece" are too vague and clunky. Since this is my party, I'm going to coin a term for a ResponsIF creation: a "riff" A *riff* is the set of authored responses as well as any HTML and CSS used to create the desired user experience.

topic: A *topic* is a subject, an idea, a thing, an event, a concept, a feeling, a matter of discourse. It can be as mundane as "lamp" or "desk", as active as "take" or "look", as lofty as "love" or as specific as "WHAT_JASPER_LOST". It can be a physical aspect of a room or an ephemeral thought. The important thing is that it's something to be responded to.

response: A *response* is a basic unit of behavior. A response can show text on the screen, it can change riff world state, it can move objects around and much more. Responses are how the riff communicates with the player. They can also be used by the riff to offer options to the player to select player actions. Responses typically occur in relation to topics.

call: A *call* is the trigger for a response. Calls are what generate responses. (This is often combined as "call and response" in music and certain interactions between speaker and listener.) The player will typically call topics into the riff through some user interface mechanism. Responses can also call their own topics, to cause subsequent behavior. Topics can also be called by text markup to insert dynamic responses in place in the output text stream.

responder: A *responder* is anything in a riff that can generate responses. The player character is a responder. If the player is in a room, the room can be a responder. Other objects in the room with the player can be responders. The help system can be a global responder. The user interface itself can be a responder.

actor: An actor is a responder with *actions,* responses invoked at the end of each turn. Actors have the ability to behave autonomously, outside of directly responding to user input.

parent: Responders may be contained within other responders. In this case, the containing responder is termed the "parent", and responders contained within the parent are known as its "children". The parent provides a "response context" for its children. Responders can be moved from one parent to another by responses.

Hello, world

To jump right into things, in the grand tradition of computer language tutorials, here is a simple riff that prints out, "Hello, world!"

```
.responses player
    .response START
        .does
            .says Hello, world!
.end
```

Result:

Hello, world!

Let's analyze this a bit.

An immediate question may be, "What's up with all the dots?" This is how a ResponsIF document is formatted, as a sequence of dotted keywords and associated data. The leading dot for a keyword was chosen to minimize conflict with actual text. A primary goal of the riff format is to allow text to be entered in as pure a form as possible, without the need for external quotes or other special delimiting characters.

It's also worthwhile noting that there is no syntactic significance to the indentation. Unlike some languages which rely on white space or line breaks to delimit aspects of a file, indentation and line breaks in ResponsIF are used solely at an author's discretion. The only white space that is significant is that which separates tokens and values. The following variations of the above file are all valid and equivalent:

```
.-----------
.- Variant 1
.-----------
.responses player
    .response START
        .does .says Hello, world!
.end
.-----------
.- Variant 2
.-----------
.responses player
    .response START .does .says Hello, world!
.end
.-----------
.- Variant 3
.-----------
.responses player .response START .does .says Hello, world! .end
```

As can be seen from the last line, some indentation is recommended to aid with readability. Long blocks of text can be split across multiple lines as well, as HTML also tends to ignore white space.

Starting at the first line, we have

```
    .responses player
```

This line begins the definition of responses for the player. There will always be at least one responder in a riff, the current "point of view" responder. The default point-of-view responder is named "player", but this can be changed if desired.

Responses will be defined until an .end token is reached.

Next, we have

```
.response START
```

This begins a new response, in this case for the topic START. The START topic is sent by the boilerplate index.html file when a new riff is begun. This response will match that topic.

```
.does
```

This begins the part of the response that describes what the response does when it is processed. It contains the "stuff to be done", which is executed in order.

```
.says Hello, world!
```

This is heart of the response. It outputs the string "Hello, world!"

```
.end
```

This is the terminator for the player responses.

In an attempt to keep the language as simple as possible, much of the explicit bracketing of content found in other languages is implicit in ResponsIF. In particular, there is no "end" marker for a response. A response is implicitly ended when either a new response begins or the end of the overall responses group is reached. There are very few constructs that use an actual ending token.

Extending the "hello world" example, we can add a second response for the same topic.

```
.responses player
    .response START
        .does
            .says Hello, world!

    .response START
        .does
            .says (And greetings to all our ships at sea.)
.end
```

Result:

Hello, world! (And greetings to all our ships at sea.)

This shows two important things. First, more than one response can trigger for the same called topics. The above example is a bit contrived, but this ability will come in handy in a number of situations, especially ones that involve giving the player multiple options or where multiple responders are present in the same context.

Second, the "says" output is simply concatenated, piece by piece. Responses will typically be processed in the order they are encountered, but there are ways to control output order in cases where that is important. Responses can also have classes, to group them together.

Interactivity

We now know how to make our riff speak. How do we make it listen?

This is where the user interface comes into action. ResponsIF doesn't dictate a particular interface but it does support a wide number of possible variants:

1) In-text links

2) Fixed links and buttons

3) Type-in areas (e.g. text parsers, keyword input)

4) Anything else you can craft in HTML

The function of the user interface is to allow the player to directly or indirectly make "calls" to generate further responses. Links and buttons can call topics. Type-in areas can be used to input varying topics. Links can come and go as desired.

At this time, ResponsIF doesn't come with a built in command line parser. The reason for this is that far better minds than mine have created far better parsers than I would probably come up with, with development spanning decades. ResponsIF should be easily integrable with any parser that can be used to generate keyword topics for JavaScript. (The original Quest version of this library was integrated with the existing Quest parser.)

ResponsIF is designed to work on the semantic level. It is not really meant to handle syntax. A response that matches the topics "look apple" will also match "apple look" and even more serious digressions like "don't look at the apple". Keyword matching simply does not work well for input validation. Believe me – I gave it a try once!

Being HTML-based, however, links are a natural – and built-in – form of user input. We will be using links for the remainder of this document. Keep in mind, though, that anything that can translate user input into topics can be used.

Linking to Topics – Link Markup

The most basic form of built-in user input in ResponsIF is the topic link. This allows the player to click on parts of the text to initiate additional responses based on what they have clicked on. This is nothing new: the Internet has been doing this since its beginnings. What is different is the way ResponsIF handles the clicks. Instead of navigating to a new page, links call topics which are then handled by responses. What happens then depends on what the responses have been set up to do.

In order to show how this works, let's begin work on a simple story riff.

When the riff starts, we want to print out some initial text:

```
.responses player
    .response START
        .does .says Once upon a time, there lived a king in a faraway kingdom.
.end
```

This is a start, but it's static. We want to create links so the reader can navigate further into the story. Let's start with a response for the word "king".

```
.responses player
    .response START
        .does
            .says Once upon a time, there lived a {!king!} in a faraway
                kingdom.
    .response king
        .does
            .says The king was getting on in years, and he was concerned
                that his kingdom would fall into chaos upon his death.
.end
```

The notation "`{!king!}`" is called *link markup*, and is the first kind of markup we will encounter in ResponsIF. (There are others – for example, *state markup* and *call markup* – which we will get to later.) The exclamation point is also known as a "bang", and in writing, it can indicate action or excitement! It is also used in the Linux world to signify the *execution of a command*.

Assuming an appropriate CSS style has been set up, the reader will see the link highlighted in some way (the style could even be no highlight). The following shows it with a basic style:

Clicking on the word "king" will call the topic "king", and any matching responses will be triggered:

Notice that, even though the text is broken across lines in the riff, it is displayed as a single line in the output. In order to add line breaks in the output, you would need to use <p> or <br> tags, per HTML content rules.

This is all quite straightforward and simple. And more links can be added for other words.

Let's says that instead of a single word, you'd like to create a link for a phrase – perhaps "upon his death." You could wrap it in link markup just as before:

```
.response king
    .does
        .says The king was getting on in years, and he was concerned
            that his kingdom would fall into chaos {!upon his death!}.
```

When that link is clicked, it will call with the topics "upon", "his", and "death". You could easily create a response for those three words like so:

```
.response upon his death
    .does
        .says (some more text)
```

And that would work. However, there is a way to have a link call topics not necessarily in the text.

Perhaps you view "upon his death" as being conceptually related to "mortality", and you'd like to have that be the topic. You can do that like this:

```
.response king
    .does
        .says The king was getting on in years, and he was
            concerned that his kingdom would fall into
            chaos {!upon his death|mortality!}.
```

By inserting a pipe character ('|') and then the desired topic or topics, you're letting ResponsIF know that you'd like to call those topics instead of using the words in the actual text. This is a handy feature if, for example, what's in the text isn't text but arbitrary HTML – for example, you can wrap link markup around an image, and then specify the topics to call when it's clicked.

(An aside: for those of you who use Twine, I have to say that I had never seen Twine's usage of the pipe character for this exact same purpose until long after I had come up with it myself. All I can figure is that programmers must look upon that little character the same way.)

Link Lifetimes

In the preceding example, clicking on a link caused more text to be output, but the original link still remained. As more and more text is generated, more and more links will likely end up in the text. This could lead to a lot of links, especially links which are no longer relevant.

What can be done about this?

One solution happens automatically. ResponsIF will remove links in the text (converting the link to normal text) following a simple rule:

A link is removed when there are no longer any eligible responses for the link's topics.

We haven't seen yet how to vary response eligibility, but the next section introduces one of them.

Say Again?

Consider our example again of the passage with "king" as a link. After the reader clicks the link and sees the new passage, if they click the link again, it will print out the exact same text, in true knee-jerk fashion. And it will do it again and again...

For some designs, this is expected and fine. Other designs avoid the problem by clearing the screen after each click. Some remove the link when it is no longer desirable.

Those options are available in ResponsIF. However, one of its design goals is to allow more dynamic handling of such situations. Let's explore some options and, in the process, learn some new features.

All responses track how many times they have been invoked. This allows a riff author to either

1) limit how many times a response will occur, or
2) provide different behaviors for different occurrences.

To limit how often a response is eligible to occur, simply add an "occurs" token and value to the response:

```
.response king
    .occurs 1
    .does
        .says The king was getting on in years, and he was concerned
            that his kingdom would fall into chaos upon his death.
```

By specifying "occurs 1", the response will only occur one time. After that, it will no longer be considered eligible to be run. In this case, the "king" link will no longer have any eligible responses, and it will be converted by ResponsIF into normal text – the link will be removed. There will be more refined uses for "occurs" later, especially in situations involving multiple matching responses with different priorities.

Another solution in this case is to vary the response based on how often the response has been triggered. Here is an example:

```
.response king
    .does 1
        .says The king was getting on in years, and he was concerned
            that his kingdom would fall into chaos upon his death.
    .does
        .says The king needed to find a solution to his dilemma.
```

This illustrates the use of multiple "does" blocks. One has the value "1", and the other has no value. When a response is being processed, if there is a "does" block that matches its current occurrence count, it will use that "does" block. Otherwise, if there is a "does" block with no occurrence value, it will use that, as default. Otherwise, it will do nothing.

Now, the first time the response occurs, it will print out the text beginning, "The king was getting on in years..." Every time after that, it will print out the text in the other "does" block. This has made the situation slightly better, as at least one additional bit of variability now exists. However, after the first occurrence, it will keep outputting the same text over and over again. So what can we do now?

There are some ways to refine this further.

One option is to make the subsequent text "auto hide". This means that the text will output as usual, but then it will disappear on the next user input. This can be used to briefly show text that doesn't impact the long-term story and which the author may not want to be a permanent part of the story transcript.

To make text "auto hide", add the "autohides" token following the text:

```
.response king
    .does 1
        .says The king was getting on in years, and he was concerned
            that his kingdom would fall into chaos upon his death.
    .does
        .says The king needed to find a solution to his dilemma. .autohides
```

Another option is to make the inconsequential text random. This can often give a sufficient illusion of variability.

```
.response king
    .does 1
        .says The king was getting on in years, and he was concerned
            that his kingdom would fall into chaos upon his death.
```

```
        .does
            .uses random
                .response
                    .does .says The king needed to find a solution to his dilemma.
                .response
                    .does .says The king pondered his problem.
                .response
                    .does .says The king was a good man facing a tough decision.
            .end
```

This introduces a "uses" block, in this case specifying to select a random nested response. There are some things to note about this.

First, "uses random" begins a *group of responses*. As such, it requires an "end" token. Second, the responses within the group do not have topics. At the point that the parent response is being executed and the random response is being selected, all the topic matching has already occurred. Any topics at that point would be ignored. Finally, while this example is only outputting text, responses can have other behaviors as well, which we will explore later.

The above two options can be combined: make each of the random response output text "auto hide" by adding the "autohides" token. Then the response will print out randomly selected text, hiding it once the reader has input a new command.

Of great importance to note in this is the way that responses can be nested. And the same control you have at the top level of responses (e.g. "occurs", multiple "does" blocks, etc.) can be used at any level, to an arbitrary level of nesting and complexity. You could, as an example, have the first random response only be used one time (with ".occurs 1"). After that response was used, the engine would then randomly select from the two remaining responses.

Offering Choices

Assume we have the following responses, based on all we have done so far:

```
.responses player
    .response START
        .does
            .says Once upon a time, there lived a {!king!} in
                a faraway kingdom.
    .response king
        .does 1
            .says The king was getting on in years, and he was
                concerned that his kingdom would fall into
                chaos {!upon his death|mortality!}.
        .does
            .uses random
                .response
                    .does .says The king needed to find a solution to
                                his dilemma. .autohides
                .response
                    .does .says The king pondered his problem. .autohides
                .response
                    .does .says The king was a good man facing a tough
                                decision. .autohides
            .end
.end
```

The next response to implement is for "mortality". At this point, let's begin to offer the reader some

choices to allow them to steer how the riff will unfold.

When a response is selected for processing, normally the response is simply executed, performing whatever actions the response is designed to take. However, if the response is assigned a prompt, then it will show the prompt as its first stage of processing, and the prompt will be clickable. If the player clicks the prompt, *then* the response will be executed. If another call is made (for example, a different link is chosen), then any prompt will go away. This allows an extra level of verification to responses that have stronger effects. Since more than one response can be selected by the same topics at once, this can be used to create a form of menu selection, where the prompts form the choices.

As an example, the topic "apple" might provide responses with prompts "Eat apple", "Throw apple", "Save apple", and "Cook apple". Selecting a prompt will execute that response, and all the prompts are then removed from the output (they "auto-hide").

Let's add some choice to our "king" story for the topic "mortality".

First, we'll have a lead-in response for "mortality". This will set up the menu to follow.

```
.response mortality
    .does
        .says One day, the king felt it was time to share his worries
              with someone close to him.
```
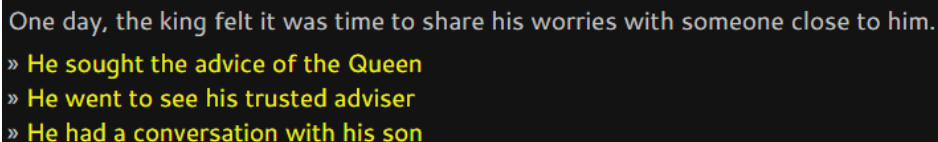
Then we have our three choices, each with a prompt.

```
.response mortality
    .prompts He sought the advice of the Queen
    .does
        .says He decided to seek the advice of the Queen, who often had insights
              that he himself did not.

.response mortality
    .prompts He went to see his trusted adviser
    .does
        .says He decided to seek the advice of his trusted adviser, whom he
              considered quite wise.

.response mortality
    .prompts He had a conversation with his son
    .does
        .says He decided to have a conversation with his son, the most
              likely heir to his throne.
```

All four of the new responses have the same topic: "mortality". That means that they will all match "mortality" at once and will all be processed. The ones with prompts will show their prompts. Here is a sample output of those responses. (This is a sample style. The style can be customized via CSS.)

Once a prompt is chosen, the prompts will disappear, and the selected response will be executed. (If something besides one of those prompts is clicked, then the prompts will auto-hide anyway, but none will be executed.)

If the player selects the second choice, then this is the result:

The prompts have been removed, and the second response has output its text.

Grouping Responses

Looking at the responses we created in the last section, we can see a certain amount of duplication. We have four responses all responding to the same topic. Moreover, the responses are all meant to be used together, as part of the same menu. It would not only reduce the number of riff lines but also make it clearer if there was some way to group those responses together,

We can accomplish this by creating a containing response to hold the related responses:

```
.response mortality
    .selects all
        .response
            .does
                .says One day, the king felt it was time to share his worries
                    with someone close to him.
        .response
            .prompts He sought the advice of the Queen
            .does
                .says He decided to seek the advice of the Queen, who often had
                    insights that he himself did not.
        .response
            .prompts He went to see his trusted adviser
            .does
                .says He decided to seek the advice of his trusted adviser, whom
                    he considered quite wise.
        .response
            .prompts He had a conversation with his son
            .does
                .says He decided to have a conversation with his son, the most
                    likely heir to his throne.
    .end
```

This will select all the contained responses for inclusion in the response candidates for a topic – but only if it itself is considered eligible to be executed (in this case, if the topics include "mortality"). If the containing response is not eligible, then the inner responses will be ignored.

Note: A "selects" block is a response group and, therefore, needs an "end". Also, a response with a "selects" block will only be used to hold responses. It will never be executed directly, as it is removed from potential response candidates when its children are added; any "does" block will be ignored.

Setting World State

So far, we have been creating responses to show text and to allow the reader to make choices. None of those choices, though, have had any lasting effects on the internal state of the riff. It's not required that responses do so. However, it is often useful to remember what has been chosen or otherwise have choices influence the riff state in a more long-term way.

Each riff lives in a conceptual "world", where various state values can be stored and retrieved. That state can then be used to control whether other responses will execute or even directly output in subsequent text messages.

In our example, each time "mortality" is called, the responses in the group will be considered as candidates. We might not want that, though – what if this should be a one-time deal, and then forever hold your peace? As this is not an executing response, there is no occurrence count (and, indeed, no way to specify different selects for different runs). How can we keep it from running again?

What we can do is to set a state to indicate that a choice has been made. Then we require that that state not be set in order for our response to take effect.

Here's one way to do it:

```
.response mortality
    .needs not shared
    .selects all
        .response
            .does
                .says One day, the king felt it was time to share his worries
                    with someone close to him.
        .response
            .prompts He sought the advice of the Queen
            .does
                .says He decided to seek the advice of the Queen, who often
                    had insights that he himself did not.
                .sets shared
        .response
            .prompts He went to see his trusted adviser
            .does
                .says He decided to seek the advice of his trusted adviser, whom
                    he considered quite wise.
                .sets shared
        .response
            .prompts He had a conversation with his son
            .does
                .says He decided to have a conversation with his son, the most
                    likely heir to his throne.
                .sets shared
    .end
```

The group response now has "needs". This indicates what the requirements are for a response to be considered eligible for use. In this case, the need is for the "shared" state to not be set, which will be true until one of the option responses is clicked and runs, at which point the "shared" state is set.

Note that a simple "on/off" sort of state can be turned on simply by listing its name in a ".sets". The "shared" state could also be turned back off by something like ".sets not shared".

Relative vs Absolute Variable Access

The variable "shared" used in the examples so far has been accessed using *relative* variable access. It is a local variable for the responder whose response is being processed. The full name in the world state for this variable would be "player:shared", since the responses are for the player responder.

A bare variable name and one preceded by a colon are both relative variables. So, for example, "shared" and ":shared" both refer to the same variable.

It's also possible to access variables by specifying the full name. So, for example, you can use the fully-

qualified name "player:shared" to access the variable from any responder or even a global context (e.g. while setting up the riff outside any responses).

This ability to define relative variables is important when responders share common responses, as for example when they all belong to the same class. This allows the variable to be attached to only the responder in question.

Note that the first component of a full path need not be an actual responder name. For example, you could use a global space for variables named "global" simply by using that text in your variable accesses for things like "global:var1" and "global:var2". There would not be a specific responder named "global", but there would be a partitioned set of global variables. (This is not recommended, by the way. It's far better to associate variables with specific areas of the riff.)

You can also (in theory), use the global component name as a variable in its own right by appending a colon onto the name. So "var:" would define a variable named "var:". It might seem a bit strange conceptually, but it does work. (Any name that has a colon in the name but without a leading colon is an absolute variable name.)

Response Weights

Now that we have a conditional response, once a choice has been made, the "mortality" topic will no longer generate a response. At that point, the "upon his death" link would be converted to normal text, as there are no longer any eligible responses to handle it. That may not be the desired behavior, however. If it's desired to keep the link, there are at least two ways to deal with this.

The first involves response weights. This is probably the simplest way to go, as you just need to add a new, lower-weight response without having to change the existing one. This makes sense, design-wise. The original "mortality" response is handling the case where a choice has not been made. Adding a new, separate response keeps the cases distinct.

The following response can be used to take up the slack once the original response is no longer eligible to run:

```
.response mortality
    .weights 0.5
    .does
        .says The king felt better after sharing his concerns.
```

This is a simple response, but with something new: ".weights 0.5". That assigns a weighting to the response. By default, if no weight is specified, a response is assigned a weight of 1.

When responses are being examined to determine which ones to run, all the current eligible responses are scored based on how well they match the called topics. Only the responses with the best score will be run. A response score will be scaled by its weight. Normally, for responses with a default weight of 1, this will leave the score unchanged. However, a score can be scaled up or down by specifying an explicit weight.

In this case, this new, lower weight response's score will be cut in half. This means that the response will be overshadowed by the original response as long as that response is eligible to run. It will not run, as its score will be lower than the original response. Once the original response ceases to be eligible (once the "shared" state has been set), then this new response will become the response with the highest score, and it will then run instead.

This allows the creation of a hierarchy of responses, with different weights determining which are currently the ones to run.

An alternative to using response weights is to group responses together using "selects first", which we will talk more about next. Whether to use one or the other becomes a design choice: is it more logical to group all responses to the same topic under one master response or to have a set of related responses which are independent but occur in different contexts? There are also considerations about how deeply you wish to nest your responses, as it can quickly become a bit unwieldy.

Selecting the First Eligible Response from a Set of Responses

An alternative to using response weights as described above is to group the responses together under a a "uses first". A "uses first" phrase is similar to a "uses random" phrase, except that instead of selecting a random response to execute, the "uses first" phrase will execute the first response that is eligible to run. This allows you to order responses in a group, based on the which are more important. The response processor will then march through the group looking for the first response it can execute (e.g. one that has all needs met and hasn't exceeded its "occurs" count). Once it find an eligible response, it will execute that response and no others.

This is the ResponsIF equivalent of an "if/then/else" statement, where you can have as many "elses" as necessary.

Using State Values in Output Text – State Markup

Besides tracking "yes/no" types of state values (also known as "flags"), a riff world can store string and numeric values. These values can be compared against in "needs" clauses (more on that later). They can also be output along with your text. This is another way of making text variable.

In our current story, instead of simply recording that the king shared his troubles with someone, let's record whom he shared his troubles with. These responses do that:

```
.response mortality
    .needs not shared
    .selects all
        .response
            .does
                .says One day, the king felt it was time to share his worries
                      with someone close to him.
        .response
            .prompts He sought the advice of the Queen
            .does
                .says He decided to seek the advice of the Queen, who often
                      had insights that he himself did not.
                .sets shared .to the Queen
        .response
            .prompts He went to see his trusted adviser
            .does
                .says He decided to seek the advice of his trusted adviser,
                      whom he considered quite wise.
                .sets shared .to his adviser
            .response
            .prompts He had a conversation with his son
            .does
                .says He decided to have a conversation with his son, the most
                      likely heir to his throne.
                .sets shared .to his son
    .end
```

Now, in the response for "mortality" that triggers after the king has shared his concerns, we can do this:

```
.response mortality
    .weights 0.5
    .does
        .says The king felt better after sharing his concerns with {=shared=}.
```

The syntax "{=shared=}" is our second kind of markup: "state markup". It is used to insert state values into output text. You can think of the "=" as meaning, "insert the value *equal* to this expression". If the adviser is chosen and then "upon his death" is clicked again, we get output like this:

One day, the king felt it was time to share his worries with someone close to him.
He decided to seek the advice of his trusted adviser, whom he considered quite wise.
The king felt better after sharing his concerns with his adviser.

Expressions also work. The markup '{=3+4=}', for example, will output '7'.

Creating a World Model

The examples we have had so far give a reasonable introduction to responses. For the ResponsIF features we wish to introduce next, a different approach will be more illustrative. The examples we will work through will be more in the vein of a traditional text adventure, with rooms and objects to manipulate. These will make up our *world model*. ResponsIF is not limited to such world models; this one was chosen as it's hopefully familiar enough to both the readers (and the author) that the concepts will be communicated clearly.

What is important to keep in mind is that objects, including "rooms", do not exist as such in ResponsIF. What exist are the responses associated with responder names, the relationships among the various responders, and any state set into the riff world. (Note: though the objects don't exist as such, it's not unreasonable to pretend they do, when considering the overall riff world model.)

We have already been introduced to one responder, the default "player" responder. And we have manipulated and responded to state. Next, let's see how to put responders in a hierarchy.

Parent/Child Relationships

In many game designs, not only are the individual responder's responses critical but the relationships among the responders may be as well. ResponsIF allows responders, if desired, to be arranged in a parent/child hierarchy, where a responder may have a single parent and any number of children. This hierarchy will be used by ResponsIF to determine which responders are currently eligible to respond, based on their relationship to the current point-of-view responder.

As an example, consider a den with a table, a chair and a fireplace. In this case, the den would be a "parent" responder, and the other three responders would be its children. If the player is then moved into that room (by making the player the room's child as well), then all five responders – player, den, table, chair and fireplace – would be examined when eligible responses are being gathered to process topics.

Responders may be moved, outside of a response's "does", by using the immediate ".move" command. This sequence of statements will set up the situation described above:

```
.setup
   .moves table .to den
   .moves chair .to den
   .moves fireplace .to den
```

Responders may also be moved by a response:

```
.response MOVE_EAST
   .does
      .says You enter the den.
      .moves player .to den
```

Note that the semantics of making a responder a child of another is to "move" it, as if the responder were physically being moved. That mirrors a standard world-building model where objects exist inside rooms, etc. However, it is only a euphemism for setting a parent/child relationship, and it can be used even where physical objects are not being modeled.

When ResponsIF is processing a topic call, it will make a list of the current available responders and their responses. This defines the current call context. The responses of those responders form the search space for possible processing. Responses for responders not in the current call context will not be examined. This creates a natural partitioning of responses, based on their current relationship to the POV.

ResponsIF always includes the point-of-view responder in the current call context. The parent of the current POV (if it exists) is also added, as well as any children of the POV and any siblings of the POV – children of the POV's parent. Above that, any parent of the parent is included, as well as any parent of *that* parent, up the chain. This allows the defining of "areas". An example would be a room contained within the floor of a building which is itself contained within the building. Or various outside locations, all contained within an overall "outside" parent (which might respond to the "sky" topic, for example). Finally, when constructing the response context, any responder whose parent is "everywhere" is included; children with this parent will be available in all contexts, which can be useful especially for certain "meta-responders" like the overall UI or hint or help systems. This set then forms the current responders to be examined.

Moving a responder from parent to parent can be used to simulate moving it from room to room. It can also be used more generally to move responders in and out of context.

The parent of a responder is stored as an attribute in the world model. For example, after the player has been moved into the den, the world state value "player:parent" would equal the string "den". Similarly, "table:parent", "chair:parent" and "fireplace:parent" would all also be "den".

The parent state variables can be used in responses or output using text markup. As an example, the following response will call the "DESCRIBE" topic if the player's parent changes:

```
.response
   .needs last_parent <> player:parent
   .does
      .sets last_parent=player:parent
      .calls DESCRIBE
```

It would be useful if a response like the above could be called after every turn. It so happens that you can do that by defining an action, as described in the next section.

Autonomous Behavior

So far, all our behaviors have been triggered in response to user input. This makes for a responsive

system, but there are times it's desirable to be able to have more autonomous, or self-directed, behavior. ResponsIF provides for the definition of *actions*. Actions are still responses, but they are processed separately from user calls, occurring after the completion of each turn.

In short: responses are called in direct response to user input. Actions are called at the completion of each turn, regardless of what the user did on that turn.

You define actions the same way that you define responses – actions are also response groups, but you use the "actions" token to begin them instead of "responses". And actions are associated with a responder, just as normal responses are. There are three major differences:

1) Actions are invoked for all responders in the system, regardless of where they are in the responder hierarchy. This makes sense: for example, NPCs should still move about and otherwise behave autonomously even if they're not in the same room as the player.

2) In the case of a normal player-initiated call, the responses for all the current responders are gathered together into one large set, and the highest priority matching responses from that set are used. With actions, each responder is processed separately. This allows the responders to act without being interfered with or overruled by other responders' behaviors.

3) As actions are not invoked in response to called topics, the responses in an action group often won't have topics associated with them. (The primary exception to this is when using long- and short-term topics to steer behavior.)

Actions are useful for any sort of autonomous behavior, whether in the game world or not. They can be used to move NPCs around in the game world, but they can also be used to update the user interface. In ResponsIF, it's all the same thing. It's all part of the game's conversation.

Here is how the "auto describe" response from the previous section would look as an action:

```
.actions auto describe
    .response
        .needs last_parent <> player:parent
        .does
            .sets last_parent=player:parent
            .calls DESCRIBE
.end
```

In this case, we define an "auto describe" responder (the name is arbitrary – note that it can have spaces) which has the job of automatically describing a new location when the player's parent changes. After each game turn, this response will run if the current player parent is not the same as the last known parent. In that case, the new parent will be remembered in the "last_parent" state variable, and the "DESCRIBE" topic will be called. It will not run again until the player moves to a new parent.

This is our first example of a responder that has no counterpart in the actual game world. Depending on the overall game design, this may not be that unusual a thing. It's the full set of responders that provide the game experience, not just the ones visible in rooms. A responder is only "visible" to the extent that the game text makes it so.

Dynamic Text Generation – Call Markup

So far, we have learned about two kinds of markup: *link markup*, which is used to insert links in the text, and *state markup*, which is used to insert state variables in the text. We're now going to explore a third, very powerful form of markup: *call markup*. With state markup, you can insert static values into the text stream. With call markup, you can insert into the text stream the result of calling topics. This

allows you to dynamically construct text, with the full flexibility, control and power afforded by responses.

Let's give an example. To start, let's assume there is an intersection in the game world, with roads leading off in various directions but particularly to the east, where there is a bustling bazaar. The response when the player looks might look like this:

```
.responses intersection
    .response DESCRIBE
        .does
            .says Dirt roads lead off in all directions. To the east is a
                    bustling bazaar. You can see merchants going about their
                    business. The sun burns brightly. Birds circle lazily overhead.
.end
```

That's reasonable, but it only details this particular intersection at one point in time. What if the game world has different times of day, and it's desired to have different text based on that time of day?

One approach would be to have entirely different responses based on the time of day:

```
.responses intersection
    .response DESCRIBE
        .needs morning
        .does .says Dirt roads lead off in all directions. To the east is a bazaar,
                    the merchants preparing their stalls for the coming day. The
                    sun is just peeking over the tree tops. Birds circle lazily
                    overhead.
    .response DESCRIBE
        .needs afternoon
        .does .says Dirt roads lead off in all directions. To the east is a
                    bustling bazaar. You can see merchants going about their
                    business. The sun burns brightly. Birds circle lazily overhead.

    .response DESCRIBE
        .needs evening
        .does .says Dirt roads lead off in all directions. To the east is a bazaar.
                    The merchants are closing up shop at the end of the day. The
                    sun is preparing to rest beyond the hills. Birds circle lazily
                    overhead.
    .==== etc
.end
```

This would work, and in some cases it might be sufficient. However, ResponsIF allows you to dynamically construct text instead. This is done by adding into the text calls out to other responses. Any text from those responses is then inserted into the current text stream.

In our example, let's first separate out the description of the bazaar.

```
.responses intersection
    .response LOOK
        .does .says Dirt roads lead off in all directions. {+bazaar+} The sun
                    burns brightly. Birds circle lazily overhead.

    .response bazaar
        .needs morning
        .does .says To the east is a bazaar, the merchants preparing their stalls
                    for the coming day.

    .response bazaar
        .needs afternoon
```

```
            .does .says To the east is a bustling bazaar. You can see merchants going
                        about their business.

        .response bazaar
            .needs evening
            .does .says To the east is a bazaar. The merchants are closing up shop at
                        the end of the day.
        .==== etc
.end
```

We now have a single response for "DESCRIBE", with multiple responses for the "bazaar" topic based on the time of day. And the "DESCRIBE" response has a "call" out to the bazaar response inline using the notation "{+bazaar+}". The "+" signifies that the text from calling the indicated topic (or topics) will be added into the output text stream.

Let's vary the "sun" text in this description as well.

```
.responses intersection
    .response DESCRIBE
        .does .says Dirt roads lead off in all directions. {+bazaar+} {+sun+}

    .response bazaar .needs morning
        .does .says To the east is a bazaar, the merchants preparing their stalls
                    for the coming day.
    .response bazaar .needs afternoon
        .does .says To the east is a bustling bazaar. You can see merchants going
                    about their business.
    .response bazaar .needs evening
        .does .says To the east is a bazaar. The merchants are closing up shop at
                    the end of the day.

    .response sun .needs morning
        .does .says The sun is just peeking over the tree tops. Birds circle
                    lazily overhead.
    .response sun .needs afternoon
        .does .says The sun burns brightly. Birds circle lazily overhead.
    .response sun .needs evening
        .does .says The sun is preparing to rest beyond the hills. Birds circle
                    lazily overhead.
.end
```

We can extract the common intro text. It's not necessary for call markup to reference variable content.

```
    .response DESCRIBE
        .does .says {+intro+} {+bazaar+} {+sun+}
    .response intro
        .does .says Dirt roads lead off in all directions.

    (the rest remains the same)
```

At this point, the original DESCRIBE response's text is looking more like a template than the original text. It's not necessary to fully vary your text, but it is possible to do so when desired.

Next, let's extract and vary the "color" text.

```
    .response sun .needs morning
        .does .says The sun is just peeking over the tree tops. {+color+}
    .response sun .needs afternoon
        .does .says The sun burns brightly. {+color+}
```

```
.response sun .needs evening
    .does .says The sun is preparing to rest for the day. {+color+}

.response color
    .does
        .uses random
            .response .does .says Birds circle lazily overhead.
            .response .does .says A rabbit scampers across a nearby field.
            .response .does .says Flies buzz around your head.
        .end
```

As shown in the example above, call markup can end up invoking further call markup, nesting to whatever arbitrary level is desired.

Finally, let's add a random story event into our looking about. That is, let's have something significant happen during the handling of a "look". To that end, we'll add a new response to the "color" response.

```
.response color
    .does
        .uses random
            .response .does .says Birds circle lazily overhead.
            .response .does .says A rabbit scampers across a nearby field.
            .response .does .says Flies buzz around your head.
            .response
                .needs menacing stranger:parent <> "intersection"
                .does
                    .says A menacing stranger approaches from the north.
                    .moves menacing stranger .to intersection
        .end
```

In this case, a story event will occur randomly – a menacing stranger will approach you. (What happens next depends on what the responses are for that menacing stranger.) Call markup can invoke the full power of responses, including being able to modify world state. This can be used for dynamic text composition, but also much more.

Check out the "random sentences" sample for an expanded example of using call markup.

Integrating the User Interface

The User Interface is Part of the Conversation

Many times, the core textual content in IF is treated as something separate from the rest of what's happening on the screen. The user interface is seen as something to dress up the content or provide access to features, and its implementation is often separate from how the main content is generated. Almost inconsequential.

ResponsIF makes the user interface part of the same overall conversation as the story text. It can be generated and manipulated with responses just as the main story text is. This allows a riff to have complete dynamic control over the user experience.

Note that this section uses some HTML-specific concepts, in particular how elements in an HTML document are referenced. Hopefully, the usage will be clear from the context, even for those with little or no HTML experience.

Specifying Specific Elements within Content

ResponsIF is HTML-based. To that end, it uses standard HTML and CSS conventions. The most important is that of the *CSS selector*. A CSS selector can be used to specify one or more pieces of HTML content based on such attributes as class or ID. For example, if you have an element like this:

```
<div id="myelement" class="myclass">Here is my text.</div>
```

then you can reference this element using (among others) "#myelement" or ".myclass". Typically, there should only be a single HTML element per ID. However, multiple elements can share the same class. (Note that at this time using classes is tricky due to the leading "." on the class selector being parsed as a ResponsIF token. The solution for this is pending. However, there is no problem using ids.)

HTML content in ResponsIF can be baked into the main index.html file. It can also be generated dynamically using responses.

Directing Content to Elements

Normally, when a response "says" text, the text is appended to the default output element. This is the standard behavior for "says". Text can also be directed into a specific HTML element. The following riff is an example of this:

```
.responses player
    .response START
        .does .says Harry can {!count!}. He says, "<span id="counter">one</span>."

    .response count
        .occurs 3
        .does 1 .says one, two .into #counter
        .does 2 .says one, two, three .into #counter
        .does 3
            .says one, two, three, four .into #counter
            .says But that's as high as he can go.
.end
```

The new part here is the use of ".into" to indicate which HTML element the text should be said into. This will direct the content into the element matching the CSS selector following .into. In this example, it replace the text "one" in the span first with "one, two", then with "one, two, three", and then finally with "one, two, three, four", also printing out some final text to say we're done counting.

The sequence looks like this, with each subsequent output occurring when "count" is clicked:

There are a few noteworthy things here. First, text that is said "into" an element replaces the text in that element, without adding any new content in the normal way (that is, appended at the bottom of existing content). Second, other content can still be added in the regular way (without using .into), even while

content is been directed to a specific element. Third, the "occurs" for the response means that the "count" link becomes deactivated after three clicks, as it is no longer meant to be valid.

(This example highlights another possible, as yet unimplemented in ResponsIF, text strategy – onto. In this example, text actually adds onto what currently exists, but it can only be accomplished by rewriting the content from the beginning each time. Using the token .onto instead of .into would append content onto the element's existing content rather than replacing it. This idea is in the "considered" stage at this time.)

Using Actions to Automatically Update UI Elements

The previous example showed updating content in-stream, but it's possible to update any content on a page using responses. One use of this is to have out-of-stream user interface "panes" to display status information or hold buttons for common functionality. By combining .says .into with actions, the user interface will be updated automatically after each player turn.

In this example, assume there is an element with ID "status", arranged as a pane beside the main content. It might look something like this:

The main output content is on the left, and the right is a status pane. The responses for this can use player-based attributes, which are displayed in the pane.

```
.setup player
    .sets race .to Elf
    .sets health=7
    .sets max health=10
    .sets strength=8
    .sets gold=0

.responses player
    .response START
        .does
            .says Welcome to the underground.
            .says {+STATUS+} .into #status

    .response STATUS
        .does
            .says Race: {=:race=}<br>
                  Health: {=:health=}/{=:max health=}<br>
                  Strength: {=:strength=}<br>
                  Gold: {=:gold=}
.end
```

These responses will give the output shown in the image above. Note that the "says" in the STATUS response is incorporated into the "says into" of the calling response. So even though the STATUS "says" doesn't have an "into", it will still honor the "into" of the calling response, as its text will become part of that output stream.

There is one problem with this: if the status attributes change, the status pane will not update to reflect

the new values. The only time currently that that content is output is in the START response. That won't do. A solution to this problem is to create an action, one which will run after each game cycle. We can do that with the following changes.

First, we'll create an action to run for the user interface.

```
.actions ui
    .response
        .does .says {+STATUS+} .into #status
.end
```

Then we can remove the status "says" from the player's START response. It's not necessary to make the actions belong to "ui" (or whatever name you wish to give the responder that manages the user interface). It could just as easily be an action on the player. However, this pattern of the user interface responder calling topics that are then filled in by other responders in the current context allows a centralization of the user interface code, and it also allows for more flexibility in the way the user interface is populated, as the UI will update depending on which responders are currently in context. As an example, if a riff had multiple point-of-view characters which were switched among, each could implement the STATUS topic in the way that makes sense for that character. The call from the UI remains the same; it's the response that is different depending on current responders.

Animations

ResponsIF provides a mechanism for animating page elements. This can be used to fade elements in or out, to change font size, color, location on screen, etc. Many CSS styles can be animated. (The support is actually dependent on what is supported by jQueryUI.)

A sample "fade-in" animation might look like this:

```
.animates #titletext .to {"opacity": 0.0} .to {"opacity": 1.0} .lasting 1000
```

Let's break this down.

The ".animates" token introduces an animation. The element to animate must come next, specified as a CSS selector (as it did with the ".into" phrase). Next comes one or more animation expressions which consist of the token ".to" followed by the target attributes. In this case, the first animation expression is to set the opacity of the element to 0, which effectively hides it. The default duration for an animation expression is immediate, so the element will start out hidden. Then the next animation expression changes the opacity to 1, but lasting for a duration of 1000 milliseconds, which is the same as one second. You would therefore see the element fade in over the period of one second. (This effect is used in the sample "The Haunting".)

The animation expression is notated as a JSON object, with one or more target attribute values. It is a convenient and easily read and parsed way to specify data.

More than one attribute can be varied at the same time. For example, if we have the following animation:

```
.animates #myelement
    .to {"font-size": "0%", "opacity": "0"}
    .to {"font-size": "100%"} .lasting 1000
    .to {"opacity": "1"} .lasting 1000
```

then the element will start out both invisible and with no height. The first animation will grow the element. This has the effect of "making room" for the element on screen. Then the element fades in. The total duration for this animation would be two seconds, as each animation takes place in turn,

sequentially. The attributes being varied could be combined into one expression if it was desired to vary them both simultaneously.

```
.animates #myelement
    .to {"font-size": "0%", "opacity": "0"}
    .to {"font-size": "100%", "opacity": "1"} .lasting 1000
```

For further information on what's possible with animations, consult the jQueryUI documentation. (That is a fairly lame cop-out for the moment, but it's all we have until more in-depth documentation is included here.)

Click Effect

There is a dedicated animation built into ResponsIF known as the "click effect". This is the animation to perform on a link when it is clicked. It can be specified in the riff as part of the game setup. Here is a click effect that makes the link blink briefly when clicked:

```
.click-effect
    .to {"color": "#c0c000"} .lasting 250
    .to {"color": "#f0f020"} .lasting 300
    .to {"color": "#c0c0c0"} .lasting 200
```

Note that this uses the same animation expressions as an "animates" token. The "click-effect" animation must be specified outside any responses, and it is immediate. At this time, there is no way to change the click effect via responses while a riff is in play.

Organizing Your Riffs

As riffs grow in size, it becomes important to be able to organize them in a reasonable way. This section outlines some techniques for both dividing up riff's into manageable chunks as well as providing names for common sequences.

Including files in other files

As your riffs get larger, there may come a time that you wish to break them up into more manageable pieces, to keep them organized and easier to deal with. ResponsIF allows you to do this using the ".include" phrase. Here's an example:

```
.include room.txt
```

You can only include files at the top level of a riff, that is, outside of things like action or response groups. However, the .include can occur anywhere within the source file, as long as it is at that top level. The file inclusion is lexical – the token/value pairs from the included file are inserted into the source file at the point of inclusion, before any parsing takes place.

Included files can .include other files.

Macros: Defining your own tokens for token sequences

ResponsIF allows you to give names to sequences of tokens. This can be useful to create a single definition for a commonly used sequence, to create a short-hand name for a complex sequence, or even just to name things more in line with your own tastes.

Consider this response:

```
.responses UI
    .response widget
        .does
            .animates #bar .to { "opacity": 1 } .lasting 2000
.end
```

The "animates" part of this may seem a bit cryptic. At the very least, if it's used in multiple places, you might want to have an easier to understand name and a single place to change how it is defined. You can do this:

```
.define fades-in-bar .animates #bar .to { "opacity": 1 } .lasting 2000 .enddef

.responses UI
    .response widget
        .does
            .fades-in-bar
.end
```

This snippet creates a new macro named "fades-in-bar", which can be invoked where needed simply by using it as a token. The macro definition needs to be at the top level of a file (that is, outside of any response or action groups), but it can be used anywhere once it has been defined.

The value for the ".define" token becomes the new name. All token/value pairs become part of the definition up to the ".enddef" token. When the macro is used, it is replaced by the tokens within it.

There is some limited ability to parametrize a macro. When a macro is invoked, it can have a value, as any token/value pair can. That value can be used inside the macro when it's being expanded. The caveat is that the called value must be used as the value for some internal token(s) in the macro. It can not replace or insert tokens in the stream.

In our example above, let's say that this "fades in" animation is common, and it would be useful to apply it to different elements. It can be written like this instead:

```
.define fades-in .animates .<value> .to { "opacity": 1 } .lasting 2000 .enddef

.responses UI
    .response widget
        .does
            .fades-in #bar
.end
```

The ".<value>" token marks where the value for the called macro should be inserted. It is set as (or appended to) the value for the preceding token during expansion. In the example above, ".<value>" will cause "#bar" to become the value for ".animates".

Macros can be nested as desired. Here we define both a general "fades-in" macro as well as a "fades-in-bar" macro that uses it.

```
.define fades-in .animates .<value> .to { "opacity": 1 } .lasting 2000 .enddef

.define fades-in-bar .fades-in #bar .enddef

.responses UI
    .response widget
        .does
            .fades-in-bar
.end
```

Here are some further examples of macros and their uses (as always, line breaks are as desired):

Macro

```
.define increments .sets .<value> = .<value> + 1 .enddef
```

Example usage

```
.increments somevariable
```

Macro

```
.define says-string
    .response
        .does
            .says .<value>
.enddef
```

Example usage

```
.response weather
    .does
        .uses random
            .says-string raining
            .says-string sunny
            .says-string snowing
            .says-string windy
        .end
```

Note that ".<value>" is a token and so must have space around it. In particular, this means that in the "increments" macro above, it must have space around it on both sides (on the left so the "." will be parsed as the beginning of a token, and on the right so that it's name will be clear), even though you might not normally put spaces around the variable in an expression.

Macros are processed after the main file and all included files have been loaded, and after tokenization but before parsing. The substitution is strictly at the token level. For example, there is no way to construct token names with macros.

Sharing responses

Sometimes it is useful to be able to share responses among a number of different responders. There is a way in ResponsIF to define responses that multiple responders can use. The only caveat is that they are the exact same responses, and so things like the "occurs" count will reflect usage by all the responders combined.

When defining top-level responses for a responder, in addition to defining normal responders, it's possible to insert a reference to another responder's responses. These responses will then be used as if they were part of the original responder's response set. Note that relative variable names will bind to the original responder, not the responder that holds the shared responses. This allows each responder to have its own set of variables, which are operated on by the common responses. Example:

```
.setup apple
    .sets name .to apple

.setup banana
    .sets name .to banana
```

```
.responses fruit
    .response DESCRIBE
        .does .says This is a nice, ripe {=name=}.
.end

.responses apple
    .reference fruit
.end

.responses banana
    .reference fruit
.end
```

Fuzzy Logic

So far, the examples have used very basic forms of logic - "sets" or "sets not" to turn flags on and off coupled with "needs" to test whether a flag is set. Also, standard composite expressions like "happy and healthy" or "open or broken" work, following standard logic rules.

ResponsIF supports standard logic in this way. However, it also implements its own extension to what is known as "fuzzy logic", where instead of just two logic values ("true" and "false"), values can range from 0 to 1, with 0 corresponding to the traditional "false" and 1 to the traditional "true". ResponsIF extends this to allow negative values as well. This extension allows modeling of not only "happy" and "not happy" but also "unhappy".

The following diagram shows how these three systems map.
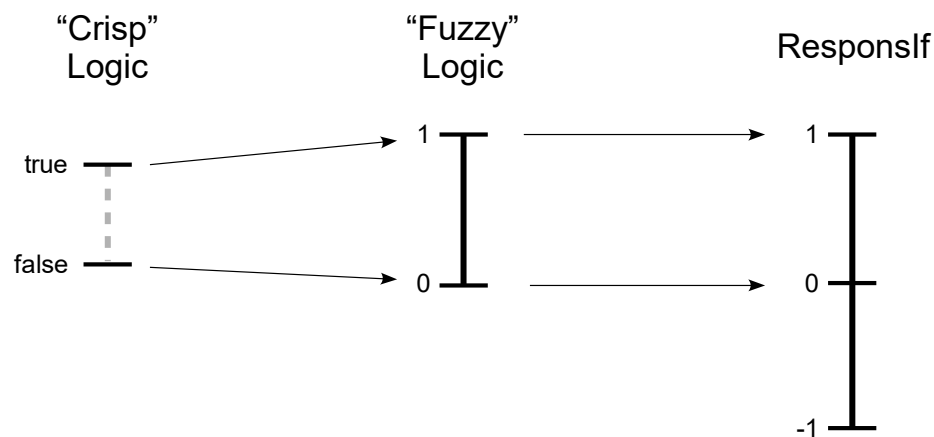
Using "happy" as an example, when setting values, a phrase like ".sets happy" will set the variable "happy" to the value 1. Similarly, the phrase ".sets not happy" will set the variable "happy" to 0. This mirrors traditional logic. There is also a new concept: "un". The phrase ".sets un happy" will set the variable "happy" to -1.
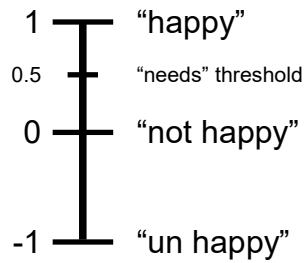
The threshold for "needs" is 0.5. If the value is 0.5 or greater, then "needs" will be satisfied. Any value below 0.5 will not satisfy needs.

The standard boolean operators ("and", "or" and "not") apply to this form of logic, though with some differences. Similarly there is the addition of "un", "more" and "less", as well as more traditional fuzzy logic operators like "difference" and "equals". These are described in detail next.

"not"

Let's begin with the simplest operator: "not". Traditionally, "not" is used to invert a logic value - "not true" is false, and "not false" is true. Fuzzy logic extends this to map 0 to 1 and 1 to 0 continuously, by setting the new value to (1-old value) . ResponsIF takes the same approach, but any values less than 0 are considered 0 when the "not" is applied. This means that any value 0 or less is mapped to 1, which makes sense since 0 or less implies a certainty for it "not" being true. The following diagram shows this mapping:
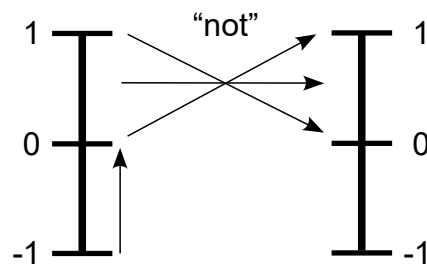
"un"

The next operator is similar to "not": "un". The "un" operator completely inverts the logic value. That is, 1 goes to -1, and -1 goes to 1. A value of 0 remains unchanged (the "un" of nothing is still nothing). Mathematically, "un" sets the new value to (0-old value). The following diagram shows this mapping:
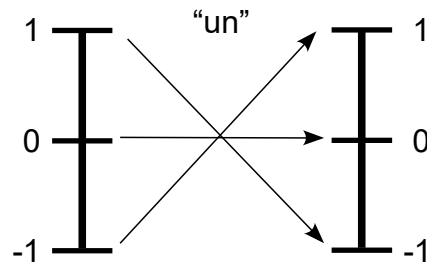

"or"

The previous two operators were both "unary" operators – that is, they operate on a single value. The "or" operator is known as a "binary" operator, as it operates on two values. In traditional logic, the result of an "or" operation is true if either operand is true. Otherwise, if both operands are false, then the result will be false.

In fuzzy logic, values can be more than just true or false. The fuzzy equivalent of "or" is to take the maximum of the two values. This turns out to correspond to the normal logic case (the max of 0 or 1 is 1). It also makes sense from a fuzzy point of view. If you require either of two values, your result is as confident as the strongest confidence you have from either one. So, in an "or" operation, the strongest value dominates.

This is true even if the values are negative. The result of an "or" operation will always be the maximum of the two values.

The following diagram illustrates "or":
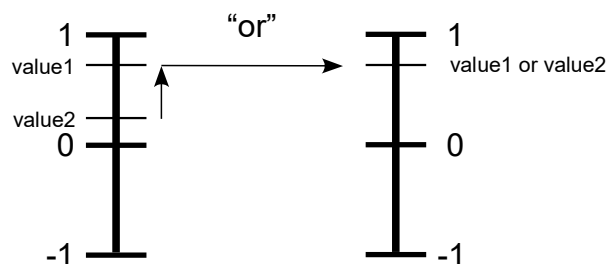
"and"

The "and" operator is used in those cases where you require two conditions to be true (or more, as they can be chained). In traditional logic, the result of an "and" is true if both of the operands are true. Otherwise, if either operand is false, then the result is false.

The fuzzy logic equivalent of this is to take the smaller of the two values. This happens to correspond to the normal logic case (the minimum of 0 and 1 is 0), and it makes sense from a fuzzy point of view – your confidence in two things being true is only as strong as the one you have the least confidence in (e.g. if you have no confidence in A, then you have equally no confidence in A and B).

This is true even if the values are negative. The result of an "and" operation will always be the minimum of the two values.

The following diagram illustrates "and":
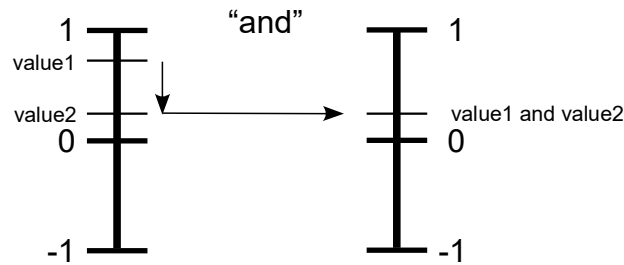
"xor"

The "xor" operator is included for completeness. However, it might not be commonly used in the fuzzy way. The basic meaning of xor ("exclusive or") is "A or B but not both A and B". In traditional true/false logic, "A xor B" is equivalent to "A <> B". In fuzzy logic, the meaning is less clear.

The approach commonly taken in fuzzy logic is to compute "(A or B) and not (A and B)" using the fuzzy operations. This corresponds to normal logic in the true/false (0/1) cases, and has suitably fuzzy results when not. Due to the possible existence of negative values in ResponsIF, there is the additional stipulation in ResponsIF that negative values are computed as if they were 0.

There is no meaningful diagram for xor that I could work out.

Another approach to take for xor is as a "not equals". The concept of "equals" is broken out as a different operator.

"equals"

The "equals" operator is quite straightforward: it returns a value corresponding to how equal two numbers are. That might sound strange – numbers are either equal or not. However, in fuzzy logic, we can deal with truth values that are not simply yes or no, true or false, equal or not equal. The basic behavior for "equals" is that it should return 1 for two values that are exactly equal and 0 for two values that differ by 1. And in between, it should vary accordingly. To that end, the value for "A equals B" is computed as (1 – abs(A-B)). This means that values differing by greater than 1 can end up being "un equal" to a corresponding degree.

"difference"

The "difference" operator returns the absolute value of the difference between its two operands, up to a value of 1 (so "1 difference -1" will still be just 1).

"more" and "less"

So far, we have seen ways to set values for variables (".sets", ".sets not" and ".sets un") as well as ways to test and combine values. However, there has been no easy way, short of explicitly setting a value to something like 0.25, to set values to anything in between the extremes.

In line with the readable operators like "not", "un", "and" and "or", ResponsIF defines two operators used to adjust values along the ranges from min to max and back: "more" and "less". An example of using "more" would be ".sets more happy" to increase the amount of happiness a character feels.

Similarly, you could use ".sets less healthy" to indicate that a character has lost vitality.

There are some important things to note about "more" and "less".

First, the stepping is not linear. They are designed such that the further the value is from the target ("more" steps toward 1, and "less" steps toward -1), the larger the step. This means, in particular, that "more less happy" is not going to be equal to the original "happy" value. It also means that "more happy" will step you further if you are "un happy" than if you are "not happy" or only marginally "happy".

Second, "more" and "less" take smaller and smaller steps as they approach their target values, and they will never exceed their target values, no matter how many times you step. In fact, as the value approaches the target, at some proximity (a difference of less than 0.001), it will "snap" to the target. So eventually, they will reach their target values exactly.

The following diagram gives a rough approximation to how "more" and "less" work.

more 1 0 less -1
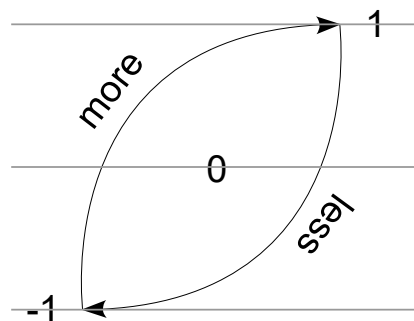
"More" and "less" can be used both as standalone changers (e.g. ".sets more happy", ".sets less healthy") or as components of an expression (e.g. ".sets content = more happy"). In the latter case, the source variable ("happy" here) is not changed.

".adjusts"

"More" and "less" are specialized forms of adjusting values along a continuum toward a target value. They are meant to be convenient and commonly used. However, there are cases where it might be useful to adjust variables toward different targets. This ability has been implemented as a general purpose ".adjusts" response phrase.

The motivation behind this is that certain responses may trigger a certain "emotion" or "mood" in a character. However, such states are not static. Happiness (or any other emotion) rarely lasts with the intensity it may have at a particular moment. Emotions tend to dissipate over time. It seemed logical that a riff author would want to be able simulate that sort of effect. However, "less" has a target of -1. It can't be used to approach 0.

Similarly, a "towards 0" sort of operator would work in some cases, but emotions might not tend toward 0 in all cases. There may be other emotions or states that combine to form a sort of non-zero "baseline" for a variable to tend towards. ".adjusts" allows a riff author to specify such a baseline, not only as an absolute value but also as the result of a fuzzy expression. The baseline can even change over time.

An .adjusts in full form looks like this:

```
.adjusts variable .toward target .stepping step
```

where *variable* is the variable to be adjusted, *target* is the target value to adjust toward, and *step* is a stepping factor, which must evaluate to a value greater than 0 and not greater than 1. The only optional phrase is ".stepping step", which will default to 1 if not provided.

When placed in an action, ".adjusts" can be used to automatically decay variables. Here is an example:

```
.actions Timmy
    .response
        .does .adjusts happy .toward 0
.end
```

This will automatically reduce "happy" to 0 over time, unless something else sets it to a different value. It's possible to use more complex expressions. For example, you could have:

```
.actions Billy
    .response
        .does .adjusts angry .toward scared or worried
.end
```

In this case, the "angry" variable will tend toward the current values for "scared" or "worried". Increases in those variables won't cause an immediate spike in anger, but they will cause a gradual shift in the angry state over time, either increasing or decreasing.

Now that we have the ability to specify and manipulate moods, emotions and other variable concepts, the question becomes how we can use them in our responses. The answer is found in the next section, which is all about response weights.

Moods and Modes to Influence Behavior

Using Response Weights to Influence Response Selection

The previous section focused on ResponsIF's use of "fuzzy logic", an approach to modeling state that is more variable and continuous and (hopefully) expressive than the true/false approach of more traditional logic. But how can these values be leveraged?

We have already seen the "yes/no" approach to response selection by using "needs". The ".needs" phrase allows a riff author to apply a rigid "yes or no" decision to whether a response should be considered eligible to execute or not. This is very useful and can work well for many situations.

ResponsIF also supports a fuzzy selection of responses by adjusting the score of a response based on specified "weights". During a call, a score is generated for each eligible response based on how many of its topic keywords match the incoming called topics. The more topics that match, the higher the score. An *additional weight* will then be applied to that score, if one has been specified for the response. This is where mechanics like "emotion" can make their presence felt.

Here is an example, for when the player attempts to pet a dog:

```
.responses Dog
    .response pet
        .weights happy and not (threatened or scared)
        .does .says The dog wags its tail and licks your hand.

    .response pet
        .weights scared
        .does .says The dog dodges your outstretched hand, moving to a
                    safer distance.
```

```
      .response pet
          .weights threatened
          .does .says The dog growls menacingly and bares its teeth.
.end
```

All three of these responses will have the same score based on the incoming topic: "pet." However, their overall score will also have the specified weights applied. Assuming the states have reasonable values, the response with the most appropriate match to the current emotional state of the dog will be chosen. In fact, if both "scared" and "threatened" have equal values, then you would get both responses firing. "The dog dodges your outstretched hand, moving to a safer distance. The dog growls menacingly and bares its teeth."

Response scores and weights are not limited to any particular range of values. The only limiting behavior of response scores is that a response must have a score greater than 0 to be considered eligible to run. A response with a resultant score of 0 or less will be removed from consideration while processing a call. Beyond that, scoring is all relative. Normal scores are based on the number of matching topics (adjusted by the topic weights).

This means that you can artificially boost response scores using weights. For example, if you have a particularly volatile character, it might be desired to weight the responses accordingly:

```
.responses Perkins
    .response greet
        .weights angry*5
        .does
            .says He glares at you and stalks past.

    .response greet
        .does
            .says He nods grudgingly before continuing on his way.
.end
```

In this case, Perkins will response angrily even if his "angry" state is as low as 0.2. And if Perkins were full-on angry, it would possibly dominate all other responses, if weighted accordingly throughout. In fact, you could actually set a "volatility" variable and use that in all the relevant responses:

```
      .response greet
          .weights angry*volatility
          .does
              .says He glares at you and stalks past.
```

These are just some simple (and undoubtedly simplistic) examples of how ResponsIF's fuzzy variables can be used to simulate and dictate emotional response in riff characters.

Response Weights Need Not Be Emotions

Response weights can be used to model emotions. However, they can also be used more generally to model any sort of varying influence. Consider the case where it's desired that the user interface for a riff vary depending on a game-wide "mood", which might be determined by the actions the player has taken so far. Dark and sinister actions on the player's part might begin to taint the overall riff, where the riff world becomes bleak and foreboding, perhaps even extending to the color scheme used by the user interface. Or perhaps the user interface styling changes based on the riff-world time of year, with subtle differences in theme based on the current season.

Another example: an elevator simulation may be implemented such that "up" and "down" are possible

moods for the elevator. If it's in an "up" mood, it will tend to continue upward, until it no longer has floors to service above. Then it will become neutral, at which point a downward request will send it into a "down" mood.

ResponsIF features have been designed initially with specific uses in mind. However, it is hoped that their generality of implementation means they can be used for a wider range of purposes than the initial concept may have encompassed. Let your imagination and creativity come to the fore and see what's possible!

Integration with JavaScript

Despite all that has been described so far, there may be times when the core response abilities lack the functionality needed for your riff's design. A full scripting language was not created for ResponsIF as HTML platforms already come with one built in: JavaScript. The ".invokes" phrase is the way to connect your responses to the full power of JavaScript.

The following example "says" the current browser version and agent:

```
.responses player
    .response START
        .does
            .invokes
                var agent = navigator.userAgent;
                var version = navigator.appVersion;
                interact.say({ text: "Your browser version is " + version + "."},
                            responder);
                interact.say({ text: "The agent is " + agent + "."}, responder);

.end
```

The first thing to notice is that the value in an "invokes" phrase is actual JavaScript code. Any legal JavaScript may be used, though any constructs involved a leading "." will confuse the ResponsIf parser.

The code in an "invokes" is run as a JavaScript function. This function is passed three parameters, which are available to the script:

– responder: This is a string containing the name of the responder that supplied the response being run.

– world: This is an object encapsulating the world model data. It has an interface which can be used to get and set state values, get and set responder parents, manage model topics (clusters), etc.

– interact: This is an object encapsulating ResponsIF's core functionality: saying text, calling topics, animating HTML, showing menus, sending a command while executing a full game cycle, etc.

The object functional interfaces will eventually be documented. At this time, the best documentation available for them is the "living" documentation embodied in the unit tests, which not only describe the APIs but also provide actual running sample code. The world test specification can be found in test/rif_world_spec.js, and the interact test specification can be found in test/rif_interact_spec.js.

Conversation and Beyond - Advanced Response Design

So far, this document has covered the basics of ResponsIF. With this section, we will now begin to get into more advanced topics. In particular, we'll begin exploring response models and how they can be used to track topical state and influence subsequent behavior in a more contextual way.

(Note that from here on out we're in the more experimental part of ResponsIF. As more riff content is created, the technology will undoubtedly evolve and mature. For now, enjoy life on the bleeding edge!)

Before we dive into response models, let's look briefly at how ResponsIF selects responses.

How Responses Are Selected

As discussed earlier, when a topic call is made, the responses for all the current responders (determined based on the responders in the current POV responder's context) are pooled together, and then the highest scoring eligible responses are selected for processing. (Action processing is similar, except each responder with actions is processed separately, using their "actions" set.) This indicates two key concepts: response eligibility and response scoring.

A response is considered eligible to be processed if

1) its "occurs" count has not been reached,

2) its "needs" are satisfied,

3) its score is not zero, and

4) any required topics are present. (This last eligibility check is a bit of a hack at the moment, but it's listed for completeness.)

A response can have specified "needs". This is an expression that is evaluated based on the current world state. For a numeric need, the result must be "fuzzy true", that is greater than or equal to 0.5. For non-numeric needs, the state must simply exist. A response whose needs are not met will not be considered eligible to be processed, regardless of score. This allows a hard, "crisp logic" selection of responses based on more or less traditional on/off state.

Responses also have scores calculated based on the topics matched. So far, the example responses we have seen have only had single topics to match. However, a response can match any number of topics, and to varying degrees. These factor into the overall score calculation.

Consider the following responses:

```
.response door
    .does .says You see left and right doors.
.response left door
    .does .says The left door has the image of a lion over it.
.response right door
    .does .says The right door has the image of a tiger above it.
```

There are three responses. The first response has topics "door", the second response has topics "left" and "door", and the third response has topics "right" and "door". If a call is made for "door", then all three responses will match equally – each has the topic "door", with the same weight. In that case, the output would be, "You see left and right doors. The left door has the image of a lion over it. The right door has the image of a tiger above it."

If a call is made, instead, for the two topics "right" and "door", then the first two responses will both match with a score of 1 – each matches a single topic ("door"). However, the third response will have a

score of 2, since it matches both called topics. In that case, only the third response would be processed, and the output would be "The right door has the image of a tiger above it." This mechanism allows the engine to select responses based on which match the called topics the best.

To allow further refinement, responses topics can be given weights. These weights affect the final score. Let's adjust the above responses as follows:

```
.response door
    .does .says You see left and right doors.
.response left door=0.5
    .does .says The left door has the image of a lion over it.
.response right door=0.5
    .does .says The right door has the image of a tiger above it.
```

Here, we have weighted the topic "door" less for the more specific responses. Now, if the topic "door" is called, the first response will have the score 1, and the second and third will have the score 0.5. This means that only the first response will be selected for processing. Similarly. if the topics called were "right" and "door", the first response would have a score of 1, the second a score of 0.5, and the third a score of 1.5. The matched topics are weighted based on the weights given and then summed. This becomes the final score... almost. There is one more way to influence the score: a response "weights".

A response can have a "weights" phrase. This specifies an expression to be used to scale the response's oevrall score. A phrase like ".weights 0.5" will cut a response's score in half. (This is useful for lower priority responses.) A phrase like ".weights 2" would double the score. And the value need not be constant. A phrase such as ".weights happiness" would scale the response's weight based on the current value of the responder's "happiness." If no "weights" is specified, it is assumed to have a value of 1.

A response can have an arbitrary number of topics and topic weights. It would not be unreasonable to have a response like this:

```
.response family farm=0.2 father=0.8 mother=0.8 home=0.5
```

This allows a single response to respond to a number of different topics (or combination thereof), each with a different emphasis. To some extent, topics and weights assign a crude form of meaning to responses.

One potential problem with the above responses is that the more specific response will possibly match in contexts that are not intended. For example, calling "left window" would match the "left door" response, since the topic "left" creates a positive. This can be done using the aforementioned hack of "required topics" (which may change in a future revision of the engine). Preceding a topic with an asterisk ("*") will mark that topic as being required. If the topic is not present in the call, then the response will fail the eligibility test, before a score is even computed. The following revised topics illustrate this:

```
.response door
    .does .says You see left and right doors.
.response left *door=0.5
    .does .says The left door has the image of a lion over it.
.response right *door=0.5
    .does .says The right door has the image of a tiger above it.
```

Now the second and third responses have the topic "door" marked as required. If that topic is not present in the call, then the response will not match. (So, for example, calling "left" will not match the "left door" case, since "door" is required but not present.)

The examples given above include cases where multiple topics are called. How might that happen?

We have seen one way already: link markup can wrap multiple words. In that case, each word in the link comes across as a separate topic. So the text "He looked at the {!right door!}." would have a link that calls the topics "right" and "door".

Similarly, you can specify multiple topics for a link outside of the text. The text "He noticed {!one of the doors|left door!} had the image of a lion above it." would create a link calling the two topics "left" and "door".

The full power of multiple topics comes into play, however, when we begin to build up responder internal state using responder models. We will discuss that next.

The Power of Topics

Our focus until now has been on responses: what they are, what they do, and how they are selected. We have encountered topics largely as triggers from the player for selecting those responses. But topics are vitally important in riffs. In fact, the rough conversational concept that led to the original response library focused on topics as a means for both *encapsulating context* (knowing what we're talking about) and for *associating responses* (connecting different ideas together to allow conversation to evolve). Those ideas remain at the heart of ResponsIF.

Responses are unit of behavior. Topics are units of meaning.

Responses define what a riff does. Topics define what a riff is about.

In the simple examples we've seen, topics have been ephemeral, existing only from the time the player clicks on a link until the response occurs. In order to model more advanced interactions, we need to be able to maintain a topical context. That is, we want our responders to be able to take in and remember topics that they have encountered. To that end, each responder in ResponsIF has its own "topic memory", a place where topics (and their associated weights) can be either stored by the responder itself or suggested by other responders in the same response context. These topics are then included with any called topics. This means that the responder's topical context can be used to influence how a responder responds to new called topics by providing additional information about what has happened in the past.

Topic Clusters

In the original incarnation of the response library, there were two hard-coded topic stores: long-term (persistent) and short-term (transient). Long-term topics persisted unchanged from turn to turn. Short-term topics decayed over time, eventually fading from memory. These concepts have been generalized and extended in ResponsIF as "topic clusters", named groups of topics with author-definable characteristics.

As an example, a responder might have the following model definition:

```
.model Eddie
    .cluster persistent .weight 0.8
    .cluster transient .decaying 0.4 .suggestible
.end
```

The model defines two clusters, one named "persistent", and one named "transient". The topics in the persistent cluster will have their weights scaled by 0.9 before being added into the current topics being called. The weights of topics in the "transient" cluster will decay over time, similar to a human being's

short term memory. The "suggestible" phrase marks the "transient" cluster as being the one to receive suggestions from other responders.

The overall design of this model is that there are both long-term and short-term topics being maintained. The long-term topics can correspond to anything desired: goals, memories, motivations. The short-term topics come in as suggestions from other responders and decay over time, providing a current topical context. The reducing weight on the long-term topics means that they will have slightly less influence over response selection than called topics or the more immediate short-term topics.

Topics from all clusters are combined together along with any called topics when responses are being selected. If topics exist in more than one place (e.g. in more than one cluster or in clusters along with the current called topics), then their weights are combined to increase the weight of the overall topic during response selection.

This is a very basic model. There can be additional clusters or fewer. Perhaps there will not be any decaying topics. Perhaps the particular responder doesn't care about (and so shouldn't track) suggested topics. The design of a responder's model is entirely up to the riff author, and each responder can have its own unique cluster arrangement, as desired.

Note: If a model is not defined for a particular responder, then topics can still be manually added and removed from clusters by responses. However, the clusters will have generic properties: no weight scaling, no automatic decay, and not suggestible. If no cluster is specified when adding or removing topics, the cluster named "persistent" is used by default.

Adding and Removing Topics

The following response adds a topic to the current responder:

```
.response
    .does .adds murder
```

As no cluster is specified, the topic "murder" will be added to the "persistent" cluster for the responder, which will cause it to be included with other topics in subsequent calls.

Similarly, the following response will remove the topic "sunshine" from the current responder:

```
.response
    .does .removes sunshine
```

As no cluster is specified, the topic will be removed from the "persistent" cluster, if the topic exists. (If it does not exist in the cluster, nothing happens.)

Multiple topics can be added at once, and topic weights can be assigned at the same time:

```
.response
    .does .adds murder=0.5 tension suspicion=0.9
```

Multiple topics can be removed at once as well:

```
.response
    .does .removes sunshine rain
```

Topics can also be added and removed from specific clusters by specifying the cluster:

```
.response
    .does .adds school .to goals
    .does .removes chess .from hobbies
```

In the preceding example, "goals" and "hobbies" are both clusters. They may be specified in a model or not. (Clusters not specified in a model have default attributes.)

It is also possible to manipulate topics for other responders. Whether or not this makes sense (e.g. whether one responder changing the internal "mental" state of another responder is sensible) depends on design. To add or remove topics from another responder, add a "for" phrase to the add or remove:

```
.response
    .does .adds school .to goals .for jenny
    .does .removes chess .from hobbies .for richard
```

Adding topics to other responders during conversation is handled in a different way. This is accomplished by responses "suggesting" topics to all nearby responders. The next section covers this.

Communicating Topics Among Responders

When one person speaks to another, information passes from the speaker to listener, at least in a normal situation. Responses in ResponseIF can output text in which characters can be seen to speak, but how does actual informational content of what was said enter other responders' states? In other words, given that topics embody what is being discussed, how do responders exchange topics?

The mechanism for this is for responses to "suggest" topics that describe themselves. The topics will typically correspond to text that the response has output (dialogue or not), but that's not required. Any responders with a cluster marked "suggestible" will receive those topics. Responders without a "suggestible" cluster will not receive the topics. That means that you don't have to worry about things like rooms or inanimate objects accumulating topics they will never use. (...though it could be interesting to ponder what it would mean from a design point of view if they did. With ResponsIF, you can give an apple, say, a cluster model and allow it to "listen" to suggested topics. What you do with them is up to you and your creativity.)

To suggest topics, use the "suggests" action:

```
.response
    .does .suggests rain
```

The above response will suggest the topic "rain" to any responders in the current response context, and the topic will be added to any "suggestible" clusters for those responder.

As with other topic specifications, you can suggest multiple topics, and you can specify topic weights:

```
.response cloudy
    .does .suggests raining=0.8 sunny=0.2 foggy=0.1
```