Storytron: Introduction

Last edited by Bill Maya 1 day, 3 hours ago

Welcome to the exciting frontier of storyworld authoring! Here is where you will build the components of a world that will interact with the storyplayer as s/he seeks to attain a goal and reach the story's ending. There may be thousands of possible paths to that ending, and the satisfaction of reaching it will depend on the richness you build into your storyworld.

Perhaps you already have ideas for what will happen in your storyworld. You may have some settings planned. Some characters. A protagonist who will represent the storyplayer. Some important objects, like a murder weapon or a magical sword. And a few key events.

These tutorials will help you translate these story elements into an interactive storyworld. The tool that you will use to do this is SWAT (StoryWorld Authoring Tool).

Let's define some storyworld elements:

Element	Examples
Stage	Joe's Bar
Prop	whiskey bottle, chair
Actor	Tom, Mary, Fred
Verb	punch, cry
Event	Tom punch Fred

Notice that the elements in the left column are both capitalized and colored. Throughout this tutorial, a capitalized word has a precise meaning in the context of Storytronics. For example, an actor is a person who peforms in plays or movies, but an Actor is an element of Storytronics that has a specific meaning. The colors applied to certain words and phrases will help you keep their qualities and functions straight later on.

The last element in this list, Event, is a special kind of Sentence. Events drive what happens in a storyworld. Notice that the example Event combines two Actors and a Verb.

"Event" and the rest of the elements listed in the left column are all classes of words in Sappho, the simplified language through which you'll tell the computer how to operate your storyworld. (See Chris's comments About Sappho.)

Sappho provides the grammer, you create the words. SWAT helps you put it all together.

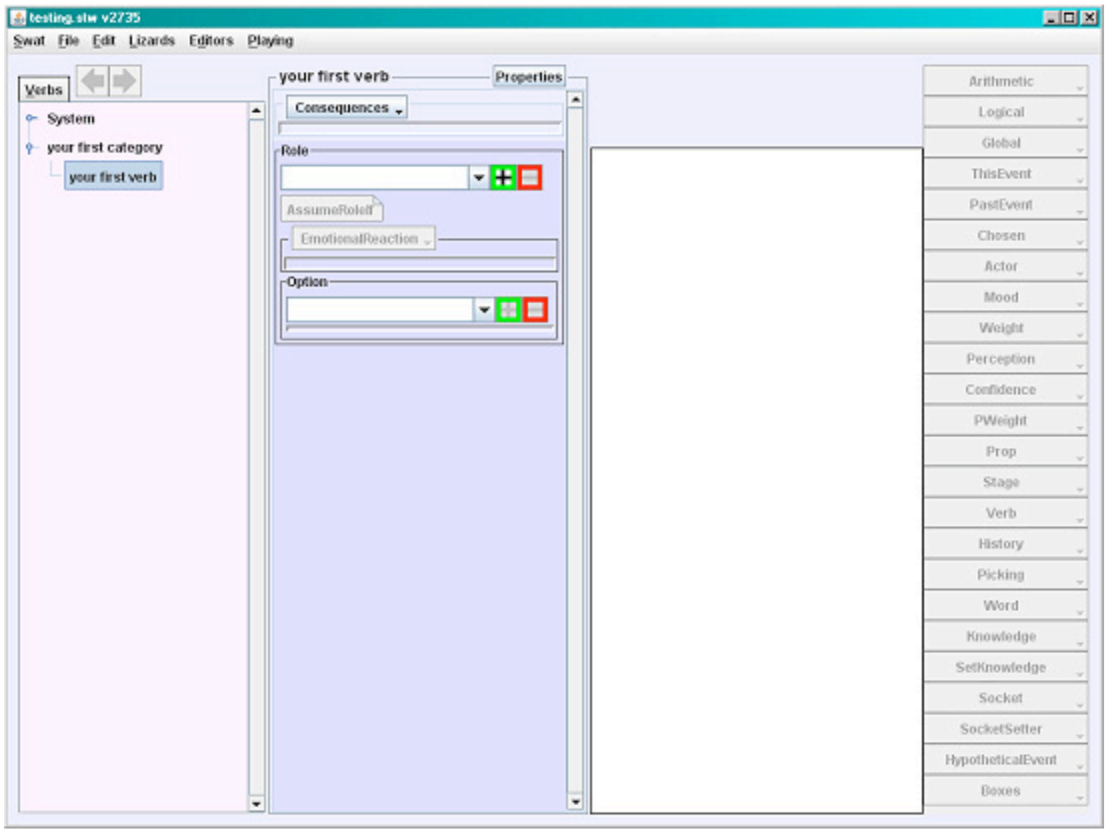
Next Tutorial: The Editors

Storytron: The Editors

Last edited by Bill Maya 1 day, 3 hours ago

The Verb Editor

SWAT starts you off in the Verb Editor, a tool you'll be using a lot. You'll see a screen with four columns.



The leftmost column (pink) shows a list of existing Verbs. The first category is System Verbs. These are used by the system and are basic to every storyworld. Only certain, limited kinds of changes are allowed for System verbs, so don't touch them for now. You'll be adding your own Verbs shortly.

The next column to the right (blue) shows the currently selected Verb and some components of the Verb that can be edited:

- The Properties button brings up the Verb's basic settings, including the assigned Emoticon (a graphic showing the Actor's facial expression), the Audience (which other Actors witness the verb), WordSockets which define which storyworld elements participate in or are affected by the Verb, and a few other details.
- The Consequences button is used to define the effects of theVerb on the

storyworld once it's used in a Sentence.

- The rest of this column is the Role Editor, where the author defines how various

Actors will respond to the Verb, i.e., what roles they will assume. A very complex Verb can include a lot of potential Roles for various Actors.

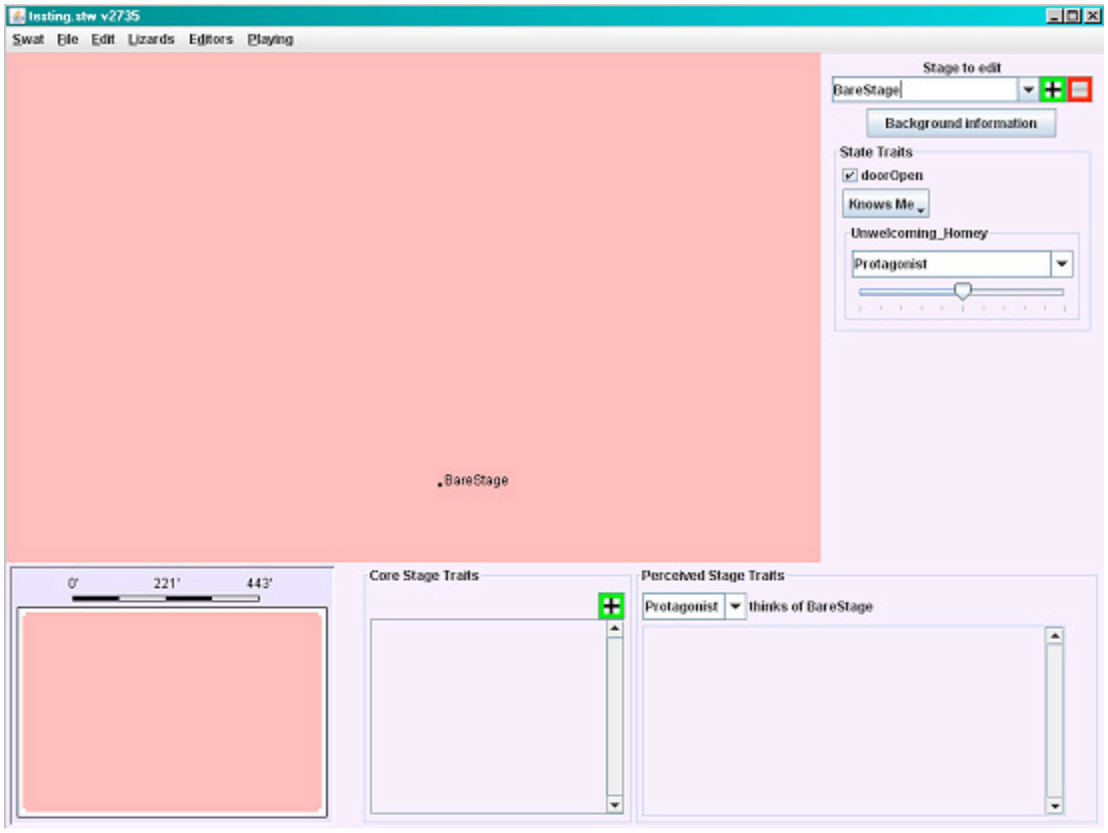
Notice the green "+" and red "-" buttons in the Role Editor.

⊞ This button always means "Add a new one of these things"
⊟ This button always means "Delete this"

The third (white) column is the Script Editor. This is where details of the Verb are defined using the scripting language Sappho.

The fourth column contains links to Operators. These are used in building scripts. The links will become available once you start editing scripts.
Verbs are the most complicated elements used in building a storyworld. We're not going to create any Verbs now; we'll come back to the Verb Editor in a bit.

The Stage Editor
On the menu bar, chose "Editors" and skip down to the Stage Editor to define a Stage (a setting) for your storyworld. The editor window looks like this:



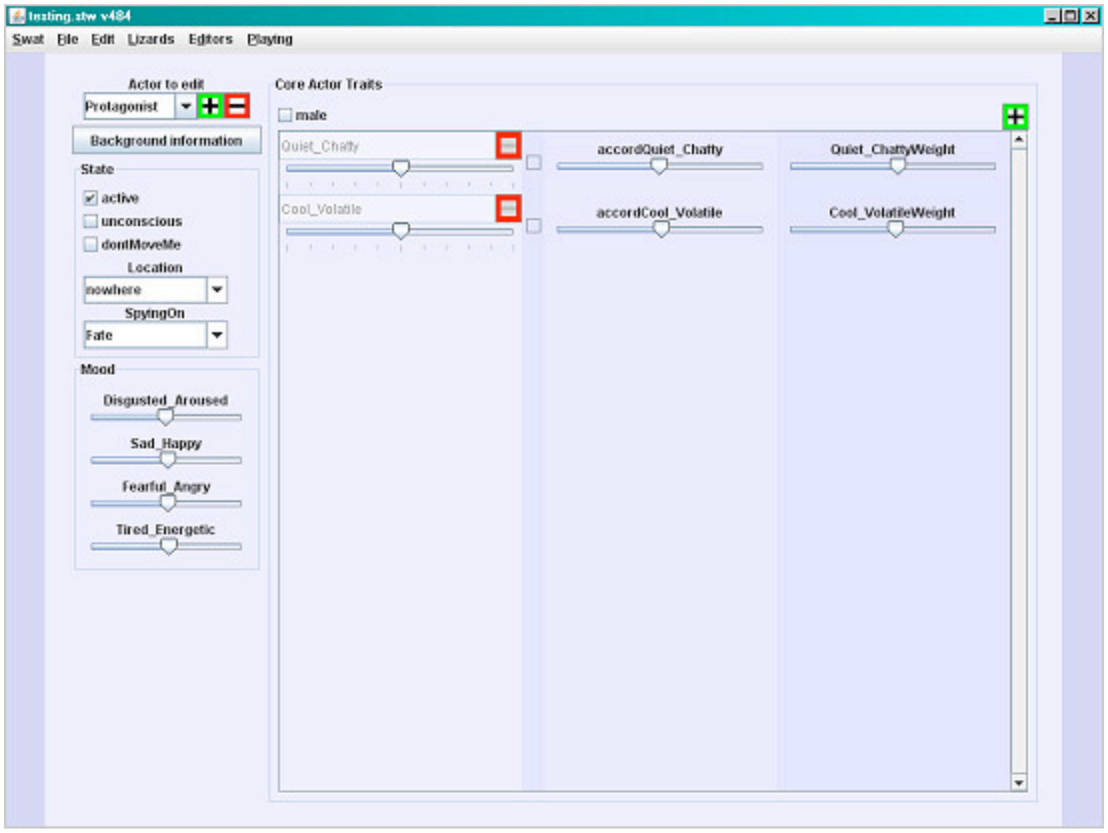
The big salmon-colored area is a map of your storyworld. There's one predefined Stage there already: BareStage. At the right, beside "Stage to Edit," click the green "+" to add a new Stage, and change this Stage's name to "Joe's Bar."

Note that Joe's Bar appears on the storyworld map. You can click the bullet by its name and drag it to wherever you want on the map.
Click "Background Information." Here you can describe the Stage, and include an image if you wish. Close the Background Information window and glance at the "State Traits" box, and the "Core Stage Traits" and "Perceived Stage Traits" at the bottom of the page. These are where you define what the Stage is like, and how it's perceived by the Actors.

But we don't have any Actors yet! Let's move on.

The Actor Editor

Select the Actor Editor from the Editors pulldown in the menu bar. You'll see this:



The default Actor is the Protagonist, the hero of your storyworld, who is driven by the player. You can change the Protagonist's name if you want; s/he will still be the Protagonist of the storyworld.

Again, there's a "Background Information" button that you can use to describe the current Actor and assign an image. Below the Background Information button are the State and Mood sections. These are some basic Attributes that apply to any Actor. They're pretty self-explanatory.

To the right of this column, the main part of the window is where "Core Actor Traits" are defined. A couple of basic traits already exist, along with a box marked "male." If that boxed is unchecked, the Actor is female. (More detail about Actor traits may be found in the Storyworld Author's Guide under Actors.)

The predefined traits are also considered Attributes of the Actor:

Quiet_Chatty	(how loquacious the Actor is)
Cool_Volatile	(how much of a temper the Actor has)

Note that each trait name consists of two extremes, such as "quiet" and "chatty." This bipolar naming structure helps you assign and quickly evaluate traits for various actors. Use a similar name structure (such as Short_Tall, Kind_Cruel, Timid_Brave, Ugly_Attractive) when you create new traits.

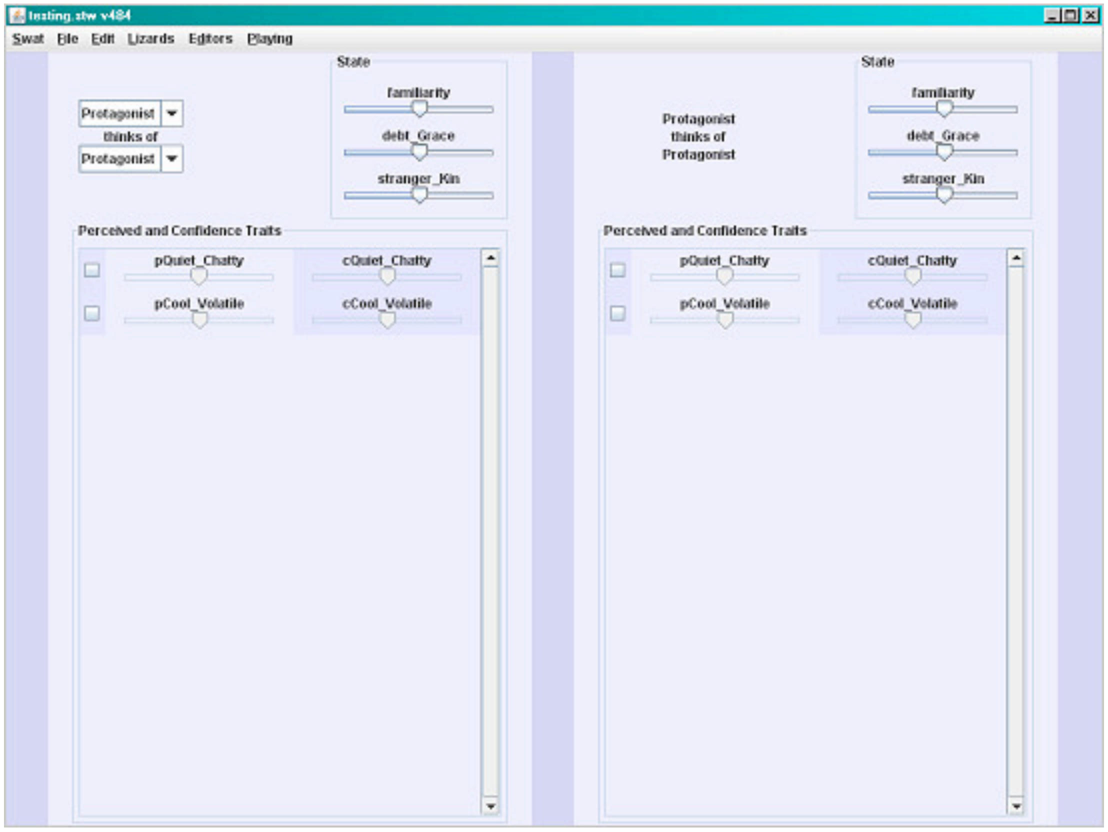
Let's create a couple of Actors, Tom and Fred.

Select "Protagonist" and enter "Tom" instead. Tom is now the Protagonist. Uncheck the "female" box under "Core Actor Traits" to make him male. Change his location to Joe's Bar.

Click the green "+" button under "Actor to Edit." Type "Fred" over "new actor." Make him male, and put him in Joe's Bar too.

The Relationship Editor

From the menu bar, choose the Relationship Editor. This editor allows you to define how well Actors know each other and what they think of each other.



Right now it shows "Tom" in both of the Actor boxes at the top left. Using the pulldown arrows to the right of the windows, change the bottom box to "Fred." Now you can adjust how Tom perceives Fred's traits, and the confidence he has in his opinion. (Details about perception and confidence traits are in the Storyworld Author's Guide, under Relationships.) Notice that the right half of the screen now shows how Fred perceives Tom's traits.

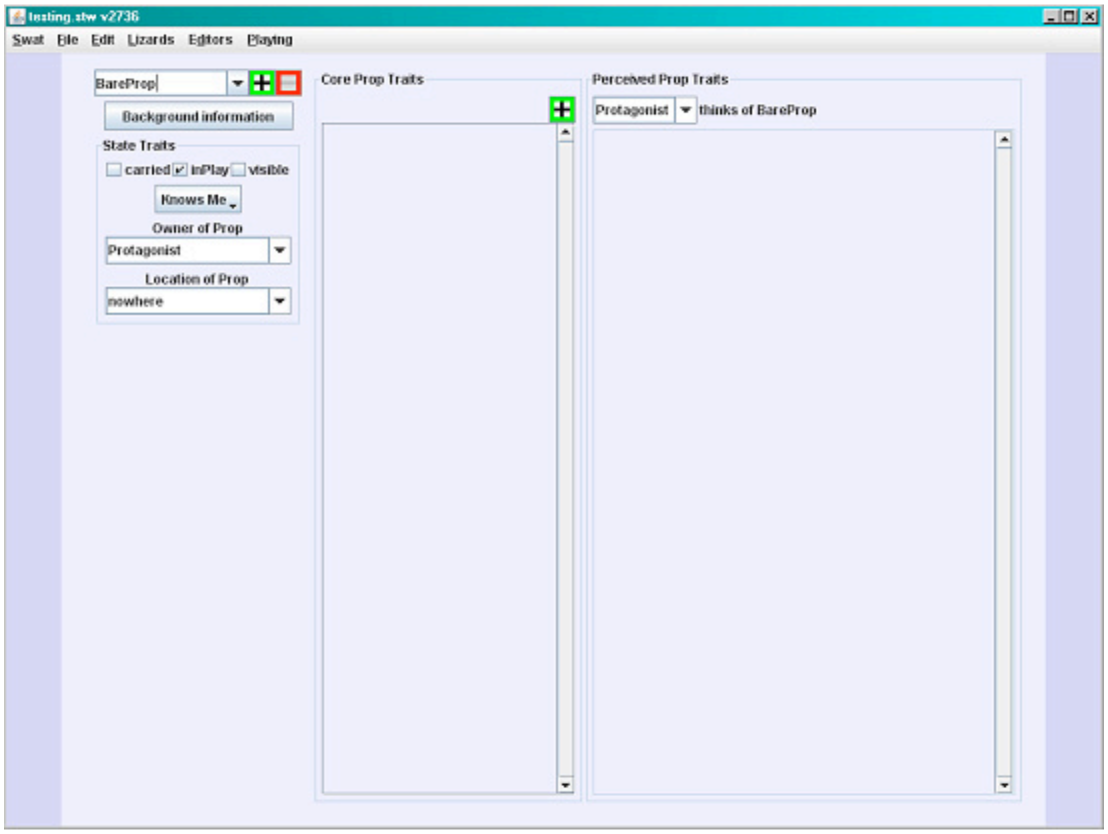
There's also a "State" box on each side of the screen with three options:

Familiarity, Debt_Grace, and Stranger_Kin. These represent the Actors' opinions, so they might not be the same in both directions. For example, Tom might feel that he's familiar with Fred, but Fred might not think he's familiar with Tom. Fred might feel he's in debt to Tom, while Tom might feel it's he who owes Fred a favor.

You can adjust any of the sliders to change the relationship between these two Actors.

The Prop Editor

From the menu bar, open the Prop Editor. This looks a lot like the Actor Editor:



Change the name of the default Prop, "BareProp," to "whiskey bottle," and set its location to Joe's Bar. Note that the owner is set as Tom, the Protagonist. Add a second Prop, "chair," also in Joe's Bar. For this Prop, the default owner is Fate.

You can define additional core traits and perceived traits for the Props. We won't bother with these at the moment.

The Copyrights Editor

This editor allows you to define the copyright status of your storyworld, as well as to provide copyright information and attributions for any outside artwork and text you decide to use. Don't worry about this Editor for now.

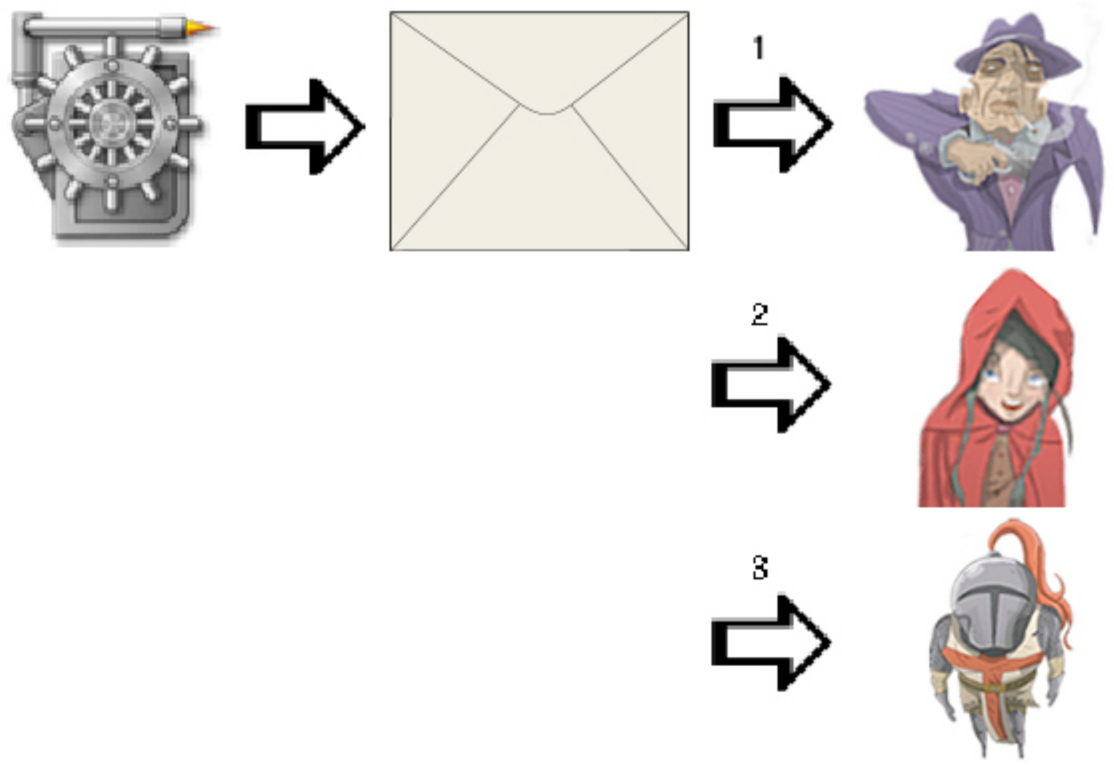
Now that you've seen all the editors and created some storyworld elements, you're ready to start developing story action. Save your storyworld at this point. We'll come back to it in a bit, but first, to help you understand how the Storyteller Engine will execute your instructions, take a look at the Engine Cycle Overview.

Next Tutorial: Engine Cycle Overview
Previous Tutorial: Introduction

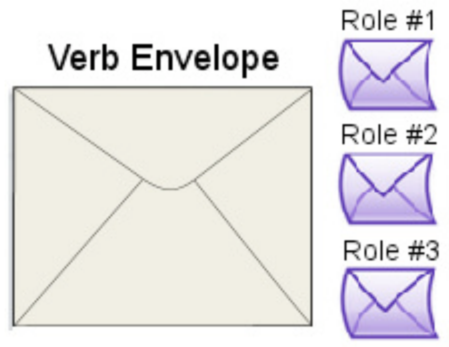
Storytron: Engine Cycle Overview

Last edited by Bill Maya 1 day, 3 hours ago

A story is a sequence of events. In Storytronics, an Event always includes a Verb. When the story engine encounters an Event, it hands an imaginary envelope (the Verb) to each of the Actors in turn:

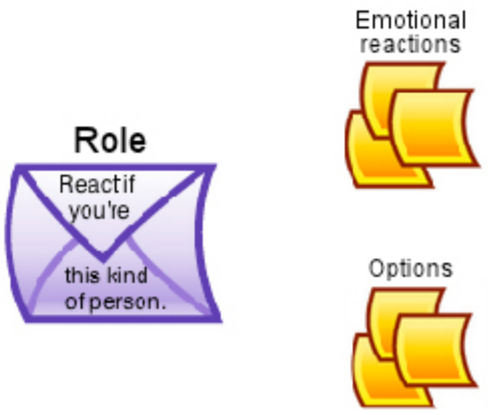


The envelope contains a group of Roles for who can respond and how they respond:



Each Role specifies:

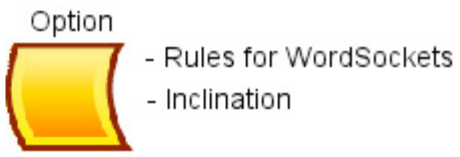
- the conditions under which an Actor may assume that Role
- the emotional reactions of an Actor taking that Role
- a group of Options for that Actor for reacting to the Event



Each Actor considers that Role and, if that Actor meets the conditions defined on the

Role envelope, then that Actor executes the Role, opening up its envelope to see the Options inside.

Each Option specifies:



- which WordSockets will be used construct the sentence for that Option
- the rules for what words will be chosen to fill those WordSockets
- the Inclination of the Actor towards executing that Option

In turn, each witnessing Actor considers the list of Roles and assumes one of them if appropriate. An Actor assuming a Role becomes the "ReactingActor" for the subsequent Role calculations.

Within that Role, a ReactingActor chooses the Option for which s/he has the highest Inclination. That chosen Option becomes a Plan.

Plans are later executed by the Engine, becoming Events.

The cycle begins again.

Many complications, but this is the basic idea.

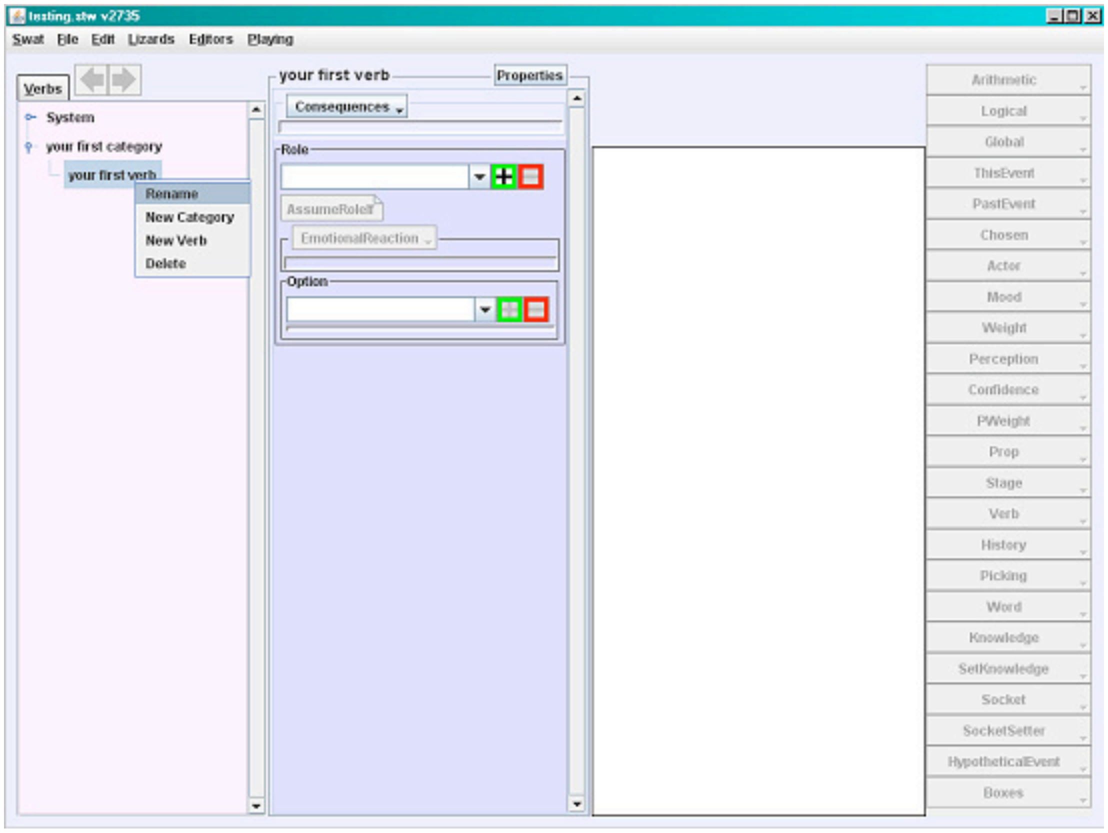
As you can see, one Event can be pretty complicated in terms of all the decisions to be made by the story engine. Fortunately for the player, most of that is invisible. For you, the author, your job is to define the possible Roles, reactions, and Options for the Actors that will be triggered by the Verb. The Verb Editor is the tool that will help you build all of this.

Next Tutorial: Verb Editing
Previous Tutorial: The Editors

Storytron: Verb Editing

Last edited by Bill Maya 1 day, 3 hours ago

Now let's use the Verb Editor to create some new verbs. When you open the editor, here's what you see:



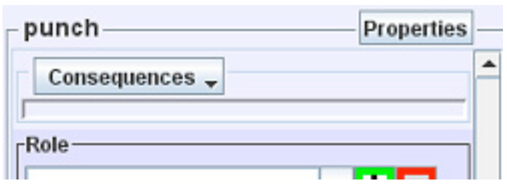
Right click (option-click on Mac) on "your first verb" and change its name to "punch." Next, right-click on "your first category" and choose "New Verb." Call this second Verb "run away from."

How do we define the details of these Verbs? Like a sentence in normal language, they can contain various parts of speech. To organize these parts of speech in a storywold, we use WordSockets.

WordSockets

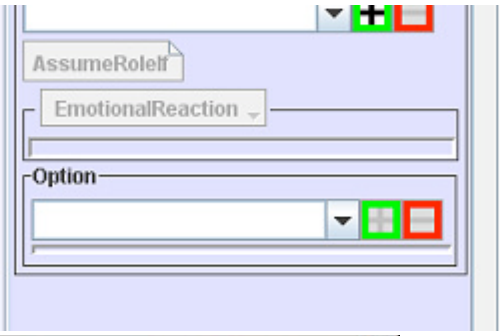
Like every normal sentence, every Event includes at least two WordSockets: Subject and Verb. You can add other WordSockets as you feel appropriate for each Verb you create. Every Verb has its own set of WordSockets.
(See WordSockets for a detailed description of how they work.)

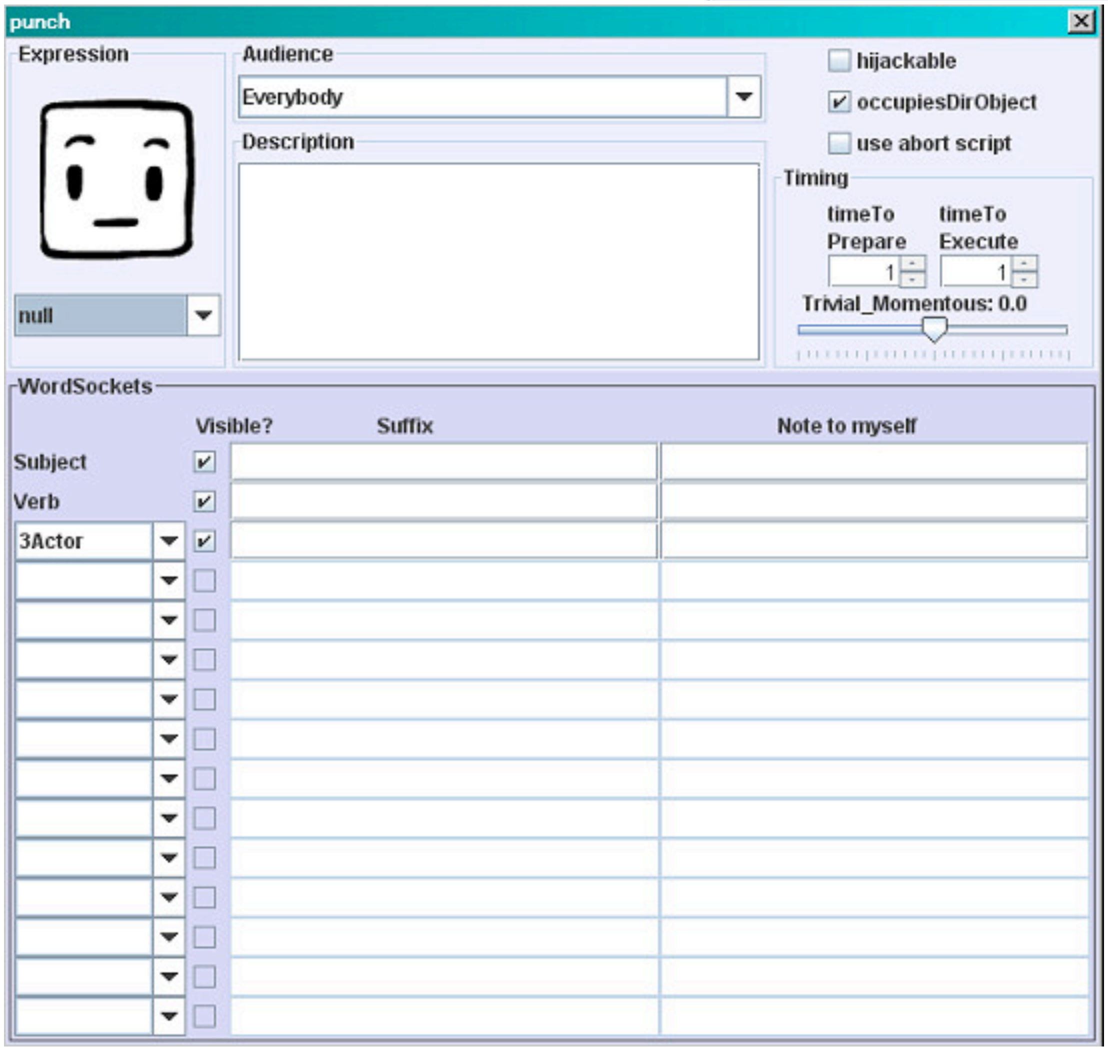
Let's set up our two Verbs, starting with punch. Select that Verb by double-clicking on it in the pink column on the left. The Verb's label now appears at the top of the second



(blue) column.

Click on the "Properties" button at the top of the second column. You'll see punch's properties in a popup:




punch

Expression: null

Audience: Everybody

Description

[ ] hijackable
[✓] occupiesDirObject
[ ] use abort script

Timing

timeToPrepare: 1
timeToExecute: 1
Trivial_Momentous: 0.0

WordSockets

	Visible?	Suffix	Note to myself
Subject	✓		
Verb	✓		
3Actor	✓		

Right now the emoticube is "null." Choose a more appropriate emoticube from the drop down menu, perhaps "angry" or "threatening" or "fighting."

Note that you've already got the two default WordSockets, Subject and Verb. Since the puncher has to have someone to punch, we need a direct object, or "DirObject" for short. This will always be an Actor. Notice that "3Actor" has been filled in on the left below "Verb." We'll assign the Role of DirObject to 3Actor later on.

If you click on the next drop down box, you'll be offered the choice of "4Actor," "4Prop," "4Stage," etc. (Don't choose anything now.) The number represents the position of the item in the WordSockets list, which determines its position in the sentence the player sees. The word represents its type. This naming method will help you later on when you're scripting the subtle details of the Verb.

Now choose the other Verb, run away from, and edit its properties. You can choose an emoticube if you like, perhaps "fearful." Note that "3Actor" is not automatically added to this Verb's WordSockets. Go ahead and add it, since any sentence using this Verb will take the form "Subjectrun away fromDirObject."

Why don't we just have DirObject as a part of the Verb, instead of creating a WordSocket for an Actor? Because just like in normal language, not every Verb has a direct object. The "Properties" box is designed to give you the maximum possible flexibility in creating Verbs.

Roles

We looked at the Role Editor earlier, and now we're going to use it. A Verb can have none, one, or many Roles, although a Verb with no Roles is rather like a bicycle without wheels. Each Role specifies how an Actormight react to an Event. Different Actors will react to the same Event differently; you create one Role for each of the different Actor-situations.

Creating a Role

Let's create a Role for the Verb punch. Close the Properties box if it's still open. Select punch by double-clicking on its listing in the pink column, then find the word "Role" in the blue column. Just underneath that word is an empty white box; to its right are a green "+" box and a red "-" box. Click on the green "+" box and you'll see "new role" in the white box. Type "punchee" as the name of the Role that you just created. It represents the Actor who gets punched.

Assuming a Role

Below the white box is a button labeled "AssumeRoleIf." This is used to define the conditions under which an Actor would assume that Role. You don't want everybody playing that Role—only the person who gets punched—so you have to specify exactly what constitutes the conditions for assuming the Role.
In this case, it's very simple to define the appropriate condition: the Role should be filled by the Actor who was punched, who is, after all, the DirObject of the Event. So you want to specify that the Role should be filled by the Actor who is the DirObject of this Event.
The way you specify this is, however, a little bass-ackwards. You don't actually tell the Engine, "The Role should be filled by the DirObject." Instead, the Engine looks at each and every Actor in turn and asks, "Should I pick this Actor to fill the Role?" and your specification answers with a simple "yes" or "no." We'll teach you

exactly how that's done in a bit.

Emotional Reaction

Below "AssumeRoleIf" is "Emotional Reaction." We're going to skip this part of the Role for now.

Options

Here's where we decide what the Actor playing the Role is going to do in response to the Event.

Obviously, the punchee has a limited set of options when he's punched. He's not going to "whistle Dixie" or "play ping-pong". His set of options should be confined to what is dramatically reasonable. You, the author of the storyworld, must tell the Engine what those dramatically reasonable options are. You do this inside the box labeled "Options" in the blue column.

You'll notice that there's already an Option in that box: "OK." That's a default Option that we automatically add to every new Role when you create the Role; it basically means the actor's OK with what happened. It's not really appropriate for "punchee"—getting punched is not OK—so go ahead and delete it. (To the right of the white box showing "OK," click on the red "-" box.)

Now it's time to add an Option. An obvious choice is "punch"—after all, most men are strong believers in reciprocity, so if one guy punches another, it's likely that the punchee will respond by punching back. So let's add the Verb "punch" to the Options list. To do that, find "punch" in the pink column on the left side of the window, and click ONCE on it. Now simply click in the green box with the plus sign next to the Options box.
Voila! The Verb "punch" has now been added as an Option of the Role "punchee."

Now add the other Verb ("run away from") to the Option list.

When you added a Verb, note that a DirObjectWordSocket box appeared underneath the Option. This is for defining the DirObject of the Verb that's a reaction to the original EventVerb. We'll leave this for later.

Beneath the DirObject box, there's a button labeled "Inclination." Again, we won't fill it in yet, but it's important to understand what this does. The Inclination of any Option is a number that tells the Engine how strongly inclined the the ReactingActor is to choose this Option (e.g., how likely an Actor is to punch or run away from another Actor). The Engine will evaluate the Inclinations of all the different Options, and select the Option with the highest Inclination value.
Inclinations are defined by writing scripts. We'll get into the basics of scripting next.

Next Tutorial: Scripting Basics
Previous Tutorial: Engine Cycle Overview

Storytron: Scripting Basics

Last edited by Bill Maya 1 day, 3 hours ago

A Script is a tiny program that tells the Engine how to make things happen in your storyworld. Every Script starts with an Operator. Some Operators take no arguments, others take one or two arguments, and some even have three, four, five, six, or more arguments.

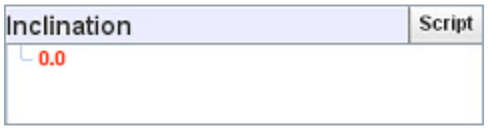
Furthermore, Operators can take other Operators for arguments. So a Script is really just a bunch of Operators nested inside each other, just like a set of nested Russian Matroshka dolls.

Sappho is the Storytronics scripting language, and it includes about 1,000 Operators for you to choose from. It has many special features to make scripting easier for the author. (See About Sappho.)

A typical Script might read something like this: "Store (this answer) into the value for the Inclination." Of course, "this answer" could be a complicated group of Operators, but the basic structure is always the same: one Operator does one thing, but it can have several Operators inside it that it uses to do that one thing. And of course those subsidiary Operators can have subsidiary operators of their own, and on and on. For a detailed discussion of how fast an Operator can get complicated, see Operators.

Right now we're just going to make a very simple script for each of our Verbs. We'll start with an Inclination Script.

Select the Verb "punch" and it will appear with its single Role "punchee." The first Option for "punchee" is "punch" (if your screen shows "run away from," use the Options menu to choose "punch." This Option, like every other, has an Inclination button. Click once on that Inclination button, and you'll see the default Inclination Script appear in the third (white) column to the right of the blue column. It's not much of a Script:

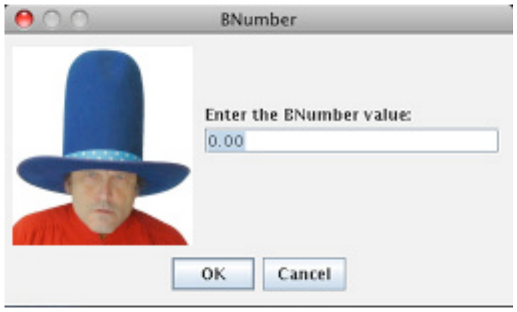


The title "Inclination" tells you that this is the Inclination Script. Ignore the little "Script" button on the right edge for now.

The default Inclination Script has a value of 0.0. Click on that number. Some new buttons appear above it, and some of the buttons along the right edge of the window activate. Those buttons along the right edge are actually menus. Click on one of those active menu buttons and a menu will pop up. Don't select anything from the menu just yet; to dismiss the menu, just click anywhere else.

Above the word "Inclination" is an octagonal button labeled "BNumberConstant."

Select that item and a dialog box appears:

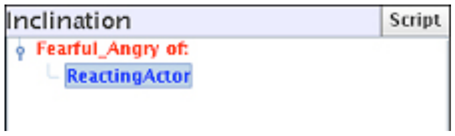


Enter the BNumber value:

0.00

This handsome fellow is our Fearless Leader, Chris Crawford, who is widely admired for his elegance and savoir faire. You can type in a number, so just type in .5 and click on the "OK" button. Ta-da! The value you typed in appears now in the Script. Your Inclination Script asserts that the Inclination to choose this Option is 0.5.
That's a boring Inclination Script. It means that anybody and everybody will always have an Inclination of 0.5 to take the "punch" Option. It doesn't care who they are, how they feel, or anything else. Let's improve on it.

The red 0.5 should still be highlighted; if it isn't, click once on it to select it. From the menus along the right edge of the window, click on the "Mood" menu and you'll see five Operators. The middle one is "Fearful_Angry." That sounds pretty good for a situation like this; if an Actor is angry, he'll be more likely to "punch." Select the "Fearful_Angry" Operator and it replaces the 0.5 in the Script. But now there's a new element as well:



```
Fearful_Angry of:
  ReactingActor
```

Where'd that ReactingActor Operator come from? It's what's called a *default value*. SWAT automatically inserts the most likely Operator in certain cases. In the great majority of Scripts, we have found that authors almost always enter ReactingActor in a spot like this. So we automatically put it there for you, saving you a little time. Isn't that sweet?

Of course, you don't have to accept our default value; if you want to put something else in there, you're welcome to do so. However, in this case, it's the correct value, because ReactingActor always means "the Actor who has assumed this Role." Hence, this Script means that the punchee'sInclination to punch back is equal to how angry the punchee is.

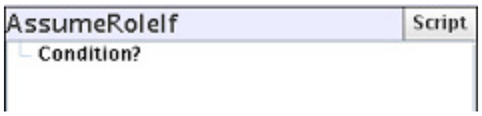
Next we'll tackle a more complex Script, and one you'll probably be using a lot: the AssumeRoleIf Script.

Next Tutorial: Scripting Exercise: AssumeRoleIf
Previous Tutorial: Verb Editing

Storytron: Scripting Exercise: AssumeRoleIf

Last edited by Bill Maya 1 day, 3 hours ago

The AssumeRoleIf script gets used a lot. For every Event, the Engine uses AssumeRoleIf to test all the Actors for the Roles to be filled in the Verb.
Click once on the AssumeRoleIf button, right underneath the "Role" box. Once again you get an empty Script, but this time instead of a default number, you see this:



AssumeRoleIf
Condition?

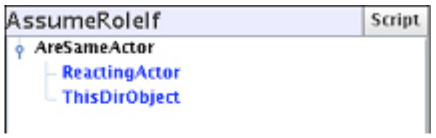
The word "**Condition?**" is what we call a *prompt*: its purpose is to prompt you to replace it with something, in this case, the condition that will determine whether an Actor assumes the Role.

Note that this prompt is black. That's because the AssumeRoleIf Script is a yes-or-no Script (boolean), which we express as "true-or-false."
Click on the Condition? prompt, and note that once again, some menus light up. Click on some of the menus you clicked on previously. Surprise! They show different menu items this time. That’s part of our "can't mix apples and oranges" protection system for you. After all, it would be nonsense for you to enter 0.5 for the Condition?, because 0.5 is not a boolean (true-or-false) value. It's just wrong here, so SWAT won't let you use it or anything else that wouldn't fit.

Let's go back to the original idea for the Role "punchee." The punchee is the DirObject of the Verb of this Event. But we don't tell the Engine "The punchee is the DirObject of the Verb of this Event." Instead, we do it backwards. The Engine goes through each of the Actors one at a time and asks the AssumeRoleIf Script, "Does this Actor fit your requirements?" We need to write an AssumeRoleIf Script that answers that question. The condition that we want is that the Actor (whom we call the ReactingActor) is the DirObject of this Event. (i.e., the one who was punched).

Make sure "Condition?" is still highlighted. (If you accidentally changed it, click on that top item and hit "delete" to go back to the "Condition?" prompt.) Click the large button below "delete" that says "AreSameActor." The condition changes to AreSameActor, and the "Actor1?" prompt is now highlighted.

Select "ReactingActor" (the Actor who has assumed this Role). The Engine will use the comparison Operator "AreSameActor" to test whether the current Actor meets the requirements for the Role. We're checking to find which "ReactingActor" is the DirObject of the Verb "punch," so For "Actor2?" choose "ThisDirObject."
Our script is done:



```
AssumeRoleIf                Script
 AreSameActor
   ReactingActor
   ThisDirObject
```

This means "Assume the Role if the ReactingActor and ThisDirObject are the same Actor."

Yes, it's a backwards way of saying it, but it still makes sense. If you're wondering, ThisDirObject means "the DirObject of this Event, the one we're reacting to."
This backwards way of thinking is one of the most difficult concepts in SWAT. Instead of asking "Who should assume the Role?" we ask "What conditions specify whether an Actor should assume the Role?" There's a reason for this backwardness: the conditions might demand none, one, or many Actors to fill the Role. If we phrase the question in the normal way, then the answer must be phrased like this:

Who should fill the Role? *Joe should fill the Role.*

That implies exactly one Actor to fill the Role. But if we phrase the question in the backwards way, we can give a more complex answer:

What conditions specify whether an Actor should fill this Role? *These conditions.*

Then we just try those conditions out on every Actor and see who ends up fitting those conditions, which could be none, one, or many Actors.

Don't feel bad if this concept leaves you baffled at first; it really is counterintuitive. It will take a while before you feel comfortable with it. But with practice, it sinks in and pretty soon backwards thinking you comfortable with will be. In the next lesson, we'll be using this kind of backwards thinking more broadly.

Next Tutorial: Acceptable and Desirable
Previous Tutorial: Scripting Basics

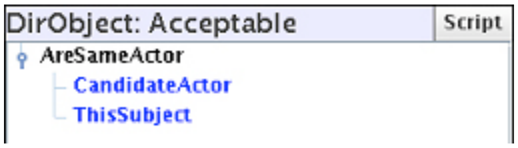
Storytron: Acceptable and Desirable

Last edited by Bill Maya 1 day, 3 hours ago

We're almost done with writing the Scripts for this Option; we have just two more to tackle. These are grouped together under the heading "DirObject", and they're labeled Acceptable and Desirable.

We never tell the Engine "this is the Actor I want" or "this is the Prop I want," because sometimes we want to include more than one Actor, Prop, or Stage. If we used the format "This is the Actor I want," then we'd have problems specifying more than one Actor. Therefore, we use a different approach, where the Engine goes through all the Actors (or Props or Stages) and for each one asks "Should I use this one?" The author tells it which Actors (or Props or Stages) are acceptable, and how desirable each one is.

Back to the punchOption. Who should the puncheepunch? Why, the guy who punched him! That Actor is the Subject of ThisEvent (the one we're reacting to). So all we want is to declare that the Acceptable criterion for the DirObject of the punch Option is that the Actor must be the Subject of ThisEvent:


DirObject: Acceptable Script
AreSameActor
CandidateActor
ThisSubject

This is just like the AssumeRoleIf Script, with two differences. First, we don't use ReactingActor—that is reserved for the punchee in this case. Instead, we use CandidateActor, which always means "the Actor that the Engine is considering for this particular task." And instead of using ThisDirObject, we use ThisSubject. If we used ThisDirObject here, then the punchee would react to being punched by punching himself.

Obviously, there's only one Actor who will pass the Acceptable test, so the Desirable test doesn't matter anymore. If you look at the Desirable Script, you'll see that it's set to a value of 0.0, which is just fine. We'll explain how you use Desirable in another lesson.

Now we need to deal with some tasks still undone. We already added the Verb "run away from" to the Options of the Role "punchee." Now we need to fill some details for this Option. In the WordSocket Acceptable and Desirable Scripts; that should be easy. (Use the same Acceptable script we used for the Option "punch," and leave Desirable at default.) We also need to fill in the Inclination Script. This task challenges your skill at translating what you know about human nature into mathematical terms.

Computers can't understand human language or motives, so we have to translate human behavior into a language the computer does understand: numbers. We tell the Engine that the likelihood that any Actor might take an action will be based on *how much* (i.e., how big a number) the Actor has an inclination to do it, and this in turn is based on his or her personality, mood, or other attributes. In other words, we use a number scale for Actor personality traits and moods, etc. Later on, we will go into more depth about how all this works, but for now, you dont have much material to work with, so we're going to simply tell you what to put into the Inclination Scripts:

Inclination for "run away from":

```
BInverse of:
        Fearful_Angry of:
           ReactingActor
```

Fearful_Angry can be found under "Mood" in the Operators menu.

Recall that we use a numerical scale for traits, moods, and characteristics. The angrier the Actor is feeling, the *higher* a number his or her Fearful_Angry will be. But if an Actor runs away, it is because he or she is afraid—not angry. So we need to flip the Actor's Fearful_Angry mood "upside down." To do this, we use BInverse. The BInverse Operator can be found under "Arithmetic" in the Operators menu. It uses the inverse (opposite) of its argument. Since the higher the value of Fearful_Angry (found under "Mood" in the Operators menu), the more angry the ReactingActor is, the inverse will be how fearful the ReactingActor is, so it fits the Verb "run away from" perfectly.

This completes our preliminary survey of scripting in SWAT. There's much more to learn, of course. In the next lesson, we'll see how all this stuff translates into action. It's time to actually do some interactive storytelling.

Next Tutorial: Running a Storyworld
Previous Tutorial: Scripting Exercise: AssumeRoleIf

Storytron: Running a Storyworld

Last edited by Bill Maya 1 day, 3 hours ago

We've waded through a lot of theoretical mud to get to this point, and at long last we're ready to see this baby in action. There's one last bit of work we need to do: hook up our Verbs to the start of the story.

In the Verb Editor, open the System Verbs menu and choose the very first Verb, "once upon a time." Every single storyworld starts with this Verb, which launches the story. We need to tell it what Event to begin with. Let's set it up so the first thing that happens is that FredpunchesTom.
"Once upon a time" has one Role, "Fate." Delete this Role and add a new one called "Fred." Click on AssumeRoleIf, and choose "AreSameActor." Our script will be:

```
AssumeRoleIf
    AreSameActor
ReactingActor
        Fred
```

Click ReactingActor, then use the "ActorConstant" dialog box to select Fred.

Next, delete the default Option "OK" and add the Option "punch." (click once on "punch" in the pink column, then click on the green "+" next to the Options box). For the Acceptable Script, use this:

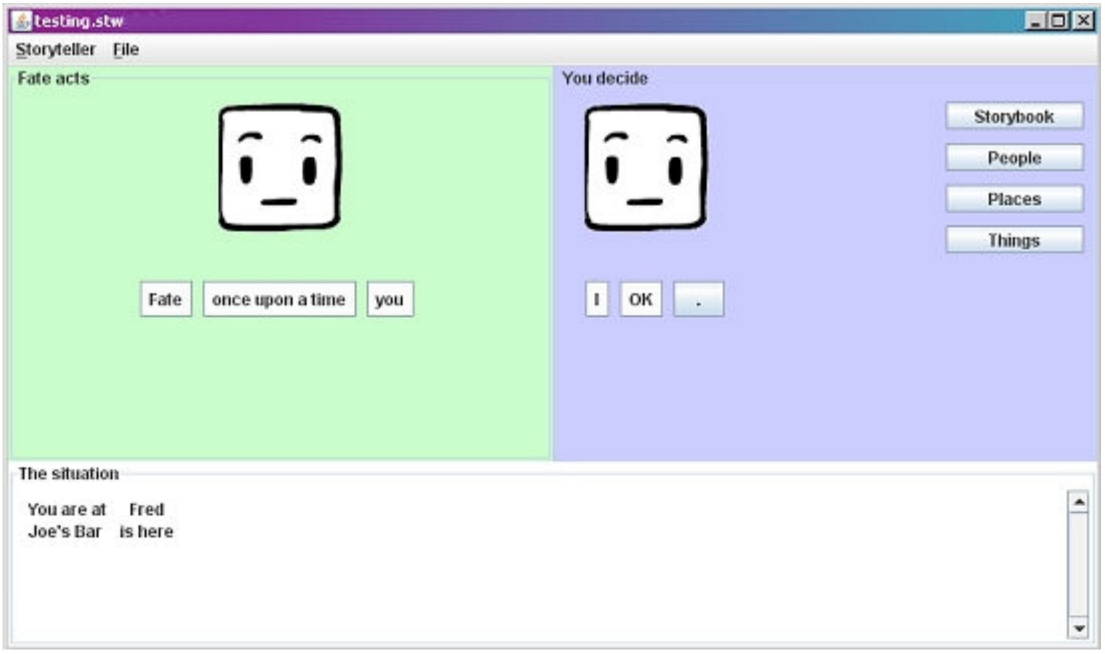
```
DirObject: Acceptable
    AreSameActor
CandidateActor
        Tom
```

Desirable and Inclination both default to 0.0. Leave them that way for now.

Click on the Properties box. Change the Audience to Anybody, and add a 3ActorWordSocket.

This insures that "once upon a time" will be followed by FredpunchingTom. Since Tom is the first Actor in the list, the human player gets to play Tom (i.e., get punched).
So now we are finally ready to play this storyworld! Look under the Lizards menu and select the menu item "Storyteller Lizard." (What's a Lizard? Briefly, it's a tool to assist the author in designing storyworlds. See Lizards for more detail.)

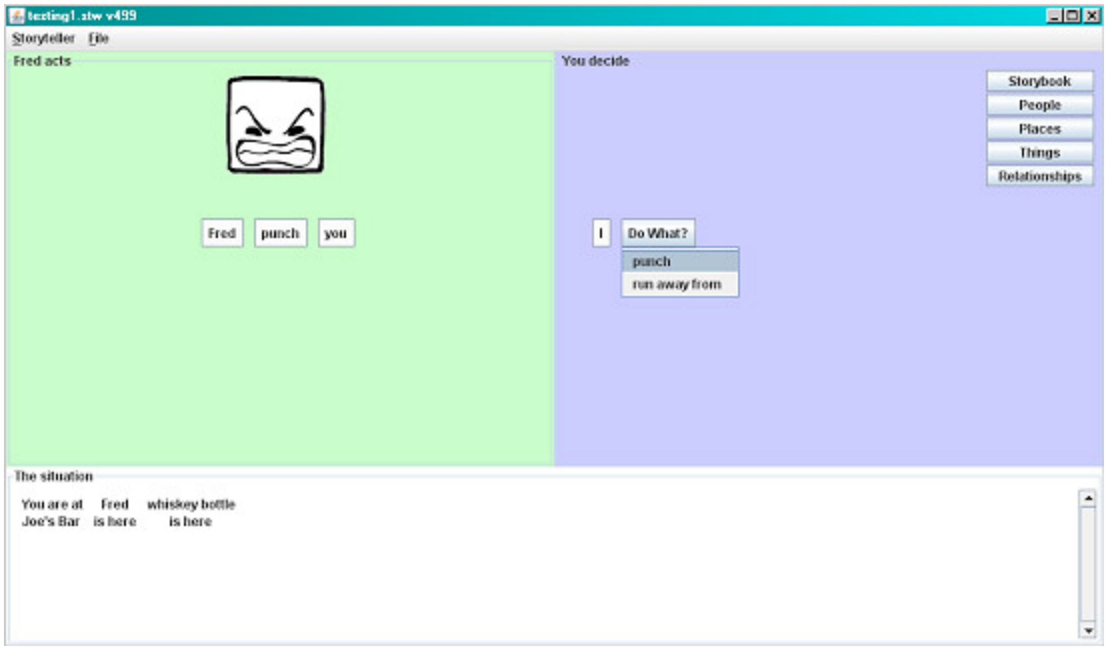
A new window will appear. If you've played a storyworld, it will look familiar:



You are at Joe's Bar   Fred is here

(For more on how to play a storyworld, see the How to Play page on the official Storytron website.)

Click the period after "I OK" on the right half of the screen. Now you'll see:



You are at Joe's Bar   Fred is here   whiskey bottle is here

This is the first Event in your new storyworld. Fred punched Tom. You, playing Tom, now get to decide whether to punchFred back or run away fromFred. Go ahead, punchFred. He'll punch you back. Punch him again. And again. And again. Note how realistically this simulates the observed behavior of males of the species Homo Sapiens in bars. When you get tired of this game, choose "run away from."

Guess what? The story ends! Well, what did you expect? When somebody runs away from a fight, there's no more fighting left to do!

OK, OK, *War and Peace* this wasn't. But it's a start. Close the Storyteller window, and you're back at the Verb Editor. Next we want to make this storyworld a little richer, a little deeper.

Next Tutorial: Enriching the Storyworld
Previous Tutorial: Acceptable and Desirable

Storytron: Enriching the Storyworld

Last edited by Bill Maya 1 day, 3 hours ago

Let’s add some Props to this storyworld. We already have two Props that we created earlier: the whiskey bottle and the chair. Let's have Tom and Fred use them as weapons.

Start by creating a new Verb, "hit with." For its WordSockets, include 3Actor and 4Prop(in case you're wondering, the numbers in front of the WordSocket terms, in this case 3Actor and 4Prop, tell you in what positions they will show up in Storyteller when someone is playing your storyworld). Now go back to the Verb "punch" and add "hit with" as an Option to the Rolepunchee. We'll need to specify the Acceptable and Desirable for the two WordSockets. For the DirObjectWordSocket, we'll simply use the same Scripts that we used with the Option "punch:"

```
DirObject
    Acceptable
        AreSameActor
             ThisSubject
            CandidateActor
   Desirable
   0.0
```

But now we need to write the Acceptable and Desirable Scripts for the 4PropWordSockets. On what basis should an Actor decide which Prop to use?

Let's assign one Prop to each of the two Actors: Tom always uses the whiskey bottle and Fred always uses the chair. We do all of this within the Acceptable Script. We'll need to organize this information in a way the Engine will understand.

First, we want to restrict our considerations to Tom and Fred The beginning of our script structure (not the actual Script) will look something like this:

```
Acceptable
    OR (either of the next two situations is acceptable)
        We're considering Tom and the whiskey bottle together
        We're considering Fred and the chair together
```

Let's flesh out the first part, the one for Tom. To logically put two requirements together, we combine them with AND, like so:

```
    AND (both conditions are met)
        AreSameActor
```

```
            ReactingActor
            Tom
        AreSameProp
            CandidateProp
            whiskey bottle
```

We'll do the same for Fred and the chair. Combining these Scripts with the initial decision (which Actor is being considered), our finished Script looks like this:

```
Acceptable
    OR
        AND
            AreSameActor
                ReactingActor
                Tom
            AreSameProp
                CandidateProp
                whiskey bottle
        AND
            AreSameActor
                ReactingActor
                Fred
            AreSameProp
                CandidateProp
                chair
```

We don't need to worry about the Desirable Script, because the decision is made by the Acceptable Script.

Going back up a level, we need to define the Inclination Script. How does the ReactingActor choose between punch, run away, and hit with? We used Anger as the Inclination for punch and the inverse of Anger as the Inclination for run away. Now it's time for a new idea: let's say that ReactingActor never repeats himself. If, the last time around, he punched, then this time around, he should hit with. And if, last time around, he hit with, then this time he should punch.

This brings us to one of the really neat features of Storytronics: Actors have memories. They remember what they did in the past and use that memory to shape their future actions. To pull off this trick, you use the *HistoryBook* (see HistoryBook Operators for full discussion of this feature).

We're going to use two new Operators: PickUpperIf and IHaventDoneThisSince. The first is in the Arithmetic menu; the second is in the History menu.
Make sure "hit with" shows in the Option window, and click on Inclination. The

prompt shows 0.0. Click on the Arithmetic menu and choose PickUpperIf. You'll see this:

```
PickUpperIf:
    Switcher?
    ValueIfTrue?
    ValueIfFalse?
```

The idea behind PickUpperIf is that sometimes you want to choose between two different numbers depending upon the circumstances. The three arguments can be defined like this:

```
PickUpperIf:
    logical circumstances
    upper number
    lower number
```

If the logical circumstances are TRUE, then it will pick the upper number for the value of Inclination. If the logical circumstances are FALSE, then it will pick the lower number.

The logical circumstance, in this case, is whether the ReactingActor has done "this" (this Option, hitting with) in a certain amount of time. Here's where we use IHaventDoneThisSince.

The Switcher? prompt should be highlighted (if it isn't, click on it), then click on the History menu. Choose IHaventDoneThisSince. You'll get a prompt, HowLongBack? Choose NumberConstant and enter 2, for 2 moments. Here's how our Script looks now:

```
PickUpperIf:
    IHaventDoneThisSince
    2
    ValueIfTrue?
    ValueIfFalse?
```

Now for the numbers. Let's make the upper number 0.99—much bigger than the Inclination for either of the other two Options. And we'll make the lower number -0.99—much lower than the Inclination for either of the other two Options. Use the BNumberConstant button to fill in these values.

```
Inclination
   PickUpperIf:
        IHaventDoneThisSince
        2
```

0.99
-0.99

So if the logical circumstances are TRUE, then ReactingActor will definitely choose the hit with Option, but if the logical circumstances are FALSE, then the ReactingActor will choose one of the other Options. If ReactingActor had indeed used the Verbhit with on his last action, then that would have taken place 1 moment ago (each action takes one game "moment") and would have been caught by IHaventDoneThisSince.

There's one more step: we need to create a Role and Options for the Verb "hit with." Here's an easy way to do so:

We should still be editing the Verb "punch." Look under the main menu option "Edit," and select "Copy Role." This copies the current (and only) Role "punchee" into the clipboard. Next, double-click on the verb "hit with" so that you're set up to edit that Verb. Select "Paste Role" and voila! A copy of the Rolepunchee has been inserted into the Role list for the Verb "hit with." It has the same Options and Scripts that the original Role had, so it will work exactly like that Role works. Change the name of the Role to "hittee," and we're done.

Let's try out our handiwork. Fire up Storyteller Lizard and see what happens. Notice that Fred always alternates between punch and hit with, while you (as Tom) get to do them in any order you wish. That's because Inclinations are not used to make decisions for human players. Only the other Scripts affect human players.

Next Tutorial: Attributes
Previous Tutorial: Running a Storyworld

Storytron: Attributes

Last edited by Bill Maya 1 day, 3 hours ago

An Attribute describes a trait, quality, or property of an Actor, Prop, or Stage. You can create as many Attributes as you want, but we advise you to be careful and not run hog-wild here, lest you create a confusing mess of Attributes. Our experience has been that authors create too many Attributes at first, and then have to pare down their list, which can be very tedious and time-consuming.

You can create Attributes to represent anything that you think is important to your dramatic needs. For example, if you want to build a storyworld about romance, you'll certainly want to have an Attributes for how attractive an Actor is. If it's a macho storyworld for guys, you'll probably want an Attribute for how strong each male Actor is. If it's a Western-type storyworld, then maybe you'll want an Attribute for how quick on the draw an Actor is. You don't want Attributes for everything and anything—you want Attributes that will directly influence the decisions that Actors make.

Attributes are created in the appropriate Editor depending on whether you're describing an Actor, Prop, or Stage. We have a convention that we have found is very helpful when it comes to naming an Attribute: we give it two names connected by an underscore, with the two names representing the meanings of the two opposite senses of the Attribute.

For example, if we have an Attribute for how attractive an Actor is, we don't call it "Attractive"—we call it "Ugly_Attractive." If we want an Attribute for how strong an Actor is, we don't call it "Strong"—we call it "Weak_Strong." Years of experience have taught us that this *bipolar labeling convention* makes it easier to work with Attributes. In fact, we've built this convention into the Storyteller, so that it's easier for the player to understand. The Storyteller takes advantage of the labeling convention to make its presentation to the player a bit easier to understand. But you don't have to follow this convention. If you want to give your Attributes other labels, be our guest. (In fact, there are a few special situations in which it's best *not* to follow the convention—but recognizing those special situations is an advanced topic.)

Each Attribute has a different value for each Actor (or Prop or Stage). You set the different values of the Attribute using the sliders for that Attribute.
But the weirdest, most confusing thing about Attributes is the number system we use for them. Attributes are always measured with a strange kind of number that we call a Bounded Number.

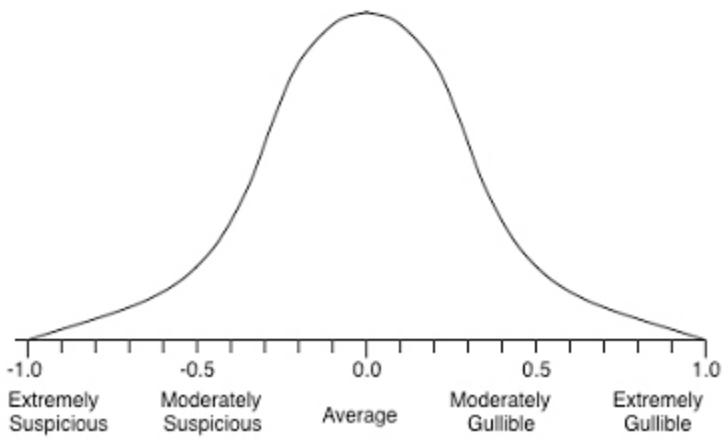
Bounded Numbers

Storytronics is a kind of simulation: it models dramatic behavior with numbers. Designing such models is always a tricky business, and one of the trickiest, nastiest, dirtiest problems in modeling comes from dealing with the numbers themselves. What, exactly, do we mean when we try to put a number on a dramatic personality trait?

For example, suppose we want to take into account how gullible Actors are. We want gullible Actors to have Inclinations that incline them towards trusting-type Options, while we want other Actors to have Inclinations that incline them away from trusting-type Options. Obviously, we'll want to create an Attribute called "Suspicious_Gullible." But what would any such numbers mean? Does a Suspicious_Gullible value of 100 mean that an Actor is really gullible? Does 5 mean that he's really suspicious? What does 3,157,287 mean? Or -37.2?

We all recognize that there are degrees of gullibility and suspiciousness. We can readily understand that Actor X is more gullible than Actor Y, or that Actor Z is just slightly suspicious. So we already know instinctively that gullibility and suspiciousness are concepts that, at least theoretically, could be measured with numbers. But *what* numbers are right for the job?

We have invented, just for Storytron, our very own kind of number that is designed especially for this problem. We call it a Bounded Number, or BNumber. We've designed it from the inside out to make perfect sense and to be easy to use.
Whenever you use a BNumber, you think in terms of the average, not the absolute. You don't think "How many units of gullibility should I assign to Fred?" Instead, you think in terms of a bell curve of suspiciousness and gullibility, like this:



Most people are close to the average in overall suspiciousness versus gullibility. Some people are more suspicious and some are more gullible. And very few are extremely suspicious or extremely gullible. So you use a value of 0.0 to indicate average or normal values, a value of 1.0 to indicate the maximum possible degree of gullibility, and -1.0 to indicate the maximum possible degree of suspiciousness. (In practice, we never use the values of +1.0 or -1.0, because they're supposed to represent "the impossible extreme." So we normally just use +0.99 to mean "really really really big one way" and -0.99 to mean "really really really big the other way.")

This reliance on thinking in terms of bell curves makes a lot of problems go away, because now you don't have to worry about the units of measurement. Suppose you want an Attribute for how short or tall an Actor is. You don't have to worry about whether an Actor is 5 foot 4 inches tall or 162 cm tall. You just decide that average height is, say, 5 foot 6 inches, and really really really short is 4 feet 0 inches and really really really tall is 7 feet 6 inches. Then you can estimate that 5 feet 4 inches corresponds to a BNumber of about -0.05.

Nor do you have to worry about weird scaling problems, like multiplying 6,487,265 "Joy thingies" by 0.125 "Honest thingies." Sad_Joyful runs from -1.0 to +1.0, and so does False_Honest. Everything is always measured on the same scale—no matter what you're using!

In BNumber Arithmetic, everything always works out right. You can add, subtract, and do whatever you want with BNumbers, and they *never* go outside of their legal range from -1.0 to +1.0. No matter what you do to them, they always stay legal. (More about Bounded Numbers in BNumbers, Unumbers, and Numbers)
Let's use an Attribute in our growing storyworld. Go to the Prop Editor and use the green "+" box in the top center to add a new Attribute under "Core Prop Traits." Call that Attribute "Harmless_Lethal." This will represent how injurious a Prop is when used in a fight. Now add three more Props: pillow, cane, and club. Be sure to set their locations to "Joe's Bar." Use the sliders to assign Harmless_Lethal values to the Props as follows:

pillow - leftmost tick (low)
cane - next tick to right (medium-low)
whiskey bottle - center tick (average)
chair - next tick to right (medium-high)
club - rightmost tick (high)

Let's use this new Attribute to make the decision-making for the Option "hit with" more interesting. Let's say that, when you hit somebody with something, you choose the *least* lethal weapon that's still *more* lethal than the one you were hit with. In other words, if somebody hits you with something, you up the ante, but you don't immediately jump all the way to the most lethal weapon. So let's change the Acceptable and Desirable scripts for the 4PropWordSocket for the Option "hit with" for the Role "hittee" for the Verb "hit with."

But first, a little digression: did you notice how clumsy the previous sentence is? We had to use a long string of prepositional phrases to specify exactly what script we're talking about. After having written too many of these long, tedious sentences, we came up with a shorthand that's much easier to use and understand. The form is as follows:

Verb: Role: Option: WordSocket: {Acceptable or Desirable}

So in this case, that entire sentence could have been reduced to:

hit with: hittee: hit with: 4Prop: Acceptable

Back to work. We need to write an Acceptable Script and a Desirable Script. Go to the Verb Editor and make sure the Verb selected is hit with. The Role should be hittee and the Option should be hit with, just as in the shorthand sentence above.

Do you recall that the Inclination Script never affects the player? The computer uses the Inclination Script to make decisions for the other Actors, but the human player always gets the full choice of Options. Well, the same principle applies to Acceptable and Desirable. We want the human player to be able to choose from *any* of the possible Props, but we want the computer-controlled Actors to choose based on the escalatory algorithm we described above.

The 4Prop Acceptable Script determines what choices the human player sees on the menu, while the Desirable Script decides which of those menu items the computer Actor will choose. This, it turns out, makes some of our Scripting work easier—and some harder.

The Acceptable Script is ridiculously easy. We want the human player to be able to choose whatever Prop he wants. Hence, the Acceptable Script looks like this:

```
Acceptable
  true
```

This means that, whenever the Engine asks, "Is this Prop acceptable?" the answer is always "Yes." Every Prop will be Acceptable! So the human player can choose any Prop he wants.

Click on Acceptable under 4Prop and erase the old Acceptable Script by clicking on the highest Operator ("OR") and hitting the delete key. Then press on the "true" button just above the Scripting box. Done.

For the computer Actors, we need to use the Desirable Script to implement the escalatory idea sketched above. We want the Desirability to be negative when the Prop in question has lethality lower than the lethality of the Prop that was just used on the Actor, and maximum for the Prop that has the lowest lethality of the remaining Props. Here's a Script that does this:

```
Desirable
```

```
PickUpperIfof:
    TopGreaterThanBottom(BNumber) of:
        Harmless_Lethal of:
            CandidateProp
        Harmless_Lethal of:
            This4Prop
    BInverse of:
        Harmless_Lethal of:
            CandidateProp
    -0.99
```

The TopGreaterThanBottom(BNumber) Operator is in the "Logical" menu; and the Harmless_Lethal Operator is in the "Prop" menu. This4Prop is in the "ThisEvent" menu.

Let's take this Script apart and explain it piece by piece. The PickUpperIf Operator will pick the upper term (the BInverse term) if the condition is true; if the condition is false, then it will return the lower value: -0.99. Obviously, a Desirable of -0.99 is very undesirable and the CandidateProp with that value won't be selected.
Let's suppose that the logical condition turns out to be true. Then the upper term (the BInverse one) will be chosen. What does that mean? BInverse is a really simple Operator: it simply reverses the sign of its argument. Thus, BInverse of -0.5 is +0.5; BInverse of +0.25 is -0.25, and so forth. So the CandidateProp with the highest value of Harmless_Lethal will have the lowest BInverse result, and the CandidateProp with the lowest value of Harmless_Lethal will end up with the highest BInverse result.

But now we have to consider the biggest term in this Script, the logical term TopGreaterThanBottom(BNumber). This yields true if the top number of its two arguments is bigger than the bottom number, and false otherwise. In other words, it asks, Is the Harmless_Lethal of the CandidateProp bigger than the Harmless_Lethal of This4Prop, the Prop that I was hit with? If so, then we pick the BInverse term. If not, we pick the -0.99 term.

Let's suppose that Joe has just hitFred with the cane, and Fred is running the Desirable Script for the Prop he's going to "hit with" back. The CandidateProp with the highest Desirable value is the whiskey bottle, which is exactly what we wanted.

Let's try it in Storyteller Lizard to watch the algorithm at work. There's one more thing we need to adjust first: we changed the 4Prop Acceptable script to "true" in the hit with Option under hit with, but the old script (the one that forces Fred to use the chair and Tom to use the whiskey bottle) is still in the hit with Option under punch. If we don't change it, the first time the player (Tom) has the choice of using hit with, the only Prop he can use is the whiskey bottle.

Welcome to the nit-picky world of scripting.

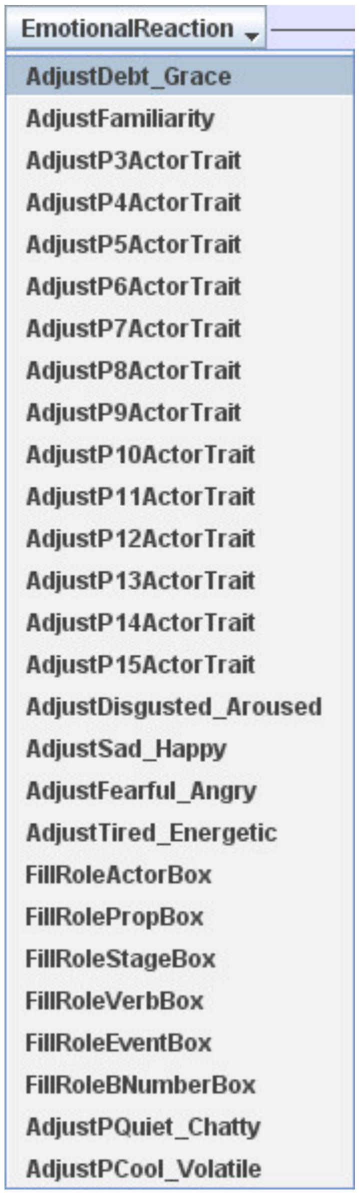
If you want to see this problem happening, try it out in Storyteller Lizard. To fix it, use the Verb Editor to edit punch, and in the hit with Option, change the 4Prop Acceptable script to "true." Now the first time Tom has the choice of using hit with, he has all the Props to choose from.
Try hittingFred with various Props, and watch how he responds. The escalation algorithm guides his choices of what Prop to hit with. Have fun!

Next Tutorial: Emotional Reactions
Previous Tutorial: Enriching the Storyworld

Storytron: Emotional Reactions

Last edited by Bill Maya 1 day, 3 hours ago

Let's look at the Emotional Reaction menu that we skipped over previously. You'll find it in the second (blue) column of the Verb Editor, just below the AssumeRoleIf Script button. When you click on it, you get this menu:



There are three groups of items here: "Adjust____" items, "FillRoleBox" items, and "AdjustP___" items. Let's tackle them in sequence.

The "Adjust____" items refer to Debt_Grace, Familiarity, Disgusted_Aroused, Sad_Happy, Fearful_Angry, and Tired_Energetic. Except for the first two, these are Moods. A Mood in Storytron is a value that will automatically die down over about 10 moments. So if you bump up a Mood, it will remain significant for about 10 moments, after which time it reverts to a value of 0.0. Let's focus on the last one, AdjustTired_Energetic. This concerns the degree to which a person is feeling exhausted or pumped up. Obviously, if a person is tired, they won't hit as hard. Also, the longer a person fights, the more tired they become. Let's implement this idea.

But first, what is meant by "Adjust?" Often we simply want to set a variable to a new value. For those situations, we have Operators that start with the word "Set." However, there are also plenty of situations in which we want to start with the existing value—whatever it may be—and nudge it one way or the other. For these situations, we use Operators that begin with "Adjust." When we use Tired_Energetic, we want to use the Adjust Operator, not the Set Operator, because an Actor who does some labor becomes *more* tired with each exertion.
So we have two changes to make in our storyworld. First, we must adjust Tired_Energetic downward each time an Actor uses the Verb "hit with." Second, we must insure that the Actor hits only as hard as his Tired_Energetic permits.

The first task is easy: all you do is go to Verb "hit with," Role "hittee" and select the menu item "AdjustTired_Energetic" from the EmotionalReaction menu. This adds a Script button "AdjustTired_Energetic in the space just underneath the EmotionalReaction menu. Click on that Script button and you'll see the Script for AdjustTired_Energetic. It's empty—it just shows the prompt "How much?" Just fill in the BNumberConstant -0.5 there. This will adjust Tired_Energetic of the ReactingActor downward (more tired) every time the Role is assumed.

Here's an extra-credit question. Did you notice what we just did? We just adjusted *ReactingActor's* Tired_Energetic! But who is the person who just did the work of hitting? It was *ThisSubject*! To put it another way, let's say Fred has just hitTom with a whiskey bottle. Fred, then, is ThisSubject—the person who did the hitting. He's the one doing the work, and he's the one who should be getting tired. But it's Tom who is getting more tired—not Fred!

However, it's probably reasonable to assume that the person getting hit is going to get worn out as well. Besides which, since both gents are in a brawl, they will both gradually get more worn out, so this approximation of reality works.

It's perfectly OK to take shortcuts like this, as long as you understand what you have done and the limitations of your assumptions.

Now we have to make Tired_Energetic affect the Actor's behavior. To do this, we'll need to make a change in the Verb "hit with."

Go to that Verb and open the Properties box. Add a WordSocket "5Quantifier." This changes the meaning of "hit with" to include the notion that an Actor can hit with varying degrees of force.

This simple change has had consequences elsewhere in your storyworld, because now you must supply the Acceptable and Desirable Scripts for the two places in which the Verb "hit with" shows up as an Option. How do you find those two places? Easy. Just look under the Lizards menu and select the menu item "ComeFrom Lizard." The little window that pops up shows you all the Roles in which the Verb you're editing (which should be "hit with") is an Option. All you do is double-click on one of those listings and poof! you're there, looking at the Option. Do so.

Now we're looking at a Role with an Option of "hit with." There is now an entry for the "5Quantifier" WordSocket, containing both an Acceptable Script and a Desirable Script. If you look at these two Scripts, you'll realize that they haven't been written yet—they just contain Operators with question marks in them. You have to write those two Scripts.

The first, Acceptable, is pretty easy to write. *Any* Quantifier is acceptable in this situation—except for the one we call the "Interrogative Quantifier," which is the term "how much?" So your Acceptable Script has only the task of locking out the Interrogative Quantifier. That's done like so:

```
NOT
    QuantifierIsInterrogative
CandidateQuantifier
```

The Operator "QuantifierIsInterrogative" is in the Words menu on the right side.

The Desirable Script is more difficult to write. Its job is to select the Quantifier that matches the level of Tired_Energetic of the Actor. If the Actor has a high value of Tired_Energetic, then you want to maximum Desirability for a large Quantifier. If the Actor has a low value of Tired_Energetic, then you want maximum Desirability for a low Quantifier. Fortunately, this is all handled quite nicely for you with a handy-dandy Operator called "Suitability." The Desirable Script looks like this:

```
Suitability of:
    CandidateQuantifier
    Tired_Energetic of:
        ReactingActor
```

That's all there is to it. The Suitability Operator, which can be found in the Word menu, figures out how the value of its second argument (the BNumber argument) fits into the scale of Quantifiers, and then compares that fit with the CandidateQuantifier. If the match is close, it gives a high value; if the match is not close, it gives a low value. This insures that Tired_Energetic will determine the chosen value of the Quantifier used in 5Quantifier.

Go ahead and add these two Scripts. Then go back to the Verb "hit with" and check ComeFrom Lizard to make sure that you have fixed both instances of "hit with" as an Option. (You can use "Copy" to add copies of the scripts to the other Option.) When you have taken care of both instances, try out the storyworld. Does it work as you expected?

Next Tutorial: Consequences
Previous Tutorial: Attributes

Storytron: Consequences

Last edited by Bill Maya 1 day, 3 hours ago

Now let us turn to the "Consequences" section near the top of the middle blue column. A Consequence is just like an EmotionalReaction, with one big difference: EmotionalReactions are subjective, while Consequences are objective.

Suppose, for example, that Fred hits Tom so hard that Tom is injured. Tom can have an EmotionalReaction to being hit: he might become more angry. That's a purely subjective response; different Actors might respond differently to being hit. It also depends on your point of view. If you're the one being hit, you're probably going to get angry. But if you're a bystander, you might not care at all. Hence, the EmotionalReaction is a purely subjective effect, and is always tied to a specific Role. But the injury is not a subjective matter; that always happens irrespective of the personality of the hittee. That's what we have Consequences for.

You apply Consequences in exactly the same fashion that you apply EmotionalReactions. Click on the Consequences button to raise the Consequences menu, which is itself a list of submenus listing different variables. When you select one of those secondary menu items, SWAT creates a new Script button just underneath the Consequences button. If you click on that Script button, you'll put that Script into the Script editing area on the right, where you can edit it.

Let's try it out. First, go to the Prop Editor and modify the ownership of the Props as follows:

Tom gets whiskey bottle, chair, and pillow
Fred gets cane and club

Next, go the Script Other: hit with: hittee: hit with: 4Prop: Acceptable (remember, this is our shorthand for Category Other, Verb hit with, Role hittee, Option hit with, WordSocket4Prop, Acceptable Script). The Acceptable condition is currently set to "true," meaning that every Prop is acceptable. Let's change this to

```
AreSameActor
    ReactingActor
    Owner of:
        CandidateProp
```

You can find "Owner" in the Prop menu.

This means that the only Props that are Acceptable are those that the ReactingActor owns. Sounds reasonable, doesn't it?

We've set up the situation, now let's make a Consequence. Let's say that, when one Actor hits another with a Prop, then the other Actor gets ownership of the

Prop. So, while still in the Verb "hit with," go to the Consequences menu, submenu "SetProp," and select "SetOwner." That creates a new Script button for SetOwner. Click on that button to edit the Script. You see the following:

Prop?
Owner?

The Prop is This4Prop, the Prop that was used to hit with. This can be found under "This Event" in the menu at right.

The Owner is ThisDirObject, the Actor who was hit. Again, it's found under "This Event."

After you fill out this script, run Storyteller and see if it works as you would expect.

Next Tutorial: Hijacking
Previous Tutorial: Emotional Reactions

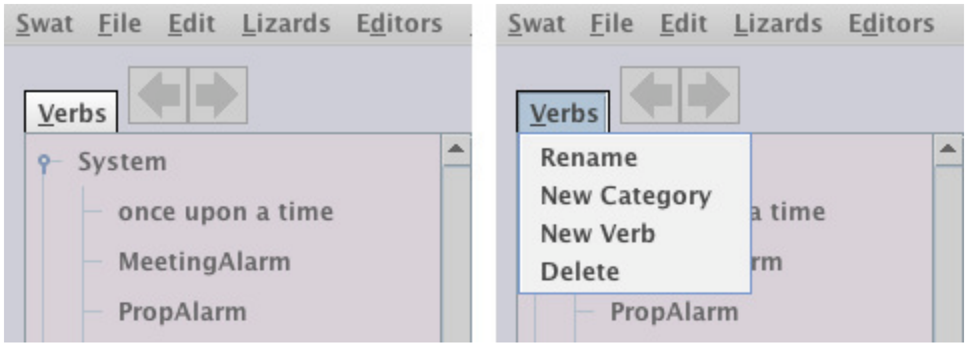
Storytron: Hijacking

Last edited by Bill Maya 1 day, 3 hours ago

There are sometimes situations in which you might want to have several Actors reacting to an Event. Sometimes you'll want them all to act, but sometimes you'll want the action of one Actor to block the actions of others. For example, if somebody shouts, "There's fire in the kitchen!" and John reacts to the Event by dousing the fire, you don't want somebody else to douse the fire as well; one dousing will do.

To demonstrate this, we need a third Actor. Go to the Actor Editor and create a new Actor, Mary. Set her location to Joe's Bar, then return to the Verb Editor. Let's say that Mary is your (Tom's) girlfriend, and she doesn't like to see people beat you up. So, if somebody (say, Fred) uses "hit with" on Tom, then Mary should intervene by pleading with Fred to desist.

So first we must create the Verb "plead to desist." Click on the "your first category," then click the Verbs tab and choose "New Verb." (You can also right-click, or control-click on Mac, anywhere in the pink column.)



In the Properties box, add a 3ActorWordSocket for the DirObject. Now go back to "hit with" and create a new Role; let's call it "girlfriend." Now, we could get clever here and create an ActorAttribute that specifies just who is hitched up with whom, but in this case, there's a shortcut: Mary is the only female here, so let's take advantage of that. However, there's another problem: we don't want Mary to plead with Fred if JoehitsTom; we want her to plead with Fred if Fred hitsTom. For the AssumeRoleIf Script, just use:

```
AND
  NOT
    Male
        ReactingActor
  AreSameActor
    ThisSubject
    Fred
```

"Male" is found in the Actor menu.

Now let's give Mary the Option to plead with Fred. Click once on the Verb "plead to desist" in the left column and then click the green "+" plus button next to the Options box. That adds "plead to desist" to the Option list for the Role "girlfriend."

Next fill in the two Scripts for the DirObjectWordSocket. They're quite simple; you want Mary to plead with the Subject of "hit with"—whomever did the hitting:

```
Acceptable
    AND
        AreSameActor
            ThisSubject
            CandidateActor
        AreSameActor
            ThisDirObject
            Tom
```

Note the second comparison, establishing that Tom is the one who was hit. If we left this out, Mary would plead with Fred if he hit anyone, not necessarily Tom. Right now we only have Tom, Fred, and Mary in the storyworld, but we might add other actors later, and this script will make sure that Mary only pleads on behalf of Tom.

The Desirable script can remain at 0.0.

We also need to assign an Inclination value. Use 0.0 for this as well.

Finally, we don't want Mary ever responding with a simple "OK," so delete that Option.

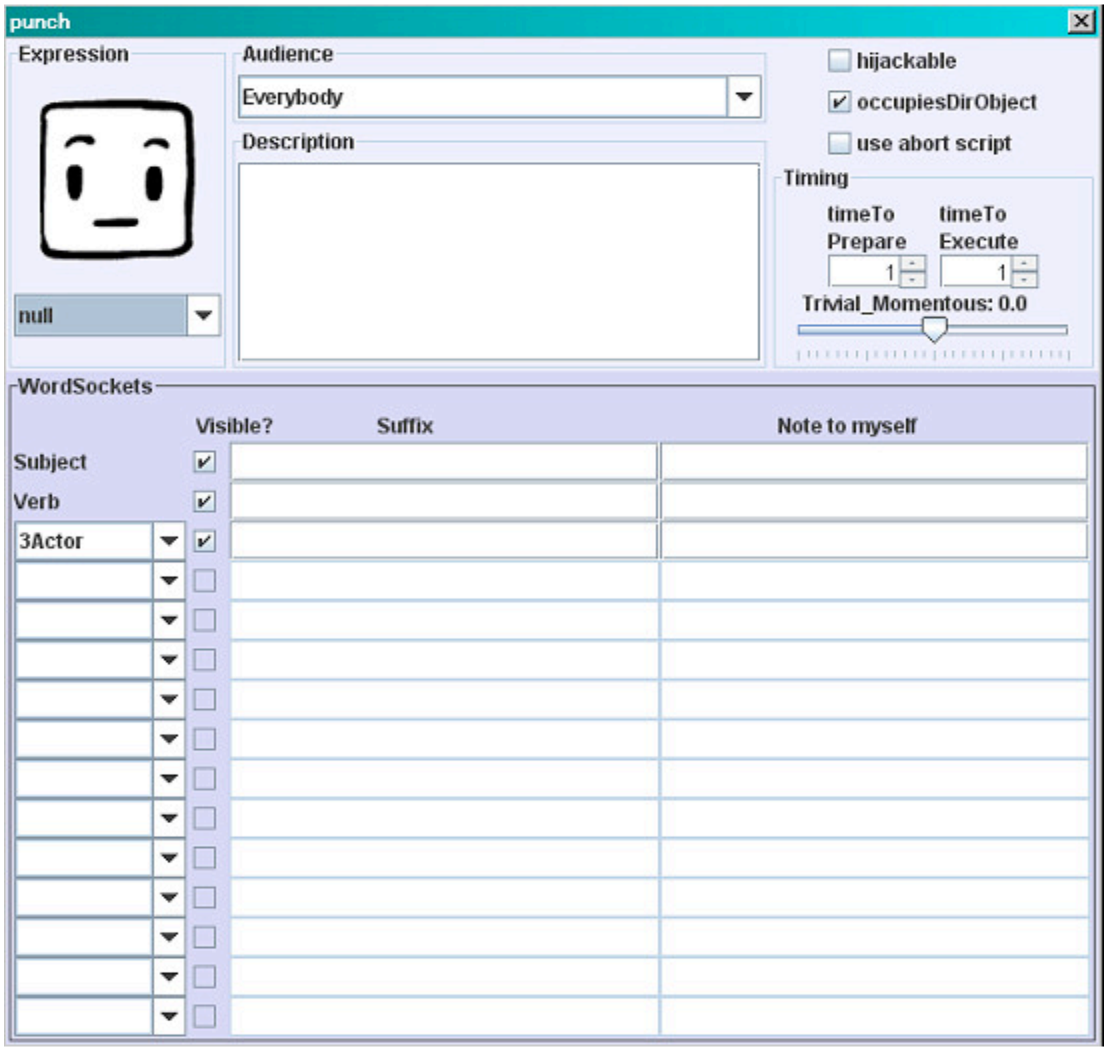
There's just one last step to take here: we need to engage the hijacking feature. To do that, click once on the "Properties" button, and check the checkbox marked "Hijackable."

You're all set. Run Storyteller and see what happens.

Next Tutorial: Properties Box
Previous Tutorial: Consequences

Storytron: Properties Box

Last edited by Bill Maya 11 hours, 48 minutes ago



Audience: Everybody

[ ] hijackable
[✓] occupiesDirObject
[ ] use abort script

timeTo Prepare: 1
timeTo Execute: 1
Trivial_Momentous: 0.0

	Visible?	Suffix	Note to myself
Subject	✓		
Verb	✓		
3Actor	✓		

We've used the Properties Box before, but never taken a full tour of its features. They are:

Expression: Used to choose the Emoticube that will represent the Verb in the Storyteller display.

Audience: Used to decide who witnesses the Event. This is covered in detail in Audience Requirements.

Description: The Author uses this to describe what the Verb does. This text will show up as a tooltip in Storyteller, telling the Player exactly what the Verb means and what will happen if they choose that Verb.

hijackable: Whether or not the Verb can be hijacked (see Hijacking).

occupies DirObject: Usually the DirObject is part of the Event and so is occupied by it (unable to do anything else until it ends). However, there are a few rare cases in which you might not want the DirObject to be occupied by the Event. These are usually cases in which the Audience is set to “Mental State” or “Subject Only”.

use Abort Script: Whether or not you want the Verb to have an Abort Script (see Abort Script).

TimeToPrepare: The amount of time that must elapse after setting the Plan before it can be executed. Usually TimeToPrepare is 1.

TimeToExecute: How long the Subject is tied down in the execution of the Verb. Usually TimeToExecute is 1.

Trivial_Momentous: Useful for measuring the dramatic import of a Verb. Will be more important in future versions of Storytron.

WordSockets: Here's where the Author creates the WordSockets used by the Verb. Every Verb has two default WordSockets: Subject and Verb. Up to thirteen additional WordSockets can be created for Actors, Props, Stages, certain Traits, and Quantifiers that will participate in the Verb (detailed examples in WordSockets).

Visible?: Whether the WordSocket is visible to the Player. Usually WordSockets are visible, but in certain cases, you may want to hide them from the Player.

Suffix: Here's where the Author defines additional words to be displayed in the Deikto Sentence depicting the Verb to the Player, making the Sentence easier to understand. Example in WordSockets.

Note to Myself: For the Author's use in keeping track of which WordSocket holds which component. Contents of this field appear as a tooltip in the Options display of the Verb Editor.

Next Tutorial: Abort Script
Previous Tutorial: Hijacking

Storytron: Abort Script

Last edited by Bill Maya 6 days ago

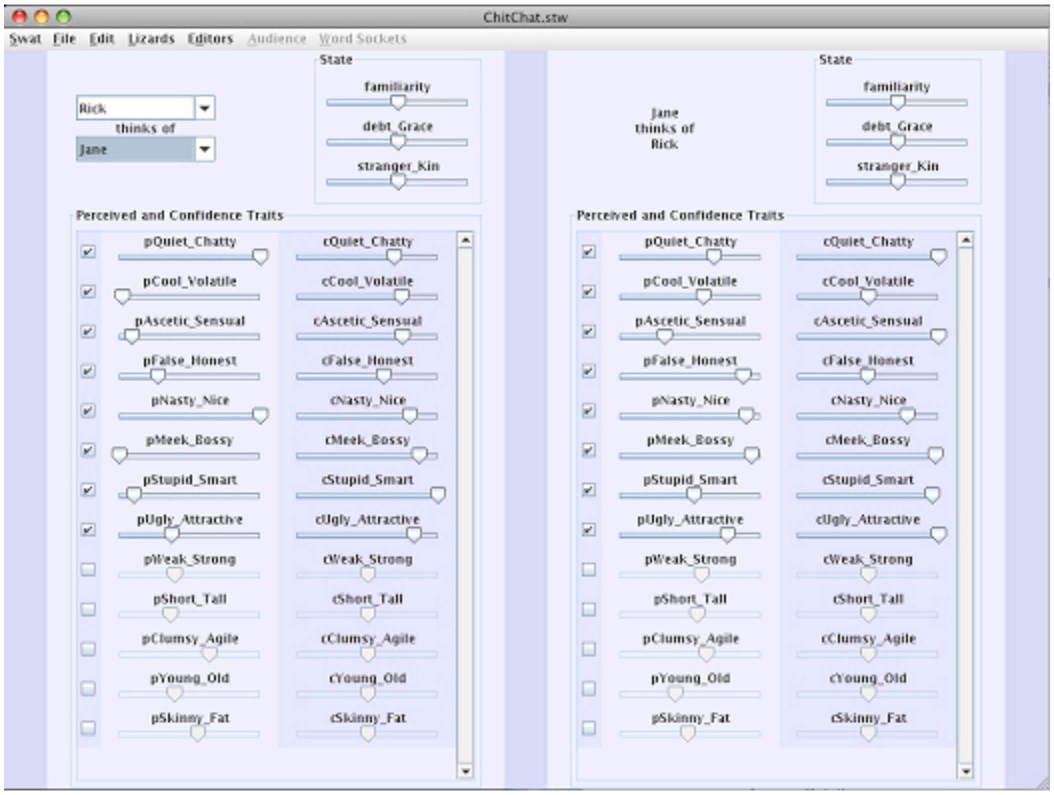
Very rarely, you'll encounter a situation in which you need an Actor to abort an already-established Plan. For example, suppose that Mary decides to poison Tom and the first step of her scheme is to cook the poison, which takes several hours. While the poison is cooking, Tom comes by to visit and sincerely apologizes for his past transgressions against Mary. They kiss and make up. However, the poison is still cooking and when the cooking is complete, Mary has a Plan in place to put the poison into Tom's coffee. How do you prevent this from happening?

The old way was to insert some sort of test in the Inclination Script that asked "Gee, has Tom apologized for his transgressions?" This turned out to be clumsy, and the Scripts necessary to make it work were long and complicated. So we designed a better system: the Abort Script. This Script is so rarely needed that we don't make it an automatic part of the normal scripting process. Instead, it is optional. To turn on the Abort Script, you have to open the Properties box for the Verb and check the "Abort" checkbox therein. This places a new Script button (Abort) above the Consequences menu button. This Script takes a boolean value; if it evaluates to *true*, then execution of the Plan containing that Verb will be aborted. The Engine checks the Abort Script whenever it is about to execute a Plan containing that Verb.This way, Mary could abort poisoning Tom and two minutes later, Jennifer could proceed with poisoning Bob.

Next Tutorial: Relationship Editor
Previous Tutorial: Properties Box

Storytron: Relationship Editor

Last edited by Bill Maya 6 days ago



The Relationship Editor handles the tedious task of defining all the relationships among all the Actors. There are quite of few of these, and it can take a lot of work to fill in all the data, so the Relationship Editor also provides a few tools for easing your workload.

But first, we have to explain a few concepts. The core concept is the ActorAttribute. You learned about Attributes in Lesson 10. What we didn't tell you is that Attributes can be *perceived* by other Actors—and that these perceived values are not necessarily the same as the actual values.

Let's illustrate this with an example: imagine an Attribute called False_Honest, representing the integrity of an Actor. Rick's False_Honest value, as specified in the Actor Editor, is how False_HonestRick truly is. The pFalse_Honest value (the "perceived" False_Honest value) of Jane for Rick is how False_Honest Jane *thinks* Rick is (i.e., how much she trusts him). The cFalse_Honest is how sure she is of her perception. For example, if she knows only what's she has heard of him on the grapevine, cFalse_Honest will be low, but if she has direct experience of Rick's False_Honest (if, for example, she knows that he lied to her), then her cFalse_Honest will be higher.
This is important because, in drama, different people have differing estimates of others. Those differing estimates are the source of so much dramatic fodder, as

people gossip about others. And these differing perceptions will be the source of some conflict themselves. Suppose, for example, that Francesca tells Jane that she thinks Rick is trustworthy (Rick's False_Honest value is high). Jane, however, believes that Rick is not trustworthy (Rick's False_Honest value is low). Jane will note the discrepancy between her existing estimation of Rick and what Francesca is telling her. Perhaps she will modify her estimation of Rick's False_Honest upwards. Perhaps she will decide that Francesca is lying. It depends on some other factors. You can see how our expert Laura Mixon handled this tricky problem in this portion of an Inclination script from her demonstration storyworld, ChitChat:

```
Tell re personality: tellee: contradict: Inclination
...
Blend of:
    1. I have a big diff in perception from you
    BAbsval of:
        BDifference of:
            CorrespondingPActorTrait of:
                ReactingActor
                This4Actor
                This5ActorTrait
            Quantifier2BNumber of:
                This6Quantifier
    2. I am certain I am right in my perception
    BInverse of:
        CorrespondingCActorTrait of:
            ReactingActor
            This4Actor
            This5ActorTrait
...
```

Going back to the Actor Editor, we can now understand the two concepts of Accordance and Weight. Accordance is how readily an Actor accords high values of the Attribute to others. Thus, if Jane has a high accordFalse_Honest, then she readily assumes that other Actors have high values of False_Honest—in other words, she's gullible. If she has a low value of accordFalse_Honest, then she's suspicious: she doesn't give people a high value of False_Honest unless they prove it to her.

False_HonestWeight is the degree to which an Actor desires to have high pFalse_Honest values—in other words, how much that Actor desires to be trusted. An Actor who is vain will have a high value of Ugly_AttractiveWeight; an Actor with a low value of Cowardly_BraveWeight doesn't care if other people think he's a coward.

There are three special variables that require your attention: familiarity, debt_Grace, and stranger_Kin. These are all BNumber variables. The first, familiarity, is useful for initializing p-values. Notice the little checkboxes next to each of the regular p-Attributes. If you check one of those, then you're telling SWAT "I'll fill in this value myself." But if you uncheck this box, then you're telling the Engine, "Fill it in automatically for me." The Engine will use the familiarity value along with the accordance value to calculate the p-value directly, so you don't have to. Here's the algorithm that the Engine uses:

1. It starts off assuming that the p value is 0.00 (the most likely case)
2. Then it biases it towards the actual value in proportion to the familiarity value.
3. Lastly, it biases it up or down in proportion to the accord value

It's a big time-saver, and it yields good results in most cases. You'll want to use the manual override (checking the box) rarely.

When you use the automatic procedure by unchecking the checkbox, the c-value is the same thing as the familiarity value.

Next comes the debt_Grace value. Normally, you'll just initialize this to zero. It represents the idea of *tanagadalang* (if you're Indonesian) or an interpersonal kind of *karma*, or just the idea that "You owe me." We've found that it can be very handy in a lot of drama. Remember, it's not necessarily symmetric: two different Actors can have very different ideas of who owes whom.

Lastly, there's the stranger_Kin relationship. This is used to handle the dramatic relationship expressed in the adage "Blood is thicker than water." It can also be used to handle kinship based on marriage, adoption, and so on. It's a little odd, though, in that 0.0, which normally expresses the "average" value, in this case should reflect something a little more than that.

Depending on the range and importance of kinship in your storyworld, you can set the Actors' relationships accordingly. For instance, in a tight-knit family drama, you may want to set 0.0 as cousins. In a storyworld about racial relations or the clash between two cultures, 0.0 might represent people of the same clan or ethnicity. In storyworld of a first contact with extraterrestrials, 0.0 might mean two Actors are simply of the same species!

Next Tutorial: Spying
Previous Tutorial: Abort Script

Storytron: Spying

Last edited by Bill Maya 6 days ago

What drama would be complete without somebody spying on somebody else? You can make this happen in your storyworlds by using the Consequence Script SetActor: SetSpyingOn.

This Operator takes three arguments:

Spy: the Actor who should do the spying

SpiedUpon: the Actor who should be spied upon

HowLong: the number of minutes that the Spy should continue spying

\Once this Script executes, the Spy follows SpiedUpon around and witnesses every overt action taken by SpiedUpon, but remains invisible to SpiedUpon. Spy cannot witness Events with Verbs of audience type MentalEvent or CheekByJowl, but does witness anything that a regular witness would see—without SpiedUpon knowing that he's being observed.

Next Tutorial: Script Editing Tips
Previous Tutorial: Relationship Editor

Storytron: Script Editing Tips

Last edited by Bill Maya 6 days ago

Here are detailed descriptions of some of the editing conveniences we provide.

Cut, copy, and paste

All of the regular cut, copy, and paste tools work inside Scripts. When you use them, they act on the selected Operator and all its arguments. Thus, you can move big chunks of script around with cut and past. Here's an example. Suppose that you realize that you need to reverse the order of subtraction in this Script:

```
BDifference of:
    Blend of:
        Nasty_Nice of:
             ThisSubject
        Nasty_Nice of:
             ThisDirObject
        0.0
   0.5
```

so that the 0.5 is on top and the Blend is on the bottom. To do this, simply select the Blend Operator, cut, select the 0.5 Operator, and paste. Lastly, enter the 0.5 value in the upper slot.

Outsertion

If you select an Operator in a Script and hold down the Control key while choosing a new Operator from a Menu, that new Operator will be "outserted"—placed above the selected Operator. Here's an example.

Existing Script:

```
    BSum of:
        Fearful_Angry of:
             Joe
        0.5
```

Select Operator "BSum," then click on "BSum of:" and hold down the Control key while opening the Arithmetic menu. Select menu item "BInverse" and get this result:

```
    BInverse of:
        BSum of:
             Fearful_Angry of:
                  Joe
```

```
      0.5
```

You'll use outsertion quite often in your daily scripting work.

Replacement:

You can replace one Operator with another by holding down the Option key before opening a menu. Example:

Existing Script:

```
  BSum of:
    Fearful_Angry of:
       Joe
    0.50
```

Select the Operator BSum, then hold down Option and open Arithmetic menu, selecting "BDifference." Get new Script:

```
BDifference of:
    Fearful_Angry of:
       Joe
    0.50
```

What's especially nice about this trick is that it preservers the arguments of the original Operator. If you used the direct route, you'd have to erase those internal arguments. This way, you don't erase them. There is a catch, however. You can't just replace any Operator with any other Operator. The only Operators that will be available for you to select are those whose argument types match the argument types of the original Operator.

This capability extends even to Operators with fewer or more than the number of arguments of the original Operator. For example, if you replace AND3 with AND4, the new AND4 will contain the previous three boolean arguments and add a fourth, undefined boolean Operator at the end. If you go in the other direction, replacing AND4 with AND3, then the last boolean argument will be deleted.

Collapsing a Script

Suppose you have this Script:

```
AND
  AreSameActor of:
    ThisSubject
    ReactingActor
  NOT
```

```
        AreSameActor of:
            ThisDirObject
            CandidateActor
```

And you want to get rid of the first "AreSameActor" term. Just click on the NOT Operator, Copy, click on AND, and Paste. Now it reads:

```
NOT
  AreSameActor of:
      ThisDirObject
      CandidateActor
```

Right-clicking

If you select an Operator and click on the right button on your mouse, then you'll see a long menu pop up next to the selected Operator, containing every single Operator that you could enter in its stead. Some of those Operators are placed inside submenus. (Mac users, you can reach the right-click/context-sensitive menu by holding down the Control key when clicking on the Operator. But the right-click is so heavily used by Swat that you really should buy a two-button mouse, such as the Apple Mighty Mouse.)

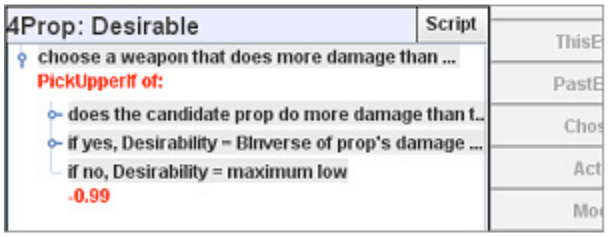
Next Tutorial: Scriptalyzer
Previous Tutorial: Spying

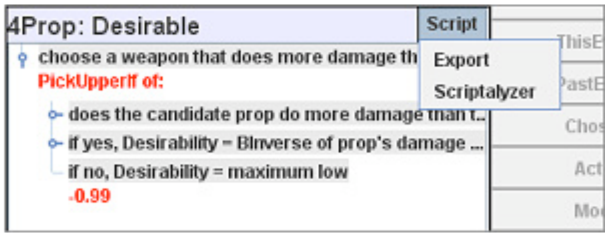
Storytron: Scriptalyzer

Last edited by Bill Maya 6 days ago

SWAT provides you with a great many tools for analyzing your storyworld and figuring out what's going on inside. One of these is Scriptalyzer. You use this tool to figure out how a Script works.

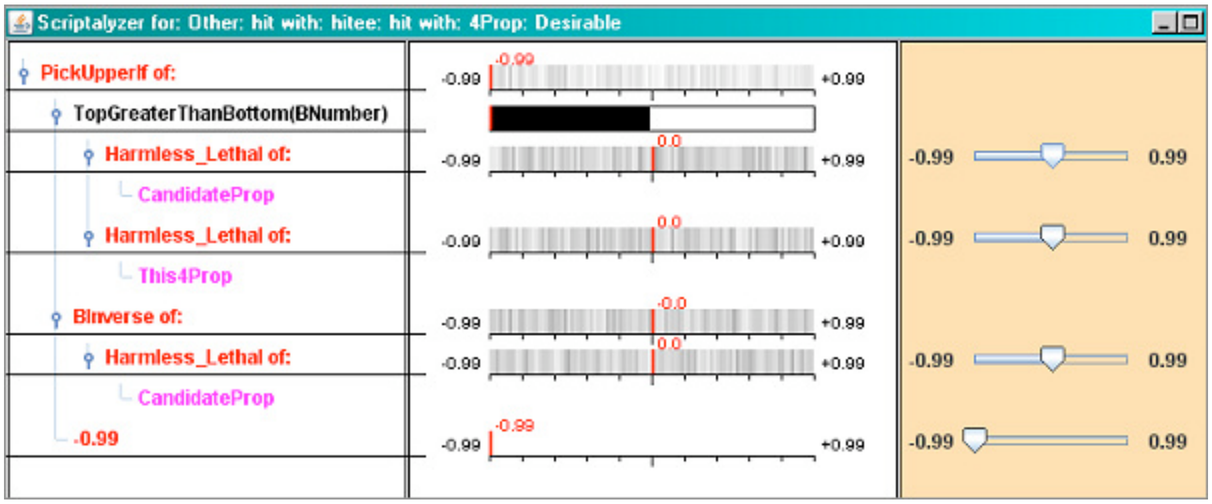
To try it out, bring up the Script for hit with: hittee: hit with: 4Prop: Desirable. T'hen click on the button in the upper right corner of the Scripting box that is labeled "Script."


4Prop: Desirable
Script
choose a weapon that does more damage than ...
PickUpperIf of:
does the candidate prop do more damage than t...
if yes, Desirability = BInverse of prop's damage ...
if no, Desirability = maximum low
-0.99


4Prop: Desirable
Script
Export
Scriptalyzer
choose a weapon that does more damage th
PickUpperIf of:
does the candidate prop do more damage than t...
if yes, Desirability = BInverse of prop's damage ...
if no, Desirability = maximum low
-0.99

You'll see a popup menu with just two menu items: "Export" and "Scriptalyzer." The Export choice is very simple: it saves an HTML file showing the Script. This is handy for printing out Scripts, or perhaps sending a Script to a friend. The Script can't be imported back into SWAT, but it can at least be read easily. Be sure to read it in an HTML display application such as a browser, not a text editor.

But the important menu item here is "Scriptalyzer." Select that menu item and you'll see a big new window appear:


Scriptalyzer for: Other: hit with: hitee: hit with: 4Prop: Desirable

PickUpperIf of:
TopGreaterThanBottom(BNumber)
Harmless_Lethal of:
CandidateProp
Harmless_Lethal of:
This4Prop
BInverse of:
Harmless_Lethal of:
CandidateProp
-0.99

Here's what it means:

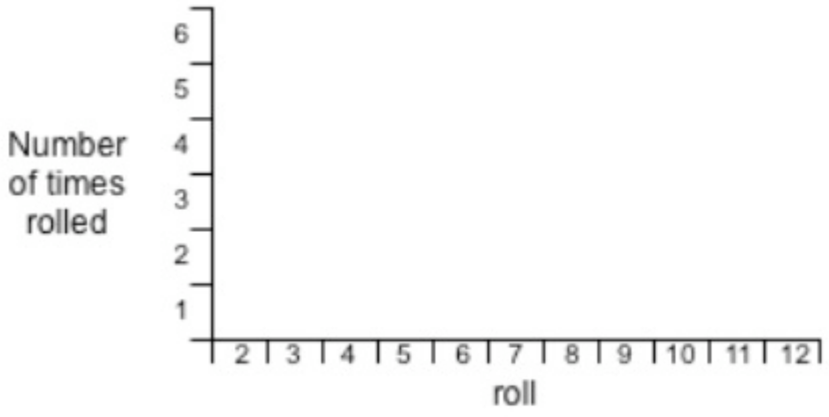
The left column shows the Script that is being analyzed. It's not interactive—it's just for reference purposes (actually, you can click on the tiny iconettes to open and close Operators, but it won't mean anything because it doesn't affect the rest of the window). The central column shows the results of the Scriptalyzer process. Those results are pretty complicated, so here goes:

The red marks show what happens when you change the slider settings in the right hand column. Change the first slider, which represents Harmless_Lethal of CandidateProp, and watch how the red marks move around in response. Notice what happens to the PickerUpperIf value as you change the Harmless_Lethal of CandidateProp. You get two different numbers, depending on whether the Harmless_Lethal of CandidateProp is greater or smaller than the Harmless_Lethal of This4Prop. That's the comparison the Script is making, and here you see it in action.

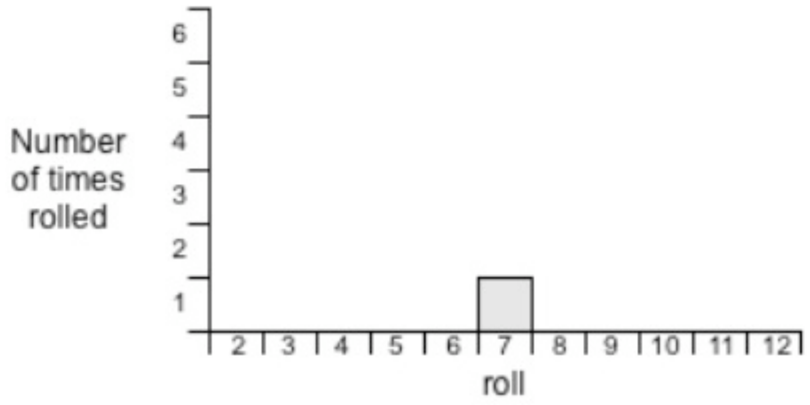
If you ever want to get a feeling for how a particular Operator works, this is an excellent place to play. The slider settings you make do not affect anything in the storyworld; they just show an analysis of the current script's effects. You can play with them as much as you want without altering your storyworld.
The gray imagery is harder to understand. Think of it as a "sideways light gray glass histogram."

Great. What's a histogram?

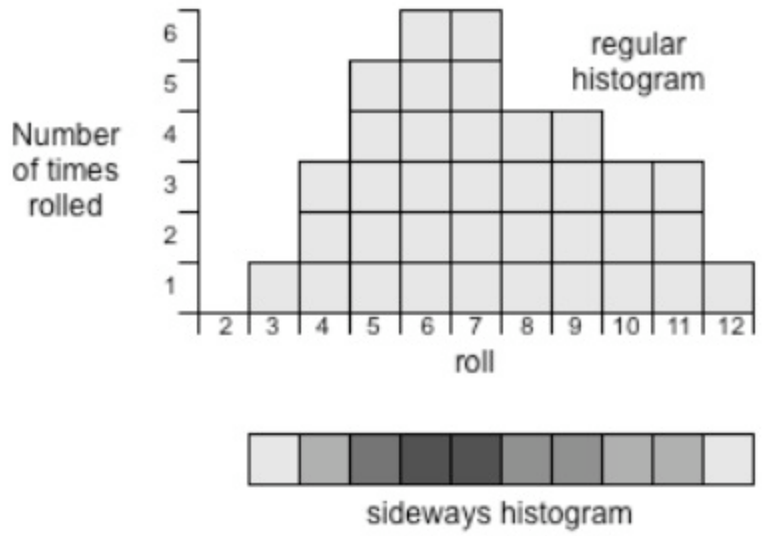
Here's a simple example of how you build a histogram. Suppose that you have a pair of dice and you roll them and note what you get. Suppose it's a 7. Instead of writing down "7," you lay out a horizontal graph like so:



and then you plunk down one square to mark the 7 that you rolled, like so:



Now do it again; say you get a 5. So put a square in the "5" space. And now repeat this experiment a bunch of times, putting down more squares, and squares on top of squares if necessary. Eventually you end up with something like this:



Now, suppose that each of those squares was really a glass cube made with light gray glass. If you were to put a light underneath the histogram and stand over it; you'd see something like the sideways histogram. Where there are a lot of cubes, you see less light (dark color). Where there are a few cubes, you see more light (lighter color).

That's what Scriptalyzer does. Those gray smears in the middle column show how the results of a thousand experiments came out.

When Scriptalyzer first opens up for a Script, it goes to each of the sliders and randomly picks a setting for that slider. The red marks all move around in response to the new settings. Scriptalyzer makes a note of where every red mark ended up, then starts over, changing the settings of all the sliders to new positions. Again, it notes where the red marks end up. Then it does this 998 more times, ending up with a big pile of data for where all the red marks ended up in all those 1,000 experiments. Then it builds a histogram out of the results.

An area that's black means that a lot of the experiments put their red marks in that area. An area that's white means that very few experiments put their red marks in that area. So you can read the glass histogram to get a quick idea of how the Script tends to come out on average.

Scriptalyzer is a very handy tool for understanding both Operators and Scripts. You can learn a lot about the individual Operators by simply playing with Scriptalyzer. You can also figure out how well your Scripts work using Scriptalyzer. If a Script is always yielding a result that's too high, say, then you can go into Scriptalyzer and wiggle around all the sliders, and pretty soon you'll see that only a few of the sliders really matter. Those are the ones that are biasing your results.

Next Tutorial: Lizards
Previous Tutorial: Script Editing Tips

Storytron: Lizards

Last edited by Bill Maya 25 minutes ago

You know how so many programs come equipped with "Wizards" that will help you automagically do all sorts of wonderful things? You know how, when you try to use those "Wizards," they are often unable to solve your problem for you, because they're actually pretty stupid? Have you ever resented the hype that calls these stupid functions "Wizards?"

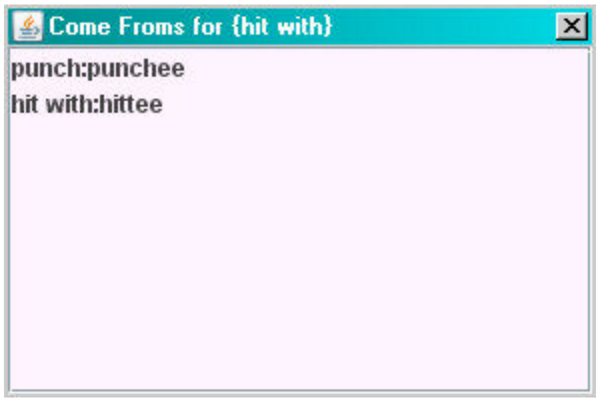
Now you know why we call ours "Lizards."

Lizards are special functions to assist the Author in creating a Storyworld. They provide you with special ways of examining your Storyworld and its performance. Scriptalyzer is a kind of lizard, but was kicked out of the Lizard Academy for being a nerd. The other Lizards are:

ComeFrom Lizard

Whenever you look at a Verb, you can readily see where it goes by just looking at the Options under the Roles for that Verb. So it's easy to see where things go—but what if you want to know where they come from? What if you want to know how somebody could have gotten to the Verb you're editing? That's what ComeFrom Lizard is designed to do. Just go to the Lizards menu and select the top item, "ComeFrom Lizard" and you'll see a new pink window pop up:

ComeFrom Lizard lists all the Verb: Role combinations that have the selected Verb as an Option. If you double-click on one of the listings, SWAT will jump to that Verb: Role. Whoosh!



punch:punchee
hit with:hittee

Notes Search Lizard

Here's a little scripting trick we haven't told you about: you can annotate your scripts. Start with any Script in your storyworld and select any Operator. Hit the "return" key on your keyboard. Look! By the magic of modern technology, a little text box appears! Of course, you don't get a lot of space in which to work;

although you are free to type as much as you want, only the first line of your text is visible when you're done. Still, it's a useful feature for two reasons:

First, you can explain what you're doing in that part of the Script. This can be very useful when you come back several weeks later and ask, "What does this do?" Think of it as a little reminder.

Second, you can use special terms that are unique to a certain type of calculation, and then later use can use them to find all the Scripts that include those special terms. This can be very handy, because after a while your storyworld gets full of thousands (we're not exaggerating: *thousands*) of Scripts and you forget what went where.

Let's try it out. Go to the hit with: hittee: hit with: 4Prop: Desirable script in our testing storyworld.

```
PickUpperIf of:
  TopGreaterThanBottom(BNumber) of:
Harmless_Lethal of:
          CandidateProp
     Harmless_Lethal of:
          This4Prop
  BInverse of:
     Harmless_Lethal of::
          CandidateProp
  -0.99
```

What the heck did this script do? Oh, yes. It chose a weapon that did more damage than the weapon the hittee was just hit with.

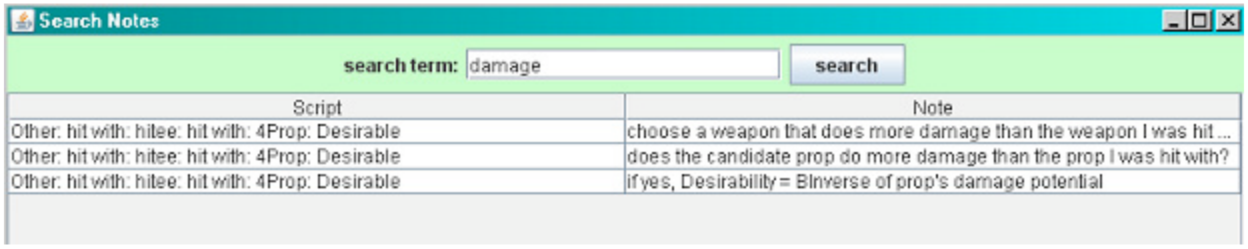
Click on PickerUpperIf of: at the top of the script, and hit return. A Notes box appears. Type in: "choose a weapon that does more damage than the weapon I was hit with."

Now click TopGreaterThanBottom(BNumber) of: and add the note, "does the candidate prop do more damage than the prop I was hit with?"

Next, add a note to BInverse of: that says "if yes, Desirability = BInverse of prop's damage potential."

Finally, click on the -0.99 in the last line and give it this note: "if no, Desirability = minimum."

Now select "Notes Lizard" from the "Lizards" menu. A wide, short window appears. Type in the keyword "damage." Notes Lizard will search through all the annotations in all the Scripts and find every Script containing that word:



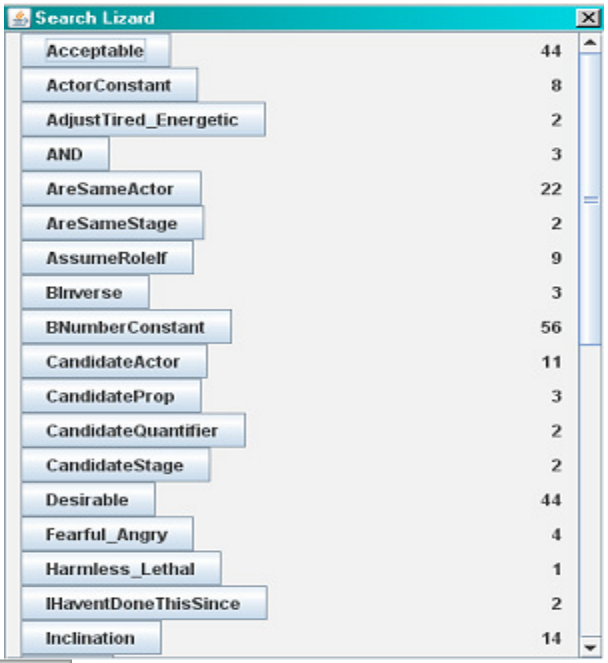
Script	Note
Other: hit with: hitee: hit with: 4Prop: Desirable	choose a weapon that does more damage than the weapon I was hit ...
Other: hit with: hitee: hit with: 4Prop: Desirable	does the candidate prop do more damage than the prop I was hit with?
Other: hit with: hitee: hit with: 4Prop: Desirable	if yes, Desirability = BInverse of prop's damage potential

If you double-click on the Script identification, the Verb Editor will jump directly to that Verb, Role, and Script (in this case they're all the same). You can use this search-notes capability by using certain keywords in your script Notes, to help you find and edit similar scripts.

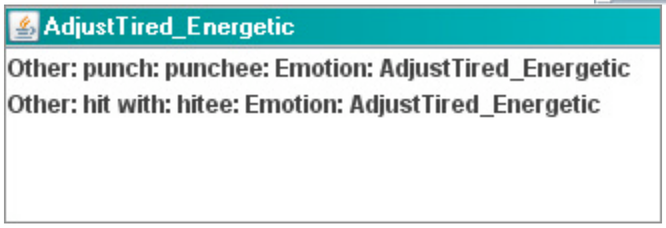
Operator Search Lizard

Suppose that you've been working on your scripts and you realize that have made a consistent mistake in the way you have used the Operator AdjustTired_Energetic. You want to correct those mistakes, but how can you find every instance of your use of AdjustTired_Energetic? Search Lizard is the lizard for you. Just select it from the Lizards menu and you'll see a window listing every single Operator you use in your Scripts, along with how many times you have used that Operator:

If you scroll down through the window, you can find AdjustTired_Energetic. Simply click on the button and you'll see a new window listing every single Script that uses AdjustTired_Energetic:



Acceptable	44
ActorConstant	8
AdjustTired_Energetic	2
AND	3
AreSameActor	22
AreSameStage	2
AssumeRoleIf	9
BInverse	3
BNumberConstant	56
CandidateActor	11
CandidateProp	3
CandidateQuantifier	2
CandidateStage	2
Desirable	44
Fearful_Angry	4
Harmless_Lethal	1
IHaventDoneThisSince	2
Inclination	14



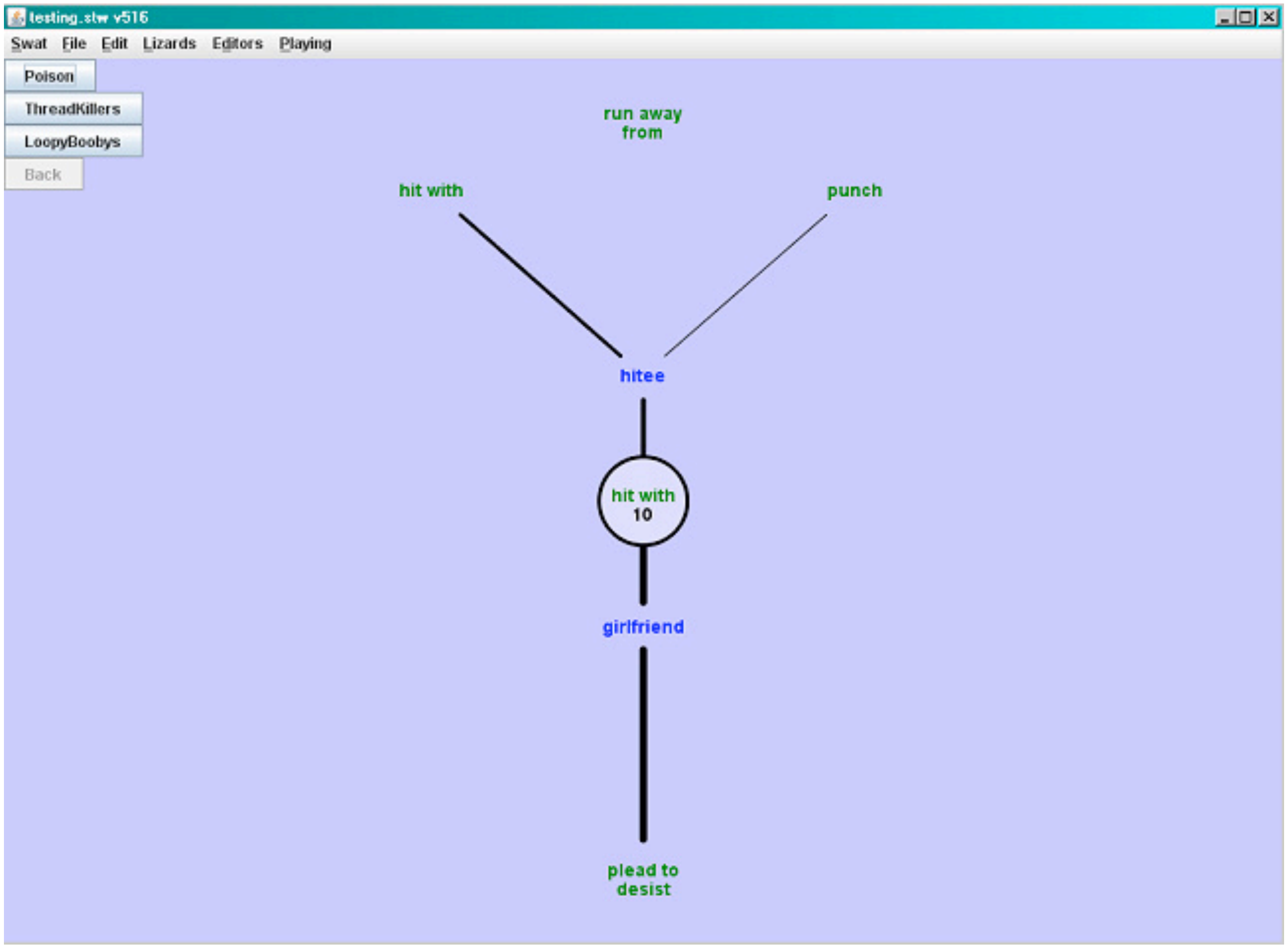
Other: punch: punchee: Emotion: AdjustTired_Energetic
Other: hit with: hitee: Emotion: AdjustTired_Energetic

Just double-click on the Script listing and the Verb Editor will jump directly to that Script so you can work with it.
Search Lizard has other uses. At the very top of the Operator listing will be any "undefined element" Operators. These all begin and end with question marks. These are the prompt Operators that are automatically inserted into a Script when you add an Operator. You are supposed to fill them in with normal Operators, but sometimes we overlook these things. When the Engine tries to run them, it creates Poison, which kills that part of the story (see Poison for further information). Therefore, you can use Search Lizard to locate any of these incomplete Operators and fill them in with the proper values.

Rehearsal Lizard

In creating a storyworld, you often set up *clusters* of Verbs that link to each other. It's hard to know from looking at the Inclination Scripts just how often Verb A leads to Verb B. You could create a rich, dense cluster with all sorts of interesting possibilities, but in practice you might see all that richness ignored and the storytrace always traversing the same path through the cluster. How can you find out whether this happens? Turn to Rehearsal Lizard, and your problem will be solved.

To use Rehearsal Lizard, you first select and jump to the first Verb in the cluster, the one that initiates the action. Then select Rehearsal Lizard. He'll show you a new window:



In the example above, the starting Verb was "hit with." That Verb was executed 10 times. It has two Roles (hittee and girlfriend). The black lines indicate how many times each of the Roles was activated. In this example, "hittee" was activated four times and "girlfriend" was activated the rest of the time.

From the Roles we branch out to the Options. The "hittee" Role leads to two Options (hit with and punch), or which hit with was chosen three times and punch was chosen once. For the "girlfriend" Role, the only Option possible was plead to desist, which was chosen six times.

Roles are drawn in blue (they represent Actors) and Options are drawn in green (they represent Verbs). The number of occurrences of the central Verb is presented inside its circle. The width of the line indicates the number of times a Role was assumed or an Option chosen. Clicking on an Option jumps to a new display showing that Option in the central position with its results.

The buttons in the upper left corner provide details on some of the common problems with storyworlds:

Poison

Lists all instances of Poison and which Script generated it.

ThreadKillers

Lists all Verbs whose execution failed to generate a reaction, killing that thread.

Loopy-Boobies

Lists all Verb sequences in which Actors got caught in a loop.

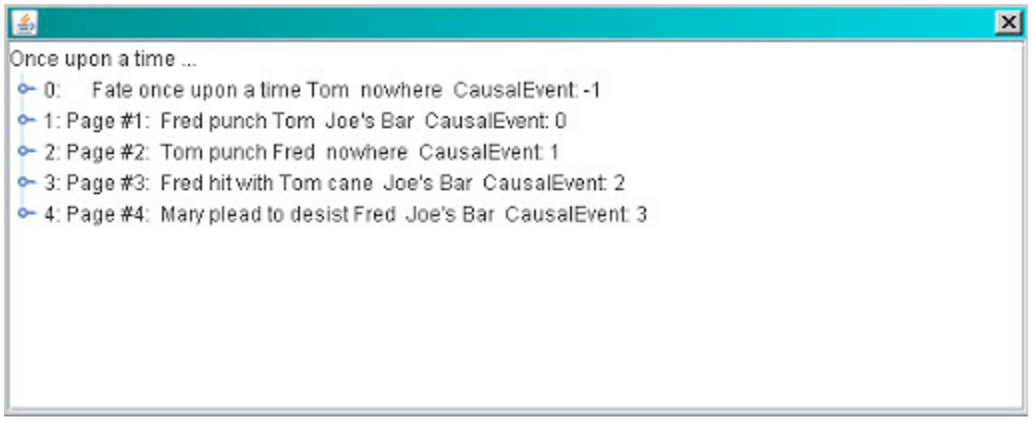
Storyteller Lizard

This Lizard runs the Storyteller package inside SWAT. Storyteller is the software the player uses to experience your storyworld. The Storyteller Lizard allows you to make test runs of your storyworld without having to leave the storyworld development environment.

Log Lizard

This is the single most powerful analytical tool for understanding the operation of storyworlds. Every time the Engine makes a critical decision, including every single Operator of every Script, it logs its decision and the basis for making that decision. This allows you to review the Events that took place during a storyworld and figure out why things happened the way they did.

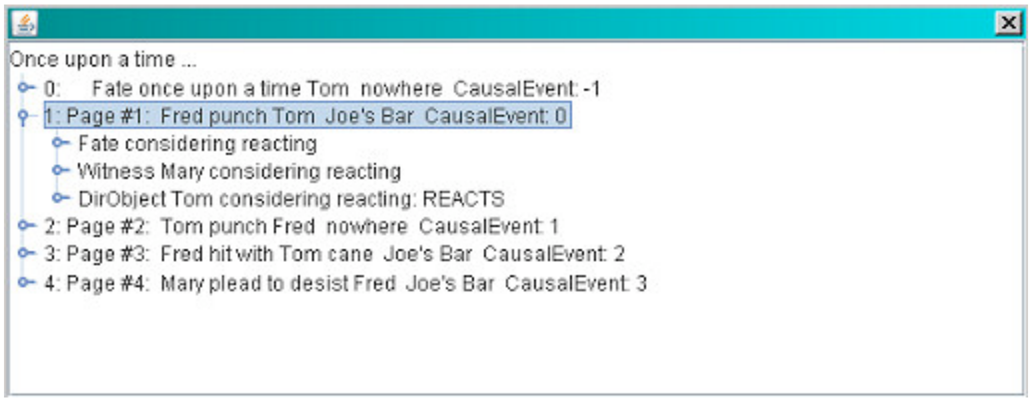
The amount of information generated by the logging is enormous. This takes a lot of memory and slows down the CPU, so we urge you to keep Storyteller Lizard sessions to less than a thousand Events. Moreover, the amount of information that Log Lizard generates is humongous, so we present it to you in an organized fashion that makes it easier to find what you're looking for:



```
Once upon a time ...
 0:     Fate once upon a time Tom  nowhere  CausalEvent: -1
 1: Page #1:  Fred punch Tom  Joe's Bar  CausalEvent: 0
 2: Page #2:  Tom punch Fred  nowhere  CausalEvent: 1
 3: Page #3:  Fred hit with Tom cane  Joe's Bar  CausalEvent: 2
 4: Page #4:  Mary plead to desist Fred  Joe's Bar  CausalEvent: 3
```

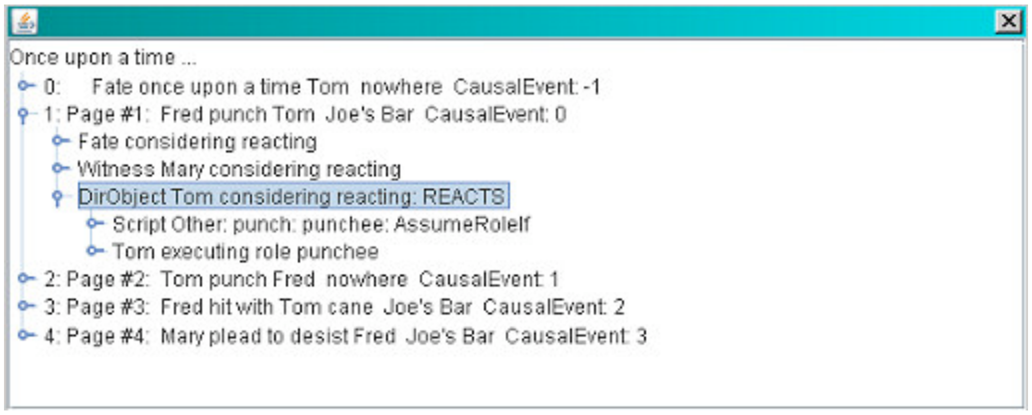
This is the basic Log Lizard window. Five Events took place during the storyworld;

each Event has its own "Page" in the HistoryBook. The time at which the Event took place is listed along the left edge. The Event itself is presented in abbreviated form, followed by the name of the Stage on which the Event took place and the page number of the Event that caused this Event to take place. For example, at time 2 and page 2, TompunchedFred. This took place on the Stage called "Joe's Bar" and was a response to Event 1, when Fredpunched Tom. Let's analyze how that happened. We do so by clicking on the sideways lollipop icon by Event 1, on the extreme left edge:



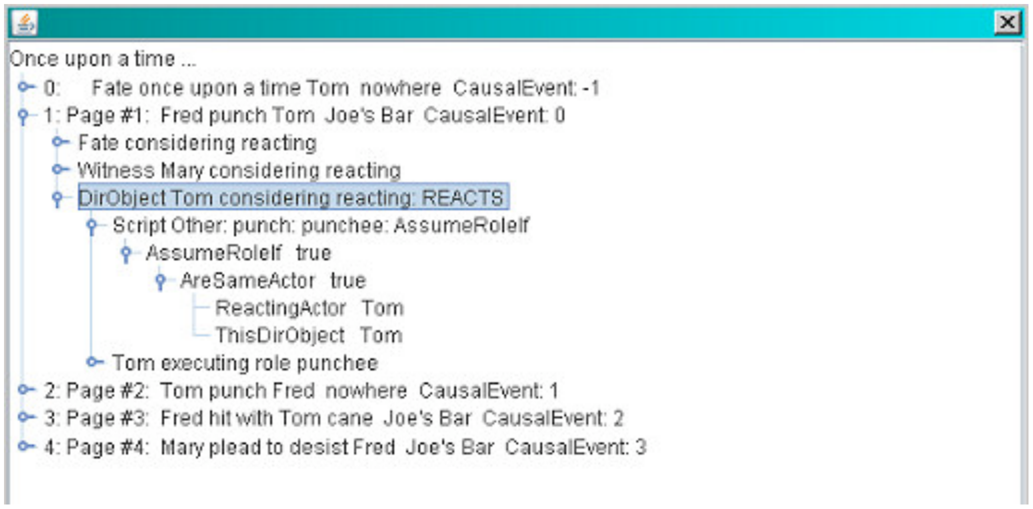
```
Once upon a time ...
  0:     Fate once upon a time Tom  nowhere  CausalEvent: -1
  1: Page #1:  Fred punch Tom  Joe's Bar  CausalEvent: 0
     Fate considering reacting
     Witness Mary considering reacting
     DirObject Tom considering reacting: REACTS
  2: Page #2:  Tom punch Fred  nowhere  CausalEvent: 1
  3: Page #3:  Fred hit with Tom cane  Joe's Bar  CausalEvent: 2
  4: Page #4:  Mary plead to desist Fred  Joe's Bar  CausalEvent: 3
```

This expands the "node" for Page 1, so we can see how each Actor reacted to Fred's action. You can see that both Mary and Tom considered reacting, but only Tom actually did react. Let's examine that process by clicking on the lollipop on the left edge of Tom's line:



```
Once upon a time ...
  0:     Fate once upon a time Tom  nowhere  CausalEvent: -1
  1: Page #1:  Fred punch Tom  Joe's Bar  CausalEvent: 0
     Fate considering reacting
     Witness Mary considering reacting
     DirObject Tom considering reacting: REACTS
        Script Other: punch: punchee: AssumeRoleIf
        Tom executing role punchee
  2: Page #2:  Tom punch Fred  nowhere  CausalEvent: 1
  3: Page #3:  Fred hit with Tom cane  Joe's Bar  CausalEvent: 2
  4: Page #4:  Mary plead to desist Fred  Joe's Bar  CausalEvent: 3
```

This doesn't add much new information; let's examine the first item more closely. This time we'll double-click on the line starting with "Script Other," which opens up everything underneath that node. (This way we don't have to single-step opening the whole thing up):



```
Once upon a time ...
0:    Fate once upon a time Tom  nowhere  CausalEvent: -1
1: Page #1:  Fred punch Tom  Joe's Bar  CausalEvent: 0
    Fate considering reacting
    Witness Mary considering reacting
    DirObject Tom considering reacting: REACTS
        Script Other: punch: punchee: AssumeRoleIf
            AssumeRoleIf  true
                AreSameActor  true
                    ReactingActor  Tom
                    ThisDirObject  Tom
        Tom executing role punchee
2: Page #2:  Tom punch Fred  nowhere  CausalEvent: 1
3: Page #3:  Fred hit with Tom cane  Joe's Bar  CausalEvent: 2
4: Page #4:  Mary plead to desist Fred  Joe's Bar  CausalEvent: 3
```

This is the AssumeRoleIf Script that determines whether an Actor assumes a Role. You can see that it generated an AssumeRoleIf value of true—it decided that yes, it would assume that Role. How did it decide that? Because the Operator underneath it (AreSameActor) had a value of true. And how did AreSameActor get a value of true? Because when it compared ReactingActor (whose value is Tom) with ThisDirObject (whose value is also Tom), it found that the two Actors are in fact the same Actor: Tom!

You can learn more about the operation of the Engine by digging into some of the other Events. It's all laid out there for you in complete detail. If ever you have a problem, you can see exactly how it happened with Log Lizard.

Next Tutorial: Operators
Previous Tutorial: Scriptalyzer

Storytron: Operators

Last edited by Bill Maya 6 days ago

You can think about Operators in terms of language. Consider this sentence:

"The farmer gave the dog that saved the life of his son a big slab of meat from the pig he had just slaughtered."

OK, it's a clumsy sentence with too many subsidiary components; a good writer would break it down into several sentences. But it breaks down into words in exactly the same way that a Script breaks down into Operators:

```
gave
    who: The farmer
    to whom: the dog
        that saved
            what: the life
            of whom: his son
    what: the slab
        of meat
            from the pig
                that slaughtered
                  who: he
                  when: had just
```

Granted, this is a rather odd way of breaking apart a sentence, but its utility arises from its defining the components of the sentence in a precise fashion. And when you're talking to computers, you have to be precise.

This nesting capability of language gives it the power to express any idea, no matter how complex. And the scripting language of Storytronics uses exactly the same nesting concept for its Operators.

Operator Types

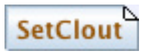
An Operator produces a value: a number, a Prop, an Actor, etc. There are three types of Operators:

- Getters
- Setters
- Crunchers

Getters are simple: they simply look up a stored value and return it. Here is an example of a Getter:

Owner of:
  ThisProp

When you use the Operator Owner of in a script, SWAT gets the name of the Actor who owns the prop that has just been used in ThisEvent, and plugs it into that slot in the script.
Setters store a new value into a variable. They are represented by Script Buttons. The Verb's Consequences, AssumeRoleIf, EmotionalReaction, WordSocket, and Inclination scripts are all Setters. The way you can recognize a Setter is by the bent corner on the button.



Crunchers perform some calculation using the values you feed them. Here is an example of a Cruncher:

BSum of:
  Quiet_Chatty of:
    ReactingActor
  Fear_Anger of:
    ReactingActor

The Operator BSum takes the number representing how talkative ReactingActor is (that is, her Quiet_Chatty value), and adds to it the value representing how fearful or angry she is. This Operator might be used to determine how likely an Actor is to speak her mind in a confrontation, for instance. The more angry she is, and/or the more outspoken she is, the more likely she will be to argue or confront.

Nesting of Operators

Many Operators have arguments. "Arguments" is just a fancy name for subsidiary Operators that provide further specifics on what the Operator is required to do. Some Operators have just one argument, like these:

Quiet_Chatty of: how loquacious ReactingActor is
  ReactingActor

(We indicate nesting by insetting the argument.)

Other Operators have two arguments, like these:

PQuiet_Chatty of: how loquacious ThisSubject perceives
ThisDirObject to be

ThisSubject
ThisDirObject

Here's an example of nesting arguments:

```
Quiet_Chatty of:
    Owner of:
        ThisDirProp
```

Some Operators have no arguments at all; here are a few:

BNumberConstant (a number like 0.0 or 0.5)
ThisSubject (the Subject of the Event that just happened)
ReactingActor (the Actor reacting to the Event that just happened)

Next Tutorial: Special Operators
Previous Tutorial: Lizards

Storytron: Special Operators

Last edited by Bill Maya 6 days ago

We provide a bunch of special Operators in Sappho to make your life easier. Here are their definitions:

This___, Past___, Chosen___

These are large groups of operators that apply to specific WordSockets (as well as some other parts) of the Event. There's an Operator called ThisSubject, which returns the Subject of the Event that has just taken place. ThisDirObject refers to the DirObject of the selfsame Event. And so on through all the other WordSockets (e.g., This4Actor, This5Stage, etc.). But there are even more Operators of this sort:

ThisTime: the time at which this Event took place.
ThisLocation: the Stage on which this Event took place.
ThisPageNumber: the page number of this Event.
ThisCausalEvent: the Event that caused this Event to happen.

ThisHijacked: whether or not this Event has been hijacked.

We also have Past___ versions of all the This___ Operators, covering exactly the same material. The only difference is that the Past___ Operator takes an Event as its argument. You have to specify the past Event that you are referring to, like so:

PastSubject of:
  ThisCausalEvent

This Operator returns the Subject of the Event that caused ThisEvent.

Notice what that tells you about ThisCausalEvent. If ThisEvent is the event that has just happened, ThisCausalEvent is the Event immediately before that. Let's take an example of a set of interactions:

1. Fred punches Joe
2. Joe passes out
3. Mary calls the police

So, if you are writing scripts to tell the Engine what it should do next for the Verb calls, ThisEvent would be Event #3 above, "Mary calls the police," which means ThisSubject is Mary, and ThisDirObject is the police. ThisCausalEvent would be Event #2, "Joe passes out." The PastSubject of ThisCausalEvent would be Joe.

Here's a challenge: Assume that ThisEvent is Event #3. What is the value of:

```
PastSubject of:
    PastCausalEvent of:
        ThisCausalEvent
```

(Answer: Fred)

This is a good example of a simple situation. However, in more complicated storyworlds, there can be simultaneous activity in different Stages, so it is not necessarily true that ThisCausalEvent immediately precedes ThisEvent; there could be other Events from other Stagesbetween these two Events.

Finally, there are the Chosen___ Operators. These are a little tricky. Their purpose is to let you use decisions that you make in one WordSocket, in a later WordSocket. They store the values of the WordSockets that you have already decided upon in your Option scripts. You can't use these Operators in any place except the Inclination or WordSocket Acceptable and Desirable scripts inside any Option. SWAT won't let you use the Chosen___ Operators improperly. And SWAT will only let you use Chosen___ Operators for WordSockets that precede the WordSocket in which you're working. So if you're writing a Script for the WordSocket5Prop then you can use ChosenDirObject or Chosen4Stage (if they exist), but you couldn't use Chosen9Actor, because it hasn't been decided as of the time of execution of the Script you're working on.

For these purposes, Inclination is the last Script executed, so you can use any of the Chosen___ Operators in the Inclination Script.

Blending Operators

Blending Operators allow you to mix numbers together in differing ratios. They serve a number of useful purposes. Suppose, for example, that you want to write an Inclination Script for Joe's inclination to punch Fred, based on two factors: how angry Joe is, and how much Joe likes Fred. You want to mix the two factors together. Here's one way to write your Script (we're assuming here that PNasty_Nice measures how much the perceiver likes the perceived):

```
Blend of:
      Fear_Anger
          Joe
      PNasty_Nice
          Joe
          Fred
      0.0
```

This means "blend Joe's Fear_Anger in equal measure with Joe's PNasty_Nice for

Fred." Blend will return the average of the two BNumbers. But here's the really nice part. Suppose that, when you're testing your storyworld, it always seems as if the Fear_Anger part plays too big a role in the final result. Once your Actors get mad, it doesn't seem to matter how much they like a person—they start punching. You want to tone it down and put more emphasis on PNasty_Nice. No problem! Just change that final 0.0to something a bit more negative. In Blend, the final term, called the "bias factor," biases the result towards one or the other argument. A positive bias factor will put more weight on the upper argument; a negative bias factor will put more weight on the lower argument. So if you put in, say, -0.4, then the PNasty_Nice part will get more weight in Blend than the Fear_Anger part. This feature of the Blend Operator makes it especially useful for fine-tuning your storyworld.

BlendBothily

This is another blending Operator that operates in a different fashion. We won't go into the mathematical subtleties involved; We'll just give you a simple rule of thumb: use Blend most of the time. Use BlendBothily only when you want either of the two arguments to stomp out the result if it's -0.99. Here's an example of what we mean:

```
Blend of:
        +0.99
        -0.99
         0.0
```

this returns a value of 0.0—the average of the two. But look at this:

```
BlendBothily of:
        +0.99
        -0.99
         0.0
```

this returns a value of -0.99 (actually, it returns -0.86, but that's even trickier to explain).

BlendBothily doesn't give the average of the two values. In other words, BlendBothily would be only useful in cases where both factors you are considering must be on the high end of the scale. For instance, let's assume you want to create a script for an Option called beg for help, which the ReactingActor will only resort to if both of these conditions are met:

1) they have a high degree of trust in the beggee
2) they are very frightened about something

In such a case, the inclination script might look like this:

```
BlendBothily of:
  PFalse_Honest of:
      ReactingActor
      ChosenDirObject
  BInverse of:
      Fearful_Angry of:
          ReactingActor
  0.0
```

This script says that the ReactingActormust have both a strong perception that the ChosenDirObject is trustworthy (that is, they perceive them to have a very high False_Honest value), and the ReactingActormust also be very afraid. If both are not very high numbers (far over on the right side of the number scale), then the Option beg for help will not be chosen.

Poly-argument Operators

A few special Operators normally take just two arguments, but in some cases you'd like to include more arguments. For example, consider BSum. It adds together two BNumbers. But suppose that you wanted to add three numbers together? Then you'd have to write something like this:

```
BSum of:
      BSum of:
            First BNumber
            Second BNumber
      ThirdBNumber
```

This is clumsy, so we created two new summing Operators: BSum3, which adds three BNumbers, and BSum4, which adds four BNumbers. We also created similar Operators for logic: OR3, OR4, AND3, and AND4.

There's also a Blend3 and a Blend4. They differ from the regular Blend in that, instead of having a single bias factor, they have separate weighting factors for each argument. Here's an example:

```
Blend3 of:
  Fear_Anger
      Joe
  0.5
  PNasty_Nice
      Joe
```

```
    Fred
0.0
False_Honest
    Mary
-0.5
```

This will blend all three factors together, but Joe's Fear_Anger will get a lot of weight, his PNasty_Nice will get less, and Mary's False_Honest will get very little weight.

Next Tutorial: HistoryBook Operators
Previous Tutorial: Operators

Storytron: HistoryBook Operators

Last edited by Bill Maya 6 days ago

The HistoryBook is an especially useful feature; it's a record of every Event that has taken place in the play of your storyworld. Think of it as "the history of the storyworld so far." You can look up Events in the HistoryBook to recall exactly what happened. Here are the Operators you can use:

EventHappened

This is a boolean Operator; it returns a simple yes-or-no answer to the question "Did the Event fitting this description ever take place?" It takes a single argument, another boolean, that provides the description in proper form. Here's an example:

```
EventHappened
  AND
      AreSameActor
          ReactingActor
          PastSubject of:
              CandidateEvent
      AreSameVerb
          punch
          PastVerb of:
              CandidateEvent
```

This means, *Has any Event ever happened in which the Subject was the ReactingActor and the Verb was "punch?"* You may be a little confused by the use of "CandidateEvent." Remember, when we ask the Engine to look for "any Event," we are telling it to look at each and every Event to make a decision; that decision has to consider each Event individually. "CandidateEvent" is that "each Event individually." So we could expand the terse description above into the following more specific version:

Engine, I want you to examine each and every Event in the whole HistoryBook. When you look at each Event, we'll call that Event you're looking at "CandidateEvent." Now, I want you to get the Subject and Verb of that CandidateEvent. If the Subject of that CandidateEvent is the ReactingActor, and the Verb is "punch," then return "true."

We also provide a handy little helper Operator: MainClauseIs. We learned in practice that many of our EventHappened Operators ended up looking like this:

```
EventHappened
  AND3
      AreSameActor
```

```
            ReactingActor
            PastSubject of:
                CandidateEvent
        AreSameVerb
            punch
            PastVerb of:
                CandidateEvent
        AreSameActor
            ThisDirObject
            PastDirObject of:
                CandidateEvent
```

Over and over again, we found ourselves specifying the Subject, Verb, and DirObject of the PastEvent. This became rather tiresome, so we came up with this handy-dandy little shortcut:

```
EventHappened
   MainClauseIs
       ReactingActor
       punch
       ThisDirObject
```

This means exactly the same thing as the previous version, but it's a lot easier to use.

Why would you use EventHappened? Normally you use it to check whether some Event has taken place that would be required for a later choice to be made. For example, suppose that you don't want the Prince to be able to rescue the Princess until *after* the Dragon has been slain? Then you might have an Inclination Script for the Optionrescue looking something like this:

```
PickUpperIf
   EventHappened
       MainClauseIs
           Prince
           slay
           Dragon
   Maxi
   Mini
```

LookupEvent

This is what you use to answer questions such as "Is this the same Prop that the Subject used to hit the ReactingActor previously?" Here is how it is typically used:

```
AreSameProp
  This4Prop
  Past4Prop of:
      LookupEvent of:
          AND
              MainClauseIs
                  ThisSubject
                  hit with
                  ReactingActor
              AreSameProp
                  This4Prop
                  Past4Prop of:
                      CandidateEvent
```

LookupEvent is found under the History Operators menu.

Here's a fine point about LookupEvent: it searches backwards from the present and stops at the first Event it finds that meets the specifications. This means that it will find the *most recent* Event that meets the specs.

CountEvents

This Operator answers the question "How many times has this happened before?" For example, you might want to have Charlie Brown ask himself, "How many times has Lucyyanked the football away when I ran to kick it?" Presumably this would be used in a Script like this:

Offer to hold football: CharlieBrown: refuse to kick football: Inclination

```
Number2BNumber of:
  quotient of:
      CountEvents of:
          MainClauseIs
              Lucy
              yank football
              Charlie Brown
      100
```

There are several important things to notice about this script.

First, note that the entire clause starts off with the Operator Number2BNumber. You might ask why we need this. If you'll recall from our tutorial on Attributes, all traits are BNumbers, and range from -1.0 to +1.0. Again, we do this to be sure we are always comparing apples to apples, and making it easier to keep our scripts contained within a similar range of values. But a storyworld's Events don't range between -1 and +1. They range from 0 on up to 100, 1,000 or more. To use CountEvents (or any regular number) in a script, we must first convert it to a BNumber.

Second, note that we divide CountEvents by 100, using the Operator quotient. This is basically saying that Charlie Brown is in fact capable of learning from his mistakes, albeit rather slowly. If we wanted to make him a quicker learner, we'd do this by making the 100 a smaller number—say, 10—or even eliminate the quotient altogether, and just have Number2BNumber of: CountEvents. Charlie Brown's inclination to refuse to kick the football would increase a good deal more rapidly in that case.

"Causal" tests

There is also a corresponding set of Operators that use a slightly different way of searching the HistoryBook. All the Operators above start at the current Event and work backwards in time. "Causal" tests (CausalEventHappened, LookupCausalEvent, CountCausalEvents) only follow the chain of causality backwards. These tests ignore unrelated Events and look only at those Events that are directly in the chain of causality leading to the Event being reacted to. This is a more precisely targeted test that is necessary when you want to make sure that you're not fooled by an Event that meets your specs but is, by some strange chance, unrelated to the current Event.

ElapsedTimeSince

This is another rarely-used Operator; you use it to find out how much time has passed since an Event took place. Example:

Offer donut: DirObject: accept donut: Inclination

```
Number2BNumber of:
  ElapsedTimeSince
      AND
          AreSameActor
              ReactingActor
              PastSubject of:
                  CandidateEvent
          AreSameVerb
              eat donut
              PastVerb of:
                  CandidateEvent
```

The ReactingActor's Inclination to accept the donut is proportional to how much time has passed since he last ate a donut. (Notice that when counting time, as with counting Events, we have to convert the regular number to a BNumber in order to use it in the script.)

IHaventDoneThisBefore

This is a particularly useful Operator that is meant to obviate repetitious behavior. What's neat about it is that it's so smart. You can bury it inside a WordSocket and it will look for matches right up to and including that WordSocket, but it will ignore anything beyond it. In other words, you don't have to specify the contents of the WordSockets; it automatically fills them in for you. That's probably confusing, so here's an example. Suppose that you are having a conversation with another Actor about a third party. You've been comparing notes about the various Attributes of that third party. You don't want to ask about an Attribute that you've previously asked about. You could do this with LookUpCausalEvent, but there's an easier way:

agree to talk: DirObject: gossip about: ActorAttribute: Acceptable

IHaventDoneThisBefore

That's all it takes! This Operator can only be used within an Option, so you will notice that it does not appear in the History menu under, for instance, AssumeRoleIf or Emotional Reaction scripts.

IHaventDoneThisSince

This is a variation on IHaventDoneThisBefore, but it adds a backwards time limit. It means "I haven't done this in the last X moments." As with IHaventDoneThisBefore, this Operator can only be used within an Option.

Next Tutorial: More Special Operators
Previous Tutorial: Special Operators

Storytron: More Special Operators

Last edited by Bill Maya 6 days ago

Bigger, Smaller

These are simple Operators; they simply return the bigger or smaller of two BNumbers.

PickUpperIf

This Operator has three arguments: a Boolean to decide whether or not to pick the upper BNumber, an upper BNumber, and a lower BNumber. Here's a quick example:

```
PickUpperIf of:
  TopGreaterThanBottom(BNumber)
      Nasty_Nice of:
          ThisSubject
      Nasty_Nice of:
          ThisDirObject
  0.5
  0.0
```

This will return a 0.5 if ThisSubject is nicer than ThisDirObject; otherwise, it will return 0.0. This is handy for situations where you must make an abrupt choice between two possibilities based on some yes-or-no factor. It can be used in cases where the author wants one Option to override all the others under a special circumstance. An example of such an Inclination Script might look like this:

```
PickUpperIf of:
  SpecialCircumstance
  Maxi
  Mini
```

(Maxi and Mini are BNumber constants. Maxi = 0.999 and Mini = -0.999)

This would insure that the Option was a shoo-in when SpecialCircumstance was true, but otherwise was out of the running.

Actor@Sum

You can use this Operator to add up the Attribute (or perceived Attribute) values of all the Actors towards one Actor. For example, suppose you want to find out how trusted Joe is by the female Actors.

First, go to the Actor Editor and click the far right green plus sign to add a trait. Name the new trait Faithless_Honest. Unclick the check box to the right of the trait. This means the trait is not readily detectable when two Actors meet (an example of a visible/detectable trait would be Short_Tall. An example of an invisible or non-detectable trait would be Cowardly_Courageous). Then return to the Verb editor and choose any Desirable or Inclination script.

Now highlight a BNumber term and then select Actor@Sum from the Actor menu, and fill out the other terms to get this:

```
Actor@Sum of:
  Female of:
      CandidateActor
          PFaithless_Honest of:
      CandidateActor
          Joe
```

This would add up all the female Actors' PFaithless_Honest values towards Joe. Note that the first term, the Boolean, allows you to filter out Actors based on whatever acceptability criteria you choose.

Prop@Sum, Stage@Sum, Event@Sum

These Operators function for Props, Stages, and Events, respectively, in the same way that Actor@Sum does for Actors.

Actor@Average, Prop@Average, Stage@Average, Event@Average

These four Operators behave the same way as their _____@Sum counterparts, except that they return the average values of the given Attributes, rather than their totals.

Actor@Tally, Prop@Tally, Stage@Tally, Event@Tally

These Operators use regular numbers, rather than BNumbers. They count up the number of Actors, Props, Stages, or Events that meet a set of criteria you specify. You can then convert them to BNumbers and use them in your scripts.

Suppose you want to have the ReactingActor make a decision on whether to confront a bully or defer the confrontation based on how many allies he or she has. The inclination script might look something like this:

```
Number2BNumber of:
  Actor@Tally of:
      TopGreaterThanBottom(BNumber)
          PEnemy_Ally of:
              CandidateActor
```

```
            ReactingActor
        0.2
```

Actor@Tally (we pronounce these operators, by the way, as for instance "ActorTally"—that is, with a silent "@"—reduces the tongue tangles) in the above script counts the number of Actors whose perception of how allied the ReactingActor is to them (that is, their PEnemy_Ally value for ReactingActor) is at least medium-large (i.e., greater than 0.2), and gives you a numerical count from 0 to the total number of Actors in your storyworld (let's call it 4).

Using the Operator Number2BNumber converts this result to a BNumber (all Desirable and Inclination scripts must result in a BNumber). Applying the BNumber magic, 4 allies translates into a willingness to confront the bully of 0.8. Since the BNumber range is about -0.9999 to +0.9999, if you have four allies, 0.8 is pretty high on the scale. A confrontation with the bully is a pretty likely scenario!

You might want to tune this script to make it less sensitive to the number of allies. Let's say you want to make it so your Actor will only confront the bully if he or she has lots of allies. To do so, you can divide the Actor@Tally by, say, 10.0. It would look like this:

```
Number2BNumber of:
   quotient of:
      Actor@Tally of:
          TopGreaterThanBottom(BNumber)
              PEnemy_Ally of:
                  CandidateActor
      ReactingActor
              0.2
          10.0
```

It would take a lot more Allies to make ReactingActor confront the bully. If ReactingActor has four allies, this script ends up with 0.29 for Inclination. This script makes a confrontation less likely than the first script above.

Play around with different values for the divisor, and watch what happens in Scriptalyzer to the likelihood of this Option being taken (For extra credit, see if you can make the script take into account to the Actor's personal courage. Hint: the ReactingActor's courage affects how many allies he or she will want, in order to feel safe).

PickBest___

This is one of the most powerful Operators in Sappho. It will pick the best Actor, Prop, Stage, Verb, Event, ActorTrait, PropTrait, StageTrait, MoodTrait, or Quantifier depending upon your specifications. And what are your specifications? Nothing more than the same Acceptable and Desirable subscripts that you use for WordSockets!

Here's an example:

Suppose we want to pick the Prop that ReactingActor owns that is most lethal. This would be good if ReactingActor is preparing for a fight.
In the Desirable script choose CorrespondingPropTrait, and then for the PropTrait, select PickBestProp under Picking in the Operator menu. Here's how the script looks:

```
CorrespondingPropTrait of:
  Candidate Prop
      PickBestPropTrait of:
            AreSameActor
                ReactingActor
                Owner of:
                    CandidateProp
            Harmless_Lethal of:
                CandidateProp
```

There will be times when your PickBest___ Operator fails to find anything at all. For example, in the above script, if the ReactingActor doesn't own any Props at all, then PickBestProp will not find anything to return. In this case, PickBestProp will generate Poison that will eliminate the Option from consideration.

Next Tutorial: Who's Fate (part 1)
Previous Tutorial: HistoryBook Operators

Storytron: Who's Fate (part 1)

Last edited by Bill Maya 6 days ago

Fate is the most important Actor in every storyworld. In fact, Fate is so important that you can't delete her—she's a permanent fixture.

You know how, in so many movies, the bad guy gets the better of the good guy, who's now hanging by his fingernails at the edge of the cliff, and the bad guy laughs demonically and lifts his foot to start mashing the good guy's fingers, when suddenly a bolt of lightning hits the bad guy and he falls over the cliff and disappears screaming? Have you ever wondered, who killed the bad guy? The answer, of course, is Fate. Fate is the one who makes things happen in every story. In Storytronics, nothing ever "just happens"—Fate makes it happen. No Event can exist without a Subject, and Fate is always the Subject of those Events that aren't executed by any other Actor. Fate is a very powerful and useful Actor.

More precisely, Fate is *you.* You're the author after all. You're the one who makes things happen. You are the god who controls the universe of the storyworld. Fateis your avatar—your *deus ex machina*. Flex your muscles.

Here are some of Fate's unique abilities. Unlike other Actors:

- Fate is everywhere in the storyworld* at once.
- Fate knows everything that happens.

These special characteristics allow you to put Fate to work in some very useful ways. You can take note of the storyworld's state, or monitor its progress, based on a set of specifications. Based on those specifications, you can have Fate trigger an Event during storyplay, such as an ending, an "act of God," or a major new Verb cluster. The sections below give you some examples.

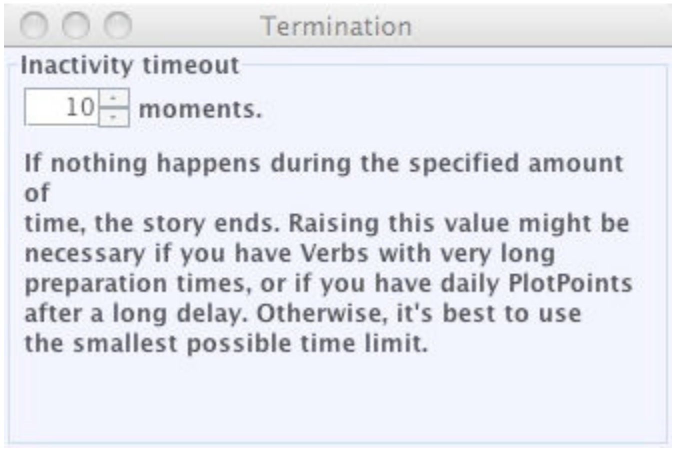
Ending a Storyworld with Timeout

Since ending a storyworld is part of Fate's purview, let's digress and talk more generally about how a storyworld ends. As you discovered when you created the walkthrough storyworld, it is not strictly necessary to take any action for your storyworld to end, as it will timeout automatically when nothing happens for a set period of time.

The default setting for the inactivity timeout is ten moments**. That is, if no one does anything for ten moments, Fate automatically triggers the penultimateVerb, which leads directly to the ending Verb, happily ever (penultimateVerb and happily ever after are System Verbs).

To view and change the timeout setting, click on the Playing menu and choose

Termination. You will see a popup like this:



Inactivity timeout

10 moments.

If nothing happens during the specified amount of
time, the story ends. Raising this value might be necessary if you have Verbs with very long preparation times, or if you have daily PlotPoints after a long delay. Otherwise, it's best to use the smallest possible time limit.

Use the up and down arrows to change the timeout setting, between 1 and 100 moments. Fate will— trigger the ending automatically.

Having Fate Intervene with ClockAlarm

But suppose you don't want your storyworld to end solely based on a timeout? Suppose you want to trigger the ending—or even different endings, or a new set of Verb clusters—based on a set of conditions that the player or other Actors have met? Fate can help you do this. Here is how you do it with a ClockAlarm, as an outcome of an Event.

Suppose you are working on a mystery storyworld, and you want to trigger the murder trial to happen shortly after the Protagonist finds the murder weapon hidden at the scene of the crime.

First, in the Prop Editor, create the prop bloody knife. Now return to the Verb Editor and create two Verbs. The first Verb is discover clue. Under Properties, give discover clue a 3Prop WordSocket for the clues to be found. Next create another Verb called murder weapon found. Under Properties, make this Verb's Audience Requirement "Under the Hood." This will hide it from the player's view in the HistoryBook (more about Audience Requirements). Then return to discover clue. If this were part of a larger web of Verbs, you would have a Role for the Protagonist under discover clue, so go ahead and create one, for form's sake. Now add a second Role and name it Fate. The AssumeRoleIf script for discover clue: Fate should look like this.

```
AND3
  PermitFateToReact
      AreSameActor:
          Fate
          ReactingActor
```

```
    AreSameProp
        ThisProp
        BloodyKnife
```

Now go to the Verb murder weapon found. Go to the Consequences menu and choose CreateClockAlarm. Its script will look like this:

```
murder weapon found: ClockAlarm
  Who?
  HowFarAhead?
```

The Who? in this case would be the judge, who might notify witnesses of the impending trial, or you might just go straight to the court setting and hold the trial. HowFarAhead? is how many storymoments you want to pass before the judge acts. This quantity would depend on how much time you want to give the Protagonist to do other things before the court appearance.

You can see from this example how to use Fate and a ClockAlarm to trigger new sets of Verbs, timed to occur when you choose (note, take care to not make the ClockAlarm time too long, or your storyworld might timeout before the cool new set of Verbs are ever triggered).

* More precisely, Fate's location is permanently set as "Nowhere" — out in the digital ether, floating above the heads of all your other Actors. In Storytronics, an Event always occurs where the Subject is. This is why, when you look in Log Lizard, you'll see that any action carried out by Fate occurs "Nowhere."
**A storyworld's time is measured in "moments." How long is a moment? As long or short as you want it to be, pretty much. Storytronic time is flexible. In most storyworlds, it might be anything from a few seconds to a few minutes, depending on what is needed for pacing purposes. In Chris's *Balance of Power 21st Century*, a "moment" is probably closer to a month.
You can set the number of moments that a Verb takes to execute in the Verb's Properties box, as mentioned in Properties Box.
Because Storytronics is linguistic and turn-based, you have some flexibility regarding how much time you want each moment to "last." You can set the duration of an Event in Properties, but also, just as in traditional storytelling, you may find that your storyworld will not require a strict clock that accounts for the passage of time in minutes, so you can play around with how much happens in one moment versus another. Experiment and see what kinds of effects you can achieve here.
A caveat: if you have a storyworld in which the duration of Events matters—for instance, a ticking-bomb storyworld or an Actor whose condition is worsening quickly and the Protagonist must secure the assistance of a specialist in time; or a spy story in which multiple threads of action occur in different stages, and those threads merge at some point—in these cases, you will need to be careful about how you manage the passage of time in your storyworld. Precision in the duration

of Verbs will be much more important.

Next Tutorial: Who's Fate (part 2)
Previous Tutorial: More Special Operators

Storytron: Who's Fate (part 2)

Last edited by Bill Maya 6 days ago

Ending a Storyworld Based on Conditions

You have the ability to track the progress of your storyworld using Fate. Here's how. Suppose your storyworld is a training storyworld that focuses on teaching the player to increase other Actors' willingness to take on and complete challenging tasks (their Timid_Confident values). Let's assume that you want the storyworld to end after the player through leadership and encouragement increases the average Actor confidence level to 0.4 or more.

First, create a new Verb called "calculate confidence." In the Properties box, make the Audience for this Verb Under the Hood. Make the preparation time 10 storymoments. Uncheck the box labeled "occupies DirObject."

Now go to once upon a time. Under the Role for Fate, the AssumeRoleIf script should look like this:

```
calculate confidence: Fate: AssumeRoleIf:
  AND
      PermitFateToReact
      AreSameActor
          ReactingActor
          Fate
```

This simply says that Fate is the only one who can use this Role.

Now remove your first Verb as an Option and add calculate confidence. Also add penultimate Verb as an Option.

(Note that, if this were a working storyworld, you would also need a separate Role here for the Protagonist. You would need to hook that Role up to a set of Verbs that would allow the player to try different tactics in engaging with the non-human Actors in your storyworld, which would lead to their losing or gaining confidence. We're not going to bother with this step—just pretend that those Verbs are all there and would gradually change all the Actors' Timid_Confidence Core Traits as the player played the storyworld.)

Recall that we said we want Fate to continue allowing the player to play this training storyworld until the other Actors' average Timid_Confidence reaches an average of greater than 0.4. This means that Fate will allow the storyworld to continue to run, as long as the average Timid_Confidence of the non-player Actors is less than or equal to 0.4. Once it exceeds that, Fate will end the storyworld. Here's what you have to do to make that happen.

For the penultimate VerbOption, here is the DirObject Acceptable script:

```
AreSameActor
  CandidateActor
  Protagonist
```

The Inclination script is this:

```
Actor@Average of:
  NOT
      AreSameActor
          CandidateActor
          Protagonist
  Timid_Confident of:
      CandidateActor
```

This script calculates the average Timid_Confident value for all Actors except the Protagonist.

The calculate confidenceInclination script is this:

```
0.4
```

Remember, the Story Engine will compare the Inclination of penultimateVerb with the Inclination for calculate confidence, and will choose the higher value. This is why we set an Inclination for calculate confidence of 0.4— so that the Story Engine continues to calculate confidence until the average Timid_Confident exceeds it. (Also remember that Actor@Average is a BNumber that can range from nearly -1 to nearly +1, so 0.4 is a pretty high confidence level.)

The Verb once upon a time always kicks off a storyworld. With this Role for Fate, you have now set things up so that, if the Actors' average confidence is greater than 0.4 at the storyworld start, it will end right away. Otherwise, Fate will wait ten moments and then perform the Verbcalculate confidence. But since we haven't given Fate
anything to do once it reaches the Verbcalculate confidence, we need to create a Role for Fate there as well.

Copy Fate's Role under once upon a time, go to calculate confidence, and paste the Role there.

By doing this, you have set up a monitoring loop for Fate. Every ten storymoments, Fate has to decide what to do: calculate confidence again, or end the storyworld? If the average confidence level of all Actors except the player stays at 0.4 or below, Fate will wait ten more minutes, and calculate confidence again. If the average Actor confidence exceeds 0.4, Fate chooses penultimate Verb instead, and the storyworld ends.

(Incidentally, you'll also need to be sure to set the Actors' Timid_Confidence Traits so that they average out below 0.4, for this to work. To test whether the script works, though, you can always set the Actors' Timid_Confidences above 0.4, and see if the storyworld ends right away. Or simply use Scriptalyzer to simulate the same effect.)

Unique Characteristics of Fate

Because the Story Engine treats Fate differently than it does the other Actors in your storyworld, there are some things you need to know, if you build Verbs using Fate.
First, Fate does not react to Verbs under normal circumstances. The Story Engine prohibits this, because you wouldn't want Fate butting in all the time. You want Fate to stay out of action until you specifically want Fate to step in. To accomplish this, simply use the special-purpose Operator FatesRole in the Role's AssumeRoleIf script, like so:

FatesRole

Next, since Fate is everywhere at once, you will not get a good result if you try to make Fate the subject of the system Verbs depart for and arrive at.
Finally, it will do you no good to designate Fate as the DirObject or other XxActor in a WordSocket. The Engine will studiously ignore you if you do, because Fate doesn't count as a regular Actor. The good news is, you don't need to put Fate into a WordSocket. If a Verb triggers and you've given Fate a Role in response to that Verb, Fate will always react, as long as you include the AssumeRoleIf script shown above.

Next Tutorial: Poison
Previous Tutorial: Who's Fate (part 1)

Storytron: Poison

Last edited by Bill Maya 6 days ago

Poison is a special layer of protection that the Story Engine provides you. OK, that probably doesn't sound comforting, but once you understand the concept, you'll appreciate our terminology.

We have already made it impossible for you to write a Script that is nonsense. You can't multiply Fred by the chair, nor can you ask whether your first stage is true or false. We simply don't permit you to write anything that is obviously nonsensical.

But what about Scripts that make sense most of the time, but, under the wrong circumstances, MIGHT make no sense? For example, consider this simple Script:

```
CouchPotato_Athletic of:
  PickBestActor
      NOT
          Female of:
              CandidateActor
      CouchPotato_Athletic of:
          CandidateActor
```

This will figure out how much of a couch potato or athlete each of the males in the cast is, and picks the most athletic of them. Sounds reasonable, doesn't it? But let's suppose that some crazy person playing your storyworld has somehow managed to kill off every last male in the cast. There aren't any men left! Which means that PickBestActor won't find anybody to pick. This is a nonsensical situation! What is our poor Engine to do?

The Engine solves this problem by Poisoning the Option in which the Script appears. In other words, the Engine says, "I can't make hide nor hair out of this Script, so I'm giving up on it, which means that I can't complete the calculations for this Option, so I'm just going to ignore the entire Option." It continues running the storyworld, but skips over the bad part. The storyworld won't crash and die if you create a script like this; it will do the best it can with what it's got. And in fact, there will be times when you decide it's OK for a particular instance of Poison to happen under certain rare circumstances.

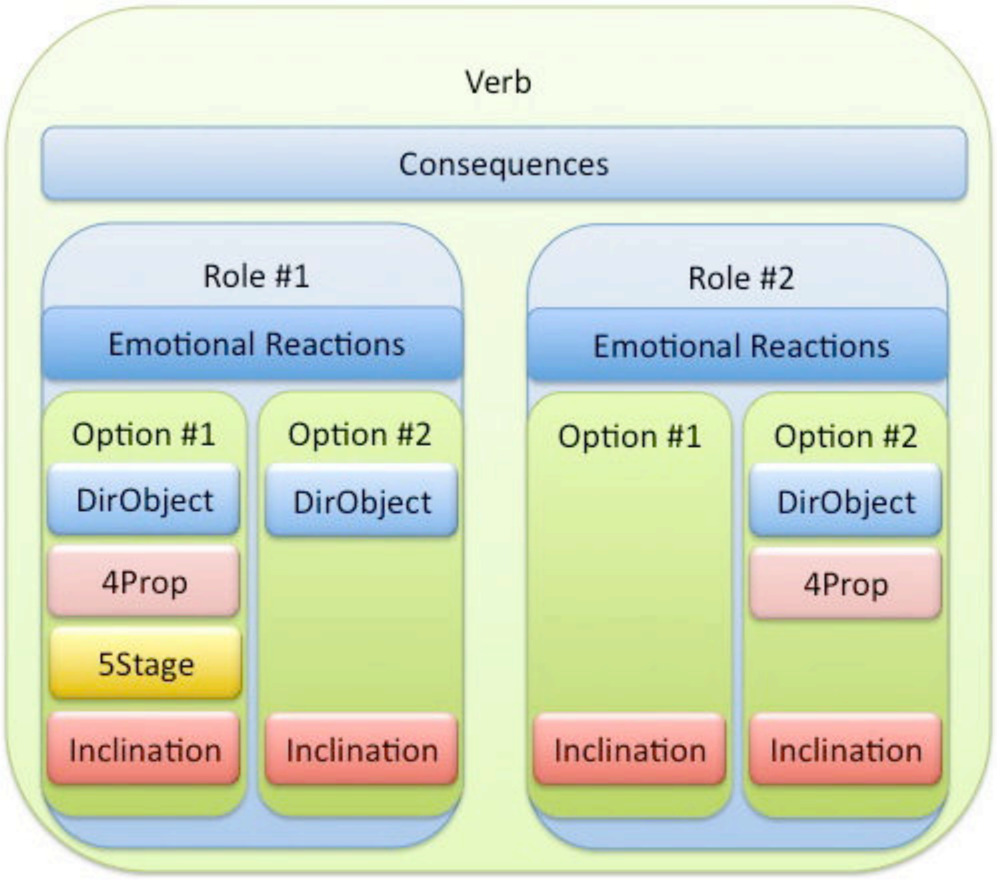
Still, Poison is a dead-end, which means it cuts off options for your player. It can also interfere with important housekeeping tasks that you might be counting on. So it's worthwhile to minimize the opportunities for Poison to occur. Here's how you can diagnose and fix common types of Poison.

How To Know if a Script Is Poisoned

Rehearsal Lizard and Log Lizard both report Poison. Search Lizard can be useful in finding all places where you use a particular Operator that might result in Poison. See Lizards for more information on how to use these features of SWAT.

What Happens If a Script Is Poisoned

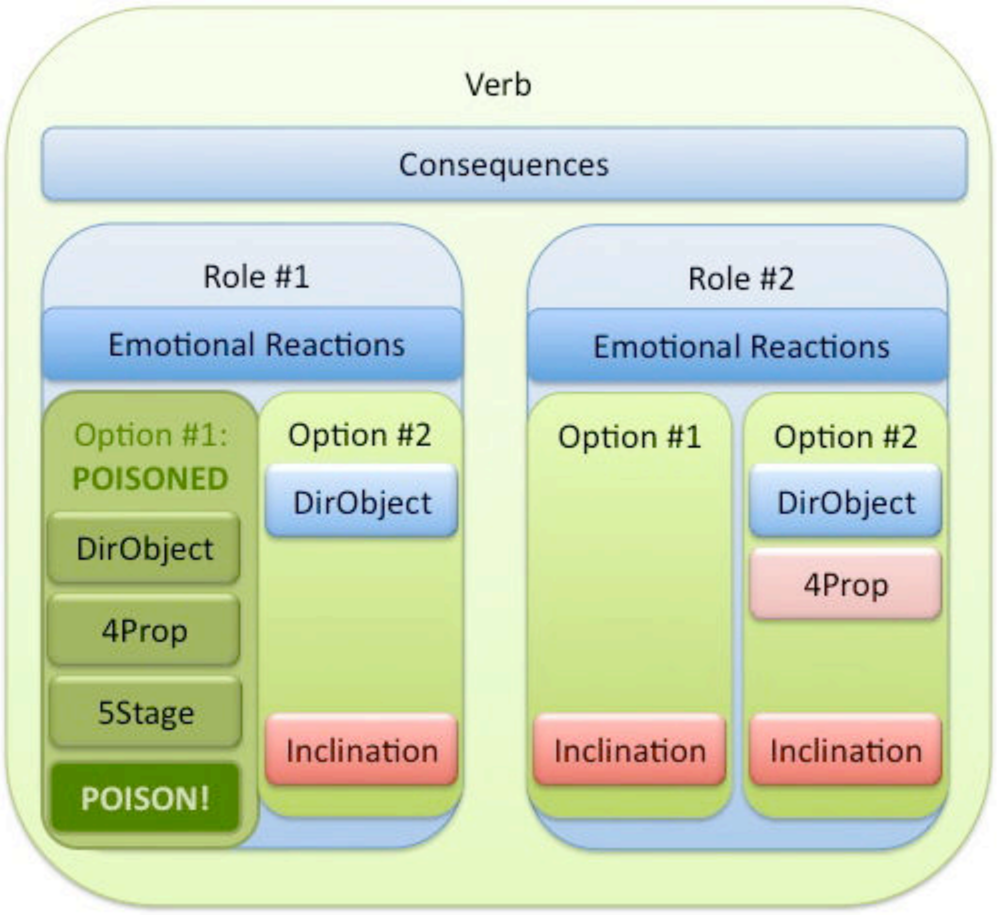
The results of a script Poison depend on where it occurs. Let's delve a little more deeply. Here is how a sample Verb would look, when all its elements are working properly:



All pieces of the Verb are in working order and thus they give meaningful results. We have some physical Consequences (perhaps a Prop changes hands, perhaps weather conditions change on a Stage; perhaps an Actor has died). We have two Roles (one could be the DirObject of ThisVerb; the other might be a Witness). Both Roles, #1 and #2, contain Emotional Reactions, and the ReactingActors for those Roles have Options (i.e., opportunities to take action). Each Option provides the ReactingActors with WordSockets telling them with whom (and/or with what and/or where) they should react. The Inclinations tell how likely they are to take the Options.

Poison in a WordSocket or Inclination

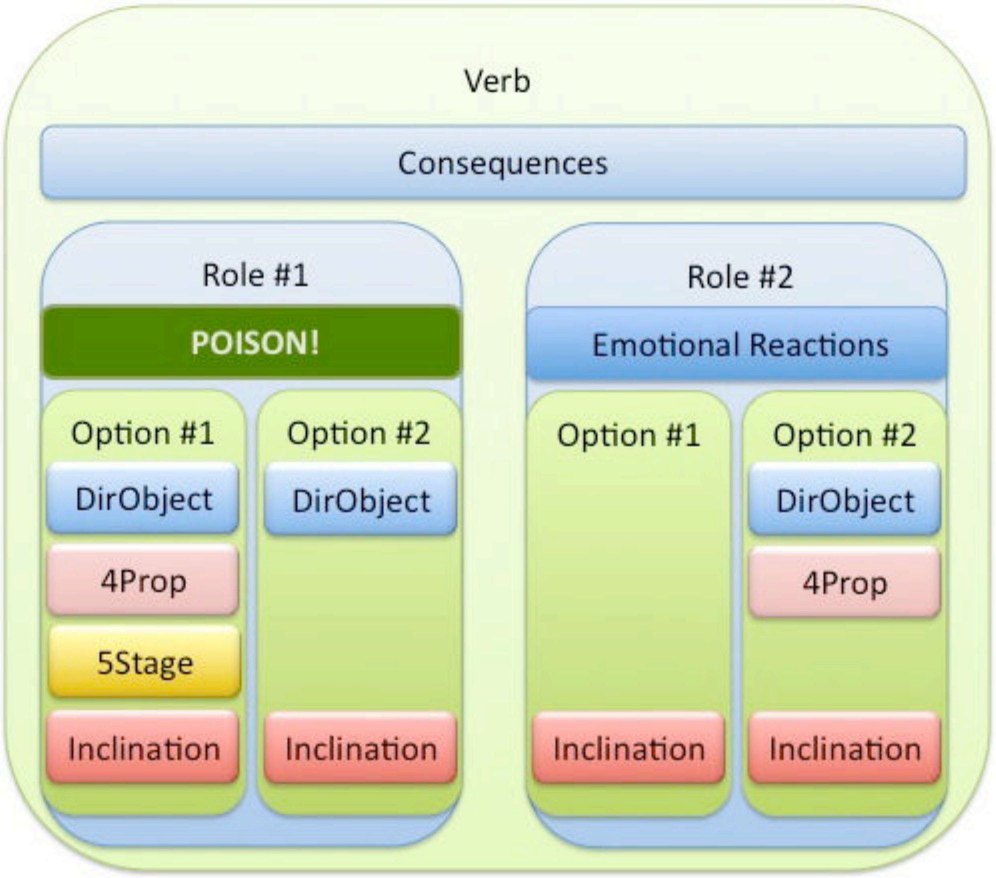
If a script gets Poisoned at the WordSocket or Inclination level, the Poison carries up to the Option level.



At the level of the Option is the lowest possible place this kind of Poison can be contained, because every WordSocket and Inclination is necessary for the Option to make sense. The Story Engine handles a Poisoned Option by making it invisible to the Actors (including the player). In the example shown above, in Role #1, the ReactingActor will have access to only one Option, Option #2.

<u>Poison in an Emotional Reaction</u>

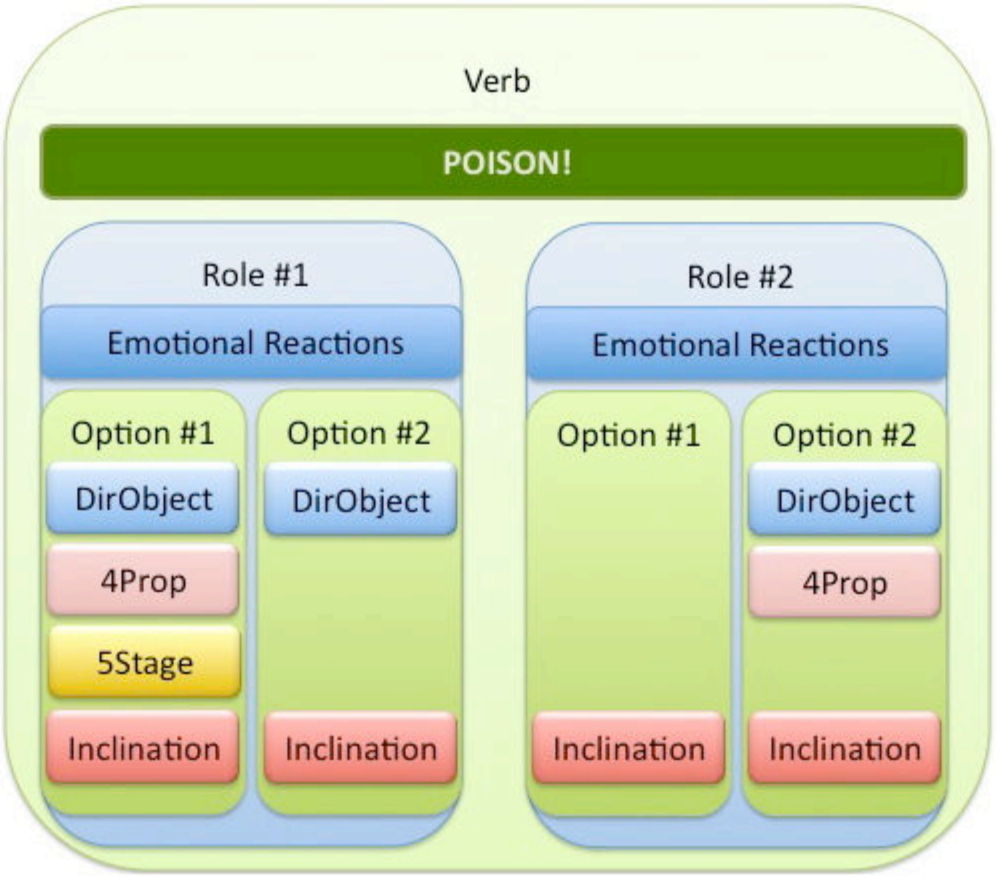
Poisoned Emotional Reaction scripts do not Poison the rest of the Role.



However, they can affect the outcome of your storyworld. For instance, a Poisoned Emotional Reaction script could lead the other Actors to treat your Protagonist exactly the same way after he murders an innocent bystander as they did beforehand.

<u>Poisoned Consequences</u>

As with Emotional Reactions, Consequences with Poison are self-contained, and do not prevent the Verb's Roles from being triggered.



But just as with the other types of Poison, they can cause some odd results. For instance, a Poisoned Consequence could lead to one Actor giving another a Prop, but the Prop not actually changing hands.

Common Poisonings and How to Prevent Them

You may not be able to prevent every possible instance of Poison in your storyworld—and sometimes you may not even want to! But armed with some foreknowledge, you can prevent the most egregious cases. Here are the most frequent causes of Poison and how to fix them.

HistoryBook Operators

There are a lot of HistoryBook operators, any of which can trigger Poison: EventHappened, CausalEventHappened, CountEvents, CountCausalEvents, ElapsedTimeSince, IHaventDoneThisBefore, and IHaventDoneThisSince. Especially early on during storyplay, the Event you are trying to look up may not have happened yet.

The only way to avoid a HistoryBook look-up Poisoning is to be certain that the Event you are attempting to identify has definitely occurred before the Verb in which you are using the HistoryBook Operator. To identify and correct this kind of Poison, use Log Lizard or Rehearsal Lizard.

PickBest_____ Operators

PickBest_____ is another notorious Poisoner. If the Story Engine can't find a match for the conditions you've specified in a PickBest___ clause, it will kill your script dead. As with Lookup, the only prevention for this is to ensure that there will always be at least one thing (Actor, Prop, Trait, or whatever) that fits your PickBest criteria.

Undefined Script Elements

Here is a very common Poison, and one that is easy to fix. Undefined terms in your script result in Poison. The fix is to use Search Lizard (see Lizards), which will generate a clickable list of undefined terms for you. (An undefined term will always start with a question mark, and will appear at the beginning of the Search list.) Click, fix, and go!

Change of Word Socket Data Type

When you change a WordSocket from one type to another (e.g., 4Actor to 4Prop), SWAT does something nifty: instead of throwing away all your scripting work, it keeps your old script. However, this means you almost certainly have some wrong data-type Operators in that WordSocket script. For instance, if you had a 5ActorWordSocket, and you changed it to a 5PropWordSocket, the script probably still contains references to CandidateActor. Since the Engine is no longer considering CandidateActors for the WordSocket, CandidateActor no longer makes sense, and if the Engine runs across this, it will Poison the WordSocket (and thus, the Option).
The way to avoid this kind of Poison is to be sure you immediately update the related script, when you change a WordSocket data type.

Past_____ (and This_____ and Chosen_____)

Another potential Poison source are the Operators Past_____, This_____, and Chosen_____ (e.g., PastSubject, This3Actor, ChosenProp, PastStageTrait). This_____ and Chosen_____ Poisonings are rare, as SWAT has some protections built in, but just as with other HistoryBook type Operators, Past_____ can easily result in Poison. The best way to track these down is to use Rehearsal Lizard or Log Lizard.

Page Number Less Than Zero

In rare instances, you might try to pin down a specific Event by calculating when it happens (that is, its page number). If the number you calculate is less than zero, you will get a Poison. Rehearsal Lizard and Log Lizard allow you to find this type of Poison.

Divide By Zero

First, a terminology check, for those who like me do multiplication and division all the time, but haven't referred to the terms for the different since high school. If you set up a script where you are dividing two terms—say a ÷ b = c—the dividend is the first number, a; the divisor is the second number, b; and the quotient is the answer, c. If your divisor (b)is 0, not surprisingly, your script will blow sky high, as dividing by zero is an arithmetic no-no. Again, the best way to find these is to run Rehearsal Lizard or Log Lizard. You can also (a) test your quotients using Scriptalyzer, and (b) double check them using Search Lizard, to head possible Poisons off at the pass.

Box Reference With An Undefined Box

If you refer to a Box in a script, but you haven't defined it yet, you will get Poison. Though this isn't foolproof, as a quick check for undefined Boxes, in Search Lizard you can compare the number of Fill___Boxes (e.g., FillVerbActorBox: the place where it's defined) versus the number of ___Boxes (e.g., VerbActorBox: the place where it is used).

Poisoning Exercises

For this tutorial, use your test storyworld with some of the above Operators in your scripts. Create conditions you know don't exist in your storyworld to generate Poison. Then try using Log, Rehearsal, and Search Lizards to find and fix them.

Next Tutorial: WordSockets
Previous Tutorial: Who's Fate (part 2)

Storytron: WordSockets

Last edited by Bill Maya 6 days ago

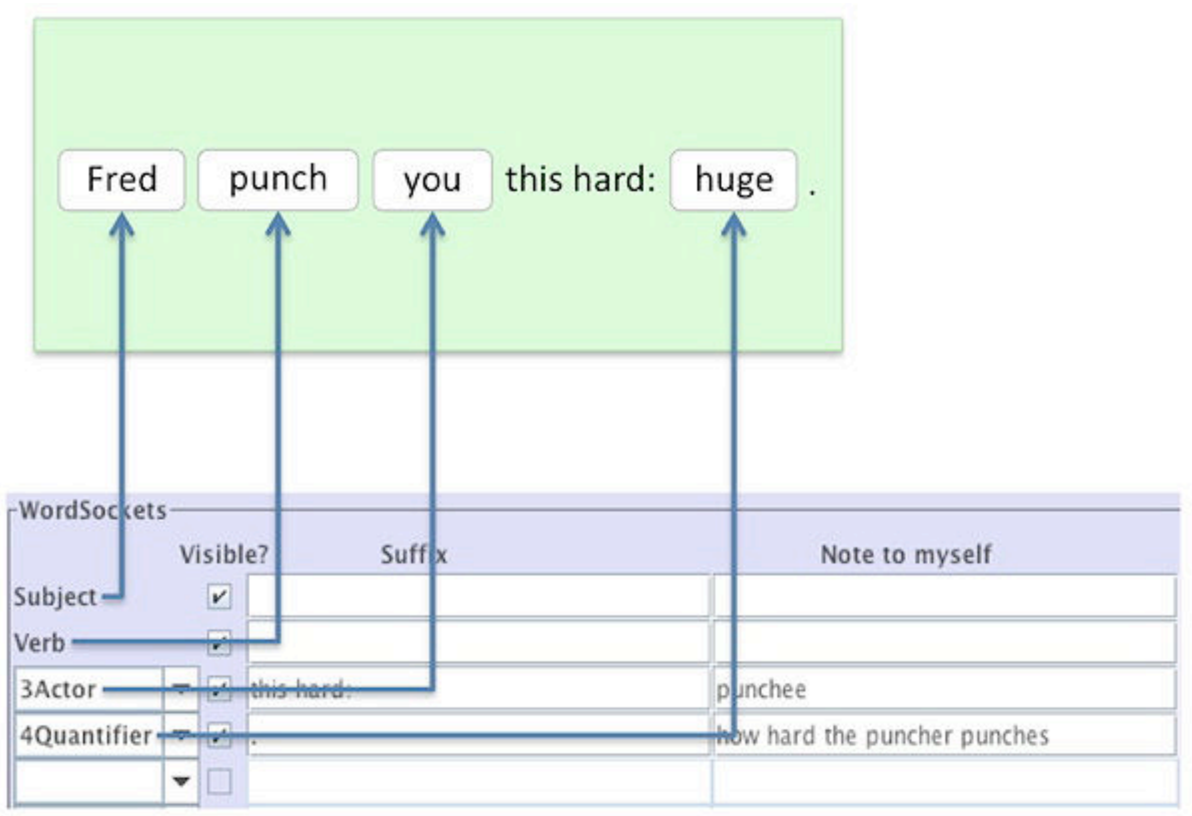
Every Verb has its own customized sentence structure. This sentence that you customize is what the Player interacts with in Storyteller. You define that sentence structure in the Properties dialog box, with the WordSockets that you set up for the Verb. The first two WordSockets are fixed and absolute: the first is the Subject, and the second is the Verb. (Why are they fixed and absolute? Storytronics is a web of interactions, which means every moment, something occurs: i.e., a sentence happens. That sentence must have somebody (the Subject) doing something (a Verb).) After that, you can have as few or as many WordSockets as you need for the Verb.

What's a WordSocket?

A WordSocket has two parts: its type and its content. The type of the WordSocket is just the standard data type for words: Actor, Prop, Stage, Verb, Quantifier, or Attribute. This aspect of WordSockets you set in the Properties box. The content is which specific object goes in the WordSocket. That piece you set in your script instructions for those WordSockets, wherever the Verb appears as an Option.

When you create a WordSocket in a Verb's Properties box, in other words, you are essentially creating a bucket designed to hold script instructions. You are telling SWAT, "These are the active words I want the player to be able to access for this Verb, in this order."

For example, the most common WordSocket is the third WordSocket, immediately after the Verb. This WordSocket often contains an Actor (e.g., the Direct Object of the Deikto sentence: "Fredpunchyou." "You" is 3Actor, the direct object of the sentence). The position of the word is the third position, and the data type for the WordSocket is "Actor." When you choose 3Actor, you are telling SWAT, "Put an Actor bucket in the third position of the sentence the player will see for this Verb." In our Walkthrough, we also included how hard Fred and Tom can punch each other. "How hard?" is a Quantifier data type, and we put it in position four of the sentence. Here is how punch's WordSockets appear as a Deikto sentence in Storyteller, versus how they look in the Properties dialog box:



WordSockets

	Visible?	Suffix	Note to myself
Subject	✓		
Verb	✓		
3Actor	✓	this hard:	punchee
4Quantifier	✓	.	how hard the puncher punches
	☐		

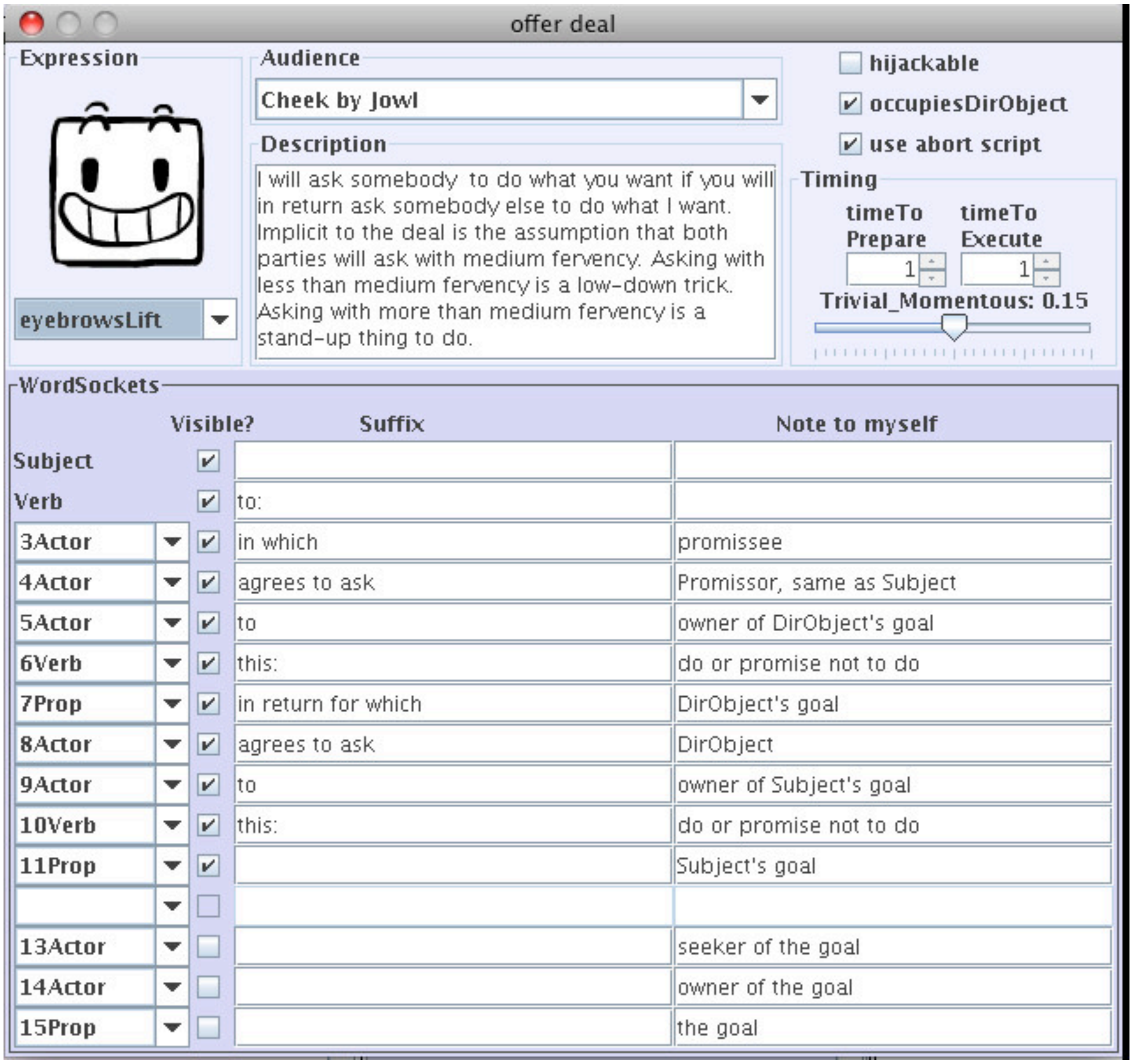
You can have a maximum of 15 WordSockets in a Sentence, and (except for the Subject and the Verb), they are designated by a standard system consisting of the socket number (where it appears in the sentence), followed by the data type (is it an Actor, a Stage, a Prop, a Quantifier, or an Attribute?. For example, "9Stage" refers to the ninth WordSocket, which is of data type "Stage."

When you first create a Verb, its Properties dialog box shows that only the first two WordSockets are active: the Subject and Verb of the Deikto sentence the player will see, as mentioned above. To add more WordSockets to your Deikto sentence, simply click on the blank space for the WordSocket at the left edge of the Properties dialog box, and a popup menu will appear listing the available data types. The topmost of these is empty, indicating indicating that you want the WordSocket to be left empty. If you select anything else, then that WordSocket is active.

However, there's a difference between "active" and "visible to the player." Why? Because sometimes you want to include information in the Sentence that you will use for your own purposes, but you don't want the player to see it. We'll talk more about this feature later, but for now, all you need to know is that the little checkbox just to the right of the popup menu will, if unchecked, prevent that word from being displayed to the player.

WordSocket Suffixes and Notes

Moving further to the right from the popup menu, there's a space for what we call the "suffix." This is additional text that you want to be included in the Deikto Sentence to make it more like normal language. This text doesn't matter at all to the Story Engine, but it helps the player. Here's a good example of the value of suffixes:



Audience: Cheek by Jowl

Description: I will ask somebody to do what you want if you will in return ask somebody else to do what I want. Implicit to the deal is the assumption that both parties will ask with medium fervency. Asking with less than medium fervency is a low-down trick. Asking with more than medium fervency is a stand-up thing to do.

Expression: eyebrowsLift

hijackable [ ]
occupiesDirObject [✓]
use abort script [✓]

timeToPrepare: 1
timeToExecute: 1
Trivial_Momentous: 0.15

	Visible?	Suffix	Note to myself
Subject	✓		
Verb	✓	to:	
3Actor	✓	in which	promissee
4Actor	✓	agrees to ask	Promissor, same as Subject
5Actor	✓	to	owner of DirObject's goal
6Verb	✓	this:	do or promise not to do
7Prop	✓	in return for which	DirObject's goal
8Actor	✓	agrees to ask	DirObject
9Actor	✓	to	owner of Subject's goal
10Verb	✓	this:	do or promise not to do
11Prop	✓		Subject's goal
13Actor			seeker of the goal
14Actor			owner of the goal
15Prop			the goal

This shows the Properties dialog box for the Verb "offer deal" in *Balance of Power: 21st Century*. Notice how many WordSockets are used—this is an exceptionally complicated Verb!

Note all the suffixes. Here's what the Deikto sentence might look like if we *didn't* have the suffixes:

USAoffer dealChinaUSAJapandoJapanapologize toChinaChinaAfghanistandoAfghanistanhand overbin Laden.

Doesn't make any sense, does it? Now, here's the same sentence *with* the suffixes:

USAoffer deal to China in which USA agrees to ask Japan to do this: Japanapologize toChina in return for which China agrees to ask Afghanistan to do this: Afghanistanhand over bin Laden.

That makes a lot more sense, doesn't it? That's the value of suffixes: they allow you to flesh out the sentence with additional text so that it makes more sense. In fact, when we created the Verb "run away from" in our testing storyworld, we could have used the suffix "from" and just called the Verb "run away from."

On the far right side of the Properties box is a set of slots called "Note to myself". These are *very* useful for keeping straight which WordSocket carries which component. We have even included a nice touch: if you hover the mouse over a WordSocket title in the Options display of the Verb Editor, your "Notes to Yourself" for that WordSocket will pop up.

WordSockets Exercise 1: Using Suffixes and Notes

For this exercise, return to your testing storyworld. Make the following changes:

Go to the Verb hit with. Open the Properties window and add the suffix "the" to 3Actor. To 4Prop, add ", this hard:" as a Suffix. Put something like this, "object you use to hit your adversary with," as a Note under 4Prop. For 5Quantifier, add a period in the Suffix field, and "how hard you hit your adversary" as a Note.

Now close the Properties dialog. Look under your Options dropdown, and select hit with as an Option. You should see WordSockets for DirObject, 4Prop, and 5Quantifier. Place your mouse over the 4Prop header and leave it there a moment. You should see your Note as a tooltip. Ditto with 5Quantifier.

Make similar kinds types of changes to the WordSocket Suffixes and Notes for punch, run away from, and plead to desist. Experiment with various Suffixes and then run Storyteller Lizard, to see what kind of effects you can get in Deikto. Try adding Notes that give you reminders as to what each WordSocket is supposed to be, and then hover over the corresponding WordSocket headers to see how they work.

Housekeeping WordSockets

Here's another useful thing you can do with WordSockets. Take another look at the *Balance of Power: 21st Century* example, above. Note that the last three

WordSockets (13Actor, 14Actor, and 15Prop) are not visible to the player. Chris discovered while working on his storyworld that in many Verbs he needs to use the "seeker of the goal", the "owner of the goal", and "the goal" in his Scripts. He decided to include them in every single Verb in WordSockets 13, 14, and 15. They get carried through every Event, so that anytime he needs to write a new Script, he already knows that he has these three key elements immediately at hand.
Using WordSockets like this saves a lot of wear and tear as you create your storyworld. It helps you avoid having to craft complicated HistoryBook Lookups that might result in a Poison.

WordSockets Exercise 2: Using Invisible WordSockets

Even in a simple storyworld, this kind of housekeeping WordSocket can be useful. Recall that in the testing storyworld, in punch: punchee: hit with: 4Prop: Desirable, we used this:

```
BInverse of:
  Harmless_Lethal of:
      CandidateProp
```

This tells the Actor to use the least lethal Prop available.

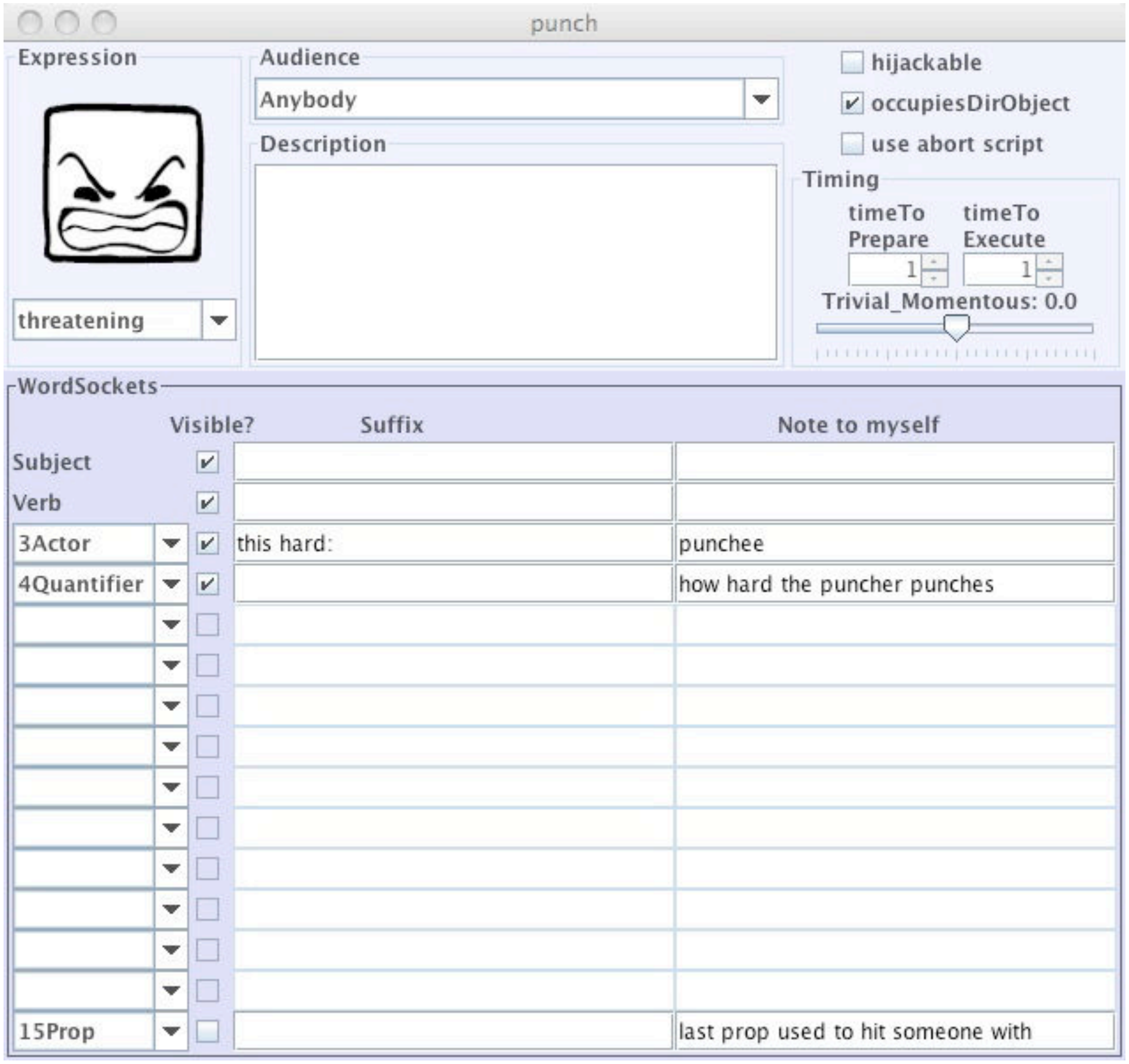
Whereas, in hit with: hittee: hit with: 4Prop: Desirable, we used this:

```
PickUpperIf of:
  TopGreaterThanBottom(BNumber)
      Harmless_Lethal of:
          CandidateProp
      Harmless_Lethal of:
          This4Prop
  BInverse of:
      Harmless_Lethal of:
          CandidateProp
  -0.99
```

This tells the ReactingActor to choose as a weapon the Prop next up the lethality scale from the Prop he just got hit with.
Why didn't we use the same 4Prop construction for the Option hit with under punch as we did for the Option hit with under the Verb hit with? Quite simply, we couldn't. If we tried to use the exact some construction of hit with: 4Prop for *punch* as we did for *hit with*, we would get an error message. To test this for yourself, go to punch: punchee: hit with: 4Prop: Desirable, and try to create the PickUpperIf script above. Notice that when you try to select This4Prop, there is no way to select it from any of the lists. Why? Because punch doesn't use a which 4Prop was used the last time someone decided to do hit with, and as we've

mentioned before, HistoryBook lookups can get very messy.

Instead, try this. For all of the Verbs in your testing storyworld except run away, go into the Properties box and create an invisible 15Prop. Label it with a Note that says something like: "last prop used to hit someone with." It should look something like this:



Audience: Anybody

Expression: threatening

[ ] hijackable
[✓] occupiesDirObject
[ ] use abort script

timeTo Prepare: 1
timeTo Execute: 1
Trivial_Momentous: 0.0

	Visible?	Suffix	Note to myself
Subject	✓		
Verb	✓		
3Actor	✓	this hard:	punchee
4Quantifier	✓		how hard the puncher punches
15Prop			last prop used to hit someone with

After completing this for all your Verbs, you will find that all Options for all three of your Verbs—punch, hit with, and plead to desist—now have a WordSocket for 15Prop. (Why did we not do this for run away from? Because run away from ends your storyworld, so you no longer care which Prop was last used to hit with. Thus there is no need to give yourself the extra work of creating a WordSocket for it.)

Now you need to do some housekeeping, to make sure 15Prop is properly carried through all the interactions. It seems as if it might be a lot of work, but in fact it's not. Wherever the Option hit with appears, this is where we first go to set 15Prop. This is because the Actors choose a *new* Prop to hit each other with whenever they choose hit with.

To start, go to punch: punchee: hit with: 15Prop: Acceptable and enter this:

```
AreSameProp
  CandidateProp
  Chosen4Prop
```

This says, whichever 4Prop the ReactingActor has chosen to clobber his adversary with, store the name of that Prop under 15Prop as well.
Now copy the entire Optionhit with (Edit > Copy Option), and go to the each of the other Roles for Verbs in your storyworld (punch and plead to desist). Wherever you find an existing Optionhit with, delete the old version. To do so, select hit with in the Options dropdown list, then click on the red minus sign. Next, paste the Option (Edit > Paste Option) to add the new version of hit with to your Options list.

Check whether you got all the new hit withOptions installed correctly by double clicking on hit with in the pink Verbs list. Select Comefroms Lizard(Lizards > Comefroms Lizard). This will give you a clickable list of all places where you used hit with as an Option. All the 15Prop scripts for Option: hit with should look like the above.
Return to the verb punch. For every other Option, the Acceptable script for 15Prop will look like this:

```
AreSameProp
  CandidateProp
  This15Prop
```

This carries 15Prop forward as the last Prop used for clobbering purposes, until someone next chooses the Optionhit with. Now you can either copy the script above, and paste it into all your non-hit-with Options for the Verbspunch, hit with, and plead to desist.

Before we go on, let's take a moment to answer a question that may be in your mind. Why the difference between 15Prop Acceptable scripts for hit with versus the others? The answer is that hit with is the only action taken that changes what the last prop used is. In fact, for extra credit, try copying and pasting the script for hit with's 15Prop into one of the others, and watch what happens. You will get an error message telling you that there is no Chosen4Prop for, say, punch. Which is true—Chosen____ can only be used for WordSockets that exist for that Option.

You can make sure you got all the 15Prop scripts correctly assigned in either of two ways. First, go to punch and select Comefrom Lizard. This will give you a clickable list of all places where you used punch as an Option. Paste the script in for 15Prop: Acceptable for all punch come-froms. Use the back arrow to return to punch, and select the next one on the list. Then do the same process for plead to desist.

Last but not least, if you want to make sure you got all of them, go to Search Lizard and look for ?Condition?. Replace any undefined terms for 15Prop: Acceptable with one of the two scripts above.

Next Tutorial: System Verbs
Previous Tutorial: Poison

Storytron: System Verbs

Last edited by Bill Maya 6 days ago

Storytron provides a set of standard Verbs that automatically provide special capabilities, which are necessary for many storyworlds. They are included in the "System" Verb category.

System Verbs have a number of uses, but because they are used in special ways inside the Story Engine, you need to take care with these Verbs except as we describe, or your storyworld may crash. ***Never delete System Verbs.*** The Engine needs them in order to do its job.

Here is a table of the System Verbs, how they are used, and a few do's and don't's.

SYSTEM VERB	WHAT IT'S FOR	HOW TO USE IT	NOTES, CAUTIONS, LIMITS
Story Start and End	**These Verbs trigger the beginning and ending of your storyworld.**		
Once Upon a Time	This Verb starts every storyworld. It automatically triggers the Verb your first verb with the Protagonist (Actor1) as the DirObject.	DO NOT USE THIS VERB.	This is a housekeeping Verb. Not recommended for use as part of your Verb web.
Your First Verb	This Verb is triggered by Fate, and is the first one your player will see.	Rename it and use it as the first Event in your storyworld.	Modify as you see fit.
Penultimate Verb	When the Engine decides that it is time for the story to end, the Event "Fate penultimate verb" will take place. Two things trigger penultimate verb: the Procedure SetStoryIsOver and the Playing menu item, Termination (Timeout).	You can add a Role to this Verb that calculates the player's score (if you have a score) or any other calculation that you think appropriate. Use the Verb happily ever after as an Option for this Role and Fate will present as the last Event "Fate happily ever after you this much: [player's score]." You can also, if you wish, insert additional Verbs between penultimate verb and happily ever after to provide more feedback to the player.	If you create an intervening set of Verbs between penultimate verb and happily ever after, every intervening Verb thread MUST end in happily ever after. If not, your storyworld will never end.
Happily Ever After	Penultimate verb leads automatically to happily ever after, unless you include intervening story feedback Verbs in between the two.	The default condition is that happily ever after directly follows penultimate verb. See above for alternative uses.	As mentioned above, every Verb thread that follows penultimate verb must end with happily ever after or your storyworld will not end properly.
Alarms	**Set these Verbs to trigger when certain special situations arise.**		
ClockAlarm	ClockAlarms are created in the Consequences section for a Verb. You'll find the Procedure "CreateClockAlarm" in the Alarms submenu of the Consequences menu.	Use a ClockAlarm to trigger something that you wish to occur after a specific time delay. When the time comes, an Event "Fate ClockAlarm YourSpecifiedActor" will take place. The player will not see that Event but you can place Roles and Options in the ClockAlarm Verb to make things happen.	When you create a ClockAlarm, you must specify two factors: who the alarm is for, and how far ahead in time it should take place. ClockAlarms can occur at any time, and they are set in relative time terms (that is, relative to the Event in which you place a ClockAlarm), not absolute time terms.

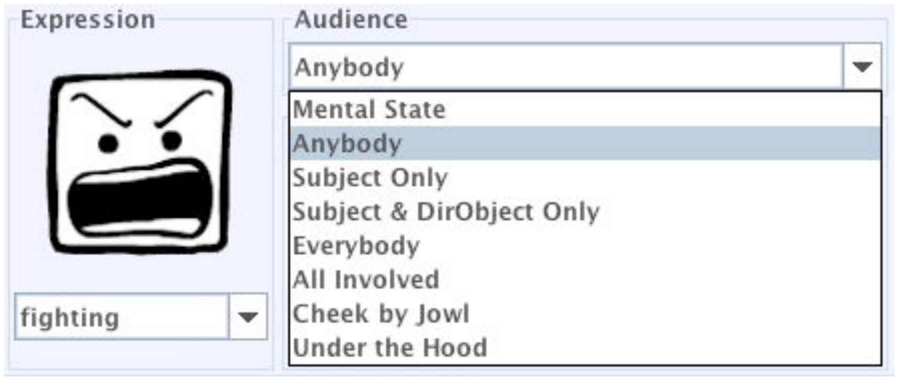
MeetingAlarm	This Alarm is created in exactly the same way that the ClockAlarm is created: with the CreateMeetingAlarm item in the Consequences menu. The most valuable use of this Verb is in providing special introductions to new Actors.	Use the MeetingAlarm to trigger special-case verbs or Roles, the first time they meet after you create the alarm. After this happens the alarm will be erased.	You must specify two arguments for CreateMeetingAlarm: Actor1 and Actor2. The next time these two Actors find themselves on the same Stage, an Event "Fate MeetingAlarm" Actor1, Actor2" will take place. The player will not see that Event but you can place Roles and Options in the MeetingAlarm Verb to make things happen.
PropAlarm	Similar to MeetingAlarm	Use a PropAlarm when the specified Actor first encounters the specified Prop. Its primary use is for providing introductions to important Props.	Similar to MeetingAlarm
StageAlarm	Similar to MeetingAlarm	Use a StageAlarm to trigger special-case Roles or Verbs, when the specified Actor first enters the specified Stage. Again, its primary use is for providing introductions.	Similar to MeetingAlarm
Player Prompts	**These are generic Verbs the Engine uses to cue the player.**		
Do what?	Used as a prompt for the player.	Not recommended for author's use.	This is a housekeeping Verb. Not recommended for use as part of your Verb web.
OK	A do-nothing Verb that allows the player to acknowledge an Event without having to take an action.	Use this Verb when you want Actors to have the Option to do nothing in response to an Event. Delete it if you want to force Actors to make a choice.	Use as an Option but do not modify.
Travel	**The Engine uses these Verbs to move the Actors from Stage to Stage.**		
Depart for	The Engine uses this Verb to move an Actor from Stage to Stage. Normally, the Engine moves Actors around for you automatically, making sure that Actors run into each other frequently. It also guarantees that if one Actor has a Plan to do something requiring another Actor, they will meet in order to permit that Plan to take place.	There are rare occasions that may require your intervention; in these cases, use depart for as an Option and the Engine will move the Actor to that Stage.	CAUTION! Do not use depart for unless the Actor absolutely, positively must get there as soon as possible. You can use SetTargetStage and the Actor will get there ASAP most of the time. In general, it's a better idea to use SetTargetStage than depart for.
Arrive at	The Engine uses this Verb to tell you that an Actor has arrived at their destination.	DO NOT USE THIS VERB.	NEVER EVER select arrive at as an Option for another Verb! It will mess up your storyworld. The Engine will handle arrival automatically.

Next Tutorial: Audience Requirements
Previous Tutorial: WordSockets

Storytron: Audience Requirements

Last edited by Bill Maya 6 days ago

The Audience setting for a Verb is specified using the Audience menu in the Properties box.



Most of the time we don't really care who else is present on the Stage on which the Subject is executing a Verb—our only concern is that the DirObject, if there is one, must be there. After all, Joe can't punchFred if Fred isn't there. However, there are a few situations that impose special considerations on who must be present—or absent—from a Stage in order for the Subject to execute the Verb. In other words, the Subject might balk at executing the Verb if the wrong people are present. We handle these requirements using the "Audience" specification for the Verb. Here are some examples:

Anybody: This is the default audience requirement setting. Anybody means anyone present on the Stage where the Event occurs. Use this setting when it doesn't matter who is present. All Actors present witness the Event.

Subject Only: Use this when the Subject is up to something that he doesn't want anybody else to know about. For example, he's engaging in shameful solitary sexual practices, or preparing the equipment for a crime, or doing something stupid. If your Verb demands privacy for the Subject, then use the "Subject Only" Audience setting.

Subject & DirObject Only: Use this Audience setting when Subject and DirObject must both be present, and nobody else. For example, you might want to use this for Verbs of an intimate romantic or sexual nature.

All Involved: Every Actor involved in the Event must be present. This would apply to any additional Actor WordSockets. You might use this for a Verb such as "counsel antagonists" wherein the Subject admonishes two other people to kiss and make up. Obviously, this Verb won't work if one of those people is missing.

The above Audience settings will cause a Subject to delay executing the Verb until the Audience requirements are satisfied. However, there are also some Audience settings that won't cause an Actor to defer execution of the Verb; they only control who gets to witness the execution of the Verb. These are:

Everybody: This applies to an Event so sensational that everybody is instantly aware of the Event. The grapevine spreads the word spontaneously. Think of how everybody in America knew about 9/11 almost immediately. When you use this Audience setting, everybody witnesses and reacts to it immediately, regardless of their locations.

Mental State: the Subject is thinking about something, making a decision. Although other Actors may be on the same Stage, they are unaware of Subject's action.

Cheek by Jowl: the Subject and DirObject are leaning close together, whispering to each other. Other Actors on the same Stage are unaware of this Event.

Under the Hood: nobody witnesses, and Event does not go into HistoryBook. You should only use Fate as the Subject of an Under the Hood Verb.

Next Tutorial: More About Attributes
Previous Tutorial: System Verbs

Storytron: More About Attributes

Last edited by Bill Maya 6 days ago

In Attributes we talked about Actor, Prop, and Stage core traits and characteristics. In Relationship Editor we touched on Accordance and perceived traits. This tutorial provides more detail on how you can put Weight Traits and Corresponding Traits to use in your scripts.

Weight Traits

In the Actor Editor, you will see that for every Actor Trait, there is a corresponding WeightTrait. WeightTraits are very powerful storytelling tools.

An Actor's Trait tells you what they are really like. Their Weight Trait tells you how they want to be perceived: i.e., how much they value the given Trait. For instance, in our testing storyworld, Fred may be rather hotheaded (that is, he has a somewhat large Cool_Volatile value), but he may want to be *perceived* as levelheaded. This disparity between your who characters are, and who they wish they were, is at the heart of good storytelling, and you can use it to achieve strong effects. See Corresponding Traits below for an example of how this disparity can be used in scripting.

Also, since WeightTraits reflect the characteristics the Actors value in themselves, as a general rule, WeightTraits can also be used as an indicator of what your Actors value in others. Here's an example.

Let's say Carmina has just learned that her friend Florence has deceived her about something fairly minor. Two different kinds of emotional reactions might be appropriate for Carmina:

1. By how much will her perception of Florence's trustworthiness change?
2. How angry she does get about being deceived?

Clearly, Carmina's perception of her friend's honesty will always decrease, if she finds out she has been lied to. But how angry she gets can vary. If she values honesty highly (that is, she has a high False_HonestWeight value—say, 0.4 or so), she might be furious over even a small deception. But if she has a low False_HonestWeight (say, -0.4)—even if she has a high False_Honest!—she does not place a high premium on honesty, either in herself, or in others. In this case, she might get mildly irritated, but would be more tolerant of Florence's fib.

Here is how the AdjustFearful_Angry Script might look:

```
fib to: fibbee: AdjustFearful_Angry
  BNumber2UNumber of:
```

False_HonestWeight of:
    ReactingActor

The BNumber2UNumber term says the Actor will never become fearful over being lied to; only angry. The extent to which the ReactingActor values the characteristic of honesty determines how mad a fib will make him or her. Create this script and then test it in Scriptalyzer. Notice how differently the script behaves for a high False_HonestWeight versus a low one.

Corresponding Traits

Here is how to use the Corresponding___Trait set of Operators.

For every invisible core Attribute, there is a corresponding set of related Attributes. For Actors the traits include:

- Xxx_Yyy (Core Trait - e.g., Cool_Volatile)
- AccordXxx_Yyy (Accordance Trait - e.g., AccordCool_Volatile)
- Xxx_YyyWeight (Weight Trait - e.g., Cool_VolatileWeight)
- pXxx_Yyy (Perceived Trait - e.g., pCool_Volatile)
- pXxx_YyyWeight (Perceived WeightTrait - e.g., pCool_VolatileWeight)
- cXxx_Yyy (Confidence Trait - e.g., cCool_Volatile)

For Props:

- Xxx_Yyy (Core Trait - e.g., Harmless_Lethal)
- pXxx_Yyy (Perceived Trait - e.g., pHarmless_Lethal)
- cXxx_Yyy (Confidence Trait - e.g., cHarmless_Lethal)

For Stages:

- Xxx_Yyy (Core Trait - e.g., Grungy_Elegant)
- pXxx_Yyy (Perceived Trait - e.g., pGrungy_Elegant)

The Corresponding___Trait Operators allow you to cross-reference these different related Attributes. Here is an example.

Assume that you want to create a sequence where Actors can insult each other. Let's assume that the Actors have three different choices. They can call their foe ugly, stupid, or mean. In the Verbinsult appearance, the Deikto sentence would look something like this:

Subject - insult appearance - DirObject - 4ActorTrait - 5Quantifier

— where 4ActorTrait is the Attribute that ReactingActor wants to insult, and 5Quantifier is the degree of insult.

To use this script, your ReactingActors have to figure out what *they believe* their foe is most sensitive about. It turns out that there is a simple way to do this. Storytronic Actors all have a perception of each others' personalities, including how important certain characteristics are to them. These latter perceptions of others' values are the Actors' p___WeightTraits. In other words, Mark's pStupid_SmartWeight for Jonathan is how much Mark *thinks* Jonathan wants to be perceived as smart.

The 4ActorTrait Desirable WordSocket, then, would look like this:

```
CorrespondingPWeight of:
   ReactingActor
   ThisSubject
   Candidate4ActorTrait
```

With this script, ReactingActor chooses the Attribute he or she believes the DirObject is most sensitive about.

(For extra credit, can you figure out what the 5QuantifierWordSocket scripts should look like? As a suggestion, you might tie the intensity of the insult to how much ReactingActor dislikes ThisSubject; that is the pNasty_Nice of ReactingActor for ThisSubject. Or you might use Fearful_Angry. A couple of hints: to make this Verb work properly, all traits that are deemed acceptable for insult will need to be bipolar, and you will need to use BInverse of BNumber2UNumber if you want the Actor to always say something negative.)

Next Tutorial: Quantifiers
Previous Tutorial: Audience Requirements

Storytron: Quantifiers

Last edited by Bill Maya 6 days ago

Another type of word that we use in Deikto is a Quantifier. This is a word that indicates the magnitude of some quantity. Quantifiers can be used to intensify a Verbor modify an Attribute. There are eleven Quantifiers:

- extra tiny (-0.99)
- tiny (-0.8)
- very small (-0.6)
- small (-0.4)
- medium-small (-0.2)
- medium (0.0)
- medium-large (+0.2)
- large (+0.4)
- very large (+0.6)
- huge (+0.8)
- extra huge (+0.99)

There are two situations in which you will write scripts using Quantifiers:

1. Writing the Acceptable and Desirable scripts for a QuantifierWordSocket; and
2. Using an existing Quantifier to calculate a result.

Let’s take them separately:

How to Write Acceptable and Desirable Scripts for a Quantifier

Assume you want to set a Quantifier to be equivalent to an Actor's Fearful_Angry. The scripts for that Quantifier's WordSocket would look like this:

```
Acceptable:
  true

Desirable:
  Suitability of:
      CandidateQuantifier
      Fearful_Angry of:
          ReactingActor
```

Suitability is a special Operator that we cooked up just to handle the problem of selecting the right Quantifier for a position. It yields the highest value for the CandidateQuantifier most closely corresponding to the value of the BNumber argument.

Following is an exercise for how you might use a Quantifier.

Exercise 1: Setting a Quantifier in your Walkthrough Storyworld

Let's harken back to the testing storyworld created in the first tutorial section. Tom and Fred get into a bar fight, and Mary intervenes with Fred when he hits Tom with a Prop. Her only choice was to plead to desist. Let's add a Verb scold.

First, we need some more Attributes to make this work properly. Go to the Actor Editor and add two Attributes: Submissive_Dominant and Nasty_Nice. Assign Submissive_Dominant, Nasty_Nice, Submissive_DominantWeight, and Nasty_NiceWeight values for each of the Actors. (Recall that the Attributes themselves describe how much of that personality trait an Actor has, whereas the Weights tell how much the Actor wants to be *perceived* as having that Trait. Consequently Weights also reflect how important it is to the Actor that *others* have that Trait.)

Now return to the Verb Editor. Create a new Verb, scold, in your walkthrough storyworld, give it 3Actor and 4QuantifierWordSockets under Properties, and add it as an Option for the girlfriendRole under hit with. Now create two emotional reactions for Mary.

your first category: hit with: girlfiend: Adjust Fearful_Angry:

```
Submissive_Dominant of:
  ReactingActor
```

This script says that Mary will either become fearful to see Tom hit with an object, or angry, depending on how submissive or dominant she is.

Next, let's adjust her opinion of Fred:

your first category: hit with: girlfiend: Adjust PNasty_Nice:

```
ThisSubject
  BInverse of:
      BNumber2UNumber of:
      Harmless_Lethal of:
          This4Prop
```

This says that Mary's opinion of ThisSubject's (that is, Fred's) niceness will always go down (BInverse of BNumber2UNumber accomplishes this). That is, Mary never likes to see anyone hitting her boyfriend, Tom. How far her opinion of how nice he is will drop is affected by how lethal the Prop is that was used to hit Tom. (To see why this script works the way it does, look at how this script behaves in Scriptalyzer. We'll talk more about UNumbers, BNumbers, and so forth in our next tutorial, BNumbers, UNumbers, and Numbers.)

Now look under the Optionscold. Make the Inclination to scold match the Fearful_Angry of ReactingActor. This says that Mary will only scoldFred if she is angry; otherwise, she will plead for him to desist.

Then create the scripts shown above for 4Quantifier.

```
Acceptable:
  true

Desirable:
  Suitability of:
      CandidateQuantifier
      Fearful_Angry of:
          ReactingActor
```

This says that the intensity of Mary's scolding of Fred will match the level of her anger.

How to Use an Existing Quantifier to Calculate a Result

Quantifiers can be used to affect Actors' decisions. To do this, the exact value of the Quantifier (e.g., "tiny," "very large," etc.) must be convert to a value that can be used in a Desirable or Inclination script. Depending on your needs, you can convert a Quantifier and use it as a BNumber, UNumber, or Number.
To convert an existing Quantifier to a BNumber, in response to an action using that Quantifier, use the following script:

```
Quantifier2BNumber of:
    This5Quantifier
```

This assumes that the Verb you are scripting in has a Quantifier in the 5th position. It tells the Engine to convert that Quantifier (extra tiny through extra huge) to a BNumber (-0.99 through +0.99) (Warning: if ThisVerb does not have a 5Quantifier in Properties, you will not be able to access the Operator "This5Quantifier.")
To convert a Quantifier to a UNumber under the same circumstances, use the following script:

```
Quantifier2UNumber of:
  This5Quantifier
```

This converts This5Quantifier to a UNumber (0.0 through 1.0).

To convert a Quantifier to a Number, use the following:

```
Quantifier2Scaler of:
  This5Quantifier
```

This converts This5Quantifier to a Number (0.0 through 1.0). (Quantifier2Scaler returns the same numerical value as Quantifier2UNumber, only with a Number data type instead of a UNumber/ BNumber data type.)

More on BNumbers, UNumbers, and Numbers can be found in the next tutorial.

Exercise 2: Scripting with a Quantifier

Carrying on from Exercise #1, above, let's assume that Fred has just been scolded by Mary. Now he has to respond, and he will respond according to how sternly Mary has scolded him.

First we need to give Fred some Options. Let's create two new Verbs: apologize and yell at. Give each new Verb a 3ActorWordSocket (hint: see the Verbs' Properties boxes). Then return to scold and add a Role, scoldee. This will be Fred's Role, but let's save ourselves some work. We'll write the AssumeRoleIf script such that others can use it as well, in case we want to allow other Actors to scold each other in different settings. Use this:

```
your first category: scold: scoldee
AreSameActor
  ReactingActor
  ThisDirObject
```

The person who will assume the Role will be the person just scolded—in our example, Fred.

Now add apologize and yell at as Options. To do so, single-click on each Verb in the Verb tree, and then use the green plus-arrow below the Options dropdown box.

For both apologize and yell at, here is the DirObjectWordSocket:

```
Acceptable:
  AreSameActor:
     CandidateActor
     This Subject
```

This says the scolder will be the person scoldee reacts to (if you wanted to get fancy, you could have him apologize to Tom instead, as Tom was the one he was hitting...but let's keep it simple for now). Desirable you can leave as is.

For yell at, use this for the Inclination:

```
your first category: scold: scoldee: yell at
   Quantifier2UNumber of:
        This5Quantifier
```

Fred's Inclination to yell atMary depends on how severely she scolded him.

Notice we converted the Quantifier to a UNumber instead of a BNumber. This means that the lowest possible value for yell at: Inclination is a hair above zero. This is because we assumed that even a tiny scold might make him inclined to react defensively instead of apologetically. However, you could use Quantifier2BNumber instead. This would make Fred less likely to yell atMary if she scolded him. See what we mean by experimenting with each Quantifier conversion Operator in Scriptalyzer.

Now, for apologize, make the Inclination this:

```
your first category: scold: scoldee: apologize
   pNasty_Nice of:
        ReactingActor
        ThisSubject
```

This says that Fred's Inclination to apologize to Mary hinges on how much he likes Mary (for how much he likes her, I've used his perception of how nice she is; i.e., his pNasty_Nice of her). The Engine will compare Fred's pNasty_Nice of Mary to how severely she scolded him, and assign Fred a plan to either yell at her, or apologize, accordingly.

Once you have the scripts all built, try your test storyworld out in Storyteller, and see what happens. Then try adjusting the Actors' personality and relationship Attributes, and see how that changes the outcome.

Next Tutorial: BNumbers, UNumbers, and Numbers
Previous Tutorial: More About Attributes

Storytron: BNumbers, UNumbers, and Numbers

Last edited by Bill Maya 6 days ago

BNumbers are the standard number type that we apply to all Attributes. They are the number type you will use most often in scripting. However, there are some situations that call for different kinds of numbers, and so we also provide two other kinds of numbers: UNumbers, and regular Numbers.

How to Use BNumbers

As mentioned in Attributes, BNumbers are designed to put everything on a consistent scale. They make it possible for you to have Actors with all of their physical, emotional, and relationship Attributes—not to mention all the Props and Stages and their Attributes—behaving in ways that can be expressed in similar mathematical terms. Storyworlds are a simplified reflection of the real world, and we need some way to put dramatic concepts into terms the computer can understand. Basically, BNumbers allow you to compare apples to oranges and typhoid to hairnets. Though they can be a hassle to figure out when you first start using them, BNumbers solve the problem of how to juggle many different kinds of information within a single storyworld.

The primary BNumber Operators are BSum, BDifference, BProduct, BInverse, BAbsVal, and Blend. Understanding these six Operators will enable you to do most of what you will want to do in scripting. There are many other BNumber Operators in addition to these. You can download and view all SWAT Operators in Snips, Tips, and Tricks. Also, see Attributes for further detail on BNumbers.

Exercise 1: Using BNumber Operators

As we mentioned in earlier tutorials, BNumbers behave very differently from regular Numbers. If BNumbers are still giving you headaches, here is a simple set of exercises that can help you wrap your head around how they behave.
Go to any Inclination or Desirable script. Insert the BNumber Operator BSum in the following format:

```
BSum of:
   Number1?
   Number2?
```

In the script, set both Arguments to 0.0. Then open Scriptalyzer. Use the slider bars as follows.

a. Set Number1 to 0.0.
b. Write down the overall result you get for BSum when Number1 is 0.0 and Number2 is -0.99.
c. Write down the result when Number2 is 0.0.

d. Write down the result when Number2+0.99.
e. Repeat a-d. with Number1 set to -0.99 and Number2 set first to -0.99, then 0.0, then +0.99. Write down the three different results.
f. Repeat a-d. with Number1 set at +0.99 and Number2 set first to -0.99, then 0.0, then +0.99. Write down the results.

Now do the exercises in a-f. above for BDifference, BProduct, BAbsVal, and Blend.

For BAbsval, you will only need one parameter: Number1 = -0.99, 0.0, and +0.99.

For Blend, you will have three Arguments, so you will need to run through a-f. three times. For the first run, start with a bias factor of -0.5 and do a-f. for the first two Arguments. Repeat this with a bias factor (third Argument) of 0.0. Then run it through a third time with a bias factor of +0.5.

This exercise may seem tedious, but I urge you to get a notepad and pencil and try it. Taking this systematic approach very quickly reveals the behavior of BNumbers in the wild and will save you time down the road.

How to Use UNumbers

UNumbers are a special version of BNumbers: they're BNumbers that are squashed into the range 0.0 to +1.0. Here's a table matching BNumbers to their UNumber equivalent:

BNumber	UNumber
-1	0
-0.5	+0.25
0	+0.5
+0.5	+0.75
+1.0	+1.0

A BNumber can be converted to a UNumber with the Operator BNumber2UNumber; a UNumber can be converted to a BNumber with the Operator UNumber2BNumber.
UNumbers have a variety of uses. For instance, they can be applied to situations in which you want a percentage scale of something. For example, suppose you want an Actor to spend some portion of his wealth on something. You can't use a BNumber to represent the percentage—after all, what does "-0.5 of your wealth"

mean? So you convert the BNumber -0.5 to the UNumber +0.25 and now you can calculate with "25% of your wealth."

There's a simple rule of thumb for when to use BNumbers and UNumbers. If you're going to use BProduct, then you must use at least one UNumber. Using two BNumbers in BProduct will probably yield results that you don't want.

Exercise 2: Using UNumbers as Mediators

One common and extremely useful application of UNumbers is as a mediator of an effect. In the prior tutorial on Quantifiers, recall that we based Fred's inclination to apologize to Mary on how much he liked her—that is, his pNasty_Nice of her. But suppose we wanted to *mediate* his reaction to her with how angry he is because of the fight?

In other words, the angrier Fred is, the more likely it is that he will react negatively to being scolded. So the new Inclination script for apologize would then look like this:

```
your first category: scold: scoldee: apologize
Inclination
  BProduct of:
      pNasty_Nice of:
          ReactingActor
          ThisSubject
      BNumber2UNumber of:
          Cool_Volatile of:
            ReactingActor
```

Create this script and then test it in Scriptalyzer. First set an average Cool_Volatile for Fred (i.e., 0.0), using the Scriptalyzer slider. Now input a range of different pNasty_Nices, first low, then medium, then high.

Next, set a very high value of Fred's Cool_Volatile (say, 0.8), and then see what you get as a result with a low, medium, and high pNasty_Nice value.
Third, set a very low value (-0.8). Run through low, medium, and high values for Cool_Volatile.

What kind of results did you get for Fred's inclination to apologize when he was level-headed, versus when he was a hot-head?

Do you see how when Fred's Cool_Volatile is very low, his inclination to apologize to Mary does not change very much in either direction, no matter how much he likes or dislikes her? On the other hand, if he is a very sensitive soul who reacts strongly to emotionally charged events (i.e., his Cool_Volatile trait is very high), his decision on whether to apologize to Mary or not will swing widely, depending on how much he likes her.

This usage of UNumbers we call mediation. Whenever you want to mediate an Actor's response based on a particular Attribute (that is, if you envision that some of your characters will have a strong response to an Event, whereas others will have a weaker response, due to a difference in their personalities, for instance), convert the Attribute to a UNumber, and multiply it by the primary BNumber factor. In the example above, the primary factor is pNasty_Nice of ReactingActor for ThisSubject, and the mediating factor is Nasty_NiceWeight of ReactingActor.

How to Use Numbers

We also use plain old everyday Numbers. You can convert a Number to a BNumber with the Operator Number2BNumber, and the other way with BNumber2Number. Here's another table showing the relationship between Numbers and BNumbers:

BNumber	Number
-1	-infinity
-0.99	-99
-0.75	-3
-0.5	-1
-0.25	-0.33
0	0
+0.25	+0.33
+0.50	+1
+0.75	+3
+0.9	+9
+1	+infinity

BNumbers and UNumbers are the same color, but regular Numbers are a slightly different color. That's because you can't use Numbers in the same places that you would use BNumbers, or vice versa. If you want to mix them, you have to use the conversion Operator, Number2BNumber. Here is an example:

```
BSum of:
  Ugly_Attractive of:
      ReactingActor
  Number2BNumber of:
      4.0
```

Numbers include such things as tallies of how many Actors have performed a given Verb, for instance; counts of different sorts; or averages or totals of Attributes. You might use a Number in a script, such as CountEvents, to determine how many times ReactingActor has chosen a particular action to slowly decrease the Actor's likelihood of taking that action again. (This gives your Actors more varied sets of behavior over time.)

For instance, in the testing storyworld we created in the first set of tutorials, if you wanted your non-human Actor Tom to get tired of hitting and punching, and eventually go off and do something else, you could create a script that looks something like this:

your first category: punch: punchee: punch: Inclination:

```
BDifference of:
  Fear_Anger of:
      ReactingActor
  Number2BNumber of:
      CountEvents of:
          MainClauseIs
              ReactingActor
              punch
              ThisSubject
```

You would do the same thing for your first category: punch: punchee: hit with: Inclination, as well as both Options under your first category: hit with: hittee. Adding this clause enables Actors to grow bored of repeating the same behavior and try new things. (If you want Fred to grow bored more slowly, simply add a quotient:

```
BDifference of:
  Fear_Anger of:
      ReactingActor
  Number2BNumber of:
      quotient of:
          CountEvents of:
              MainClauseIs
                  ReactingActor
```

punch
ThisSubject
10.0

You get similar basic arithmetic Operators for regular Numbers that you get for BNumbers: Sum, Difference, Product, AbsVal, Inverse, and so forth. See the Downloads page for details.

PostScript: Crunching the (B-, U-, and Q-) Numbers

For the uber-geeks among us who need to work the numbers themselves to see how they work, the Downloads page contains a downloadable Operator file with the equations for:

- Converting a Number to a BNumber and vice versa;
- Converting a BNumber to a UNumber and vice versa;
- The primary BNumber Operators: BSum, and BDifference; and
- Converting BNumbers and UNumbers to a Quantifier and vice versa.

Next Tutorial: All You Need Is Blend (part 1)
Previous Tutorial: Quantifiers

Storytron: All You Need Is Blend (part 1)

Last edited by Bill Maya 6 days ago

In Special Operators we talked a little about Blend and the kinds of things it lets you do. The Blend Operator is far and away the most useful tool for calculations with BNumbers. With this one Operator, you can perform just about all of the common calculations we find in Scripts.

Averaging Two BNumbers (with or without a Bias)

Blend's most frequent use is to take two values and find a value between them, or in other words, to average them. It also allows you to put your thumb on the scale toward one or the other value, by use of the third term, the bias factor. For instance, let's suppose you have a family drama sequence in which a character, a ne'er-do-well, gets into trouble and has to go around begging family members for help. All family members have to decide whether to intervene for him. Let's further suppose you want each family member to take into account both how close their kinship is to the begging Actor, as well as how much affection they have for him.
The script might look like this:

```
beg for help:beggee: intervene on behalf of: Inclination
Blend of:
  Kinship of:
     ReactingActor
     ThisSubject
  PNasty_Nice of:
     ReactingActor
     ThisSubject
  0.0
```

In this example, the Actor who has to make a decision whether to intervene is the ReactingActor and the ne'er-do-well who has just begged for help is ThisSubject. Notice that this script assumes the degree of Kinship is equally important to the outcome as the degree of affection ReactingActor has for ThisSubject. But you can also use the bias factor to change the balance. Let's walk through some examples.

(1) In the example above, if Kinship is +0.5 (ReactingActor and ThisSubject are cousins) but PNasty_Nice is -0.5 (ReactingActor actively dislikes ThisSubject), you get this:

```
Blend of:
  +0.5
  -0.5
```

  0.0

This yields a result of 0.0—the average of +0.5 and -0.5. In this scenario, ReactingActor might intervene for ThisSubject. But don't bet your life savings on it; there's only a 50/50 chance that it'll happen, all other things being equal. This is what you get when you place equal importance on Kinship and PNasty_Nice.

(2) You can also do weighted averages, which give greater weight to one of the Blend factors. Let's try making the degree of Kinship a little more important than PNasty_Nice. In this case, as above, ReactingActor and ThisSubject are still cousins (Kinship is +0.5), and ReactingActor still dislikes ThisSubject (PNasty_Nice is -0.5), but we change the bias factor to +0.2—that is, we put our thumb on the scales and tip the result toward the first term, Kinship.

Blend of:
  +0.5
  -0.5
  +0.2

(The +0.2 means we are biasing toward the first term, Kinship. If we used a -0.2, we'd be biasing toward the second term, PNasty_Nice.)
This yields a result of +0.1, because the bias factor of +0.2 gives slightly greater weight to the +0.5. ReactingActor is a bit more likely to help his irritating cousin out.

(3) What if we want to make ReactingActor's PNasty_Nice for ThisSubject more important than than Kinship?

Blend of:
  +0.5
  -0.5
  -0.6

A -0.6 biases the result strongly toward the second term, PNasty_Nice. This yields a result of -0.3. In this case, ReactingActor is quite a bit less likely to help out his or her irritating cousin than in either of the prior two examples.

If you play around with the Blend operator in Scryptalyzer you can see this principle at work.

Next Tutorial: All You Need Is Blend (part 2)
Previous Tutorial: BNumbers, UNumbers, and Numbers

Storytron: All You Need Is Blend (part 2)

Last edited by Bill Maya 6 days ago

Pushing Away from, Pulling Toward—Blending with the Center and the Ends

It turns out Blend can be used to do a lot of different kinds of things besides averaging two values with a bias. To understand these uses takes a little explanation first.
Blend works like this:

Blend of:
   A (FromValue?)
   B (ToValue?)
   C (HowFar?)

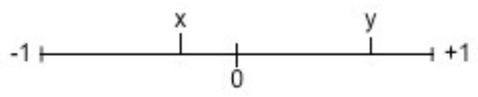
What's really happening when you use Blend is that you start with the first value, A, and move toward B. How far you move from A to B to get your answer is proportional to the value of C. (A, B, and C are all BNumbers, and thus can range from nearly -1 to nearly +1).

This is exactly what we did in the examples above. We started with Kinship between ReactingActor and ThisSubject, and moved in the direction of ReactingActor's PNasty_Nice toward ThisSubject. How far we moved in the direction of PNasty_Nice depended on the value of the bias factor, C. So a more general way of describing what Blend does is to say, "Start at A and head toward B, in proportion to C."

We've already talked about how to use this for two Attributes that you want to average, by assigning your two different factors to A and B, and the bias as C, but Blend has some other interesting uses, as well.

Blending from Minimum to a Value

The trick lies in thinking about Blend in terms of a number line:

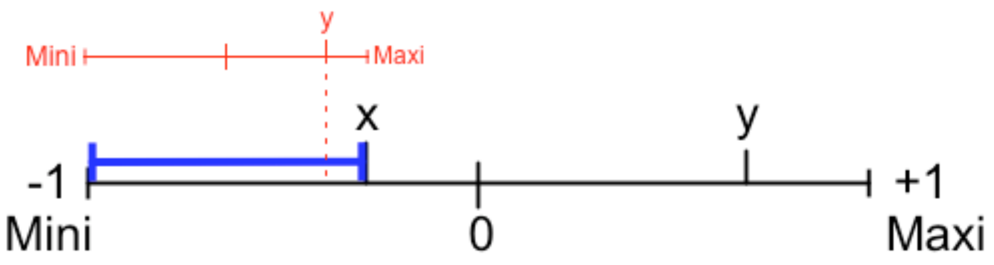


Imagine that x and y represent BNumbers that you want to mess with. Here's the trick: the three numbers -1, 0, and +1 are also values that can be used with Blend. (Actually, this is not technically true. In BNumber arithmetic, the values -1.0000 and +1.0000 are not allowed. Instead, you have to use -0.9999 or +0.9999. To make this easier, we have created created two special constants, Maxi and Mini, that you can find in the Arithmetic menu, that represent these two extremes.)

Consider, for example, this use of Blend:

Blend of:
  Mini
  x
  y

Here's how to visualize it:

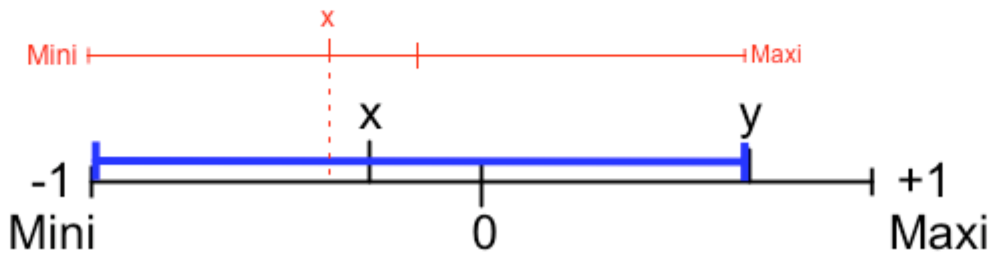


The main number line is shown in black. The blue line marks the range inside which the Blend result must fall, because Blend always produces a number *between* its first argument and the its second argument (between Mini and x, in this example). Now think of the blue line as the number line that y sits on. We have drawn an image in red of y's numberline superposed over the blue line. (In effect, we have taken the full black number line and squeezed it down to fit it over the blue line.) Now y marks the result given by Blend. To express the idea verbally: Blend pushes x towards Mini in proportion to y.

Just for fun (yeah, right: math is fun...), let's reverse it, like so:

Blend of:
  Mini
  y
  x

Can you guess what the result will be? Here's the visualization:



In this case, Blend pushes y towards Mini in proportion to x.

An example of how you might use this construction of Blend is if you wanted something to be very unlikely to happen except under certain conditions. Suppose you are doing a psychological drama in which a troubled soul has to resist listening to his inner demons and taking revenge on someone who insulted him. An Inclination script for the Optionplot revenge might look like this:

```
plot revenge: Inclination:
   Blend of:
   Mini
   Sane_Psychotic of:
        ReactingActor
   Fearful_Angry of:
        ReactingActor
```

This script pushes ReactingActor's Inclination to plot revenge toward the minimum value, starting at their Sane_Psychotic value, in proportion to their level of fear or anger. That is, the highest possible value you will get is the value of ReactingActor's level of insanity, if they are in a state of unbridled outrage. Here are two different scenarios.

1. With this script, a very sane person (say, with a Sane_Psychotic ActorTrait of -0.25 or less) would be very unlikely to plot revenge no matter how angry they are. For instance, a Fearful_Angry mood of +0.8 would result in an Inclination of -0.32—a pretty low number.
2. On the other hand, if Jack the Ripper, with a Sane_Psychotic value of +0.9, got that angry, his Inclination to plot revenge would be highly likely (an Inclination of 0.71). I wouldn't want to hang around such a person!

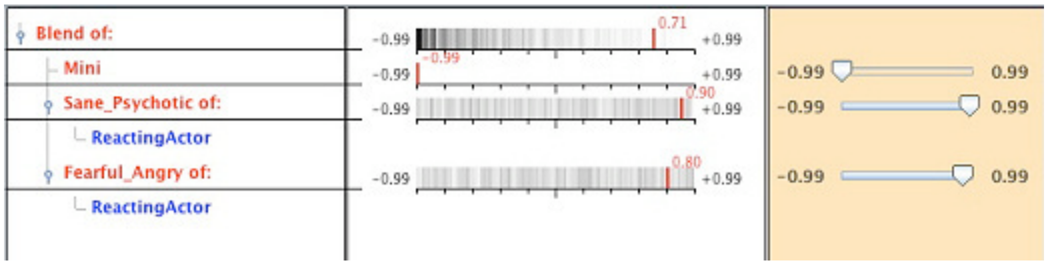
To test this, go to the Actor Editor and create a new Actor Core Trait called "Sane_Psychotic." Then go to the Verb Editor and create an Inclination script with the following construction:

```
Blend of:
   Mini
   Sane_Psychotic of:
        ReactingActor
   Fearful_Angry of:
        ReactingActor
```

Now click on Script > Scriptalyzer on the upper right of the scripting pane. Drag the slider for "Mini" to the lowest possible point on its slider. Create the Jack the

Ripper example provided in Scenario (2) above, with +0.90 for Sane_Psychotic of ReactingActor, and +0.80 for his Fearful_Angry mood. Do you get the same answer? (Remember, the final result of the script calculation always shows up in red on the top number line.)

Here is what it should look like if you do the example above, with an angry Jack the Ripper:



```
Blend of:                    -0.99  0.71   +0.99
  Mini                       -0.99  -0.99  +0.99   -0.99  0.99
  Sane_Psychotic of:         -0.99  0.90   +0.99   -0.99  0.99
    ReactingActor
  Fearful_Angry of:          -0.99  0.80   +0.99   -0.99  0.99
    ReactingActor
```

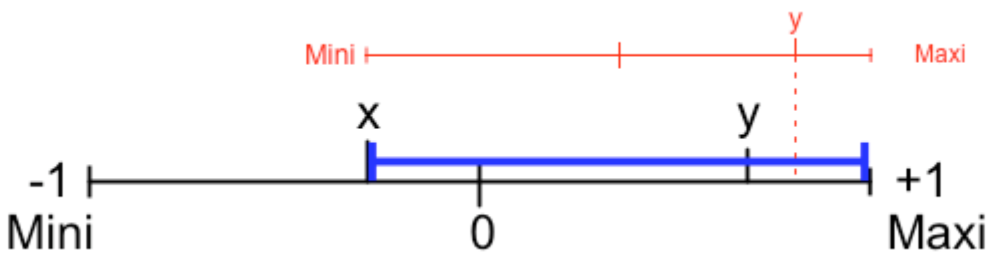
Then try out other values for Sane_Psychotic and Fearful_Angry, and see if the results you get are what you would expect them to be. Envision a character and choose a Sane_Psychotic value appropriate for his or her personality. Then set the second term to that value, and see what kind of Inclination you would get if they were neither fearful nor angry (which would correspond to a BNumber of 0.0, right in the middle of the scale.
Next, try changing their mood by moving the third slider bar into the negative (thus making them more fearful) or positive (making them more angry), and watch what happens to their Inclination.

Blending from Maximum to a Value

OK, now let's try something else. What if we use Maxi in place of Mini in the above example:

Blend of:
  Maxi
  y
  x

Here's the visualization:



Or, to express it verbally, Blend starts at Maxi and moves towards x in proportion to y.

You would use this construction if you want to stack the deck in favor of making something likely to happen, but you want to mediate it with key personality, relationship, or other Attributes that might make a difference in rare cases. For instance, suppose you are creating a mystery storyworld and an Actor has to decide whether to deceive the Protagonist (ThisSubject).

Let's assume ReactingActor, on the basis of *turnabout is fair play*, bases their decision on how deceitful ThisSubject appears to be (note that this would be the *inverse* of PFalse_Honest), mediated by their confidence in ThisSubject's perceived level of honesty.

Let's make it so that the ReactingActor is predisposed toward deceiving the detective (perhaps they are protecting a lover or family member, and don't trust the detective to play fair). They base their decision how deceitful ThisSubject appears to be (note that this would be the inverse of PFalse_Honest), mediated by their confidence in their assessment of ThisSubject's deceitfulness.

```
decide to deceive:Inclination
  Blend of:
      Maxi
      BInverse of:
          PFalse_Honest of:
              ReactingActor
              ThisSubject
      CFalse_Honest of:
          ReactingActor
          ThisSubject
```

The script is saying this: if ReactingActor trusts ThisSubject a lot, and has a high degree of confidence in that assessment, ReactingActor will be unlikely to deceive ThisSubject. For instance, if ReactingActor perceives ThisSubject's honesty as a strong 0.4, and a very high degree of confidence in that assessment, say a CFalse_Honest of 0.8, their likelihood to deceive ThisSubject is pretty low: -0.25. Otherwise, ReactingActor is very likely to deceive ThisSubject.

As with the Jack-The-Ripper Mini-Blend example above, start with a Blend script that looks like this:

```
Blend of:
  Maxi
  BInverse of:
```

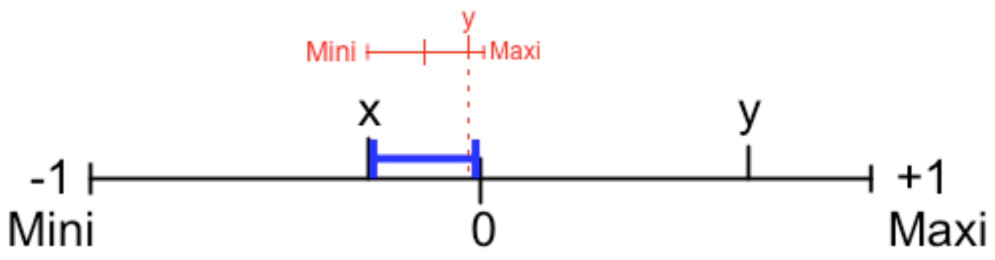
PFalse_Honest of:
    ReactingActor
    ThisSubject
CFalse_Honest of
    ReactingActor
    ThisSubject

Set the first slider to the maximum possible amount, and move the second and third sliders in accordance with an Actor's perception of another's deceptiveness, and their confidence in that evaluation, and see if you get the final result you expect for in Inclination. Does the Inclination to deceive go up and down as you would expect it?

Blending Between A Value and Zero

Now let's try another variation:

Blend of:
  0.0
  x
  y



Blend starts at 0.0and moves toward x in proportion to y.

The simple rule is this:

Blend starts with the first argument and moves toward the second argument in proportion to the third argument.

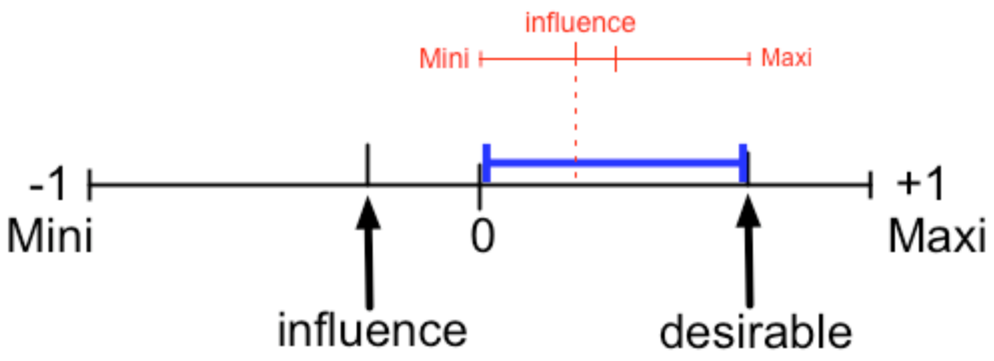
Blending Between Two Values, Using Zero

Here's a variation on Blend that is used in BoP2K: We have a country evaluating a deal. it has to evaluate two factors: the value of what it's getting, and the value of what it's giving. The value of what it's getting depends on two factors: how much it desires that outcome (PUndesirable_Desirable), and the likelihood that it will happen (PInfluence). So the Script looks like this:

```
Blend of:
   PUndesirable_Desirable of:
        ReactingActor
        This5Prop
   0.0
   PInfluence of:
        ThisSubject
        Owner of:
             This5Prop
```

There are three arguments to the Blend Operator. The first argument represents how much the ReactingActor desires This5Prop. However, the deal doesn't actually deliver This5Prop—it means only that ThisSubject will attempt to induce the Owner of This5Prop to deliver it. So we must take into account the likelihood that ThisSubject will succeed in his attempt. That depends upon how much influence that ThisSubject has over the Owner of This5Prop.

Now, if ReactingActor were 100% certain that he'd get This5Prop, then the net desirability would be the desirability of This5Prop—but in fact the chances of getting This5Prop are less than 100%, so the net desirability is less than the actual desirability of This5Prop. How much less? That depends on the probability of success. If the probability of success is zero, then the net desirability is zero—ReactingActor won't get anything out of the deal. So we BlendPUndesirable_Desirable with 0.0, and the BlendingRatio will be the likelihood of success (PInfluence).



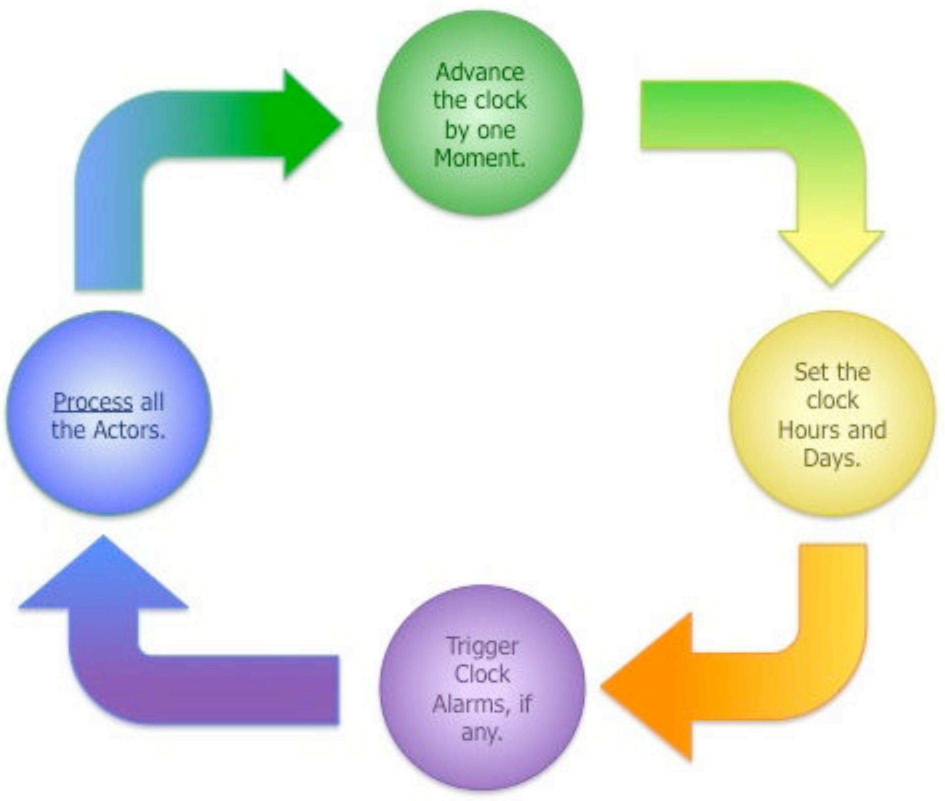
Remember, Blend starts with the first argument, and moves from the second argument towards it, in proportion to the third argument. Hence, the result will move from PUndesirable_Desirable toward 0.0 in proportion to PInfluence.

Next Tutorial: Engine Operation Overview
Previous Tutorial: All You Need Is Blend (part 1)

Storytron: Engine Operation Overview

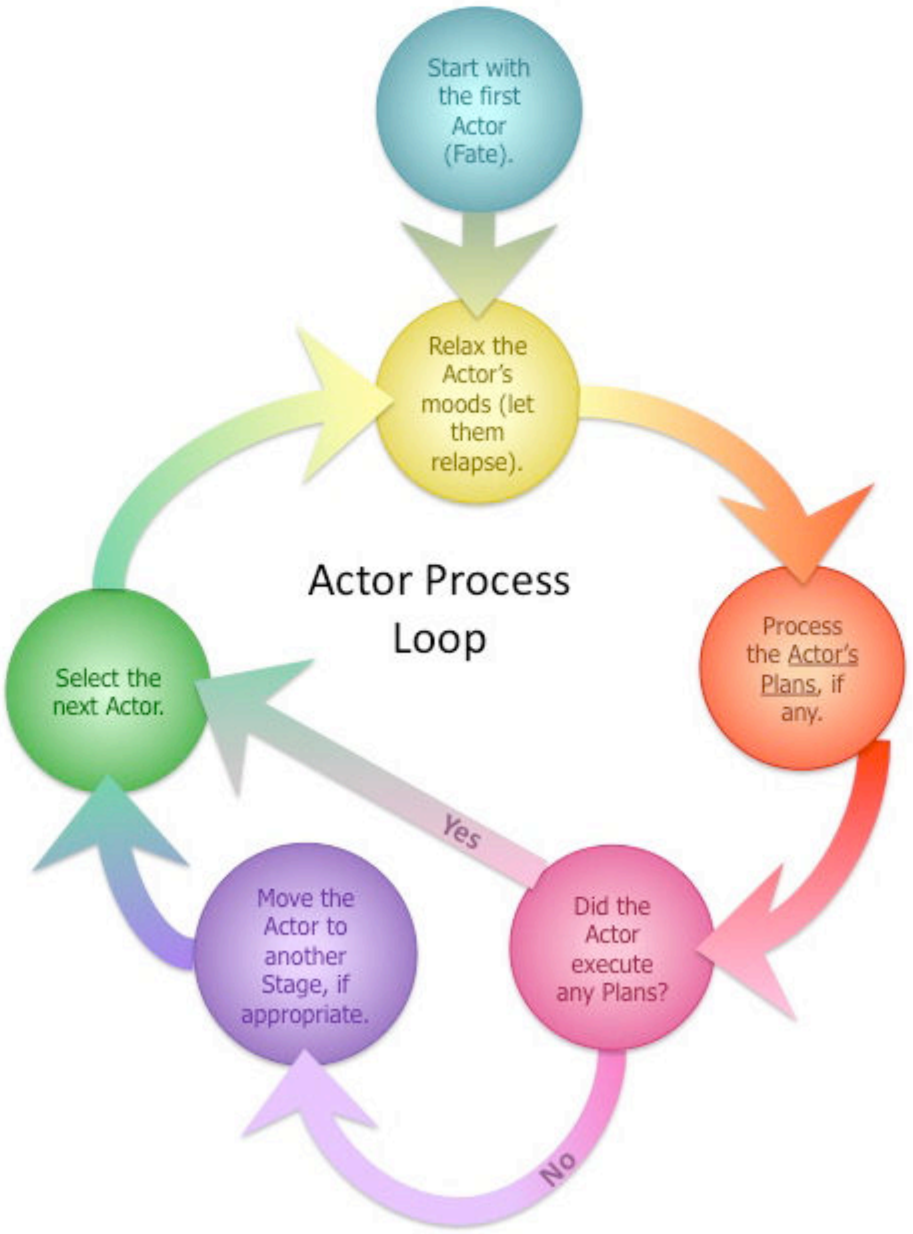
Last edited by Bill Maya 6 days ago

This series of flow charts depicts, in deepening levels of detail, the processing decisions made by the Story Engine in running the storyworld. First is the main engine loop:

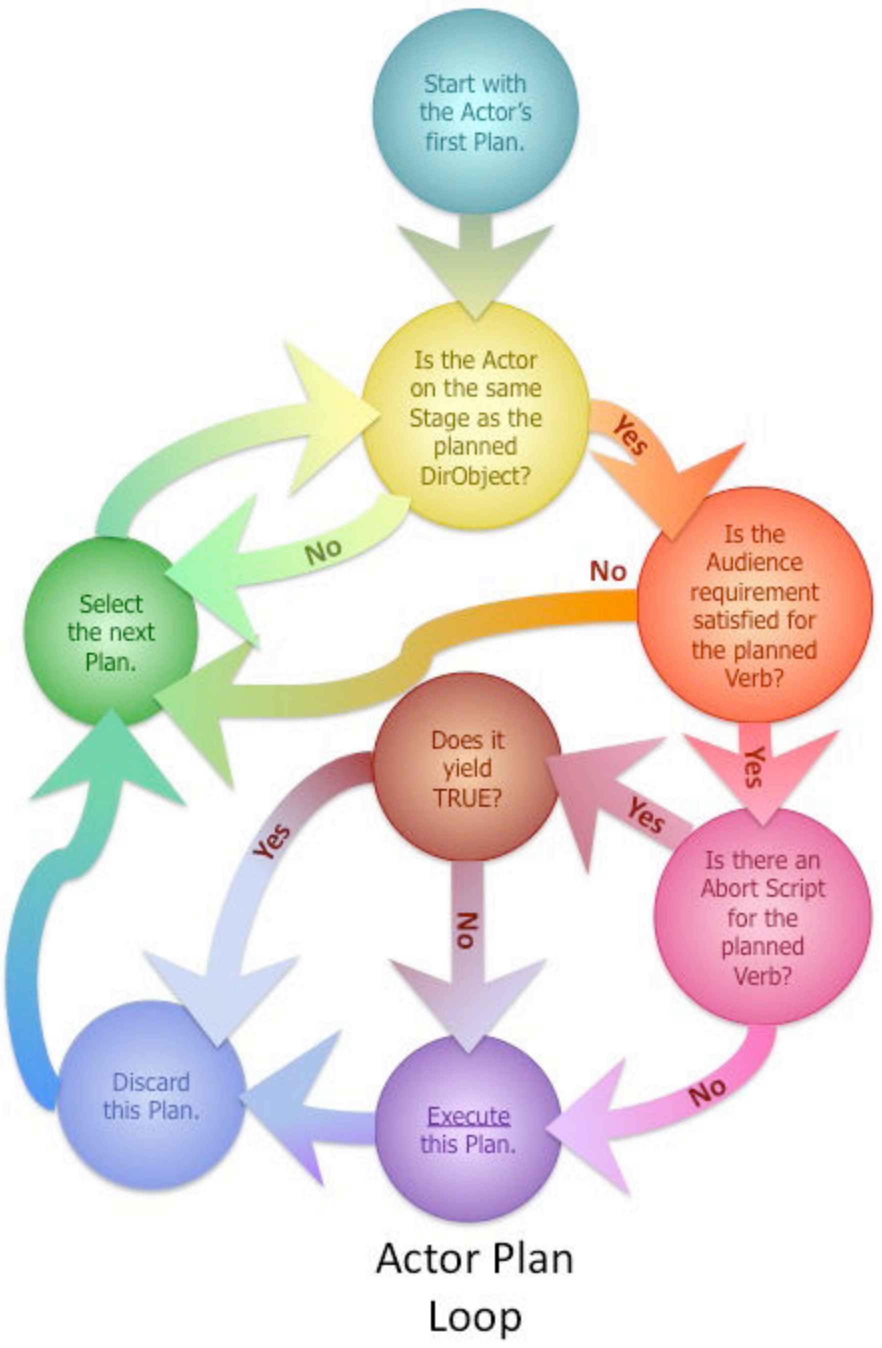

Advance the clock by one Moment.
Set the clock Hours and Days.
Trigger Clock Alarms, if any.
Process all the Actors.

Main Engine Loop

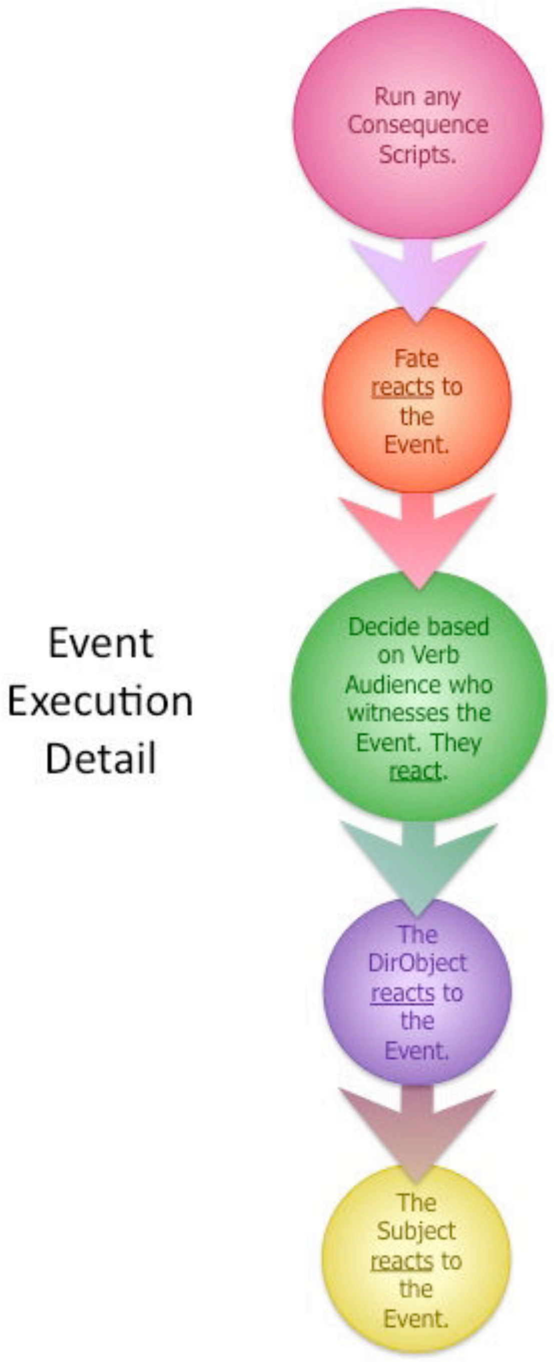
The Story Engine processes all Actors' actions and travel:


Start with the first Actor (Fate).
Relax the Actor's moods (let them relapse).
Actor Process Loop
Process the Actor's Plans, if any.
Did the Actor execute any Plans?
Yes
No
Move the Actor to another Stage, if appropriate.
Select the next Actor.

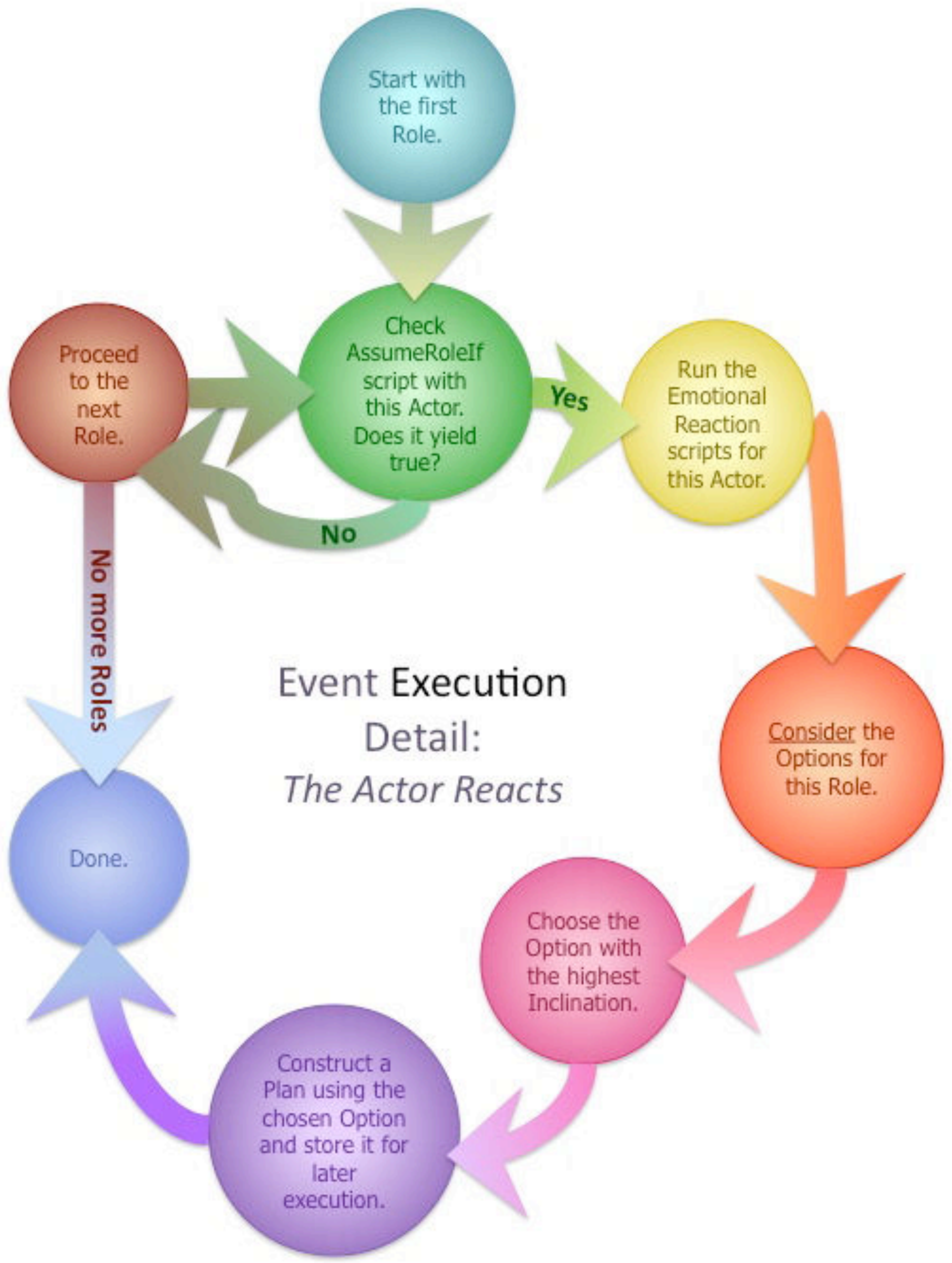
For each Actor, the Story Engine makes a cascading set of decisions and processes his or her Plan(s):


Start with the Actor's first Plan.
Is the Actor on the same Stage as the planned DirObject?
Yes
No
Select the next Plan.
Is the Audience requirement satisfied for the planned Verb?
No
Yes
Is there an Abort Script for the planned Verb?
Yes
Does it yield TRUE?
Yes
No
No
Discard this Plan.
Execute this Plan.
Actor Plan Loop

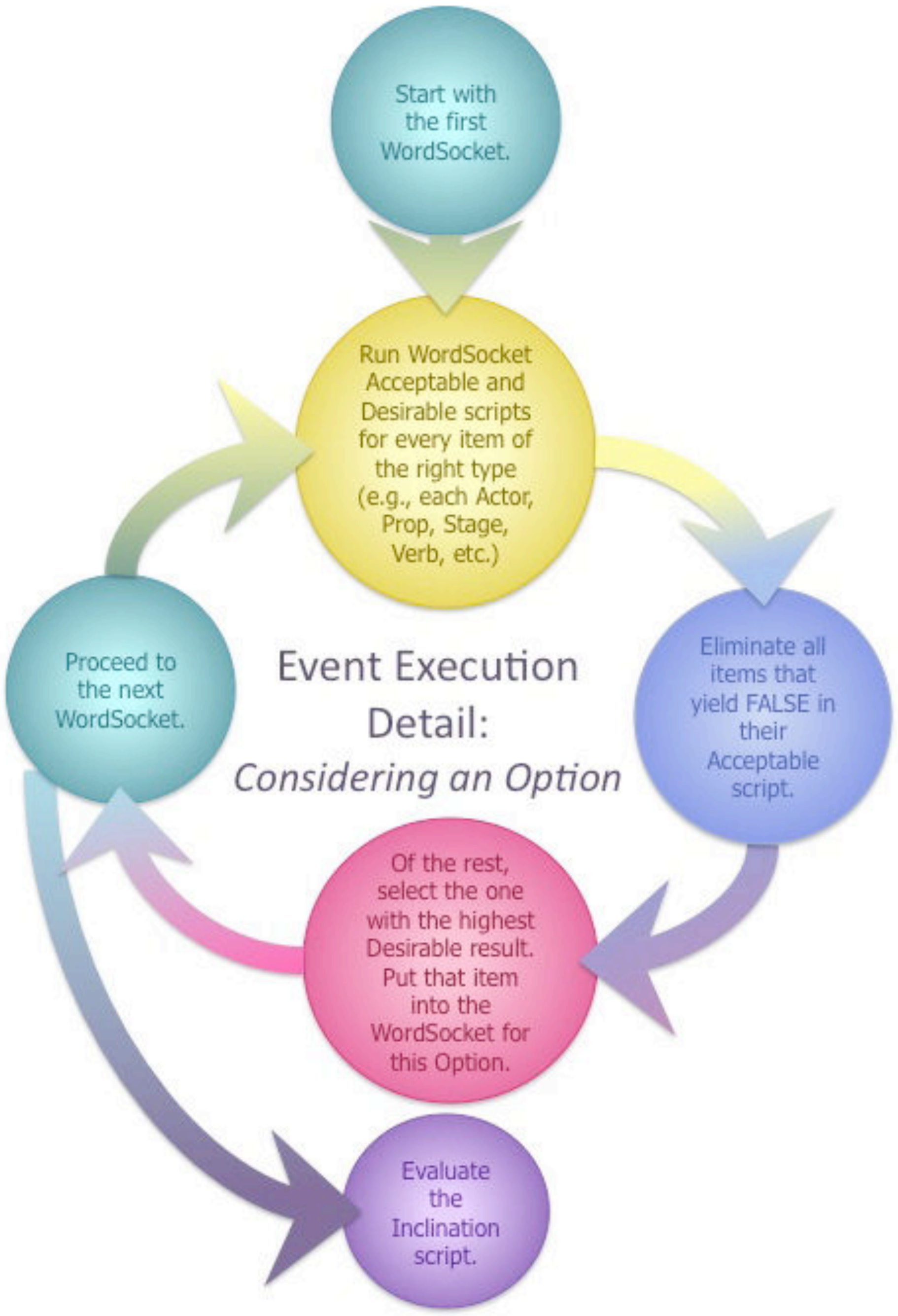
To execute a Plan entails the following series of steps:


Event Execution Detail
Run any Consequence Scripts.
Fate reacts to the Event.
Decide based on Verb Audience who witnesses the Event. They react.
The DirObject reacts to the Event.
The Subject reacts to the Event.

The Actor's Reaction involves a set of decisions and calculations, as follows:


Start with the first Role.
Check AssumeRoleIf script with this Actor. Does it yield true?
Yes
Run the Emotional Reaction scripts for this Actor.
Consider the Options for this Role.
Choose the Option with the highest Inclination.
Construct a Plan using the chosen Option and store it for later execution.
Done.
No more Roles
Proceed to the next Role.
No
Event Execution Detail: The Actor Reacts

The Story Engine performs the following steps for each Option under consideration by a ReactingActor:


Start with the first WordSocket.
Run WordSocket Acceptable and Desirable scripts for every item of the right type (e.g., each Actor, Prop, Stage, Verb, etc.)
Eliminate all items that yield FALSE in their Acceptable script.
Of the rest, select the one with the highest Desirable result. Put that item into the WordSocket for this Option.
Proceed to the next WordSocket.
Evaluate the Inclination script.
Event Execution Detail:
Considering an Option

And that's all there is to it! All right, that's a fib...believe it or not, this is a simplified version of what happens. But it gives you the big picture of what the Engine does, in what order, to make things happen in your storyworld.

The next tutorial contains this process in written form with some more detail, including Actor travel.

Downloadable versions of these flowcharts can be found in Downloads section.

Next Tutorial: Engine Operation Detail
Previous Tutorial: All You Need Is Blend (part 2)

Storytron: Engine Operation Detail

Last edited by Bill Maya 6 days ago

The Engine runs on a cycle. What actually happens can be very complicated, but here is a more thorough (though still simplified) explanation of the process that the Engine uses.

```
Top of Cycle:
    Advance the clock by one Moment.
    Set the clock Hours and Days
    Check to see if nothing has happened for at least 10 Moments;
        if so, terminate the story.
Check the clock time; if it's time to do so, trigger the ClockAlarm.
    For each and every Actor, starting with Fate, do the following: (LOOP)
        Relax the Actor's moods (let them relapse towards normalcy)
        For each and every one of the Actor's Plans, do the following: (LOOP)
Check that the Plan's Execution Time has elapsed.
Check that the Actor is on the same Stage as the Plan's DirObject;
                if not, skip this Plan.
            Check the Audience requirement for the Plan's Verb;
                if the Audience requirement is satisfied by the situation, then proceed;
                 otherwise, skip this Plan.
            Check for an Abort Script for this Verb; if there is one, execute it;
                 if it returns "true," then abort executing this Plan and throw it away.
            If we've gotten this far without skipping this Plan, then execute it:
                store the Plan into the HistoryBook as an Event.
                execute any Consequence Scripts for this Event
                if this Verb is "depart for," do some special things to remove the Subject
                    from his current Stage.
                if this Verb is "arrive at," do some special things to place the Subject at his
                    destination.
                Have Fate react to the Event.
                based on the Audience requirement for the Verb, decide who on the Stage
                    actually witnesses the Event.
                Have each witness in turn react to the Event.
                Have the DirObject react to the Event
                Have the Subject react to the Event.
                Throw away the Plan.
            Do not execute any more Plans for this Actor (unless the Actor is Fate).
        End of Plans loop
    Did the Actor execute any Plans? If so, then proceed to the next Actor. If not, do this:
        Consider whether to go to another Stage:
            Does the Actor have a previously defined TargetStage? If so, go there.
            If not, sort through the Plans and find the most important Plan.
            If there is one, go to the Stage where the planned DirObject is
                (so we can execute the Plan on the DirObject).
            If the Subject can't do that, then pick a new Stage to go to based on:
                how long the Actor has been sitting around at the current location.
                how Unwelcoming_Homey another Stage is for this Actor
    End of Actor loop
End of main loop—go back to the top of the loop
```

If it's time to terminate the story, set a Plan for Fate to execute "Fatepenultimate verbProtagonist." This system Verb will then lead to the very last Event, "Fatehappily ever after Protagonisthow much." You, the author, may insert intervening Verbs between penultimate verb and happily ever after; however, if you do so, see the cautionary notes in System Verbs

You may also write WordSocket scripts for the Quantifier in the very last sentence, "Fatehappily ever after You this much: [Quantifier]," to reflect how well or poorly your player did. If you wish, however, you can just leave this at its default setting of "medium," or you can remove the Quantifier. To remove the happily ever after Quantifier, go to System Verbs: happily ever after: Properties and eliminate 4Quantifier from the list of WordSockets.

Next Tutorial: Deikto
Previous Tutorial: Engine Operation Overview

Storytron: Deikto

Last edited by Bill Maya 6 days ago

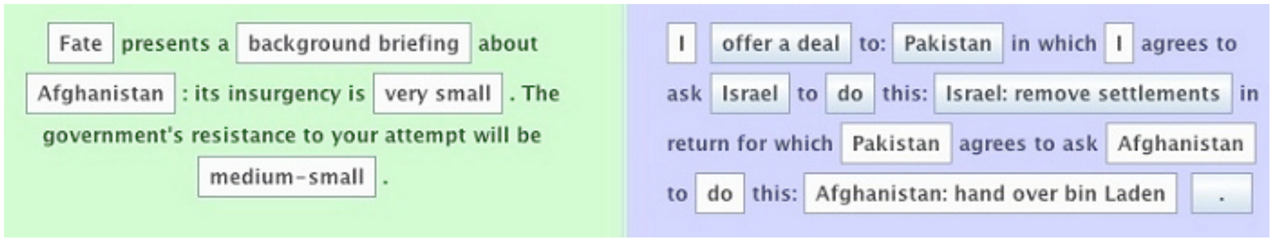
Interactive storytelling is controlled by the language of interaction. The basic rules of interaction are these:

- If you can't say it or hear it, you can't interact.
- You can't say much with the devices we use for computer input (mouse, keyboard, etc.).
- The obvious solution is language, but real language can't be done on a computer.
- It's impossible because of the Sapir-Whorf hypothesis: our language mirrors the reality in which we live.
- Reality is too big to fit inside a computer.
- Ergo, we can't fit natural language (which mirrors reality) into the computer.

The solution is to create a toy language to go along with the toy reality of interactive storytelling.

Usually we define the toy reality, then try to make a language to fit it. That never works, because language is itself very complicated. The Deikto solution is to make the language and the reality one and the same. Define the reality by defining the words of the language in terms of what they do.

Deikto (DEEK-toh) is a system for generating toy languages. It provides the grammar, the authors provides the words that plug into that grammar. Deikto appears in Storyteller, the software you used to play a storyworld. This is an example of a Deikto sentence:



Fate presents a background briefing about Afghanistan : its insurgency is very small . The government's resistance to your attempt will be medium-small .

I offer a deal to: Pakistan in which I agrees to ask Israel to do this: Israel: remove settlements in return for which Pakistan agrees to ask Afghanistan to do this: Afghanistan: hand over bin Laden .

Deikto is displayed to the player in Storyteller, the storyworld playing software. Each Deikto word has a form the author has to fill out to define it. The form depends on the type of word (Actor, Stage, Prop, etc). You will find these forms in SWAT, (StoryWorld Authoring Tool), the software you use to create a storyworld.

Next Tutorial: Sappho
Previous Tutorial: Engine Operation Detail

Storytron: Sappho

Last edited by Bill Maya 6 days ago

Sappho (*SAF-foh*) is the scripting language for Storytronics. It is the language you use in SWAT to fill out the forms for the Deikto words that the player sees. It's a very strange—and very powerful—scripting language, which is designed for writers and other storytelling professionals. It has many special features:

- Syntax errors (saying something that confuses the language) are impossible.
- Initialization errors (forgetting to set things up properly) are impossible.
- Point-and-click editing (no typos to cause big trouble)
- Argument prompts (when you need to fill in a blank, it tells you)
- Scripting is organized in a tree structure (easier on the eyes)
- Only legal Operators are made available at any given moment

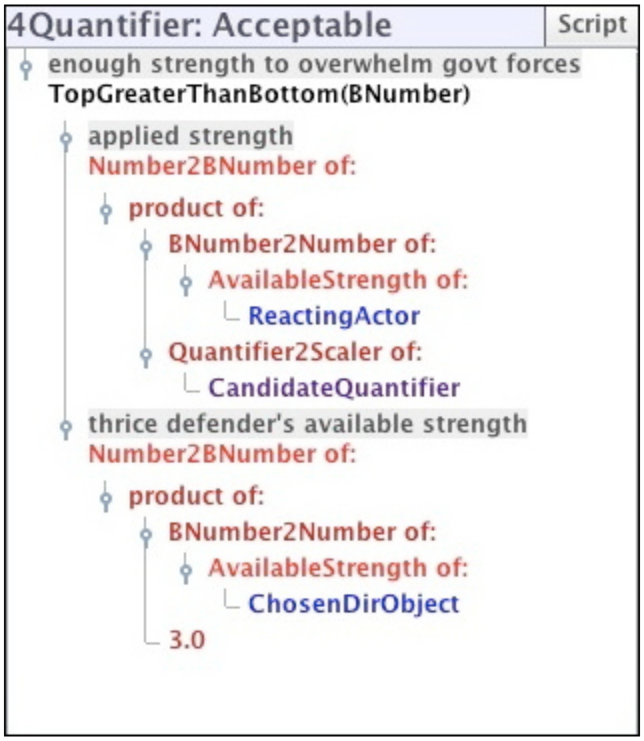
Color-coded strong data typing:

Bright Red	Bounded or Unipolar Number
Dark Red	Regular Number
Blue	Actor
Green	Verb
Black	Boolean
Magenta	Prop
Orange	Stage
Cyan	Event
Blue-Green	Trait
Purple	Quantifier
Gray	Comment

This means that you can't accidently mix apples and oranges (or Props and Actors) because it won't let you put a Prop into a slot meant for an Actor, or vice versa. And the colors make it easy to tell the apples from the oranges.

There are about 600 Operators available—most are simple. There is no "flow control" (branching, looping, or subroutines) as is common in most programming languages. (But there are *implicit* looping Operators.) Run-time errors generate Poison —a system that protects you from minor scripting mistakes.

This is an example of a Sappho script:



```
4Quantifier: Acceptable                              Script
enough strength to overwhelm govt forces
TopGreaterThanBottom(BNumber)
    applied strength
    Number2BNumber of:
        product of:
            BNumber2Number of:
                AvailableStrength of:
                    ReactingActor
            Quantifier2Scaler of:
                CandidateQuantifier
    thrice defender's available strength
    Number2BNumber of:
        product of:
            BNumber2Number of:
                AvailableStrength of:
                    ChosenDirObject
            3.0
```

Next Tutorial: Boxes
Previous Tutorial: Deikto

Storytron: Boxes

Last edited by Bill Maya 6 days ago

These are temporary places to store intermediate values; they save you time and tedium. There are three kinds of boxes:

- Verb boxes
- Role boxes
- Global boxes

Boxes hold onto a particular item for you within a given scope. (For instance, a VerbActorBox will hold onto a specific Actor designation for use throughout the Verb's scripts. A RoleEventBox will hold onto a particular Event for use throughout that Role.) Boxes are useful when you have a fairly hairy script, or a script that involves HistoryBook lookups—e.g., PastActor of an Event with a bunch of different parameters, which you want to use multiple times.

There are Boxes for each of the major data types: Actors, Props, Stages, Events, and Verbs, as well as one BNumber box.

There are also four Global Box sets, again with one Box for each of the listed data types. These Boxes never forget the values you put into them; if you store something into a Box during one calculation, you can come back to it much later in a completely different calculation and it will still have that value in it.

Here is an example for how to use Boxes. It is not uncommon to use the same script for an Inclination and the Desirable script for a key WordSocket. Suppose you have a character who might want to run a con, and must choose both whom to run it on, and how likely he is to run the con, based on how gullible the intended target is. In a case like this, you can use a single script to both select a desired DirObject for the scam, and to determine how likely the ReactingActor is to run the con on that DirObject.

These two uses are confined to a single Role—conman—so the appropriate Box to use would be a RoleBox. In fact, in this example, you need two Roleboxes: one to pick the proposed DirObject, and another to give the BNumber corresponding to ReactingActor's perception of the DirObject's gullibility.

Before you can use a Box in your scripts, you must first fill it. You do this either in Consequences (for a VerbBox) or Emotional Reactions (for a RoleBox). Once it is filled, it becomes available throughout the rest of that Verb or Role, respectively.

In our example, the FillRoleActorBox script will look like this:

PickBestActor:

```
  true
  BInverse of:
      pGullible_Skeptical of:
          ReactingActor
          CandidateActor
```

This picks the Actor who the conman believes is the most gullible (note that to get the most gullible Actor, we needed to invert the Attribute).
Next you would choose FillRoleBNumberBox. The script would look like this:

```
pGullible_Skeptical of:
  ReactingActor
      RoleActorBox
```

This fills the Box with a BNumber corresponding to the conman's perception of his chosen target's gullibility.

Now you can use the following script for the Inclination to run the con:

```
RoleBNumberBox
```

For the DirObject Acceptable WordSocket, you would use:

```
AreSameActor:
  CandidateActor
  RoleActorBox
```

Ta-da! No muss, no fuss.

Where Boxes really save you time and effort is when you have a script that involves numerous Attributes, Lookups, and PickBests, which you use multiple times within a given Role or Verb. If you intend use a script more than once, use Boxes.

Here's how you know when to use a VerbBox versus a RoleBox. If the item you want to use in multiple places is confined to a single Role, you must use a Role box. If you intend to use that Box throughout the Verb, use a VerbBox.

Global Boxes work in a similar way to VerbBoxes and RoleBoxes, but they apply to the storyworld as a whole. Be forewarned: it is all too easy to get into trouble using Global Boxes. We strongly urge that, if you want to use any Global Boxes, you decide at the very beginning exactly what that Global Box will hold and never, ever change that in mid-stream. Otherwise, you'll get confused about the meaning of the Global Box and create monster headaches for yourself.

Previous Tutorial: Sappho