v2.0 Quest Design Bible

Created	@February 9, 2024
Creator	Chandler Bastin
Tags	Project Cascade Roleplay

Table of Contents

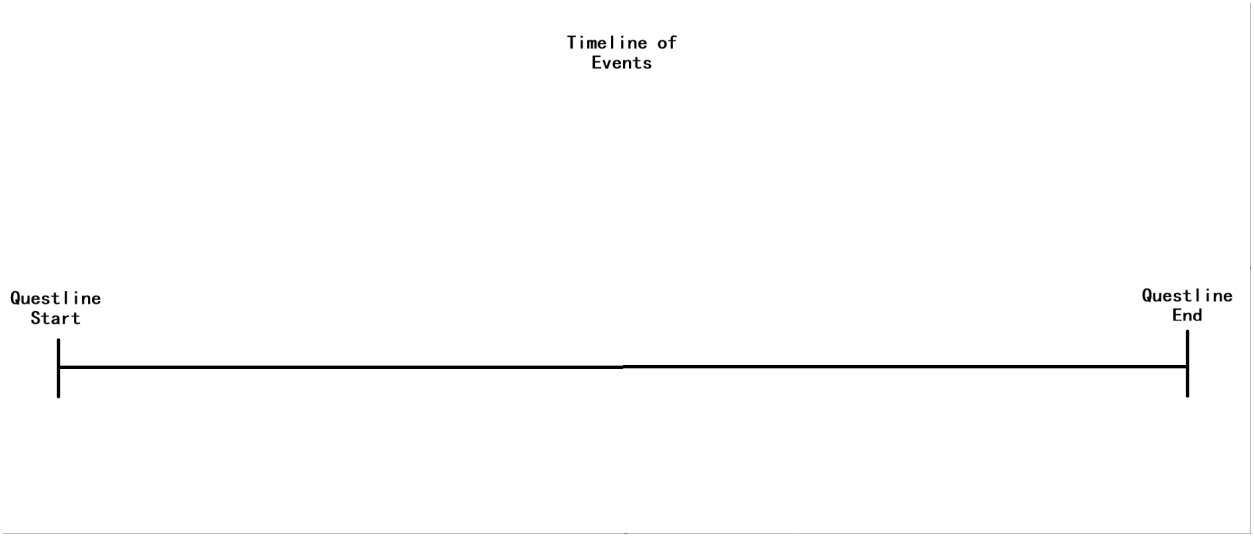
Overview

This document covers the basic questline development procedure, specifically-

1. Fundamentals of a MINDFLUX Questline
2. How to prepare a quest.
3. Variable Explanation
4. Brief Design Explanation
5. Variable Implementation in Articy

Basic Questline Structure Explorations

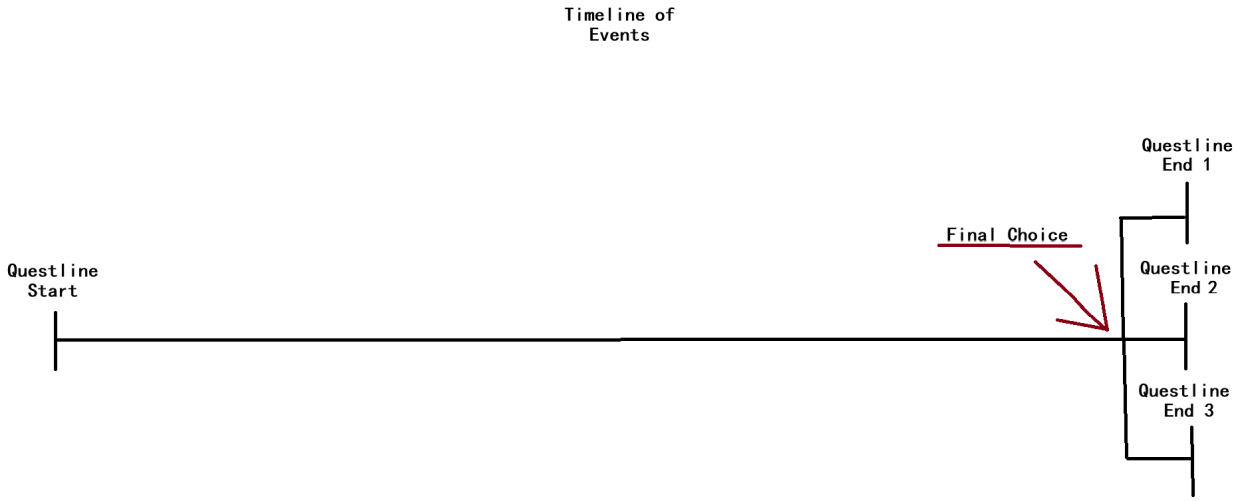
In MINDFLUX, questlines can be represented by a timeline of events. In its most basic form, quests resemble a straight line with a beginning and end:



Linear Questline

Above is what we would call a linear quest. The player has a defined start, travels along a set path, and witnesses a defined end. These types of quests are usually seen in MMO's or adventure games, such as the Last of Us.

Occasionally, developers will add variance to their titles in the form of multiple endings. This most often occurs in a decision at the tail end of the game:

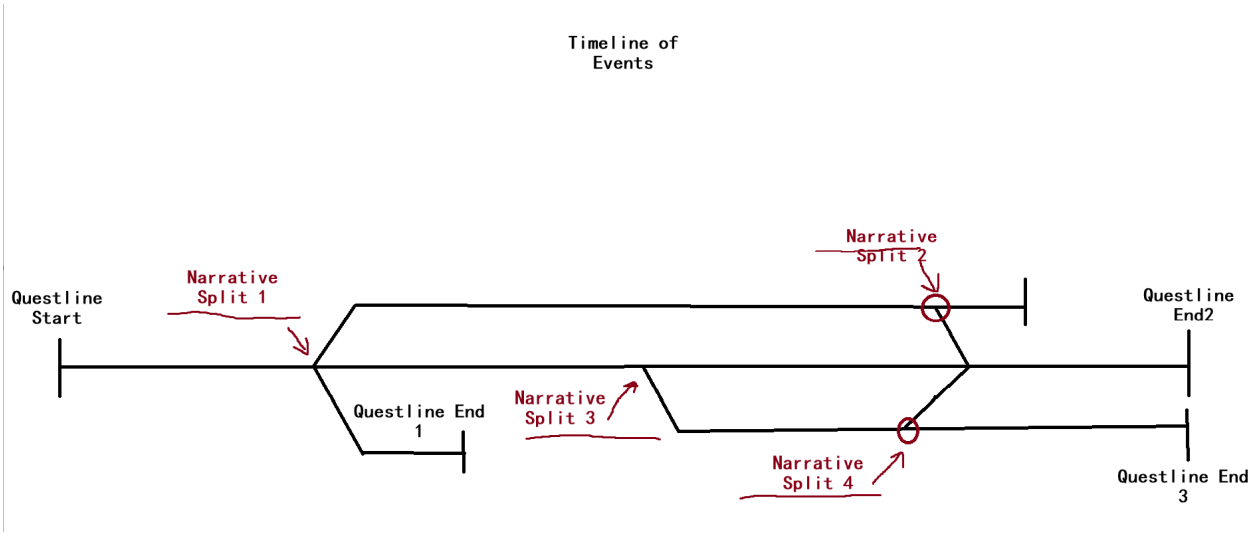


Final Split Questline

In this example, events play out the same until a final decision, which determines the ending. Games like Ghosts of Tsushima and Infamous use this questline type.

It’s still linear, but with the addition of a final split, thus it’s name, the Final Split Questline.

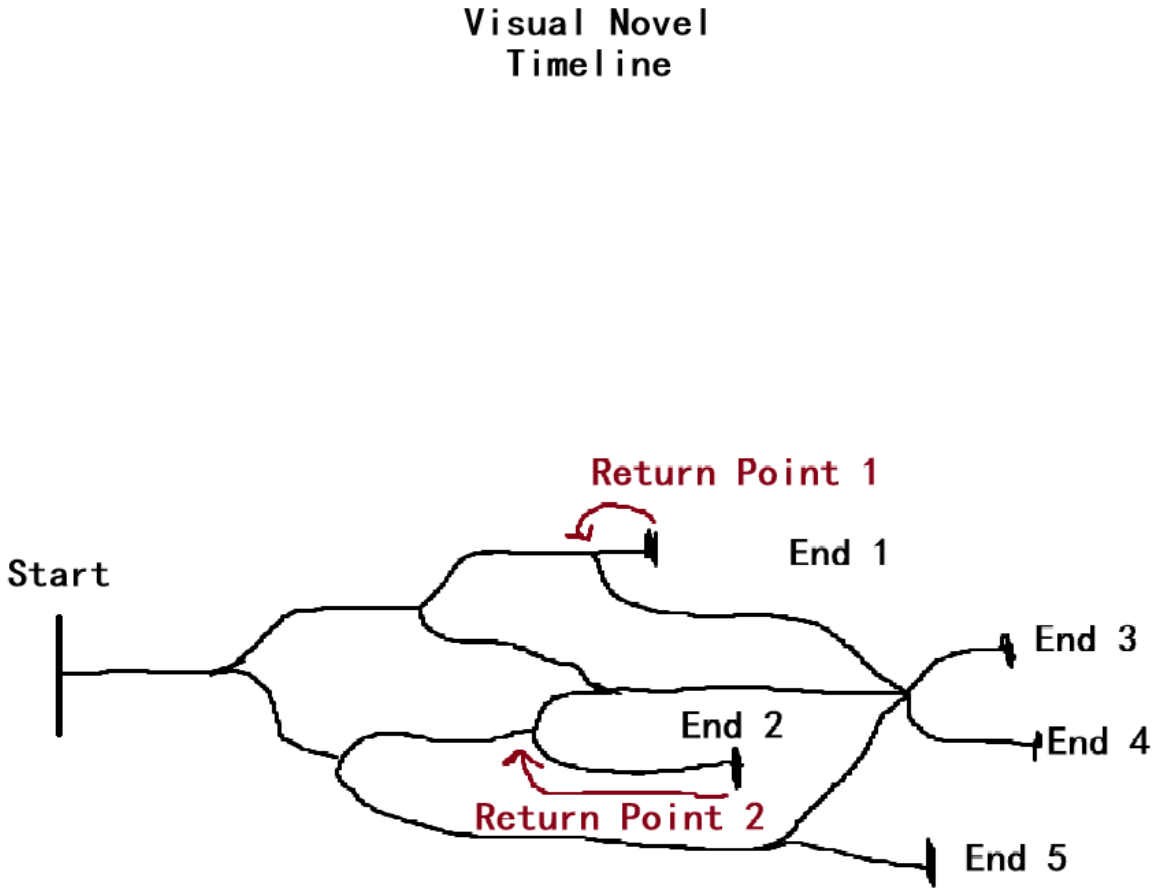
Some games add decisions throughout the questline that can affect the ending, or place the final choice earlier in the story:



The “Choose Your Own Adventure” Questline

In the above example, there are splits that equate out to three possible endings. In some cases, right before and ending, the developer will place a final choice that can return you to the “main” or “intended” timeline of events, indicated by narrative splits 2 and 4. Books like the “Choose your own Adventures” series implement this style of game design.

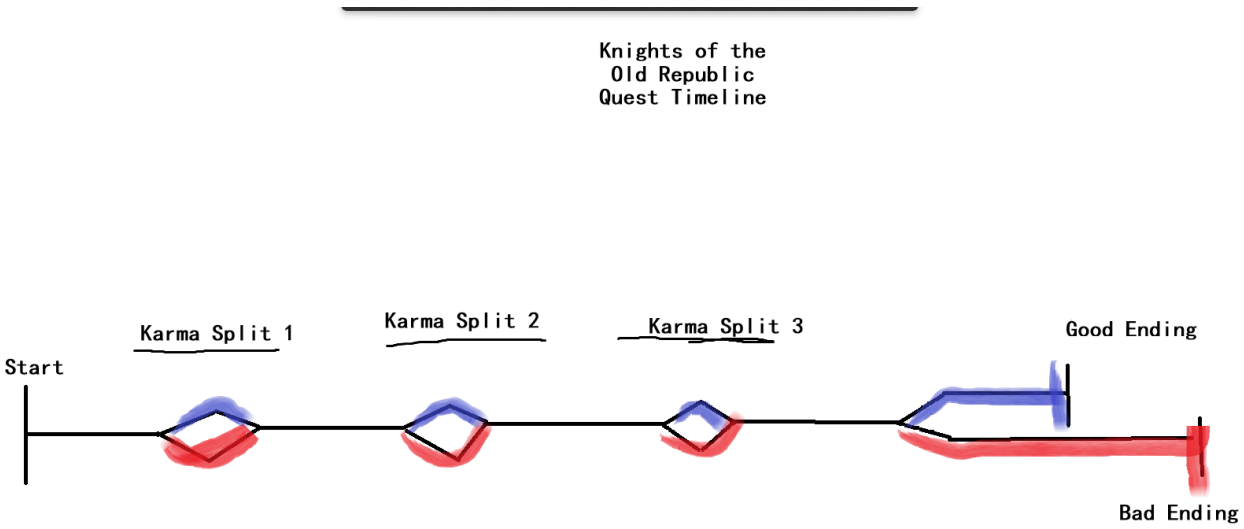
Many games have developed variances on the “choose your own adventure” style, either by simplifying it or adding complexity.



Visual Novel Questline

Visual novels often employ what's called a “return point”, where players can receive a less-than satisfactory ending, and be deposited back at the moment the decision was made. They also have splits that allow the player to return back to different narrative paths.

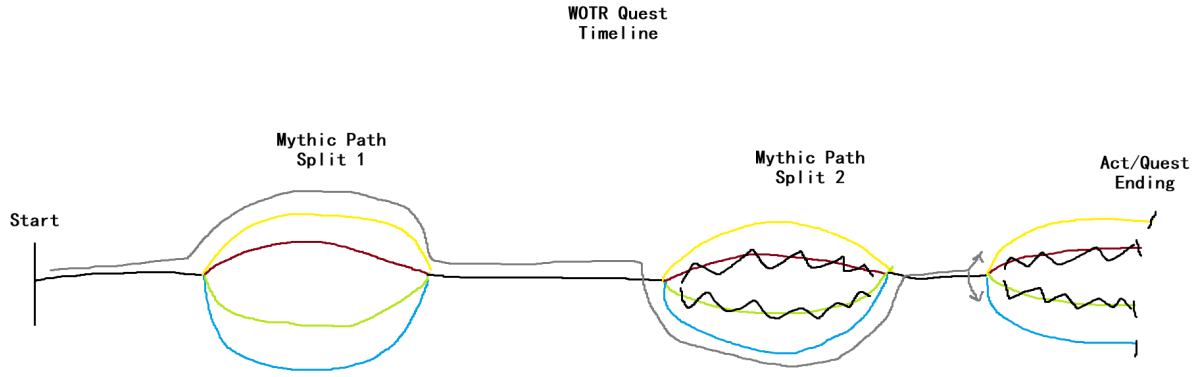
Obviously, return points aren’t necessarily the best for narratives found in CRPGs, though some have managed to use them to great effect (Disco Elysium). So how does a game within our genre deal with questlines?



Karma Decision Questline

In KOTOR, instead of using splits, the writers provided numerous moments within the questline where the player can make a KARMA decision, swaying their character toward the dark or light side. These decisions more often than not make little to no impact on the ending of the quest. Eventually, the player is given a final decision which can usually be made regardless of the players Karma. This is a free-flow system, and there is *rarely* blockers to the player making whatever decision they want. Their decision to do this allowed the writers the freedom to create more quests without the need for massive balancing changes based on feedback at the cost of roleplay variety and nuance.

But what if this system was taken to a more extreme level? Let's take a look at Wrath of the Righteous, a game that takes the fundamentals shown in BioWare games and takes them to a whole new level.



The “Mythic Path” Questline

Instead of creating a massive, complex flow, WOTR creates a hybrid quest timeline that incorporates the simplicity and manageability of linear quests with roleplaying diversity. For the bulk of the story, the player goes down the same questline, in this case it’s “Stop the Demon Invasion of Golarion”. At key moments in the story, the player will be granted a choice on *how* they go about stopping the invasion. Depending on their decision, they are blocked from other paths. In the above example, the player, indicated in grey, chose the yellow path at their first split. This blocks off the red and green paths entirely, but the blue path can still be chosen. At the second split, the player chooses the blue path, granting them the choice of either the yellow or blue paths endings.

So that’s all fine and good, but what does that have to do with our game? Let’s take a look at how a quest timeline would look in MINDFLUX.

MINDFLUX Quest Flow Breakdown

> *“It’s not what you achieved, it’s how you achieved it that scares me.” -Unknown*

MINDFLUX questlines take inspiration from KOTOR as well as Wrath of the Righteous to create our own unique flow.

In our game, we don't have karma or "mythic paths". Our narrative has a particular focus on the intricacies of an NPC's character, and more importantly, how a player interacts with that NPC using the Cascade roleplaying system. Therefore, our narrative splits have to reflect tangible choices made in relation to that character, much like a visual novel.

i.e. "Discovering Hobbes' Secret" rather than "Dark Side Evil Choice"

On top of this, our game can't rely on waypoints, mini map's, fast travel, or other non-contextual systems to orient the player. We also have a heavy focus on self-discovery in the form of investigation, encouraging a more "non-linear" method of quest design.

If planned poorly, quests can become muddled, losing all context for the player.

I've taken the liberty of charting and naming each of the major elements that comprise our questline flow from a design perspective. These terms are as follows:

Glossary

1. Encounter Split- A hub of sorts where the player can choose to pursue different leads.
2. Major Split- A narrative beat where the player can make an ending-defining decision.
3. Minor Split- A narrative beat where the player can make a non-ending defining decision.
4. Split Goal- The primary goal of a split that must be accomplished regardless of choices made.
5. Variable- A symbol that provides short hand for decisions made by the player.
6. Minor Variance- A minor variance in the quest that doesn't equate to ending changes.

To create a questline flow for MINDFLUX, we will need to complete the following steps:

1. Identify each Major Split and total required quest bubble.
2. Identify primary goals of each quest bubble.
3. Identify each variable path.
4. Assign each path a variable.
5. Create endings in line with paths.

Let's walk through the above steps in an example questline:

STEP 1 Identifying each Quest Bubble

Let's say that we're designing a quest where the player is trying to determine if Hobbes' had something to do with a murder that took place in the quarry. We go through the process of workshopping the idea, modifying the narrative inflection points, doing character studies, the whole smash. We are ready to start charting out the player flow.

First thing we need to do is identify every location where the player can make a decision on what to do first.

In this questline, we decided in the prewriting phase to have two parts, one where the player can grill Hobbes', and another where the player can find clues from the two witnesses of the crime, Bys and the Arms Dealer.

We noticed then, that the player could, after a short introduction to the quest, decide which to accomplish first. From this consideration, we've determined there are two Major Splits that appear in the narrative:

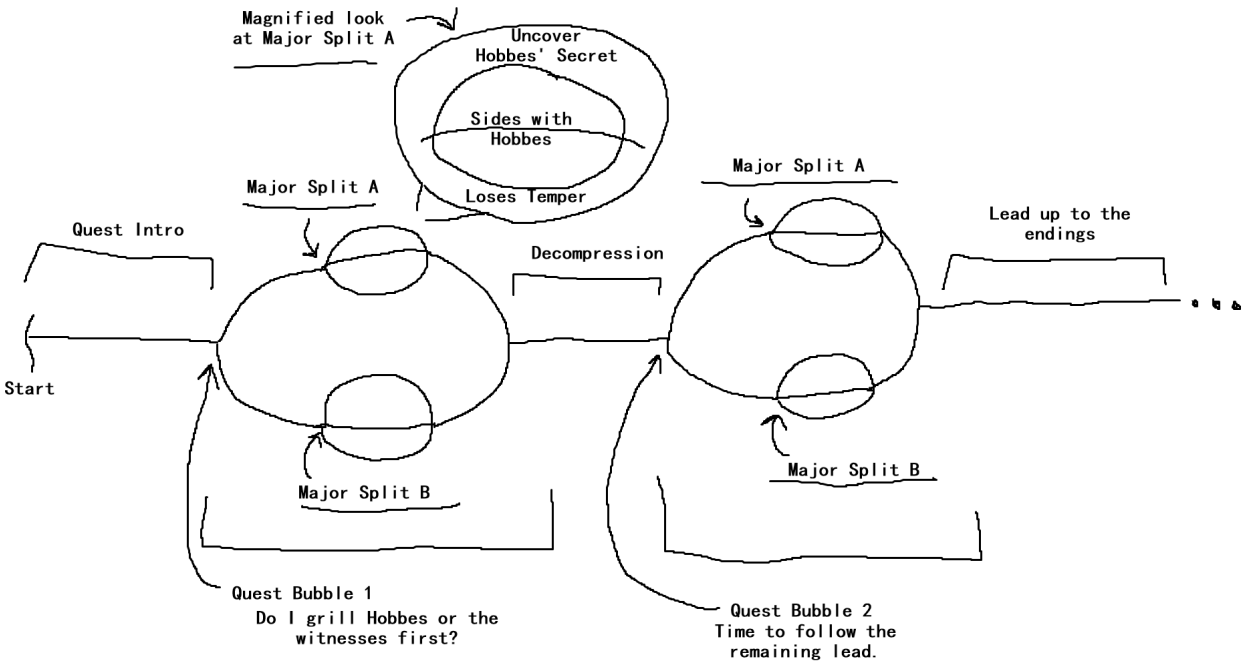
1. The Player Grills Hobbes and either:
    a. Discovers their secret.
    b. Loses their temper at Hobbes.
    c. Finds nothing of consequence.
2. The Player Grills Bys and the Arms Dealer and either:

a. Sides with the Arms Dealer

b. Sides with Bys

c. Comes to the wrong conclusion.

When charted in a timeline, it would look something like this:



In this example, we can see that the player has the choice to either pursue Major Split A (Grilling Hobbes) or Major Split B (Grilling Witnesses). From there, they can make major decisions which will effect the ending of the game.

STEP 2 Identify Primary Goals of Each Major Split

Next, we need to determine what progress the player must absolutely have made by the end of each split in order for the questline to proceed.

Major Split A, we know through our prewriting that the major goal is to obtain enough evidence (or perceived evidence) in order to make a decision on whether or not Hobbes is guilty. Therefore, the primary goal would be to Obtain Evidence of Hobbes' Guilt.

An important note: You will have noticed the "Sides with Hobbes" path in the Major Split A diagram above. Each time there is a line going through curved

options, that indicates a always accessible option, meaning that, no matter the player's build, they can still proceed through the quest (though not always to the best of results). Often, making these selections will lead to what we call a "Dump Ending", which will be talked about later in this document.

STEP 3 Isolate Each Major Split and Minor Split

Shown above, next we create a flow showing each major and minor split, minus the endings.

STEP 4 Key each Major Split Choice with a symbol (And Variable)

Next, we need to simply assign each major split choice with a symbol so we have a short hand when discussing our flows. This will allow them to be cleaner. We will need to create a variable in Articy as well for each split choice we need to track in order to output the appropriate endings/dialogue variations.

KEY:

1. Red Diamond = Discover Hobbes' Secret
2. Yellow Circle = Scream at Hobbes'
3. Purple Star = Side with Bys
4. Green Heart = Side with the Arms Dealer

STEP 5 Calculate each ending based on Split Choices Made

Finally, we pull everything together in order to isolate and implement each possible ending. Below is a potential outline of how endings can be triggered. In this example, the encounter with Hobbes is the primary encounter that shifts the player to one of three endings.

If the player discovered Hobbes secret, they are diverted to the diamond ending split.

If the player got angry at Hobbes, they are diverted to the circle ending split.

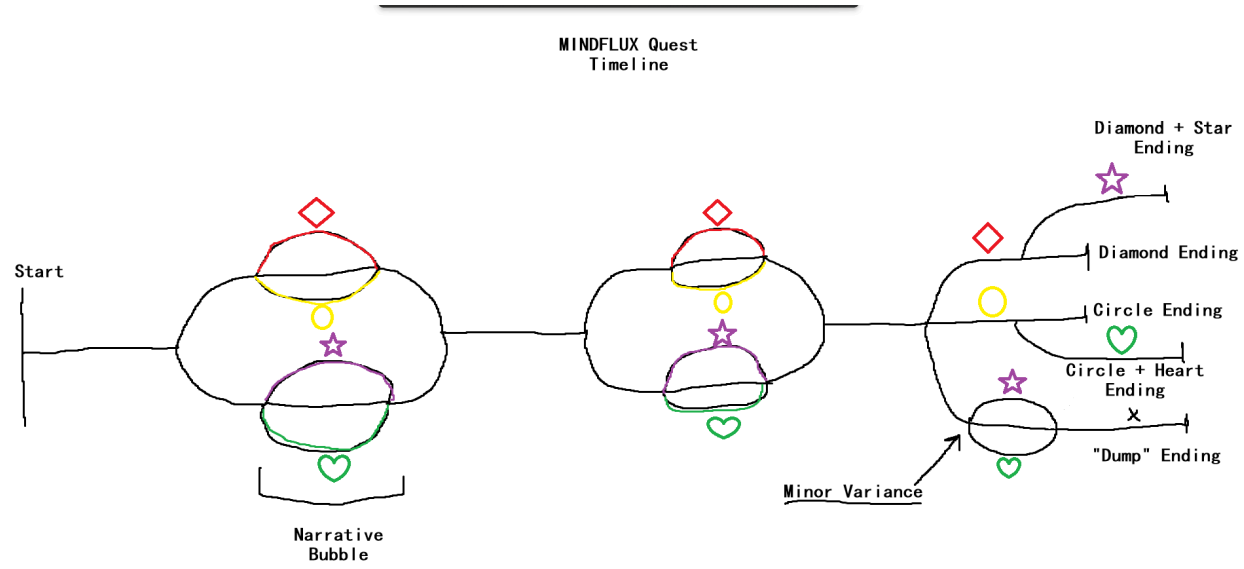
If they sided with Hobbes, they are diverted to the dump split, as that option did not require use of the personality traits or roleplaying system at large.

You will notice that there is another split branching off the diamond and circle endings. This is where the second encounter comes into play.

If the player discovered Hobbes' secret and sided with Bys, the uncover the full truth, as bitter as it is, granting rewards in line with that path.

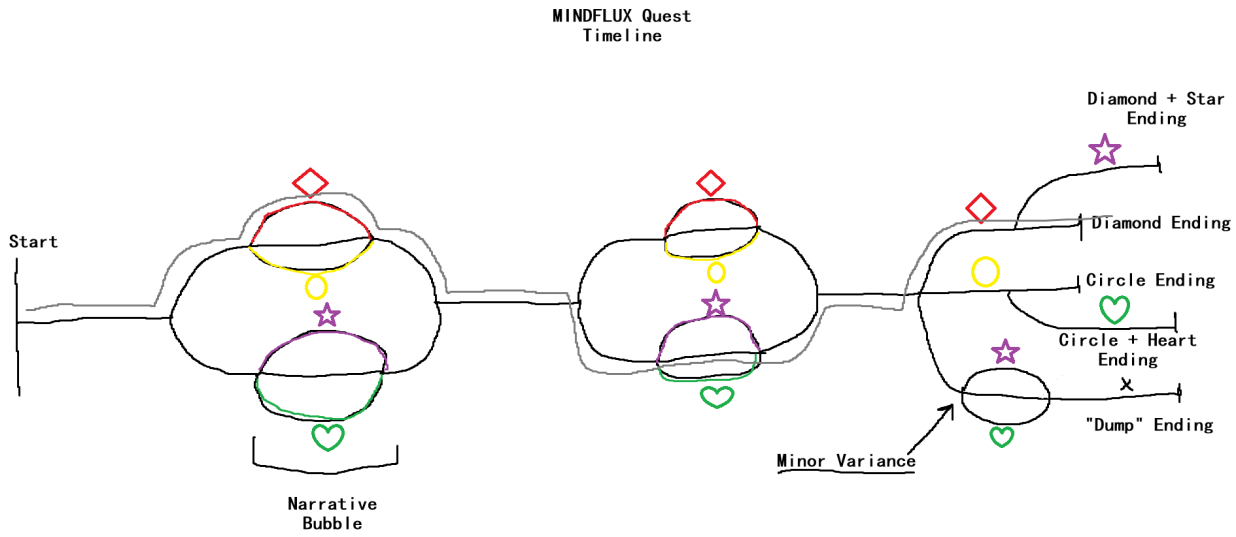
If the player got angry with Hobbes' and sided with the Arms dealer, they frame Hobbes', granting rewards in line with that path.

If the player sided with Hobbes, they would be granted a minor variance based on what they did with the witnesses, but in the end, Hobbes' blames you for the murder, forcing you to run from the scene.



MINDFLUX Questline

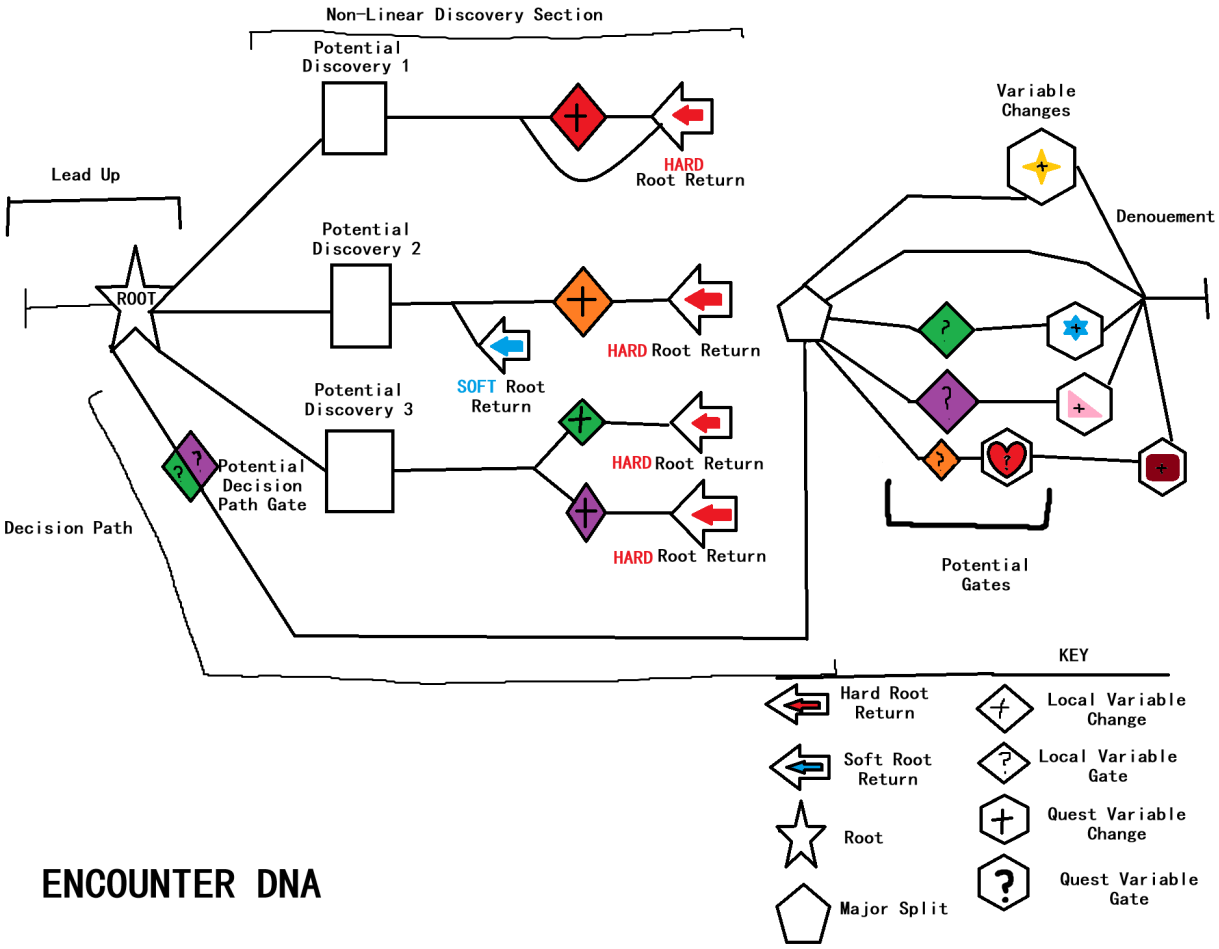
Player Flow example:



1. Player chose Hobbes' Major Split first.
2. Discovered Hobbes' Secret.
3. At the second Quest Bubble, player forced to do next encounter with witnesses as they've already done Hobbes.
4. Discovered nothing of use.
5. Player obtains the pure diamond ending.

D.N.A. of an Encounter

Now that we understand the way we can chart out an entire questline within the confines of a flow, what about an encounter?



Encounters are broken down into two major sections:

1. Discovery Segment
    a. The player is uncovering information, exploring options, or otherwise discovering insights concerning the encounter.
    b. No major encounter-ending decisions can happen during this segment.
2. Decision Segment
    a. Occurs after the discovery segment.
    b. Where the player makes a final decision.
    c. Is usually locked, meaning once the player has entered into the decision segment, they will be forced to make a choice. This adds pressure.

d. Decisions can be gated by both local and questline variables.

e. Choices often end with a questline variable change, but sometimes won't depending on the decision being made and the logic of the quest.

In the above image, every thing LEFT of the Major Split icon is the **Discovery Segment**.

The area RIGHT of the Major Split icon is the **Decision Segment**.

Questline Planning Terms

Glossary:

1. Root
    a. The homebase of the discovery segment.
    b. Can be either a literal narrative hub in dialogue, or a baseline state where the player isn't actively investigating an object.
    c. For the vast majority of flows, it is an invisible concept, not a physical dialogue structure.
2. Hard Root Return
    a. Returns the player back to the root and blocks the thread it is attached to.
    b. This prevents the player from continually doing the same investigation over and over.
3. Soft Root Return
    a. Returns the player back to the root without blocking the attached discovery path.
4. Variables (Global, Local, Etc)
    a. Explained Below.

Variables, Instructions, Conditions

The following section will discuss VARIABLES, and why/how they are used in Articy to accomplish all narrative logic.

What Is A Variable?

A variable, by definition, is a marker for something that can *change* in the game. It is the primary method in which we keep track of what the player has or has not done.

For example, let's say the player has the choice to either kill or save an NPC. They choose the kill option. In order for us to keep track of that decision, and more importantly use that decision in a later encounter, we would need a variable.

Variables also allow us to communicate information to custom-built systems in our project. For instance, can use a variable to compare a players skill potency against a skill gate to determine whether or not their check is successful.

What Categories of Variables are there?

From the perspective of the Narrative Designer, there are FOUR major types of variables:

1. Global Narrative Variables
    a. These variables are created in Articy and are used to track important decisions that can be used later in the narrative. If you need to keep track of a choice if the player leaves that conversation, you would use a Global Narrative Variable.
    b. Global Narrative Variables are also tracked by the Save/Load system (obviously), therefore they must be implemented on the Unity side as well by a designer.

RoleplayDemoVariables

Enter description here

Filter:

Name	Description	Type	Default value
AD_Gate_ArmIntimidate_Thread	Closes AD Arm intimidate thread once complete	Integer	0
AD_Gate_ArmRequisit_Thread	Closes AD Arm requisitioned thread once complete	Integer	0
AD_Gate_Medallion_Thread	Closes AD medallion thread once complete	Integer	0
AD_Gate_SaarpArmTech_Thread	Closes AD Saarpen arm tech thread once complete	Integer	0
AD_Gate_SaarpEnemies_Thread	Closes AD Saarpen enemies thread once complete	Integer	0
AD_Gate_SarpBusiness_Thread	Closes AD Saarpen business thread once complete	Integer	0
AD_Gate_Tech_Thread	Closes AD tech thread once complete	Integer	0
AD_Gate_WhereGotSaarpArm_Thread	Closes AD Where got Saarpen's arm thread once complete	Integer	0
AD_Gate_WhereSaarp_Thread	Closes AD Where is Saarpen thread once complete	Integer	0
AD_GateBys_Thread	Closes AD Bys thread once complete	Integer	0
AD_Info_DifferentKindOfDeal		Integer	0
AD_Info_HasMarketTech		Integer	0
AD_Info_Medallion	Tracks whether the player identified AD's medallion	Integer	0
AD_Info_NoLikeMarketists	Tracks that AD has revealed his dislike of the Marketists	Integer	0
AD_Info_NotLikeBys	Tracks that the player has noticed AD's annoyance at the mention of Bys	Integer	0
AD_Info_WantsSaarpenTech	Tracks that AD has revealed he wants a piece of Saarpen's tech	Integer	0
AD_Kickout_ArmNegotiation	Tracks when player kicked out of AD convo	Integer	0
AD_Kickout_CRIQuestion	Tracks when player kicked out of AD convo	Integer	0
AD_Kickout_MainHub	Tracks when player kicked out of AD convo	Integer	0
AD_Kickout_SaarpenHub	Tracks when player kicked out of AD convo	Integer	0
AD_Tally_AmbroseDesperate	Increases each time the player does NOT uncover a clue that AD is holding back information about Saarpen	Integer	0
AD_Tally_KnowsSomething	Increases each time the player uncovers a clue that AD is holding back information about Saarpen	Integer	0
Global_Info_ADClanAmina	Tracks that AD mentioned Clan Amina	Integer	0
Global_Info_ADHasSaarpArm	Tracks that AD has revealed he has the arm	Integer	0
Global_Info_ADMedallionLie		Integer	0
Global_Info_BysSawADWithPlate	Tracks if the player learns that Bys saw AD throw out the shoulder plate	Integer	0
Global_Info_TalkedToAD	Tracks that the player has talked to AD at least once	Integer	0
Global_Info_TalkedToBys	Tracks that the player has talked to Bys at least once	Integer	0

2. Local Narrative Variables
    a. These variables are created in Articy and are used only if the data DOES NOT need to be retained once the player leaves the conversation.

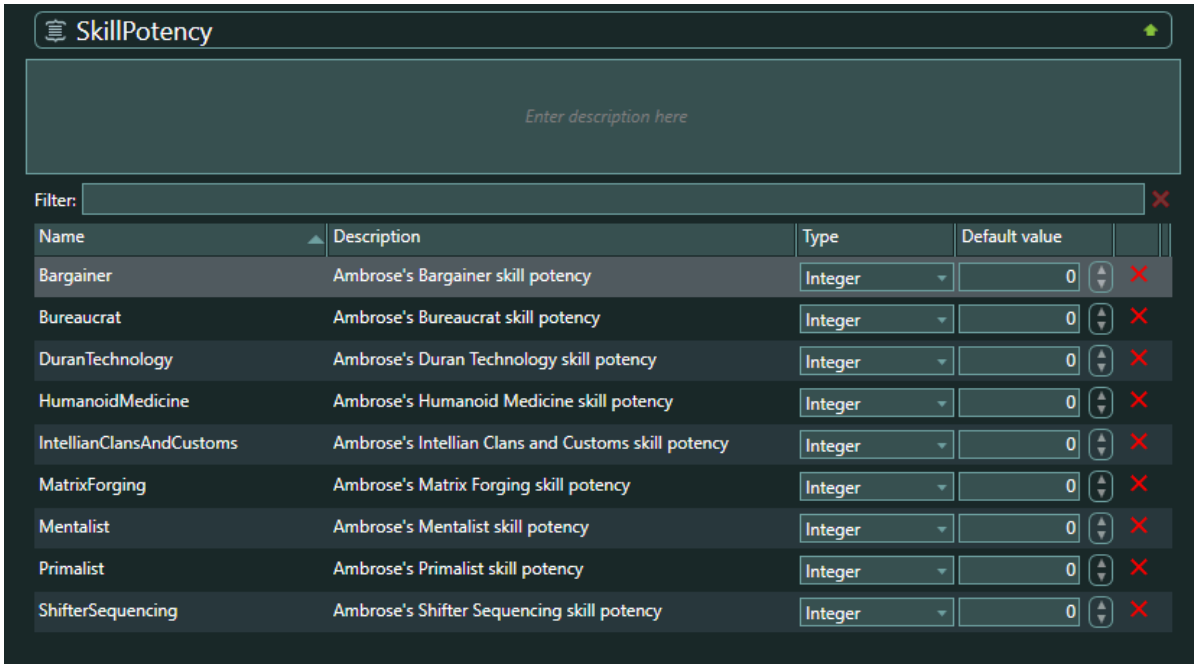
b. These variables are not transferred over to unity, which means they will not be saved by our save/load system. Once the conversation is closed, they will not be retained.

c. As of right now, we are not using a local variable system, but there is potential use for this in the future.

3. Skill Potency Variables

a. These variables are used to check the player's brain map skills potency, most often used to create gates.

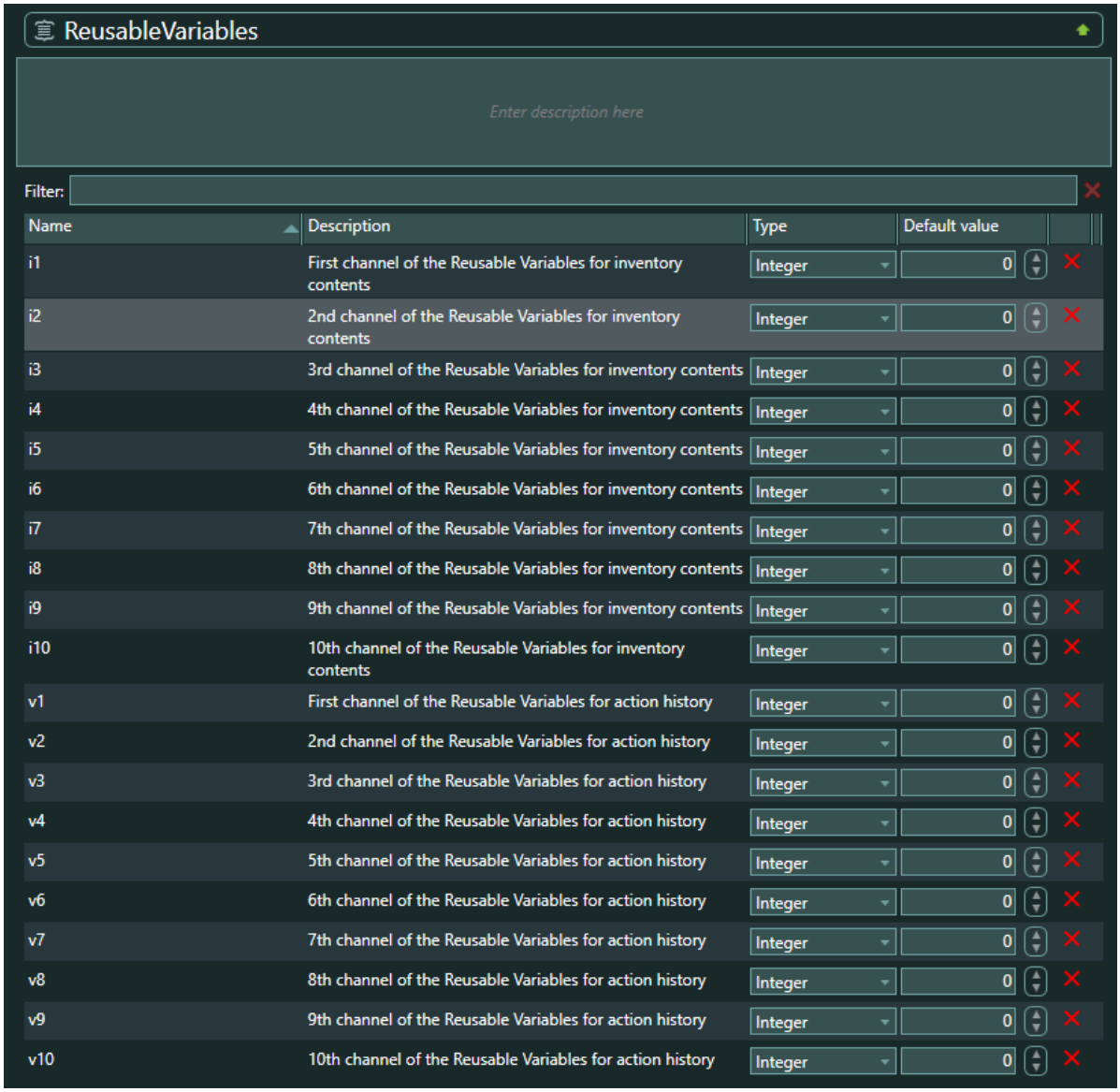
b. These will NOT be created by you, rather they're maintained by a game designer. More information on these variables below.



4. Item/Reusable Variables

a. These variables are used to check if an item is in the player's inventory.

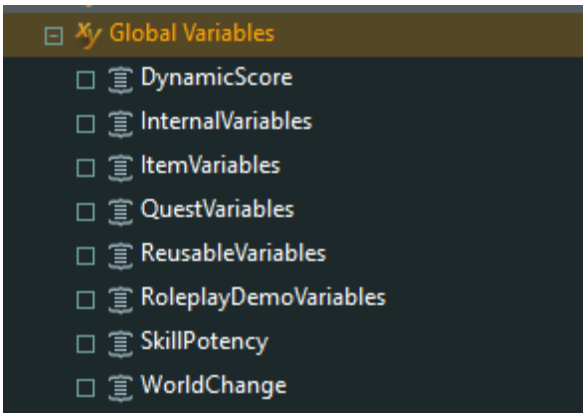
b. These will NOT be created by you, rather they're maintained by a game designer. More information on these variables below.



How Do I Make a Global Variable?

In order to create a variable, you must complete the following steps.

1. First, create or identify which LIST the variable must exist under. Variables are in fact stored in named lists, found under the Global Variable header in Articy:



The above image contains our current lists. For the Roleplay Demo, the majority of our “narrative specific” (non-external system related) variables exist inside of the RoleplayDemoVariables list, which looks like this:

RoleplayDemoVariables

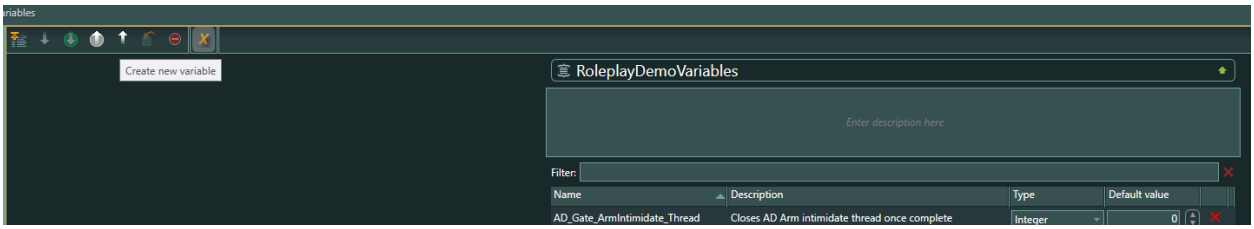
Enter description here

Filter:

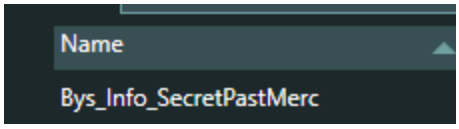
Name	Description	Type	Default value
AD_Gate_ArmIntimidate_Thread	Closes AD Arm intimidate thread once complete	Integer	0
AD_Gate_ArmRequisit_Thread	Closes AD Arm requisitioned thread once complete	Integer	0
AD_Gate_Medallion_Thread	Closes AD medallion thread once complete	Integer	0
AD_Gate_SaarpArmTech_Thread	Closes AD Saarpen arm tech thread once complete	Integer	0
AD_Gate_SaarpEnemies_Thread	Closes AD Saarpen enemies thread once complete	Integer	0
AD_Gate_SarpBusiness_Thread	Closes AD Saarpen business thread once complete	Integer	0
AD_Gate_Tech_Thread	Closes AD tech thread once complete	Integer	0
AD_Gate_WhereGotSaarpArm_Thread	Closes AD Where got Saarpen's arm thread once complete	Integer	0
AD_Gate_WhereSaarp_Thread	Closes AD Where is Saarpen thread once complete	Integer	0
AD_GateBys_Thread	Closes AD Bys thread once complete	Integer	0
AD_Info_DifferentKindOfDeal		Integer	0
AD_Info_HasMarketTech		Integer	0
AD_Info_Medallion	Tracks whether the player identified AD's medallion	Integer	0
AD_Info_NoLikeMarketists	Tracks that AD has revealed his dislike of the Marketists	Integer	0
AD_Info_NotLikeBys	Tracks that the player has noticed AD's annoyance at the mention of Bys	Integer	0
AD_Info_WantsSaarpenTech	Tracks that AD has revealed he wants a piece of Saarpen's tech	Integer	0
AD_Kickout_ArmNegotiation	Tracks when player kicked out of AD convo	Integer	0
AD_Kickout_CRIQuestion	Tracks when player kicked out of AD convo	Integer	0
AD_Kickout_MainHub	Tracks when player kicked out of AD convo	Integer	0
AD_Kickout_SaarpenHub	Tracks when player kicked out of AD convo	Integer	0
AD_Tally_AmbroseDesperate	Increases each time the player does NOT uncover a clue that AD is holding back information about Saarpen	Integer	0
AD_Tally_KnowsSomething	Increases each time the player uncovers a clue that AD is holding back information about Saarpen	Integer	0
Global_Info_ADClanAmina	Tracks that AD mentioned Clan Amina	Integer	0
Global_Info_ADHasSaarpArm	Tracks that AD has revealed he has the arm	Integer	0
Global_Info_ADMedallionLie		Integer	0
Global_Info_BysSawADWithPlate	Tracks if the player learns that Bys saw AD throw out the shoulder plate	Integer	0
Global_Info_TalkedToAD	Tracks that the player has talked to AD at least once	Integer	0
Global_Info_TalkedToBys	Tracks that the player has talked to Bys at least once	Integer	0

2. Once you have identified the correct list, simply check it out using the “claim partition” button.

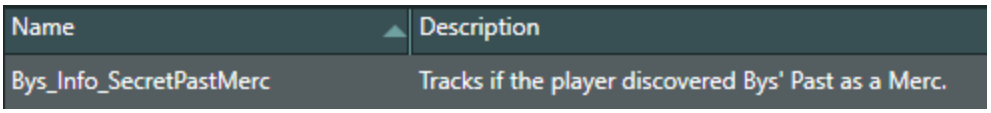
3. In order to create a new variable, press the X button (highlighted below).



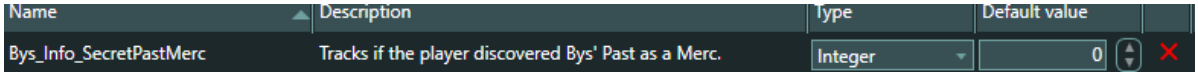
4. Next, name your variable using our standardized naming convention:
    a. CharacterName_VariableType_VariableUse
        i. CharacterName = Full or Abbreviated Character name that relates to the variable.
        ii. VariableType = What function does the variable serve? More info on Variable Types below.
        iii. VariableUse = Short descriptor of what the variable is tracking.
    b. For an example, lets say that you need a variable to track if the player has uncovered info regarding Bys’ past as a mercenary. You would create a variable named:
        i. Bys_Info_SecretPastMerc



5. Now create a brief description of what function this variable provides.



6. Lastly, change the “Type” field to Integer. It will default the variable’s value to 0.
    a. As a shorthand, 0 essentially means FALSE, or that the player HAS NOT done this action.



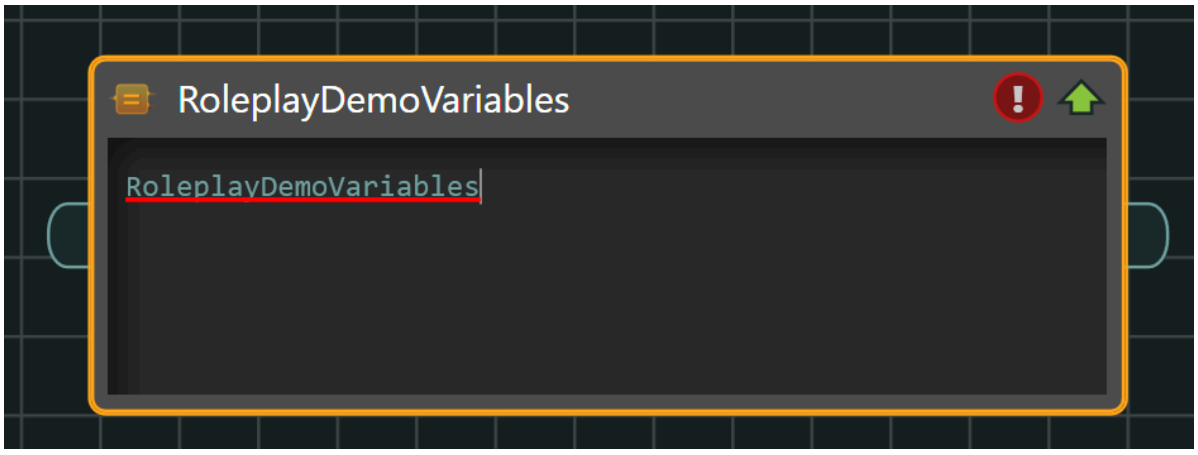
You have successfully created a variable.

How Are Variables Used in My Flows?

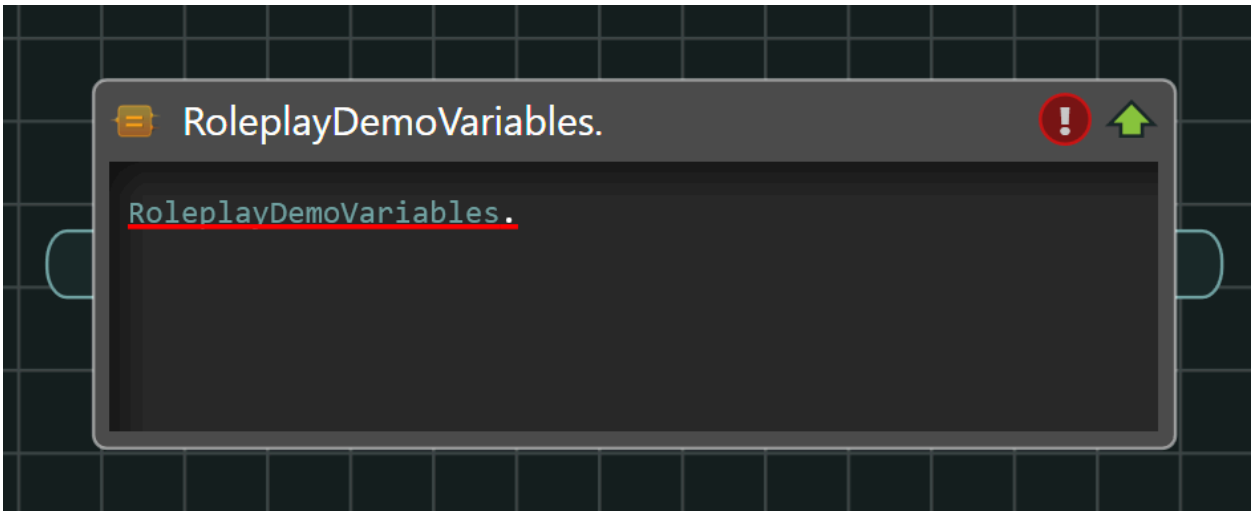
In Articy, variables are used inside the confines of CONDITIONS or INSTRUCTIONS. These two fragments are used to either CHECK or CHANGE the value of a variable. Referencing a variable in either a condition or instruction looks the same, though there are differences indicated below.

To reference a variable, you simply need to do the following:

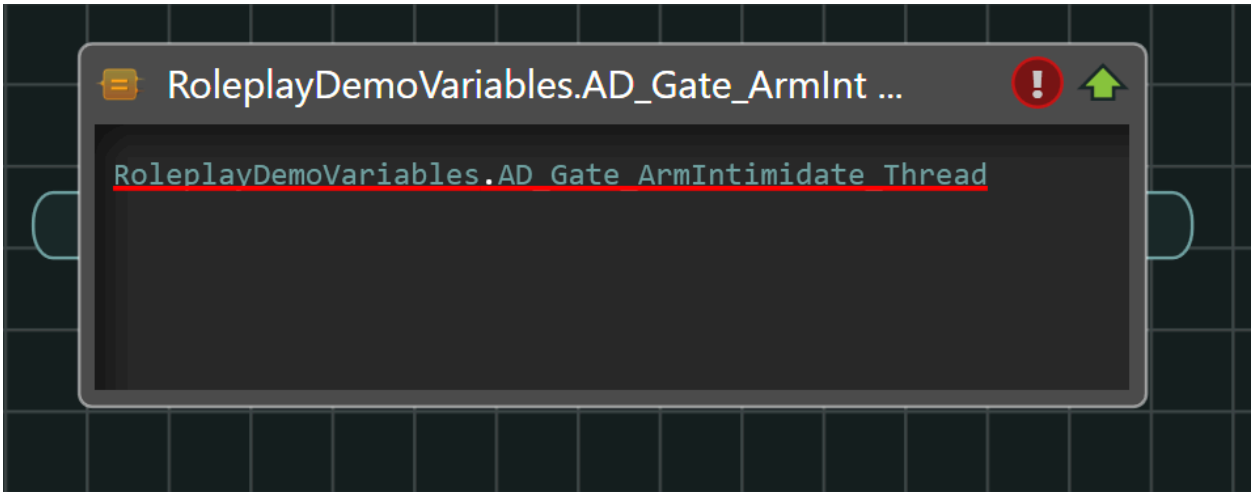
1. Type the Variable List the needed variable exists in. In this case, let’s say our variable lives in RoleplayDemoVariables, so we type that:



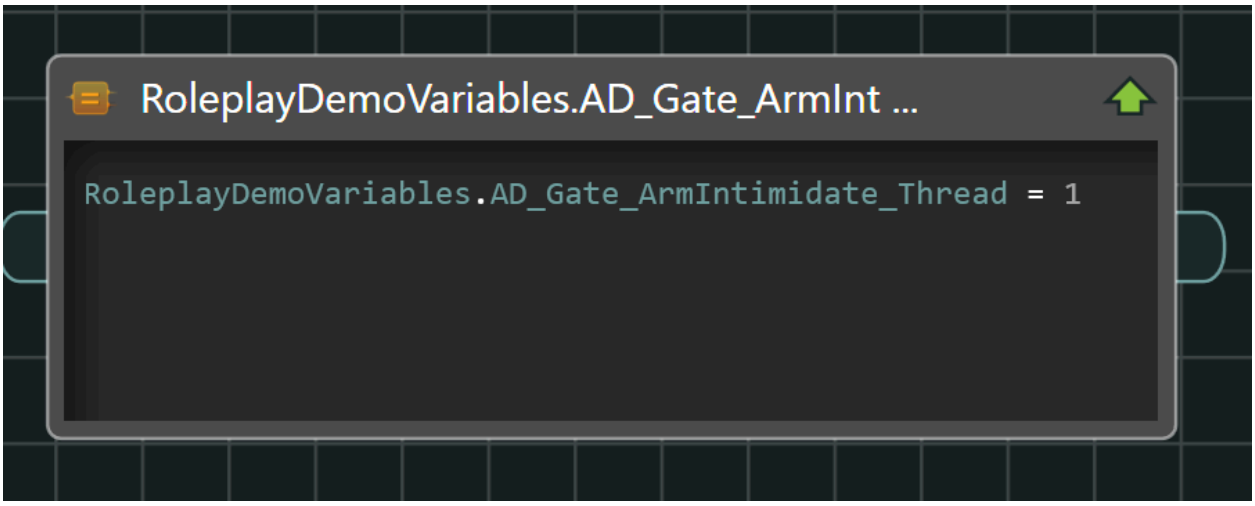
2. Add a period.



3. Type the variable you need to reference.

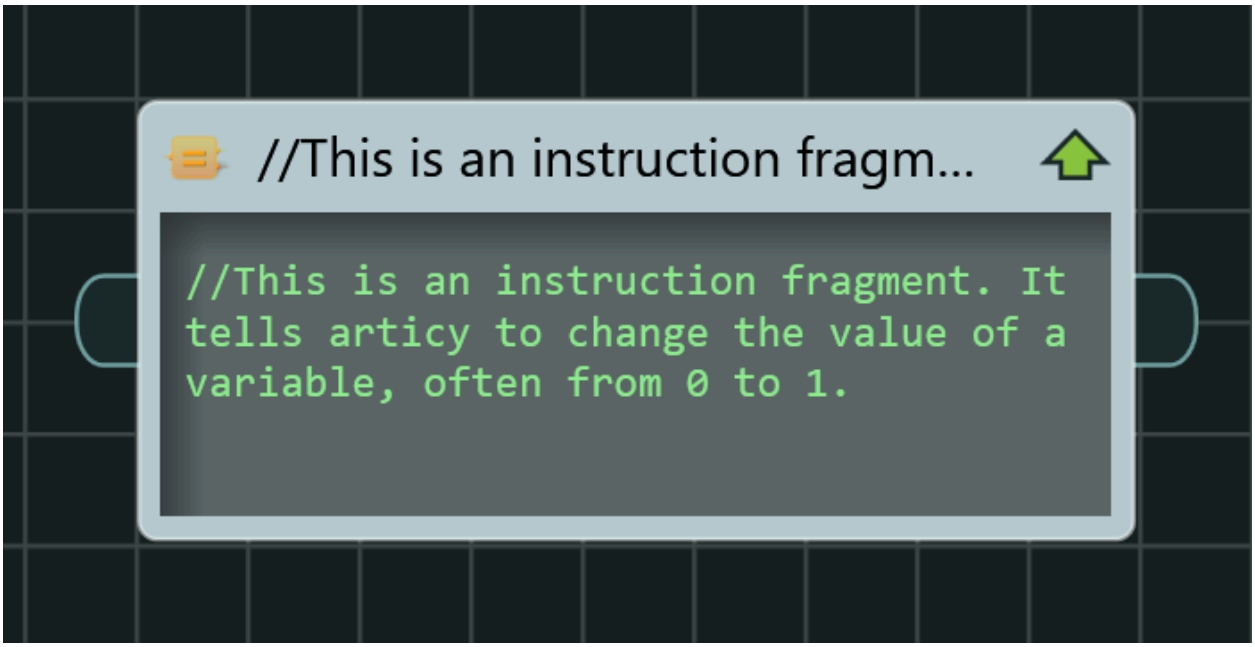


4. Add the subsequent instruction/conditions (explained in depth later!)

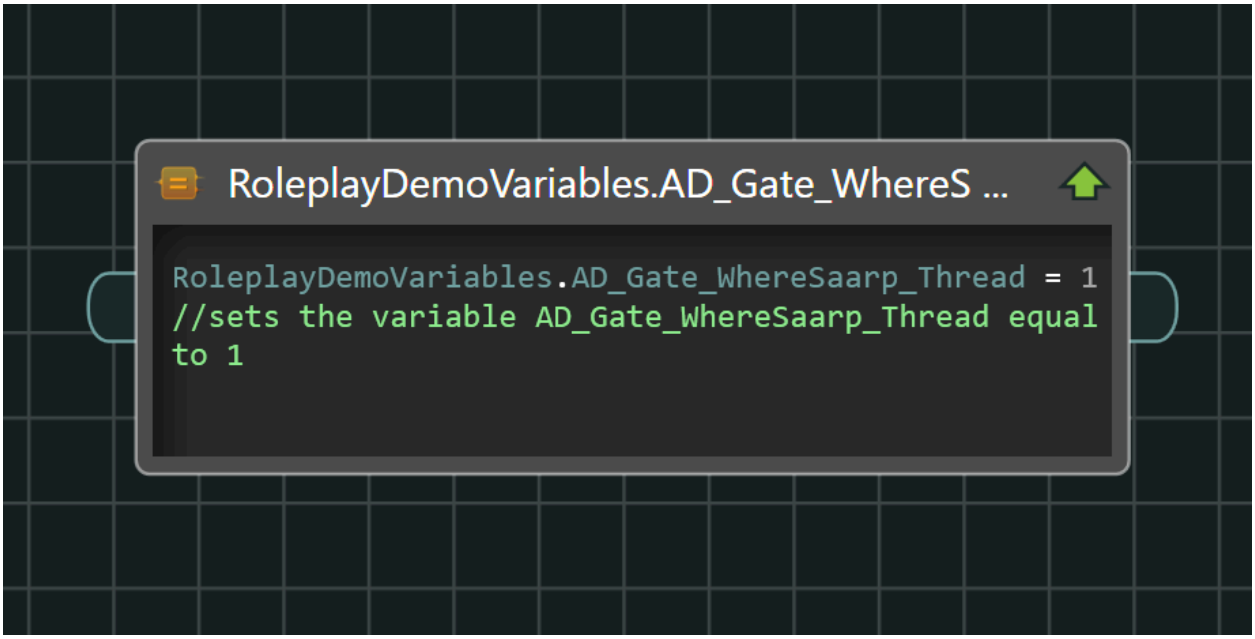


In practice, they look like this:

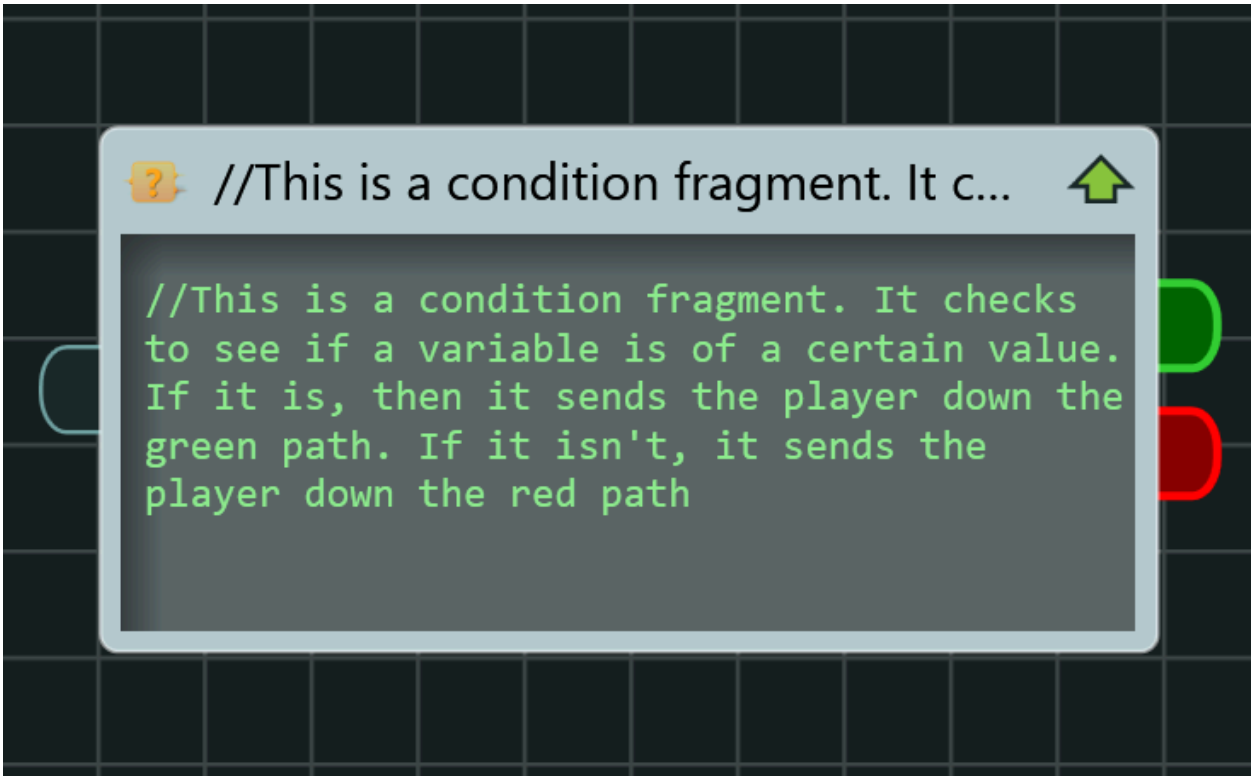
Instructions



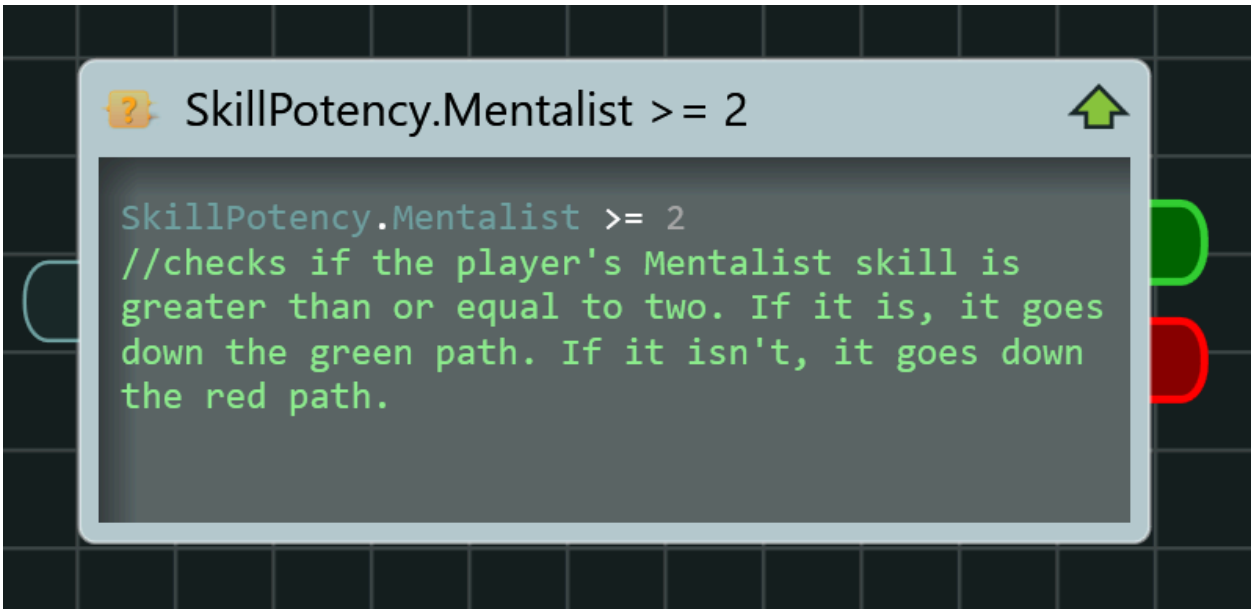
And this is what and Instruction looks like while being used:



Conditions



And this is what a condition looks like while being used:

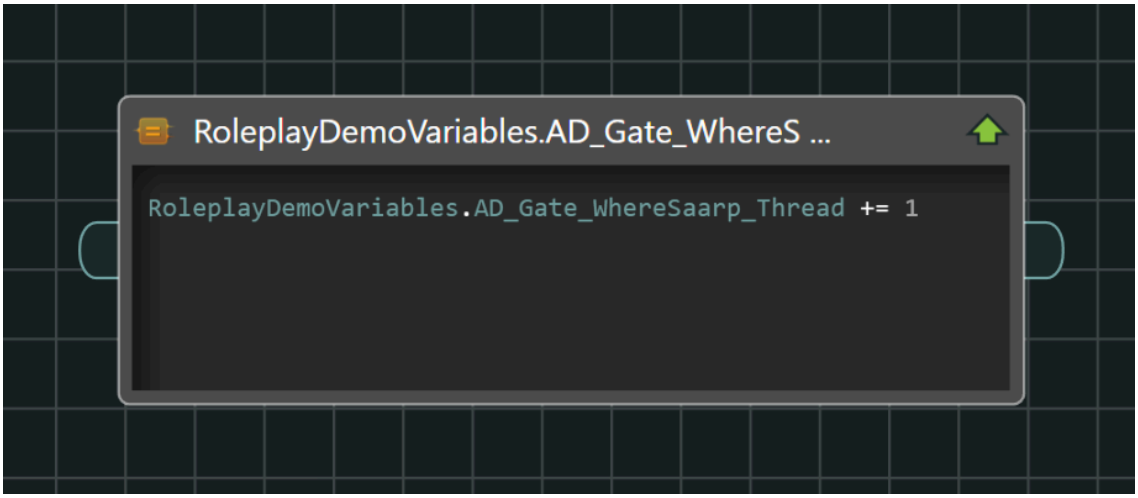


How Do I Write an Instruction?

Instructions can only do one thing: Change the VALUE OF A VARIABLE. They can make a variable equal whatever number you want, or increase/decrease a variable by a certain value.

1. Setting A Variable's Value Equal to the number listed:
    a. =
    b. EX: RoleplayDemoVariables.Variable_Name_Here = 1
        i. Sets the listed variable's value to 1.
        ii. If the variable's value was 10, using this instruction would set it to 1.

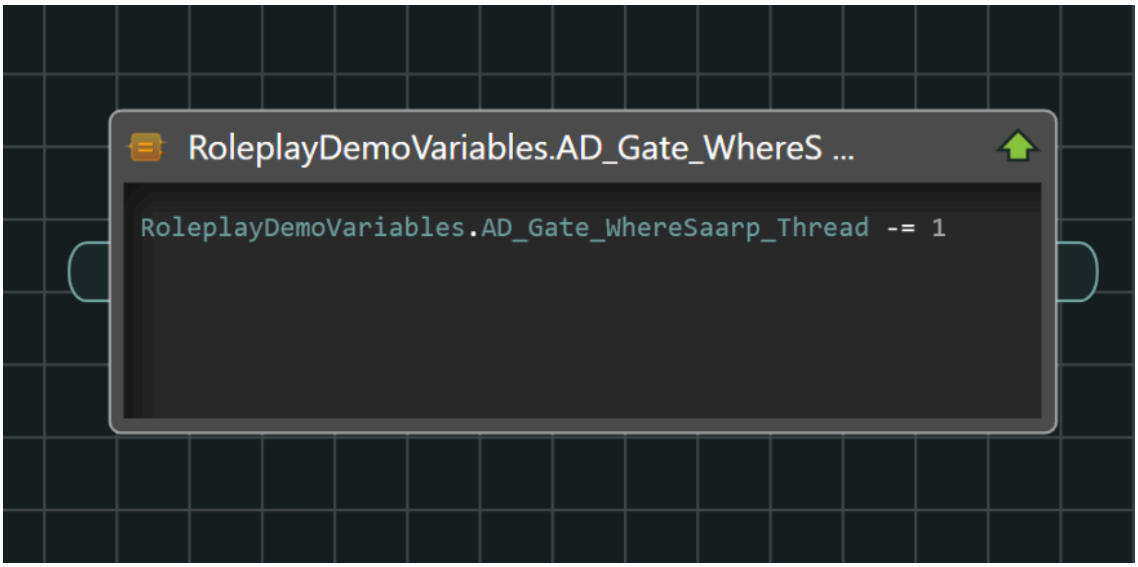
2. Increase the variable's value by an amount.
    a. +=
    b. EX: RoleplayDemoVariables.Variable_Name_Here += 1
        i. Adds +1 to the value of a variable.
        ii. If the variable's value was 10, using this instruction would increase it to 11.



3. Decrease the variable's value by an amount.
    a. -=
    b. EX: RoleplayDemoVariables.Variable_Name_Here -= 1

i. Subtracts -1 to the value of a variable.

ii. If the Variable's Value was 10, using this instruction would reduce it to 9.

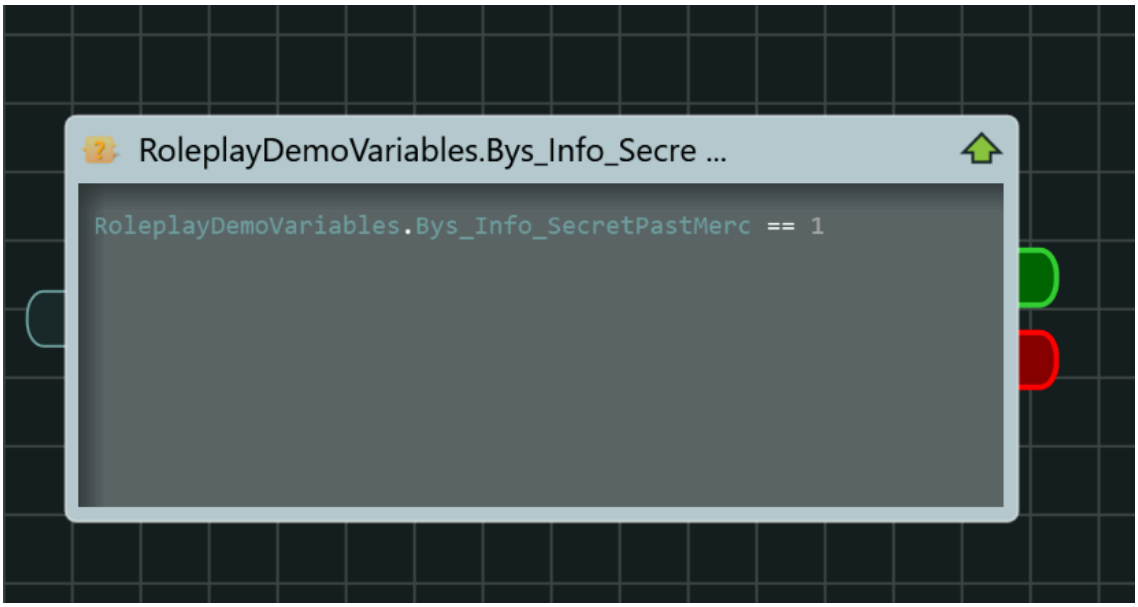


How Do I Write a Condition?

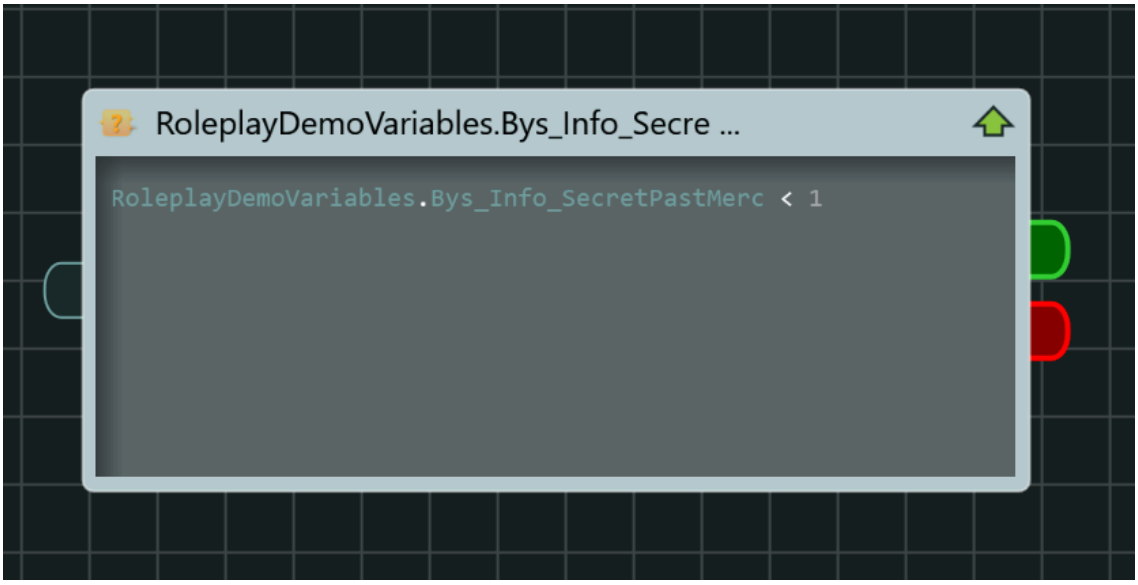
Conditions can only compare a variable's value to a referenced amount.

Simply put, conditions ask the question "Is the indicated variable's value equal to/less than/greater than/less than or equal to/greater than or equal to this number?"

1. If you want to see if a variable is equal to a specified value:
    a. ==
    b. EX: RoleplayDemoVariables.Variable_Name_Here == 1
        i. Checks to see if the variable listed is exactly equal to 1.
        ii. If it is equal to 1, green path. If not, red path.



2. If you want to see if a variable is less than a specified value:
    a. <
    b. EX: RoleplayDemoVariables.Variable_Name_Here < 1
        i. Checks to see if the variable listed is less than one.
        ii. If it equals 0 or below, green path. If it equals 1 or above, red path.

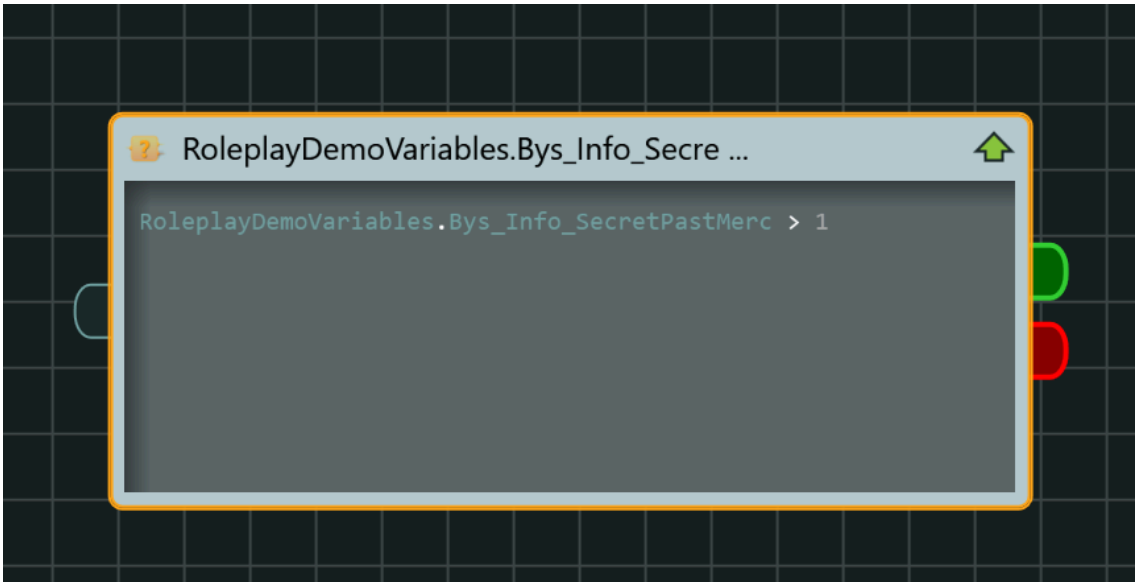


3. If you want to see if a variable is greater than a specified value:
    a. >

b. EX: RoleplayDemoVariables.Variable_Name_Here > 1

   i. Checks to see if the variable listed is greater than one.

   ii. If it equals 2 or above, green path. If it equals 1 or below, red path.



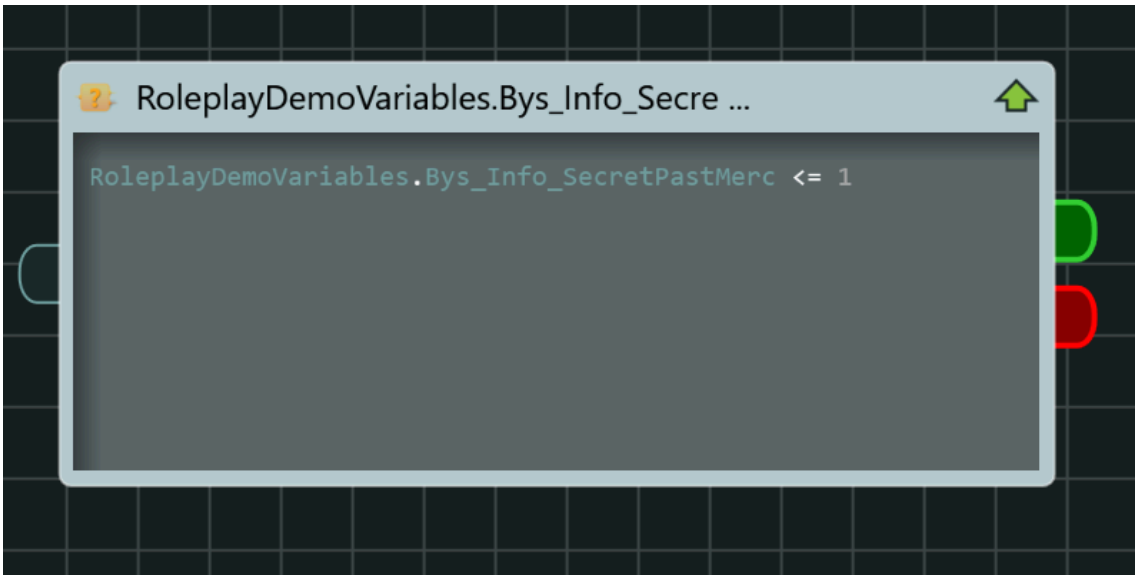
4. If you want to see if a variable is less than or equal to a value:

   a. <=

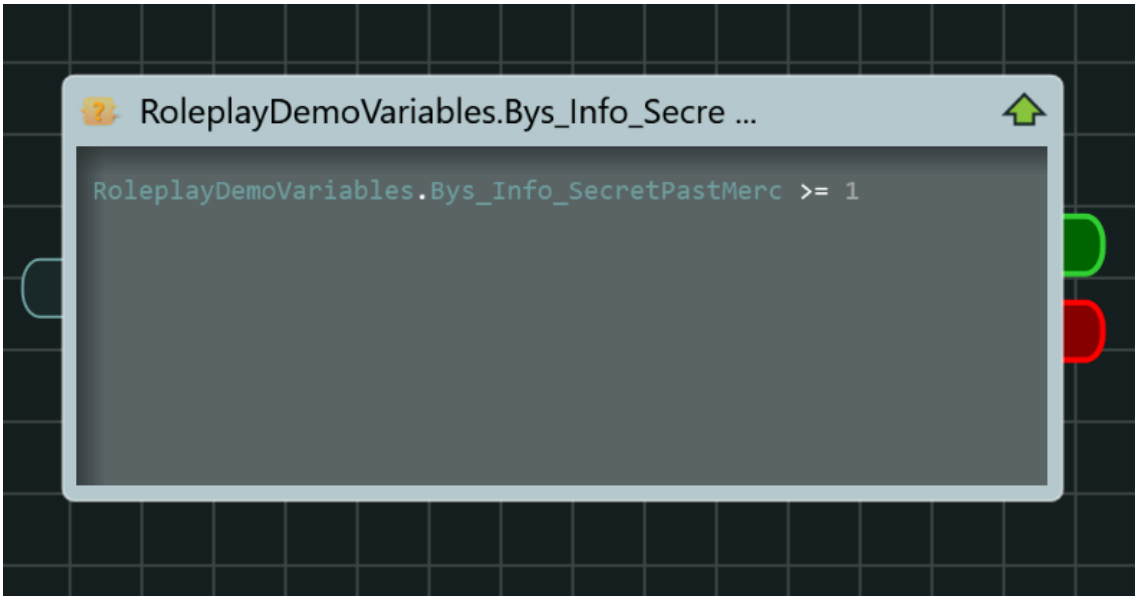
   b. EX: RoleplayDemoVariables.Variable_Name_Here <= 1

      i. Checks to see if the value equals or is less than 1.

      ii. If one or below, green. If 2 or above, red.



5. If you want to see if a variable is greater than or equal to a value:
    a. >=
    b. EX: RoleplayDemoVariables.Variable_Name_Here >= 1
        i. Checks to see if the value equals or is greater than 1.
        ii. If one or above, green. If 0 or below, red.

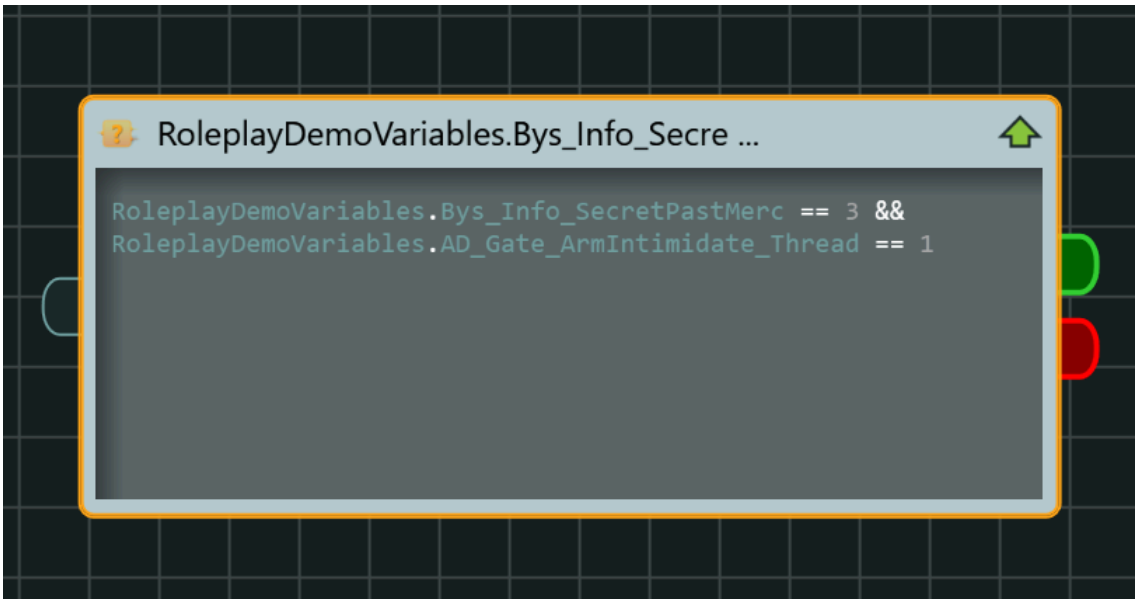


6. If you want to chain MULTIPLE conditions:
    a. &&

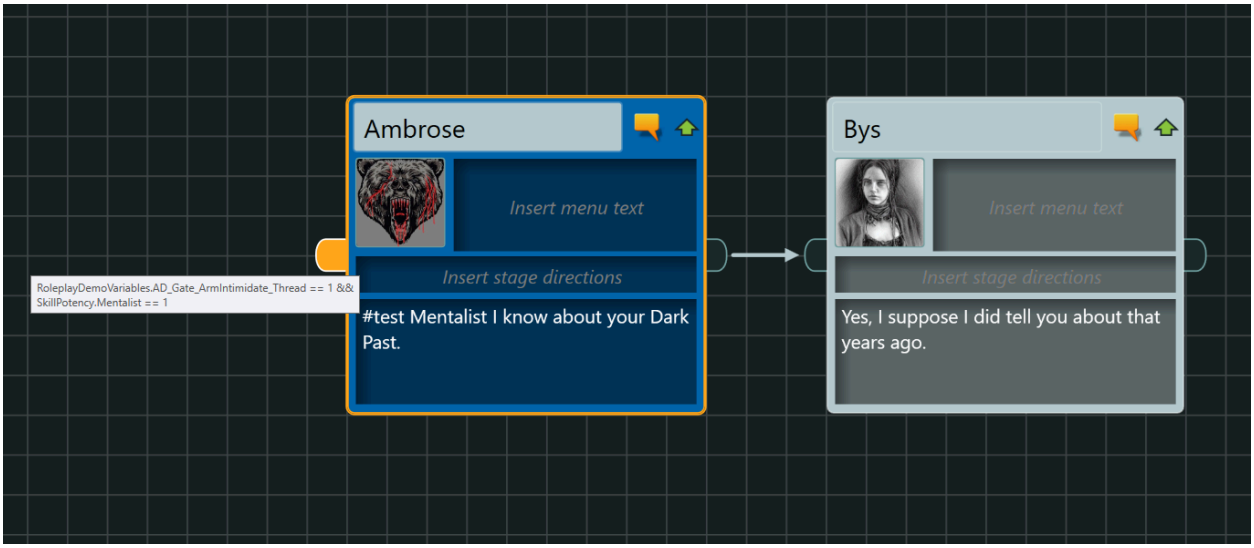
b. EX: RoleplayDemoVariables.Variable_Name_Here1 == 3 && RoleplayDemoVariables.Variable_Name_Here2 == 1

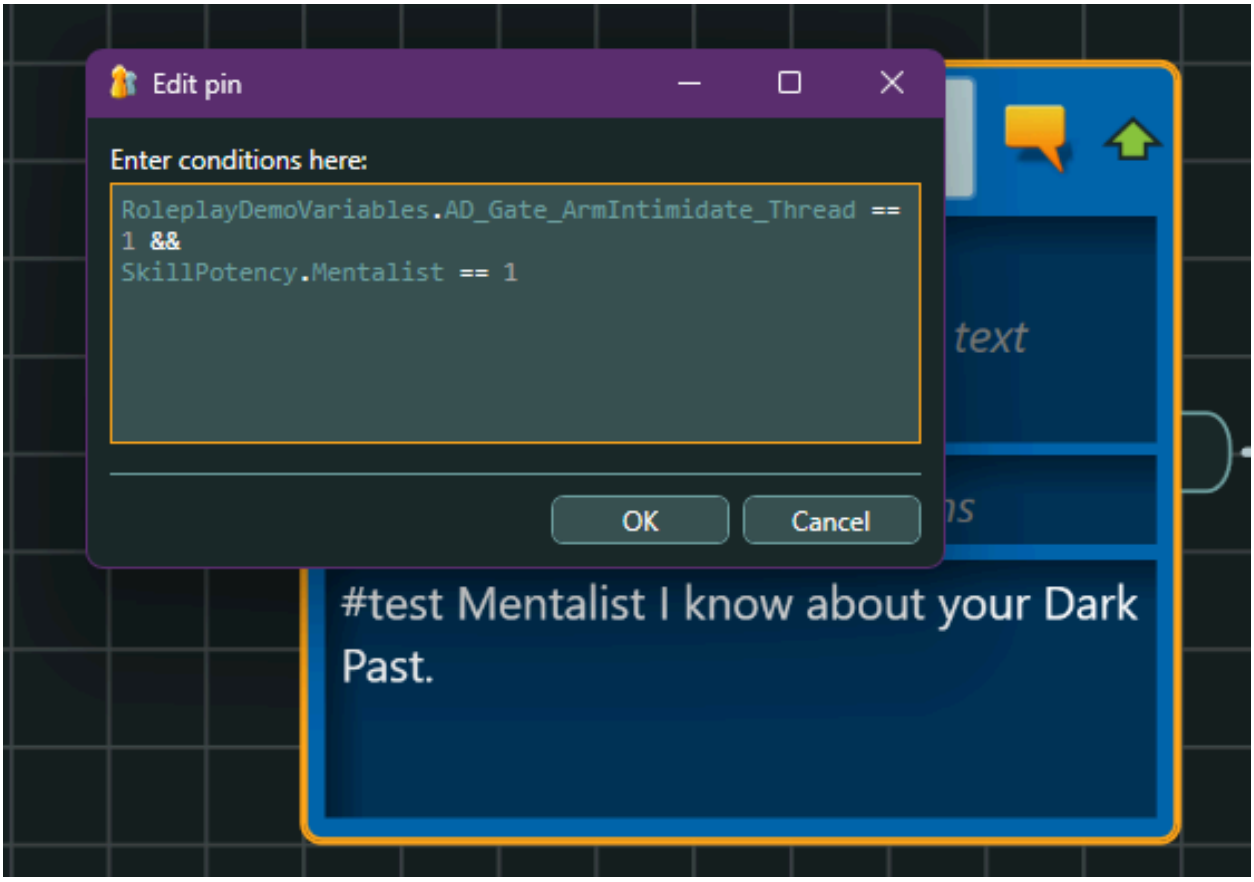
   i. Checks to see if variable one equals 3 and variable two equals 1.

   ii. If one = 3 and two = 1, green. If one is anything else or two is anything else, red.



1. It is good practice to put conditionals into the INPUT PINS of options you want gated:

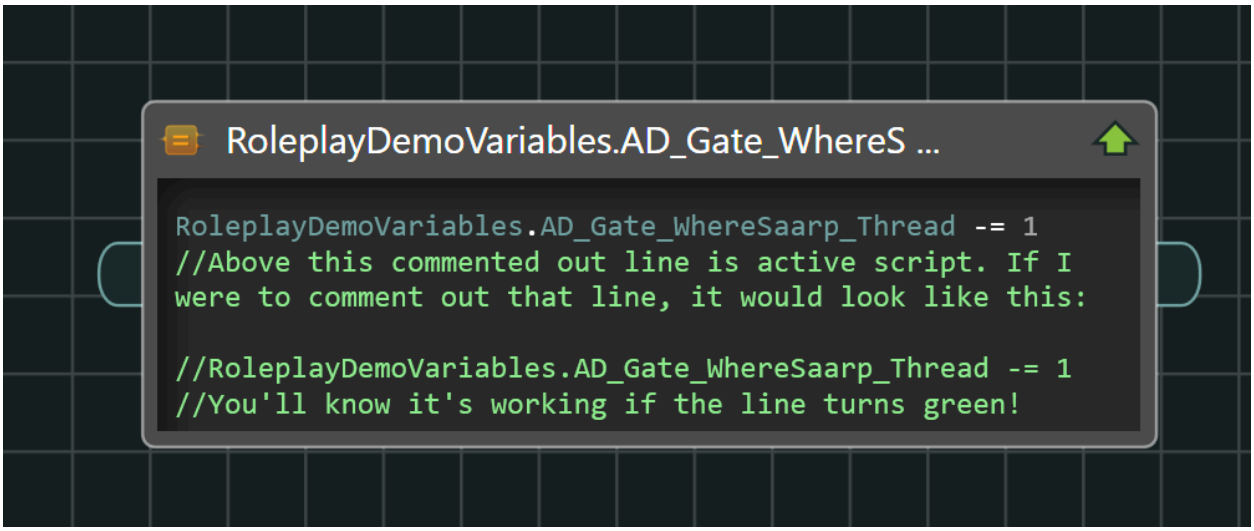




What is “Commenting Out Script” and How Do I Do It?

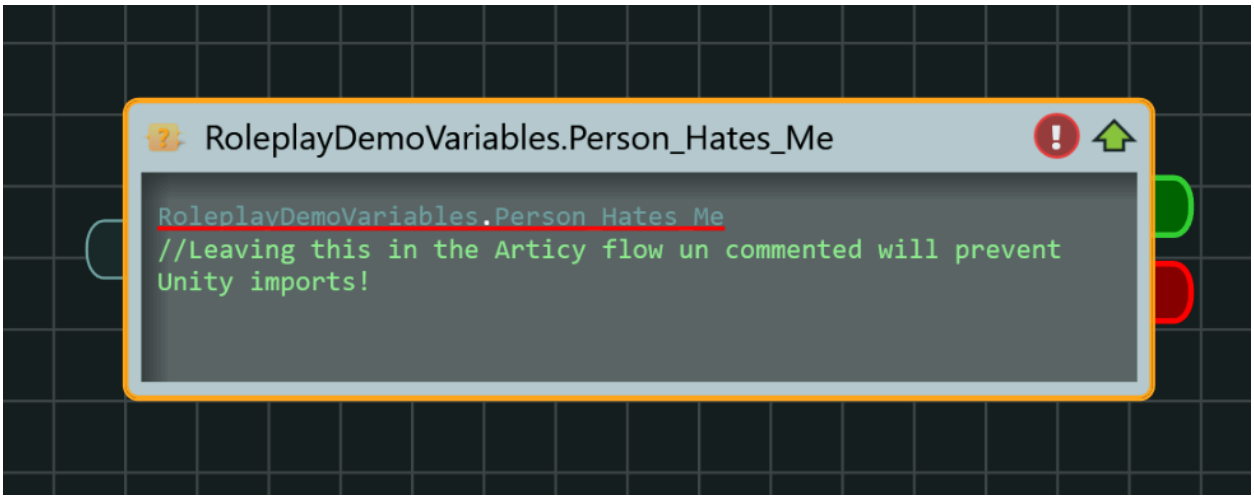
Commenting out is when you use // to turn live script into text that is not read by Articy or Unity. This is used to add clarifying notes or to cancel out logic without needing to delete it.

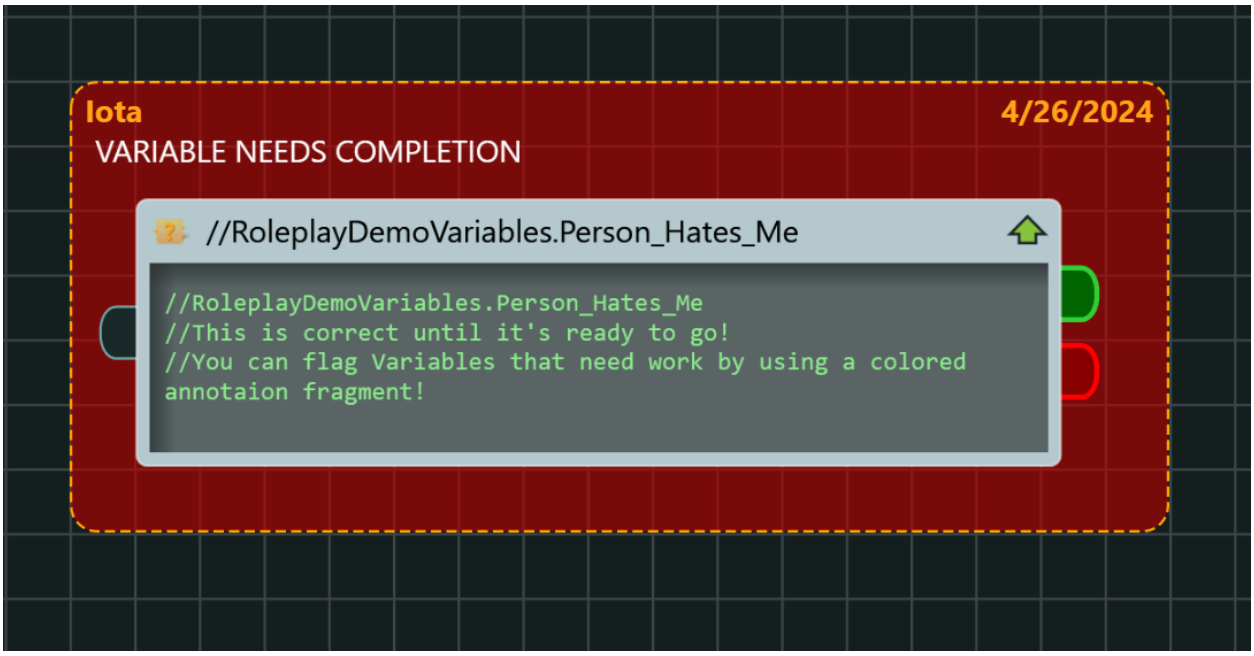
It is done by adding two slashes // before the scripted line, and looks like this:



Policy on Unfinished Variables and Commenting

ALWAYS comment out script that is not ready to be implemented or has RED LINES UNDERNEATH. If you do not, the Articy project cannot be imported into Unity! Always comment them out until they are 100% ready to go!





When Should I Use an Instruction?

You use an Instruction fragment when you want to CHANGE the value of a variable, usually from a 0 to a 1. We only do this to out custom GLOBAL NARRATIVE VARIABLES, never to our ITEM VARIABLES or SKILL POTENCY VARIABLES as those are managed by Unity.

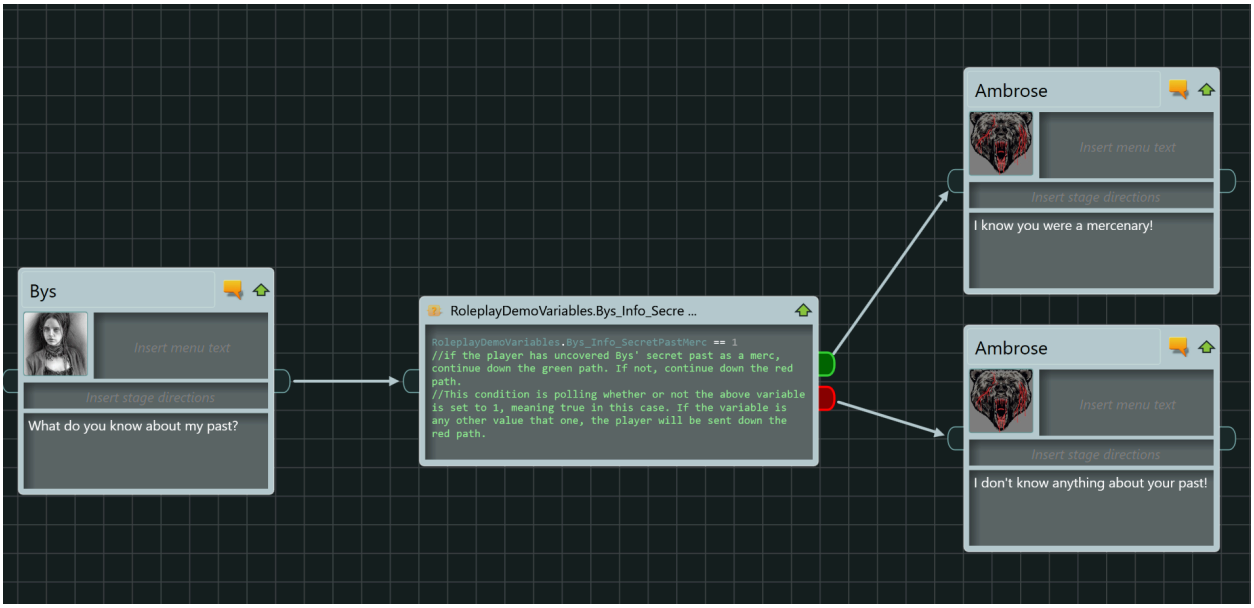
For example, let's say that we want to track if the player has uncovered Bys' secret past as a mercenary. We'd create a variable called Bys_Info_SecretPastMerc, make it an integer set to 0, then create an instruction that changes that variable to 1 (true) at the moment in the narrative where the player uncovers that information.

In the above example, the player asked about Bys' secret past and received an answer. We can now use this variable in the future to create gates, like below.

When Should I Use a Condition?

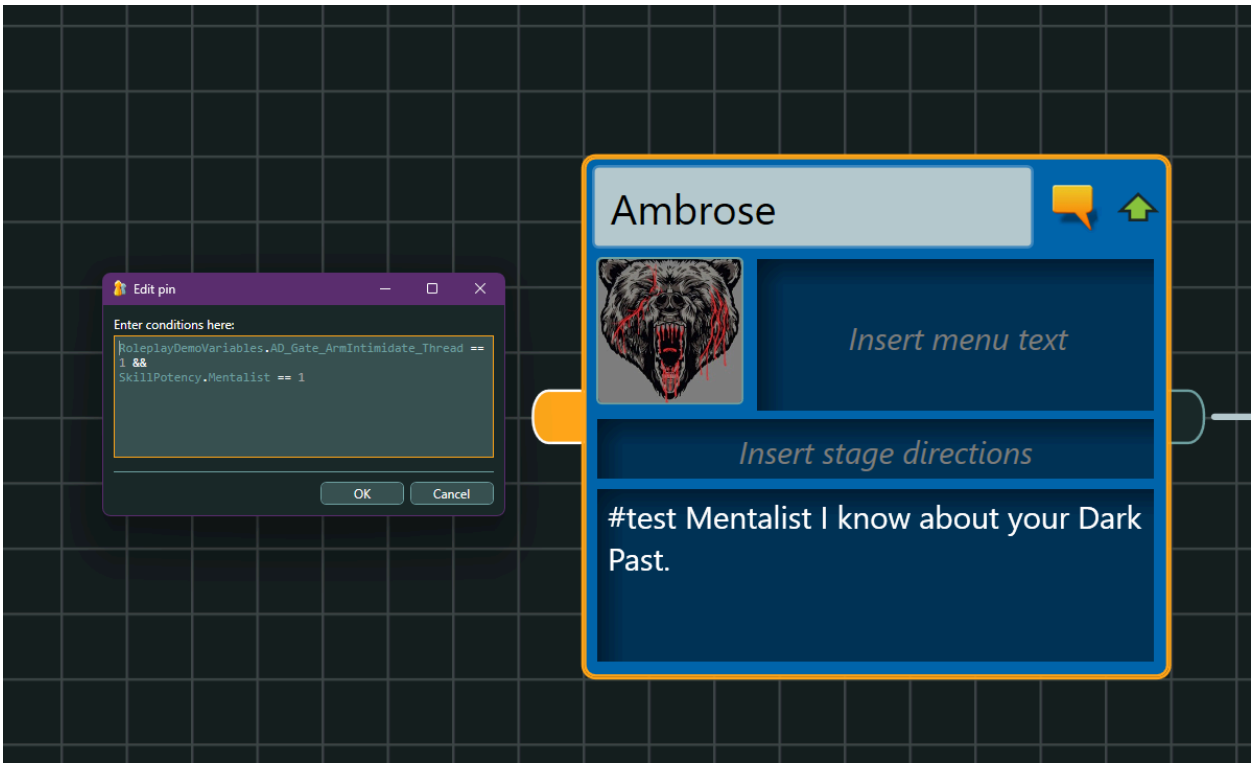
You use a Condition fragment when you want the player's narrative path to split due to past choices or character build statistics. We do this by CHECKING a variable's value, then sending the player down the appropriate path depending on said value.

For example, let's say that we want to create a new story thread that only opens up if the player has uncovered Bys' secret past, a variable we created in our Global Variable List called Bys_Info_SecretPastMerc.



In the above example, we have a condition that checks the variable to see if it is set to 1. If it is, you go down the green path. If it isn't, you go down the red.

Conditions can also be used in PINS. Often, this is a necessity as it limits the amount of fragments you have in your flows.



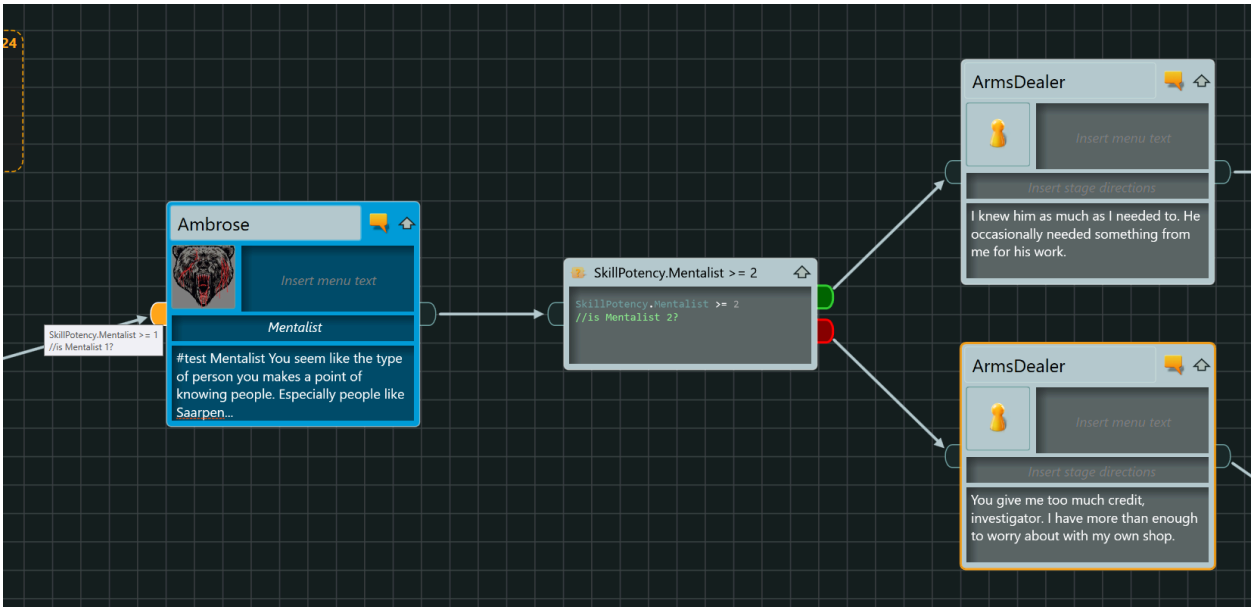
Variable Use Cases

There are technically two major ways that variables are utilized in our game- Checks and Gates. Each of these rely on the various variable sub groups created to help quickly differentiate what task a variable is accomplishing.

It is important to note that the original groupings (Global Narrative, Local Narrative, Skill Potency, Item/Reusable) are sometimes broken down into sub-categories, especially the Global Narrative group. Technically, the sub-categories all function the exact same way, we just give them different names AND colorations in Articy to help us quickly identify their specific use case while looking at them in a list or in the flow. Try your best not to get confused.

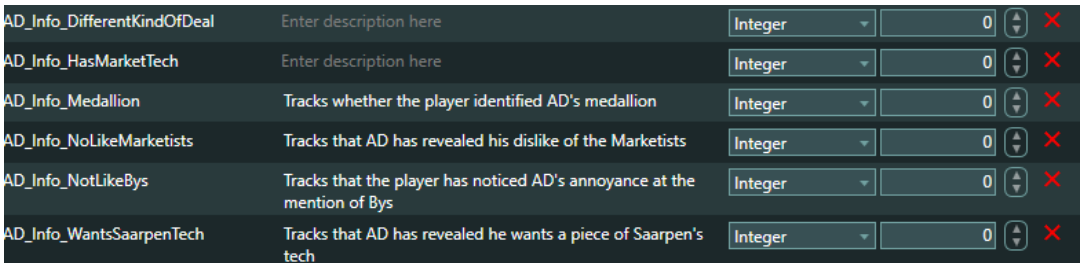
Lets do a quick overview of each type of use case:

1. Check
    a. Checks are used when you need to compare a variables stat against an expected value.
    b. Most often used via the Skill Potency Variable set to compare Brain map statistics against expected values and the Global Narrative Variables.



Example of a skill potency check structure.

2. Gate
    a. Gates are used when you need to block what threads the player has access to.
    b. Most commonly, gates use the Global Narrative Variable types.
    c. The following are all normal Global Narrative Variables, but their names indicate their use case. They all technically function the exact same.
        i. Global
            1. Used when the variable is used in multiple flows.
        ii. Info
            1. Used to identify information that the player has uncovered about someone.

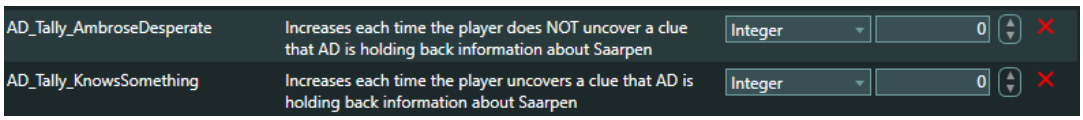


iii. Lockout (Currently called “Gate”)

1. Used to mark when a thread has been completed and never needs to be accessed again.
2. Prevents the player from endlessly looping conversation threads that shouldn’t be repeated.

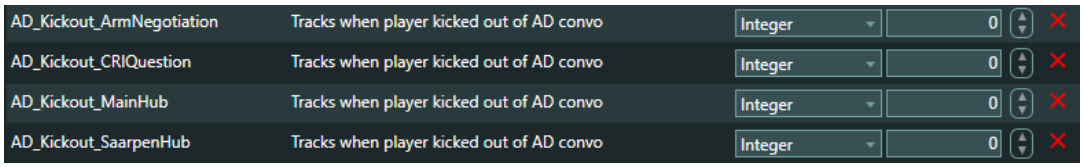
iv. Tally

1. A stacking variable that increments higher than 1. Used when a player needs to complete numerous actions to continue down a questline. Prevents chaining multiple gates and making multiple variables.



v. Kickout

1. Used when a player is kicked out of a conversation due to narrative reasons, such as the NPC getting mad. When the player re-enters the conversation.



MINDFLUX Narrative Gameplay Loop

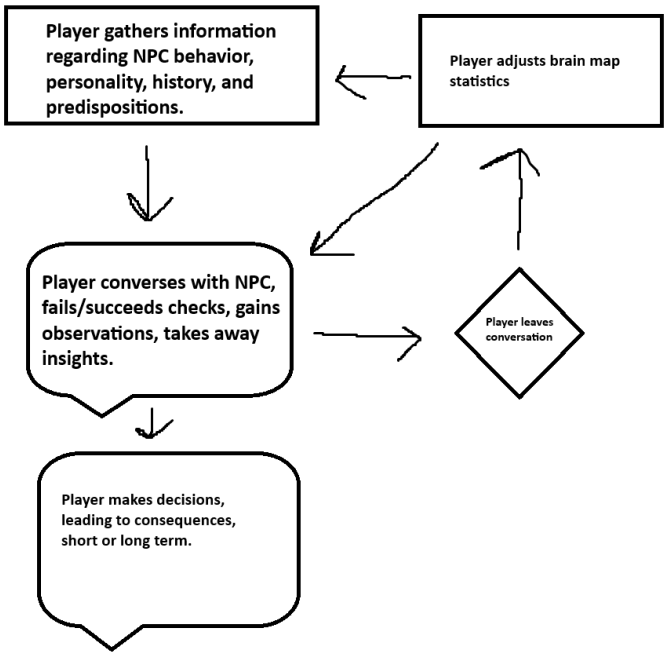
1. The *Cascade* roleplaying system has been described as “Tactical Roleplay”, and for good reason. The name of the game is *STRATEGY*. NPCs are living, breathing people with wants, hopes, and desires. These may run parallel or contrast to your own. In order to get what you want, you’re going to have to learn about them and push their buttons.
2. The player fantasy of MINDFLUX is that of the Mechanized Man- a person who can modify their personality at will to get what they want.
3. The computer brain of a Cerebroid groups major conversational paradigms into four categories:
    a. Mentalist
        i. Using subtle manipulation and psychological tricks.
    b. Bureaucratic
        i. Using the law and ones own authority.
    c. Primalist
        i. Using primal, animalistic strategies.
    d. Bargaining
        i. Using trade or bartering, tit for tat.
4. The effectiveness of these conversational paradigms (called Personality Traits or Skills), is directly related to how much of a resource is dedicated to it. This resource is called X85, the artificial neuron structure of the Axons.
5. The more X85 you dedicate to a skill, the better it runs. Better, of course, is subjective. If a person is highly adept at avoiding certain conversational paradigms, they might react poorly to using one with a such a high degree of power.
6. The problem is, X85 is a limited resource. Therefore, a Cerebroid must be strategic as to what conversational paradigm they wish to rely on during a given encounter. There are often many answers to the same problem, which is where roleplaying comes into account.

7. Knowing what to power comes from *INVESTIGATION*. Learning about NPCS, making observations, discovering items and analyzing them. The more you know about a person, the more likely you are to get what you want.
8. This means that, as a Narrative Designer, you must know the character you're writing for to a high level.
9. Remember- this is an NPC-focused title. These are living, breathing people who want what they want, not just some mindless quest giver.
10. As an example, let's say you're writing an NPC who is a double agent. How would they react to a player using a high-level Primalist skill? If they've been trained well, Primalist would most likely not work in the way the player might expect. What about Mentalist? Would they react poorly because they've been trained against such tactics, or would they get excited because the player knows how to play "the spy game"? What potency levels to use for checks is directly based on the characters personality. There are no hard and fast rules here, only the rule of character.
11. Below is a sample of the MINDFLUX narrative gameplay loop. This will not be sufficient to train you to write effectively for us, but it will provide an excellent foundation.

Discovery Mechanics:

1- Journal Log Entries
a. Discoveries logged via journal entries.
2- Observations
a. Observations seen during dialogue/item investigation segments that give insights to the task at hand
3- Item Investigations
a. Items that can be used to grant further insight to an NPC or quest situation.
4- Successful Skill Checks
a. Can be used to obtain useful information or to make fundamental decisions regarding narrative direction.

Variables are used to track player decisions, important information, or items obtained. These variables can then be used to open or restrict narrative threads.



11. The below sections describe the structures we use to write narratives in Articy. These are more technical rundowns. In order to become an effective writer, you must master the technical side as well as the creative design side.

Design Structures of MINDFLUX

In MINDFLUX, we have a smattering of design structures that control how are narrative is built.

1. Checks
    a. Skill Check
    b. Observation Check
2. Gates
    a. Global Gate
    b. Info Gate
    c. Kickout Gate
    d. Lockout Gate
    e. Tally Gate
    f. Item/Reusable Gate
3. Tags
    a. Growth
    b. Test
    c. Item

Checks

Skill Check

Dialogue Option Colored:

Mentalist: ***BLUE***

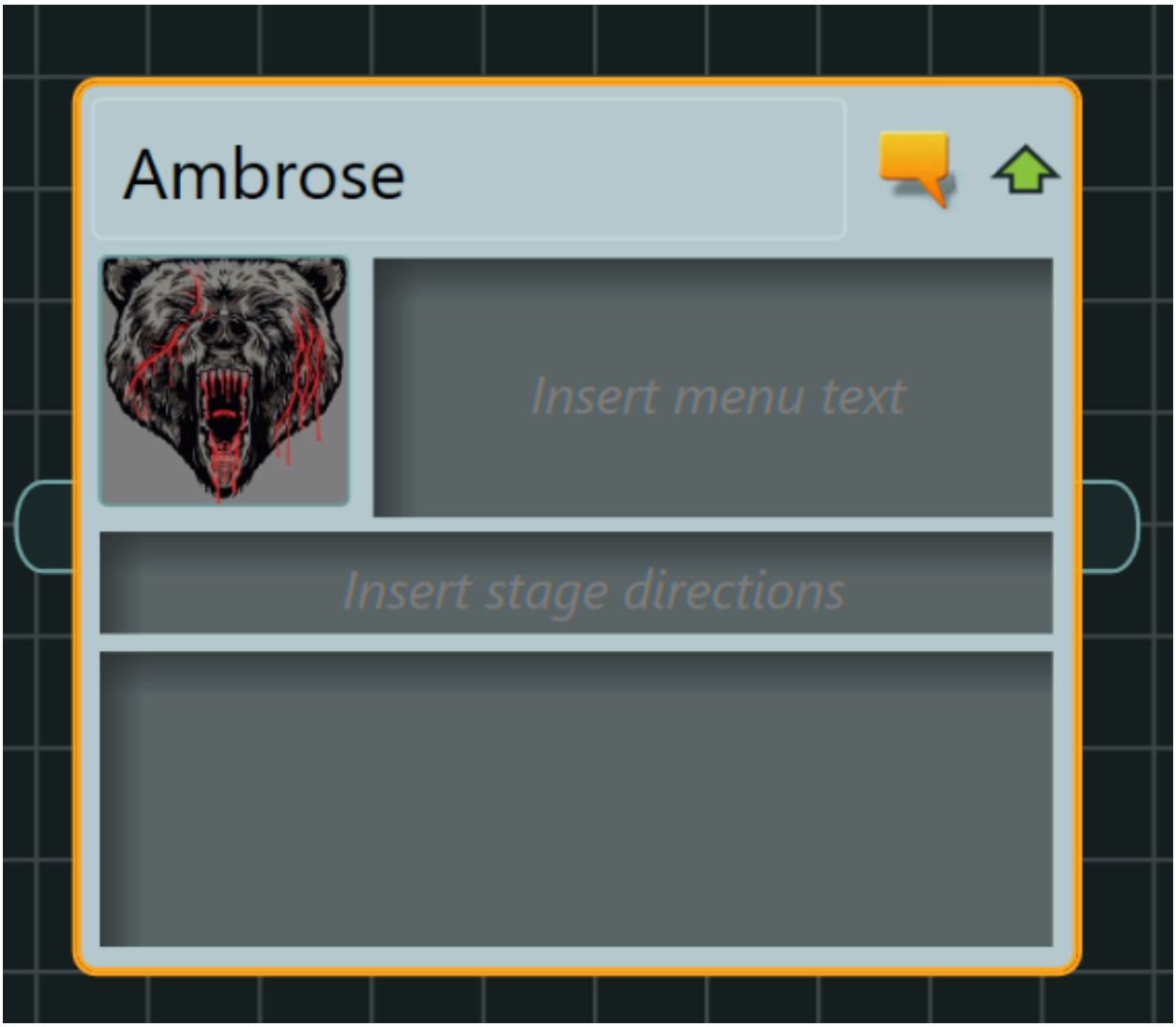
Bargainer: ***PURPLE***

Bureaucrat: ***GREEN***

Primalist: ***RED***

Hardware: ***BLACK***

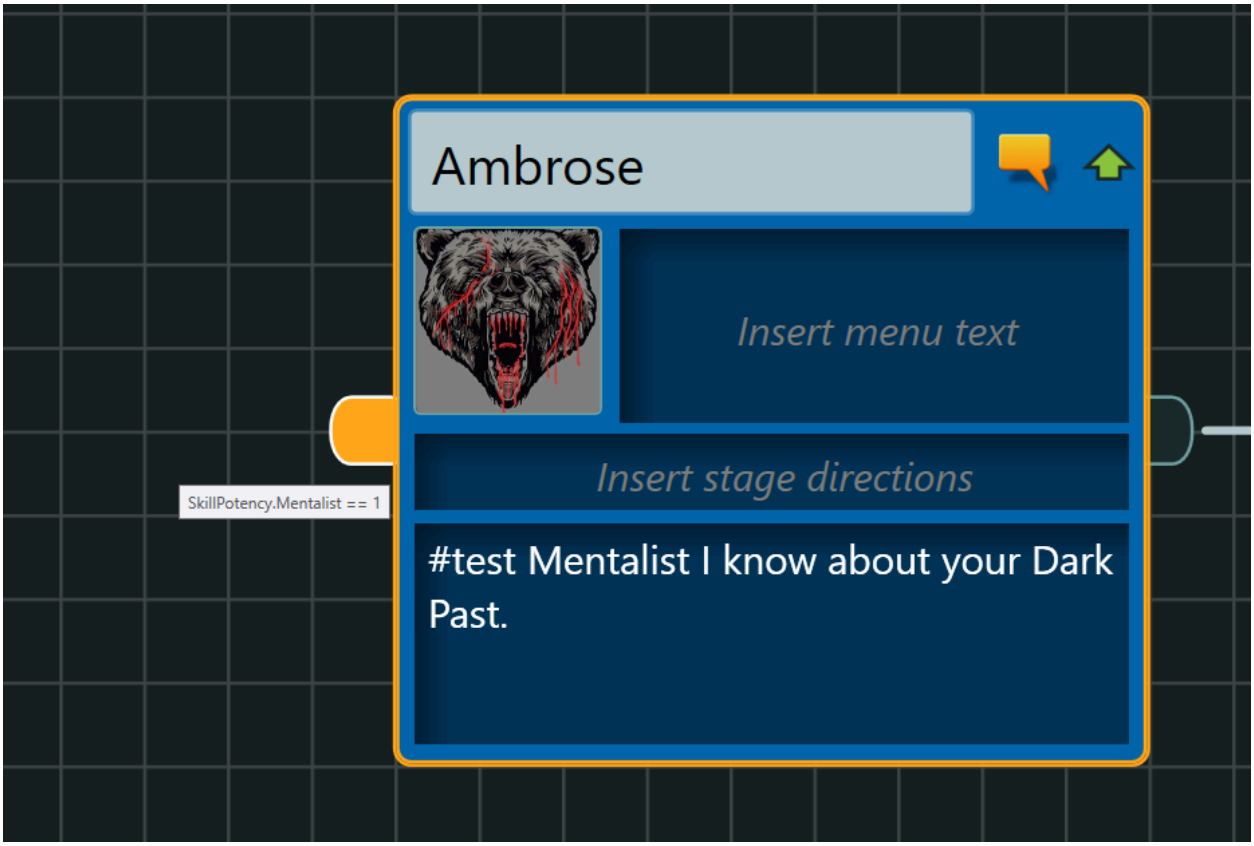
1. A Skill Check where a player's potency value is compared against an expected number. If the potency value of the player's trait is greater than or equal to ≥ the check value, it is passed. If it is lower, it fails.
2. This is the bread and butter of our roleplay system. There is a lot that goes into deciding when to use a skill check and what potency level should be applied. You will need verbal guidance as to our design disciplines in order to be an effective writer. There is no way a document and be comprehensive enough to get you prepared for all necessary gameplay considerations.
3. As an added warning, there are also various ways you can soft lock the player by not adding the necessary exits to the conversation. It is close to impossible to "read" how to prevent this. You will need consistent reviews and conversation to build that instinct.
4. To create a skill check, first start with an Ambrose dialogue fragment:



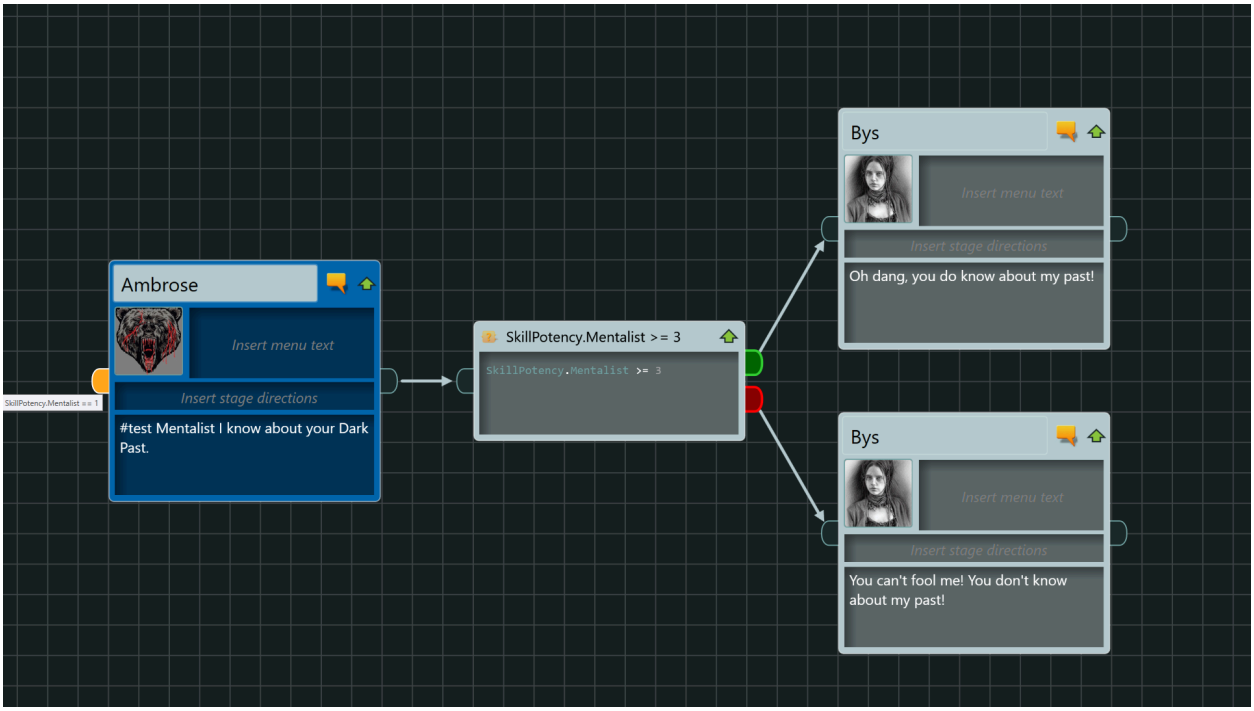
4. Decide which of the five possible checks you wish to use.
5. Color the fragment based on your skill selection.
6. Use the #test command to ID the appropriate skill (more on #test below).
7. Write your intended check like. This option should be narratively written generic enough to indicate the action of what is about to occur, while also communicating the weight of the action.
8. In our game, the four main personality skills can technically be turned off. If the player has done this, those related dialogue lines should NOT appear in their selection box. In the input pin, add your “Is this mod powered?” check in the form of:

a. ~~SkillPotency.[SkillName] == 1~~ This should read SkillPotency.[SkillName]≥1

b. This simply asks if the skill is on at all. If it is, the line will appear.



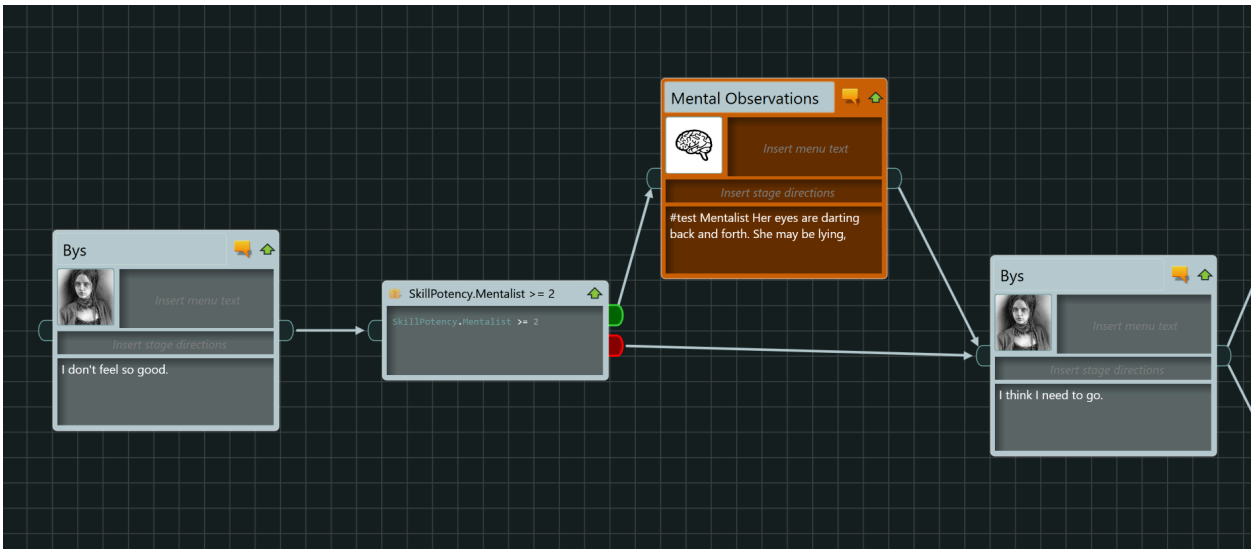
9. From there, we will create a conditional which will provide the main skill check for players. If we want the check to ask “Is the player’s mentalist greater than or equal to 3?”, we would put SkillPotency.Mentalist >= 3



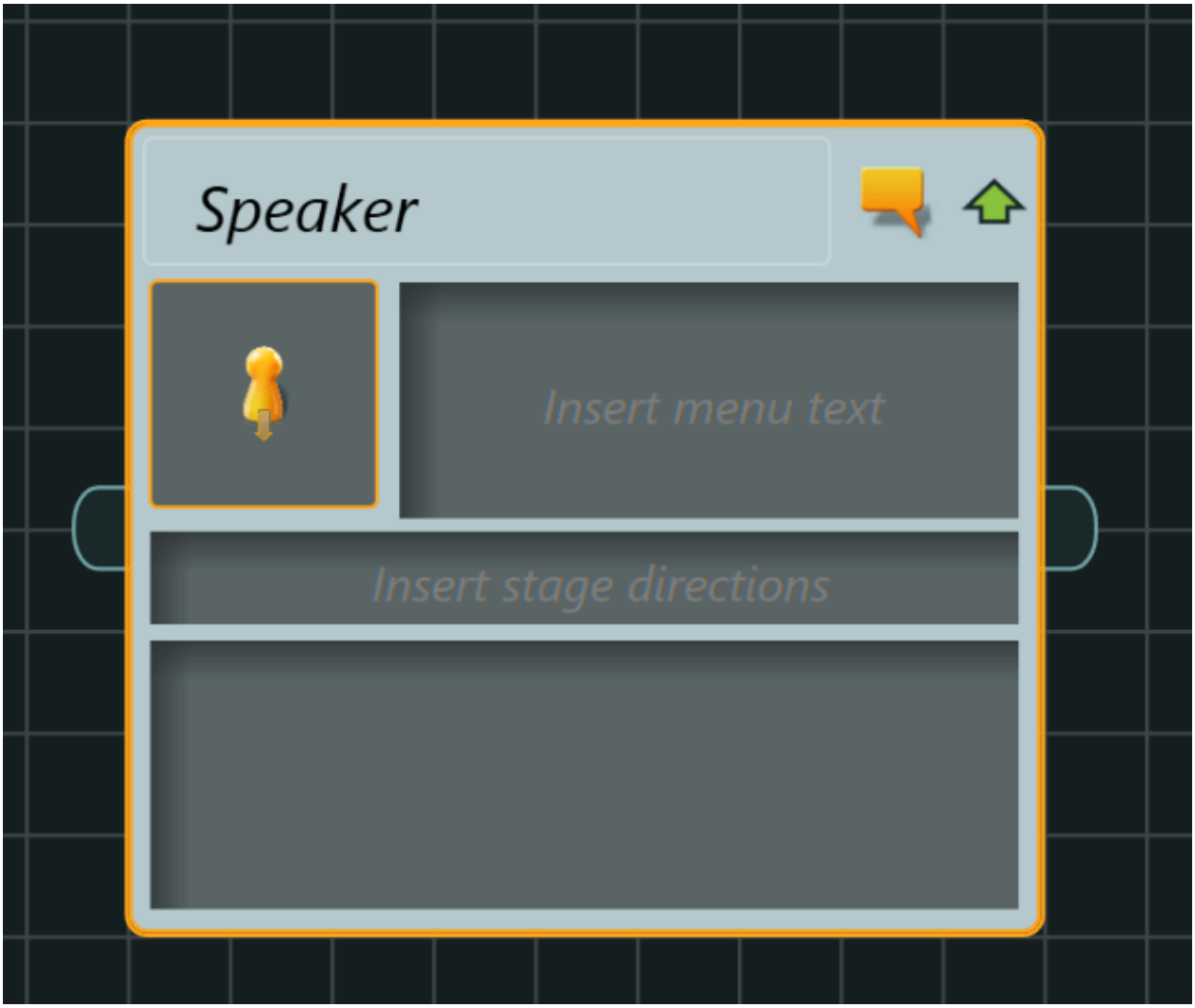
10. The line splits into two responses by the NPC, one being the fail line, the other the success line.
11. This sequencing will allow interesting threads to be explored depending on the effectiveness of the Player's personality trait without being forced to create weaker initial options.
12. SkillPotency variables are connected to our brain map system in Unity. Please do not create any of your own! They will not work. Plus, it's a heavy design decision to create new skills. Imagine just randomly deciding to add another skill to Dungeons and Dragons on a whim, like Dancing. What does dancing get its modifier from? What classes get dancing bonuses? What items assist with dancing? In short, don't randomly add new skills. Talk to a systems designer if you have an idea for one.

Observation Check

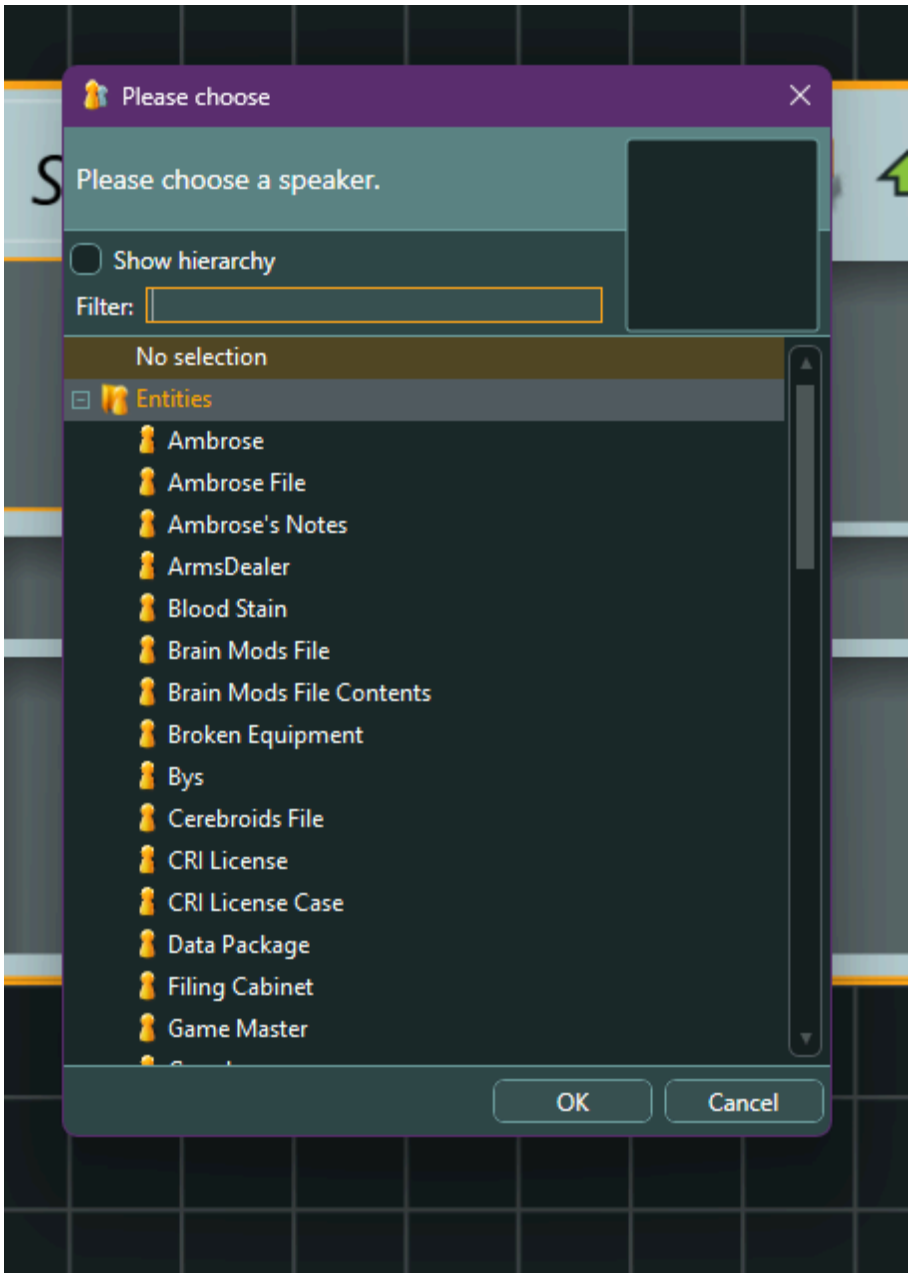
1. Skill Potency is also used to create OBSERVATIONS:



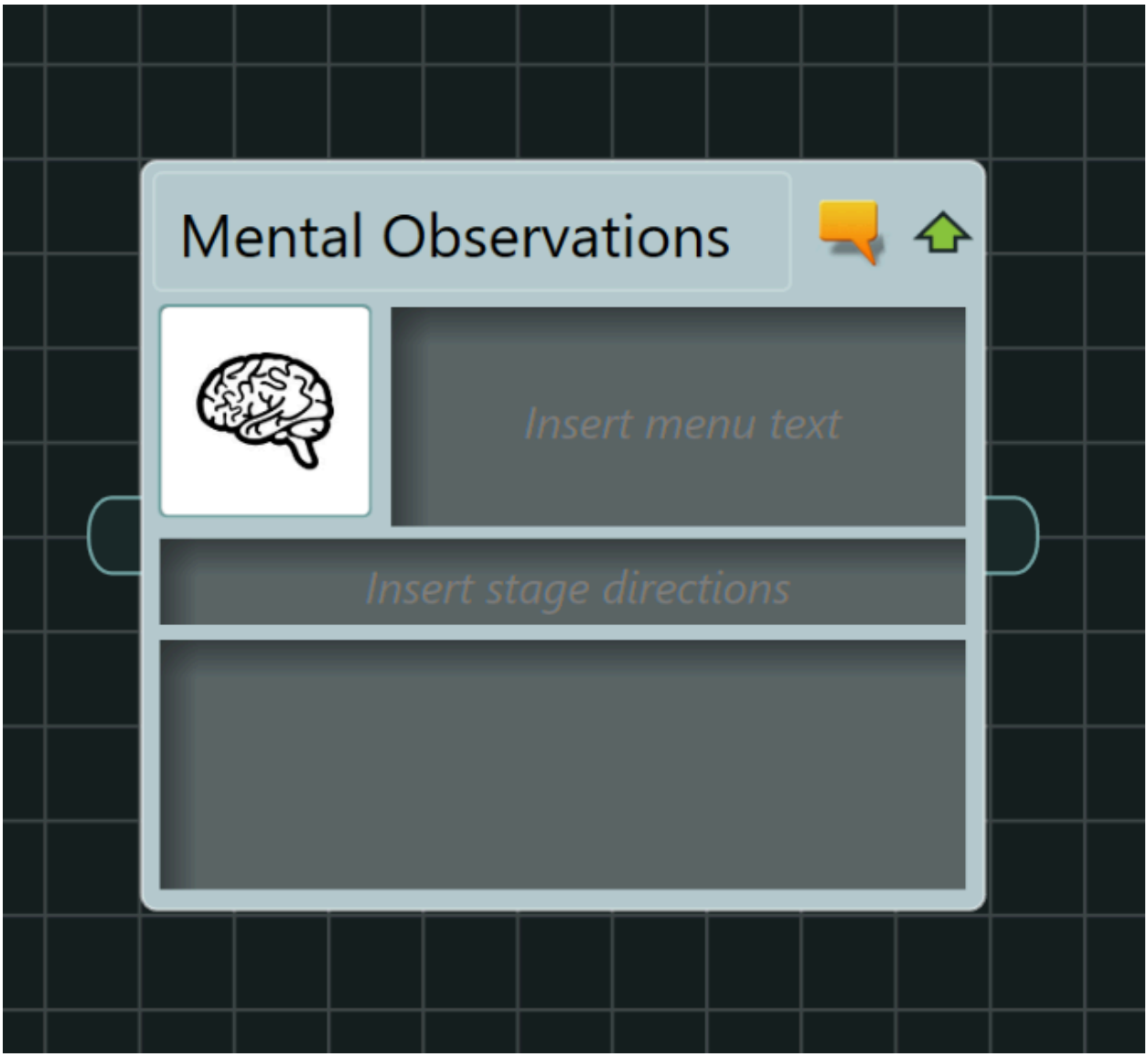
2. Shown above is a simple Observation. By using SkillPotency variables, we can limit what the player can or cannot observe.
3. Observations are grouped into three categories:
    a. Mental Observations
        i. Used for observations relating to psychological or subconscious phenomenon.
            1. Sweating, eyes darting, quiver in voice, unintentional movements, etc.
    b. Physical Observations
        i. Used for observations relating to ones physical appearance.
            1. Dirt under nails, bags under eyes, unkempt hair, blemishes, tattoos, clothing.
    c. Verbal Observations
        i. Used for observations relating to what one says.
            1. Inconsistencies in ones story, lies, knowledge levels on specific subjects.
4. To make an observation, first double click an empty fragment's speaker image.



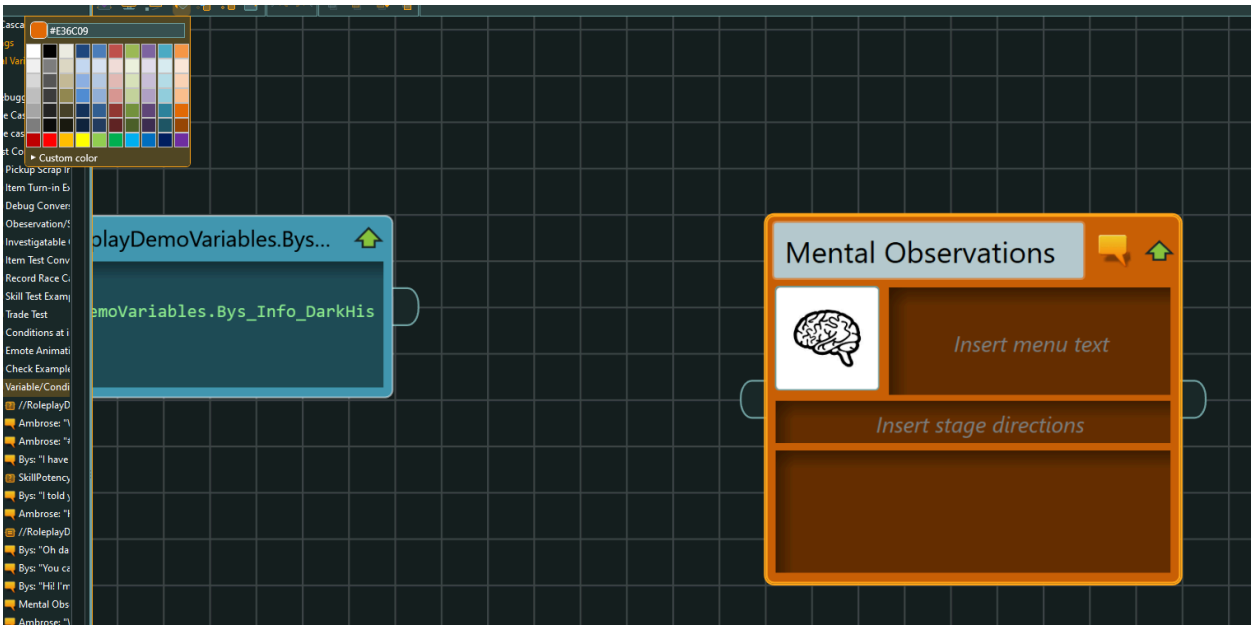
5. It will open up a list of known speaker entities:



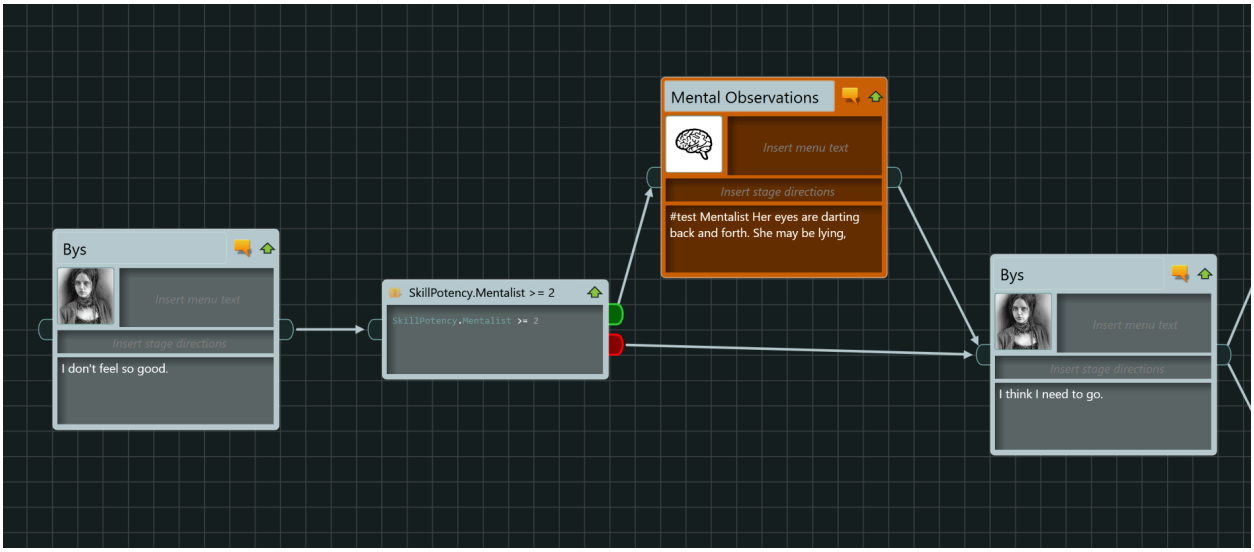
6. Search either Mental, Physical, or Verbal. Select the one you would like:



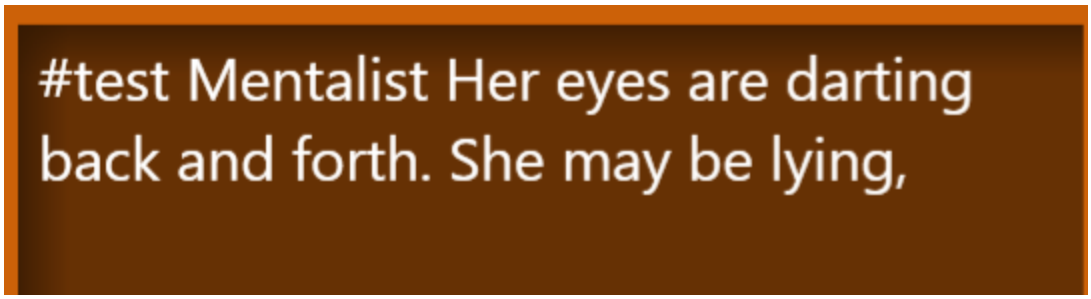
7. Color the fragment orange:



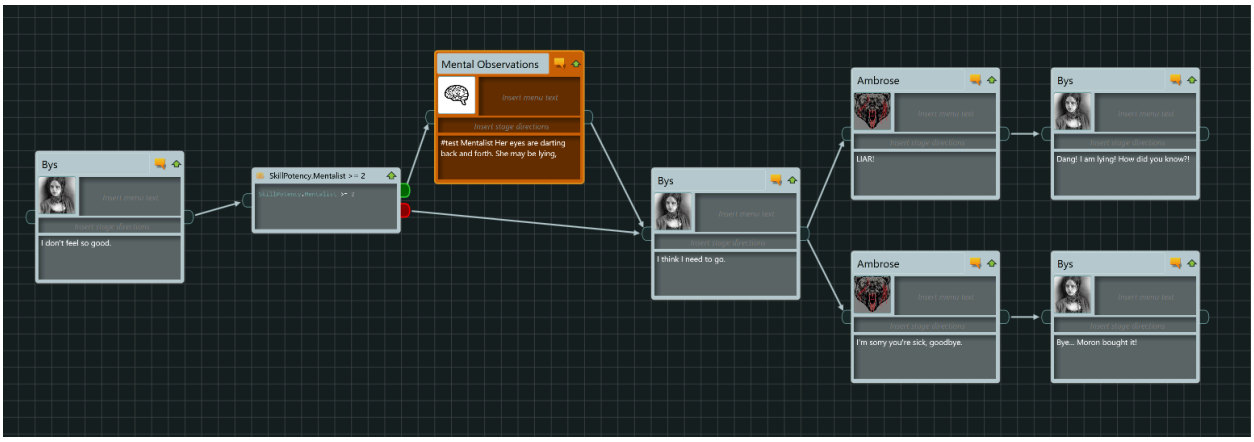
8. Add a condition fragment and add the skill potency check. Connect it to the previous line, the observation, and the following line:



9. Use the #test tag followed by the skill required to activate this observation in the main text body:



10. With this structure, a player with Mentalist 2 will see the observation. A player without mentalist 2 will not see the observation.
11. This could lead to the player making different decisions based on their observations. For instance:



12. In the above case, the player who received the observation might have greater cause to call her a liar. This is an extremely obvious example, but should help you understand how observations play a role in our game.
13. NEVER put the skill check in the input pin of the observation. THIS WILL NOT WORK. You must use a conditional fragment.

Gates

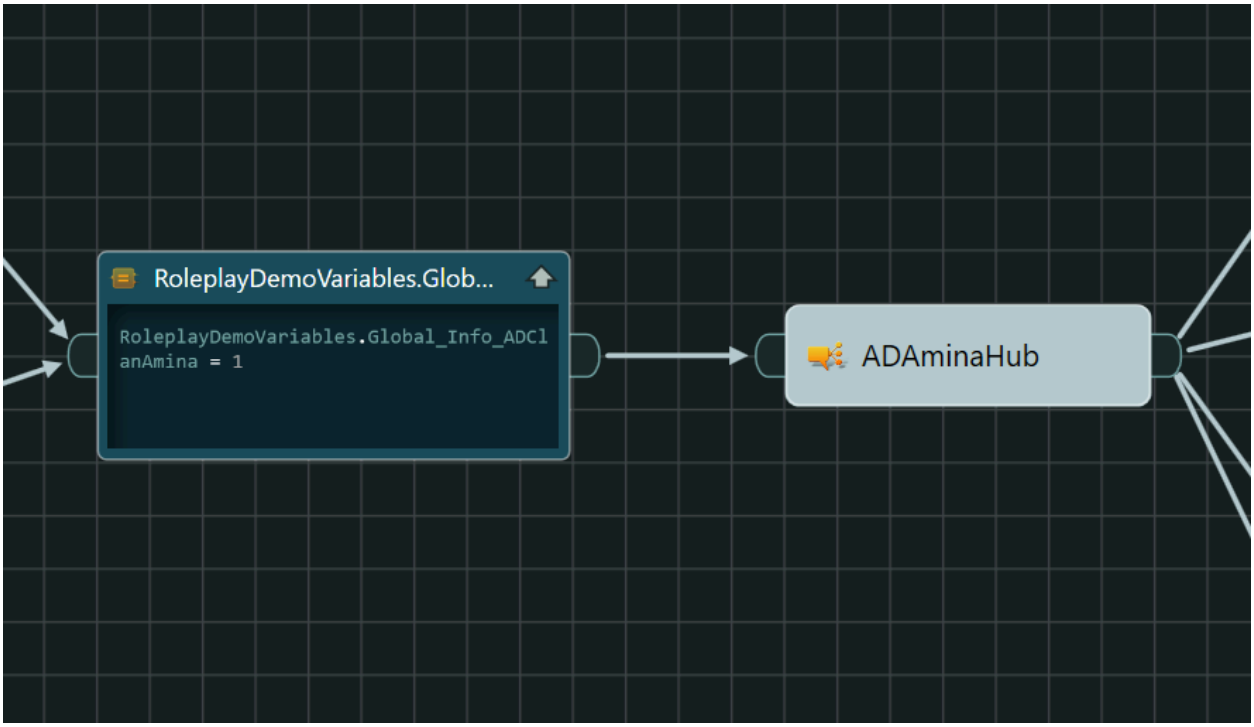
Global Gate

Global Instruction Color: DARK BLUE

Naming Structure: [Variable Set].Global_[Variable Type]_[Variable Name]

Example: RoleplayDemoVariables.Global_Info_ADClanAnima

1. Global Instructions/Conditions are technically the same as Info Instructions/Conditions, only they represent variables that will be used outside of this conversation, usually in the future or with another NPC.
2. For example, if you uncover information by talking to an NPC that the player relay back to a quest giver, it would be a Global Gate.
3. Global Gates have the term “Global” where the NPC name would be.
4. These are the first variables that should be identified during pre-writing for the quest, as they are the foundations of the storyline and subsequent endings.



Info Gate

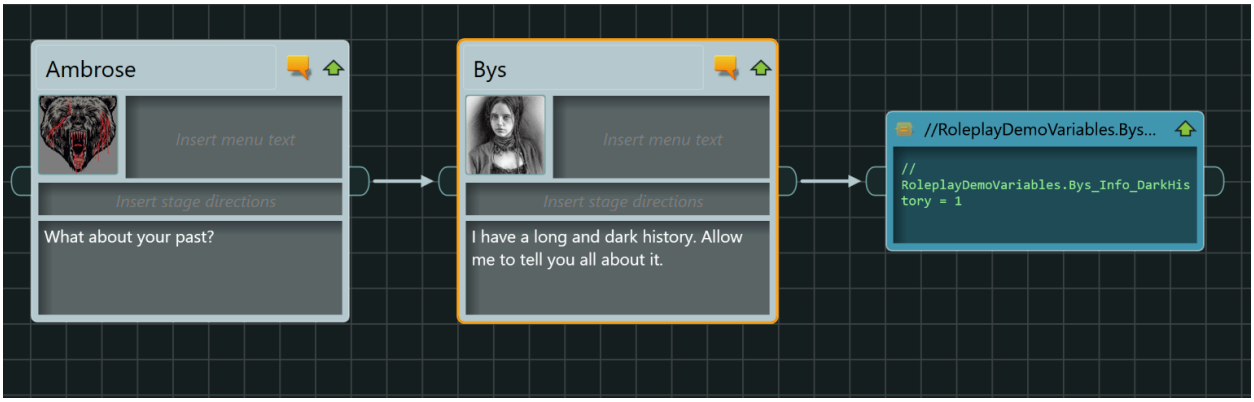
Info Instruction Color: LIGHT BLUE

Naming Structure: [Variable Set].[Name of NPC]_Info_[Variable Name]

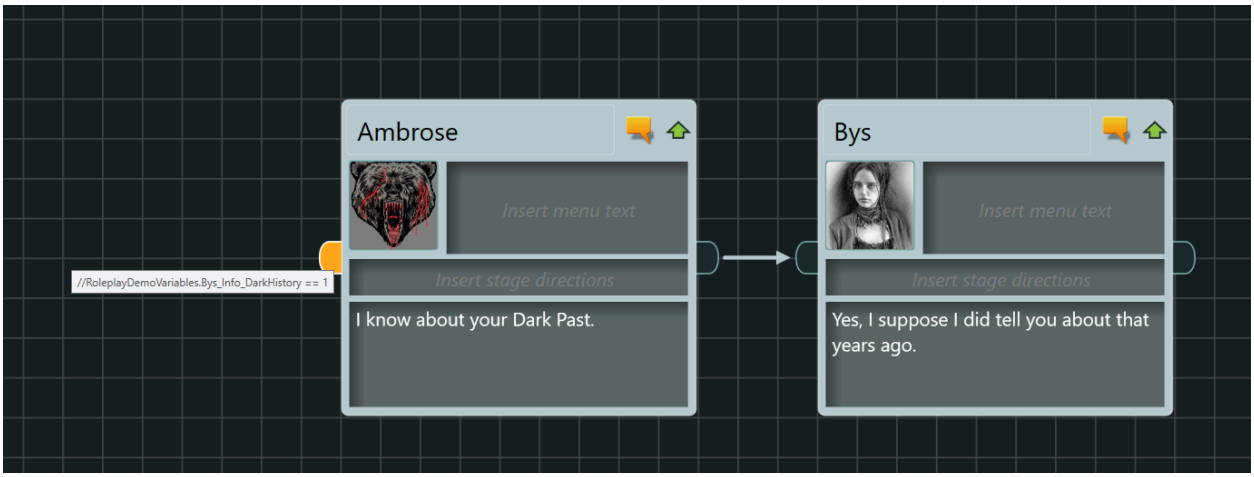
Example: RoleplayDemoVariables.AD_Info_SecretPast

1. An Info Gate check is used when you would like to write a thread that shouldn’t be accessed unless the player has uncovered certain information.

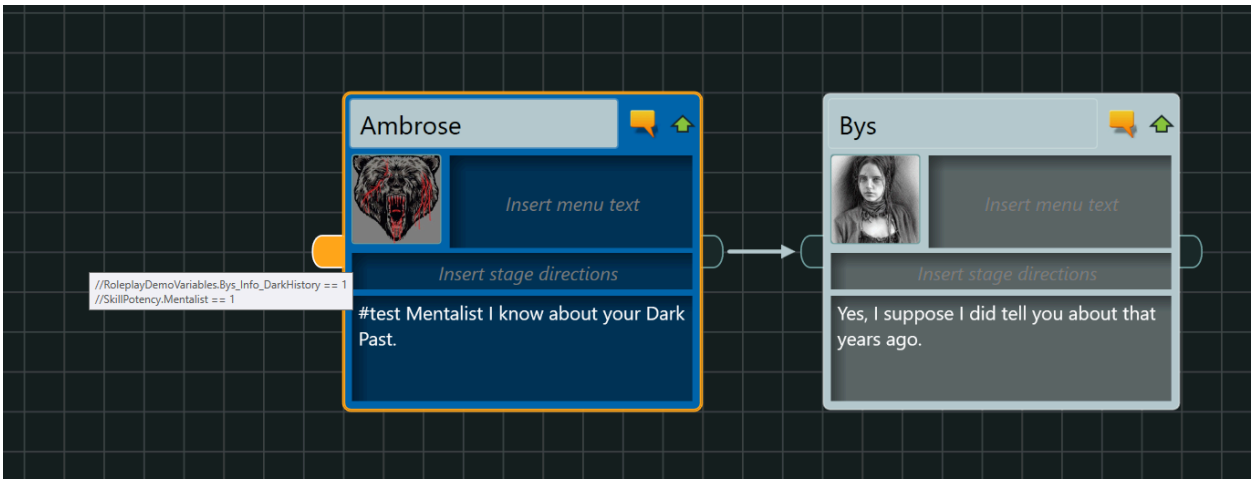
2. For example, lets say you don't want the player to be able to ask a certain question about a person's past unless they've uncovered cause to question said past. In the flow, you would block out the place where the player uncovers the information using an instruction:



3. Then, we use the pin leading into the selection we want gated by the above discovery by placing it in the pin:



3. Potency Checks and Gate Checks can be combined as well:



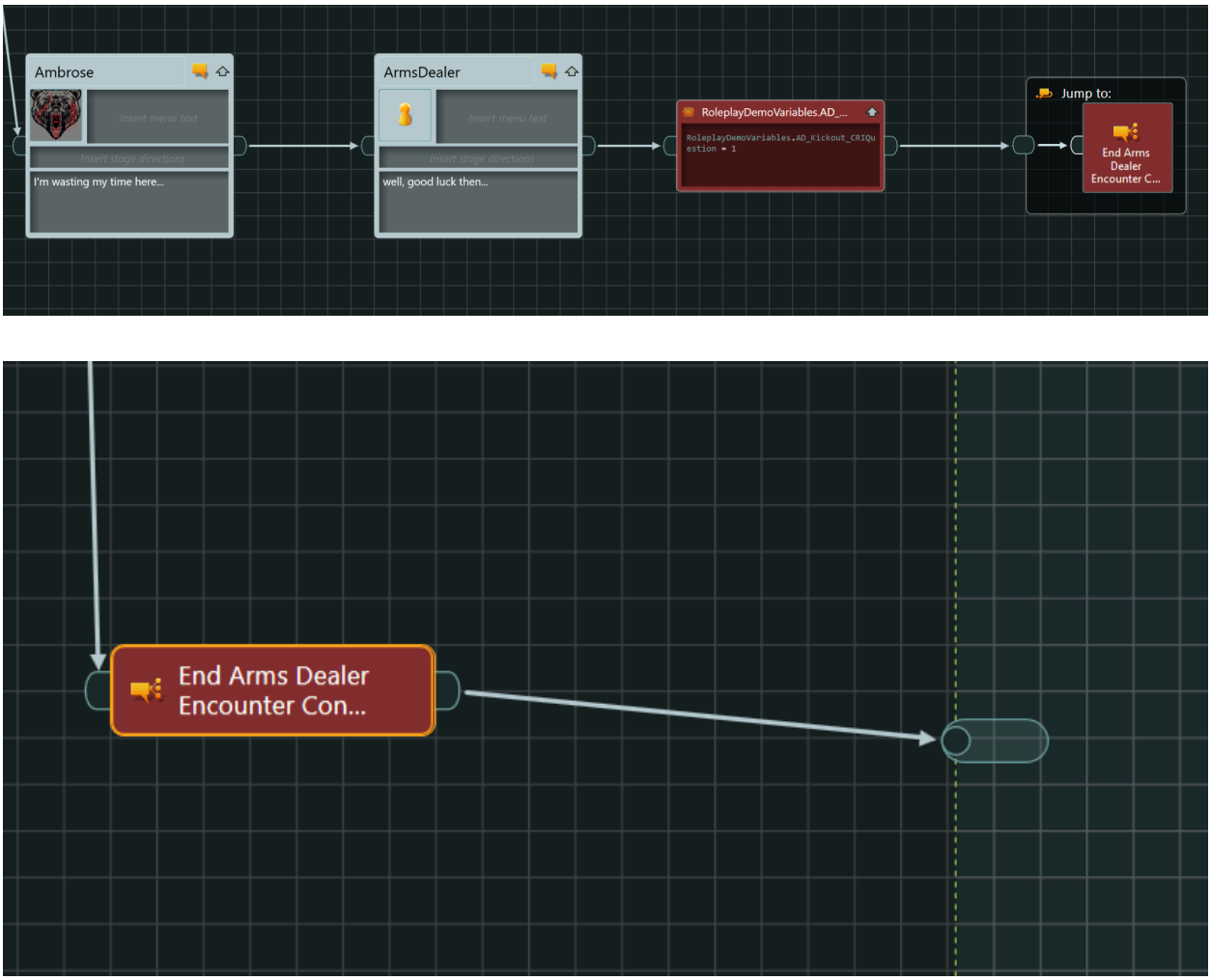
Kickout Gate

Kickout Instruction Color: RED

Naming Structure: [Variable Set].[Name of NPC]_Kickout_[Variable Name]

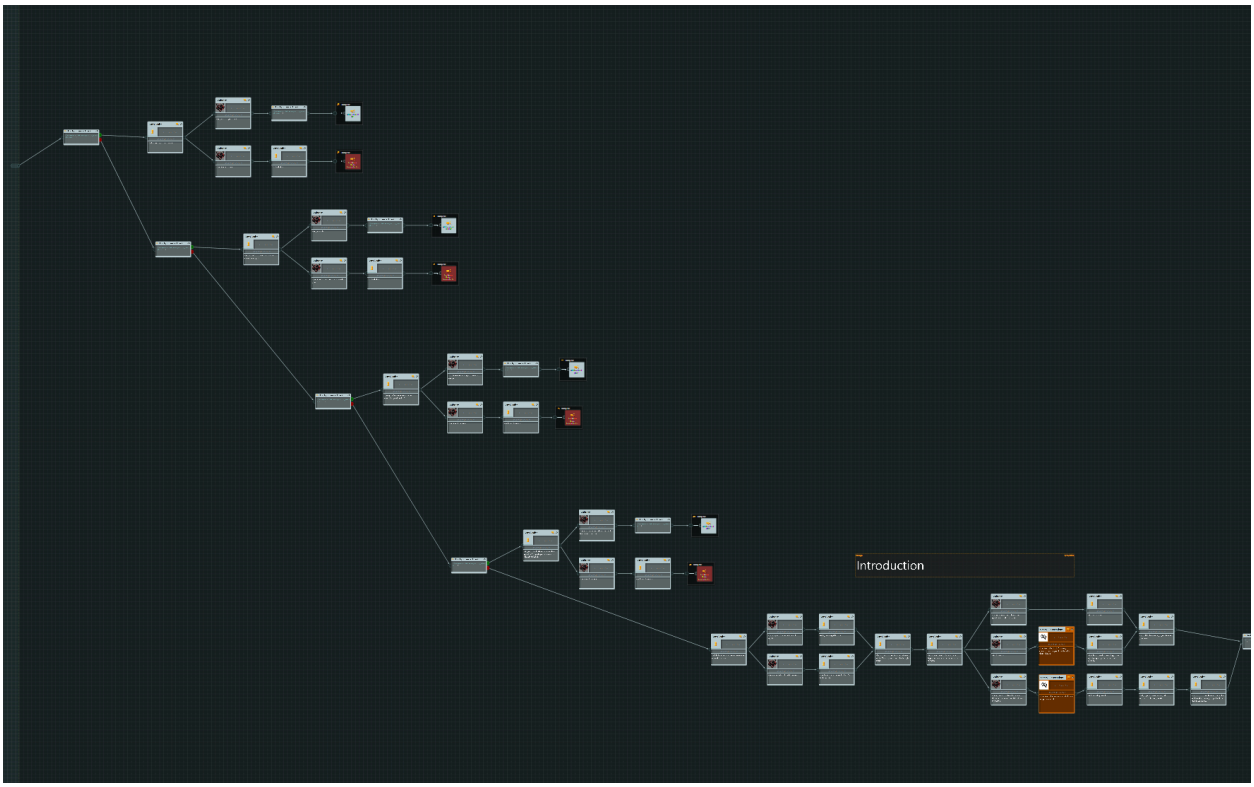
Example: RoleplayDemoVariables.AD_Kickout_CRIQuestions

1. Occasionally, you'll want to kick the player out of a conversation for various narrative reasons. In more complex flows, this could lead to a narrative breakdown as without kickout gates, the player would have to repeat the same lines of dialogue every time they leave the conversation and re-enter it.
2. For example, let's say half-way through the conversation, you want the player to have the option to bail out and adjust their brain maps. Without kickout gates, the player would enter the conversation from the very beginning. That's no good! Therefore, we would create a variable that can allow us to start the player off at a different point in the conversation, creating narrative cohesion.
3. In the below image, you can see a place in which the player can leave the conversation. We write a kickout variable, set it to one, then push the player out of the conversation using a jump hub that ends the whole flow (This is accomplished by putting a hub that is connected to the output pin of the whole dialogue).

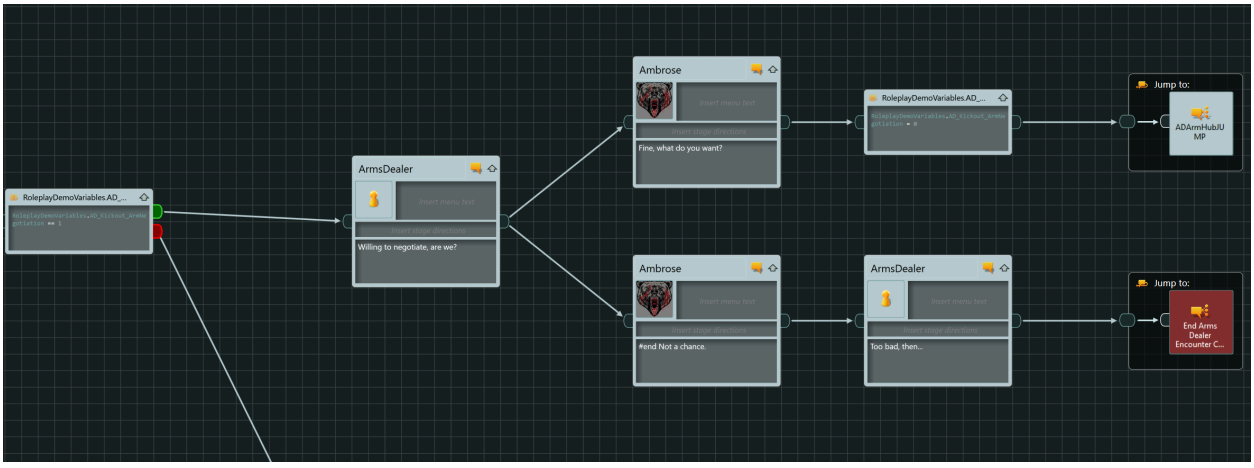


Where the Jump To fragment in the first image leads.

4. From there, we can create a chain of Kickout Gates at the beginning of the dialogue, seamlessly directing the player to the appropriate conversation point depending on where they jumped out.



An overview of what a chained kickout gate sequence may look like. Each thread has a jump that puts the player in the most appropriate point in the story depending on where they exited.



A close-up look as to what one of these may look like

5. The ORDER of these Kickout Gates MATTERS IMMENSLEY. Since they are essentially chained together (shown two images above), if the player has one of those values set == 1, Articy will send them down that path. If you mis order the chain, the player could potentially get sent back to an earlier part of the conversation, completely sequence breaking everything.

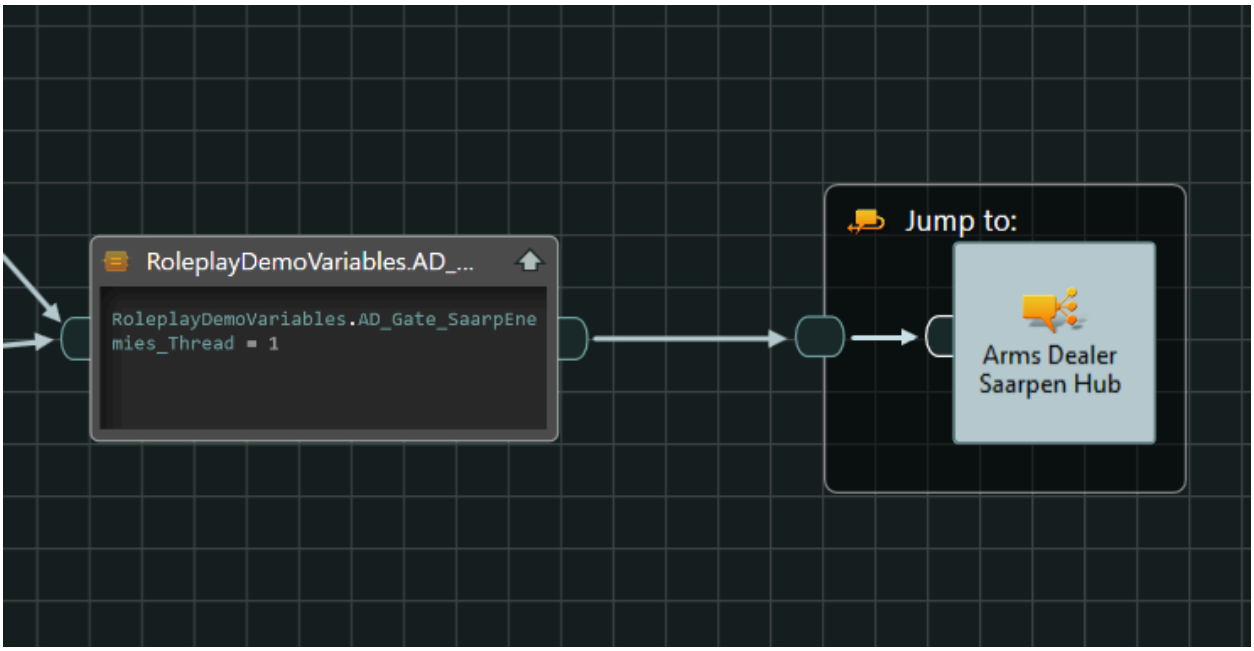
Lockout Gate

***Lockout Instruction Color:** GREY*

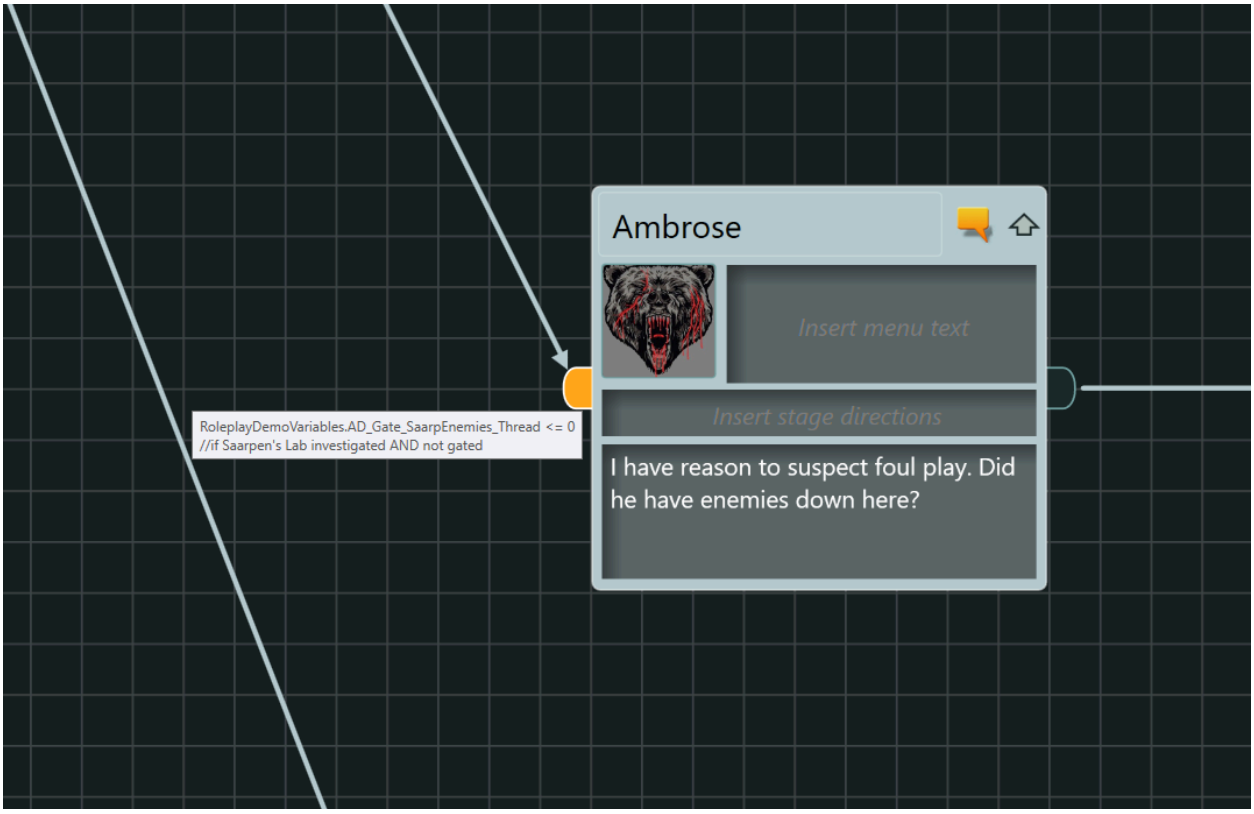
Naming Structure: [Variable Set].[Name of NPC]_Lockout_[Thread Name]_Thread

Example: RoleplayDemoVariables.AD_Lockout_SaarpEnemies_Thread

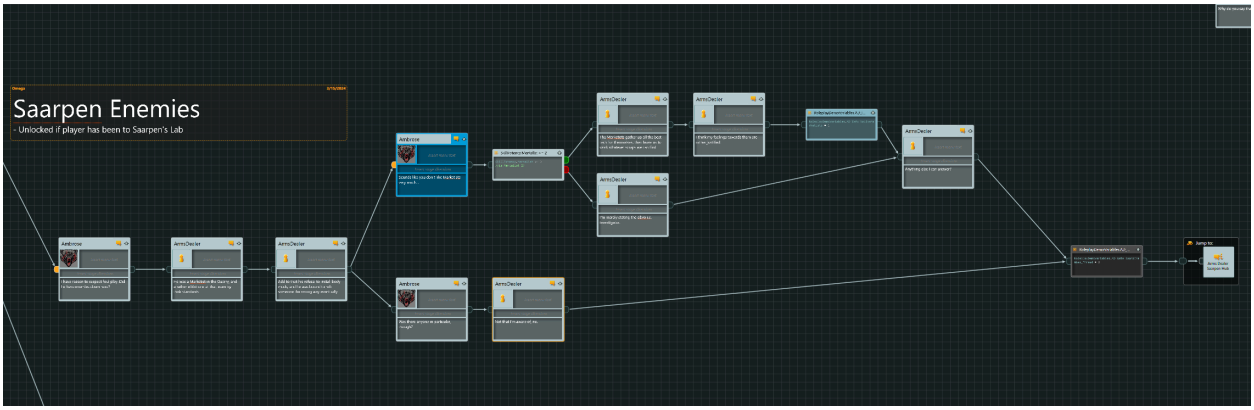
1. Lockout Gates are used if you do not want a player to repeat a thread multiple times. This could be for narrative reasons, or if the player received a growth reward, etc.
2. These gates are often used in sections where the player can return to a root hub which gives the risk of the player repeating lines of questioning that at times should not be returned to.
3. In order to make a lockout gate, simply place an instruction at the end of the thread, often right before the jump fragment.



4. From there, simply add a conditional statement in the pin that prevents the player from returning to that thread.



5. Here is the full view of this thread:



Tally Gate

Additive Instruction Color: YELLOW

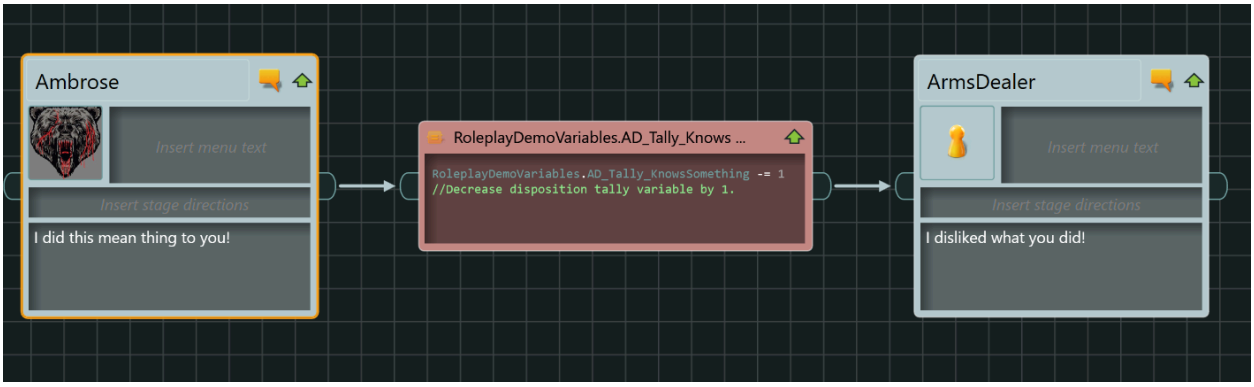
Subtractive Instruction Color: MUAVE

Naming Structure: [Variable Set].[Name of NPC]_Tally_[Variable Name]

Example: RoleplayDemoVariables.AD_Tally_FirstEncDisposition

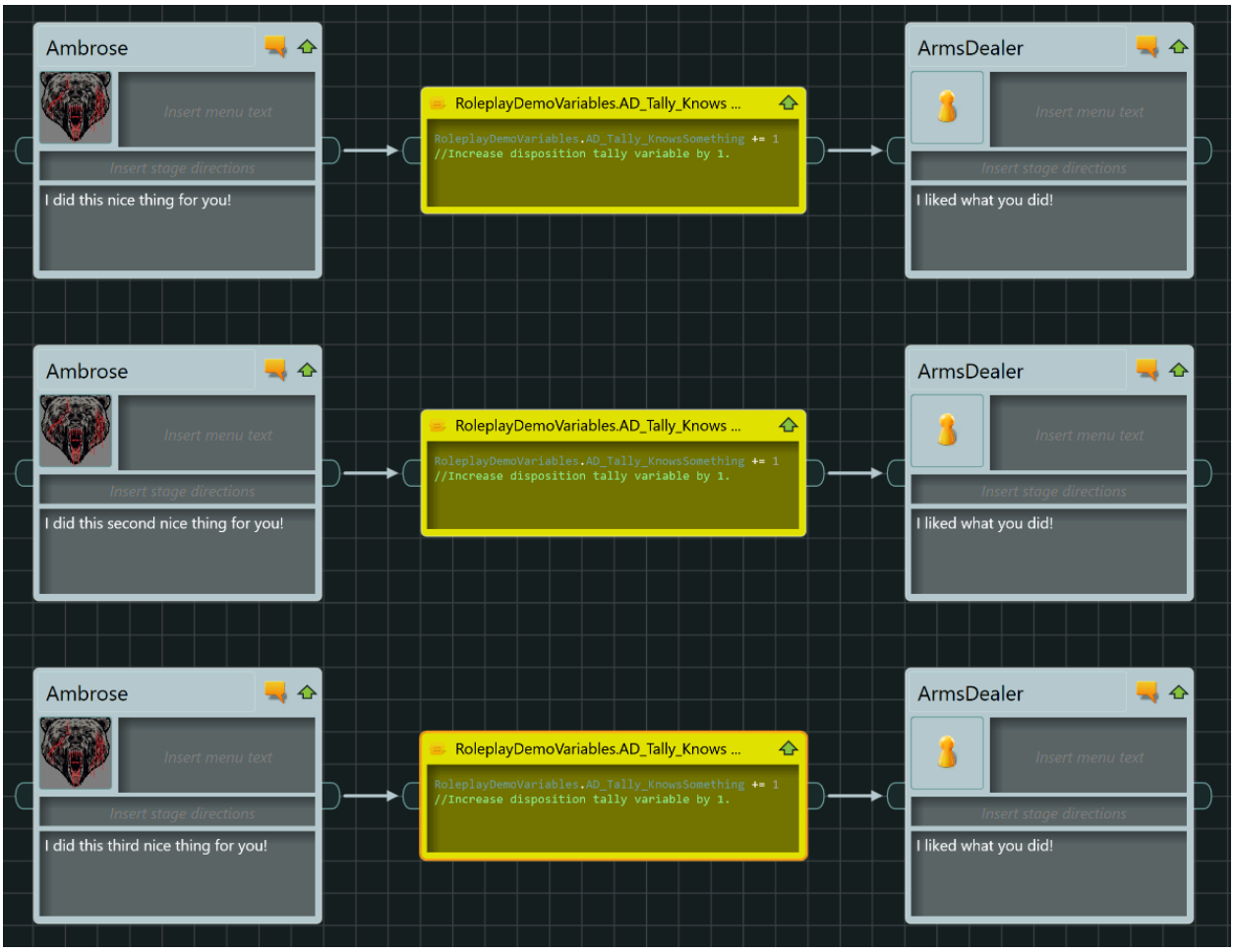
1. Tally gates are used if you wish to keep a running score of players decisions without using a large quantity of variables.
2. For example, let's say you are trying to convince a person you are trustworthy. Each action that can cause a shift in that trust can increase or decrease the variable total. That stat can then be used to flow the player down the appropriate narrative ending.

Increase tally variable example.



Decrease tally variable example.

3. The obvious use case for this type of variable is if there are multiple things a player needs to do in order to access a new thread. For instance, lets say the player needs to say three nice things to the Arms Dealer in order for him to trust them. In the flow, you would see three places in which you could increase the Arms Dealers Disposition:

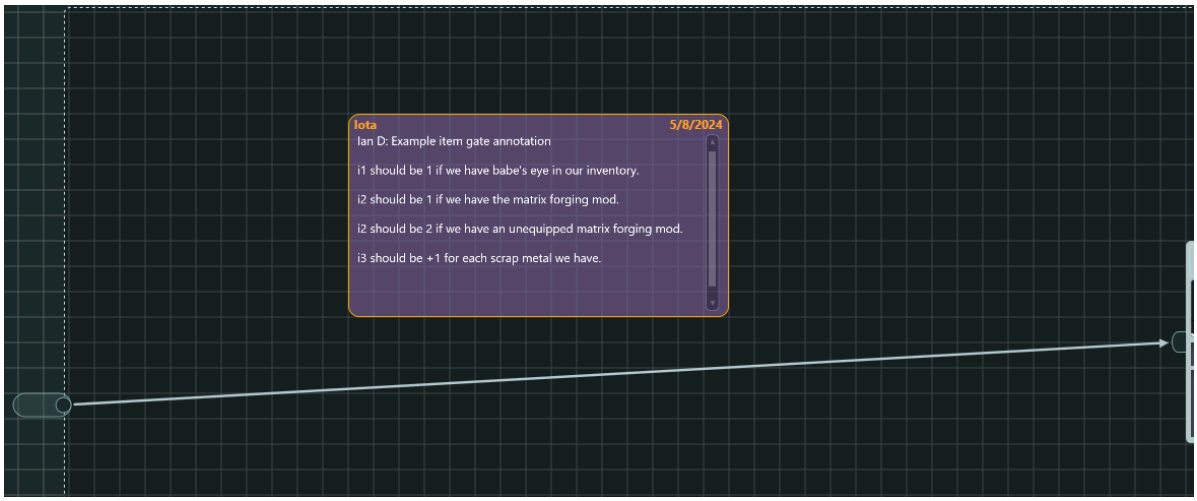


4. There would then be a gate to a line that can only be accessed if the RoleplayDemoVariables.AD_Tally_KnowsSomething variable's value was three:

5. The same can be done for the decreasing tally variable.

Reusable Gates: Items

1. Item gates, and reusable variable gates in-general, should be used when we want to check the status of things that exist outside of dialogue.
2. Each of the item variables in “ReusableVariables” is controlled on a per-conversation basis by scripts on the Unity-side of the project. This allows us to use the same variables in multiple conversations to represent and check vastly different things without creating new unique variables for those checks. Customizing these variables for one conversation won’t affect any other conversation.
3. Item variables, like “ReusableVariables.i1”, can be used to check if there are items in the player’s inventory. There are 10 different item variables (ReusableVariables.i1, ReusableVariables.i2, etc), and each can be customized on a per-encounter basis on the Unity side.
4. When using an item variable, you’ll need to provide some notes to the development team as an annotation at the start of the dialogue flow with a list of which item variables you are using, what item they are checking for, and what values they should be set to if those items are found.



5. Item variables are extremely flexible in how they can be used and counted on the Unity-side. If you are planning an encounter that might be checking multiple combinations of items, it might be best to contact the development team to assist in setting up the item gates. Here are some of the ways we can use reusable item variables.

a. Single Checks: We can check if the player has a specific item, and set one of those variables to a specific integer if the player has it (set i1 to 1 if the player has babe's eye). This is most helpful if we are just checking if the player has a specific item.

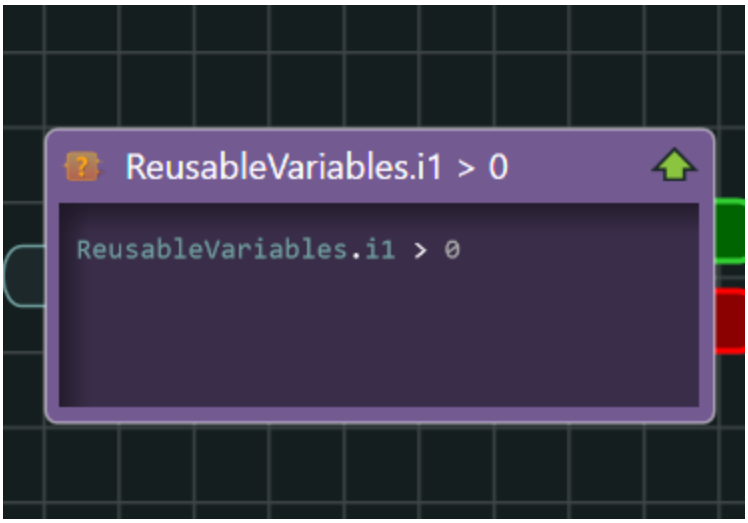
b. Find Best: We can check for a list of items, each with different "values", and set the variable to the highest found item value. (check for babe's eye at 1, armor plate at 2, and special key at 3). This can be helpful if we want to trigger unique dialogue based on items that the player found, but weighted so that the "most important" item is mentioned out of the ones that the player presented.

c. Single Addative: We can check for a specific item, count how many the player has, and add them up to a single integer (set i1 to +1 for each scrap metal). This is handy if we want to check if the player has enough of a specific item.

d. Group Addative: we can check for multiple items, with the same value or different values, and add those values up based on how many of each the player has. This could be useful if we want an NPC to mention that we are traveling light, carrying around a bunch of junk, or carrying a items that are relevant to a particular character's interests.

6. We can also place restrictions on how item variables are counted, depending on why we are checking. If we are checking the player's inventory for items because we plan to take an item from their inventory (like turning in a quest item), we should use a separate item variable to make sure that the player has one that is unequipped. That way the NPC can correctly identify that the player has what they are looking for, but also tell the player that they need to un-equip the item first if they don't have a spare copy. If you want a variable to only include unequipped items, be sure to mention that in the annotation.

7. When implementing an item gate, they behave like any other condition. The simplest item gates would be set up to check for >0 or ==1, but more complex gates can be made depending on what you are checking for. Using light purple for the annotations and the gates will make it much easier for the implementation team to find and test these values. You may want to add a comment in the condition itself to re-iterate what item it's checking for.

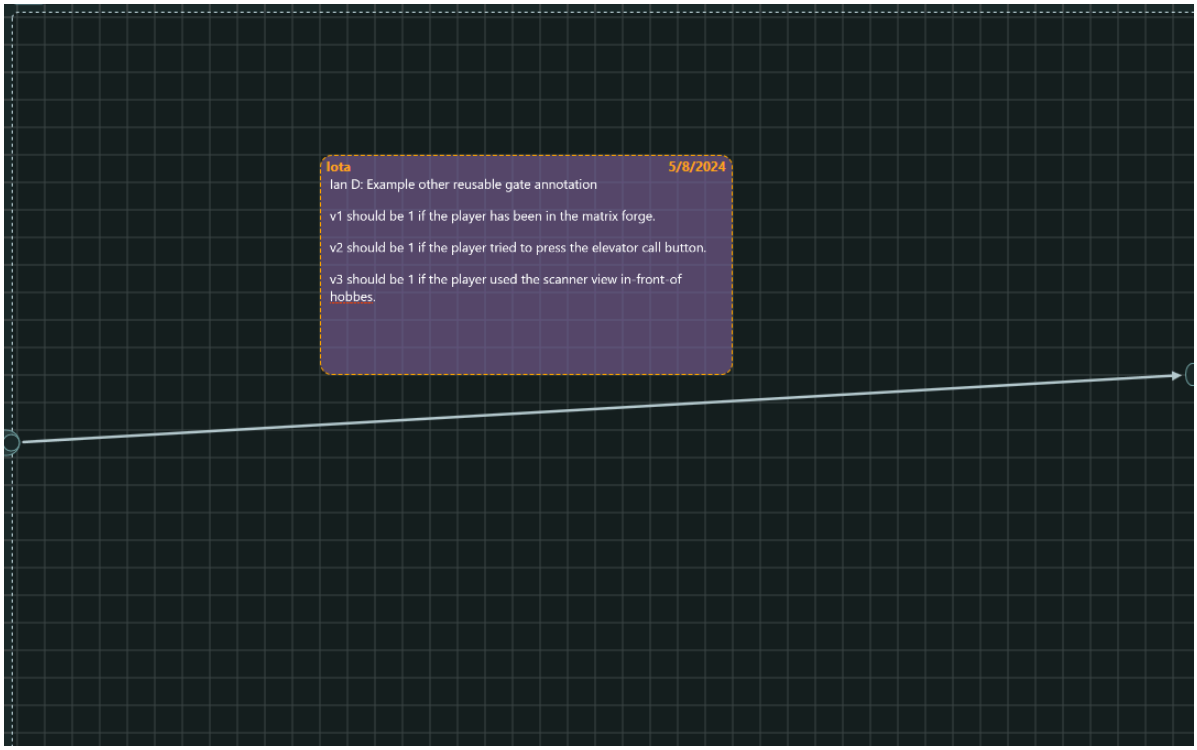


Reusable Gates: Other

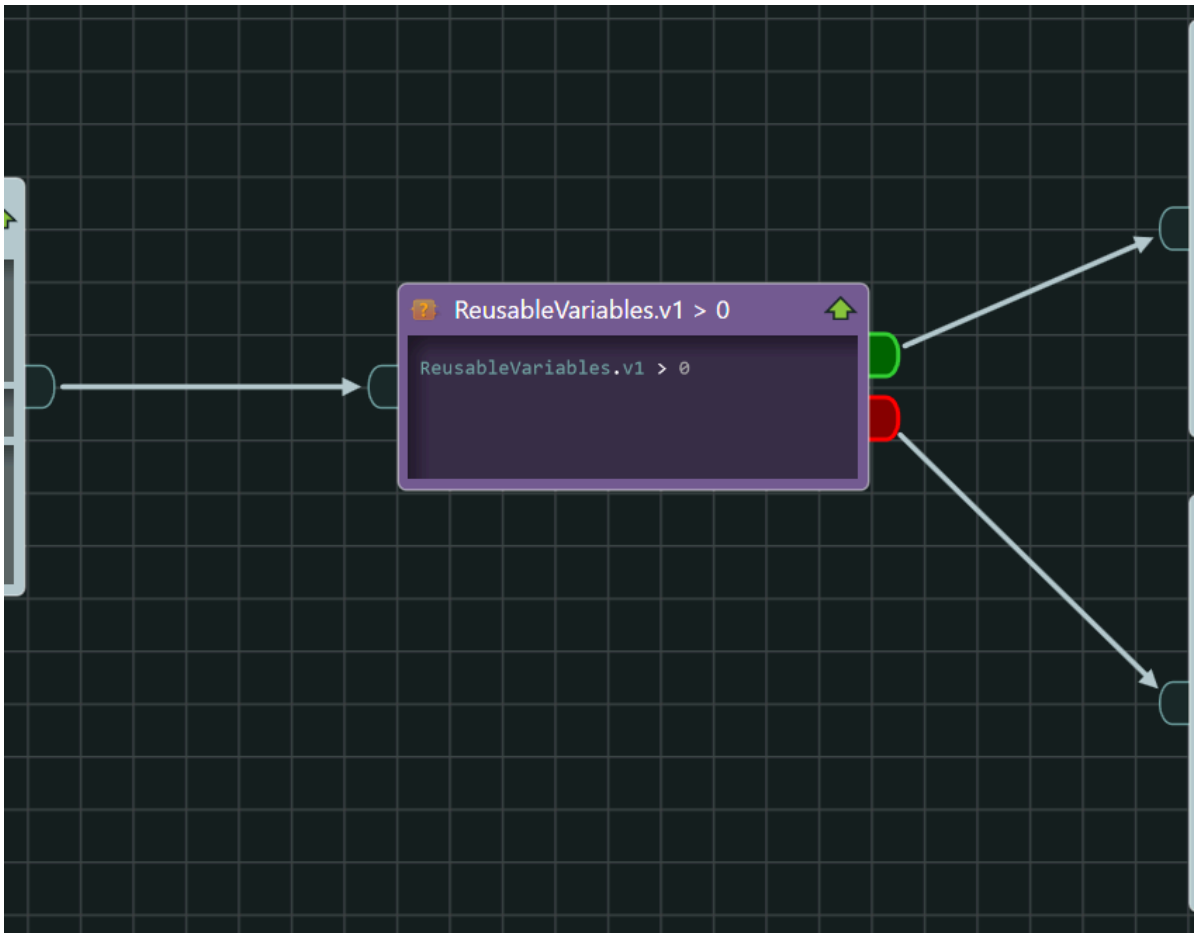
1. There are other reusable variables that can be used to check the status of things that occurred, both inside and outside of dialogue.
2. Like the item variables, there are 10 reusable variables in the set (ReusableVariables.v1, ReusableVariables.v2, etc) and these are also customized on a per-conversation basis.
3. Unlike item variables, these are used to check if specific events have occurred in the game that were recorded to the “Action History” system.
4. You might find that during implementation, some dialogue gets edited to include #record tags along with some additional text. These tags are used by other game systems on the unity side to trigger screen fades, character moves, and to enable or disable game objects after dialogue ends. The messages recorded by that tag can also be accessed and used to set the values of ReisableVariables.v1-v10 in a similar way to the item variables.
5. For the demo version of the game, these other reusable variables should only be used if we need to check if an out-of-dialogue action was performed, like interacting with a simple button, or entering a specific room, etc.
6. Since these out-of-dialogue events might not use the action history system by default, you’ll need to notify the development team if you would like to use one of these out-of-dialogue events to gate part of a conversation. Even something simple might require a unique script or tool in-engine to implement,

and it might be hard to predict the scope without discussing it with the development team first.

7. Like with item variables, you'll want to add an annotation at the start of the dialogue with which reusable v variables you'll be using, and how those values should be set.



8. Using other reusable variables in the flow is just like any other condition. Using light purple for the annotation and for the gates, just like with item variables, will make it easier for the implementation team to find and test these during setup. You might find it helpful to add a comment to the conditions to re-iterate what event or out-of-dialogue thing it's dependent on.



Tags

Tags are design tools that communicate with Unity to make something happen in a custom system.

#growth

1. Growth tags are used to trigger Personality Skill (Mentalist/Bureaucrat/Bargainer/Primalist) level ups.
2. They are limited to the end of major decisions and questlines as a lump-sum reward.
3. They are written inside of the menu text option, and are comprised of four parts:
    a. #growth “NameOfSKill” “Growth Size” “ID Number

i. EX: #growth Mentalist Large 1205

4. The #growth tells Unity what script needs to run.
5. Mentalist tells Unity what size of reward needs to be granted (in essence, how much XP to that skill).
    a. Can use Mentalist, Bargainer, Bureaucrat, or Primalist depending on the narrative context.
6. Large tells Unity what size of reward should be granted.
    a. Can use Small, Medium, or Large
7. 1205 is the identifier of the growth tag.
    a. The first two numbers (12 in this case) represent which dialogue flow this fragment is found in.

b. The last two numbers (05 in this case) represent which growth tag we are looking at in regards to the dialogue. In this case, this is the fifth growth tag of dialogue ID 12.

c. IDs are stored in our Quest Data Master List:

After using a growth tag in a test, record the ID number you used here so that it isn't re-used by other players. You can choose to use a relevant quest step ID as the Growth ID Number

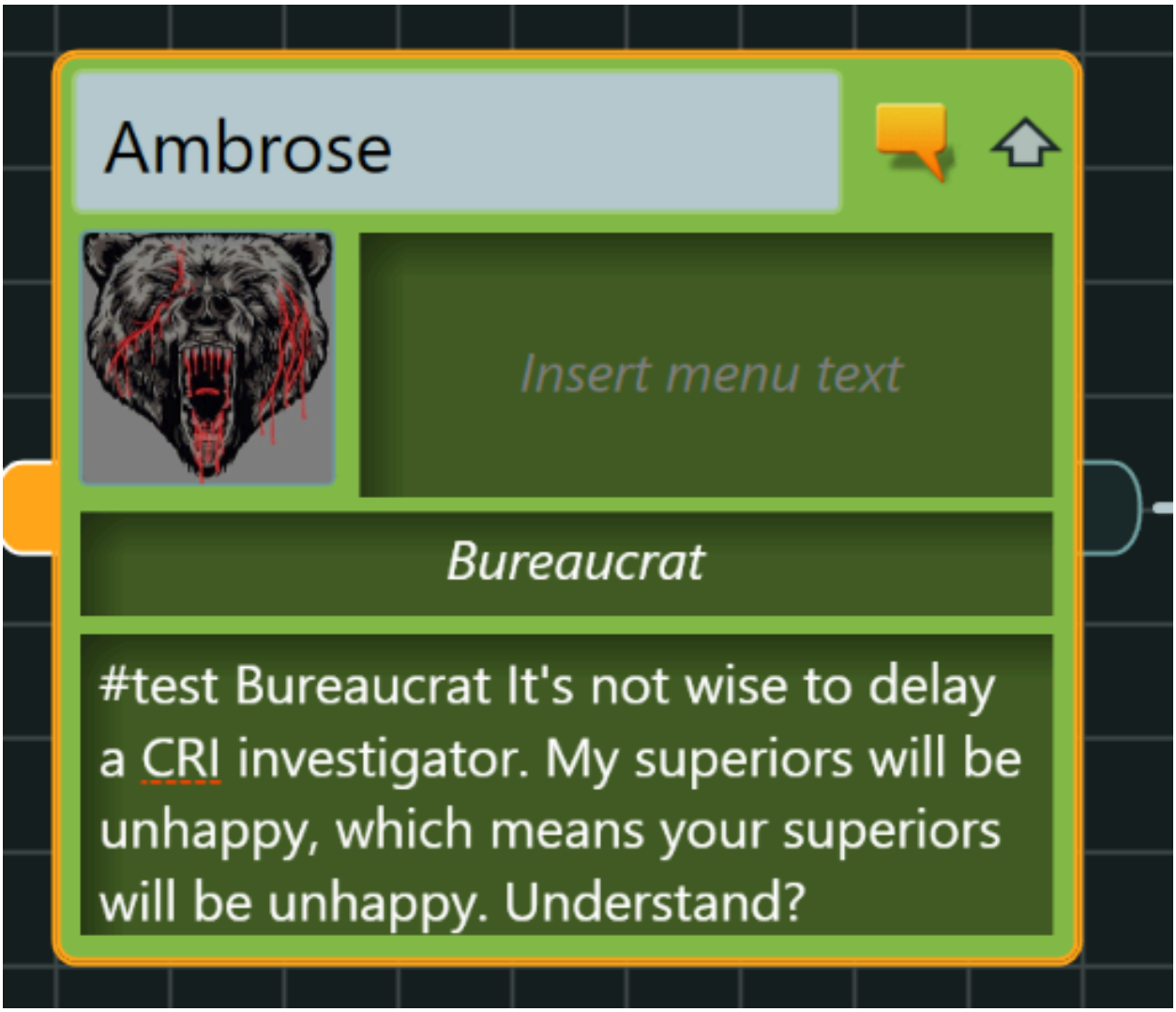
Growth ID Number	Articy Conversation Name
-9999	(name of articy conversation node where the code is used)
1201	Guard Encounter pt.1
1202	Guard Encounter pt.2
1203	Guard Encounter pt.2
1204	Guard Encounter pt.3
1205	Guard Encounter pt.3
1301	In Hobbes' Office pt.1
1302	In Hobbes' Office pt.2
1303	In Hobbes' Office pt.3
1304	In Hobbes' Office pt.3
1305	In Hobbes' Office pt.3
1306	In Hobbes' Office pt.3
1401	Arrival at Saarpen's Lab
1501	Hobbes after First Saarpen Visit
1502	Hobbes after First Saarpen Visit
1503	Hobbes after First Saarpen Visit
1504	Hobbes after First Saarpen Visit
1601	Bys Initial Encounter pt.1
1602	Bys Initial Encounter pt.3
1603	Bys Initial Encounter pt.3
1701	Arms Dealer Encounter
1702	Arms Dealer Encounter
1703	Arms Dealer Encounter
1704	Arms Dealer Encounter
1705	Arms Dealer Return Encounter
1706	Arms Dealer Return Encounter
1801	Arms Dealer Return Encounter
1802	Arms Dealer Return Encounter
1803	Arms Dealer Return Encounter
1804	Arms Dealer Return Encounter
1805	Arms Dealer Return Encounter
1901	Bys Return Encounter
1902	Bys Return Encounter
1903	Bys Return Encounter
2001	Hobbes after Returning to Saarpen's Lab
2002	Hobbes after Returning to Saarpen's Lab
2101	At the Safe
2102	At the Safe
2201	Entering the Hidden Chamber
2202	Entering the Hidden Chamber
2203	Entering the Hidden Chamber
2301	OBJECT Terminal System
2401	Deciding Hobbes' Fate

Conversation IDs	Conversation Node
11	Ambrose in the Tunnel
12	Guard Encounter
13	In Hobbes' Office
14	First Visit to Saarpen's Lab
15	Hobbes After First Saarpen Visit
16	Bys Initial Encounter
17	Arms Dealer Ecounter
18	Arms Dealer Return Encounter
19	Bys Return Encounter
20	Hobbes After Returning to Saarpen's Lab
21	At the Safe
22	Entering the Hidden Chamber
23	OBJECT Terminal System

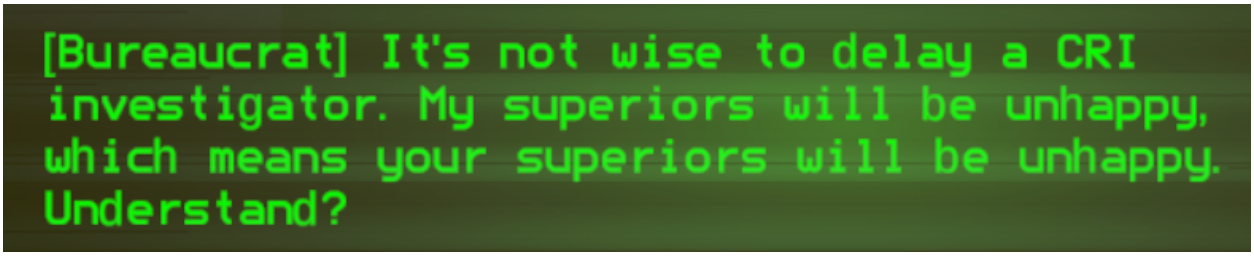
8. A game designer would be the usual person to keep up with the IDs, reward sizes, etc. What YOU need to do is mark where you feel the best place for the reward should be and what Personality Skill should receive that reward.

#test

1. The “Test” tag is not named in a clear way, but it’s function is imperative.
2. The #test tag tells unity to put brackets around whatever follows the #test tag:



How it appears in Articy.

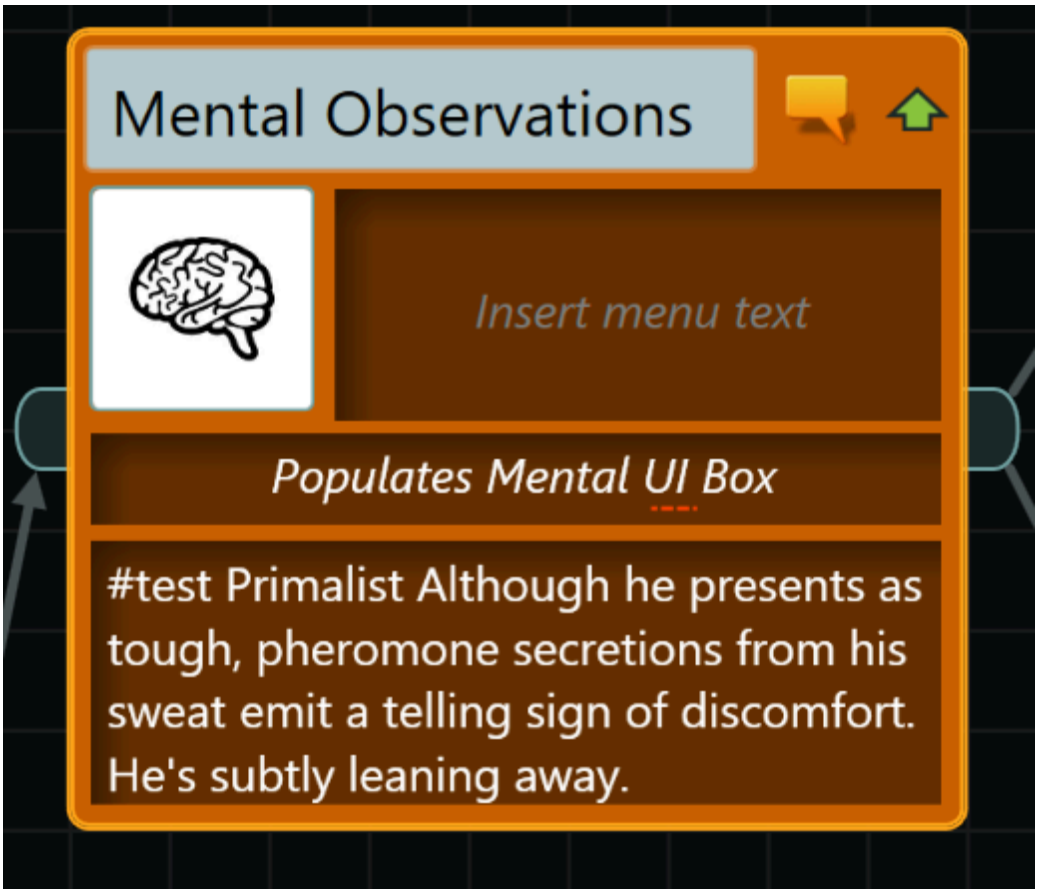


How it appears in Unity.

3. The following are the usable #test tags.

Test Tag syntax	replacement text	Date requested	Date added to project
#test primalist	[Primalist]	n/a	11/10/2023
#test mentalist	[Mentalist]	n/a	11/10/2023
#test bureaucrat	[Bureaucrat]	n/a	11/10/2023
#test bargainer	[Bargainer]	n/a	11/10/2023
#test humanoidMedicine	[Humanoid Medicine]	n/a	11/10/2023
#test matrixForging	[Matrix Forging]	n/a	11/10/2023
#test intellianClans	[Intellian Clans & Customs]	n/a	11/10/2023
#test shiftSequencing	[Shift Sequencing]	n/a	11/10/2023

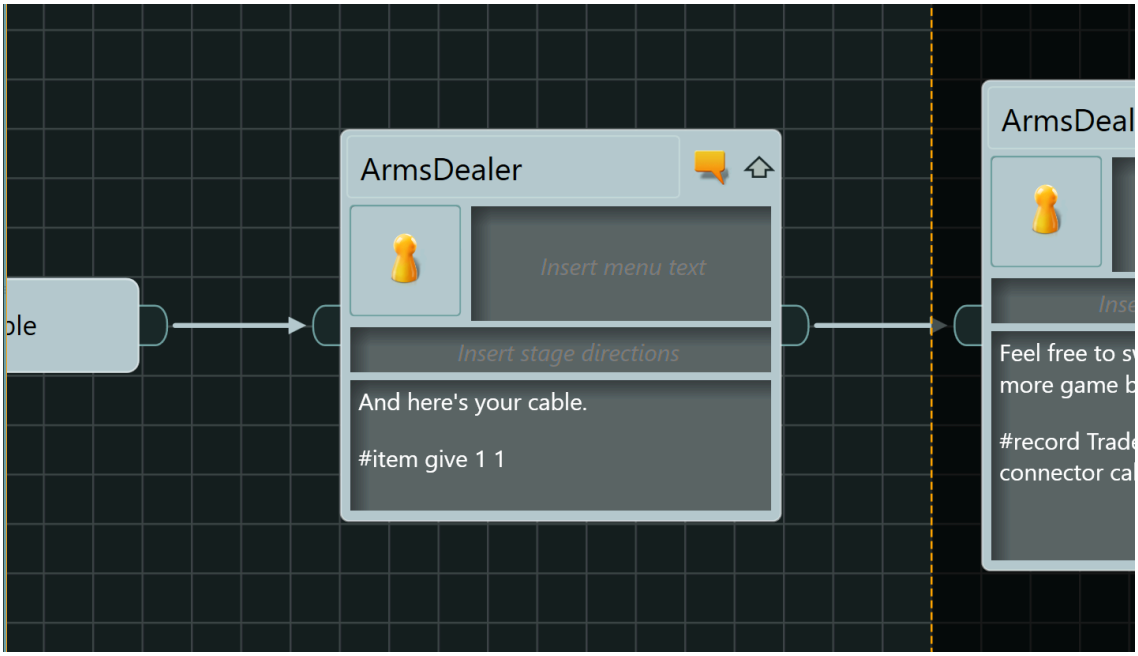
4. You need to use #test tags any time a personality skill or hardware mod is being used for a check or an observation:

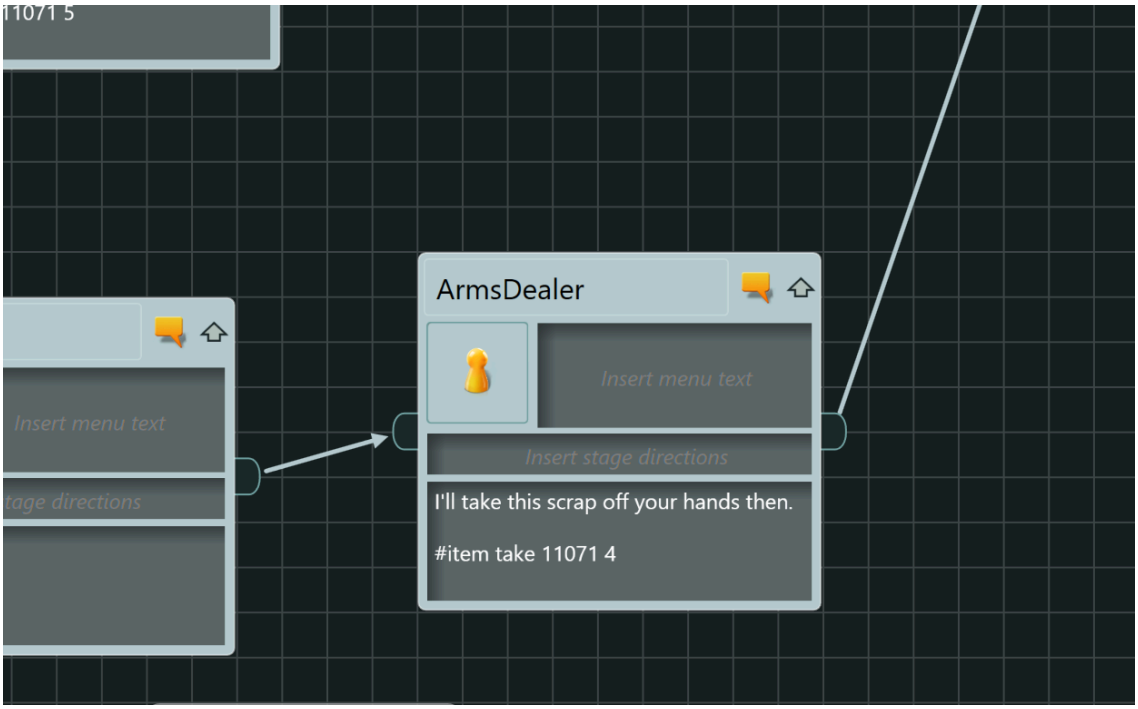


#item

1. Item tags are used to add or remove items from the player's inventory during dialogue.
2. An item tag has 4 main parts: the tag, command, item code, and quantity. An example of a properly implemented item tag: "#item take 11073 1"
    a. Tag: When we put #item in the dialogue box, the tag parser on the Unity-side is expecting it to be followed by the other parts of the item tag, separated by spaces.
    b. Command: the next part of the tag needs to be either "give" or "take". Using "give" will add the item(s) to the player's inventory. Using "take" will attempt to remove the item(s) from the player's inventory. It's best practice to make sure the player actually has the items, unequipped, before taking them using the reusable item variables as shown above. However, if we attempt to take items from the player that they don't have, it shouldn't

cause any errors. The “give” command will also attempt to remove the given items from the NPC’s inventory if they have one, but the player will always receive the items regardless of whether the NPC has the items to give or not.





c. Item Code: the next part of the tag is the item's ID value, which is an integer implemented on the unity-side of the project. The Item ID's are also on the Quest Data Master List (In the Inventory Objects List sheet), but be sure to highlight any changes or additions you make to the list and include a date with the changes, since those will have to be edited in-engine on the Unity side by the implementation or development team.

d. Quantity: the last part of the tag is the number of items that are being given or taken. This needs to be an integer value above zero. This can't be skipped even if we are only giving or taking a single item, otherwise it will cause errors.

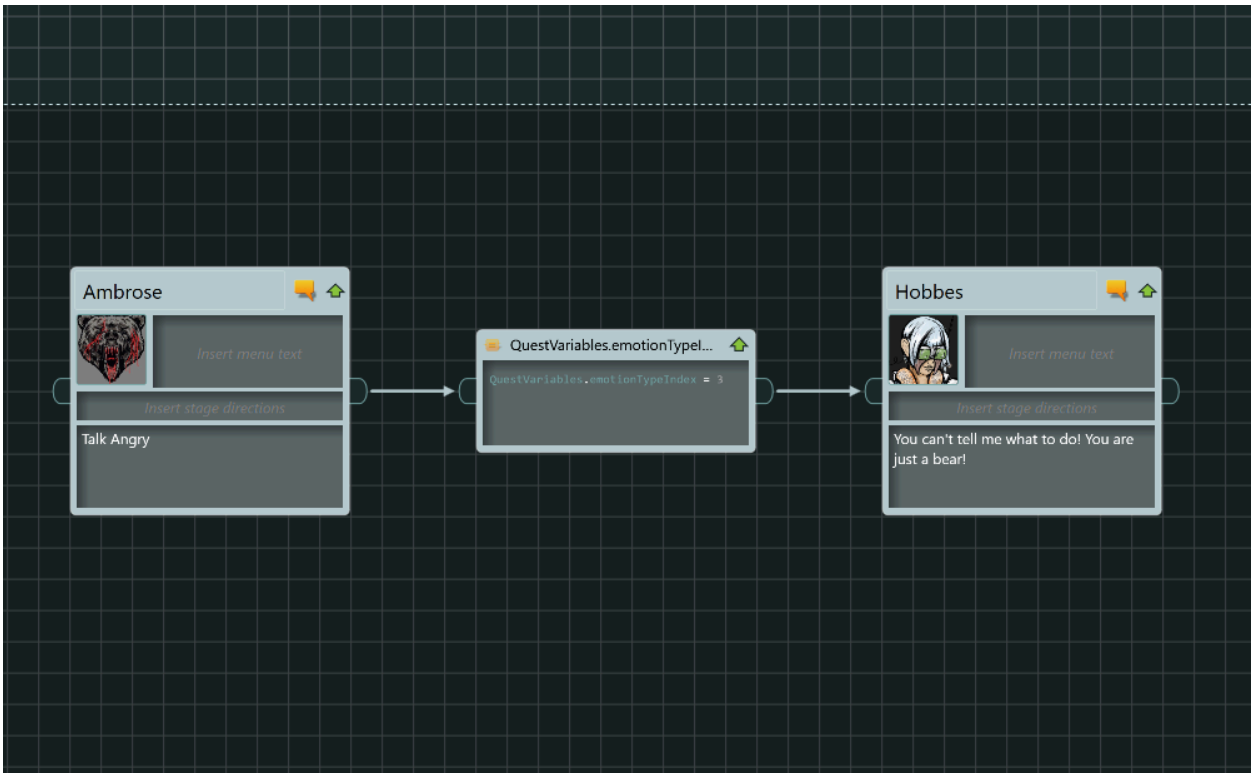
3. When using an item tag, make sure that the player can't return and re-do the section of dialogue where the item tag lives. Otherwise, a player might be able to repeat part of a conversation and end up giving or taking items repeatedly.

Talking Head Emotions

Talking head emotions are triggered by setting the QuestVariables.emotionTypeIndex variable to a number between 0 and 7. Each number corresponds to a specific emotion animation that will then play on the talking head. Here are the emotions and their associated number values.

1/0: Neutral
1: Happy
2: Sad
3: Angry
4: Disgusted
5: Focused
6: Scared
7: Sympathetic

The emotion must be set via an instruction or pin immediately BEFORE the line you want to play with the emotion. For example, if you want a particular NPC line to have the Angry emotion, you need to set the variable to 3 right before that dialogue line. You can put that instruction in the pin leading to that dialogue line, or you can put an instruction node right before that line.



After each dialogue line, the emotionTypeIndex variable is re-set to 0, resulting in a neutral talking head emotion. If you want multiple lines in a row to display the same emotion, you'll need to keep setting the emotionTypeIndex variable to your desired emotion value before each of those lines individually.

Important Rules

1. Never use anything other than size 11 font in Articy fragments (you can use different font types in annotations).
2. Only use integer-based variables.
3. ALWAYS consider the possibility of soft locks. The player must be allowed forward through the quest even if there is nothing powered in their brain.
4. NEVER leave unfinished script active in Articy. Always comment them out until they are 100% functional.

5. You CANNOT use an instruction at the very end of a dialogue thread or conversation. There must ALWAYS be a dialogue fragment after an instruction. This is due to how Unity digests Articy script.

FAQ

Info

Feedback Images