

Schattenlander – Code base structure

Conceptually, Schattenlander adds several different areas to the M&B baseline. Here's an overview of what they are. All line references are to the August 28, 2007 codebase; later modifications will of course throw these off, but they're only included to help you navigate anyway.

1) Dice-rolling code (module_scripts.py, line 2370). Centers on the script "script_basic_roll". Documentation on how to use it is at that line. This is the basic randomizing logic the rest of the mod uses anywhere it needs to test a skill or just about any other random situation. Anyone familiar with dice-pool systems such as Vampire: The Masquerade or The Riddle of Steel should be perfectly at home with this mechanic.

2) "True" stats, aka Aspects (module_constants.py, line 330). This is one of the biggest hidden secrets, and it permeates everything in the mod. Aspects incorporate three related functions:

- All PC heroes, including the player, possess "true" values for their stats, skills, and proficiencies. These values are stored in a series of slots on their Troop, called Aspects. The various aspect_xxx constants show which slots hold the true values of which stats. When we modify a stat/skill value, whether through experience or enhancements, what we actually modify is the associated Aspect. A periodic trigger checks to make sure that the "game values" which are displayed on the character sheet and which are used by the game engine correspond appropriately to the Aspect values. For stats and proficiencies, this is equal to the game score. For skills, the Aspect is out of 100, and the game value is aspect/10. This has several benefits, notably (a) it allows skills to increment in amounts smaller than one skill point, and (b) it functions as a kind of cheat-check, since if you adjust your stats in any way, the periodic trigger will catch that and revert them.

- All NPC enemies possess "true" values for their stats, stored as Aspects, again in the Aspect's numbered slot on their Troop. A script run at game startup goes through and records the values you gave them (via module_troops.py) in these slots. This happens for a similar, but distinct, reason to the above. The NPC's original stats are a template which can be applied not just to themselves, but to *any* troop. So if I want half a dozen antagonists with the stats of a Dirty Street Rat, I can read the Dirty Street Rat's aspect slots, and assign those stats and gear to half a dozen troops whose original stats were different... and then retrieve all of their original stats again when I'm done, via the slots which were set at game start. NPC enemies' Aspect values should **never** be altered during play, unless for some reason you want all (e.g.) Dirty Street Rats from here on forward to have their stats modified.

- Enhancements and Effects (see later for the distinction) use the Aspect number to describe what's being modified. For instance, an enhancement of +2 Strength would be stored as two values – we would store, separately, "+2" and "aspect_strength", and together they define this modification. More on this under those sections.

3) Saints (module_saints.py, and a few instances elsewhere in various files). Each Saint has a wide variety of characteristics, as seen at the beginning of module_saints. Each Saint is a Faction, in game terms; see line 941 of module_saints for the list. (No party or troop ever belongs to this Faction, it's a data storage object only.) This applies a hard limit of 125 Saints in the game – 128 factions possible, three are marked "do not tamper" in module_factions.py.

The characteristics of a saint can be covered in three basic groupings:

- Biographical and invocation data. Does the party have knowledge of this saint yet? What pic should we use (once we put the saints' pics into the mod), what descriptive text string? What's the virtue requirement, invocation difficulty, and base DF cost? This is in slots 0-13 at this time.
- Stat and skill boosts. Each saint has up to ten separate Enhancements it can grant, such as "large boost to Strength" and so forth. This is covered in the Darklands manual and replicated in the strings & start-of-mod slot assignments located in module_saints_init.py. See "Enhancements and Effects" below for more details on how this works.
- Contexts. This is a list of slots covering all of the circumstances (encounter menus) where invocation of some saint and/or use of some potion might affect the flow of menus. This is distinct from their stat effects. For instance, St. Jehosephat might turn evil clergy into clowns, quite changing that encounter to an entirely different flow of menus. Okay, he doesn't exist, but he is honoured in our hearts as the first saint ever implemented in the mod.

Invocation of a saint can happen through dialogue with a party member (see "Dialogue-based Party Settings" below); it can happen while interacting with encounter menus (see "Random Encounters" below); or it can happen on the battlefield (see "The Orders Interface" below). In each case, however, it triggers a sequence of menus and scripts which can be hard to follow from the code. In short:

- If triggering from a random encounter menu, we ask who should be praying. When triggering through dialogue or onfield, this is already known through the appropriate interface (who you talked to or who you issued the order to).
- Saint selection menu displays only contextually appropriate, known saints, a few at a time. This menu can recurse to call itself if you ask for the next page.
- In the most common contexts, "context_buff" (stat boost during battle) or "context_peacetime_buff" (stat boost triggered through dialogue with party member), a similar selection menu lets you pick who to pray for.
- The workhorse is the "mnu_saint_description". This is where most of the work gets done. We go ahead and assess your basic odds of success on the prayer, for each form of praying you might do – this allows us to give you odds estimates as part of the menu choices.
- Once you pick a method, we redo the assessment of odds (because it's a subtly different calculation process than the predictions we did, and computing cycles are cheap), and make a roll. The actual math behind the roll is as follows:

A) If your DF is below zero, you will lose as many dice as your DF is in the hole.

- B) The praying char's Virtue score (out of 100) is compared to the Saint's virtue requirement (again out of 100). You can potentially gain a max of three dice for having way more Virtue than needed, but more importantly if you don't meet the requirement the Saint's difficulty score goes up rapidly (one per 5pts or portion thereof that you're not worthy).
- C) If today happens to be the Saint's day, divide the difficulty score by three. The Risk, which we'll discuss in a sec, is also divided by three.
- D) Your dice pool for each mode of prayer is based on a flat amount plus a factor times your Religion skill. For instance, "murmur" prayers have a flat 2 dice, plus a measly 3% of your Religion skill – largely insensitive to learning. "Ceremony" on the other hand has a flat -2 dice, but gains a huge 25% of your Religion skill (out of 100). Given that there's a fairly high minimum Religion score to use Ceremony at all, it's pretty much always better, usually much better; the -2 just tweaks the numbers a bit.
- E) Pay for your prayers. The prayer costs you (a) one point of DF per point of Cost on the saint – this is a flat minimum, IIRC in the range of 0-4 or so – plus (b) the results of a dice roll based on the form of prayer you chose. A Ceremony rolls 14 dice, for instance, averaging about 6 more points of DF spent, while a Murmur rolls only 2 dice, averaging about 0.8 points of DF spent. DF bottoms out at an absolute value of -10; most people have a max DF of +5, though a few character types will have a bit higher (up to 7 for a Hermit, if I recall correctly).
- F) Maybe you get results. Roll your dice pool versus the saint's difficulty pool; get at least one net success and it works. Otherwise, your DF is spent and you get nothing. What "It works" and "nothing" turn into is situational – for instance, during battle this is run as a pure script; "nothing" just means end the script. Whereas if you invoked it in a city, we need to launch you back to the city location you were, which is a bit more code. Similarly, if you're in battle and it worked, it generates Effects (in-battle modifiers), whereas if you're praying out of battle, it generates Enhancements (lasting out-of-battle modifiers) – see those sections for details.

4) Use of Potions (module_alchemy.py). Potions, as usable items, are hugely parallel to saints. "Do you know one" is replaced by "do you have one", there's no random roll involved (usually), and so forth, but the basic idea is the same. They decay at a different rate, probabilistic rather than dawn-or-dusk, but we'll discuss that under Enhancements/Effects.

Potion use is more complex chiefly because each potion is two things at once, with the **same ID number** for each. It's a Party Template (though no Party is ever spawned from it), for the slots to store information about it like which stats it affects and by how much, etc., but it's also an Item, for the actual bottle in game. Careful ordering of their placement in module_items.py and module_party_templates.py is required to achieve this effect; be very careful in tinkering with any of this.

For the potions with simple stat effects like Quickmove, the parallel pretty much ends there. Offensive potions such as the Arabian Fire potion require a whole 'nother complex of code which I'll document separately in a bit.

5) Random Encounters (module_random_encounters.py). The heart of this system can be found in module_triggers.py (line 38), and also note that the scripts part of the random encounters is way down at the end of module_random_encounters (that file is mostly menus, only a couple scripts right at the end).

The script "assess_locale" is currently mostly undeveloped, but is intended to assess whether you're on a road or in the forest, whether it's night or day, so that the random encounter can be tailored.

All random encounters actually take place with the same party, "p_random_encounter_party", which is both far away (encounter is manually triggered, not triggered by running into it) and invisible, and which has its members reassigned and cleared as necessary.

The random encounter menus are translations, one by one, of the contents of the original Darklands stuff. The directory /dev-notes/MsgFiles Triage contains all of those originals, parsed and separated out. Blankshield has been working on these somewhat, pseudocoding them based on what I showed him of the menu system, and may be willing to continue doing so if contacted.

6) City travel, camping, and similar. The key to all of this is to gain a good understanding of the innocuous-looking stuff near the end of module_simple_triggers.py. These triggers (auto_enter_town and auto_menu) are extremely powerful. What's happening when you move around in town, or when you exit back out of menus into party-member dialogue in the wild, or in several other circumstances, is that you're actually ending one encounter, with some special variables set which cause the game to hurl you back into a new encounter right away.

So, for instance, moving from the market to the town square causes code within module_city_contents.py to calculate your total in-city travel time, if it comes to some multiple of 60 minutes to set you up to rest for N hours once the current encounter is finished, set "\$auto_enter_town" to the current town (this will cause the simple trigger to fire as soon as you're done resting), set "\$return_menu" to the menu you're going to end up at when you get where you're going, and then terminate your encounter with the town. You're thrown back to the wild, forced to sleep if appropriate, and then automatically re-encounter the same town, and are directed to that menu (aka location in town). Yes, this happens *every time* you travel between town locations. Usually it's invisible, but this is a source of an occasional current bug where the auto_enter_town doesn't fire for some reason.

It gets even more complicated occasionally – say you're praying when at the town square. Then we need to know that (a) for the prayer, jump to the praying menus, but then (b)

when you're done with that and dumped back to the wild again, jump to the town square. This is what "\$return_menu_stacked" does for us. Mostly you won't need this and can leave the _stacked variable as zero.

7) Character creation (module_chargen.py). This bit is pretty insane in its own right, and hopefully won't need to be changed much. Professions (including family backgrounds like "Commoner" and later professions like "Monk/Nun") are Troops, for data storage purposes only. They have a lot, but not all, of their data stored in troop slots during an init step. For example, a given profession gives a flat +N to Crossbows (stored in aspect_crossbow), and the option to add up to another +M points from your discretionary amount (stored in maxpts_crossbow, defined in this file and used here only).

However, the professions also have a bunch of dependencies, which aren't in the slots, they're buried in the checks for dialogue choices when choosing a profession. For example, Monk/Nun requires prior experience as a Novice Monk/Nun, Noble Heir, or Courtier, plus Per skill at 20+ and Int/Cha of 6+ each – see line 927 of module_chargen.py.

Other than that, the best thing I can say is that the chargen dialogues are well-contained within this file, and you won't need to go outside of it much. About the only things you might need are the initiating dialogues when you first meet a given Companion, which are unique per Companion – for instance, Franz's dialogue doesn't let you pick his family background, it's chosen for you, as is his first career as a Student; his dialogue sets the variables and conversation-jumps such that you enter the chargen process late with those already set and executed for you, as if you'd gone through the dialogues to choose them yourself. Similarly, Jurgen/Anyu always has the same family background you do (duh). Other Companions may offer an option on background but not on initial career, etc.

There are also some links to code in module_advancement.py, which you might need to examine if you're working on this part in detail.

8) Advancement/Experience (module_advancement.py). Most of this chunk is pretty clear by comparison. It's the engine which drives the conversion between Aspects and game stats. Training in general rolls a skill contest between your skill and your trainer's effective skill rating – if you win, your skill does not improve, but if the trainer wins, it goes up by a number of points based on how thoroughly he won. Sometimes the "trainer" can be circumstances – for instance, successfully convincing bandits to let you past without a fight should give you an immediate trainer roll with a fixed skill value. This has not been fully incorporated into the random encounters, and in fact I've been waffling on whether this phenomenon should just get inserted into the dice-rolling code rather than being manually inserted when relevant.

A couple of routines in here deserve mention for the obscure roles they play... one is the routine which fixes the fact that the skill "Alchemy" is actually M&B's First Aid skill, which has combat relevance. So during combat we overwrite everyone's "Alchemy" skill

with a new value (the successes on party's best Healing skill roll – thus generally low even with a high Healing), and then after combat we replace the Alchemy scores with what they should be (reading from the Aspects).

The other is the routine "Postcombat Advancement" which actually does many things related to cleaning up after combat. For instance, if your proficiency score with a blunt weapon increased, you have a chance of improving your Power Strike skill, but you also have a chance of weapon quality decay on any blunt weapons equipped. This item decay needs tuning yet – it's too fast right now, for both weapons and also for armour. Armour decays based on how much damage you suffered, amplified up by how much armour you're wearing so that in theory the same amount of damage-before-armour should give roughly the same decay, even though the better armour reduced the only thing we can measure – the damage taken after armour.

9) "Preptime" and "Permissions" (module_scripts.py). This is legacy code from before I was able to get in-combat orders working. It does NOTHING and should never be called. However, I believe it is still referenced in certain semi-legacy routines within the alchemy section, and thus the Permissions code and the Preptime variables are still kicking around because to throw them out would produce errors at compile-time.

It might even still be being called to determine whether someone has a potion in their possession, or the relevant ingredients, etc., with the rest of the Permissions stuff never being triggered. Double-check that before removing.

In short, you can ignore all of this code unless you're trying to clean it up. These days, if we want an ambush, we'll set up the ambush using nearby spawn points, and if you have time to get a prayer off (using the in-combat orders), then you do, and if you don't, you don't. No preptime variables required.

10) Combat Initiation (module_scripts.py, module_game_menus.py). This is Fisheye's biggest contribution to the mod. It centers on a menu, named "combat_normal", which starts on line 287 of module_game_menus. The key to understanding this menu is the fact that combat always starts from a menu, and then after combat returns to *the same menu* automatically. So this menu is detecting how many times it's been called, and behaving very differently depending on the answer. Read it with that in mind and it should be relatively straightforward. The order of events within this code can be pretty sensitive – recommend not mucking with this unless necessary.

Looting is also Fisheye's code at heart, but that's relatively straightforward – see the scripts starting at line 2317 of module_scripts.py. Minor modifications of the original Native versions of these routines.

One of the subtle things you need to do when setting up a fight in Schattenlander is based on the fact that the potion-throwing code only works if every *agent* is a unique instance of a distinct *troop*. So, for instance, in Schattenlander you should never see two copies of

the Dirty Street Rat troop on a battlefield. Ever. When we want two "Dirty street rats" for a fight, we use two scripts to set this up.

The script "namerange" is called first, and it takes two parameters – the start and end of the range of troops whose names we wish to appear above the heads of the troop type we're about to spawn. So, for instance, if I want a half-dozen "Dirty street rats" or equivalent, I need to run script_namerange with two troops which, including themselves, cover a range of six or more troops in the module_troops file. Thus, for instance, I might use (call_script, "script_namerange", "trp_murderous_thug", "trp_misguided_youth"). Dirty street rat happens to be somewhere in the middle of this set (see module_troops), but that's actually a coincidence – the range of names does not need to overlap with the troop whose stats I'm planning to use at all. Basically, since I can't change their names, I'm just stealing the souls (including names and faces) of a random collection of the troops within the range I specify using this routine. If I was looking for a half-dozen street rats, calling (call_script, "script_namerange", "trp_dirty_street_rat", "trp_reeking_scum") would produce an error shortly... because I'm trying to pick a half-dozen individuals, but I've specified a range that only includes three people.

After using namerange to specify a range, the script "add_enemies" does the work. The parameters it takes are a troop (the "key enemy") and a quantity. The key enemy provides the *stats and gear* for the resulting bunch of baddasses; they'll all have their current stats overridden by the key enemy's stats, and their gear cleared to make room for the key enemy's gear. Basically, they become statistical clones of him.

In the special case where the quantity is one, you don't need to run script_namerange first. You'll get one instance of the key enemy, by his own name and face, with his own gear. This saves time when throwing in leaders and uniques and so forth. In all other cases you should run script_namerange *immediately before* script_add_enemies – if for no other reason than that this way you can't forget.

Generally speaking (cf. the fight scenes in module_random_encounters.py), what you want to do is clear the encountered troop, then run all your calls to namerange and add_enemies in succession. The result is a well-formed, stocked, enemy party, ready to be fought. Set the variables expected by combat_normal, for instance which menus to jump to in case of victory, defeat, or retreating, and then jump to combat_normal for the fight itself.

11) Wildsites (module_wildsites.py). These are fairly straightforward and are a design decision we made during the planning stage. Rather than do as Darklands did, with a fixed map of villages and so forth, we wanted the exploration aspect to have some randomness to it, which is where the concept of wildsites came from. Campgrounds, castles, villages, and so forth are collectively "wildsites" – map objects which are subject to discovery during play. All of their code is in here, except the wildsite templates which are still in module_party_templates.py because attempting to move them into module_wildsites created a file dependency loop. It's pretty straightforward. Once

villages and castles and so forth become more developed, their menus would also get gathered in this file.

12) Enhancements, Item Enhancements, and Effects (all mostly in `module_scripts.py`). This is one of the biggest pieces of code here, and these three terms have strictly defined meanings within the code. Before I get into how the code works, here are the definitions:

An "Enhancement" is a long-lasting effect on a party member's stats, skills, proficiencies, or any other Aspect. (Other Aspects include nonquoted 'skills' such as Virtue, and other properties such as regeneration or fire resistance.) It may be positive (usually a bonus) or negative, depending on the source. Potions, saints, and encounter aftereffects are typical sources of Enhancements. They can never be applied to enemies, only your own companions.

An "Item Enhancement" is similar but uses a different set of mechanisms for generating them, decaying them, and tracking them. These are always long-term as well, as far as I recall, because you can't alter the items on an agent during combat, so they have to be done ahead of time – even weapon/armour blessings, which in theory you can invoke during a fight, won't actually kick in until after it's over. Enemies can be given enhanced items and they'll use them on you during the fight, though I don't recall if I ever got looting the enhanced items to work properly. (We need to create the relevant bookkeeping entries so that the item enhancement can decay and be tracked. It's not enough to just hand them the item.)

An "Effect" is similar to an Enhancement but is generated during combat, and only lasts for the duration of combat. Stun, itching, acid damage, and the like are typical examples of Effects. These are the only stat effects which can be applied to enemies directly. Unlike Enhancements, these don't modify Aspects; they change the actual game stats directly, and only those. Otherwise we'd be modifying the "base stats" for the affected troop type – no good.

12A) Enhancements. When you pray to a saint, one of his benefits might be listed as a "large increase in Strength". That's an Enhancement. If he also grants a "small bonus to Charisma" that's a separate Enhancement; it may happen to expire at the same time, be on the same target, etc., but they're handled separately from the moment of creation on. Here's what happens when you pray successfully to that saint (potions are very similar):

a) The script "apply_saint_portfolio" goes through the saint's listing and extracts how many effects the saint grants. (The term 'effect' is not used correctly in these variables – this predates Effects as a category. In general it's still clear when you look at the code.) Each is listed as an Aspect and a Magnitude.

b) The master script "enhance_aspect" kicks in for each benefit. This chiefly does two things. One, it actually alters the Aspect of the character. This includes checking for caps on skills and so forth, and applying only as much of the benefit as they can take.

Two, it records the Enhancement on a series of dedicated array-troops. For each enhancement we record four things: the Recipient, the Aspect affected, the Magnitude (the actual bonus we granted), and the Type.

Type can take only a few values, as defined in `module_constants.py` line ~481. It can be a "blessing" in which case the enhancement will wear off completely at the next Dawn or Dusk. It can be one of three qualities of potion, low/medium/high (or the not-accessible-to-PCs category of "potionlike extended"), in which case it wears off probabilistically... that is, each point of the enhancement has a separate X% chance per hour to wear off. So on average a +8 bonus would drop to +4 after some N hours, then to +2 after another N hours, then to +1 after another N hours, and eventually to zero. X is lower, and thus N is longer, for the higher-quality potions. These have been fairly carefully tuned and I'm pretty happy with them right now.

The last category of Type is "secondary enhancement". This deserves special mention, because a lot of the trickiest code in `enhance_aspect` and `decay_aspect` keys to this functionality. In Darklands, a Strength bonus was a pretty big deal. In M&B, the effect of Strength is partitioned out amongst various skills (e.g. Power Strike, etc), and only a relatively small amount of it is actually inherent in the Strength stat. Thus, I decided that a Strength bonus (and the same goes for the other three attributes) should grant a corollary enhancement to all of the skills which should be improved by improving the skill. So when a Strength enhancement is created, it also creates a list of secondary enhancements, each of which is tracked as its own enhancement (a bonus to Power Strike is still distinct from a bonus to Strength stat, and we need to be able to remove both and cap them separately). This works well and shouldn't need tinkering at this point.

c) When the `enhance_aspect` script is done, it uses the normal "your Aspects are the real values – you should update your stats to match" code to propagate the changes over to the character's actual game stats.

Enhancement decay is then handled (on an hourly trigger) by going through the list of enhancements on record, and reducing their magnitude (and adjusting character Aspects to match) or removing them altogether, as appropriate. Once an enhancement is gone entirely, we do some cleanup steps because we don't have infinite slots available for tracking purposes and so we need to free up the tracking slot, and because sometimes the rounding doesn't properly get rid of "orphaned secondary enhancements" without explicit cleanup.

12B) Item Enhancements. This code is one of the last pieces written and one of the least debugged, although I believe it to be working fairly well at this time.

Item enhancements depend crucially on some code pieces and on a careful setup of the `module_items.py`, `header_item_modifiers.py`, and `/languages/en/item_modifiers.csv` files. The governing script here is "classify_item", which uses this ordering of the items file to extract a great deal of information about each object when asked. Weapons, for instance, are divided into types (blunt, edged, missile, etc), and within that into

size/mass/difficulty-of-enhancing-them classes (light, medium, heavy, etc), as well as listing duplicate items where needed for the enhancement process.

Item enhancements work in one of two ways, depending on the enhancement source. Either they swap out the item for a modified item with better stats, but the same imod, or they overwrite the imod with a new one but keep the item the same.

- Blessings are a straight bonus – the blessed item is a separate item with slightly better stats than the original. They wear off cleanly at dawn or dusk like all blessings. Legacy code alert: originally "Blessed" was to be an imod. There's commented-out code handling this all over the place in these scripts. It didn't work out and was scrapped. So the commented-out bits referring to blessings may not actually be meaningful anymore; don't trust them.
- The Trueflight potion grants sizable boosts to missile weapons, equal or better than blessings, but has a chance (based on the potion user's Artifice skill) to damage the item as the potion is being applied; these are listed as "Treated ____" in three strengths. It decays by reducing the effect quality one step, until gone.
- The Deadly Blade potion (listed as "Caustic ____") swaps out the item for one with a flat +5 damage, and *never* wears off. What it does do, however, is gradually (speed depends on potion qual – we're talking periods of many hours) eat away at the item's imod, corroding it to a poisoned-but-rusty blade and ultimately to useless junk. Not yet implemented was a plan to allow the Essence of Grace potion to function as a "cleanser" to wipe off the acid/poison and get you back the original blade type.
- The Greatpower potion (listed as "Fuming ____" in three strengths) has a substantial damage boost but if you have one equipped during combat there's a periodic check for stun or even damage to the user. It wears off by reducing in strength to the next lowest-qual version, until it's gone.
- The Strongedge potion and Hardarmor potion both grant the imod "Enhanced" to the target weapon/armor, and store the original imod under the item enhancement's tracking info. When they wear off, there's a chance that the imod that "emerges" will be inferior to the original by a step or two, as though it had been in an additional hard fight. Note that you could keep using these potions to "hide" the quality losses until the item actually shatters and is gone.

The decay code for item enhancements is found in `module_scripts.py` next to the other enhancement codes. But if you look, you'll realize that there's nothing there to actually create those item enhancements. That's because blessings are tucked into `module_saints.py` and the alchemical enhancement process (including the "Enhancing vat" encounter) is in `module_alchemy.py`.

As with Enhancements, the Item Enhancements have a set of records of their existence, again as array-troops. The original itemID, original imod, new itemID, new imod, type of enhancement (which potion, etc), and quality. The "original itemID" array is also the actual troop used for the Enchanting Vat, since their actual inventories aren't relevant to their use as slot-arrays. These arrays are quite sensitive because the process of figuring out *which* item should decay is quite a bitch to do... if you swap your Rusty Caustic

Broadsword to Anya for the Fine Caustic Broadsword she was holding, it's a hell of a job (right now) to detect that. But they're distinct as far as the records are concerned. So we end up searching based on the known parameters, which is why we carefully record both the original and the modified versions for each item enhancement. (In theory this infrastructure would allow 'stacking' enhancements where feasible – mixing Strongedge/Hardarmor with blessed or Deadly Bladed items. There may even be some legacy code in there from my attempts to do this. It's disabled at present because that made the detect-which-item-to-affect more than twice as hard to do, and I see no real need to allow this functionality given how hairy the implementation will be.)

There's also a set of code surrounding the script "remove_untradables" which was designed to not let you sell these modified items to merchants. It's broken and does not work at this time. Likely won't get fixed until the next version comes out, at which point it may be much, much easier to handle this issue (pray for nondestructive imod testing).

12C) Effects in combat. Virtually everything described under the basic Enhancements applies here, except that these ones are applied directly to stats. For an example of Effect creation, see module_throwing_code.py, line 1465 (the Eyeburn potion effects). Decaying onfield effects uses the same basis as decaying enhancements, except that the timescale is much shorter, so that effects (if potion-type) wear off in tens of seconds instead of hours. Effects use their own set of tracking array-troops, though the logic here is precisely the same.

There are some areas needing attention under Effects. One is that it's not certain that the postcombat cleanup of Effects on your party members is foolproof. Cleanup of Effects on enemy troops isn't necessary, since add_enemies will reset their stats next time you fight anyway. But I'm not 100% sure that the PCs are properly getting "scrubbed".

The other is the interaction between Enhancements generated on the field (such as quaffing a Quickmove potion) and Effects which may have been in effect prior. Since Enhancements end by resetting your game-stats based on your Aspects, this will override any Effects which may have been in place prior. This is a pretty minor issue but you should be aware that it does exist.

13) The Orders Code (module_orders.py). This one is, IMO, in some ways a bigger innovation than the potion-throwing code. This section permits the player to use just the two hotkeys he's given (I and Tab) to select either a preset order or even to construct one onfield, and have the order's recipient (which may be himself) execute the specified action, such as to use a potion or call on a saint. It also allows you to monitor for prearranged conditions such as "you're low on HP" and execute an action automatically when that condition is detected, although this part is sparser and less well-tested.

Fundamental to this code is the idea of "interface modes". This is only meaningful during combat. The "I" key cycles between relevant modes, and the Tab key does stuff once in that mode.

- The default starting mode is "normal" where Tab has its usual functionality, in theory.
- If you have any throwable potions equipped, hitting I once puts you into Aiming Mode; more on that under the throwing code section.
- If you don't have any thrown potions equipped, or you hit I again, then if you have any preset orders arranged, you enter Preset Orders mode. In this mode, Tab cycles between your various prearranged signals, and you pick one by simply leaving it "up" without Tabbing out of it for a short time (3s IIRC).
- If you don't have any prearranged orders which can be signalled, or if you hit I again from Preset Orders mode, you enter Construct Orders mode. In this mode, Tab cycles as before, but only a small part of the command cycles each time. So, for instance, the first element is the order's target – who are you telling to act? Cycle between team members until you get the answer you want, then wait. It'll show up again in a new colour to confirm the selection. Now Tab will change the next part of the order – what action do you want them to do? Cycle with Tab, select by waiting, and continue.
- Hitting I one more time brings you back to the normal mode.

The meat of the orders code, as you can see looking at the file, is in the triggers it uses. Including this block of triggers in your mod is done using Python trickery – see `module_mission_templates.py` for how I've done it here. The only scripts the orders code uses are some string-manipulation strings which we use to turn the game data (order target, action to take, recipient of action, etc) into human-readable form. Complex but working fine, as far as I can tell.

The triggers are doing a lot of checks to the current state of the system, massively overloading the `ti_tab_pressed` trigger condition. The list of preset orders is in a set of associated array-troops (sound familiar?), where a given ID number gets you one slot from each troop, carrying one piece of info about the order. Retrieving those characteristics and displaying them is all that Tab has to do in the Preset Orders mode. In Construct Orders mode we pay even more attention to the current system-state and generate all sorts of partial strings for the incomplete orders.

All of this is moderately well-tested in isolation and in simple battlefield conditions... but should still be watched as the mod develops further, as it's not guaranteed perfectly stable and it certainly needs expanding as we add new types of orders to the lists. The plan was to extrapolate this so as to also provide a mechanism to do some interesting tactical things such as having your companions find sniper positions if available, etc.; none of that exists yet.

Also note that because NPC throwing of potions keys to their activating a weapon, we cannot order that to occur on command. So "target enemy X with your potion NOW", although intuitive, would be at best difficult and at worst impossible, at least under the 0.8xx module system.

14) The Throwing Code (module_throwing_code.py). This is the big one that I know many of you would like to use. I'll try and document what's going on here as best I can.

If you want to use this code at all, you *must* ensure that every agent on the battlefield is a unique troop. No two Dirty Street Rats allowed, even if they don't hold potions themselves. See the "namerange" and "add_enemies" scripts described in this file, under point 10 (Combat) above. Otherwise it simply will not work.

You will also need to either replicate the perfect symmetry used here, where each throwable Item has a matching data-holding entity (Party Template in this case) with the same ID number as the item. Or, understand what's going on with these bits well enough to replace with your own means of figuring out what the characteristics of a given projectile should be. If you only have one type of throwable item, this may be easy. If items get slots in the next version, this will become easy. Take your pick.

The throwing code is in several bits, which it's worth describing separately. There's the piece which sets it all up, much of which actually executes before combat, and is inspired by Winter's limited-ammo script. There's the auto-targeter with its little crossed-swords marker. There's the piece which actually launches the projectiles, which is where the setup phase comes into its own. There's the piece which actually makes 'em fly. And there's the piece which makes them explode.

One at a time, then:

Setup –

The module system (0.8xx) cannot detect the location of an item directly. Nor can it detect which instance of an item may have caused a trigger to fire. So in order to pin down the start point of the throwing arc, we have to isolate things so that there are never two identical throwable potions on the field at all, and so that we can map uniquely from (object that caused the trigger) to (troop who owns that object) to (agent who incarnates that troop on this battlefield).

Thus, during the setup phase, we take away all of the Hugh's Arabian Fire or Solomon's Eyeburn potions (etc.) possessed by all NPCs, friendly or foe. We replace them with a "backup potion" which is still a throwable weapon, and we store the true identity of their potions in an array associated with the backup potion. I've included ten discrete backup potions, which means that the field can contain a max of ten NPCs with throwable potions (or equivalent, such as demons' fireballs) at one time, plus the player. This should more than cover any known Darklands fight.

The backup potions have ammo = 2, and a routine will check them after the throw to see if the NPC should, in truth, have any more potions he can throw – and, if so, reload them up to 2 again. Can't be ammo = 1 because then as soon as he throws once, there's "nothing to reload" according to the hardcoded rules. This does mean that every NPC who can throw potions but only has one potion will have one more "attempt to throw" than he should, as he fires off that last one – but it won't launch anything. Also, the

backup potions had to have "real" thrown weapon characteristics so that they'd be (invisibly and hopefully harmlessly) thrown the proper distance – otherwise the AI won't throw until way too close. Whereas the PC's thrown potion items have basically zero range, they fall unobtrusively to his feet.

When we're done combat, by the way, the reverse of the setup script will return the actual PC-usable potions to everyone's inventories (minus any that they're recorded as having thrown!), and remove the backup potion items.

Auto-Targeter –

This function is partly dependent on the interface mode (see Orders Code, above), and in fact was the original inspiration for the approach that evolved into orders here. In default mode (or, indeed, any mode besides Aiming Mode), the autotargeter is turned off. If you throw a potion, it'll target your max throwing range, dead ahead, and launch at that point. If Aiming Mode is on, however, then on a rapid trigger we look up the locations of every enemy on the field, checking to see if they lie within a narrow "pie slice" centered on dead-ahead of the player, whose angular width depends on your Thrown Weapon proficiency – the less good you are, the closer to dead-on you have to be for the autotargeter to kick in. If we find more than one, we use the closest one. This works well because of the way the M&B interface works... as soon as we pull back an arm to throw, our agent's facing-arrow becomes hard-linked to the mouse movement – the player-agent turns to face your aim point, so that pie-slice is centered on the crosshairs. When this is not the case, your mouse doesn't aim the autotargeter, it stays linked to the direction you're running (or whatever).

If the autotargeter found a subject, then it marks them with the crossed-swords scene prop, and makes said scene prop spin over time.

Actually it's even more complicated than that, because another trigger running frequently keeps track of what direction everyone on the field is moving and how fast. Based not only on how fast they're moving but also how consistent that vector has been, we lead the targets based on their expected movement during the time-of-flight of a potion. (This value is standardized, they always take the same length of time, partly because of things like this.) If their consistency-of-vector is low, we don't put much faith in it, and take it into account only slightly. If consistency is high, we put more faith in it. Then what we look for in that pie-slice isn't actually the current location of everyone on the field – it's their predicted location on this basis. [The same trigger which records those predictive vectors, by the way, also logs the distance you and your companions travel, both on foot and ahorse; we use this in the post-combat experience step. If you want your Athletics to go up, spend a lot of time during combat running around and covering ground. This info is basically free since we're tracking movement anyway.]

NPCs don't have a constantly-updated autotargeter since it's the AI's job to line up a target and throw at it. But the autotargeter code (minus the crosshairs prop) does kick in once, at the instant of the throw, allowing us to pick out (hopefully) exactly who we think

they are aiming at with this throw. We then use that spot as the aim point for the potion prop.

Using Tab while in Aiming Mode, incidentally, causes your maximum range to retract suddenly but then sweep forward over time. Basically this allows you to pick a point even if it's not the predicted position of any given enemy, and *also* not at full range for your throwing ability (which includes Power Throw skill). By retracting the maximum range of your throw, we retract your aim point when there's no autotarget found, and also exclude enemies outside of the current sweep range.

Launching the projectiles –

This is done by weapon triggers on the potion or backup-potion items. These triggers specify to us which item fired the trigger (that is, each one is coded to specify its own ID, we still can't detect that automatically). Based on this we call a script which initiates a "projectile" object in our tracking array. (Unlike the other tracking arrays in the mod, this array is just one party template, using blocks of its own slots for each of the different pieces of data such as who threw it, how fast it's flying, etc., for up to (IIRC) ten projectiles at a time. If I were to do it again I'd probably use separate troop-arrays instead, and so we could have up to 256 or something in the air at a time – but the distinction's pretty moot.)

There's also code for NPCs which reads through their list of available potions and selects the one which (a) will do as much damage as possible to the enemy, without hurting any friends at all, assuming it lands right on target, or failing that option (b) will do as little damage to friendlies as possible.

Flight –

The exact characteristics of flight are set at launch time based on the aim point (full range dead ahead, or the position of the autotarget victim if found). This is where the physics comes in. First, we tweak the aim point a little, based on our skill – the potion scatters by some amount from where it had been intended to go, less if we're better at throwing things. We do some vector math to break up the displacement between the revised aim point and the launch point, into thirty even steps (round to nearest cm – this might add some extra scatter but it's guaranteed to be small, < 30cm) along each axis. For X and Y, this is it; we record those vectors and we'll move by that much each step in flight. For Z, we also add a fixed upward vector to this amount.

Then, every 3ms, we advance all projectiles by one step. They move by their recorded velocities in each axis. Their Z velocity is decreased slightly (gravity!). The initial Vz value and the Vz loss each step have been calculated so that the net movement over thirty steps due to these factors is zero; at the end of thirty steps, the Z displacement will be only that Z displacement needed to reach the revised aim point.

To make it tumble, incidentally, we have a second position which accompanies the projectile's position of record. It's that position, plus a rotation which we also increment each step. Think of having two markers. One slides along the path of the throw, moving according to vectors as above. The other is pinned to it, so it doesn't need to track movement, but rotates at a defined rate. The scene prop moves to this second position.

After exactly thirty steps, always, we arrive at the adjusted aim point. On the thirtieth step we make sure that the explosion itself (the particle system prop) has been moved into position directly underground from the impact point – mostly this is in case another explosion of the same kind was using the particle system before. Then we switch modes and trigger the explosion.

Explosion –

Remember that we recorded what kind of potion caused this projectile to exist? Now we go back and reference that value, and that tells us which of several routines to invoke for the explosion. They're all much the same in broad terms, though. They all...

- Move the potion-bottle prop out of sight deep under the earth.
- Move the explosion-prop associated with their type to the spot the potion-bottle was. (Can't turn them on or off right now, so the best we can do is hide and reveal them by moving them around.)
- Trigger a sound effect.
- Run through the current positions of all agents in a double loop. The double loop lets us detect agents *in order from closest to furthest*. This is purely a cosmetic thing – I want to see the severe burns show up first, followed by the mild scalds after, on the message queue.
- For each agent, we can be in one of three relationships with the blast epicenter. We can be (a) outside the Outer Radius, taking no damage (or Effects); (b) inside the Inner Radius, taking full damage (or Effects); or, (c) between the two, taking damage in proportion to where we are in that stretch. For instance, Stone-Tar does 30% of its effects if you're exactly at the Outer Radius, 100% if you're at the Inner Radius or within, and scales between the two if you're between those radii (e.g. 65% if you're exactly halfway). All these radii are tuned for the specific type of blast.
- Based on that, the blast inflicts damage, or inflicts Effects, as appropriate. This may include animations (such as falling down). It may check if this agent is a horse and do different things if so – especially important if animations are involved! It will generally also include `display_message` strings to let us know what happened.

We don't stop there. The potion also has a duration counter. We keep the "projectile" alive during this time – only now, that entry's serving as the entry for the ongoing blast. The shortest one, the Thunderbolt, lasts only one step. The Arabian Fire lasts a slightly more visible and palpable three steps. The Breath of Death sticks around many many steps, and invokes its explosion-damage routine each time – though this time, on each

step, you're not guaranteed to suffer damage even if in range, instead you have a small percentage chance of being affected just now, 2% per 1/100th of a second step – meaning you take damage about twice a second, but you don't spam the queue on exactly the half-second mark with everyone taking damage at once. And so forth.

The final stage is when the duration expires. Then we tidy up the projectile entry, pop the explosion-prop back into subterranean storage (soon, maybe we can actually just turn particles on and off, instead of having to hide neverending fountains underground), and stop counting this projectile with each step.

And that's it!

For anyone wishing to continue work on Schattenlander, rather than porting this to their own, please note that the Sunburst potion when affecting the player is a sucky workaround (what I want to do is blur the actual game view the player sees), and the Stone-Tar potions still need extremely careful tuning if their current mechanism – enforced "walk" animations with a small chance of being enforced per tick – is to work properly. All of the rest are pretty satisfactory at this time.

=====

That's pretty much it. A few small sections are untouched here – the renamed/retasked Skill listing, for instance, and the Occupations code for "what do you do when spending a night somewhere" – but those are generally fairly self-explanatory, I think.

If anyone has any questions, please do feel free to contact me via MBX private messages – or, perhaps better, in the MBX Discussion forum (with a PM to poke me), so that others can learn from the Q&A as well.

Anyone reopening the codebase should please make sure to post to this effect clearly, on MBX's Schattenlander forum. If nothing else, you may get help, and those reading this document will then know that its line references may be off from what they see here. Simple courtesy.

Steal anything you want.

Cheers!

- Hellequin

August 28, 2007