

Design Patterns

Note: OSRS = Old School RuneScape

Factory Method

The factory method pattern is a creational pattern where there is an interface, ideally a superclass. The superclass defers instantiation logic to its subclasses via a method, called a factory method. Runelite uses Java code, and Java supports the factory method pattern very neatly in the form of abstract classes. While you can use regular classes with inheritance, abstract (super)classes make sense because they forcefully delegate the instantiation logic to subclasses because subclasses must override abstract methods in said abstract (super)class.

In Java, when declaring a variable whose type is a superclass, sometimes you don't know what the type of that variable will be. Some ways to declare such a variable are

- Superclass variable = new Subclass();
- Superclass variable = **factoryMethod()**;

The factory method will determine what type of Superclass to return.

The Plugin abstract class is an essential feature to the Runelite client. Plugins separate Runelite from other OSRS clients and allow small, modular, and independent features to be added or removed from the client at will. The Plugin class provides a template for plugins and as such numerous subclasses implement it. Because Plugin is abstract, it cannot be instantiated and one must subclass it in order to instantiate it.

An example of how Plugins follow the factory design method are in the screenshots below. In PluginManager.java, there is a private method called `instantiate(List<Plugin> scannedPlugins, Class<Plugin> clazz)`. It takes a list of Plugins and a Class of plugins. In an earlier assignment, when scouting for Plugin usages, we found two files with a list of Plugins. One list contained the names of all the Plugins' Java files, and the other contained their corresponding *Plugin*.class files. We imagine that somewhere in the code, the system will feed those two list files' contents into this `instantiate()` method. This method returns a Plugin and since Plugin is abstract, the code will have to return a Plugin instance, whose type presumably depends on the method's logic. The first argument is likely the Plugins from the list and the second is likely the type of Plugin to return. This is an example where a factory method will decide what subclass to instantiate.

This pattern is useful for Runelite because it allows adding and removing plugins at will. A single PluginManager manages all the plugins, and as long as people build Plugins according to the abstract class Plugin, Runelite will treat all Plugins equally. There is no need to specially program or additionally configure Runelite to handle a specific Plugin.

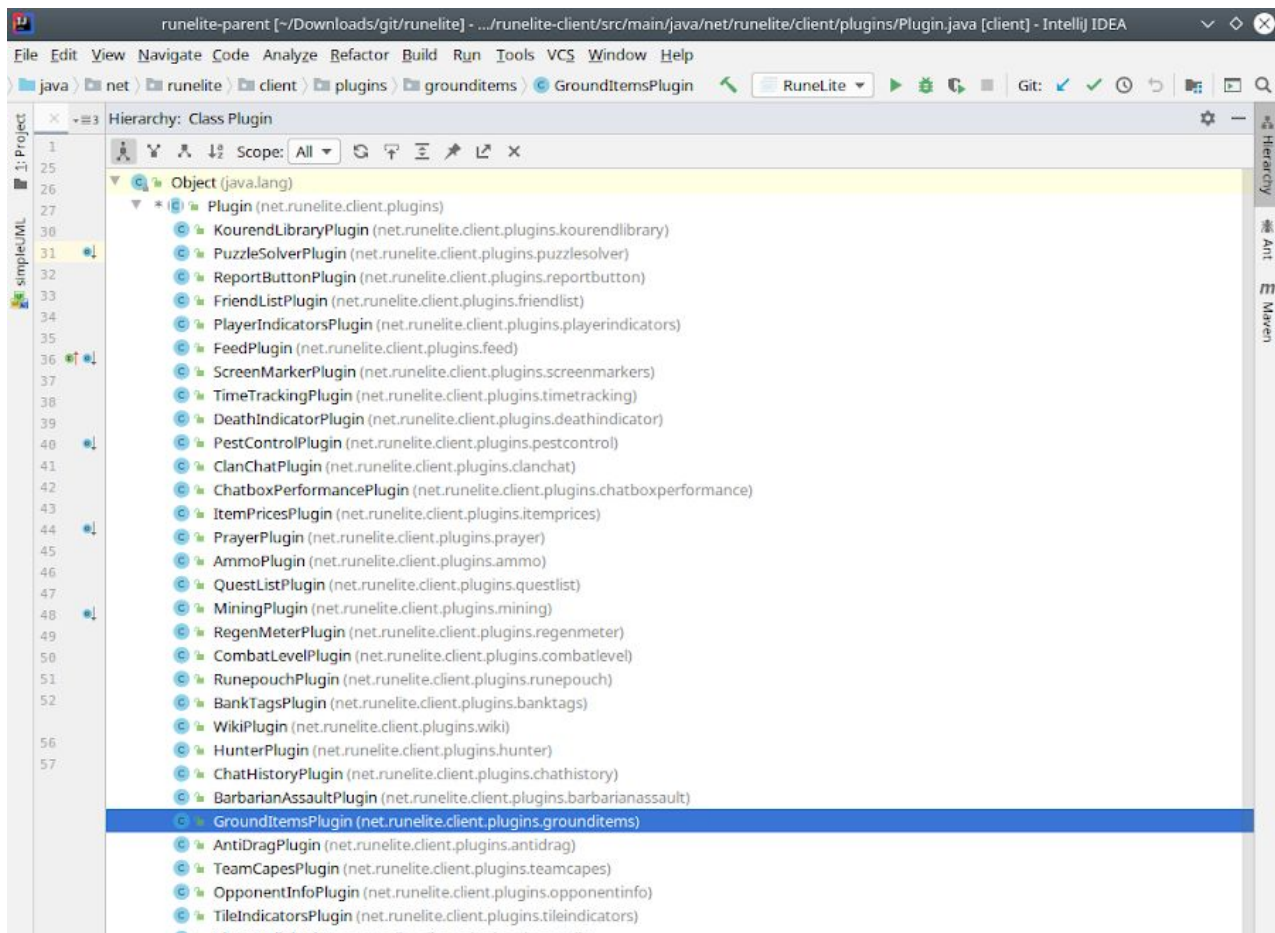


Figure 1: Runelite contains numerous Plugins and the Plugin Manager not only unifies them, but provides a skeleton that delegates instantiation to a subclass and/or to a factory method.

```

private Plugin instantiate(List<Plugin> scannedPlugins, Class<Plugin> clazz) throws PluginInstantiationException
{
    PluginDependency[] pluginDependencies = clazz.getAnnotationsByType(PluginDependency.class);
    List<Plugin> deps = new ArrayList<>();
    for (PluginDependency pluginDependency : pluginDependencies)
    {
        Optional<Plugin> dependency = scannedPlugins.stream().filter(p -> p.getClass() == pluginDependency.value()).findFirst();
        if (!dependency.isPresent())
        {
            throw new PluginInstantiationException("Unmet dependency for " + clazz.getSimpleName() + ": " + pluginDependency
.value().getSimpleName());
        }
        deps.add(dependency.get());
    }

    Plugin plugin;
    try
    {
        plugin = clazz.getDeclaredConstructor().newInstance();
    }
    catch (ThreadDeath e)
    {
        throw e;
    }
    catch (Throwable ex)
    {
        throw new PluginInstantiationException(ex);
    }

    try
    {
        Module pluginModule = (Binder binder) ->
        {
            binder.bind(clazz).toInstance(plugin);
            binder.install(plugin);
            for (Plugin p : deps)
            {
                Module p2 = (Binder binder2) ->
                {
                    binder2.bind((Class<Plugin>) p.getClass()).toInstance(p);
                    binder2.install(p);
                };
                binder.install(p2);
            }
        };
        Injector pluginInjector = RuneLite.getInjector().createChildInjector(pluginModule);
        pluginInjector.injectMembers(plugin);
        plugin.injector = pluginInjector;
    }
    catch (CreationException ex)
    {
        throw new PluginInstantiationException(ex);
    }

    Log.debug("Loaded plugin {}", clazz.getSimpleName());
    return plugin;
}

```

Figure 2: Above is the implementation for the `instantiate()` method, which returns a `Plugin` of an indeterminate type. We deduce that the method's second argument will determine the type of the `Plugin`.

```

for (Class<? extends Plugin> pluginClazz : sortedPlugins)
{
    Plugin plugin;
    try
    {
        plugin = instantiate(this.plugins, (Class<Plugin>) pluginClazz);
        newPlugins.add(plugin);
        this.plugins.add(plugin);
    }
    catch (PluginInstantiationException ex)
    {
        log.warn("Error instantiating plugin!", ex);
    }

    loaded++;
    if (onPluginLoaded != null)
    {
        onPluginLoaded.accept(loaded, sortedPlugins.size());
    }
}

return newPlugins;

```

Figure 3: Here is an example of using the `instantiate()` method. Note the local variable 'plugin', which is declared as type `Plugin`. `Plugin` is abstract, so `instantiate()` must return an instance of a `Plugin`; that instance will be a subclass of `Plugin`.

Facade

The facade design pattern is a structural pattern where a single and simple interface combines many complicated interfaces. Facade is a key element in engineering in general. For example, the average person doesn't understand everything about a car. When he/she drives the car, he knows how to accelerate, brake, change gears, and steer. A car provides a basic interface that masks a whole plethora of interfaces underneath. Some more complicated interfaces are

- The engine
- The fuel line
- Gears
- Axles and tyres

These interfaces don't concern the driver; the driver only needs to know how to drive the car.

Runelite has classes all over the place. To reuse code and make life easier, the developers defined a large number of **Manager* classes which manage their respective interfaces. These Managers consolidate a lot of the logic into single classes. The aforementioned *PluginManager* is one such manager and it manages Plugins, which are small interfaces that make their way into the client at runtime. In the below screenshot are some of the numerous Managers. They don't all behave uniformly; for example, the *PluginManager* has a data structure that holds the Plugins, while the *GameEventManager* is an event bus instead. The core idea is that the developers are wise and centralize the code logic, taking full advantage of Java's powerful object-oriented features.

```
(base) debba@dieWeisseMaschine:~/Downloads/git/runelite$ find . -iname '*manager*.java' -exec basename {} \; | sort
AreaManager.java
ChatboxPanelManager.java
ChatCommandManager.java
ChatMessageManager.java
ChatMessageManagerTest.java
ClanManager.java
ClanMemberManager.java
ClientSessionManager.java
ClockManager.java
ColorPickerManager.java
CommandManager.java
ConfigManager.java
ConfigManagerTest.java
Djb2Manager.java
DrawManager.java
ExternalPluginManager.java
FontManager.java
FontManagerTest.java
GameEventManager.java
HiscoreManager.java
InfoBoxManager.java
InterfaceManager.java
InterfaceManagerTest.java
InventoryManager.java
ItemManager.java
ItemManager.java
ItemManagerTest.java
KeyManager.java
LootManager.java
```

Figure 4: Above are some of the numerous **Manager* classes that Runelite uses.

Observer

The observer pattern is similar to a publish-subscribe style in that there are objects that want to observe or subscribe to listen to changes on a certain event, and there are other objects that notify all listeners of the change. This pattern allows the notifier to send the events to all listeners without needing to know or be associated with those listeners. In other words, these objects are loosely coupled.

The example that we found for the observer pattern occurs in the Runelite class (Runelite.java) within the start() method. Here, there exists a single EventBus object that manages registering and unregistering objects that would like to be added as a subscriber to a map of subscribers. We can see in EventBus.java that there is a register() and unregister() method registers the passed in subscriber to the event bus or unregisters all subscribed methods from the object passed in, respectively (Figure 5). The post method goes through each subscriber in the map to invoke the event passed in, allowing those subscribers to be notified that the event was triggered (Figure 5). Lastly, the example that we found in Runelite occurred on lines 316 to 337, when the event bus registers certain event listeners such as the ClientUI, the PluginManager, the ExternalPluginManager, and so on (Figure 6).

```
88 public EventBus() { this((e) -> Log.warn("Uncaught exception in event subscriber", e)); }
92
93 /** Registers subscriber to EventBus. All methods in subscriber and it's parent classes are checked for ...*/
100 public synchronized void register(@Nonnull final Object object)
101 { ... }
175
176 /** Unregisters all subscribed methods from provided subscriber object. ...*/
181 public synchronized void unregister(@Nonnull final Object object)
182 { ... }
209
210 /** Posts provided event to all registered subscribers. Subscriber calls are invoked immediately, ...*/
216 public void post(@Nonnull final Object event)
217 {
218     for (final Subscriber subscriber : subscribers.get(event.getClass()))
219     {
220         try
221         {
222             subscriber.invoke(event);
223         }
224         catch (Exception e)
225         {
226             exceptionHandler.accept(e);
227         }
228     }
229 }
230 }
```

Figure 5: The EventBus class which contains the methods register (line 100), unregister (line 181), and post (line 216) that partake in the observer pattern.


```

315 // Register event listeners
316 EventBus.register(clientUI);
317 EventBus.register(pluginManager);
318 EventBus.register(externalPluginManager);
319 EventBus.register(overlayManager);
320 EventBus.register(drawManager);
321 EventBus.register(infoBoxManager);
322
323 if (!isOutdated)
324 {
325     // Initialize chat colors
326     chatMessageManager.get().loadColors();
327
328     EventBus.register(partyService.get());
329     EventBus.register(overlayRenderer.get());
330     EventBus.register(clanManager.get());
331     EventBus.register(itemManager.get());
332     EventBus.register(menuManager.get());
333     EventBus.register(chatMessageManager.get());
334     EventBus.register(commandManager.get());
335     EventBus.register(lootManager.get());
336     EventBus.register(chatboxPanelManager.get());
337     EventBus.register(hooks.get());

```

Figure 6: In *Runelite.java* within the *start()* method, the event bus registers objects such as the *ClientUI*, *PluginManager*, and *ExternalPluginManager* as subscribers to an internal map of subscribers within its own class.

The pattern is used so that other objects and classes can listen to the events that they are specifically interested in, and do some work once those events are fired. For example, once a plugin is successfully loaded and started, the *PluginManager* calls *post()* on the event bus to notify all listeners of the *PluginChanged* event. One class that is interested in this event is the *VirtualLevelPlugin* class, which contains an *onPluginChanged()* method with the *@Subscribe* annotation to do some work once it is notified that a plugin has changed (*Figure 7*). In this case, the specific work that it performs is to invoke a method called *simulateSkillChange()* as a runnable on the client thread. The benefit of using observers for Runelite is that the system now has a way to manage what objects are interested in what event, so that each object can perform its own work once that event is fired. These objects, like the plugin manager, or the external plugin manager, don't need to know about each other. They are loosely coupled, and are only connected via the event bus. The pattern allows for object-to-object communication across the entire system without objects being required to elicit hard dependencies.

```

387 EventBus.register(plugin);
388 schedule(plugin);
389 EventBus.post(new PluginChanged(plugin, loaded: true));

```

Figure 7: The first image shows that the *PluginChanged* event is passed in as an argument to the event bus's *post()* method, to notify objects that are interested in this event.

```

83 @Subscribe
84 public void onConfigChanged(ConfigChanged configChanged)
85 {
86     if (!configChanged.getGroup().equals("virtuallevels"))
87     {
88         return;
89     }
90
91     clientThread.invoke(this::simulateSkillChange);
92 }

```

The second image displays the *onConfigChanged()* method with the *@Subscribe* annotation. This method exists within *VirtualLevelPlugin.java* and is fired once the *PluginChanged* event is posted via the event bus.

Iterator

The iterator pattern is a very common pattern that allows for a way to iterate over a collection of objects regardless of the type of list. Without this pattern, we could possibly use something like arrays or ArrayList but the problem with that is that we would have to change the way we write the loop. For instance to iterate through all elements within an array, we would specify the array's length property, but for an ArrayList we would use its size() method. By using an Iterator, we can specify whatever kind of list, set, or collection while keeping the underlying representation from becoming exposed.

Within Runelite, the iterator pattern is used in many classes primarily to iterate over collections of a certain type to do similar work on each element of the collection. For example, within the invoke() method of ClientThread.java, the Iterator class of the utility package provided by Java is used by calling the iterator() method on the private field ConcurrentLinkedQueue<BooleanSupplier>. In other words, an iterator object that is able to iterate over a ConcurrentLinkedQueue collection, which expects its elements to be of type BooleanSupplier, is returned when iterator() is called on line 92 in *Figure 8*. This allows the iterator to check if an element exists via hasNext(), and then allows the iterator to call next() to return the next non-null BooleanSupplier object. The loop itself attempts to call getAsBoolean() on the BooleanSupplier in order to see if that supplier should be removed from the queue, and does so if the boolean it got back returned true. This is shown in *Figure 8* where remove is checked to be true within the condition before actually removing the object from the queue.

```
89     void invoke()
90     {
91         assert client.isClientThread();
92         Iterator<BooleanSupplier> ir = invokes.iterator();
93         for (; ir.hasNext(); )
94         {
95             BooleanSupplier r = ir.next();
96             boolean remove = true;
97             try
98             {
99                 remove = r.getAsBoolean();
100             }
101             catch (ThreadDeath d)
102             {
103                 throw d;
104             }
105             catch (Throwable e)
106             {
107                 log.warn("Exception in invoke", e);
108             }
109             if (remove)
110             {
111                 ir.remove();
112             }
113         }
114     }
```

Figure 8: Within ClientThread.java, the invoke() method gets the iterator from the collection specified by the variable invokes. The iterator() method is expected to return an Iterator that can iterate through elements of type BooleanSupplier.

The benefit of using the Iterator pattern in this example is that it provides an easy and convenient way to access objects of type `BooleanSupplier` within the collection, in this case a `ConcurrentLinkedQueue` (*Figure 9*). However, what is more important is that if the developers decide to update `Runelite` and the `ClientThread` class specifically and they decided that concurrency is no longer a problem or that a different collection should be used, they only simply need to change the collection type in its field declaration. They don't need to change the implementation at all within the `invoke()` method. This pattern allows for flexibility when the developers of a system decide to use different structures to model their objects but wish to keep the implementation the same.

```
35  @Singleton
36  @Slf4j
37  public class ClientThread
38  {
39      private ConcurrentLinkedQueue<BooleanSupplier> invokes = new ConcurrentLinkedQueue<>();
```

Figure 9: The field variable `invokes` is of type `ConcurrentLinkedQueue<T>`. If developers decide to change the collection, they need only change this line due to their use of the Iterator pattern.

Singleton

A singleton ensures a class only has one instance, and provides a global point of access to it. Although the use of singletons should be generally avoided, this really depends on the system that is being developed and the architecture of the system. If there is only one instance of a class, this saves the system from generating multiple objects of classes that it may not need. In this perspective, singletons can be used to create a more efficient system especially in terms of memory. The reason why it is generally avoided is because it is a global variable, and this can cause an issue if it is accessed from multiple classes or if the system relies heavily on maintaining concurrency. For the Runelite client, this isn't an issue and instead, the singleton pattern is a huge benefit to it and is used throughout the entire source code.

In Runelite if we search for the annotation `@Singleton`, we can see that there are 82 uses, in 79 files. We will choose one example to talk about, which is a core feature of Runelite, the `Notifier.java` class.

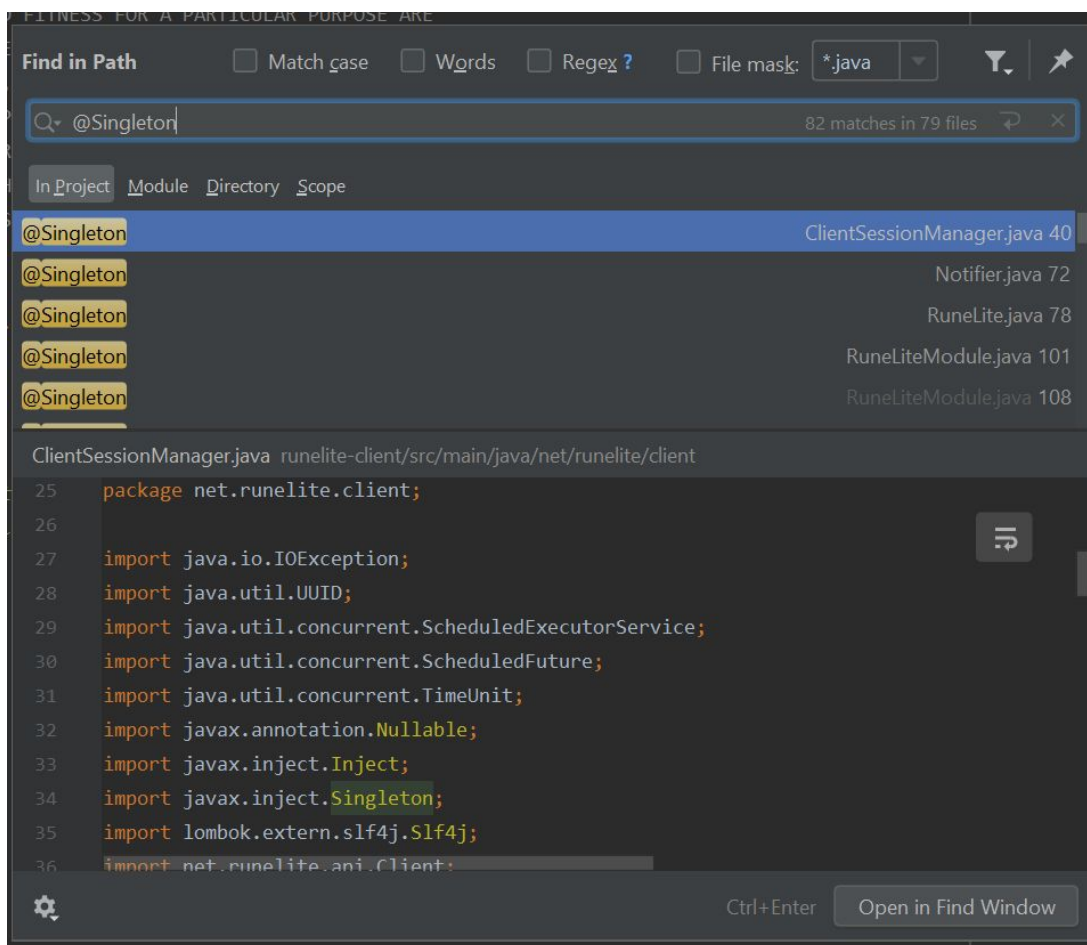


Figure 10: Using the search function within IntelliJ, the query for "`@Singleton`" returns several classes that uses the singleton pattern.

```

72  @Singleton
73  @Slf4j
74  public class Notifier
75  {
76      @Getter
77      @RequiredArgsConstructor
78      public enum NativeCustomOff
79      {
80          NATIVE( name: "Native"),
81          CUSTOM( name: "Custom"),
82          OFF( name: "Off");
83
84          private final String name;
85
86          @Override
87          public String toString() { return name; }
88      }
89
90
91
92
93      // Default timeout of notification in milliseconds
94      private static final int DEFAULT_TIMEOUT = 10000;
95      private static final String DOUBLE_QUOTE = "\"";
96      private static final Escaper SHELL_ESCAPE = Escapers.builder()
97          .addEscape( c: '"', replacement: "\\")
98          .build();
99
100      // Notifier properties
101      private static final Color FLASH_COLOR = new Color( r: 255, g: 0, b: 0, a: 70);
102      private static final int MINIMUM_FLASH_DURATION_MILLIS = 2000;
103      private static final int MINIMUM_FLASH_DURATION_TICKS = MINIMUM_FLASH_DURATION_MILLIS / Constants.CLIENT_TICK_LENGTH;
104
105      private static final String appName = RuneLiteProperties.getTitle();
106
107      private final Client client;
108      private final RuneLiteConfig runeLiteConfig;
109      private final ClientUI clientUI;
110      private final ScheduledExecutorService executorService;
111      private final ChatMessageManager chatMessageManager;
112      private final EventBus eventBus;
113      private final Path notifyIconPath;
114      private final boolean terminalNotifierAvailable;
115      private Instant flashStart;
116      private long mouseLastPressedMillis;

```

Figure 11: *Notifier.java* class, which displays the `@Singleton` annotation and dependency on line 72.

We can see that on top of the class, there is a `@Singleton` annotation, which helps with Injecting the class into a singleton. Notifier needs to be a singleton, because of the essential role that it plays for Runelite. It is used by over 20 plugins like the IdleNotifier or Fishing plugin to send notifications to the host operating system. Imagine all those plugins creating their own instance of Notifier and sending notifications to the operating system at once, it would be chaos. Hence why making Notifier a singleton makes sense. It saves the system from instantiating the same type of object that it doesn't need, and minimizes the work that the garbage collector has to do when the object is no longer used. The singleton pattern in Notifier occurs almost everywhere in Runelite within many other classes, so it is a reoccurring and a prevailing pattern.







Issue we chose to tackle

Issue link: <https://github.com/runelite/runelite/issues/9601>

This was one of the interesting issues we looked at in the previous homework. Essentially it requires contributors to get the specific center tile's coordinates in game for each and every 125 possible locations to close the issue. The hard part of this issue and why community contribution is required can only be explained with some background context.

In Old School Runescape(OSRS from here on out), there is an activity that players can participate in called Treasure Trails (https://oldschool.runescape.wiki/w/Treasure_Trails). Players can participate in this activity by receiving a Clue Scroll item, and continuously solving the clues until a Reward Casket is received.

Clue Scrolls come in 6 difficulty tiers with varying step lengths, requirements, and puzzles to solve. This can be seen in the image on the right.

	Tier	Length
	Beginner	1-3 steps
	Easy	2-4 steps
	Medium	3-5 steps
	Hard	4-6 steps
	Elite	5-7 steps
	Master	6-8 steps

For each step in the Clue Scroll players can receive a random type of clue listed in the table below. Our issue is only interested in the Hot Cold clue type of Master clues. Which is a tiny subset of all clues in the game.

	Anagrams	Challenge scrolls	Ciphers	Coordinates	Cryptic clues	Emote clues	Hot Cold	Light boxes	Maps	Puzzle boxes
Beginner	Solutions	N/A	N/A	N/A	Solutions	Solutions	Solutions	N/A	Solutions	N/A
Easy	N/A	N/A	N/A	N/A	Solutions	Solutions	N/A	N/A	Solutions	N/A
Medium	Solutions	Solutions	Solutions	Solutions	Solutions	Solutions	N/A	N/A	Solutions	N/A
Hard	Solutions	Solutions	Solutions	Solutions	Solutions	Solutions	N/A	N/A	Solutions	Solutions
Elite	Solutions	Solutions	N/A	Solutions	Solutions	Solutions	N/A	N/A	Solutions	Solutions
Master	Solutions	N/A	N/A	Solutions	Solutions	Solutions	Solutions	Solutions	N/A	Solutions

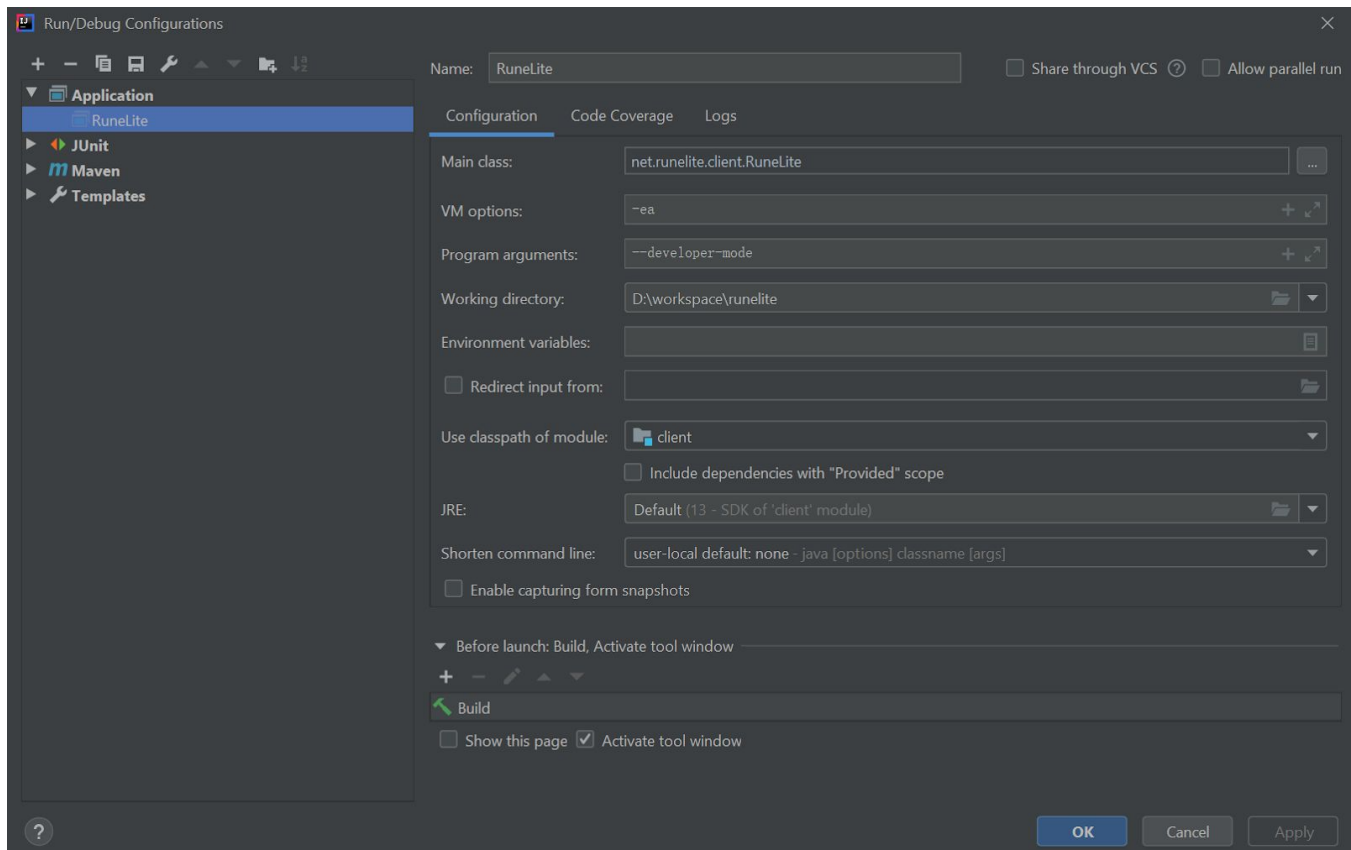
Further increasing the difficulty of this issue. Clue Scrolls are usually acquired randomly from monster drops, or while gathering resources in game, with increasing rarity with their difficulty level, with the

exception of Master Clues, which can only be acquired from the Reward Casket of lower difficulty clues, or by turning in 1 of each lower difficulty Clue Scrolls. This makes the Master Clue Scroll a lot rarer compared to the previous elite difficulty, with a 1 in 5 chance of getting it from an Elite Reward Casket.

Coupled with the fact that more than half of the 125 locations are already centered, and an average of only being able to get one Hot Cold step per Master Clue Scroll, this issue becomes extremely hard to contribute to, hence having the “help wanted” tag, needing the community to contribute and solve together.

After finding an uncentered location in game, contributors are required to provide image proof (screenshot) of the location with the 9 by 9 tiles highlighted via the Runelite “Ground Markers” plugin. Also the image should contain the location overlay plugin that can only be enabled in Runelite’s developer mode.

To get developer mode running in Runelite, Runelite must be compiled and built manually with the program argument `--developer-mode` enabled.



Base	2984, 3568
Local	6720, 5696
Scene	52, 44
Tile	3036, 3612, 0
Map regions	11831
	11832
	11833
	12087
	12088
	12089
	12343
	12344
	12345

Hot/Cold Clue

Possible locations:

Wilderness:

- South-east of the Dark Warriors' Fortress, level 12 Wilderness.

All that needs to be changed was one number of one line in the `HotColdLocation.java` file that contained all the clue location's coordinates and their description.

```
✓ HotColdLocation: Center South-east Dark Warriors' Fortress location

deon9718 committed 24 minutes ago    commit 0a4e2b74f9053da46ac881808623673737a2d

▼ 2 ■■■ .../src/main/java/net/rumelle/client/charscrolls/chars/hotcold/HotColdLocation.java  ...

381  @ -147,7 +147,7 @@
382  WESTERN_PROVINCE_TYRAG(new WorldPoint(2208, 3158, 0), WESTERN_PROVINCE, "Near Tyras Camp."),
383  WESTERN_PROVINCE_ZUL-ANDRA(new WorldPoint(1219, 3857, 0), WESTERN_PROVINCE, "The northern house at Zul-Andra."),
384  WILDERNESS_5(new WorldPoint(3373, 3556, 0), WILDERNESS, "North of the Grand Exchange, level 5 Wilderness."),
385  WILDERNESS_12(new WorldPoint(3038, 3612, 0), WILDERNESS, "South-east of the Dark Warriors' Fortress, level 12 Wilderness."),
386  WILDERNESS_20(new WorldPoint(3275, 3676, 0), WILDERNESS, "East of the Corporal Beast's lair, level 20 Wilderness."),
387  WILDERNESS_27(new WorldPoint(3126, 3738, 0), WILDERNESS, "Inside the Ruins north of the Gateway of Shadows, level 27 Wilderness."),
388  WILDERNESS_28(new WorldPoint(3377, 3737, 0), WILDERNESS, "East of Wenevetti's nest, level 28 Wilderness."),

389  WESTERN_PROVINCE_TYRAG(new WorldPoint(2208, 3158, 0), WESTERN_PROVINCE, "Near Tyras Camp."),
390  WESTERN_PROVINCE_ZUL-ANDRA(new WorldPoint(1219, 3857, 0), WESTERN_PROVINCE, "The northern house at Zul-Andra."),
391  WILDERNESS_5(new WorldPoint(3373, 3556, 0), WILDERNESS, "North of the Grand Exchange, level 5 Wilderness."),
392  WILDERNESS_12(new WorldPoint(3038, 3612, 0), WILDERNESS, "South-east of the Dark Warriors' Fortress, level 12 Wilderness."),
393  WILDERNESS_20(new WorldPoint(3275, 3676, 0), WILDERNESS, "East of the Corporal Beast's lair, level 20 Wilderness."),
394  WILDERNESS_27(new WorldPoint(3126, 3738, 0), WILDERNESS, "Inside the Ruins north of the Gateway of Shadows, level 27 Wilderness."),
395  WILDERNESS_28(new WorldPoint(3377, 3737, 0), WILDERNESS, "East of Wenevetti's nest, level 28 Wilderness.");
```

The pull request was then submitted after changing 3038 to 3036, the mega issue was mentioned as well by having #9601 (the issue's number) in the comments.

runelite / runelite

Watch
83
Star
2.7k
Fork
3.6k

Code
Issues 1,136
Pull requests 450
Actions
Wiki
Security
Insights

HotColdLocation: Center South-east Dark Warriors' Fortress location #10917

Merged
Nightfirecat merged 1 commit into runelite:master from deon9718:hot-cold-fix 2 minutes ago

Conversation 0
Commits 1
Checks 0
Files changed 1

deon9718 commented 2 days ago • edited

Contributor
+

Image proof:

```

Base: 2904, 2508
Local: 6728, 5636
Sign: 52, 14
Tile: 1836, 3612, 0
Playregions:
11841
11832
11830
12007
12006
12008
12009
12010
12011
12012
12013
12014
12015

```

HotCold clue
Possible locations:
Wilderness
- South-east of the Dark
Warriors' Fortress, level 12
Wilderness

#9601

1

HotColdLocation: Center South-east Dark Warriors' Fortress location
0cae82b

deathbeam added the clues label 2 days ago

Nightfirecat merged commit 0eaf678 into runelite:master 2 minutes ago

View details
Revert

Trevor159 mentioned this pull request 2 minutes ago

HotColdClues: mega issue for centering dig locations #9601

Open

70 of 125 tasks complete

Reviewers
No reviews

Assignees
No one assigned

Labels
clues

Projects
None yet

Milestone
No milestone

Linked issues
Successfully merging this pull request may close these issues.
None yet

Notifications
Customize

Unsubscribe

You're receiving notifications because you authored the thread.

3 participants

The pull request was merged 3 days later and the list in the mega issue was updated with a reference to our pull request number #10917.

- ☐ In the centre of the Rimmington mine.
- ☒ South-east of the Dark Warriors' Fortress, level 12 Wilderness. (#10917)
- ☒ Northern part of the Lovakengj blast mine. (#9602)