Designing a Modular RuneLite Plugin Core (HeistCore)

Current Project Structure

HeistCore Repository – The project (HeistCore) is organized in multiple branches, notably a `core-rl` branch for RuneLite-specific integration and a `core-java` branch for core logic in plain Java. This separation suggests an architecture where game-specific hooks (in `core-rl`) are decoupled from the generic bot logic (in `core-java`). In other words, `core-rl` contains the plugin code that interacts with the RuneLite client API (receiving game events, invoking actions, etc.), while `core-java` holds reusable logic that can be easily maintained and tested without direct RuneLite dependencies.

Plugins vs. Handlers – Currently, you likely have individual plugins for activities like woodcutting, fishing, etc., each containing similar code for common tasks (e.g. dropping loot when inventory is full). The goal is to refactor these into **lightweight plugins** that delegate to centralized **handler/services** modules. For example, rather than every skilling plugin implementing its own inventory dropping logic, a shared `InventoryService` can provide that functionality. This will drastically reduce code duplication and make it much easier to develop new plugins (like quest automation) on top of a robust core.

Branch Contents (Hypothesized):

- *core-rl branch*: Contains RuneLite plugin classes (annotated with `@PluginDescriptor`, extending RuneLite's `Plugin` class) and uses RuneLite's API via injection (e.g. `@Inject Client client;`). This is where the game event handlers (`@Subscribe` methods for events like `GameTick`, `AnimationChanged`, etc.) live, forwarding those events to the core logic as needed. It likely also contains any RuneLite-specific utilities (for example, using `Client.invokeMenuAction(...)` to simulate clicks, or using the RuneLite event bus).

- *core-java branch*: Contains the definition of various **services/handlers** (Inventory handling, movement, combat, etc.) and possibly a framework for scripting sequences of actions (especially useful for quests). This code is written in pure Java and does not depend on RuneLite internals except through abstract interfaces – making it easier to test or even port to other environments. In practice, the `core-rl` plugin would call into these core-java services, providing them with necessary game-state data from the RuneLite client.

(Even if the current code isn't perfectly split yet, this is the direction to head: a clean separation of concerns between game API integration and the logic of what to do.)

Goals Recap

Before designing the new system, let's restate the core goals for clarity:

- **Reusable Handlers for All Scenarios:** Develop a set of modular **handler services** (e.g. `InventoryService`, `MovementService`, `CombatService`, etc.) that any plugin can call. This will make plugin code much lighter. Whether the plugin is for skilling (woodcutting/fishing), PvM boss fights, PvP/PKing, or questing, it should utilize the same core services for common tasks (inventory management, moving the player, interacting with game objects/NPCs, combat actions, etc.).

- **Maximize Modularity and Clarity:** The system should be structured so cleanly that creating a new plugin (for example, automating a quest from a wiki guide) is straightforward. The plugin should mostly consist of high-level steps or state definitions, while the *how* (the low-level clicking, waiting, item using, etc.) is handled by the core services.

- **Game State Driven Logic: Never hard-code assumptions about game state** – always derive state from the RuneLite client data. For example, don't "assume" the player is chopping a tree just because a chop action was sent; instead, confirm by checking the player's animation is the woodcutting animation ID, or that the tree object actually vanished, etc. Every action should be verified via the game's feedback (animations, object IDs, varbits, etc.). This makes the bot more robust and in line with how RuneLite plugins typically monitor game events (no blind waits).

- **Use RuneLite API & Conventions:** Always reference the official RuneLite API or existing open-source plugins for the correct way to perform actions. This includes using RuneLite's provided constants for IDs (object IDs, item IDs, animation IDs, etc.) rather than magic numbers, and utilizing RuneLite's event bus and client thread for interactions. We want to leverage the proven patterns from RuneLite's codebase as much as possible for stability and compliance. (Citations below will point to relevant RuneLite or plugin code.)

With these goals in mind, let's outline the **ideal modular system** design and then discuss how to implement it step-by-step.

Proposed Modular System Design

Core Service Modules

We will introduce several **service classes** (or managers/handlers) that encapsulate specific domains of functionality. Each service will expose high-level methods that plugins can call, and internally use the RuneLite client API to carry out the actions. All services should be singleton-like (one shared instance, or at least one per game session) so that state is consistent across plugins. Key services include:

- **InventoryService** – Manages all inventory and item-related actions and queries:
- *Dropping items:* e.g. `dropItem(Item item)` or `dropAllExcept(Set<Integer> itemIds)`. This would use RuneLite's menu action API to perform item drops. For example, RuneLite's client provides an `invokeMenuAction` method to simulate menu clicks on items [1]. To drop an item,

one would call something like `client.invokeMenuAction("Drop", itemName, itemId, MenuAction.ITEM_SECOND_OPTION.getId(), slotIndex, 9764864);` – using the proper opcode for "drop" (in default OSRS, "Drop" is typically the second menu option on items) and the item's inventory slot parameters. We can lookup or compute the correct `param0/param1` (usually the interface and container IDs) from the RuneLite API. The service can hide this complexity behind a simple method call. (If using the RuneLite API's enums, we'd use `MenuAction.ITEM_SECOND_OPTION` for drop – these constants are defined in RuneLite's API [2] [3].)

- *Item usage:* e.g. `useItemOnItem(int itemIdSrc, int itemIdDst)` or `useItemOnObject(int itemId, int objectId)`. This would handle scenarios like using a tinderbox on logs, or using a quest item on a shrine, etc. RuneLite's API supports these via menu actions as well (there are `MenuAction.ITEM_USE` and related opcodes). For convenience, the service could provide methods like `InventoryService.use(itemNameOrID)` to click "Use" on an item, and then the `ObjectInteractionService` (described later) could handle clicking the target. In practice, some community APIs (like OpenOSRS's **unethicalite** API) have high-level methods: for example, in an open-source snippet, `Inventory.getFirst("Oak logs").useOn(Inventory.getFirst("Tinderbox"));` is used to light a fire [4]. Our service can implement a similar logic: find the item in the inventory and invoke the appropriate menu actions to use it on the target.
- *Equipping/Unequipping:* e.g. `equipItem(int itemId)` to wear a piece of equipment. This would likely simulate the "Wear" action on an inventory item. (RuneLite constants like `MenuAction.ITEM_FIRST_OPTION` could be the "Wear" if it's the top option for that item, otherwise we might need to find the option index if not default.) The service can also check if the item is already equipped, etc., via the equipment inventory container.
- *Consuming items:* e.g. `eatFood(int foodId)` or `drinkPotion(String potionName)`. These would trigger the appropriate menu action ("Eat", "Drink") on the item. As with other actions, it should verify success by checking resulting state (e.g., after eating, player's health should increase or the item disappears from inventory).

- *Inventory State Queries:* The service can also expose methods like `isFull()` (inventory 28/28) or `contains(int itemId)` etc., abstracting the RuneLite client's inventory container. For example, one can get the inventory via `client.getItemContainer(InventoryID.INVENTORY)` and check its size. This is already done in some plugins – e.g., a bot script checks `if (Inventory.isFull()) { ... }` to decide when to drop or bank items [5]. We'll implement similar utility in our service.

- **MovementService** – Handles player movement and positioning:

- *Walking to Locations:* The service could offer `walkTo(WorldPoint target)` or `walkTo(int x, int y)` which under the hood finds a path or at least clicks towards the target. RuneLite itself doesn't provide a built-in pathfinder to arbitrary points (except the "Shortest Path" plugin on the hub uses the world map graph). We might integrate an existing pathfinding algorithm or simply use tile-by-tile navigation for now. At minimum, we can use `client.invokeMenuAction("Walk here", "", 0, MenuAction.WALK.getId(), destX, destY)` to click on the ground [6]. The MovementService could also tie into **RuneLite's scene data** to avoid obstacles, or call an external API for complex paths if needed. But a simpler approach is to use a predefined path or successive "walk here" clicks in the direction of the target.

- *Running & Energy:* Expose methods to toggle run (`setRunning(true/false)`) when appropriate (e.g. if the player has enough energy and needs faster movement). RuneLite API provides the run energy level via `client.getEnergy()` and a setting to toggle run (this is often done by simulating a click on the run button widget or setting a varbit). Our service can monitor energy and automatically turn on run when beneficial (for questing or long walks, for example).
- *Object/NPC traversal:* For questing or bossing, MovementService might also provide higher-level moves like `walkToObject(int objectId)` – it could locate the object in the scene and move the player adjacent to it. This could reuse the **ObjectInteractionService** (below) to find the object's location, then call walkTo that tile.

- *Avoiding Idle/Waiting:* The MovementService, like others, should ensure not to spam clicks. It must check if the player is currently busy moving before issuing another command. Typically, you can check `client.getLocalPlayer().isMoving()` or monitor the `GameTick` event to see if the player's position changes. Only issue a new move when the previous move is complete (or if stuck).

- **ObjectInteractionService** – Responsible for finding and interacting with game world objects (trees, rocks, bank booths, doors, etc.):

- *Locating Objects:* Provide methods to get game objects by ID or name in the nearby scene. RuneLite's client has a region scene graph you can traverse (e.g., `client.getScene().getTiles()` or use the query classes). There is also a `GameObjectSpawned` event we can listen to, but for on-demand finding, one can loop through tiles around the player to find a `GameObject` with a matching `ID` (RuneLite's **ObjectID** constants should be used for readability – e.g., `ObjectID.OAK_TREE` [7]).
- *Interacting (Clicking) Objects:* Once an object is identified, the service can invoke the menu action to interact. For example, to chop a tree: `client.invokeMenuAction("Chop down", "Tree", treeId, MenuAction.GAME_OBJECT_FIRST_OPTION.getId(), treeLocalX, treeLocalY)` (the exact parameters would come from the `GameObject` and the `MenuAction` for the first object option). This encapsulates all object clicks like "Talk-to" for an NPC (which is technically similar but for NPCs) or "Open" a door, etc.
- *Ensuring Correct Interaction:* After calling an interaction, the service should verify it by observing game state changes. For instance, if we clicked "Chop down Tree", we expect the player's animation to soon change to a woodcutting animation. We can wait for an `AnimationChanged` event or monitor `client.getLocalPlayer().getAnimation()` to become the woodcutting animation ID. **Only if that happens do we confirm the action succeeded**; if not, we might retry or handle the failure (maybe the tree was already gone, etc.).
    - RuneLite provides animation IDs (via an `AnimationID` class for common ones) – e.g., woodcutting animations have specific IDs depending on the axe used. These should be referenced from the RuneLite API rather than hard-coded. In open-source data, for example, there's an `AnimationID` list in RuneLite/OpenOSRS we can consult [8]. Always using these named constants makes the code easier to update when OSRS changes (and more self-documenting).
- *NPC Interactions:* We could fold NPC interactions here or have a separate **NPCInteractionService**. The mechanics are similar: use NPC IDs (RuneLite's `NpcID` constants [9]) to find targets and then invoke menu actions like "Attack" or "Talk-to". RuneLite's menu action for NPC first option might be `NPC_FIRST_OPTION` (and so on) which we would use accordingly.

- *Dialogues (Quest related):* If the interaction triggers a dialogue, this service could defer to the DialogueService (below) to handle continuing the conversation. For example, after initiating a "Talk-to" with an NPC, DialogueService can take over to click through chat options.

- **DialogueService** – Streamlines handling of in-game dialogues and interfaces, crucial for questing:

- *Advancing Dialogue:* Provide a method like `continueDialogue()` which clicks the "Continue" button in chat. RuneLite identifies the "Continue" menu action as `MenuAction.WIDGET_CONTINUE` [10]. We can subscribe to the `DialogProcessed` or simply watch for the chat interface (Widget IDs for the chat box) being open. A simple approach is to call `client.invokeMenuAction("Continue", "", 0, MenuAction.WIDGET_CONTINUE.getId(), 0, 0)` repeatedly when a dialogue is open. This effectively presses the spacebar/continue on dialogues. (RuneLite's own menu entry swapper does similar for dialogues.)
- *Choosing Options:* For dialogues with options (e.g. a quest choice), the service can select an option by widget ID. RuneLite's `WidgetInfo.DIALOG_OPTION` identifiers can be used to find the correct widget for each choice. For example, if the option appears as a widget, we can invoke a `WIDGET_FIRST_OPTION` on it to select the first dialogue option, etc. The DialogueService could expose `chooseOption(String optionText)` to scan the dialogue options for a matching text and then click it.
- *Detecting Dialogue Completion:* It should detect when dialogues end (maybe by the disappearance of the chat interface or the presence of the game window). At that point, it returns control to the quest script to proceed to the next step.

- *Handling Yes/No Prompts or Other UI:* Many quests have special interfaces (such as confirmation dialogs, puzzle interfaces, etc.). The DialogueService (or perhaps a more general **InterfaceService**) could handle clicking specific widgets on these interfaces. For instance, if a quest prompt appears ("Are you sure you want to do X?" with Yes/No), the service can detect the interface ID and call the appropriate widget action. (RuneLite's `WidgetID` class lists all interface groups [11], which we should reference to identify these cases.)

- **CombatService** – Focused on combat interactions, which is vital for bossing and PKing:

- *Attacking targets:* Method `attackNpc(int npcId)` or `attackNpc(String npcName)` to find the NPC and invoke the "Attack" action. Similar to general object interaction, but we ensure it checks combat states. For example, after issuing an attack, verify the player's **interacting entity** is the target NPC (RuneLite's `client.getLocalPlayer().getInteracting()` should point to the NPC if attack was successful).
- *Special attacks:* Provide a helper to toggle special attack (by clicking the special attack orb or setting the varbit). RuneLite doesn't allow direct varbit setting from a plugin for such things unless via menu simulate. However, an approach is to call `client.invokeMenuAction("Use <br> Special Attack", "", 1, MenuAction.CC_OP.getId(), -1, SPEC_ORB_WIDGET_ID)` (where SPEC_ORB_WIDGET_ID is the widget for special attack). The service can manage when to do this (e.g., if special attack energy >= required amount).
- *Prayer management:* Methods to activate or deactivate prayers (e.g. `setPrayer(Prayer p, boolean enable)`). This can be done by clicking the prayer widgets. The service could cross-link with InventoryService for drinking prayer potions when prayer is low, etc., or with Combat logic to

enable protection prayers when fighting. We should always use the client's knowledge: e.g., check `client.isPrayerActive(Prayer.PROTECT_FROM_MELEE)` to avoid redundant toggles.
- *Healing and Buffs:* In dangerous situations (bossing or PKing), an Auto-heal feature can be part of CombatService or a separate **SafetyService**. For example, `healIfNeeded()` which checks HP and uses food via InventoryService if below threshold, or `drinkPotionIfNeeded(Stat stat)` for combat boosting potions. RuneLite provides current HP via `client.getBoostedSkillLevel(Skill.HITPOINTS)` vs max `getRealSkillLevel`. Similarly for Prayer points. We can use those to trigger item usage (food, potions) when necessary.
- *Status Monitoring:* CombatService should subscribe to events like `HitsplatApplied` (to see damage taken), and monitor `client.getLocalPlayer().getHealth()` etc. This way it can react promptly (e.g., eat immediately if a big hit landed). It can also track the opponent's HP if needed (for boss plugins).

- *PK-specifics:* If focusing on PvP, additional features like opponent detection (perhaps via RuneLite's player indicators or just scanning local players), and activating appropriate counters (e.g., overhead prayers, teleports) could be included. This might be beyond initial scope, but the architecture should allow extension for these.

- **BankService** – To support skilling/questing that involves banking:

- *Bank navigation:* If inventory is full (and not just dropping items), a skilling script may need to bank items. BankService can handle opening the bank (finding the nearest bank booth or chest via ObjectInteractionService, then invoking "Bank" action).
- *Item deposit/withdraw:* Provide methods like `depositAllExcept(int[] itemIds)` or `withdrawItem(int itemId, int quantity)`. This involves interacting with the bank interface widgets. RuneLite's WidgetID for bank inventory and player inventory are known (e.g., bank inventory group ID). We can simulate widget clicks for deposit/withdraw. For example, a deposit-all could be a single widget button click ("Deposit inventory" button). With our service approach, a quest plugin could simply call `bankService.ensureItemInInventory(itemId, quantity)` which checks if you have X of an item, and if not, opens bank and withdraws it – all encapsulated.

- *Conditions:* BankService should also verify actions (ensure bank actually opened by checking `client.getItemContainer(InventoryID.BANK)` becomes non-null, etc., and close the bank via the interface if needed).

- **QuestScript / Task Engine** – This is more of a framework on top of the services:

- *Quest Task Structure:* For questing, it's ideal to have a structured way to define steps. We can create classes like `QuestTask` or simply use a state machine in the plugin. Each quest step would have:
    - A **condition** to detect if the step is already done (e.g., a specific varbit or varplayer value, or an item in inventory, etc.).
    - An **action** to perform for that step using the core services (e.g., "talk to NPC X", "use item Y on object Z", "go to location L"). This action can be composed of multiple service calls in sequence with appropriate waiting.
    - Optionally, a **post-condition** or verification (though often the condition for the next step implicitly serves that purpose).

- *Example:* Suppose a quest step is "Chop a tree and gather logs". We might define:
  - Condition: `Inventory contains Logs` (or quest varbit updated).
  - If condition not met, Action: `ObjectInteractionService.interact(ObjectID.TREE, "Chop down")`, then wait until the player's animation is woodcutting and then returns to idle (tree finished) **and** logs appear in inventory. Use InventoryService to confirm logs added. Only then mark step done.
  - Then next step triggers, etc.
- This kind of structured approach means writing a new quest plugin is as simple as listing out these steps and their parameters (object IDs, NPC IDs, dialogues, etc.), without rewriting *how* to chop, how to walk, how to talk – since those are handled by services. It achieves the goal: given a wiki page, a developer can mostly translate the textual instructions into a sequence of service calls and conditions.

- *Utilizing RuneLite Quest Data:* We should leverage RuneLite's knowledge base for quests. For instance, RuneLite/QuestHelper already defines quest state varbits/varplayers and item requirements. The **QuestVarPlayer/QuestVarbits** from QuestHelper provide mappings of quest stages [12]. Our quest framework can use those to check progress. For example, if a quest's varplayer value equals 50, that might mean quest completed. Using these constants (from RuneLite API and QuestHelper's database) is crucial rather than using arbitrary numbers – it ensures accuracy and easy updates if the quest data changes.

- **HUD/Overlay Service** – (Optional but useful) Provides on-screen feedback:

- This could be a simple overlay that displays the current state or next action. For debugging and user information, it's invaluable. Each plugin could push status messages (e.g. "Status: Chopping tree...", "Inventory full, dropping logs.", "Moving to bank") to a central HUD service which renders them on the RuneLite overlay panel.
- Additionally, for quest plugins, the HUD might highlight the NPC or object you need to interact with (similar to QuestHelper highlights) – using RuneLite's Overlay and rendering tools. We can draw highlights around the target object/NPC using the scene coordinates (RuneLite API can draw polygons on objects).
- This HUD should be part of core so that every plugin automatically benefits from consistent displays (and we don't write overlay code for each plugin separately). It aligns with the earlier HUD concept you mentioned – showing useful info to the user such as current task, next waypoint, etc., in a standardized way.

Each service should be designed with **RuneLite best practices** in mind: use the client's data and methods, avoid blocking the game thread, and coordinate via events. We will now discuss some implementation strategies and references to RuneLite API to achieve this.

Event-Driven Architecture and State Verification

To adhere to the *"game-state driven"* philosophy, our plugins and services will rely heavily on RuneLite's **event bus** and client state queries instead of arbitrary delays. Some key patterns:

- **Animation and Idle Detection:** Use the `AnimationChanged` and `GameTick` events to know what the local player is doing. For example, subscribe to `AnimationChanged` events in core-rl plugin; if `event.getActor() == client.getLocalPlayer()`, we know our player's animation

changed. The InventoryService or Task engine can use this to detect when an action completes. RuneLite's Idle Notifier plugin logic is a great reference – it checks for when the player's animation goes from a non-idle state back to idle to trigger notifications (e.g., after woodcutting stops). In our case, we can use a similar check to move to the next action. In code, an idle state is typically when `client.getLocalPlayer().getAnimation()` returns -1 (no animation) and the player is not moving.

For example, in an unethicalite plugin, they ensured the player is idle before proceeding to the next action like so: `if (client.getLocalPlayer().isMoving() || !Players.getLocal().isIdle()) return;` [13]. This ensures that if the player is currently moving or animating (not idle), the script waits (does nothing) until idle. We will incorporate such checks in **every loop or repeated action** to avoid interrupting ongoing actions.

- **Game Object/Item Spawn Events:** For actions like looting or reacting to objects, services can subscribe to `ItemSpawned`, `GameObjectSpawned`, etc., if needed. For instance, if we chop a tree, we might listen for a `GameObjectDespawned` event of that tree ID to know the tree is gone (meaning success). Or if waiting for loot to drop, `ItemSpawned` on ground can signal that. However, these can often be handled by simpler means (animation ending and inventory count increased for woodcutting = log acquired).

- **GameTick for Polling:** The `GameTick` event (which fires every game tick, ~0.6 seconds) is useful for periodic checks. It's often used to implement state machines in RuneLite plugins. For example, a simplified woodcutting plugin might on each tick: check if not currently chopping and inventory not full -> find a tree and click it; if inventory full -> drop items; etc. We can use a similar pattern in our Task engine or services. The key is to make these checks lightweight and quick.

- In our design, the **Quest Task Engine** could evaluate the current step's condition each tick and, if not complete and player is free, execute one incremental action (like click tree). Then it waits for the next relevant event (animation finish, etc.) to truly mark completion, rather than spamming clicks every tick.

- The GameTick can also be used to decrement timers or manage timeouts (e.g., if something takes too long, assume it failed and retry).

- **Client Thread and Threading Considerations:** One crucial RuneLite rule: *All game-interaction must happen on the client thread.* RuneLite's `ClientThread` ensures thread-safe calls into the game engine. Our service methods that call `invokeMenuAction` or manipulate widgets must run on the client thread. In the core-rl plugin, you should inject the `ClientThread` and use `clientThread.invokeLater(Runnable)` to schedule actions properly. For example, to drop an item we might do:

```
clientThread.invokeLater(() -> {
    client.invokeMenuAction("Drop", itemName, itemId,
MenuAction.ITEM_SECOND_OPTION.getId(), itemSlot, 9764864);
});
```

This pattern is used widely in RuneLite. In fact, the RuneLite devs encourage using `clientThread.invokeLater()` or `clientThread.invoke()` to ensure the call runs in sync with the game loop [14]. Failing to do so (e.g., calling `invokeMenuAction` directly from an event callback or another thread) can cause race conditions or even client crashes. We will wrap all such calls accordingly.

Additionally, if we run any long computations or waiting loops, we should **not block the client thread**. The Reddit discussion on RuneLite's ClientThread explains that if you do a while-loop on the client thread, you'll freeze the game [15]. Instead, lengthy logic (like pathfinding or complex decision-making) can run on a separate thread or be split across ticks/events. Our architecture could include a small scheduler (as shown in that Reddit post) to run script logic in a background thread and use `invokeLater` for actual game actions. This is more advanced, but important for scalability.

Using RuneLite APIs and Data

To ensure we use the RuneLite API correctly (and avoid reinventing the wheel), we will constantly refer to RuneLite's code and classes:

- **Game Constants:** Leverage the classes RuneLite provides for IDs and values:
- `ObjectID`, `NpcID`, `ItemID`, `AnimationID`, `GraphicID`, etc. These are auto-generated constants mapping in-game IDs to names [16]. Using them makes code readable and robust. For example, use `ItemID.SALMON` instead of `329` or `ObjectID.TREE` instead of some number. This also helps when the game updates – RuneLite updates these references promptly.
- **ItemCollections:** QuestHelper plugin maintains useful collections of items (e.g., all types of pickaxes, axes, teleport jewelry, etc.) in its `ItemCollections` class [17] [18]. We can similarly maintain common item groups in our core (or even import QuestHelper's if license allows) to easily check "does the player have *any* axe?" or choose the best food available, etc.

- **Varbits/VarPlayers:** Use RuneLite's `Varbits` and `VarPlayer` enums for checking quest states, settings, or other toggles. For example, there is a Varbit for the "shift-click to drop" setting, and Varbits for each diary or quest stage. QuestHelper's `QuestVarbits` and `Quest` classes list those for quests [12], which we can use to know if a quest is complete or how far along it is. This is extremely useful for quest plugin logic (to skip steps already done, etc.). Always prefer using these names (e.g., `VarPlayer.QUEST_COOKS_ASSISTANT` with a value) over raw indices.

- **RuneLite GitHub References:** When implementing a specific feature, check how existing RuneLite or hub plugins do it:

- *Example:* You want to implement an agility course handler – check the RuneLite **Agility** plugin for how it detects course obstacles (they likely use object ID constants and maybe an `Obstacle` enum). You want to implement a timer – see how the **Gauntlet** plugin uses GameTick and overlay.
- *Menu Actions:* The built-in **Menu Entry Swapper** plugin shows how to modify or prefer certain menu options, which can hint at the indices of actions for items/objects.
- *Idle Detection:* RuneLite's **IdleNotifier** uses animation IDs and interaction states to decide when to alert [19] [20]. You can mimic that logic for determining when an action has actually stopped.

- *NPC Combat:* The **NPC Aggression Timer** plugin knows when you get out of combat; it might use `client.getLocalPlayer().getInteracting()` and ticks since last combat. We can glean patterns for resetting combat or deciding to re-attack.
- *Quest Helper:* While QuestHelper is aimed at guiding the player manually, its step logic and use of varbits is a goldmine for our automation. For instance, QuestHelper's `QuestStep` classes often have conditions like `if Varplayer X == 3 do this step`. We can use those same conditions but instead of highlighting an NPC, our code will actually perform the action on that NPC.

By grounding our implementation in these references, we ensure the behavior is aligned with what RuneLite expects and we reduce the chance of using a wrong function or approach.

Next Steps and Implementation Plan

Designing is great, but implementing step-by-step is crucial. Here's a suggested roadmap to build this modular system:

1. **Define Service Interfaces/Classes:** Start by creating the classes for each service (`InventoryService`, `MovementService`, etc.) in the core module. Define their public methods based on the functionality outlined above. Even if the methods are stubs initially, having the interface allows plugins to start depending on them. For instance, `InventoryService.dropAllExcept(int... itemIds)` or `MovementService.walkTo(WorldPoint point)`.

2. **Connect Services with RuneLite (core-rl):** In your RuneLite plugin (core-rl branch), set up dependency injection for the services. One approach is to make them singletons and manually inject the `Client` and `ClientThread` into them on startup. For example, your main plugin's `startUp()` could do:

   ```
   InventoryService.initialize(client, clientThread, configManager, etc.);
   MovementService.initialize(client, clientThread, ...);
   ```

   Alternatively, use Guice to bind them as singletons and annotate with `@Inject` where needed (RuneLite's plugin system might allow this via the Plugin constructor injection). Ensure each service has access to `Client` (for game state) and `ClientThread` (for actions), as well as the event bus if it needs to subscribe to events internally.

3. **Implement InventoryService (high priority):** This is one of the most universally needed services. Implement:

4. Methods to query inventory (using `client.getItemContainer(InventoryID.INVENTORY)`).
5. Methods to perform item actions. For now, you can implement dropping and using items. Use `invokeMenuAction` with appropriate parameters. Test dropping in a controlled scenario (maybe a small plugin command that triggers dropping one item) to ensure the right opcode and parameters. Adjust as needed by watching the game (or consult RuneLite menu entry creation in the source).

6. Use RuneLite's item and widget IDs for inventory slots. The inventory interface ID is constant (e.g., 149 for inventory interface, if I recall correctly, or use WidgetInfo). The `param0` and `param1` in `invokeMenuAction` correspond to widget IDs: typically `param1` is the inventory interface ID << 16 | item slot index, and `param0` is the item ID or other identifier. (This can be found by printing MenuEntry's fields when you manually drop an item to see what RuneLite uses.)
7. Provide utility like `isFull()` using container.getItems().length or count != 28 (account for nulls).

8. **Testing:** Run the client with this service and a sample plugin or command to drop items, move items, etc., to verify it works without errors.

9. **Implement MovementService basics:** Focus on a simple walking mechanism:

10. Perhaps use `WorldPoint` and the client's local pathfinder: RuneLite has a method `client.findPath()` (if available) or you might use something like `WorldPoint.calculatePath`. If not, you can do a crude step-by-step: get player's current location and click towards the target (for example, if target is far, click on the minimap halfway, etc.).
11. You can also integrate an existing path service (some bots use DaxWalker API via web – but that's external; for now keep it simple).
12. Implement run toggling: check `client.getEnergy()` each tick and if run is off and energy > threshold (e.g., 50%), then simulate clicking run orb (the run orb is a widget in the interface, which can be toggled via `MenuAction.WIDGET_TYPE_1` on that widget).

13. Test by commanding the player to walk to a nearby tile or a known coordinate and see if it works.

14. **ObjectInteractionService and NPCInteractionService:** Implement finding and clicking:

15. Leverage RuneLite's **queries** or scene graph. RuneLite's API has classes like `TileObjects` in some utility libraries (as seen in the CrossBoosting example, they imported `net.unethicalite.api.entities.TileObjects` which likely has methods to get objects by name/ID). You might write your own simple utility: iterate `for (Tile tile : client.getScene().getTiles()[playerPlane])` and check `tile.getGameObjects()` for matching ID. Or use the Query approach (RuneLite's script or Unethicalite's pre-built queries).
16. Once you have a `GameObject`, use its `getWorldLocation()` and `getId()`. For interacting:
    - The `identifier` in `invokeMenuAction` for objects is usually the object's ID, `param0` might be the object's world point packed (x, y, height), and `param1` might be the object's plane << 16 | some other info. Actually for objects, RuneLite uses a different mechanism (the object's sceneX, sceneY and id). It's easier: **RuneLite provides a method** `GameObjectInteract` via menu entries when you right-click. Possibly you can obtain a MenuEntry from the object by calling `client.createMenuEntry(...)`. However, for simplicity, using `client.invokeMenuAction` with coordinates might suffice.
    - You might need to experiment or search RuneLite code (e.g., how does the default "Xray" tool or developer tools do object examining? They also invoke actions).

17. Implement this carefully and test on a simple object like a tree or a door in-game.

18. **DialogueService:** Implement continue and option selection:

19. Use the `DialogOptionClicked` event if available (RuneLite might have an event when a dialogue option widget is clicked, but we can also simply drive it).
20. The continue option: an easy way is to watch for the chat interface group ID (e.g., 219 for chat) and whenever `client.getWidget(WidgetInfo.DIALOG_NPC_TEXT)` is non-null, call `invokeMenuAction("Continue", "", 0, MenuAction.WIDGET_CONTINUE.getId(), 0, 0)`. This simulates a spacebar press. You can throttle it by only doing once per tick or so to avoid spam clicking.
21. Options: If a dialogue option interface (WidgetID.DIALOG_OPTION_GROUP_ID) is open, the child widgets for options can be clicked. For example, option 1 might be WidgetInfo.DIALOG_OPTION1 which gives (group 219, child X). Invoke a `WIDGET_TYPE_1` or similar on it. (This may require looking up how MenuEntry is made for widget interactions – but since we have the widget ID, we can do `client.invokeMenuAction("Select", optionText, widgetID, MenuAction.WIDGET_FIRST_OPTION.getId(), 0, 0)` or use `MenuAction.WIDGET_CONTINUE` if it's the same mechanism.)

22. Verify by initiating a test dialogue (talk to an NPC) and see if the DialogueService can advance it automatically.

23. **CombatService and others:** Start with basic attack functionality:

24. Find NPC by id or name (similar to object finding, but via `client.getNpcs()` list).
25. Use `invokeMenuAction("Attack", npcName, npcId, MenuAction.NPC_FIRST_OPTION.getId(), npcIndex, 0)` to attack. Here `npcIndex` is the index in the local NPC array or the NPC's index, which you get from `NPC.getIndex()` in RuneLite API.
26. Test on a dummy monster to ensure it works.
27. Implement health/prayer checks using `client.getBoostedSkillLevel(Skill.HITPOINTS)` etc. Connect with InventoryService to consume food/potions when thresholds are met. Initially, you can log when it *would* eat, then later actually call the eat method.
28. Special attack: find the widget ID for spec (e.g., WidgetInfo.MINIMAP_SPEC_ORB or so) and invoke a widget action. Ensure to check Varplayer for spec percent (Varplayer 300 I think is spec energy).

29. This service can become complex, so implement incrementally, and possibly later integrate more advanced logic like prayer flicking or gear swaps.

30. **Tie into Plugins (Refactor existing plugins):** Once the core services have basic functionality, refactor one of your existing plugins (say woodcutting plugin) to use them:

31. The woodcutting plugin's job reduces to: on game tick, if not currently chopping and tree available, call `ObjectInteractionService.interact(TreeID, "Chop down")`; if inventory full, call `InventoryService.dropAllExcept({axeId})`.
32. All the lower-level details (ensuring player is idle, etc.) are handled within those service methods. For example, `dropAllExcept` could internally ensure the player isn't in the middle of an action before dropping to avoid interrupting.
33. You'll find the plugin code becomes much shorter and clearer. Do similar for fishing (e.g. interact with fishing spot, drop fish via InventoryService), etc.

34. Test these plugins thoroughly to iron out any issues in the services. It's likely you'll adjust the services as you discover edge cases (e.g., maybe need to wait a tick after an action before taking the next, etc.).

35. **Quest Plugin Framework:** With the core solid, outline how to implement a quest plugin easily:

36. Perhaps create a `QuestService` or simply a base `QuestPlugin` class that your individual quest plugins can extend. This base class could contain a list of `QuestStep` objects, and in `onGameTick` it checks the current step's condition and runs the step's action if not done.
37. Write a simple quest plugin for an easy quest (like **Cook's Assistant**) using this framework as a proof of concept. For Cook's Assistant:
    - Step1: If you don't have bucket of milk, egg, flour – go get them:
    - Sub-steps: go to cow pen, use bucket on dairy cow (milk) – use MovementService to walk, ObjectInteractionService to use bucket on cow, etc.
    - Then go to chicken farm, pick up egg (ObjectInteractionService to take item on ground).
    - Then go to mill, use wheat on mill etc. (This one is complex but good to test).
    - Step2: Once items acquired (condition: inventory has all items), talk to Cook in Lumbridge Castle to complete quest (DialogueService to handle conversation).
38. This is a non-trivial script but if your services are working, it mostly becomes a sequence of calls. You will likely find gaps (e.g., *"How to handle item on ground?"* – you might need a GroundItem handler: similar to ObjectInteractionService but for GroundItems using `client.getGroundItems()`).
39. Fill in any missing pieces (ground item picking, using ladders which involve scene loading – perhaps watch for `GameStateChanged` events when loading areas).

40. The result should demonstrate that following a wiki guide is mostly configuring the steps, not writing new low-level code.

41. **Review and Optimize:** Once everything is functional:

    - Clean up the structure (maybe group services in a package, ensure naming is consistent – e.g., use suffix *Service or* Handler uniformly).
    - Add plenty of comments and perhaps a wiki for your project explaining how to use each service for future contributors.
    - Incorporate feedback from RuneLite's conventions: e.g., if some of your services can be made to implement RuneLite's standard interfaces or utilize its utilities, do so.
    - Test edge cases: e.g., dropping items while in combat (should probably avoid), walking while an interface is open (maybe MovementService should check no dialog open, etc.). Each service can have safeguards for context (for instance, DialogueService might temporarily disable MovementService until dialogue is done, etc. – coordination between services).

Throughout this process, **continue to reference RuneLite's API and plugins** to verify you're using the right approach. If something is unclear (like the correct parameters for `invokeMenuAction` for a given action), search the RuneLite GitHub or OpenOSRS forks – chances are there's example code for it. By basing our implementation on known good code, we prevent mistakes.

Conclusion

By introducing a well-structured modular system of handlers, **HeistCore** will evolve into a powerful framework on which many plugins (skilling, bossing, PKing, questing) can be built with minimal effort. The **InventoryService** will take care of item management (from dropping logs to eating food), the **MovementService** will handle travel, **Interaction services** will abstract away the clicks on objects and NPCs, and specialized services like **DialogueService** and **CombatService** will cover complex interactions. All these will operate grounded in real game data – using animations, object IDs, varbits, etc., as signals – just like proper RuneLite plugins do.

The end result: if someone hands you an OSRS Wiki page for a quest, you could implement that quest's plugin by simply chaining together calls like `movementService.walkTo(bankArea); bankService.withdraw(ItemID.COINS, 30); movementService.walkTo(npcArea); npcService.talkTo(NpcID.SHOPKEEPER); dialogueService.continueDialogue();` and so on – the heavy lifting (pathfinding, clicking, waiting for outcomes) is handled behind the scenes. The plugin code remains clean and declarative.

By following the implementation plan and continually validating against RuneLite's APIs, you'll create a **beautifully structured, robust system** that meets your goals. It's a significant upfront effort, but once in place, you'll be able to develop new plugins or maintain existing ones with unparalleled ease and consistency. Remember to always test thoroughly and adjust using RuneLite's debugging tools and logs, and your HeistCore will become an incredibly powerful automation core for OldSchool RuneScape.

References:

- RuneLite API Constants for game elements like Object IDs and Animation IDs [16] [8] – used to avoid magic numbers in code.
- Example of checking player idle state and inventory contents in an OSRS plugin [13] – demonstrates the importance of waiting until the player is not busy before the next action.
- Inventory item usage via a high-level API (OpenOSRS example) [4] – similar concept to our InventoryService using one item on another.
- Guidance on using RuneLite's ClientThread to invoke actions safely on the game thread [14] – we apply this for all game-interaction calls to avoid freezing or concurrency issues.
- Quest state varbits/varplayers from RuneLite & QuestHelper [12] – shows how to fetch quest progress, enabling our quest plugins to make decisions based on in-game quest state rather than assumptions.

---

[1] Help with own plugin : r/RunescapeBotting
https://www.reddit.com/r/RunescapeBotting/comments/1j1n43t/help_with_own_plugin/

[2] [3] [6] [10] MenuAction (RuneLite API 1.12.6 API)
https://static.runelite.net/runelite-api/apidocs/net/runelite/api/MenuAction.html

[4] [5] [13] CrossBoostingPlugin.java · GitHub
https://gist.github.com/Magnusrn/113e7ce08ddd03441d392cc5c0a4adc6

7 8 9 11 12 16 17 18 RuneLite/OpenOSRS Code Snippets - Snippets - DreamBot - Runescape OSRS Botting
https://dreambot.org/forums/index.php?/topic/22572-runeliteopenosrs-code-snippets/

14 15 Development - Understanding Runelite's ClientThread : r/RunescapeBotting
https://www.reddit.com/r/RunescapeBotting/comments/13vk0q6/development_understanding_runelites_clientthread/

19 1.10.23 Release - RuneLite
https://runelite.net/blog/show/2024-02-24-1.10.23-Release/

20 1.4.15 Release - RuneLite
https://runelite.net/blog/show/2018-08-30-1.4.15-Release/