

**1. What Is Extract Method Refactoring?**

**Goal:** Quickly bring everyone on the same page.  
**Visuals:**

* Title: *“Refactoring — Changing the Code, Not the Behaviour”*
  + Simple “before/after” diagram of tangled vs. modular code.
  + Garage analogy (“same tools, better layout”).
* JetBrains survey graph (77 % refactor weekly; 40 % daily).
  + Message: “Developers refactor constantly — but often manually.”
* Java example from the report showing simple Extract Method (“before/after”).

👉 **Key takeaway to say:**

“Extract Method is the simplest refactor to explain, but in Rust, it’s one of the hardest to automate.”

**2. Why It’s Hard in Rust + How AoL and REM Tackled It**

**Goal:** Link difficulty → first attempt → motivation for your project.  
**Visuals:**

* **Slide 1 – “Rust’s Safety Model = Compiler as Bodyguard”**  
  Diagram: boxes for ownership, borrowing, lifetimes, arrows showing invalid move.  
  Simple labels: *‘Move’, ‘Immutable Borrow’, ‘Mutable Borrow’*.  
  Caption: “What keeps Rust safe also makes refactoring tricky.”
* **Slide 2 – “When a Naïve Extraction Fails”**  
  Side-by-side code: before / after, with compile-error annotation (“value borrowed after move”).
* **Slide 3 – “Adventure of a Lifetime (2023)”**  
  Use Figure 1 from your report (AoL algorithm).  
  2–3 bullets:
  + Encoded control flow using enums (Ok, Return, Break)
  + Used ownership analysis to infer &, &mut, or move
  + Verified via repeated compiler checks
* **Slide 4 – “REM: First Prototype”**  
  IntelliJ plugin screenshot or architecture sketch.  
  Two columns:
  + ✅ Breakthrough: handled lifetimes, control flow
  + ❌ Limitations: no async/generics, fragile integration, slow due to cargo check

👉 **Bridge line:**

“My work began where AoL and REM left off — rebuilding it from the ground up for modern Rust.”

**3. My Contributions: Expanding REM**

**Goal:** Show clear technical contributions.  
**Visuals:**

* **Slide 1 – “A New Core Architecture”**  
  Diagram: VS Code → CLI → IR Server → Extraction Engine  
  Bullets:
  + Incremental compilation via IR server
  + Async/await + generics + control-flow aware
  + Editor-agnostic CLI
* **Slide 2 – “Unified CLI and VS Code Extension”**  
  Screenshots or mock-ups:
  + CLI before/after example
  + Extension UI (highlight → refactor → preview)  
    Tagline: “Fast, portable, and lightweight — no IntelliJ required.”
* **Slide 3 – “Performance and Robustness”**  
  Simple bar graph: *Extraction time (REM Classic vs New)*  
  Notes:
  + 60+ real-world cases
  + Complete coverage of old REM
  + Feels instantaneous (< 2 s per extraction)

**4. Verification Toolchain: Proving It’s Correct**

**Goal:** Make formal verification approachable.  
**Visuals:**

* **Slide 1 – “Compilation ≠ Correctness”**  
  Cartoon: “It compiles!” → “But is it the same?”  
  Lead into motivation.
* **Slide 2 – “The Verification Pipeline”**  
  Use simplified Figure 3.1:  
  Rust → CHARON → AENEAS → Coq → ✅ Proof.  
  Add logos and arrows; label each with 1-line purpose.
* **Slide 3 – “What We Prove”**  
  Formula on screen: ∀x. f(x) = f′(x)  
  Bullet explanation: “For all valid inputs, original and refactored return the same.”  
  Mention: fail-early design = no false positives.

**5. Results and Evaluation**

**Goal:** Demonstrate effectiveness of both toolchains.  
**Visuals:**

* **Slide 1 – “Extraction Evaluation”**  
  Bar or stacked chart: success rate & average latency across 60 cases.  
  Caption: “Faster and more comprehensive than REM Classic or IDE extractors.”
* **Slide 2 – “Verification Evaluation”**  
  Table/plot (from Table 4.2):
  + 20 cases → all succeeded
  + 2–5 s typical, 30 s worst-case
  + Runs in background, acceptable for IDE use  
    Callout: “Academic-level assurance, practical turnaround.”

**6. Conclusions & Future Work**

**Goal:** Leave a polished finish.  
**Visuals:**

* **Slide 1 – “Key Takeaways”**  
  Icons grid:
  + 🧠 Smarter extraction (async/generics)
  + ⚡ Lightweight + portable CLI/extension
  + 🔒 Formal verification for trust
* **Slide 2 – “What’s Next”**  
  Roadmap arrows:
  + Integrate verifier into live IDE workflow
  + Broader coverage → unsafe / concurrency
  + Long-term: merge with Rust Analyzer
* Closing line on slide:

“From prototype to production-ready — bringing trustworthy refactoring to Rust developers.”

I need some help presenting the attached (WIP) research paper at a seminar.

The seminar is going to be approximately 10-12 minutes long.

I would like there to be 6 distinct sections to the seminar (although they don’t have to be / shouldn’t be the same length). We need to bear in mind that I am presenting to mostly engineering students not computer science students, so this cannot be a jargon heavy / obtuse presentation). Some aspects from the paper that I am looking to cover off on are below. I want you to help me structure the presentation, explain the major content on the slide groups, and also suggest what images / infographics / code excerpts should be on each slide.

* What is extract method as a concept
* Why is this tool / concept needed (I have a great research paper from Jetbrains I can include data from). It has stats etc showing how developers spend time refactoring manually. Also explain why this is so hard in Rust (due to lifetimes etc)
* Where this all originally stemmed from (being Adventure of a Lifetime and REM) – what REM did with lifetimes.
* What I have accomplished based on REM
  + A completely redone frontloaded extraction tool that is lightweight, fast, and capable of extracting advanced language features such as generics and asynchronous code along with controlflow awareness and rewriting. Relies on a custom IR server that implements incremental compilation to ensure its responsiveness even on very large codebases.
  + A VSCode Extension that serves as a proof of concept integration
  + A unified CLI interface that ties both my tools and contributions together with the improvements that I have made to REM into one tool that is both simple to use (and as the VSCode extension demonstrates) straightforward to implement into any editor (as opposed to the original REM which relied on a clunky relationship with a very specific version of IntelliJ IDEA)
* What the verification toolchain is and why it is needed. How it works (integrating CHARON and AENEAS into the CLI, and then using automatically generated Coq proofs to show that the extraction hasn’t changed the semantics of the code.
  + Fail early design ensures that no false positives should be returned (i.e. verification engine shouldn’t ever say code is correct when its not)
  + Good error reporting through the CLI can easily be propagated up to the IDE interface
  + The actual proof obligation we are trusting / generating against
  + Developer can choose to engage to verifier where necessary, not forced on as many cases won’t necessarily require it.
* Results for both the main toolchain and the verification toolchain
  + Main toolchain:
    - Analysed against over 60 real world cases. Compared with extraction tools like IntelliJ IDEA, Rust Rover, VSCode, etc
    - Complete coverage of all cases that the original REM toolchain could solve.
    - Faster and more comprehensive than the original REM toolchain. Fast enough that there is no appreciable delay when attempting to extract (although this could be a difference between VSCode and Intellij IDEA, with VSCode being a much more lightweight editor)
    - Any other results / concepts worth talking about here (noting the results section of the report isn’t written)
  + Verification
    - Evaluated against 20 cases, 10 real world and 10 examples. Returned a success in all cases
    - Takes between 2-5 seconds to evaluate the cases (although one very large project took approx 30 seconds).
      * Attributed to the time taken to compile the crate using CHARON to get the LLBC. At this stage there is no appreciable way to speed this up / use incremental compilation as CHARON needs access to the whole crate not just individual files. Future work as platform matures
      * Can still happen in the background as the developer continues to work so not the end of the world
    - Admittedly this is more of an academic tool, similar to where REM was when I started this project. However it is integrated into the rest of the extension and will definitely still have its uses!
* Conclusion and Future work

Ok The final stage that we need to work on this the results, evaluation, and conclusions drawn from those results. This is a critical part of the overall presentation.

From the assessment sheet, we need to do the following:

* The approach and methods are succinctly described, the results are clearly presented and analysed in a meaningful way, with appropriate conclusions drawn.

We have done the first half, but need to focus on the second (I am thinking for three slides, one direct evaluation / discussion of results for the extraction toolchain and verifiyer, and one to draw conclusions from the results).  
  
It is important to note that a decent chunk of these results haven’t been fully obtained yet, so the seminar will take some creative liberties to interpret what the results will be and talk about them as if they have been fully obtained.

For the main toolchain:

* We drew on EvanLi’s Github ranking tool, and from there were able to get the 20 largest rust projects. (Excluding the Rust compiler). These repositories included projects like Deno, Tauri, Zed and more. We then searched these repositories for examples of manual extract method refactoring, and recreated these examples using the new toolchain. Admittedly, we were specifically looking for examples that could showcase the features of our new toolchain (e.g. generics, asynchronous code, and control flow rewriting) Together we were able to obtain 40 such cases, showcasing the entire spectrum of the Rust language. We are currently sitting at 26 successful extraction cases of those 40, with another 4 hinging on minor bugs in output format.
* We also tested against the original 40 cases that the first iteration of REM was evaluated against. We have been able to achieve an equivalent level of coverage on these examples. Where the REM toolchain took on average 0.949 seconds, we were able to break this down into a query response time of < 0.01 seconds and then an average extraction time of < 0.5 seconds. Crucially this also includes the overhead of the VSCode extension – we are measuring real world performance where REM just measured the performance of their part of the toolchain (post IntelliJ extraction) !
* In the next week we will be evaluating the original REM against the new cases, alongside with attempting to evaluate either VSCode or IntelliJ against the new cases to get us a baseline to include in the report!

For the verification toolchain:

* Evaluated against 20 cases, 10 real world (but quite small) and 10 examples. Returned a success in all cases
* Takes between 2-5 seconds to evaluate the cases (although one very large project took approx 30 seconds).
* Attributed to the time taken to compile the crate using CHARON to get the LLBC. At this stage there is no appreciable way to speed this up / use incremental compilation as CHARON needs access to the whole crate not just individual files. Future work as platform matures. On average, CHARON takes ~80-90% of the verification time.
* Key point is that this can still happen in the background as the developer continues to work on the code so this is not a limitation of the tool as much as it would be if it was in the actual extraction
* Admittedly this is more of an academic tool, similar to where REM was when I started this project. However, it is integrated into the rest of the extension and will definitely still have its uses!
* Future work – having just presented the verifier at SPLASH in Singapore (last week), a big question raised by peers was being able to show the capacity / limitations of the verifier – as such this week we are going to focus extensively on attempting to find counter examples (within the scope of supported language features) that fail.
  + Feedback from conference: If you still have some time, would be nice to find examples where verification fails or at least show that you really tried hard to find them but didn't. This would include: recursive outer functions, refactoring within a loop, async / concurrency (this will fail at Aeneas stage right?). Also it would be important to state whether the equality depends on functional verification or is purely syntactic. That is, does the complexity of the outer function matter at all? Or it doesn't matter as syntactically after simplification the bodies are the same?