Design Rationale for REQ3: Farmer’s Descent into the Hollow Roots

Design Goals

1. **Extensibility** – support future tools (axe, hoe, watering can), new ore types, and additional reward mechanics.
2. **Separation of Concerns** – keep combat, farming, mining, and merchant logic in distinct modules.
3. **Maintainability** – minimise impact on existing game code (especially A2’s BuyWeaponAction), use clear abstractions.

Implementation A: BuyToolAction for Pickaxe Purchases

We introduce a new BuyToolAction rather than repurposing BuyWeaponAction.

Pros	Cons
Clearly separates weapon sales from tool sales	Adds one extra Action class
Facilitates selling any future tool (axe, hoe, watering can…)	Merchant code must register each new tool
Leaves existing BuyWeaponAction unchanged (preserves A2)	Slightly more boilerplate for new tool types

Example Scenario:
A new “Magic Hoe” tool is added in the future. We simply add:

- actions.add(new BuyToolAction(new MagicHoe(), 120));

in the merchant’s allowableActions. No changes to weapon logic are required.

- **Maintainability:** Tools and weapons remain decoupled; modifying tool-purchase logic never touches the weapon code.
- **Future Improvement:** Tool catalog could be data-driven to avoid hardcoding.

Implementation B: ShimmeringGlyph + MineAction

We represent mineable ore as a ShimmeringGlyph ground tile (char ⤧) with randomised durability (1–3), and encapsulate a MineAction to perform strikes.

Pros	Cons
Mining logic centralised in MineAction (stamina cost, direction)	Requires wiring two classes (Ground + Action)
ShimmeringGlyph handles its own durability and reward logic (SRP)	Durability hardcoded at instantiation; not data-driven
Clear extension point: subclass ShimmeringGlyph for new ore behaviors	Direct string-based checks for “Pickaxe” brittle

Example Scenario:
Designers decide some ores take 5 hits instead of 1–3. They subclass:

- public class IronGlyph extends ShimmeringGlyph {
- public IronGlyph() { super(5); }
- }

No changes to MineAction or other classes.

- **Maintainability:** Mining expense (stamina) and drop logic live in one place.
- **Future Improvement:** Extract reward probabilities into a configurable strategy class rather than inline nextInt logic.

Implementation C: Reward Randomisation in breakOre()

We simplified to a 10-point roll:

- **1/10** spawn fossil (OmenSheep/SpiritGoat)
- **4/10** give CharmPiece
- **5/10** give OrePiece

Pros	Cons
Easy to read and tune probability ratio	Inline logic may grow unwieldy
All three outcomes handled in one place	Harder to unit-test without refactoring

Example Scenario:
To change fossil rarity to 5%, adjust roll == 0 instead of roll == 1. No new classes needed.

- **Maintainability:** Single method controls reward distribution.
- **Future Improvement:** Move to an enum-based strategy or external config file for probabilities.

Implementation D: GlowingSinkhole + TeleportAction

We use our own TeleportAction (written in A3 REQ1) in the game code to link two maps via a GlowingSinkhole ground tile.

Pros	Cons
Reuses existing game code (TeleportAction)	Sinkhole locations still hard-coded in Application
Keeps teleport logic in one small class (GlowingSinkhole)	Cannot dynamically open/close sinkholes at runtime
Clear separation: ground only offers the action, action performs move	Requires map-to-map wiring in Application

Example Scenario:
A designer wants a one-way exit: subclass GlowingSinkhole to remove the return link.

- **Maintainability:** Teleport logic remains consistent; only ground placement changes.
- **Future Improvement:** Introduce a teleport registry or data file to configure sinkhole pairs without editing Application.

Implementation E: SellOreAction at MerchantKale

We add a SellOreAction so that when adjacent to MerchantKale, each OrePiece in the Farmer's inventory can be sold for 10 runes.

Pros	**Cons**
Mirrors BuyActions, consistent UX	Merchant must scan inventory; small performance cost
Keeps selling logic co-located with the merchant class	Fixed price; no volume discounts or dynamic pricing
Adds only one new Action class in game code	Cannot sell multiple at once without repeated selection

Example Scenario:
Player collects ten OrePieces and sells each in turn via the action menu. No code changes needed for batch selling.

- **Maintainability:** Sell logic lives entirely in SellOreAction.
- **Future Improvement:** Add SellAllOreAction to clear inventory in one go.

Alternatives & Why They Were Rejected

1. **Merging tool and weapon purchases**
    - *Rejected:* Breaks separation of domains and A2 weapon requirements.

2. **Data-driven map connections**
    - *Rejected for v1:* Adds complexity; simple hard-coding suffices for current scope.

Trade-Offs & Future Directions

- **String-based item checks** are brittle—future work could introduce a Tool capability to actors/items.
- **Hard-coded sinkhole placement** could be improved via a config file for map designers.
- **Reward logic** may outgrow its single method; a refactored RewardStrategy pattern would enhance testability.

Conclusion:

The REQ3 implementation cleanly extends our game code by adding minimal, focused classes (BuyToolAction, ShimmeringGlyph, MineAction, SellOreAction, GlowingSinkhole) without touching engine internals. Each feature adheres to SOLID tenets locally, and the overall design supports foreseeable future tools, terrains, and reward mechanisms.