

Design Document: Loot Modifier System

1. Objective

To implement a flexible and extensible loot modifier system where items (weapons, armor, etc.) have a random chance to be generated with one or more magical properties. These properties will provide statistical bonuses, elemental effects, or other unique benefits, and will be visually indicated to the player.

2. Core Data Structures

We will introduce two new data structures to define what a modifier is.

ModifierType.java (New Enum)

This enum will act as a "master list" of all possible modifier *effects* in the game. This makes it easy for other systems (like combat) to query an item for specific bonuses.

Java

```
// in core/src/main/java/com/bpm/minotaur/gamedata/
public enum ModifierType {
    // Basic Stats
    BONUS_DAMAGE,      // +N to base damage (War & Spiritual)
    BONUS_DEFENSE,     // +N to base armor (Armor & Rings)
    BONUS_WAR_STRENGTH, // +N to player's max War Strength
    BONUS_SPIRITUAL_STRENGTH, // +N to player's max Spiritual Strength

    // Elemental/Effect Damage (Weapons)
    ADD_FIRE_DAMAGE,
    ADD_ICE_DAMAGE,
    ADD_POISON_DAMAGE,
    ADD_BLEED_DAMAGE,

    // Elemental/Effect Resistance (Armor/Rings)
    RESIST_FIRE,
    RESIST_ICE,
    RESIST_POISON,
    RESIST_BLEED,
    RESIST_DISEASE,
    RESIST_DARK,
    RESIST_LIGHT,

    // Material (Tags, value is 0 or 1)
```

```

MATERIAL_STEEL,
MATERIAL_MITHRIL,
MATERIAL_TITANIUM,

// Bane (vs. Specific Monster Types)
BANE_UNDEAD,      // e.g., vs. GHOUL, SKELETON
BANE_MONSTER,     // e.g., vs. GIANT_ANT, SPIDER
BANE_HUMAN,       // e.g., vs. DWARF
BANE_MYTHICAL,    // e.g., vs. DRAGON, MINOTAUR
}

```

ItemModifier.java (New Class)

This will be a simple data container object that gets attached to an **Item**. It is designed to be easily serialized to JSON.

```

Java
// in core/src/main/java/com/bpm/minotaur/gamedata/
public class ItemModifier {
    public ModifierType type;
    public int value;
    public String displayName; // e.g., "Fiery" (+5 Fire Dmg), "+2", "of Ice Warding"

    // No-arg constructor for JSON serialization
    public ItemModifier() {}

    public ItemModifier(ModifierType type, int value, String displayName) {
        this.type = type;
        this.value = value;
        this.displayName = displayName;
    }
}

```

DamageType.java (New Enum)

To make resistances work, attacks must have a type.

```

Java
// in core/src/main/java/com/bpm/minotaur/gamedata/
public enum DamageType {
    PHYSICAL,
    SPIRITUAL,
    FIRE,
}

```

```

    ICE,
    POISON,
    BLEED,
    DISEASE,
    DARK,
    LIGHT
}

```

3. Data Model Changes (Existing Classes)

Item.java

The `Item` class will be modified to *have* a list of modifiers.

- Add field: `private List<ItemModifier> modifiers = new ArrayList<>();`
- Add methods:
 - `public void addModifier(ItemModifier modifier) { this.modifiers.add(modifier); }`
 - `public List<ItemModifier> getModifiers() { return this.modifiers; }`
 - `public boolean isModified() { return !this.modifiers.isEmpty(); }`
 - `public String getDisplayName():` This method will be created/updated to build a name from the item's base name and its modifiers (e.g., "Fiery Bow of Slaying").

ChunkData.java

To ensure modified items are saved and loaded, `ChunkData.ItemData` must also store them.

- In `ItemData` nested class: Add `public List<ItemModifier> modifiers;`
- In `ItemData(Item item)` constructor: Add `this.modifiers = new ArrayList<>(item.getModifiers());`
- In `buildMaze()`: When reconstructing an `Item` from `ItemData`, loop through `data.modifiers` and call `item.addModifier()` for each one.

Monster.java

To support `BANE_` modifiers, monsters need a "family" or type.

- Add field: `private MonsterFamily family;` (This would be a new enum, e.g., `UNDEAD`, `BEAST`, `HUMANOID`, `MYTHICAL`).

- In `Monster` constructor: `switch(type) { ... case SKELETON: this.family = MonsterFamily.UNDEAD; ... }`
- Add getter: `public MonsterFamily getFamily() { return this.family; }`

4. Spawning & Generation Logic

This is where modifiers are rolled and added to items.

`LootTable.java` (New Class)

This class will be a static data store, similar to `SpawnData.java`, defining *what* modifiers can spawn, *when*, and *how powerful* they can be.

Java

```
// in core/src/main/java/com/bpm/minotaur/gamedata/
public class LootTable {
    // Defines a potential modifier roll
    public static class ModInfo {
        public ModifierType type;
        public int minLevel; // Integrates with TIER system
        public int maxLevel;
        public int minBonus;
        public int maxBonus;
        public String displayName; // e.g., "Fiery", "of Brawn"
        public Item.ItemCategory category; // e.g., WAR_WEAPON, ARMOR

        public ModInfo(ModifierType type, int min, int max, int minB, int maxB, String name,
            Item.ItemCategory cat) { ... }
    }

    public static final List<ModInfo> MODIFIER_POOL = Arrays.asList(
        // TIER 1 (Levels 1-5)
        new ModInfo(ModifierType.BONUS_DAMAGE, 1, 10, 1, 1, "+1",
            Item.ItemCategory.WAR_WEAPON),
        new ModInfo(ModifierType.BONUS_DEFENSE, 1, 10, 1, 1, "+1",
            Item.ItemCategory.ARMOR),
        new ModInfo(ModifierType.RESIST_FIRE, 3, 10, 1, 2, "Warm",
            Item.ItemCategory.ARMOR),

        // TIER 2 (Levels 6-12)
        new ModInfo(ModifierType.BONUS_DAMAGE, 6, 15, 2, 3, "+2",
            Item.ItemCategory.WAR_WEAPON),
        new ModInfo(ModifierType.ADD_FIRE_DAMAGE, 7, 15, 1, 3, "Fiery",
            Item.ItemCategory.WAR_WEAPON),
```

```

        new ModInfo(ModifierType.BANE_UNDEAD, 8, 20, 3, 5, "Disrupting",
Item.ItemCategory.WAR_WEAPON),
        // ... etc.
    );
}

```

SpawnManager.java

This class will be updated to use the `LootTable`.

- In `spawnRegularItem()` and `spawnContainer()` (for loot inside), after the `Item` is created:
 1. `attemptToModifyItem(item, this.level, item.getItemColor());`
- Create new private method: `private void attemptToModifyItem(Item item, int level, ItemColor color):`
 1. **Calculate Chance:** Base chance (e.g., 10%) + bonus from `ItemColor`. The `color.getMultiplier()` can directly influence this.
 - `float spawnChance = 0.1f + ((color.getMultiplier() - 1.0f) * 0.1f);`
 2. **Roll for Modification:** `if (random.nextFloat() > spawnChance) return; // No modification`
 3. **Find Valid Modifiers:** Filter `LootTable.MODIFIER_POOL` based on:
 - `mod.minLevel <= level && mod.maxLevel >= level` (TIER system)
 - `mod.category == item.getCategory()` (e.g., no `BONUS_DAMAGE` on a `POTION`)
 4. **Add Modifier:**
 - Pick one random valid `ModInfo` from the filtered list.
 - Roll the bonus: `int value = random.nextInt(mod.maxBonus - mod.minBonus + 1) + mod.minBonus;`
 - Create and add the modifier: `item.addModifier(new ItemModifier(mod.type, value, mod.displayName));`
 5. **Roll for Additional Modifiers:** Implement a diminishing returns roll (e.g., 25% chance for a *second* modifier, 10% for a *third*).

5. Combat Integration

Systems that use items must be taught to read modifier lists.

CombatManager.java

- `playerAttack()`:
 1. Get modifiers: `List<ItemModifier> mods = weapon.getModifiers();`
 2. Iterate `mods` and sum up bonuses *before* dealing damage.
 3. `int bonusDamage = 0; int fireDamage = 0; int baneDamage = 0;`
 4. `for (ItemModifier mod : mods) { switch (mod.type) { ... } }`
 5. `case BONUS_DAMAGE: bonusDamage += mod.value; break;`
 6. `case ADD_FIRE_DAMAGE: fireDamage += mod.value; break;`
 7. `case BANE_UNDEAD: if (monster.getFamily() == MonsterFamily.UNDEAD) baneDamage += mod.value; break;`
 8. Call `monster.takeDamage(..., DamageType.PHYSICAL)` for base damage and `monster.takeDamage(..., DamageType.FIRE)` for elemental.
- `monsterAttack()`:
 1. When a monster attacks, it must specify a `DamageType`.
 2. Example: `player.takeDamage(damage, DamageType.PHYSICAL);`
 3. Example (Dragon): `player.takeSpiritualDamage(damage, DamageType.FIRE);`

Player.java

- `takeDamage(int amount, DamageType type)` and `takeSpiritualDamage(int amount, DamageType type)`:
 1. These methods must be updated to accept `DamageType`.
 2. `int elementalResist = getElementalResist(type);`
 3. `int finalDamage = Math.max(0, amount - damageReduction - elementalResist);`
- Create new private helper: `private int getElementalResist(DamageType type)`:
 1. This method will iterate through `wornHelmet`, `wornShield`, `wornBreastplate`, `wornRing`, etc.
 2. For each *equipped* item, it will iterate `item.getModifiers()`.
 3. It will look for a matching `RESIST_` type (e.g., if `type == DamageType.FIRE`, it looks for `ModifierType.RESIST_FIRE`) and sum the `value`.
 4. Returns the total resistance.

6. Rendering (Visual Feedback)

Modified items must "glow".

EntityRenderer.java

- `drawItemTexture()` (Modern Mode):
 1. `if (item.isModified()) { ... }`
 2. Before drawing the item, draw the *same texture* again, but slightly larger and tinted gold/yellow.
 3. `spriteBatch.setColor(1.0f, 0.9f, 0.2f, 0.4f); // Translucent Gold`
 4. `spriteBatch.draw(texture, stripe - 2, drawY - 2, 1 + 4, spriteHeight + 4, ...); // Draw slightly bigger`
 5. `spriteBatch.setColor(Color.WHITE); // Reset color`
 6. Draw the normal item sprite on top.
- `drawAsciiSprite()` (Retro Mode):
 1. `if (entity instanceof Item && ((Item)entity).isModified()) { ... }`
 2. Before the main pixel loop, draw the "glow" pixels.
 3. Iterate the sprite data (`px, py`): if `spriteData[py].charAt(px) == '#'`:
 - `shapeRenderer.setColor(Color.GOLD);`
 - Draw 1-pixel rects offset from the main pixel: `rect(currentX+1, currentY, ...), rect(currentX-1, currentY, ...), rect(currentX, currentY+1, ...), rect(currentX, currentY-1, ...).`
 4. After the glow pass, draw the main sprite *on top* with its normal color (`shapeRenderer.setColor(item.getColor());`).

Hud.java

- The `Hud` class (which draws the 2D inventory) must be modified.
- When it draws the `Item` sprites in the inventory slots, it will perform a similar check: `if (item.isModified()) { ... }`.
- Before drawing the item's sprite, it will draw a `Texture` (or a `ShapeRenderer` rect) behind it, tinted gold, to create a "glow" effect in the UI slot.

7. Summary of New Files

1. `core/src/main/java/com/bpm/minotaur/gamedata/ItemModifier.java`
 2. `core/src/main/java/com/bpm/minotaur/gamedata/ModifierType.java`
 3. `core/src/main/java/com/bpm/minotaur/gamedata/DamageType.java`
 4. `core/src/main/java/com/bpm/minotaur/gamedata/MonsterFamily.java` (Enum: UNDEAD, BEAST, etc.)
 5. `core/src/main/java/com/bpm/minotaur/gamedata/LootTable.java`
-

This design provides a flexible, data-driven foundation. We can add new `ModifierType` enums and `LootTable.ModInfo` entries at any time without having to change the core combat or rendering logic.