battle.py

```
"""Battle engine: handles all combat calculations and the battle loop."""
from __future__ import annotations
import random
import math
from typing import List, Optional, Tuple, Dict
from data_types import *
from character import Character
from monster_unit import MonsterUnit
from monsters_db import MonsterTemplate
from skills_db import SKILL_DB
from items_db import ITEM_DB


def _clear():
    import os; os.system('clear' if os.name == 'posix' else 'cls')


def calculate_physical_damage(attacker_patk: int, target_pdef: int,
                              power: float = 1.0, crit: bool = False,
                              element_bonus: float = 1.0,
                              weakness_bonus: float = 1.0,
                              break_bonus: float = 1.0) -> int:
    base = max(1, attacker_patk - target_pdef // 2)
    dmg = int(base * power * element_bonus * weakness_bonus * break_bonus)
    if crit:
        dmg = int(dmg * 1.5)
    dmg = max(1, int(dmg * random.uniform(0.9, 1.1)))
    return dmg


def calculate_magical_damage(attacker_matk: int, target_mdef: int,
                             power: float = 1.0, crit: bool = False,
                             element_bonus: float = 1.0,
                             weakness_bonus: float = 1.0,
                             break_bonus: float = 1.0) -> int:
```

```
    base = max(1, attacker_matk - target_mdef // 3)
    dmg = int(base * power * element_bonus * weakness_bonus * break_bonus)
    if crit:
        dmg = int(dmg * 1.5)
    dmg = max(1, int(dmg * random.uniform(0.9, 1.1)))
    return dmg


def calculate_heal(healer_mdef: int, power: float = 1.0) -> int:
    return max(1, int(healer_mdef * 2.5 * power * random.uniform(0.9, 1.1)))


def check_hit(attacker_acc: float, target_eva: float) -> bool:
    hit_chance = min(0.99, max(0.01, attacker_acc - target_eva))
    return random.random() < hit_chance


def check_crit(crit_rate: float) -> bool:
    return random.random() < crit_rate


class Battle:
    def __init__(self, party: List[Character], monsters: List[MonsterUnit]):
        self.party = party
        self.monsters = monsters
        self.turn = 0
        self.battle_log: List[str] = []
        self._defending: set = set()   # character names defending this round

    # ── Display helpers ──────────────────────────────────────────
    def display_battle_state(self):
        print("\n" + "="*65)
        print(" BATTLE STATUS")
        print("="*65)
        print("  ENEMIES:")
        for i, mu in enumerate(self.monsters):
            if mu.is_dead:
                print(f"  [{i+1}] {mu.instance_name} - ☠ DEFEATED")
            else:
                broken = " ⚡BREAK!" if mu.is_broken else ""
```

```
            print(f"  [{i+1}] {mu.instance_name} Lv{mu.template.level}{broken}")
            print(f"       HP: {mu.hp_bar()}")
            print(f"       Shield: {mu.shield_bar()}{mu.status_str()}")

    print()
    print("  ALLIES:")
    for i, ch in enumerate(self.party):
        if ch.is_dead:
            print(f"  [{i+1}] {ch.name} [DEAD]")
        elif ch.is_ko:
            print(f"  [{i+1}] {ch.name} [KO - {3 - ch.ko_turns} turns left]")
        else:
            print(f"  [{i+1}] {ch.name} Lv{ch.level} {ch.job.value}{ch.status_str()}")
            print(f"       HP: {ch.hp_bar()} | MP: {ch.mp_bar()}")
    print("="*65)

def display_turn_order(self, order: List):
    units = []
    for u in order:
        if isinstance(u, Character):
            units.append(f"{u.name}({u.spd})")
        elif isinstance(u, MonsterUnit):
            units.append(f"{u.instance_name}({u.spd})")
    print(f"\n  Turn Order: {' → '.join(units)}")

def _log(self, msg: str):
    self.battle_log.append(msg)
    print(msg)

# ── Turn order ──────────────────────────────────────────────────
def get_turn_order(self) -> List:
    units = []
    for ch in self.party:
        if not ch.is_ko and not ch.is_dead:
            units.append(ch)
    for mu in self.monsters:
        if not mu.is_dead:
            units.append(mu)
    # Sort by speed descending; defenders get priority
    units.sort(key=lambda u: (
```

```
        1 if (isinstance(u, Character) and u.name in self._defending) else 0,
        u.spd
    ), reverse=True)
    return units

# ── Main battle loop ──────────────────────────────────────────────
def run(self) -> bool:
    """Returns True if party wins."""
    print("\n" + "★"*65)
    print("  ★  BATTLE START!  ★")
    print("★"*65)
    input("  [Press Enter]")

    while True:
        self.turn += 1
        self._defending.clear()
        print(f"\n{'─'*65}")
        print(f"  ── TURN {self.turn} ──")
        self.display_battle_state()

        order = self.get_turn_order()
        self.display_turn_order(order)

        for unit in order:
            if self._check_battle_end():
                break
            if isinstance(unit, Character):
                if unit.is_ko or unit.is_dead:
                    continue
                self._player_turn(unit)
            elif isinstance(unit, MonsterUnit):
                if unit.is_dead:
                    continue
                self._monster_turn(unit)

        # End of round: status ticks, KO timers, break recovery
        self._end_of_round()

        if self._check_battle_end():
            break
```

```
        return self._is_victory()

    def _check_battle_end(self) -> bool:
        all_monsters_dead = all(mu.is_dead for mu in self.monsters)
        all_party_out = all(ch.is_ko or ch.is_dead for ch in self.party)
        return all_monsters_dead or all_party_out

    def _is_victory(self) -> bool:
        return all(mu.is_dead for mu in self.monsters)

    # ── Player turn ──────────────────────────────────────────────
    def _player_turn(self, ch: Character):
        print(f"\n  ── {ch.name}'s Turn ({ch.job.value}) ──")

        if ch.is_incapacitated():
            incap_se = [se for se in ch.status_effects
                        if se.effect_type in {StatusEffectType.SLEEP, StatusEffectType.STUN,
                                              StatusEffectType.FREEZE,
StatusEffectType.PARALYZE,
                                              StatusEffectType.PETRIFY}]
            if incap_se:
                self._log(f"  {ch.name} is {incap_se[0].effect_type.value}! Can't act.")
                # Wake up chance for sleep
                if incap_se[0].effect_type == StatusEffectType.SLEEP:
                    if random.random() < 0.25:
                        ch.remove_status(StatusEffectType.SLEEP)
                        self._log(f"  {ch.name} woke up!")
            return

        while True:
            print(f"\n  Actions: [1] Attack  [2] Skill  [3] Item  [4] Defend")
            choice = input("  > ").strip()

            if choice == "1":
                target = self._pick_enemy_target()
                if target:
                    self._do_basic_attack(ch, target)
                    break
            elif choice == "2":
```

```
            if not ch.can_use_skills():
                print("  Skills are sealed!")
                continue
            skill_id = self._pick_skill(ch)
            if skill_id is None:
                continue
            skill = SKILL_DB[skill_id]
            if not ch.can_use_magic() and skill.skill_type == SkillType.MAGICAL:
                print("  Cannot use magic (Silenced)!")
                continue
            if ch.current_mp < skill.mp_cost:
                print(f"  Not enough MP! (Need {skill.mp_cost}, have {ch.current_mp})")
                continue
            self._do_skill(ch, skill)
            break
        elif choice == "3":
            if not ch.can_use_items():
                print("  Items are sealed!")
                continue
            used = self._use_item_menu(ch)
            if used:
                break
        elif choice == "4":
            self._do_defend(ch)
            break
        else:
            print("  Invalid choice.")

def _pick_enemy_target(self) -> Optional[MonsterUnit]:
    alive = [mu for mu in self.monsters if not mu.is_dead]
    if not alive:
        return None
    if len(alive) == 1:
        return alive[0]
    print("  Choose target:")
    for i, mu in enumerate(alive):
        broken = " ⚡BREAK" if mu.is_broken else ""
        print(f"    [{i+1}] {mu.instance_name} HP:{mu.current_hp}/{mu.max_hp}{broken}")
    while True:
        try:
```

```
            idx = int(input("  > ")) - 1
            if 0 <= idx < len(alive):
                return alive[idx]
        except ValueError:
            pass
        print("  Invalid choice.")

    def _pick_ally_target(self, include_ko=False) -> Optional[Character]:
        if include_ko:
            valid = [ch for ch in self.party if not ch.is_dead]
        else:
            valid = [ch for ch in self.party if not ch.is_ko and not ch.is_dead]
        if not valid:
            return None
        if len(valid) == 1:
            return valid[0]
        print("  Choose ally:")
        for i, ch in enumerate(valid):
            status = "KO" if ch.is_ko else f"HP:{ch.current_hp}/{ch.max_hp}"
            print(f"    [{i+1}] {ch.name} ({status})")
        while True:
            try:
                idx = int(input("  > ")) - 1
                if 0 <= idx < len(valid):
                    return valid[idx]
            except ValueError:
                pass
            print("  Invalid choice.")

    def _pick_skill(self, ch: Character) -> Optional[int]:
        if not ch.unlocked_skills:
            print("  No skills available!")
            return None
        print("  Choose skill (0=back):")
        for i, sid in enumerate(ch.unlocked_skills):
            if sid in SKILL_DB:
                sk = SKILL_DB[sid]
                print(f"    [{i+1}] {sk.name} "
                      f"[{sk.tier.value}|{sk.skill_type.value}|{sk.element.value}] "
                      f"MP:{sk.mp_cost} PWR:{sk.power} ACC:{int(sk.accuracy*100)}%")
```

```
        while True:
            raw = input("  > ").strip()
            if raw == "0":
                return None
            try:
                idx = int(raw) - 1
                if 0 <= idx < len(ch.unlocked_skills):
                    return ch.unlocked_skills[idx]
            except ValueError:
                pass
            print("  Invalid choice.")

    def _use_item_menu(self, ch: Character) -> bool:
        """Returns True if an item was successfully used."""
        from game_state import GameState
        # Items accessed via global inventory - we'll use a simplified approach
        # The GameState will be passed in during actual game run
        print("  (Item system: handled by game state)")
        return False  # placeholder - overridden in actual game

    def _do_basic_attack(self, ch: Character, target: MonsterUnit):
        # Hit check
        hit = check_hit(ch.acc, target.eva)
        if not hit:
            self._log(f"  {ch.name} attacks {target.instance_name}... MISS!")
            return

        is_crit = check_crit(ch.crit)

        # Weakness check
        is_weak = target.hit_weakness(ch.weapon, ch.element)
        weakness_bonus = 1.3 if is_weak else 1.0
        break_bonus = 1.5 if target.is_broken else 1.0

        dmg = calculate_physical_damage(
            ch.patk, target.pdef, power=1.0,
            crit=is_crit, weakness_bonus=weakness_bonus, break_bonus=break_bonus
        )

        # Defending reduction
```

```
        if ch.name in self._defending:
            dmg = int(dmg * 0.7)

        actual = target.take_damage(dmg)

        crit_str = " CRITICAL!" if is_crit else ""
        weak_str = " □WEAKNESS□" if is_weak else ""
        self._log(f"  {ch.name} attacks {target.instance_name}!{crit_str}{weak_str}")
        self._log(f"    Dealt {actual} damage!")

        # Shield break
        if is_weak:
            broke = target.reduce_shield(1)
            if broke:
                self._log(f"  ⚡ {target.instance_name} is BROKEN! All defenses down for 1
turn!")
            else:
                self._log(f"  Shield: {target.shield_bar()} ({target.shield_points}
remaining)")

        # Counter
        if target.has_status(StatusEffectType.COUNTER) and not target.is_dead:
            cdmg = calculate_physical_damage(target.patk, ch.pdef, power=0.5)
            ch.take_damage(cdmg)
            self._log(f"  {target.instance_name} COUNTER! {ch.name} takes {cdmg} damage!")

    def _do_skill(self, ch: Character, skill: Skill):
        ch.current_mp -= skill.mp_cost

        # Determine targets
        targets_enemies = []
        targets_allies = []
        if skill.target == SkillTarget.SINGLE_ENEMY:
            t = self._pick_enemy_target()
            if t: targets_enemies = [t]
        elif skill.target == SkillTarget.ALL_ENEMIES:
            targets_enemies = [m for m in self.monsters if not m.is_dead]
        elif skill.target == SkillTarget.RANDOM_ENEMY:
            alive = [m for m in self.monsters if not m.is_dead]
            if alive:
```

```
                targets_enemies = random.choices(alive, k=skill.hits)
        elif skill.target == SkillTarget.SINGLE_ALLY:
            t = self._pick_ally_target(include_ko=skill.skill_type == SkillType.HEAL)
            if t: targets_allies = [t]
        elif skill.target == SkillTarget.ALL_ALLIES:
            targets_allies = [c for c in self.party if not c.is_dead]
        elif skill.target == SkillTarget.SELF:
            targets_allies = [ch]

        elem_name = f"[{skill.element.value}]" if skill.element != Element.NONE else ""
        self._log(f"\n  ✦ {ch.name} uses {skill.name}!{elem_name}")

        # Heal skills
        if skill.skill_type == SkillType.HEAL:
            for target in targets_allies:
                if skill.mp_cost == 20 and skill.id == 46:  # Resurrection special case
                    if target.is_ko:
                        target.revive(100)
                        self._log(f"    {target.name} is REVIVED with full HP!")
                    else:
                        self._log(f"    {target.name} is already standing.")
                else:
                    heal_amt = calculate_heal(ch.mdef, skill.power)
                    actual = target.heal(heal_amt)
                    self._log(f"    {target.name} recovers {actual} HP!")
            return

        # Buff skills
        if skill.skill_type == SkillType.BUFF:
            if skill.effect:
                for target in targets_allies:
                    se = StatusEffect(skill.effect.status, skill.effect.duration,
skill.effect.stat_modifier)
                    target.add_status(se)
                    self._log(f"    {target.name} gains {skill.effect.status.value}!")
            return

        # Debuff skills
        if skill.skill_type == SkillType.DEBUFF:
            for target in targets_enemies:
```

```
                if skill.effect and skill.effect.status:
                    chance = skill.effect.chance if skill.effect.chance > 0 else 0.8
                    if random.random() < chance:
                        se = StatusEffect(skill.effect.status, skill.effect.duration)
                        target.add_status(se)
                        self._log(f"    {target.instance_name} is afflicted with
{skill.effect.status.value}!")
                    else:
                        self._log(f"    {target.instance_name} resists!")
            return

        # Damage skills (physical / magical / special)
        num_hits = skill.hits if skill.target != SkillTarget.RANDOM_ENEMY else 1
        raw_targets = targets_enemies

        if skill.target == SkillTarget.RANDOM_ENEMY:
            alive = [m for m in self.monsters if not m.is_dead]
            raw_targets = []
            for _ in range(skill.hits):
                if alive: raw_targets.append(random.choice(alive))

        for target in raw_targets:
            for hit_n in range(num_hits if skill.target != SkillTarget.RANDOM_ENEMY else 1):
                # Hit check
                if not check_hit(ch.acc * skill.accuracy, target.eva):
                    self._log(f"    Hit {hit_n+1}: MISS on {target.instance_name}!")
                    continue

                is_crit = check_crit(ch.crit)
                is_weak = target.hit_weakness(
                    ch.weapon if skill.skill_type == SkillType.PHYSICAL else None,
                    skill.element if skill.element != Element.NONE else ch.element
                )
                weakness_bonus = 1.3 if is_weak else 1.0
                break_bonus = 1.5 if target.is_broken else 1.0
                # Weakness mark doubles weakness
                if target.has_status(StatusEffectType.WEAKNESS_MARK) and is_weak:
                    weakness_bonus *= 1.3

                if skill.skill_type == SkillType.PHYSICAL:
```

```
                    dmg = calculate_physical_damage(
                        ch.patk, target.pdef, skill.power,
                        is_crit, weakness_bonus=weakness_bonus, break_bonus=break_bonus)
                else:
                    # Magic boost
                    matk = ch.matk
                    if ch.has_status(StatusEffectType.MAGIC_BOOST):
                        matk = int(matk * 1.4)
                    # Reflect check
                    if target.has_status(StatusEffectType.REFLECT):
                        self._log(f"    {target.instance_name} REFLECTS the spell!")
                        friendly = [c for c in self.party if not c.is_ko and not c.is_dead]
                        if friendly:
                            rf_target = random.choice(friendly)
                            rdmg = calculate_magical_damage(matk, rf_target.mdef, skill.power)
                            rf_target.take_damage(rdmg)
                            self._log(f"    Reflected! {rf_target.name} takes {rdmg} damage!")
                        continue
                    dmg = calculate_magical_damage(
                        matk, target.mdef, skill.power,
                        is_crit, weakness_bonus=weakness_bonus, break_bonus=break_bonus)

                actual = target.take_damage(dmg)
                crit_str = " CRITICAL!" if is_crit else ""
                weak_str = " □WEAKNESS□" if is_weak else ""
                self._log(f"    {target.instance_name} takes {actual}
damage!{crit_str}{weak_str}")

                # Shield damage for weakness hits
                if is_weak:
                    broke = target.reduce_shield(1)
                    if broke:
                        self._log(f"  ⚡ {target.instance_name} is BROKEN!")
                    else:
                        self._log(f"  Shield: {target.shield_bar()}")

                # Apply effect
                if skill.effect and skill.effect.status and not target.is_dead:
                    chance = skill.effect.chance if skill.effect.chance > 0 else 0.5
                    if random.random() < chance:
```

```
                        se = StatusEffect(skill.effect.status, skill.effect.duration,
                                          skill.effect.stat_modifier)
                        target.add_status(se)
                        self._log(f"    {target.instance_name} afflicted with
{skill.effect.status.value}!")

    def _do_defend(self, ch: Character):
        self._defending.add(ch.name)
        ch.add_status(StatusEffect(StatusEffectType.DEFENDING, 1))
        self._log(f"  {ch.name} takes a defensive stance! (DMG -30% next turn, speed
priority)")

    # ── Monster turn ──────────────────────────────────────────────
    def _monster_turn(self, mu: MonsterUnit):
        if mu.is_incapacitated():
            self._log(f"\n  {mu.instance_name} is incapacitated and cannot act!")
            return

        alive_party = [ch for ch in self.party if not ch.is_ko and not ch.is_dead]
        if not alive_party:
            return

        action = mu.choose_action(alive_party)
        target = random.choice(alive_party)

        if action["action"] == "attack":
            self._monster_basic_attack(mu, target)
        elif action["action"] == "skill":
            skill_id = action["skill_id"]
            if skill_id in SKILL_DB:
                skill = SKILL_DB[skill_id]
                self._monster_skill(mu, skill, alive_party)
            else:
                self._monster_basic_attack(mu, target)
        elif action["action"] == "defend":
            self._log(f"  {mu.instance_name} braces for impact!")

    def _monster_basic_attack(self, mu: MonsterUnit, target: Character):
        hit = check_hit(mu.acc, target.eva)
        if not hit:
```

```
            self._log(f"\n  {mu.instance_name} attacks {target.name}... MISS!")
            return

        is_crit = check_crit(mu.template.base_stats.crit)
        # Defend reduction
        defending = target.has_status(StatusEffectType.DEFENDING)
        dmg = calculate_physical_damage(
            mu.patk, target.pdef, crit=is_crit)
        if defending:
            dmg = int(dmg * 0.7)

        # Confusion: might attack ally
        if mu.has_status(StatusEffectType.CONFUSION):
            alive_enemies = [m for m in self.monsters if not m.is_dead and m != mu]
            if alive_enemies and random.random() < 0.5:
                friendly_target = random.choice(alive_enemies)
                actual = friendly_target.take_damage(dmg)
                self._log(f"\n  {mu.instance_name} is confused and attacks
{friendly_target.instance_name}! ({actual} dmg)")
                return

        crit_str = " CRITICAL!" if is_crit else ""
        self._log(f"\n  {mu.instance_name} attacks {target.name}!{crit_str}")
        target.take_damage(dmg)
        self._log(f"    {target.name} takes {dmg} damage!")

        # Counter
        if target.has_status(StatusEffectType.COUNTER) and not target.is_ko:
            cdmg = calculate_physical_damage(target.patk, mu.pdef, power=0.5)
            mu.take_damage(cdmg)
            self._log(f"  {target.name} COUNTER! {mu.instance_name} takes {cdmg}!")

    def _monster_skill(self, mu: MonsterUnit, skill: Skill, alive_party: List[Character]):
        self._log(f"\n  {mu.instance_name} uses {skill.name}!")

        # Silence check for magic
        if skill.skill_type == SkillType.MAGICAL and mu.has_status(StatusEffectType.SILENCE):
            self._log(f"  {mu.instance_name} is silenced!")
            return
```

```
        target = random.choice(alive_party)

        if skill.skill_type in (SkillType.PHYSICAL, SkillType.MAGICAL, SkillType.SPECIAL):
            for _ in range(skill.hits):
                if target.is_ko: break
                hit = check_hit(mu.acc * skill.accuracy, target.eva)
                if not hit:
                    self._log(f"    MISS on {target.name}!")
                    continue

                is_crit = check_crit(mu.template.base_stats.crit)
                defending = target.has_status(StatusEffectType.DEFENDING)

                if skill.skill_type == SkillType.PHYSICAL:
                    dmg = calculate_physical_damage(mu.patk, target.pdef, skill.power,
is_crit)
                else:
                    # Reflect check
                    if target.has_status(StatusEffectType.REFLECT):
                        self._log(f"    {target.name} REFLECTS the spell!")
                        rdmg = calculate_magical_damage(mu.matk, mu.mdef, skill.power)
                        mu.take_damage(rdmg)
                        self._log(f"    Reflected! {mu.instance_name} takes {rdmg} damage!")
                        continue
                    dmg = calculate_magical_damage(mu.matk, target.mdef, skill.power, is_crit)

                if defending:
                    dmg = int(dmg * 0.7)

                crit_str = " CRITICAL!" if is_crit else ""
                target.take_damage(dmg)
                self._log(f"    {target.name} takes {dmg} damage!{crit_str}")

                if skill.effect and skill.effect.status and not target.is_ko:
                    chance = skill.effect.chance if skill.effect.chance > 0 else 0.4
                    if random.random() < chance:
                        se = StatusEffect(skill.effect.status, skill.effect.duration,
                                          skill.effect.stat_modifier)
                        target.add_status(se)
                        self._log(f"    {target.name} afflicted with
```

```
{skill.effect.status.value}!")

        elif skill.skill_type == SkillType.HEAL:
            heal_amt = calculate_heal(mu.template.base_stats.mdef, skill.power)
            # Heal self or an alive ally monster
            alive_m = [m for m in [mu] if not m.is_dead]
            for m in alive_m:
                old = m.current_hp
                m.current_hp = min(m.max_hp, m.current_hp + heal_amt)
                self._log(f"    {m.instance_name} recovers {m.current_hp - old} HP!")

    # ── End of round processing ──────────────────────────────────
    def _end_of_round(self):
        print(f"\n  ── End of Turn {self.turn} ──")

        # Status effect ticks for party
        for ch in self.party:
            if not ch.is_ko and not ch.is_dead:
                msgs = ch.tick_status_effects()
                msgs += ch.tick_buffs()
                for m in msgs: self._log(m)

        # KO timer management
        for ch in self.party:
            if ch.is_ko and not ch.is_dead:
                ch.ko_turns += 1
                if ch.ko_turns >= 3:
                    ch.is_dead = True
                    self._log(f"  □□{ch.name} has DIED! (Too long KO'd)")

        # Status ticks for monsters
        for mu in self.monsters:
            if not mu.is_dead:
                msgs = mu.tick_status_effects()
                for m in msgs: self._log(m)
                mu.tick_break()

        # Remove Defending status
        for ch in self.party:
            ch.remove_status(StatusEffectType.DEFENDING)
```

```
    if not self._check_battle_end():
        input("\n  [Press Enter to continue]")

def calculate_exp_reward(self) -> int:
    total = sum(mu.template.exp_reward for mu in self.monsters)
    return total

def use_item_in_battle(self, ch: Character, item_id: int,
                       inventory: Dict[int,int]) -> bool:
    """Use an item from inventory in battle. Returns True if used."""
    if item_id not in inventory or inventory[item_id] <= 0:
        print("  You don't have that item!")
        return False

    item = ITEM_DB[item_id]
    inventory[item_id] -= 1
    if inventory[item_id] <= 0:
        del inventory[item_id]

    self._log(f"  {ch.name} uses {item.name}!")

    if item.item_type == ItemType.RECOVERY:
        # Determine target
        if item.target in (SkillTarget.SINGLE_ALLY, SkillTarget.SELF):
            valid = [c for c in self.party if not c.is_dead]
            if item.target == SkillTarget.SELF:
                valid = [ch]
            if not valid: return True
            target = valid[0] if len(valid)==1 else self._pick_ally_target()
            if not target: return True
            targets = [target]
        else:
            targets = [c for c in self.party if not c.is_dead]

        for t in targets:
            if item.effect_value == -100:
                t.current_hp = t.max_hp
                t.current_mp = t.max_mp
                self._log(f"    {t.name} fully restored!")
```

```
            elif item.effect_value < 0:
                pct = abs(item.effect_value)
                amt = int(t.max_hp * pct / 100)
                actual = t.heal(amt)
                self._log(f"    {t.name} recovers {actual} HP!")
            elif item.id in (6,7,8,9,12,20):  # MP items
                mp_gain = min(item.effect_value, t.max_mp - t.current_mp)
                t.current_mp += mp_gain
                self._log(f"    {t.name} recovers {mp_gain} MP!")
            else:
                actual = t.heal(item.effect_value)
                self._log(f"    {t.name} recovers {actual} HP!")

        elif item.item_type == ItemType.REVIVAL:
            valid = [c for c in self.party if c.is_ko and not c.is_dead]
            if not valid:
                self._log("    No KO'd allies to revive!")
                return True
            target = valid[0] if len(valid)==1 else self._pick_ally_target(include_ko=True)
            if not target or not target.is_ko: return True
            if item.target == SkillTarget.ALL_ALLIES:
                for t in valid:
                    t.revive(item.effect_value)
                    self._log(f"    {t.name} revived with {item.effect_value}% HP!")
            else:
                target.revive(item.effect_value)
                self._log(f"    {target.name} revived with {item.effect_value}% HP!")

        elif item.item_type == ItemType.STATUS_CURE:
            valid = [c for c in self.party if not c.is_dead]
            if item.target == SkillTarget.ALL_ALLIES:
                targets = valid
            else:
                t = self._pick_ally_target()
                targets = [t] if t else []
            for t in targets:
                if item.status_cure:
                    for stype in item.status_cure:
                        t.remove_status(stype)
                    self._log(f"    {t.name} cleansed of ailments!")
```

```
elif item.item_type == ItemType.BUFF:
    valid = [c for c in self.party if not c.is_dead and not c.is_ko]
    if item.target == SkillTarget.ALL_ALLIES:
        targets = valid
    elif item.target == SkillTarget.SELF:
        targets = [ch]
    else:
        t = self._pick_ally_target()
        targets = [t] if t else []
    for t in targets:
        if item.stat_buff:
            for stat, mult in item.stat_buff.items():
                t.add_buff(stat, mult, item.buff_duration)
            self._log(f"    {t.name} gains buffs!")

elif item.item_type == ItemType.OFFENSIVE:
    alive_enemies = [m for m in self.monsters if not m.is_dead]
    if item.target in (SkillTarget.SINGLE_ENEMY,):
        t = self._pick_enemy_target()
        if not t: return True
        targets = [t]
    else:
        targets = alive_enemies
    for t in targets:
        is_weak = item.element and t.hit_weakness(None, item.element)
        w_bonus = 1.3 if is_weak else 1.0
        b_bonus = 1.5 if t.is_broken else 1.0
        dmg = int(item.effect_value * w_bonus * b_bonus)
        actual = t.take_damage(dmg)
        w_str = " □WEAKNESS□" if is_weak else ""
        self._log(f"    {t.instance_name} takes {actual} damage!{w_str}")
        if is_weak:
            broke = t.reduce_shield(1)
            if broke:
                self._log(f"  ⚡ {t.instance_name} is BROKEN!")

elif item.item_type in (ItemType.ADVANCED, ItemType.REVIVAL):
    # Handle advanced items
    if item.effect_value == -100 or item.effect_value == 9999:
```

```
                valid = [c for c in self.party if not c.is_dead]
                for t in valid:
                    if item.target == SkillTarget.ALL_ALLIES:
                        pass
                    elif item.target == SkillTarget.SINGLE_ALLY:
                        t = self._pick_ally_target()
                        valid = [t] if t else []
                        break
                for t in valid:
                    if item.effect_value == 9999:
                        actual = t.heal(9999)
                        self._log(f"    {t.name} recovers {actual} HP!")
                    else:
                        t.current_hp = t.max_hp
                        t.current_mp = t.max_mp
                        self._log(f"    {t.name} fully restored!")
                    if item.stat_buff:
                        for stat, mult in item.stat_buff.items():
                            t.add_buff(stat, mult, item.buff_duration)

        return True
```

Revision #1
Created 2026-03-18 16:20:22 UTC by Samuel Lee
Updated 2026-03-18 16:20:50 UTC by Samuel Lee