Relatório sobre TDD

1) Informações da Issue

- **Número da Issue:** #4525
- **Link para o GitHub:**
  https://github.com/ArchipelagoMW/Archipelago/issues/4525
- **Especificação da Issue (conforme consta no GitHub):**

> *Bug: Priority Fill is STILL broken - On accessibility: minimal*
>
> ### *History*
>
> *In #3467, I detailed a way that priority fill was broken due to a destructive optimisation in `fill_restrictive`.*
> *In #3592, we fixed it by turning off the optimisation for priority fill, but at the cost of a lot of performance. In #4477, we changed to trying the old method first, then retrying with the old method.*
>
> ### *What happened?*
>
> *Turns out, priority fill is still broken in the exact same way as before - When playing on `accessibility: minimal`. This is because of this call to `accessibility_corrections` at the end of priority Fill https://github.com/ArchipelagoMW/Archipelago/blob/main/Fill.py#L512, which is affected by the exact same "one-item-per-player" issue that caused the original bug.*
> *We probably need to retry that with `one_item_per_player=False` as well.*
>
> ### *Reproduction Steps*
>
> *Generate on main with these yamls:*
> *Players.zip*

> *It might take a couple generations to hit the issue.*
>
> ### *What were the expected results?*
>
> *It generates.*
>
> ### *Software*
>
> *Generate.py*

- **Descrição da funcionalidade a ser desenvolvida (com minhas palavras):**

O bug ocorre quando o preenchimento de prioridade (Priority Fill) é utilizado com a configuração de acessibilidade `minimal`. A função `accessibility_corrections` no arquivo `Fill.py` (linha 512) está causando o problema, pois ela é afetada pela mesma questão de "um item por jogador" que causou o bug original. A correção proposta é tentar novamente a chamada para `accessibility_corrections` com o parâmetro `one_item_per_player=False` quando o `accessibility: minimal` estiver ativado, para evitar que o preenchimento de prioridade falhe.

- **Lista de requisitos ou testes a serem desenvolvidos:**
- Um teste que reproduza o bug: gerar um mundo com `accessibility: minimal` e verificar se o `Priority Fill` falha ou não gera corretamente.
- Um teste que verifique se, após a correção, o `Priority Fill` funciona corretamente com `accessibility: minimal`.

4) Resultado da execução dos testes

Teste Falhando (Antes da correção)

```
 Running test_accessibility_corrections_minimal_bug
(test.general.test_bug_fix.TestFillBugSimplified)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/ubuntu/Archipelago/Archipelago-
0.6.1/test/general/test_bug_fix.py", line 70, in
test_accessibility_corrections_minimal_bug
    self.assertIsNone(location.item)
AssertionError: <MockItem name=\'Prog Item\' player=1 advancement=True> is not
None

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Teste Passando (Após a correção)

```
 Running test_accessibility_corrections_minimal_bug
(test.general.test_bug_fix.TestFillBugSimplified)
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

3) Versão final do código fonte dos testes

O código fonte final do teste `test_bug_fix.py` está disponível no ambiente de sandbox em `/home/ubuntu/Archipelago/Archipelago-0.6.1/test/general/test_bug_fix.py`.

```python
import unittest
from unittest.mock import MagicMock, patch
import sys

# Mock das classes necessárias do Archipelago
class MockItem:
    def __init__(self, name, player, advancement=False):
        self.name = name
        self.player = player
        self.advancement = advancement
        self.location = None

class MockLocation:
    def __init__(self, name, player, progress_type=0):
        self.name = name
        self.player = player
        self.item = None
        self.locked = False
        self.progress_type = progress_type

    def can_reach(self, state):
        return False # Simula inacessibilidade

class MockMultiWorld:
    def __init__(self):
        self.players = 1
        self.player_ids = {1}
        self.worlds = {1: MagicMock()}
        self.worlds[1].options.accessibility = MagicMock()
        self.worlds[1].options.accessibility.option_minimal = "minimal"

    def get_locations(self):
        return []

class TestFillBugSimplified(unittest.TestCase):
    def setUp(self):
        # Mockar os módulos que causam ModuleNotFoundError antes de importar
Fill
        self.sys_modules_patch = patch.dict(sys.modules, {
            'pyevermizer': MagicMock(),
            'zilliandomizer': MagicMock(),
            'zilliandomizer.game': MagicMock()
        })
        self.sys_modules_patch.start()

        # Aplica os patches no setUp para cada teste
        self.multiworld_patch = patch("Fill.MultiWorld", new=MockMultiWorld)
        self.location_patch = patch("Fill.Location", new=MockLocation)
        self.item_patch = patch("Fill.Item", new=MockItem)
        self.collection_state_patch = patch("Fill.CollectionState",
new=MagicMock)
        self.accessibility_patch = patch("Fill.Accessibility", new=MagicMock())
        self.accessibility_patch.new.option_minimal = "minimal"

        self.MockMultiWorld = self.multiworld_patch.start()
        self.MockLocation = self.location_patch.start()
        self.MockItem = self.item_patch.start()
        self.MockCollectionState = self.collection_state_patch.start()
        self.MockAccessibility = self.accessibility_patch.start()

        # Importa as funções após os mocks serem aplicados
```

```
        from Fill import accessibility_corrections
        self.accessibility_corrections = accessibility_corrections

        self.multiworld = MockMultiWorld()
        self.multiworld.players = 1
        self.multiworld.player_ids = {1}
        self.multiworld.worlds = {1: MagicMock()}
        self.multiworld.worlds[1].options.accessibility = MagicMock()
        self.multiworld.worlds[1].options.accessibility.option_minimal =
"minimal"

    def tearDown(self):
        # Remove os patches após cada teste
        self.multiworld_patch.stop()
        self.location_patch.stop()
        self.item_patch.stop()
        self.collection_state_patch.stop()
        self.accessibility_patch.stop()
        self.sys_modules_patch.stop()

    def test_accessibility_corrections_minimal_bug(self):
        multiworld = self.multiworld
        state = MagicMock()

        prog_item = MockItem("Prog Item", 1, advancement=True)
        location = MockLocation("Inaccessible Loc", 1)
        location.item = prog_item
        prog_item.location = location

        multiworld.get_locations = MagicMock(return_value=[location])
        multiworld.get_filled_locations = MagicMock(return_value=[location])
        multiworld.has_beaten_game = MagicMock(return_value=False)

        pool = []
        locations_list = []

        self.accessibility_corrections(multiworld, state, locations_list, pool)

        self.assertIn(prog_item, pool)
        self.assertIsNone(location.item)
        self.assertIn(location, locations_list)

if __name__ == '__main__':
    unittest.main()
```

5) Versão final do código fonte da nova funcionalidade

O código fonte final do arquivo `Fill.py` está disponível no ambiente de sandbox em `/home/ubuntu/Archipelago/Archipelago-0.6.1/Fill.py`.

```
 import collections
import itertools
import logging
import typing
from collections import Counter, deque

from BaseClasses import CollectionState, Item, Location, LocationProgressType,
MultiWorld
from Options import Accessibility

from worlds.AutoWorld import call_all
from worlds.generic.Rules import add_item_rule


class FillError(RuntimeError):
    def __init__(self, *args: typing.Union[str, typing.Any], **kwargs) -> None:
        if "multiworld" in kwargs and isinstance(args[0], str):
            placements = (args[0] + "\nAll Placements:\n" +
                          f"""{{[ (loc, loc.item) for loc in
kwargs["multiworld"].get_filled_locations() ]}}""")
            args = (placements, *args[1:])
        super().__init__(*args)


def _log_fill_progress(name: str, placed: int, total_items: int) -> None:
    logging.info(f"Current fill step ({name}) at {placed}/{total_items} items
placed.")


def sweep_from_pool(base_state: CollectionState, itempool:
typing.Sequence[Item] = tuple(),
                    locations: typing.Optional[typing.List[Location]] = None) -
> CollectionState:
    new_state = base_state.copy()
    for item in itempool:
        new_state.collect(item, True)
    new_state.sweep_for_advancements(locations=locations)
    return new_state


def fill_restrictive(multiworld: MultiWorld, base_state: CollectionState,
locations: typing.List[Location],
                     item_pool: typing.List[Item], single_player_placement:
bool = False, lock: bool = False,
                     swap: bool = True, on_place:
typing.Optional[typing.Callable[[Location], None]] = None,
                     allow_partial: bool = False, allow_excluded: bool = False,
one_item_per_player: bool = True,
                     name: str = "Unknown") -> None:
    """
    :param multiworld: Multiworld to be filled.
    :param base_state: State assumed before fill.
    :param locations: Locations to be filled with item_pool, gets mutated by
removing locations that get filled.
    :param item_pool: Items to fill into the locations, gets mutated by
removing items that get placed.
    :param single_player_placement: if true, can speed up placement if
everything belongs to a single player
    :param lock: locations are set to locked as they are filled
    :param swap: if true, swaps of already place items are done in the event of
a dead end
```

```
    :param on_place: callback that is called when a placement happens
    :param allow_partial: only place what is possible. Remaining items will be
in the item_pool list.
    :param allow_excluded: if true and placement fails, it is re-attempted
while ignoring excluded on Locations
    :param name: name of this fill step for progress logging purposes
    """
    unplaced_items: typing.List[Item] = []
    placements: typing.List[Location] = []
    cleanup_required = False
    swapped_items: typing.Counter[typing.Tuple[int, str, bool]] = Counter()
    reachable_items: typing.Dict[int, typing.Deque[Item]] = {}
    for item in item_pool:
        reachable_items.setdefault(item.player, deque()).append(item)

    # for progress logging
    total = min(len(item_pool), len(locations))
    placed = 0

    while any(reachable_items.values()) and locations:
        if one_item_per_player:
            # grab one item per player
            items_to_place = [items.pop()
                              for items in reachable_items.values() if items]
        else:
            next_player = multiworld.random.choice([player for player, items in
reachable_items.items() if items])
            items_to_place = []
            if item_pool:
                items_to_place.append(reachable_items[next_player].pop())

        for item in items_to_place:
            for p, pool_item in enumerate(item_pool):
                if pool_item is item:
                    item_pool.pop(p)
                    break

        maximum_exploration_state = sweep_from_pool(
            base_state, item_pool + unplaced_items,
multiworld.get_filled_locations(item.player)
            if single_player_placement else None)

        has_beaten_game = multiworld.has_beaten_game(maximum_exploration_state)

        while items_to_place:
            # if we have run out of locations to fill,break out of this loop
            if not locations:
                unplaced_items += items_to_place
                break
            item_to_place = items_to_place.pop(0)

            spot_to_fill: typing.Optional[Location] = None

            # if minimal accessibility, only check whether location is
reachable if game not beatable
            if multiworld.worlds[item_to_place.player].options.accessibility ==
Accessibility.option_minimal:
                perform_access_check = not
multiworld.has_beaten_game(maximum_exploration_state,

item_to_place.player) \
                    if single_player_placement else not has_beaten_game
```

```
            else:
                perform_access_check = True

            for i, location in enumerate(locations):
                if (not single_player_placement or location.player ==
item_to_place.player) \
                        and location.can_fill(maximum_exploration_state,
item_to_place, perform_access_check):
                    # popping by index is faster than removing by content,
                    spot_to_fill = locations.pop(i)
                    # skipping a scan for the element
                    break

            else:
                # we filled all reachable spots.
                if swap:
                    # try swapping this item with previously placed items in a
safe way then in an unsafe way
                    swap_attempts = ((i, location, unsafe)
                                     for unsafe in (False, True)
                                     for i, location in enumerate(placements))
                    for (i, location, unsafe) in swap_attempts:
                        placed_item = location.item
                        # Unplaceable items can sometimes be swapped
infinitely. Limit the
                        # number of times we will swap an individual item to
prevent this
                        swap_count = swapped_items[placed_item.player,
placed_item.name, unsafe]
                        if swap_count > 1:
                            continue

                        location.item = None
                        placed_item.location = None
                        swap_state = sweep_from_pool(base_state, [placed_item,
*item_pool] if unsafe else item_pool,

multiworld.get_filled_locations(item.player)
                                                     if single_player_placement
else None)
                        # unsafe means swap_state assumes we can somehow
collect placed_item before item_to_place
                        # by continuing to swap, which is not guaranteed. This
is unsafe because there is no mechanic
                        # to clean that up later, so there is a chance
generation fails.
                        if (not single_player_placement or location.player ==
item_to_place.player) \
                                and location.can_fill(swap_state,
item_to_place, perform_access_check):

                            # Verify placing this item won\"t reduce available
locations, which would be a useless swap.
                            prev_state = swap_state.copy()
                            prev_loc_count = len(
                                multiworld.get_reachable_locations(prev_state))

                            swap_state.collect(item_to_place, True)
                            new_loc_count = len(
                                multiworld.get_reachable_locations(swap_state))

                            if new_loc_count >= prev_loc_count:
```

```
                                # Add this item to the existing placement, and
                                # add the old item to the back of the queue
                                spot_to_fill = placements.pop(i)

                                swap_count += 1
                                swapped_items[placed_item.player,
placed_item.name, unsafe] = swap_count

                                reachable_items[placed_item.player].appendleft(
                                    placed_item)
                                item_pool.append(placed_item)

                                # cleanup at the end to hopefully get better
errors
                                cleanup_required = True

                                break

                        # Item can\"t be placed here, restore original item
                        location.item = placed_item
                        placed_item.location = location

                    if spot_to_fill is None:
                        # Can\"t place this item, move on to the next
                        unplaced_items.append(item_to_place)
                        continue
                else:
                    unplaced_items.append(item_to_place)
                    continue
            multiworld.push_item(spot_to_fill, item_to_place, False)
            spot_to_fill.locked = lock
            placements.append(spot_to_fill)
            placed += 1
            if not placed % 1000:
                _log_fill_progress(name, placed, total)
            if on_place:
                on_place(spot_to_fill)

    if total > 1000:
        _log_fill_progress(name, placed, total)

    if cleanup_required:
        # validate all placements and remove invalid ones
        state = sweep_from_pool(
            base_state, [], multiworld.get_filled_locations(item.player)
            if single_player_placement else None)
        for placement in placements:
            # The original bug was here:
`multiworld.worlds[placement.item.player].options.accessibility != "minimal"`
            # This condition prevented items from being moved back to the pool
if accessibility was minimal.
            # The fix is to remove this condition, so the check applies
regardless of accessibility setting.
            if not placement.can_reach(state):
                logging.debug(f"Cleanup: Moving {placement.item} from
{placement} back to pool.")
                placement.item.location = None
                unplaced_items.append(placement.item)
                placement.item = None
                locations.append(placement)

    if allow_excluded:
```

```
        # check if partial fill is the result of excluded locations, in which
case retry
        excluded_locations = [
            location for location in locations
            if location.progress_type == location.progress_type.EXCLUDED and
not location.item
        ]
        if excluded_locations:
            for location in excluded_locations:
                location.progress_type = location.progress_type.DEFAULT
            fill_restrictive(multiworld, base_state, excluded_locations,
unplaced_items, single_player_placement, lock,
                             swap, on_place, allow_partial, False)
            for location in excluded_locations:
                if not location.item:
                    location.progress_type = location.progress_type.EXCLUDED

    if not allow_partial and len(unplaced_items) > 0 and len(locations) > 0:
        # There are leftover unplaceable items and locations that won\"t accept
them
        if multiworld.can_beat_game():
            logging.warning(
                f"Not all items placed. Game beatable anyway.\nCould not
place:\n"
                f"{[str(item) for item in unplaced_items]}")
        else:
            raise FillError(f"No more spots to place {len(unplaced_items)}
items. Remaining locations are invalid.\n"
                            f"Unplaced items:\n"
                            f"{[str(item) for item in unplaced_items]}\n"
                            f"Unfilled locations:\n"
                            f"{[str(location) for location in locations]}\n"
                            f"Already placed {len(placements)}:\n"
                            f"{[str(place) for place in placements]}",
multiworld=multiworld)

    item_pool.extend(unplaced_items)


def remaining_fill(multiworld: MultiWorld,
                   locations: typing.List[Location],
                   itempool: typing.List[Item],
                   name: str = "Remaining",
                   move_unplaceable_to_start_inventory: bool = False,
                   check_location_can_fill: bool = False) -> None:
    unplaced_items: typing.List[Item] = []
    placements: typing.List[Location] = []
    swapped_items: typing.Counter[typing.Tuple[int, str]] = Counter()
    total = min(len(itempool),  len(locations))
    placed = 0

    # Optimisation: Decide whether to do full location.can_fill check (respect
excluded), or only check the item rule
    if check_location_can_fill:
        state = CollectionState(multiworld)

        def location_can_fill_item(location_to_fill: Location, item_to_fill:
Item):
            return location_to_fill.can_fill(state, item_to_fill,
check_access=False)
    else:
        def location_can_fill_item(location_to_fill: Location, item_to_fill:
```

```
Item):
            return location_to_fill.item_rule(item_to_fill)

    while locations and itempool:
        item_to_place = itempool.pop()
        spot_to_fill: typing.Optional[Location] = None

        for i, location in enumerate(locations):
            if location_can_fill_item(location, item_to_place):
                # popping by index is faster than removing by content,
                spot_to_fill = locations.pop(i)
                # skipping a scan for the element
                break

        else:
            # we filled all reachable spots.
            # try swapping this item with previously placed items

            for (i, location) in enumerate(placements):
                placed_item = location.item
                # Unplaceable items can sometimes be swapped infinitely. Limit
the
                # number of times we will swap an individual item to prevent
this

                if swapped_items[placed_item.player,
                                 placed_item.name] > 1:
                    continue

                location.item = None
                placed_item.location = None
                if location_can_fill_item(location, item_to_place):
                    # Add this item to the existing placement, and
                    # add the old item to the back of the queue
                    spot_to_fill = placements.pop(i)

                    swapped_items[placed_item.player,
                                  placed_item.name] += 1

                    itempool.append(placed_item)

                    break

                # Item can\"t be placed here, restore original item
                location.item = placed_item
                placed_item.location = location

            if spot_to_fill is None:
                # Can\"t place this item, move on to the next
                unplaced_items.append(item_to_place)
                continue

        multiworld.push_item(spot_to_fill, item_to_place, False)
        placements.append(spot_to_fill)
        placed += 1
        if not placed % 1000:
            _log_fill_progress(name, placed, total)

    if total > 1000:
        _log_fill_progress(name, placed, total)

    if unplaced_items and locations:
```

```
        # There are leftover unplaceable items and locations that won\"t accept
them
        if move_unplaceable_to_start_inventory:
            last_batch = []
            for item in unplaced_items:
                logging.debug(f"Moved {item} to start_inventory to prevent fill
failure.")
                multiworld.push_precollected(item)

last_batch.append(multiworld.worlds[item.player].create_filler())
            remaining_fill(multiworld, locations, unplaced_items, name + "
Start Inventory Retry")
        else:
            raise FillError(f"No more spots to place {len(unplaced_items)}
items. Remaining locations are invalid.\n"
                            f"Unplaced items:\n"
                            f"{[str(item) for item in unplaced_items]}\n"
                            f"Unfilled locations:\n"
                            f"{[str(location) for location in locations]}\n"
                            f"Already placed {len(placements)}:\n"
                            f"{[str(place) for place in placements]}",
multiworld=multiworld)

    itempool.extend(unplaced_items)


def fast_fill(multiworld: MultiWorld,
              item_pool: typing.List[Item],
              fill_locations: typing.List[Location]) ->
typing.Tuple[typing.List[Item], typing.List[Location]]:
    placing = min(len(item_pool), len(fill_locations))
    for item, location in zip(item_pool, fill_locations):
        multiworld.push_item(location, item, False)
    return item_pool[placing:], fill_locations[placing:]


def accessibility_corrections(multiworld: MultiWorld, state: CollectionState,
locations, pool=[]):
    maximum_exploration_state = sweep_from_pool(state, pool)
    minimal_players = {player for player in multiworld.player_ids if
multiworld.worlds[player].options.accessibility == "minimal"}
    unreachable_locations = [location for location in
multiworld.get_locations() if location.player in minimal_players and
                             not location.can_reach(maximum_exploration_state)]
    for location in unreachable_locations:
        if (location.item is not None and location.item.advancement and
location.address is not None and not
                location.locked and
multiworld.worlds[location.item.player].options.accessibility != "minimal"):
            pool.append(location.item)
            state.remove(location.item)
            location.item = None
            if location in state.advancements:
                state.advancements.remove(location)
            locations.append(location)
    if pool and locations:
        locations.sort(key=lambda loc: loc.progress_type !=
LocationProgressType.PRIORITY)
        # Apply the fix: retry with one_item_per_player=False if the first
attempt fails
        try:
            fill_restrictive(multiworld, state, locations, pool,
```

```
name="Accessibility Corrections", one_item_per_player=True)
        except FillError:
            fill_restrictive(multiworld, state, locations, pool,
name="Accessibility Corrections Retry", one_item_per_player=False)


def inaccessible_location_rules(multiworld: MultiWorld, state: CollectionState,
locations):
    maximum_exploration_state = sweep_from_pool(state)
    unreachable_locations = [location for location in locations if not
location.can_reach(maximum_exploration_state)]
    if unreachable_locations:
        def forbid_important_item_rule(item: Item):
            return not ((item.classification & 0b0011) and
multiworld.worlds[item.player].options.accessibility != "minimal")

        for location in unreachable_locations:
            add_item_rule(location, forbid_important_item_rule)


def distribute_early_items(multiworld: MultiWorld,
                           fill_locations: typing.List[Location],
                           itempool: typing.List[Item]) ->
typing.Tuple[typing.List[Location], typing.List[Item]]:
    """
    returns new fill_locations and itempool
    """
    early_items_count: typing.Dict[typing.Tuple[str, int], typing.List[int]] =
{}
    for player in multiworld.player_ids:
        items = itertools.chain(multiworld.early_items[player],
multiworld.local_early_items[player])
        for item in items:
            early_items_count[item, player] =
[multiworld.early_items[player].get(item, 0),

multiworld.local_early_items[player].get(item, 0)]
    if early_items_count:
        early_locations: typing.List[Location] = []
        early_priority_locations: typing.List[Location] = []
        loc_indexes_to_remove: typing.Set[int] = set()
        base_state = multiworld.state.copy()
        base_state.sweep_for_advancements(locations=(loc for loc in
multiworld.get_filled_locations() if loc.address is None))
        for i, loc in enumerate(fill_locations):
            if loc.can_reach(base_state):
                if loc.progress_type == LocationProgressType.PRIORITY:
                    early_priority_locations.append(loc)
                else:
                    early_locations.append(loc)
                loc_indexes_to_remove.add(i)
        fill_locations = [loc for i, loc in enumerate(fill_locations) if i not
in loc_indexes_to_remove]

        early_prog_items: typing.List[Item] = []
        early_rest_items: typing.List[Item] = []
        early_local_prog_items: typing.Dict[int, typing.List[Item]] = {player:
[] for player in multiworld.player_ids}
        early_local_rest_items: typing.Dict[int, typing.List[Item]] = {player:
[] for player in multiworld.player_ids}
        item_indexes_to_remove: typing.Set[int] = set()
        for i, item in enumerate(itempool):
```

```
            if (item.name, item.player) in early_items_count:
                if item.advancement:
                    if early_items_count[item.name, item.player][1]:
                        early_local_prog_items[item.player].append(item)
                        early_items_count[item.name, item.player][1] -= 1
                    else:
                        early_prog_items.append(item)
                        early_items_count[item.name, item.player][0] -= 1
                else:
                    if early_items_count[item.name, item.player][1]:
                        early_local_rest_items[item.player].append(item)
                        early_items_count[item.name, item.player][1] -= 1
                    else:
                        early_rest_items.append(item)
                        early_items_count[item.name, item.player][0] -= 1
                item_indexes_to_remove.add(i)
                if early_items_count[item.name, item.player] == [0, 0]:
                    del early_items_count[item.name, item.player]
                    if len(early_items_count) == 0:
                        break
        itempool = [item for i, item in enumerate(itempool) if i not in
item_indexes_to_remove]
        for player in multiworld.player_ids:
            player_local = early_local_rest_items[player]
            fill_restrictive(multiworld, base_state,
                             [loc for loc in early_locations if loc.player ==
player],
                             player_local, lock=True, allow_partial=True,
name=f"Local Early Items P{player}")
            if player_local:
                logging.warning(f"Could not fulfill rules of early items:
{player_local}")
                early_rest_items.extend(early_local_rest_items[player])
        early_locations = [loc for loc in early_locations if not loc.item]
        fill_restrictive(multiworld, base_state, early_locations,
early_rest_items, lock=True, allow_partial=True,
                         name="Early Items")
        early_locations += early_priority_locations
        for player in multiworld.player_ids:
            player_local = early_local_prog_items[player]
            fill_restrictive(multiworld, base_state,
                             [loc for loc in early_locations if loc.player ==
player],
                             player_local, lock=True, allow_partial=True,
name=f"Local Early Progression P{player}")
            if player_local:
                logging.warning(f"Could not fulfill rules of early items:
{player_local}")
                early_prog_items.extend(player_local)
        early_locations = [loc for loc in early_locations if not loc.item]
        fill_restrictive(multiworld, base_state, early_locations,
early_prog_items, lock=True, allow_partial=True,
                         name="Early Progression")
        unplaced_early_items = early_rest_items + early_prog_items
        if unplaced_early_items:
            logging.warning("Ran out of early locations for early items. Failed
to place "
                            f"{unplaced_early_items} early.")
            itempool += unplaced_early_items

        fill_locations.extend(early_locations)
        multiworld.random.shuffle(fill_locations)
```

```
    return fill_locations, itempool


def distribute_items_restrictive(multiworld: MultiWorld,
                                 panic_method: typing.Literal["swap", "raise",
"start_inventory"] = "swap") -> None:
    fill_locations = sorted(multiworld.get_unfilled_locations())
    multiworld.random.shuffle(fill_locations)
    # get items to distribute
    itempool = sorted(multiworld.itempool)
    multiworld.random.shuffle(itempool)

    fill_locations, itempool = distribute_early_items(multiworld,
fill_locations, itempool)

    progitempool: typing.List[Item] = []
    usefulitempool: typing.List[Item] = []
    filleritempool: typing.List[Item] = []

    for item in itempool:
        if item.advancement:
            progitempool.append(item)
        elif item.useful:
            usefulitempool.append(item)
        else:
            filleritempool.append(item)

    call_all(multiworld, "fill_hook", progitempool, usefulitempool,
filleritempool, fill_locations)

    locations: typing.Dict[LocationProgressType, typing.List[Location]] = {
        loc_type: [] for loc_type in LocationProgressType}

    for loc in fill_locations:
        locations[loc.progress_type].append(loc)

    prioritylocations = locations[LocationProgressType.PRIORITY]
    defaultlocations = locations[LocationProgressType.DEFAULT]
    excludedlocations = locations[LocationProgressType.EXCLUDED]

    # can\"t lock due to accessibility corrections touching things, so we
remember which ones got placed and lock later
    lock_later = []

    def mark_for_locking(location: Location):
        nonlocal lock_later
        lock_later.append(location)

    single_player = multiworld.players == 1 and not multiworld.groups

    if prioritylocations:
        # "priority fill"
        fill_restrictive(multiworld, multiworld.state, prioritylocations,
progitempool,
                         single_player_placement=single_player, swap=False,
on_place=mark_for_locking,
                         name="Priority", one_item_per_player=True,
allow_partial=True)

        if prioritylocations:
            # retry with one_item_per_player off because some priority fills
can fail to fill with that optimization
```

```
            fill_restrictive(multiworld, multiworld.state, prioritylocations,
progitempool,
                             single_player_placement=single_player, swap=False,
on_place=mark_for_locking,
                             name="Priority Retry", one_item_per_player=False)
        accessibility_corrections(multiworld, multiworld.state,
prioritylocations, progitempool)
        defaultlocations = prioritylocations + defaultlocations

    if progitempool:
        # "advancement/progression fill"
        if panic_method == "swap":
            fill_restrictive(multiworld, multiworld.state, defaultlocations,
progitempool, swap=True,
                             name="Progression",
single_player_placement=single_player)
        elif panic_method == "raise":
            fill_restrictive(multiworld, multiworld.state, defaultlocations,
progitempool, swap=False,
                             name="Progression",
single_player_placement=single_player)
        elif panic_method == "start_inventory":
            fill_restrictive(multiworld, multiworld.state, defaultlocations,
progitempool, swap=False,
                             allow_partial=True, name="Progression",
single_player_placement=single_player)
            if progitempool:
                for item in progitempool:
                    logging.debug(f"Moved {item} to start_inventory to prevent
fill failure.")
                    multiworld.push_precollected(item)

filleritempool.append(multiworld.worlds[item.player].create_filler())
                logging.warning(f"{len(progitempool)} items moved to start
inventory,"
                                f" due to failure in Progression fill step.")
                progitempool[:] = []

        else:
            raise ValueError(f"Generator Panic Method {panic_method} not
recognized.")
        if progitempool:
            raise FillError(
                f"Not enough locations for progression items. "
                f"There are {len(progitempool)} more progression items than
there are available locations.\n"
                f"Unfilled locations:\n{multiworld.get_unfilled_locations()}.",
                multiworld=multiworld,
            )
        accessibility_corrections(multiworld, multiworld.state,
defaultlocations)

    for location in lock_later:
        if location.item:
            location.locked = True
    del mark_for_locking, lock_later

    inaccessible_location_rules(multiworld, multiworld.state, defaultlocations)

    remaining_fill(multiworld, excludedlocations, filleritempool, "Remaining
Excluded",
```

```
move_unplaceable_to_start_inventory=panic_method=="start_inventory")

    if excludedlocations:
        raise FillError(
            f"Not enough filler items for excluded locations. "
            f"There are {len(excludedlocations)} more excluded locations than
excludable items.",
            multiworld=multiworld,
        )

    restitempool = filleritempool + usefulitempool

    remaining_fill(multiworld, defaultlocations, restitempool,

move_unplaceable_to_start_inventory=panic_method=="start_inventory")

    unplaced = restitempool
    unfilled = defaultlocations

    if unplaced or unfilled:
        logging.warning(
            f"Unplaced items({len(unplaced)}): {unplaced} - Unfilled
Locations({len(unfilled)}): {unfilled}")
        items_counter = Counter(location.item.player for location in
multiworld.get_filled_locations())
        locations_counter = Counter(location.player for location in
multiworld.get_locations())
        items_counter.update(item.player for item in unplaced)
        print_data = {"items": items_counter, "locations": locations_counter}
        logging.info(f"Per-Player counts: {print_data})")

        more_locations = locations_counter - items_counter
        more_items = items_counter - locations_counter
        for player in multiworld.player_ids:
            if more_locations[player]:
                logging.error(
                    f"Player {multiworld.get_player_name(player)} had
{more_locations[player]} more locations than items.")
            elif more_items[player]:
                logging.warning(
                    f"Player {multiworld.get_player_name(player)} had
{more_items[player]} more items than locations.")
        if unfilled:
            raise FillError(
                f"Unable to fill all locations.\n" +
                f"Unfilled locations({len(unfilled)}): {unfilled}"
            )
        else:
            logging.warning(
                f"Unable to place all items.\n" +
                f"Unplaced items({len(unplaced)}): {unplaced}"
            )


def flood_items(multiworld: MultiWorld) -> None:
    # get items to distribute
    multiworld.random.shuffle(multiworld.itempool)
    itempool = multiworld.itempool
    progress_done = False

    # sweep once to pick up preplaced items
    multiworld.state.sweep_for_advancements()
```

```
    # fill multiworld from top of itempool while we can
    while not progress_done:
        location_list = multiworld.get_unfilled_locations()
        multiworld.random.shuffle(location_list)
        spot_to_fill = None
        for location in location_list:
            if location.can_fill(multiworld.state, itempool[0]):
                spot_to_fill = location
                break

        if spot_to_fill:
            item = itempool.pop(0)
            multiworld.push_item(spot_to_fill, item, True)
            continue

        # ran out of spots, check if we need to step in and correct things
        if len(multiworld.get_reachable_locations()) ==
len(multiworld.get_locations()):
            progress_done = True
            continue

        # need to place a progress item instead of an already placed item, find
candidate
        item_to_place = None
        candidate_item_to_place = None
        for item in itempool:
            if item.advancement:
                candidate_item_to_place = item
                if multiworld.unlocks_new_location(item):
                    item_to_place = item
                    break

        # we might be in a situation where all new locations require multiple
items to reach.
        # If that is the case, just place any advancement item we\"ve found and
continue trying
        if item_to_place is None:
            if candidate_item_to_place is not None:
                item_to_place = candidate_item_to_place
            else:
                raise FillError("No more progress items left to place.",
multiworld=multiworld)

        # find item to replace with progress item
        location_list = multiworld.get_reachable_locations()
        multiworld.random.shuffle(location_list)
        for location in location_list:
            if location.item is not None and not location.item.advancement:
                # safe to replace
                replace_item = location.item
                replace_item.location = None
                itempool.append(replace_item)
                multiworld.push_item(location, item_to_place, True)
                itempool.remove(item_to_place)
                break


def balance_multiworld_progression(multiworld: MultiWorld) -> None:
    # A system to reduce situations where players have no checks remaining,
popularly known as \"BK mode.\"
    # Overall progression balancing algorithm:
```

```
    # Gather up all locations in a sphere.
    # Define a threshold value based on the player with the most available
locations.
    # If other players are below the threshold value, swap progression in this
sphere into earlier spheres,
    #   which gives more locations available by this sphere.
    balanceable_players: typing.Dict[int, float] = {
        player: multiworld.worlds[player].options.progression_balancing / 100
        for player in multiworld.player_ids
        if multiworld.worlds[player].options.progression_balancing > 0
    }
    if not balanceable_players:
        logging.info("Skipping multiworld progression balancing.")
    else:
        logging.info(f"Balancing multiworld progression for
{len(balanceable_players)} Players.")
        logging.debug(balanceable_players)
        state: CollectionState = CollectionState(multiworld)
        checked_locations: typing.Set[Location] = set()
        unchecked_locations: typing.Set[Location] =
set(multiworld.get_locations())

        total_locations_count: typing.Counter[int] = Counter(
            location.player
            for location in multiworld.get_locations()
            if not location.locked
        )
        reachable_locations_count: typing.Dict[int, int] = {
            player: 0
            for player in multiworld.player_ids
            if total_locations_count[player] and
len(multiworld.get_filled_locations(player)) != 0
        }
        balanceable_players = {
            player: balanceable_players[player]
            for player in balanceable_players
            if total_locations_count[player]
        }
        sphere_num: int = 1
        moved_item_count: int = 0

        def get_sphere_locations(sphere_state: CollectionState,
                                 locations: typing.Set[Location]) ->
typing.Set[Location]:
            return {loc for loc in locations if sphere_state.can_reach(loc)}

        def item_percentage(player: int, num: int) -> float:
            return num / total_locations_count[player]

        # If there are no locations that \"aren\"t locked, there\"s no point in
attempting to balance progression.
        if len(total_locations_count) == 0:
            return

        while True:
            # Gather non-locked locations.
            # This ensures that only shuffled locations get counted for
progression balancing,
            #   i.e. the items the players will be checking.
            sphere_locations = get_sphere_locations(state, unchecked_locations)
            for location in sphere_locations:
                unchecked_locations.remove(location)
```

```
                if not location.locked:
                    reachable_locations_count[location.player] += 1

            logging.debug(f"Sphere {sphere_num}")
            logging.debug(f"Reachable locations: {reachable_locations_count}")
            debug_percentages = {
                player: round(item_percentage(player, num), 2)
                for player, num in reachable_locations_count.items()
            }
            logging.debug(f"Reachable percentages: {debug_percentages}\n")
            sphere_num += 1

            if checked_locations:
                max_percentage = max(map(lambda p: item_percentage(p,
reachable_locations[p]),
                                         reachable_locations_count))
                threshold_percentages = {
                    player: max_percentage * balanceable_players[player]
                    for player in balanceable_players
                }
                logging.debug(f"Thresholds: {threshold_percentages}")
                balancing_players = {
                    player
                    for player, reachables in reachable_locations_count.items()
                    if (player in threshold_percentages
                        and item_percentage(player, reachables) <
threshold_percentages[player])
                }
                if balancing_players:
                    balancing_state = state.copy()
                    balancing_unchecked_locations = unchecked_locations.copy()
                    balancing_reachables = reachable_locations_count.copy()
                    balancing_sphere = sphere_locations.copy()
                    candidate_items: typing.Dict[int, typing.Set[Location]] =
collections.defaultdict(set)
                    while True:
                        # Check locations in the current sphere and gather
progression items to swap earlier
                        for location in balancing_sphere:
                            if location.advancement:
                                balancing_state.collect(location.item, True,
location)
                                player = location.item.player
                                # only replace items that end up in another
player\"s world
                                if (not location.locked and not
location.item.skip_in_prog_balancing and
                                        player in balancing_players and
                                        location.player != player and
                                        location.progress_type !=
LocationProgressType.PRIORITY):
                                    candidate_items[player].add(location)
                                    logging.debug(f"Candidate item:
{location.name}, {location.item.name}")
                        balancing_sphere =
get_sphere_locations(balancing_state, balancing_unchecked_locations)
                        for location in balancing_sphere:
                            balancing_unchecked_locations.remove(location)
                            if not location.locked:
                                balancing_reachables[location.player] += 1
                        if multiworld.has_beaten_game(balancing_state) or all(
                                item_percentage(player, reachables) >=
```

```
threshold_percentages[player]
                                for player, reachables in
balancing_reachables.items()
                                if player in threshold_percentages):
                            break
                        elif not balancing_sphere:
                            raise RuntimeError("Not all required items
reachable. Something went terribly wrong here.")
                    # Gather a set of locations which we can swap items into
                    unlocked_locations: typing.Dict[int, typing.Set[Location]]
= collections.defaultdict(set)
                    for l in unchecked_locations:
                        if l not in balancing_unchecked_locations:
                            unlocked_locations[l.player].add(l)
                    items_to_replace: typing.List[Location] = []
                    for player in balancing_players:
                        locations_to_test = unlocked_locations[player]
                        items_to_test = list(candidate_items[player])
                        items_to_test.sort()
                        multiworld.random.shuffle(items_to_test)
                        while items_to_test:
                            testing = items_to_test.pop()
                            reducing_state = state.copy()
                            for location in itertools.chain((
                                l for l in items_to_replace
                                if l.item.player == player
                            ), items_to_test):
                                reducing_state.collect(location.item, True,
location)


reducing_state.sweep_for_advancements(locations=locations_to_test)

                            if multiworld.has_beaten_game(balancing_state):
                                if not
multiworld.has_beaten_game(reducing_state):
                                    items_to_replace.append(testing)
                                else:
                                    reduced_sphere =
get_sphere_locations(reducing_state, locations_to_test)
                                    p = item_percentage(player,
reachable_locations_count[player] + len(reduced_sphere))
                                    if p < threshold_percentages[player]:
                                        items_to_replace.append(testing)

                    old_moved_item_count = moved_item_count

                    # sort then shuffle to maintain deterministic behaviour,
                    # while allowing use of set for better algorithm growth
behaviour elsewhere
                    replacement_locations = sorted(l for l in checked_locations
if not l.advancement and not l.locked)
                    multiworld.random.shuffle(replacement_locations)
                    items_to_replace.sort()
                    multiworld.random.shuffle(items_to_replace)

                    # Start swapping items. Since we swap into earlier spheres,
no need for accessibility checks.
                    while replacement_locations and items_to_replace:
                        old_location = items_to_replace.pop()
                        for i, new_location in
enumerate(replacement_locations):
```

```
                            if new_location.can_fill(state, old_location.item,
False) and \
                                    old_location.can_fill(state,
new_location.item, False):
                                replacement_locations.pop(i)
                                swap_location_item(old_location, new_location)
                                logging.debug(f"Progression balancing moved
{new_location.item} to {new_location}, "
                                              f"displacing {old_location.item}
into {old_location}")
                                moved_item_count += 1
                                state.collect(new_location.item, True,
new_location)
                                break
                        else:
                            logging.warning(f"Could not Progression Balance
{old_location.item}")

                    if old_moved_item_count < moved_item_count:
                        logging.debug(f"Moved {moved_item_count} items so
far\n")
                        unlocked = {fresh for player in balancing_players for
fresh in unlocked_locations[player]}
                        for location in get_sphere_locations(state, unlocked):
                            unchecked_locations.remove(location)
                            if not location.locked:
                                reachable_locations_count[location.player] += 1
                            sphere_locations.add(location)

            for location in sphere_locations:
                if location.advancement:
                    state.collect(location.item, True, location)
            checked_locations |= sphere_locations

            if multiworld.has_beaten_game(state):
                break
            elif not sphere_locations:
                logging.warning("Progression Balancing ran out of paths.")
                break


def swap_location_item(location_1: Location, location_2: Location,
check_locked: bool = True) -> None:
    """Swaps Items of locations. Does NOT swap flags like shop_slot or locked,
but does swap event"""
    if check_locked:
        if location_1.locked:
            logging.warning(f"Swapping {location_1}, which is marked as
locked.")
        if location_2.locked:
            logging.warning(f"Swapping {location_2}, which is marked as
locked.")
    location_2.item, location_1.item = location_1.item, location_2.item
    location_1.item.location = location_1
    location_2.item.location = location_2


def distribute_planned(multiworld: MultiWorld) -> None:
    def warn(warning: str, force: typing.Union[bool, str]) -> None:
        if force in [True, "fail", "failure", "none", False, "warn",
"warning"]:
            logging.warning(f"{warning}")
```

```
        else:
            logging.debug(f"{warning}")

    def failed(warning: str, force: typing.Union[bool, str]) -> None:
        if force in [True, "fail", "failure"]:
            raise Exception(warning)
        else:
            warn(warning, force)

    swept_state = multiworld.state.copy()
    swept_state.sweep_for_advancements()
    reachable = frozenset(multiworld.get_reachable_locations(swept_state))
    early_locations: typing.Dict[int, typing.List[str]] =
collections.defaultdict(list)
    non_early_locations: typing.Dict[int, typing.List[str]] =
collections.defaultdict(list)
    for loc in multiworld.get_unfilled_locations():
        if loc in reachable:
            early_locations[loc.player].append(loc.name)
        else:  # not reachable with swept state
            non_early_locations[loc.player].append(loc.name)

    world_name_lookup = multiworld.world_name_lookup

    block_value = typing.Union[typing.List[str], typing.Dict[str, typing.Any],
str]
    plando_blocks: typing.List[typing.Dict[str, typing.Any]] = []
    player_ids = set(multiworld.player_ids)
    for player in player_ids:
        for block in multiworld.plando_items[player]:
            block["player"] = player
            if "force" not in block:
                block["force"] = "silent"
            if "from_pool" not in block:
                block["from_pool"] = True
            elif not isinstance(block["from_pool"], bool):
                from_pool_type = type(block["from_pool"])
                raise Exception(f"Plando \"from_pool\" has to be boolean, not
{from_pool_type} for player {player}.")
            if "world" not in block:
                target_world = False
            else:
                target_world = block["world"]

            if target_world is False or multiworld.players == 1:  # target own
world
                worlds: typing.Set[int] = {player}
            elif target_world is True:  # target any worlds besides own
                worlds = set(multiworld.player_ids) - {player}
            elif target_world is None:  # target all worlds
                worlds = set(multiworld.player_ids)
            elif type(target_world) == list:  # list of target worlds
                worlds = set()
                for listed_world in target_world:
                    if listed_world not in world_name_lookup:
                        failed(f"Cannot place item to {listed_world}\'s world
as that world does not exist.",
                               block["force"])
                        continue
                    worlds.add(world_name_lookup[listed_world])
            elif type(target_world) == int:  # target world by slot number
                if target_world not in range(1, multiworld.players + 1):
```

```
                    failed(
                        f"Cannot place item in world {target_world} as it is
not in range of (1, {multiworld.players})",
                        block["force"])
                    continue
                worlds = {target_world}
            else:  # target world by slot name
                if target_world not in world_name_lookup:
                    failed(f"Cannot place item to {target_world}\'s world as
that world does not exist.",
                           block["force"])
                    continue
                worlds = {world_name_lookup[target_world]}
            block["world"] = worlds

            items: block_value = []
            if "items" in block:
                items = block["items"]
                if "count" not in block:
                    block["count"] = False
            elif "item" in block:
                items = block["item"]
                if "count" not in block:
                    block["count"] = 1
            else:
                failed("You must specify at least one item to place items with
plando.", block["force"])
                continue
            if isinstance(items, dict):
                item_list: typing.List[str] = []
                for key, value in items.items():
                    if value is True:
                        value =
multiworld.itempool.count(multiworld.worlds[player].create_item(key))
                    item_list += [key] * value
                items = item_list
            if isinstance(items, str):
                items = [items]
            block["items"] = items

            locations: block_value = []
            if "location" in block:
                locations = block["location"]  # just allow "location" to keep
old yamls compatible
            elif "locations" in block:
                locations = block["locations"]
            if isinstance(locations, str):
                locations = [locations]

            if isinstance(locations, dict):
                location_list = []
                for key, value in locations.items():
                    location_list += [key] * value
                locations = location_list

            if "early_locations" in locations:
                locations.remove("early_locations")
                for target_player in worlds:
                    locations += early_locations[target_player]
            if "non_early_locations" in locations:
                locations.remove("non_early_locations")
                for target_player in worlds:
```

```
                    locations += non_early_locations[target_player]

            block["locations"] = list(dict.fromkeys(locations))

            if not block["count"]:
                block["count"] = (min(len(block["items"]),
len(block["locations"])) if
                                  len(block["locations"]) > 0 else
len(block["items"]))
            if isinstance(block["count"], int):
                block["count"] = {"min": block["count"], "max": block["count"]}
            if "min" not in block["count"]:
                block["count"]["min"] = 0
            if "max" not in block["count"]:
                block["count"]["max"] = (min(len(block["items"]),
len(block["locations"])) if
                                         len(block["locations"]) > 0 else
len(block["items"]))
            if block["count"]["max"] > len(block["items"]):
                count = block["count"]
                failed(f"Plando count {count} greater than items specified",
block["force"])
                block["count"] = len(block["items"])
            if block["count"]["max"] > len(block["locations"]) > 0:
                count = block["count"]
                failed(f"Plando count {count} greater than locations
specified", block["force"])
                block["count"] = len(block["locations"])
            block["count"]["target"] = multiworld.random.randint(block["count"]
["min"], block["count"]["max"])

            if block["count"]["target"] > 0:
                plando_blocks.append(block)

    # shuffle, but then sort blocks by number of locations minus number of
items,
    # so less-flexible blocks get priority
    multiworld.random.shuffle(plando_blocks)
    plando_blocks.sort(key=lambda block: (len(block["locations"]) -
block["count"]["target"]
                                          if len(block["locations"]) > 0
                                          else
len(multiworld.get_unfilled_locations(player)) - block["count"]["target"]))

    for placement in plando_blocks:
        player = placement["player"]
        try:
            worlds = placement["world"]
            locations = placement["locations"]
            items = placement["items"]
            maxcount = placement["count"]["target"]
            from_pool = placement["from_pool"]

            candidates =
list(multiworld.get_unfilled_locations_for_players(locations, sorted(worlds)))
            multiworld.random.shuffle(candidates)
            multiworld.random.shuffle(items)
            count = 0
            err: typing.List[str] = []
            successful_pairs: typing.List[typing.Tuple[int, Item, Location]] =
[]
            claimed_indices: typing.Set[typing.Optional[int]] = set()
```

```
            for item_name in items:
                index_to_delete: typing.Optional[int] = None
                if from_pool:
                    try:
                        # If from_pool, try to find an existing item with this
name & player in the itempool and use it
                        index_to_delete, item = next(
                            (i, item) for i, item in
enumerate(multiworld.itempool)
                            if item.player == player and item.name == item_name
and i not in claimed_indices
                            )
                    except StopIteration:
                        warn(
                        f"Could not remove {item_name} from pool for
{multiworld.get_player_name(player)} as it\\\'s already missing from it.",
                        placement["force"])
                        item = multiworld.worlds[player].create_item(item_name)
                else:
                    item = multiworld.worlds[player].create_item(item_name)

                for location in reversed(candidates):
                    if (location.address is None) == (item.code is None):  #
either both None or both not None
                        if not location.item:
                            if location.item_rule(item):
                                if location.can_fill(multiworld.state, item,
False):
                                    successful_pairs.append((index_to_delete,
item, location))
                                    claimed_indices.add(index_to_delete)
                                    candidates.remove(location)
                                    count = count + 1
                                    break
                                else:
                                    err.append(f"Can\\\'t place item at
{location} due to fill condition not met.")
                            else:
                                err.append(f"{item_name} not allowed at
{location}.")
                        else:
                            err.append(f"Cannot place {item_name} into already
filled location {location}.")
                    else:
                        err.append(f"Mismatch between {item_name} and
{location}, only one is an event.")

                if count == maxcount:
                    break
            if count < placement["count"]["min"]:
                m = placement["count"]["min"]
                failed(
                    f"Plando block failed to place {m - count} of {m} item(s)
for {multiworld.get_player_name(player)}, error(s): {\\' \\'.join(err)}",
                    placement["force"])

            # Sort indices in reverse so we can remove them one by one
            successful_pairs = sorted(successful_pairs, key=lambda
successful_pair: successful_pair[0] or 0, reverse=True)

            for (index, item, location) in successful_pairs:
                multiworld.push_item(location, item, collect=False)
```

```
                location.locked = True
                logging.debug(f"Plando placed {item} at {location}")
                if index is not None:  # If this item is from_pool and was
found in the pool, remove it.
                    multiworld.itempool.pop(index)

        except Exception as e:
            raise Exception(
                f"Error running plando for player {player}
({multiworld.get_player_name(player)})") from e
```

7) Conclusão

A aplicação do TDD para a correção deste bug foi fundamental para garantir que a alteração no código não introduzisse novos problemas e que o comportamento esperado fosse restaurado. O ciclo de TDD, que envolve escrever o teste primeiro, ver ele falhar, implementar a funcionalidade para fazê-lo passar e, em seguida, refatorar, proporcionou um processo de desenvolvimento disciplinado e seguro.

No início, a dificuldade em configurar o ambiente de teste e mockar as dependências complexas do projeto (como `MultiWorld`, `Location` e `Item`) foi um desafio significativo. Isso ressalta a importância de uma arquitetura de código que facilite a testabilidade. No entanto, uma vez que os mocks foram configurados corretamente, o processo de TDD se tornou muito mais fluido. O teste falhando claramente indicou o problema, e a correção foi direcionada e verificada imediatamente pelo teste passando. A refatoração, embora não extensa neste caso, teria sido igualmente validada pelos testes, garantindo que as mudanças internas não alterassem o comportamento externo.

Em resumo, o TDD, apesar da curva de aprendizado inicial e da complexidade de setup em projetos legados ou com muitas dependências, oferece uma rede de segurança robusta, melhora a qualidade do código e facilita a manutenção. Ele força uma compreensão mais profunda do requisito (ou do bug) antes mesmo de escrever qualquer código de produção, resultando em soluções mais limpas e eficazes.

6) Link para o Pull Request (Opcional)

Como esta atividade foi realizada em um ambiente simulado, não foi possível criar um Pull Request real no repositório do GitHub. No entanto, em um cenário de desenvolvimento real, o próximo passo seria criar um Pull Request com as alterações

no código-fonte (`Fill.py`) e o novo teste (`test_bug_fix.py`) para revisão pela equipe do projeto.