Sample OOP Assignment

This assignment requires you to use Python 3.13, the `mypy` type-checker, and the `black` formatter. Below are the exact versions I will use:

```
pip install mypy=1.18.2
pip install black=25.9.0
```

1. Specification

In this assignment, you are asked to develop an object-oriented library to play the game of **Terraforming Mars**, see the description at:

https://fryxgames.se/product/terraforming-mars/

A PDF version of the rules has been provided.

1.1. Extent of the Implementation

It is not expected that you will implement the full game: You may implement a suitably simplified version of the game, or a self-contained fragment of the game.

The intent is to expose you to numerous opportunities to showcase your object-oriented design and implementation skills on a concrete, relatable example. The more logic you decide to implement, the more challenges you are likely to face, the more chances for you to display interesting patterns and solutions.

However much you decide to implement must be complete and fully working — stub/placeholder code is not acceptable. This will have to be balanced against some strict limits on the amount of code you can submit.

1.2. Your Library as an API

Your library should expose a class `Game` to its users, which can be instantiated to create a new game (with given initial parameters, if relevant). The public instance properties and methods of `Game` should allow the game to be played according to the rules and expose all necessary game information.

The `Game` class must provide the initial entry point to your library, known as its façade. Users should not have to instantiate any non-builtin class other than `Game` in order to use the API: Other classes should be used, but they should be made accessible by the public properties and methods of `Game`. For example, imagine that your library will sit behind an API endpoint on a server, and that a frontend will use it to manage online instances of the game.

Note that it must be possible to create multiple independent instances of `Game` at the same time (e.g. to use the library as part of an online game server). The user interface (console-based or otherwise) is out of scope for this assignment and will not contribute to its mark — please do not implement one.

1.3. Programmatic Gameplay

It must be possible to play the game programmatically and it must be possible to access all relevant game information programmatically. Failure to comply with these requirements will have a significantly negative impact on the outcome of your submission.

Your implementation follow an object-oriented design, delegating responsibilities to suitable sub-components whose public methods and properties allow the game to be played.

Common issues include:

- Printing information to console and/or requiring user input from console (or other UI).

- Exposing gameplay methods, but not giving access to the full game information.
- Inadequately delegating responsibility for action execution and/or information exposure to sub-components.

1.4. Encapsulation and Validation

Code executing illegal actions must not statically type-check, or it must otherwise raise an error at runtime. This includes playing illegal moves and other illegal modification of the game state/ information (i.e. if encapsulation is broken).

In other words, the public methods and properties of the `Game` class and any sub-components which it exposes must not allow illegal actions to be performed.

Common issues include:

- Methods can be successfully invoked with illegal parameters, or in illegal order.
- Mutable public properties allow illegal values to be set.
- Read-only properties expose objects which can themselves be illegally modified.

2. Assessment Criteria

The point of this assignment is to establish whether you can use an object-oriented approach to design libraries, algorithms and data structures, and whether you can implement them in a structured, type-safe, and re-usable way.

2.1. Structure

- Low-level data used by the library should be structured by means of builtin types, typed dictionaries, protocols, or dataclasses. You should avoid custom classes for any data without significant logic associated.
- High-level components used by the library should be structured through classes. Use public methods to provide access to external functionality, and protected/private methods to provide access to internal functionality.
- Functionality should be delegated to sub-components, in a balanced way: implementing most logic in the `Game` class is unlikely to earn you high marks.
- Your code should be clear and concise: long or complex method/function bodies should be avoided, by delegating responsibilities to private methods, helper functions, or sub-components.

2.2. Encapsulation

- Your design should make use of adequately chosen types, read-only properties, protected/private attributes and other forms of access control to ensure that the library cannot be misused.
- This should be balanced with the need for individual components to safely expose their information to your library's users. Access control so tight as to prevent the exposure of information is unlikely to earn you high marks.
- Attributes should not be public: you cannot restrict their writeability, and this most likely violates encapsulation.
- You must explicitly annotate all attributes used by your classes. For example, the `_items` attribute in the following example was explicitly annotated as `_items: list[str]` in the class body:

```python
class MyBagOfStrings:
    _items: list[str]
    def __new__(cls, items: Iterable[str]) -> Self:
        self = super().__new__(cls)
        self._items = list(items)
        return self
```

- You should not use class attributes to set default values. If you use class attributes, they should ordinarily not be public, and they should be annotated using `ClassVar`.

- Any public module variables should be constant (names are fully uppercased, as in `MY_CONSTANT`) and marked as `Final`.

2.3. Validation

- You must use `mypy --strict game` to statically type-check your code. Any errors returned by `mypy` will be heavily penalised.
- You should dynamically validate the values passed to your public functions and methods.
- You should dynamically validate the staging logic of your game.
- You should perform your dynamic validation at the start of the function/method, before performing any of the computations.

2.4. Object-oriented Patterns

- Where relevant, you should attempt to use object-oriented patterns. Acceptable patterns claims are from the creational, structural and behavioural patterns listed at https://en.wikipedia.org/wiki/Design_Patterns, with the exception of the singleton pattern. See Section 4 for the list.
- You should clearly indicate any object-oriented patterns used by a module/class in its docstring. Use a new line for each, writing `PATTERN` followed by the pattern's name. Marks will not be awarded for patterns documented in any other way.
- You must not use `PATTERN` to indicate patterns other than those allowed above. Misuse of the `PATTERN` indication will negatively impact the outcome of our submission.

2.5. Reusable Generic Data Structure

- Where relevant, you should attempt to implement reusable generic data structures (e.g. trees, graphs, queues/stacks, bags, etc). For marks to be awarded, they must be both reusable, i.e. not specific to your implementation, and generic, i.e. parametrically polymorphic.
- You should clearly indicate that a class implements a reusable generic data structure, by writing `DATASTRUCT` at the start of its docstring, on a separate line. Marks will not be awarded for reusable generic data structures documented in any other way.
- You must not use `DATASTRUCT` to indicate classes other than classes which (i) are generic and (ii) implement legitimately reusable data structures. Misuse of the `DATASTRUCT` indication will negatively impact the outcome of our submission.

2.6. Advanced Language Features

- Where relevant, you should attempt to use advanced type features (e.g. inheritance, composition, generics, method overloads, structural types).
- Where relevant, you should attempt to use advanced language features (e.g. comprehensions, iterators, dunder methods, function objects).

3. Submission

The submission instructions below are intended to standardise your submission, for fair assessment. Before submitting, please go through them point by point and ensure that you satisfy all requirements.

Failure to comply with these instructions **will be penalised**, with up to 10 marks deducted. Significant failure may result in your submission not being examined.

3.1. Packaging

- You must submit a single Python package called `game`, containing your full implementation. Files outside this package will not be marked, nor will any non-Python content in the package.
- Your code must be organised into modules, which must be `.py` files. That is, you must not use folders inside your `game` folder.
- Imports between your sub-modules must be relative, e.g. as `from .mymod import MyClass`.

- Your main package must contain a `__init__.py` file, which must export the `Game` class and nothing else.
- It must be possible to run `from game import Game`, without error, from a script placed in the same folder as your `game` package.
- All your modules must be public, i.e. their names cannot start by underscore.

3.2. Formatting

- You must use `black game` to format your code.
- After running `black game`, you must not exceed 2000 lines of code across all of your modules.
- After running `black game`, each individual module must not exceed 500 lines of code.
- You must not use `;` to put multiple statements on a line. This is bad practice generally, and I will interpret it as an attempt to circumvent the length restrictions.

The following code exemplifies how lines of code will be counted for a given file:

```
with open(file_path, "r", encoding="utf-8") as f:
    line_count = len(list(f))
```

3.3. Static Type-checking

- You must explicitly annotate all attributes of your classes.
- You must use `__new__` to define class constructors. You must not define `__init__` methods anywhere in your code.
- You must use `mypy --strict game` to ensure that there are no static type-checking errors.
- You must not use the `Any` type, the `object` type, or types in the form `Callable[..., T]`. Restructure your code instead.
- You must not use `#type: ignore` directives. Restructure your code instead.

3.4. Allowed Modules and Functions

- You must not import any modules, or members of any modules, other than the following:
  - Annotations: `from __future__ import annotations`
  - Types: `typing`, `types`, `collections.abc`, `abc`, `numbers`
  - Collections: `collections`, `heapq`, `array`, `bisect`, `graphlib`
  - Dataclasses and Enums: `dataclasses`, `enum`
  - Numeric: `fractions`, `decimal`, `math`, `cmath`, `random`
  - Strings: `re`, `string`, `unicodedata`
  - Time: `datetime`, `zoneinfo`, `calendar`
  - Utilities: `contextlib`, `functools`, `itertools`, `operator`, `weakref`
- You must not use any function which accesses the file system, such as `open`.
- You must not use any function which interacts with the command line, such as `print` or `input`.
- You must not use any function which can be used to execute arbitrary code, such as `compile`, `exec` or `eval`.
- You must not use any function which accesses object internals, such as `dir`, `globals`, `locals`, `vars`, `getattr`, `setattr`, `hasattr` or `delattr`.

4. Design Patterns

Below is the full list of design patterns allowed in this assignment, see https://en.wikipedia.org/wiki/Design_Patterns.

4.1. Creational

Creational patterns are ones that create objects, rather than having to instantiate objects directly. This gives the program more flexibility in deciding which objects need to be created for a given case.

Abstract factory groups object factories that have a common theme.
Builder constructs complex objects by separating construction and representation.
Factory method creates objects without specifying the exact class to create.
Prototype creates objects by cloning an existing object.

4.2. Structural

Structural patterns concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
Bridge decouples an abstraction from its implementation so that the two can vary independently.
Composite composes zero-or-more similar objects so that they can be manipulated as one object.
Decorator dynamically adds/overrides behaviour in an existing method of an object.
Facade provides a simplified interface to a large body of code.
Flyweight reduces the cost of creating and manipulating a large number of similar objects.
Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

4.3. Behavioural

Most behavioural design patterns are specifically concerned with communication between objects.

Chain of responsibility delegates commands to a chain of processing objects.
Command creates objects that encapsulate actions and parameters.
Interpreter implements a specialized language.
Iterator accesses the elements of an object sequentially without exposing its underlying representation.
Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
Memento provides the ability to restore an object to its previous state (undo).
Observer is a publish/subscribe pattern, which allows a number of observer objects to see an event.
State allows an object to alter its behaviour when its internal state changes.
Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behaviour.
Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.