Tutorial: Object-Oriented Programming (OOP) and Design Patterns in Your Parser Project

This guide explains object-oriented programming (OOP) and design patterns using examples from your parser project. It's written in Markdown and tailored to your code, avoiding generic examples like "Dog" since they don't apply to your work. Everything here connects directly to your project's components.

---

1. Introduction to OOP

What is OOP?

OOP organizes code around "objects" that combine data and actions. Think of it like a toolbox: each tool (object) has a purpose and works with others to get the job done.

Key Concepts

- **Class**: A template defining data (properties) and actions (methods).
- **Object**: An instance created from a class, like a specific tool built from a design.

Why Use OOP?

OOP keeps related things together, making your code easier to manage, reuse, and grow.

Where It's Used in Your Parser Project

Your project uses OOP in classes like `AppConfig` (settings), `FileManager` (file handling), and `GameParser` (log parsing).

How It's Used

Define a class, then create objects to use its features.

---

2. Types of Classes in Your Project

Your project uses different class types, each with a unique role.

2.1 Vanilla (Regular) Classes

What Are They? Regular classes hold data and actions together.

Why Use Them? They're great for objects that need to store info and do tasks, like a worker with a name and a job.

Where They're Used `FileManager` manages file operations.

How They're Used Define properties with `self` and add methods.

Example

```
class FileManager:
    def __init__(self, base_dir):
        self.base_dir = base_dir

    def load_text(self, path):
        full_path = f"{self.base_dir}/{path}"
        with open(full_path, 'r') as file:
            return file.read()

# Usage
manager = FileManager("/data")
text = manager.load_text("game_log.txt")
```

`FileManager` uses `base_dir` to load files.

2.2 Dataclasses

What Are They? Dataclasses store data with less code—Python handles the setup.

Why Use Them? Perfect for simple data containers, like a settings list.

Where They're Used `AppConfig` holds configuration details.

How They're Used Use `@dataclass` and list properties.

Example

```
from dataclasses import dataclass

@dataclass
class AppConfig:
    data_path: str
    seed: int
    output_format: str

# Usage
config = AppConfig(data_path="/data", seed=42, output_format="json")
```

`AppConfig` stores settings cleanly.

2.3 Abstract Classes

What Are They? Abstract classes are blueprints that can't be instantiated—they guide subclasses.

Why Use Them? They enforce rules, like requiring all exporters to have an `export` method.

Where They're Used `Exporter` defines a standard for export classes.

How They're Used Use `abc` and `@abstractmethod`.

Example

```python
from abc import ABC, abstractmethod

class Exporter(ABC):
    @abstractmethod
    def export(self, log, dest):
        pass

class JsonExporter(Exporter):
    def export(self, log, dest):
        print(f"Exporting to {dest} as JSON")

# Usage
exporter = JsonExporter()
exporter.export(log, "output.json")
```

`Exporter` ensures consistency.

2.4 Protocols

What Are They? Protocols specify methods a class must have, without requiring inheritance.

Why Use Them? They offer flexibility—any class with the right method fits.

Where They're Used `Transformer` lets any class with `transform` work as a transformer.

How They're Used Use `typing.Protocol`.

Example

```
from typing import Protocol

class Transformer(Protocol):
    def transform(self, data):
        pass

class TeamTransformer:
    def transform(self, data):
        return transformed_data

# Usage
def process(transformer: Transformer, data):
    return transformer.transform(data)

transformer = TeamTransformer()
result = process(transformer, raw_data)
```

`TeamTransformer` works because it has `transform`.

3. Types of Methods in Your Project

Methods are actions in a class, and your project uses three types.

3.1 Instance Methods

What Are They? Instance methods act on a specific object, using its data.

Why Use Them? They tie actions to an object's state.

Where They're Used `FileManager.load_text` reads files using `base_dir`.

How They're Used Use `self`.

Example

```
class FileManager:
    def __init__(self, base_dir):
        self.base_dir = base_dir

    def load_text(self, path):
        full_path = f"{self.base_dir}/{path}"
        with open(full_path, 'r') as file:
            return file.read()
```

```
# Usage
manager = FileManager("/data")
text = manager.load_text("game_log.txt")
```

load_text uses the object's base_dir.

3.2 Class Methods

What Are They? Class methods work on the class itself, not an object.

Why Use Them? They handle class-level tasks, like creating objects.

Where They're Used AppConfig.load_from_file builds an AppConfig from a file.

How They're Used Use @classmethod and cls.

Example

```
class AppConfig:
    @classmethod
    def load_from_file(cls, path):
        data = load_data(path)
        return cls(data["data_path"], data["seed"], data["output_format"])

# Usage
config = AppConfig.load_from_file("config.yaml")
```

load_from_file creates an instance.

3.3 Static Methods

What Are They? Static methods are helper functions inside a class, not tied to objects or the class.

Why Use Them? They group related utilities.

Where They're Used UniqueIdGenerator.generate_team_id makes IDs.

How They're Used Use @staticmethod.

Example

```
class UniqueIdGenerator:
    @staticmethod
    def generate_team_id(name):
        return f"team_{hash(name)}"

# Usage
team_id = UniqueIdGenerator.generate_team_id("Red Sox")
```

`generate_team_id` needs no object.

---

4. Design Patterns in Your Parser Project

Design patterns solve common problems—your project uses these.

4.1 Facade Pattern

What Is It? A facade simplifies a complex system with one easy interface.

Why Use It? It hides complexity, like a single button for a big task.

Where It's Used `GameFlowManager` runs the whole parsing process.

How It's Used Wrap steps in one method.

Example

```
class GameFlowManager:
    def run(self, source_path, outputs):
        raw_text = self.file_manager.load_text(source_path)
        normalized = self.normalizer.normalize(raw_text)
        self.game_parser.parse(normalized, outputs)

# Usage
manager = GameFlowManager()
manager.run("game_log.txt", ["json", "csv"])
```

`run()` does it all.

4.2 Registry Pattern

What Is It? A registry tracks objects by name for later use.

Why Use It? It lets you pick options dynamically.

Where It's Used `GameParser` registers exporters.

How It's Used Use a dictionary.

Example

```
class GameParser:
    def __init__(self):
        self.exporters = {}

    def register_exporter(self, name, exporter):
        self.exporters[name] = exporter

    def parse(self, log, formats):
        for fmt in formats:
            exporter = self.exporters[fmt]
            exporter.export(log, f"output.{fmt}")

# Usage
parser = GameParser()
parser.register_exporter("json", JsonExporter())
parser.parse(log, ["json"])
```

The registry manages exporters.

4.3 Composition

What Is It? Composition builds objects from smaller parts.

Why Use It? It keeps code modular.

Where It's Used `Normalizer` combines `Transformer` objects.

How It's Used Store parts as properties.

Example

```
class Normalizer:
    def __init__(self):
        self.steps = []

    def add_step(self, step):
        self.steps.append(step)

    def normalize(self, raw_text):
        data = raw_text
        for step in self.steps:
            data = step.transform(data)
        return data

# Usage
```

```
normalizer = Normalizer()
normalizer.add_step(TeamTransformer())
normalized = normalizer.normalize(raw_text)
```

`Normalizer` uses transformers together.

4.4 Strategy Pattern

What Is It? The strategy pattern swaps behaviors easily.

Why Use It? It makes actions flexible.

Where It's Used `Transformer` lets you swap processing steps.

How It's Used Define an interface and plug in options.

Example

```
class Transformer(Protocol):
    def transform(self, data):
        pass

class EventTransformer:
    def transform(self, data):
        return transformed_data

# Usage
normalizer.add_step(EventTransformer())
```

Swap transformers as needed.

4.5 Abstract Factory Pattern

What Is It? An abstract factory creates related objects.

Why Use It? It ensures consistency.

Where It's Used `Exporter` produces export types.

How It's Used Use an abstract class with subclasses.

Example

```
class Exporter(ABC):
    @abstractmethod
    def export(self, log, dest):
        pass
```

```python
class CsvExporter(Exporter):
    def export(self, log, dest):
        print(f"Exporting to {dest} as CSV")

# Usage
exporter = CsvExporter()
exporter.export(log, "output.csv")
```

`Exporter` standardizes exports.

---

5. Conclusion

What You've Learned

- **OOP**: Classes and objects organize code.
- **Classes**: Regular, dataclasses, abstract, protocols.
- **Methods**: Instance, class, static.
- **Patterns**: Facade, registry, composition, strategy, abstract factory.

In Your Project

These make your parser structured and adaptable.

Next Steps

- Add a new `Transformer`.
- Register another exporter.
- Run it all with `GameFlowManager`.

Keep building!