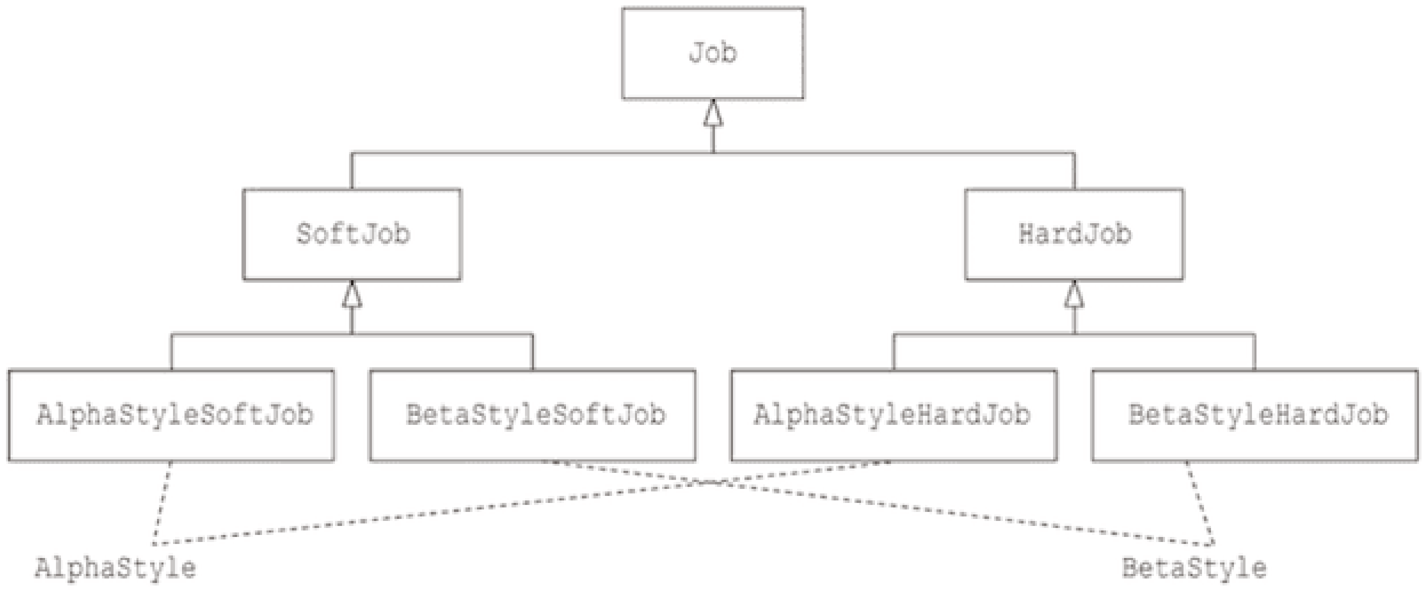
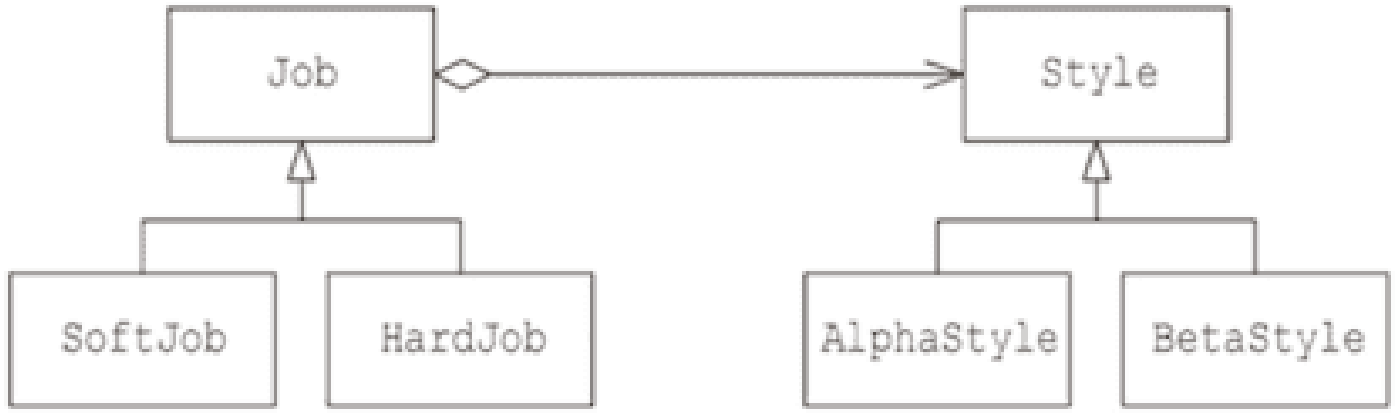
Tease Apart Inheritance

Split an **overloaded inheritance hierarchy** into **separate hierarchies**, linked by **composition** or **delegation**.

- In this example, some sub classes are related to AlphaStyle, and some to BetaStyle.



- In this case, we should separate the subclasses and connect them using delegation.



We call this **Bridge DP**.

Example: CSV Reader

- We have a CSV Reader Interface.

```
class CSVReader(ABC):
    CSV_PATTERN = re.compile(r'\s*,\s*')
    @abstractmethod
    def read_csv(self) -> Optional[List[str]]:
        pass
    @abstractmethod
    def close(self):
        pass
```

Requirements

> As a user, I want to read CSV files so that I can access information in the CSV format files.

```
class CSVFileReader(CSVReader):
    def __init__(self, filename: str):
        self.file = open(filename, 'r')
    def read_csv(self) -> Optional[List[str]]:
        line = self.file.readline()
        if not line:
            return None
        line = line.strip()
        return CSVReader.CSV_PATTERN.split(line)
    def close(self):
        self.file.close()
```

> As a user, I want to read CSV strings so that I can access information in the CSV strings.

```python
class CSVStringReader(CSVReader):
    """CSV string reader factory (after refactoring)"""

    def __init__(self, string: str):
        string_io = StringIO(string)
        super().__init__(string_io)
```

Added Requirements

> As a user, I want to print CSV files in tree format so that I can show the CSV file structure in a tree.

We can inherit from existing CSVFileReader.

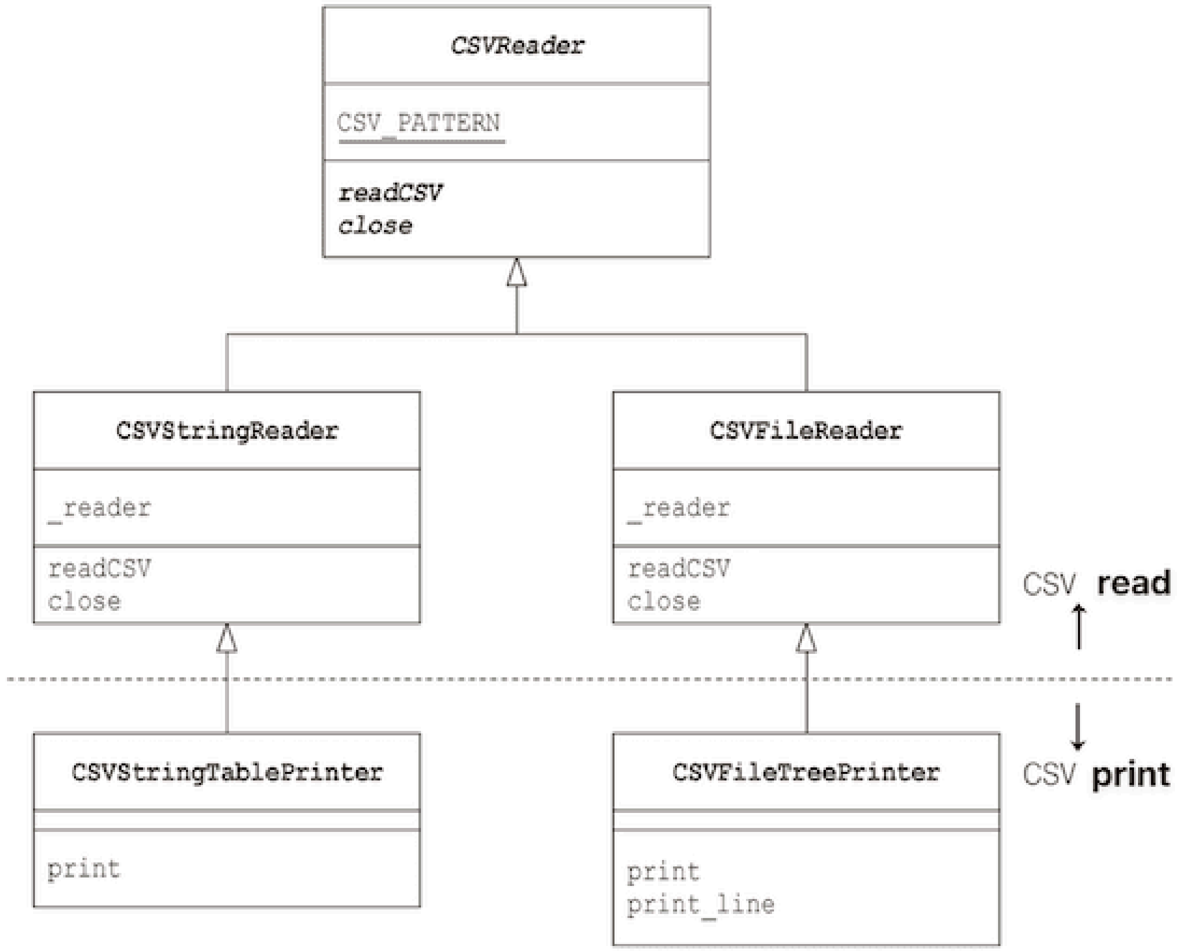
```python
class CSVFileTreePrinter(CSVFileReader):
    def __init__(self, filename: str):
        super().__init__(filename)
    def print(self):
        prev_item = []
        row = 0
        ...
```

> As a user, I want to print CSV strings in string table format so that I can show the CSV strings in a table.

We can inherit from existing CSVStringReader.

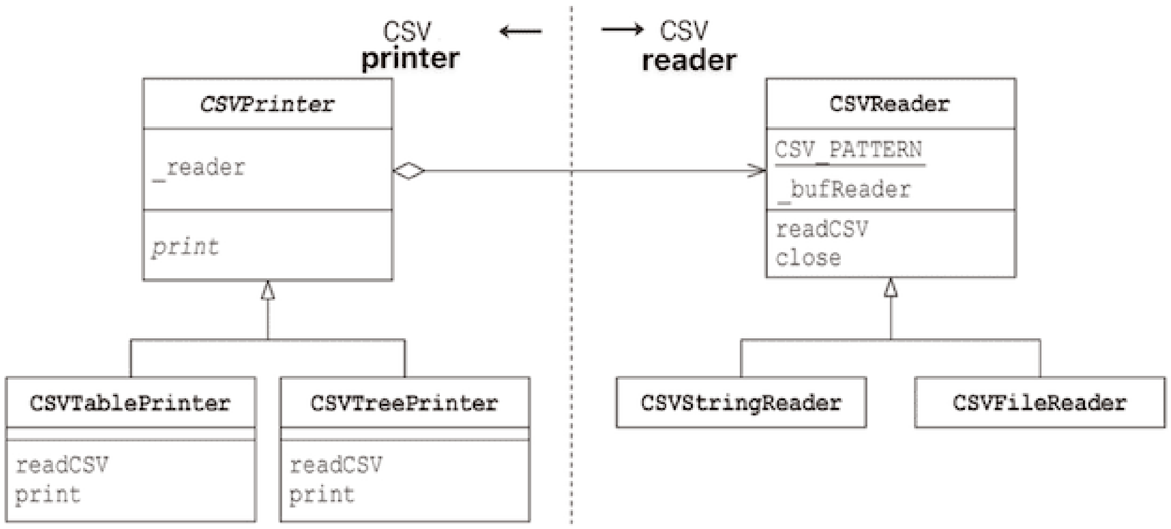
```python
class CSVStringTablePrinter(CSVStringReader):
    def __init__(self, string: str):
        super().__init__(string)
    def print(self):
        print("<table>")
        row = 0
        ...
```

- However, the inheritance is mixed: it's hard to debug and add features.



Refactoring

- We need to refactor using tease apart (untangle) inheritance.



CSVReader

- The first step is to tease apart the Readers.

```python
class CSVReader:
    CSV_PATTERN = re.compile(r'\s*,\s*')
    def __init__(self, reader: TextIO):
        self.reader = reader
    def read_csv(self) -> Optional[List[str]]:
        line = self.reader.readline()
        if not line:
            return None
        line = line.strip()
        return CSVReader.CSV_PATTERN.split(line)
    def close(self):
        self.reader.close()
```

CSV File and String Readers

```python
class CSVFileReader(CSVReader):
    def __init__(self, filename: str):
        file_handle = open(filename, 'r')
        super().__init__(file_handle)
```

```python
class CSVStringReader(CSVReader):
    def __init__(self, string: str):
        string_io = StringIO(string)
        super().__init__(string_io)
```

CSVPrinter

- The next step is tease apart the printers.
- We don't inherit the CSVReader, but aggregate it.

```python
class CSVPrinter(ABC):
    def __init__(self, csv_reader: CSVReader):
        self.csv_reader = csv_reader
    @abstractmethod
    def print(self):
        """Print the CSV data"""
        pass
```

CSV String and Table Printers

- We don't inherit the CSVReader, but aggregate it.

```python
class CSVTreePrinter(CSVPrinter):
    def __init__(self, csv_reader: CSVReader):
        super().__init__(csv_reader)
    def read_csv(self) -> Optional[List[str]]:
        return self.csv_reader.read_csv()
    def print(self):
        prev_item = []
        row = 0
        ...
            prev_item = item
            row += 1
    def _print_line(self, indent: int, s: str):
        print("    " * indent + s)
```

```python
class CSVTablePrinter(CSVPrinter):
    def __init__(self, csv_reader: CSVReader):
        super().__init__(csv_reader)
    def read_csv(self) -> Optional[List[str]]:
        return self.csv_reader.read_csv()
    def print(self):
        print("<table>")
        row = 0
        ...
            print("</tr>")
            row += 1
        print("</table>")
```

After the Refactoring

Now, we have the classes

- CSVReader
    - CSVFileReader
    - CSVStringReader
- CSVPrinter
    - CSVTablePrinter
    - CSTreePrinter

These classes enhance orthogonality so that each class can be easily used for the super class of other classes.

Relationship to Design Patterns

- This is how we refactor to use the `Bridge Design Pattern.`

Discussion

Benefits

1. **Single Responsibility** - each hierarchy handles one concern
2. **Reduces duplication** - eliminates duplicate code across branches
3. **Better extensibility** - can vary each dimension independently
4. **Composition flexibility** - can combine different aspects at