Object-Oriented Programming

Design Pattern Composite

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Directory

```
dir
├── a.cpp
├── b.cpp
└── c.cpp
```

Attempt #1: Design

```
dir
├── a.cpp
├── b.cpp
└── c.cpp
```

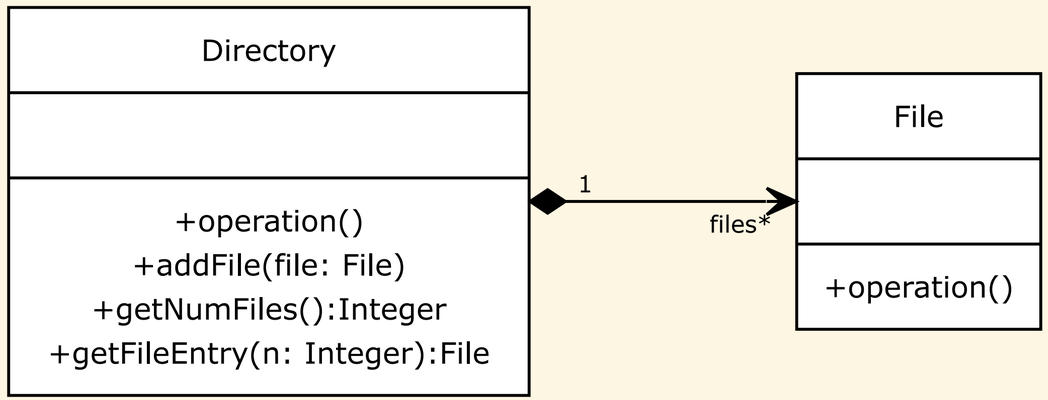
```cpp
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n);
private:
    std::vector<File> files;
};
```

Attempt #1: Usage

```
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n);
private:
    std::vector<File> files;
};
```

```
// create structure in memory
Directory dir("dir");
dir.addFile(File("a.cpp"));
dir.addFile(File("b.cpp"));
dir.addFile(File("c.cpp"));

// process directory
for (int i = 0; i < dir.getNumFiles(); ++i) {
    dir.getFileEntry(i).operation();
}
```

Attempt #1: UML




```cpp
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n);
private:
    std::vector<File> files;
};
```

Directory

```
dir
└── src
    ├── a.cpp
    ├── b.cpp
    └── c.cpp
```

Attempt #2: Design

```
dir
└── src
    ├── a.cpp
    ├── b.cpp
    └── c.cpp
```

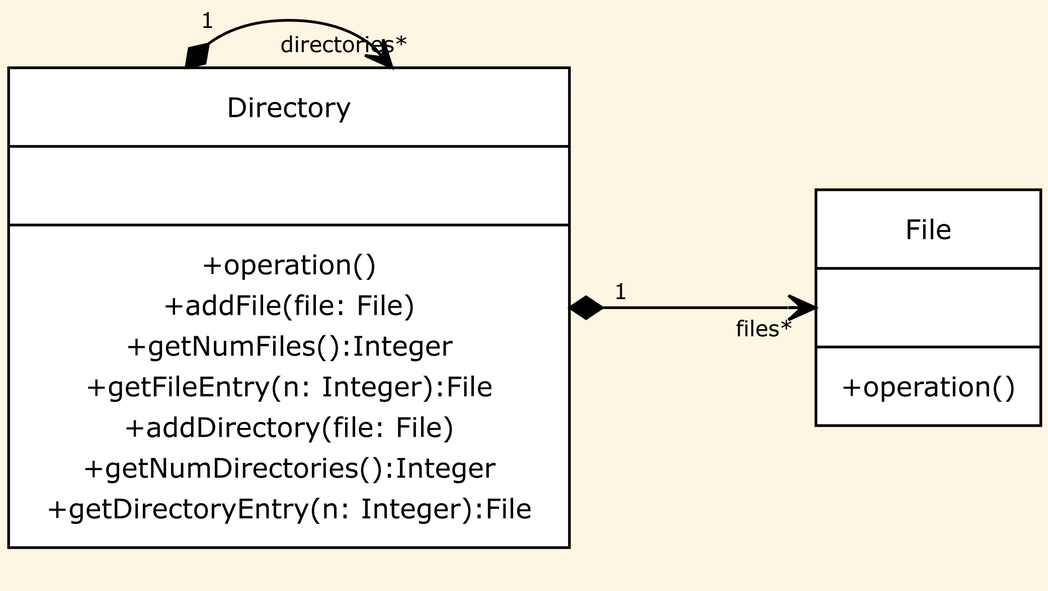
```cpp
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n) const;
    void addDirectory(const Directory& Directory);
    int getNumDirectories() const;
    Directory getDirectoryEntry(int n) const;
private:
    std::vector<File> files;
    std::vector<Directory> directories;
};
```

Attempt #2: Usage

```
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n) const;
    void addDirectory(const Directory& Directory);
    int getNumDirectories() const;
    Directory getDirectoryEntry(int n) const;
private:
    std::vector<File> files;
    std::vector<Directory> directories;
};
```

```
// create structure in memory
Directory dir("dir");
Directory src("src");
dir.addDirectory(src);
src.addFile(File("a.cpp"));
src.addFile(File("b.cpp"));
src.addFile(File("c.cpp"));

// process files in directory
for (int i = 0; i < dir.getNumFiles(); ++i) {
    dir.getFileEntry(i).operation();
}
// process subdirectories
for (int i = 0; i < dir.getNumDirectories(); ++i) {
    dir.getDirectoryEntry(i).operation();
}
```

Attempt #2: UML




```cpp
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n) const;
    void addDirectory(const Directory& Directory);
    int getNumDirectories() const;
    Directory getDirectoryEntry(int n) const;
private:
    std::vector<File> files;
    std::vector<Directory> directories;
};
```

Problems

```
class File {
public:
    File(std::string_view name);
    void operation();
};

class Directory {
public:
    Directory(std::string_view name);
    void operation();
    void addFile(const File& file);
    int getNumFiles() const;
    File getFileEntry(int n) const;
    void addDirectory(const Directory& Directory);
    int getNumDirectories() const;
    Directory getDirectoryEntry(int n) const;
private:
    std::vector<File> files;
    std::vector<Directory> directories;
};
```

- Have to store files and directories separately and keep them ordered separately
- Client code has to treat file and directory objects differently
- Complex code for the client to handle recursive structure

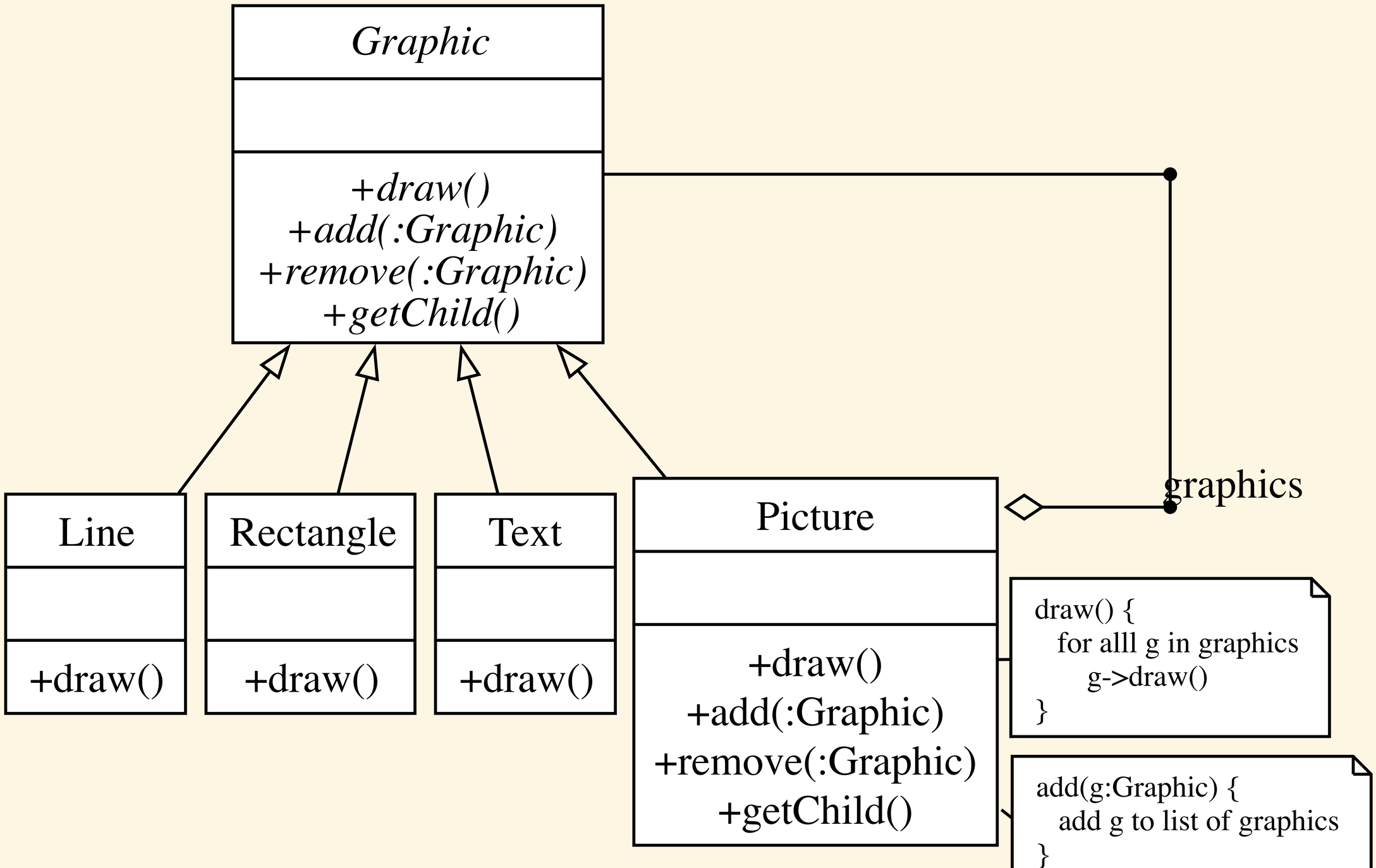
Composite

Compose objects into tree structures to represent part-whole hierarchies

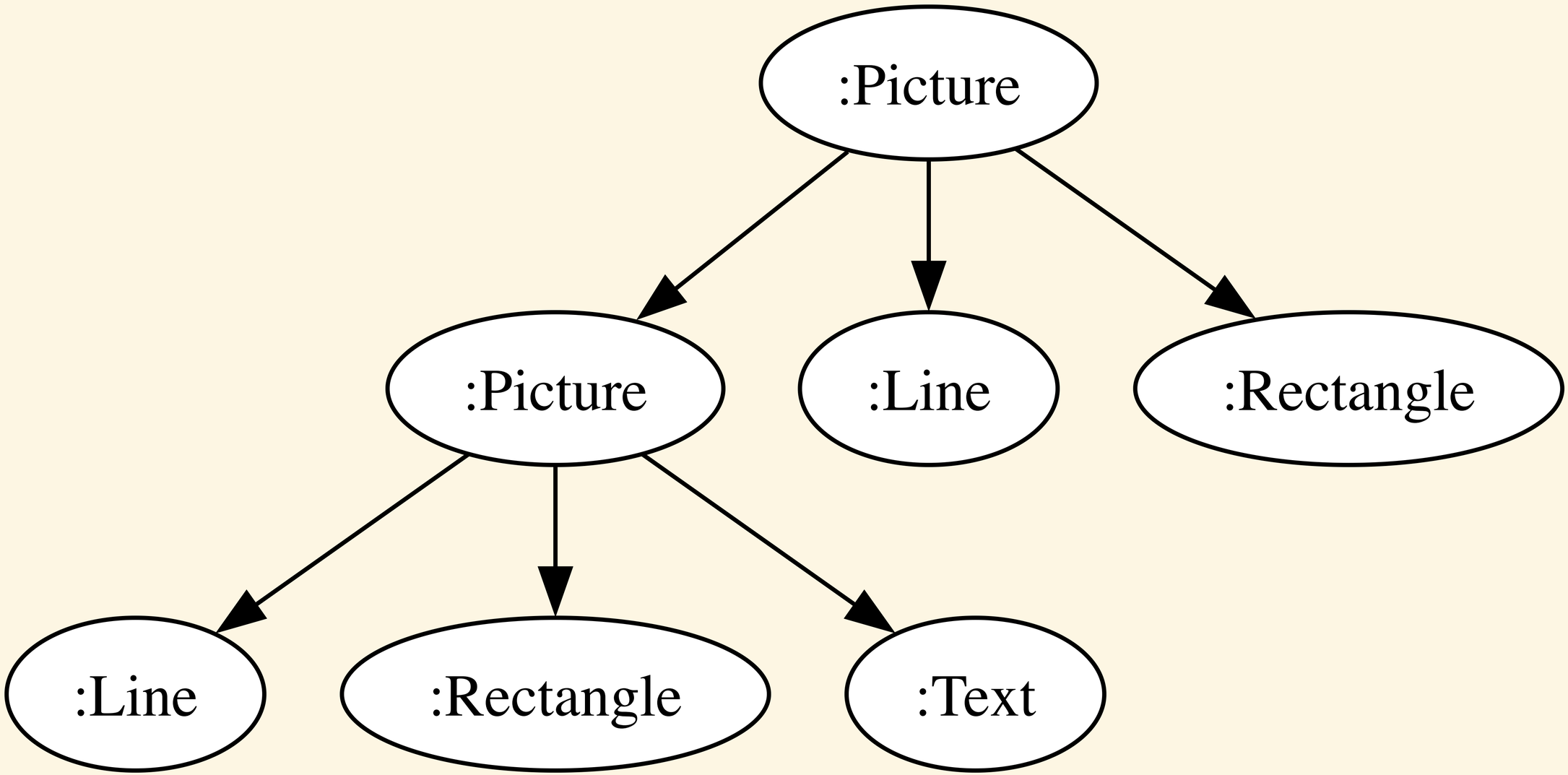
Composite lets clients treat individual objects and compositions of objects uniformly

- Structural Pattern

Composite: Motivation



Object Tree



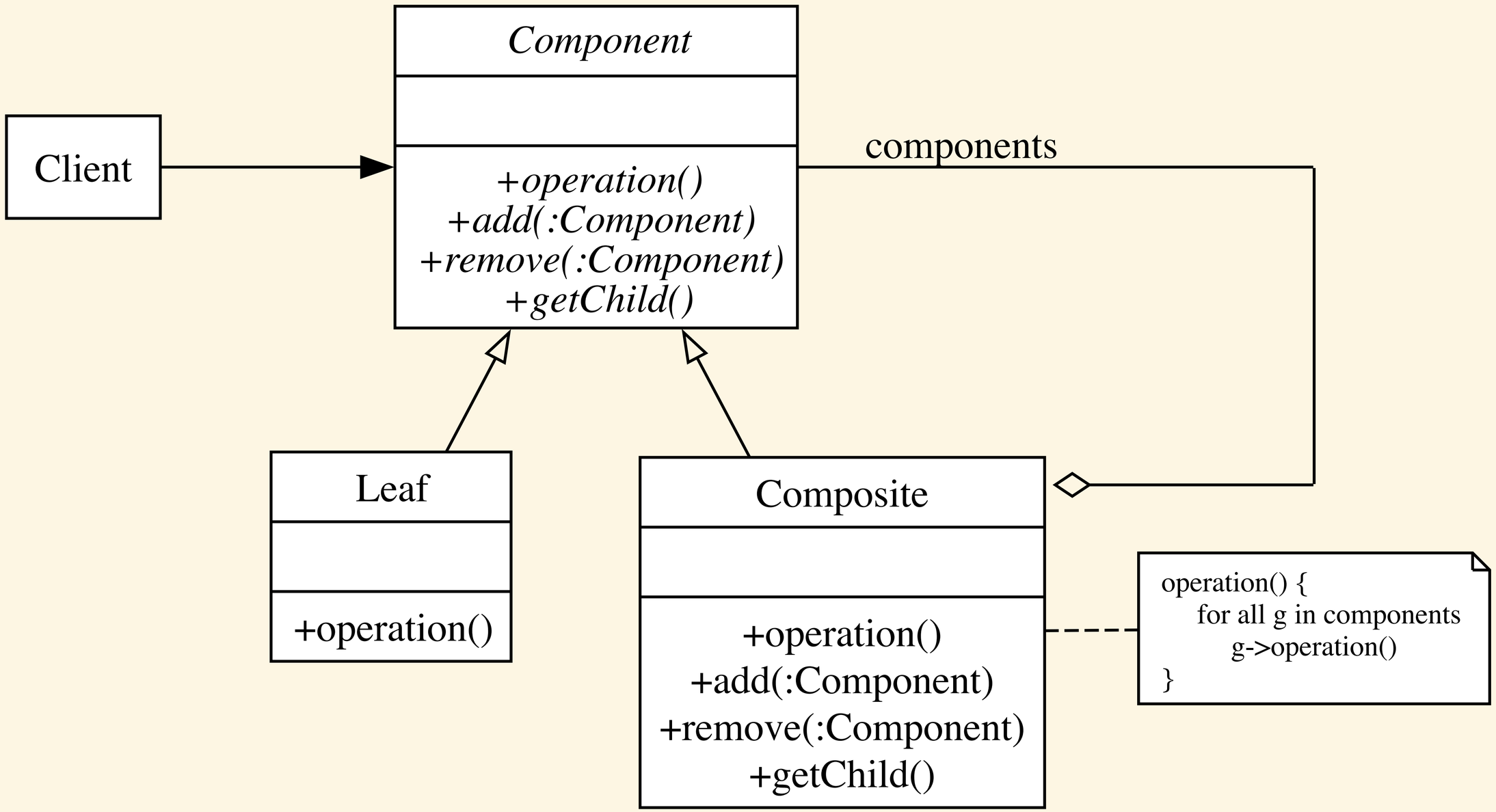
Composite: Applicability

- Use the Composite pattern when you:

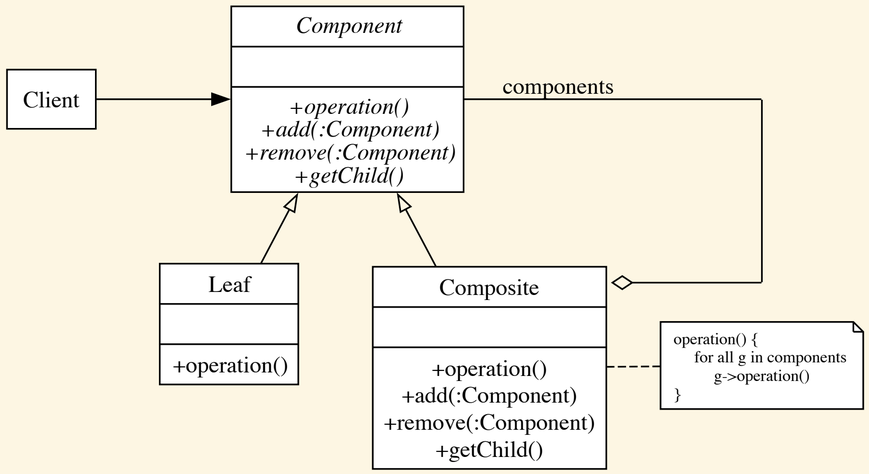
  Want to represent part-whole hierarchies of objects

  Want clients to be able to ignore the difference between compositions of objects and individual objects, as clients treat all objects in the composite structure uniformly

Composite: Structure



Composite: Participants



- Component (e.g., Graphic)

  Declares the interface for objects in the composition

  Implements the default behavior for the interface common to all classes as appropriate

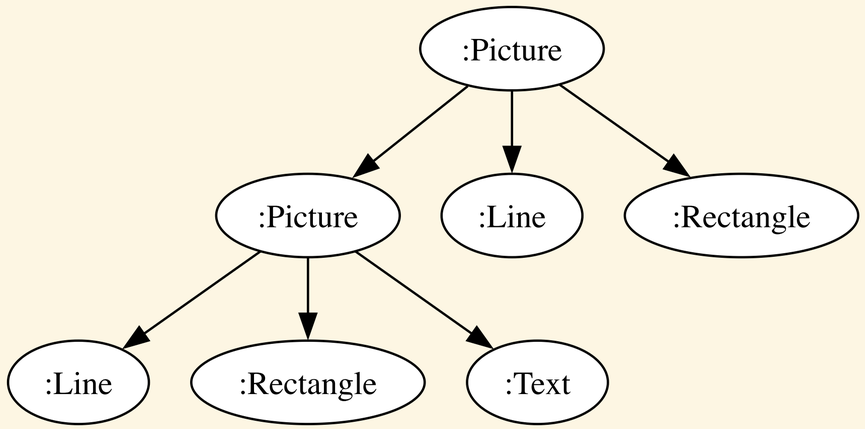
  Declares an interface for accessing and managing its child components

  Optionally defines an interface for accessing a component’s parent

- Client

  Manipulates objects in the composition through the Component interface

Composite: Object Participants



- Leaf (e.g., Rectangle, Line, Text, etc.)

  Represents leaf object (leaves have no children) in the composition

  Defines behavior for primitive objects in the composition

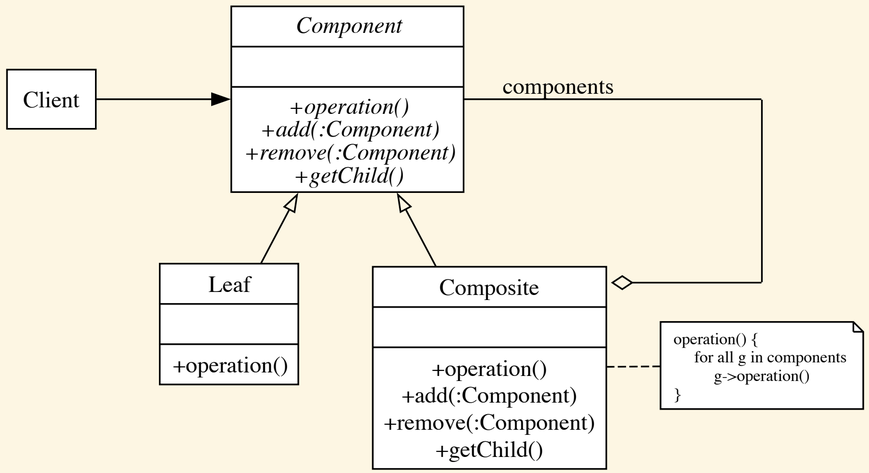
- Composite (e.g., Picture)

  Defines behavior for components having children

  Stores child components

  Implements child-related operations in the Component interface

Composite: Collaborations

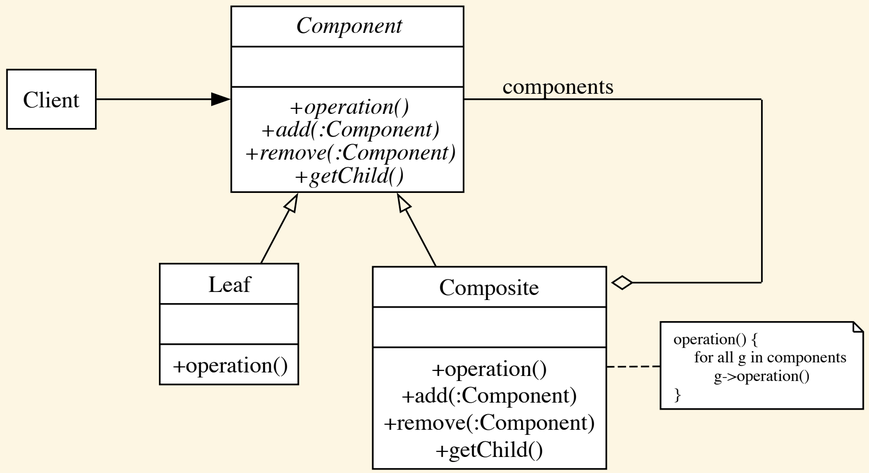


- Clients use the Component class interface to interact with objects in the composite structure:

- If the recipient is a Leaf, then the request is handled directly

- If the recipient is a Composite, it usually forwards requests to its child components, possibly performing additional operations before and after forwarding.

Composite: Consequences



- A class hierarchy of primitive and composite objects that permits recursive structures
- Clients are simple as they handle primitive and composite objects in the same way
- Can add new kinds of components, both Composite or Leaf, that work automatically with existing client code
- However, being able to add any Component type can make the design overly general

Composite #1: Design

```
class Component {
public:
    Component(std::string_view name)
        : name(name) {}
    virtual void operation() = 0;
    virtual ~Component() = default;
private:
    std::string name;
};

class File : public Component {
public:
    File(std::string_view name)
        : Component(name) {}
    void operation() override {}
    ~File() = default;
};
```

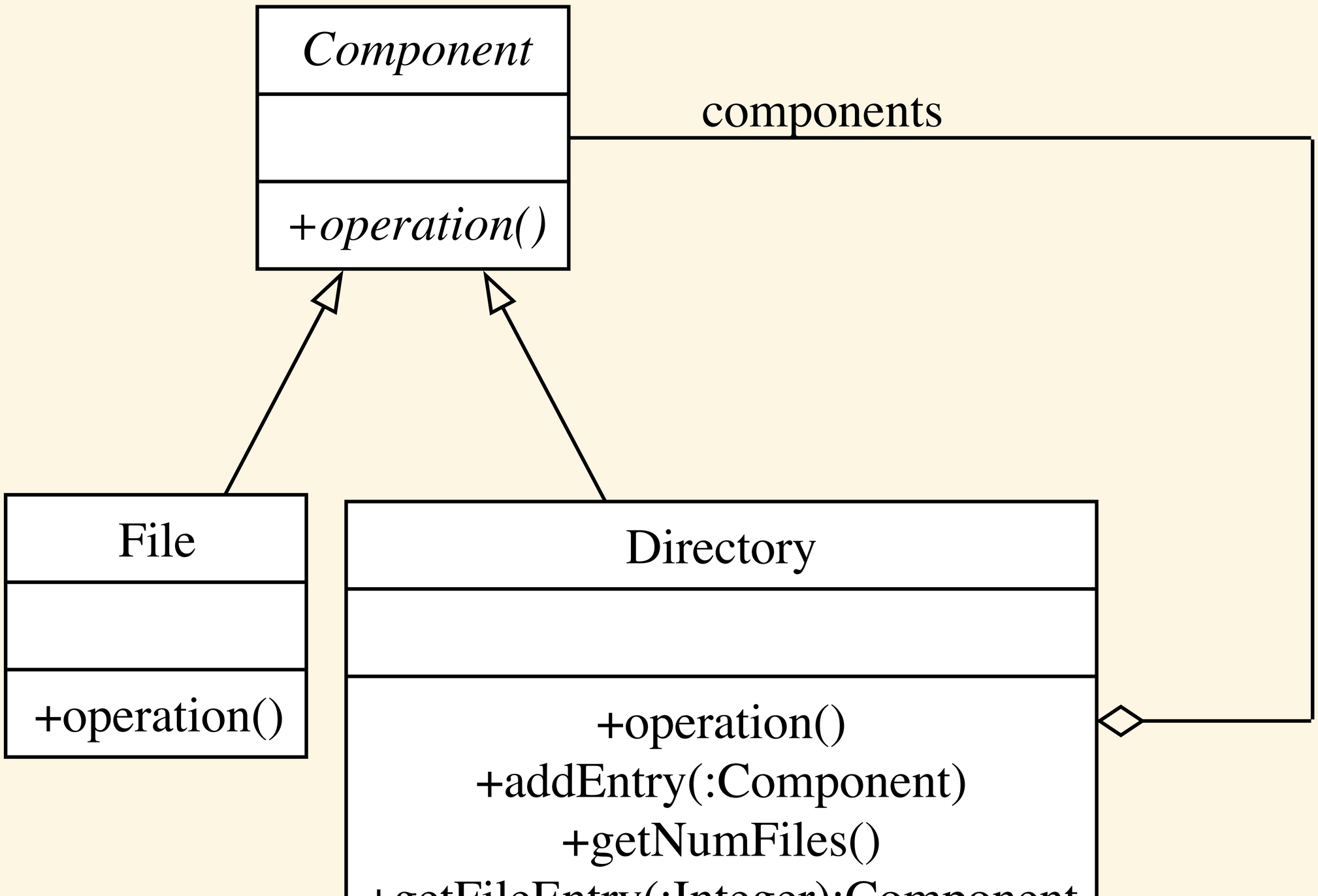
```
class Directory : public Component {
public:
    Directory(std::string_view name)
        : Component(name) {}
    void operation() override {
        for (auto entry : entries) {
            entry->operation();
        }
    }
    void addEntry(Component* Component) {
        entries.push_back(Component);
    }
    ~Directory() {
        for (auto entry : entries) {
            delete entry;
        }
    }
private:
    std::vector<Component*> entries;
};
```

Composite #1: Usage

```cpp
class Component {
public:
    Component(std::string_view name)
        : name(name) {}
    virtual void operation() = 0;
    virtual ~Component() = default;
private:
    std::string name;
};

class File : public Component {
public:
    File(std::string_view name)
        : Component(name) {}
    void operation() override {}
    ~File() = default;
};

class Directory : public Component {
public:
    Directory(std::string_view name)
        : Component(name) {}
    void operation() override {
        for (auto entry : entries) {
            entry->operation();
        }
    }
    void addEntry(Component* Component) {
        entries.push_back(Component);
    }
    ~Directory() {
        for (auto entry : entries) {
            delete entry;
        }
    }
private:
    std::vector<Component*> entries;
};
```

```cpp
Directory dir("dir");
Directory* subdir(new Directory("src"));
dir.addEntry(subdir);
subdir->addEntry(new File("a.cpp"));
subdir->addEntry(new File("b.cpp"));
subdir->addEntry(new File("c.cpp"));

// perform operation on directory tree
dir.operation();
```

Composite UML



Composite #2: Design

```cpp
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() = default;
};

class File : public Component {
public:
    File(std::string_view name) {}
    void operation() override {}
};
```

```cpp
class Directory : public Component {
public:
    Directory(std::string_view name) {}
    void operation() override {
        for (auto& entry : entries) {
            entry->operation();
        }
    }
    void addEntry(std::unique_ptr<Component> Component) {
        entries.push_back(std::move(Component));
    }
private:
    std::vector<std::unique_ptr<Component>> entries;
};
```

Composite #2: Usage

```
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() = default;
};

class File : public Component {
public:
    File(std::string_view name) {}
    void operation() override {}
};

class Directory : public Component {
public:
    Directory(std::string_view name) {}
    void operation() override {
        for (auto& entry : entries) {
            entry->operation();
        }
    }
    void addEntry(std::unique_ptr<Component> Component) {
        entries.push_back(std::move(Component));
    }
private:
    std::vector<std::unique_ptr<Component>> entries;
};
```

```
Directory dir("dir");
std::unique_ptr<Directory> subdir(new Directory("src"));
subdir->addEntry(std::unique_ptr<Component>(new File("a.cpp")));
subdir->addEntry(std::unique_ptr<Component>(new File("b.cpp")));
subdir->addEntry(std::unique_ptr<Component>(new File("c.cpp")));
dir.addEntry(std::move(subdir));

// perform operation on directory tree
dir.operation();
```

Implementation Issues

```cpp
class Component {
public:
    virtual void operation() = 0;
    virtual ~Component() = default;
};

class File : public Component {
public:
    File(std::string_view name) {}
    void operation() override {}
};

class Directory : public Component {
public:
    Directory(std::string_view name) {}
    void operation() override {
        for (auto& entry : entries) {
            entry->operation();
        }
    }
    void addEntry(std::unique_ptr<Component> Component) {
        entries.push_back(std::move(Component));
    }
private:
    std::vector<std::unique_ptr<Component>> entries;
};
```

- Storing parent references
- Sharing components
- Size and contents of Composite interface, e.g., traversing children
- Child management
- Should Component implement a list of Components?
- Child ordering
- Caching to improve performance
- Who should delete components?
- What's the best data structure for storing components?

Related Patterns

- *Composite*

  Used to implement MacroComposites

- *Memento*

  Stores the state the Composite requires for an undo

- *Prototype*

  Composite objects are used in Prototypes