Design Patterns (Composite)

Reading: Freeman, Robson, Bates, Sierra, *Head First Design Patterns*, O'Reilly Media, Inc. 2004
Ch 9: Composite and Iterator Patterns
Electronic text available from UW Library Web site
Today's Agenda

**Design patterns**: codified solutions that put design principles into practice, to improve the modularity of our code.

### OO Basics
- Separation of Concerns
- Encapsulate what is likely to change
- Encapsulate Data Representation
- Abstraction (interfaces, ADTs)
- Reuse (through composition, inheritance)
- Polymorphism

### OO Principles
- Open Closed Principle
- Favour Composition over Inheritance
- Single Responsibility Principle
- Dependency Inversion Principle
- Liskov Substitutability Principle
- Law of Demeter

### Design Patterns
- Strategy
- Template Method
- Adaptor
- Facade
- Observer
- Model-View-Controller (MVC)
- Composite
- Iterator
A **compound object** represents a composition of heterogeneous, possibly recursive, component objects.

**Law of Demeter**: client code interacts with compound object.
The Composite Pattern takes a different approach: gives the client access to all member types in a compound object via a uniform interface.
Metaphor: Composite Pattern

collection of devices, each with its own interface

Universal Interface for (recursive) collection of devices
Composite Pattern

Problem: composite object consists of several heterogenous parts
Client code is complicated by knowledge of object structure.
Client must change if data structure changes.

Solution: create a uniform interface for the object's components.
Interface advertises all operations that components offer.
Client deals only with the new uniform interface.
Uniform interface is the union of the components' services.
Example

U Waterloo CS247 (Spring 2017) — p.7/20
Uniform Interface

class TeamMember {
public:
    virtual ~TeamMember() {}  

    // leaf-only operations
    virtual int salary() const { return 0; }

    // component-only operations
    virtual void add(TeamMember*) {} 
    virtual TeamMember* getMember(int) const 
    { return nullptr; }

    // shared operations
    virtual void print() const { std::cout << name_; } 

protected:
    TeamMember( const std::string& name );

private:
    std::string name_; 
};
The leaf classes override the behaviour of leaf-object operations.

class Developer : public TeamMember {
public:
    Developer( const std::string&, int);

    // redefine leaf-only operations
    virtual int salary() const { return salary_; }

    // inherit component-only operations

    // redefine shared operations
    virtual void print() const;

private:
    int salary_;
Concrete Composite Class

class Team : public TeamMember {
public:
    Team ( const std::string& );
    virtual ~Team();

    // inherit leaf-only operations
    // redefine component-only operations
    virtual void add(TeamMember*);
    virtual TeamMember* getMember(int) const;

    // redefine shared operations
    virtual void print() const;

private:
    vector<TeamMember*> members_;
};

TeamMember* Team::getMember(int i) const {
    return members_.at(i);
}
Uniformity vs. Safety

Whether to include component-specific operations in the component interface involves a trade-off between:

**uniformity** - preserving the illusion that component objects can be treated the same way.
   - Promoted by the **Composite Design Pattern**.

**safety** - avoiding cases where the client attempts to do something meaningless, like adding components to Leaf objects.
   - Promoted by **Liskov Substitutability Principle**.
Another Example: Expressions

Expressions:
- \( a + b \)
- \( a \times b + c - d \)
- \( a \)
Consequences:

- Client deals only with the new uniform interface.
- New leafs and composite types are easy to add.
- New operations are harder to add (Visitor Pattern).

How can client code iterate through a composite object without knowing the composite's structure?
Recap: Composite Pattern

Problem: composite object consists of several heterogenous parts. Client code is complicated by knowledge of object structure. Client must change if data structure changes.

Solution: create a uniform interface for the object's components. Interface advertises all operations that components offer. Client deals only with the new uniform interface. Uniform interface is the union of the components' services.
What's the story? First you tell us Single Responsibility and Substitutability, and now you are giving us a pattern where one class manages the operations of two completely different subclasses.
When Should I Use the Composite Pattern?

What's the story? First you tell us Single Responsibility and Substitutability, and now you are giving us a pattern where one class manages the operations of two completely different subclasses.

When client mostly treats the structure uniformly
- client usually ignores differences in element types
- mostly transverses entire composition
- there exist reasonable default implementations of operations
When Should I Use the Composite Pattern?

What's the story? First you tell us Single Responsibility and Substitutability, and now you are giving us a pattern where one class manages the operations of two completely different subclasses.

When client mostly treats the structure uniformly
- client usually ignores differences in element types
- mostly transverses entire composition
- there exist reasonable default implementations of operations

Alternative is to keep element types distinct
- all calls to inappropriate operations are caught
When Should I Use the Composite Pattern?

What's the story? First you tell us Single Responsibility and Substitutability, and now you are giving us a pattern where one class manages the operations of two completely different subclasses.

When client mostly treats the structure uniformly
- client usually ignores differences in element types
- mostly traverse entire composition
- there exist reasonable default implementations of operations

Alternative is to keep element types distinct
- all calls to inappropriate operations are caught

skeptic
Summary

The goal of design patterns is to encapsulate change.

**Facade Pattern**: encapsulates a collection of (complex?) classes.

**Composite Pattern**: encapsulates the structure of a heterogeneous, possibly recursive data structure.