Object-Oriented Design

EECS 211

Winter 2018
Design patterns are common solutions to common object-oriented design problems
Some design patterns

Flyweight a factory returns small objects that share state

Singleton a class allows for only one instance

Adapter an class adapts an object from one interface to another

Builder instead of taking all the constructor arguments at once, a class provides an API for assembling the object piece by piece

Composite single objects and groups of objects are treated alike via an interface

Bridge each object has a pointer to a separate implementation, allowing each to vary independently
Flyweight Pattern example

Context: In a compiler, should variable names be strings?
Flyweight Pattern example

Context: In a compiler, should variable names be strings?
Problem: Strings are slow to compare.
Flyweight Pattern example

Context: In a compiler, should variable names be strings?

Problem: Strings are slow to compare.

Solution: Use symbols (pointers to strings), and ensure that for any given string value there is only one pointer.
Flyweight Pattern example

Context: In a compiler, should variable names be strings?

Problem: Strings are slow to compare.

Solution: Use symbols (pointers to strings), and ensure that for any given string value there is only one pointer.

This is called “interning”
class symbol
{
    public:
        const std::string& name() const;
        
        bool operator==(const symbol& that) const
        { return ptr_ == that.ptr_; }
    
    private:
        std::shared_ptr<std::string> ptr_;  
};

Interning strings: symbol class
class Symbol_table
{
public:
    symbol intern(const std::string&);

private:
    unordered_map<string, shared_ptr<string>> table_;
Problem: multiple symbol tables?

Problem: Interning only works if every time we intern a string, we intern it in the same table
Problem: multiple symbol tables?

Problem: Interning only works if every time we intern a string, we intern it in the same table

Solution: Make the symbol table class a singleton

- Make its constructor and destructor private
- Require accessing it through a static member function
Problem: multiple symbol tables?

Problem: Interning only works if every time we intern a string, we intern it in the same table

Solution: Make the symbol table class a singleton

How?:

- Make its constructor and destructor private
- Require accessing it through a static member function
Singleton symbol table class

class Symbol_table
{
public:
    symbol intern(const std::string&);
    static Symbol_table& instance();

private:
    Symbol_table();
    ~Symbol_table();
    unordered_map<string, shared_ptr<string>> table_;}

To CLion!

See symbol.h.
Adapter Pattern

**Client**

**Target**
- + f()

**Adaptee**
- + g()

**Adapter**
- - adaptee: Adaptee
- + f()
bit_io: Adapter Pattern

**Client**

```
bostringstream
- buf: string
+ write()
```

```
bostream
+ write()
```

```
bofstream
```

```
bofstream-adapter
- adaptee: ostream
+ write()
```

```
ostream
+ write()
```

---

11
To CLion!

See bit_io.h.
The telescoping constructor anti-pattern

class Pizza
{
    public:
    ...;
    Pizza();
    explicit Pizza(crust_t, sauce_t = sauce_t::regular);
    Pizza(crust_t crust,
          sauce_t left_sauce,
          const vector<topping_t>& left_tops,
          sauce_t right_sauce,
          const vector<topping_t>& right_tops);
    ...;
};
Solution: The Builder Pattern

class Pizza
{
public:

…

class Builder
{
public:

Builder& crust(crust_t);
Builder& sauce(sauce_t, side_t = side_t::both);

…

Pizza build() const;

…

};


To CLion!

See pizza.h.
The Composite Pattern

- **Component**
  - + operation()
  - 0..* children

- **Leaf**
  - + operation()

- **Composite**
  - + operation()
Composite example: string matchers

**Client**

**IMatcher**
- matches()
- describe()

**EndsWith**
- pat: string
  - matches()
  - describe()

**Contains**
- pat: string
  - matches()
  - describe()

**Conjunction**
- left: IMatcher
- right: IMatcher
  - matches()
  - describe()
To CLion!

See matcher.h.
Composite example: renderables

- **Client**
  - **Renderable**
    - + `sample()`
    - + `get_bbox()`
  - *layers*
  - **Circle**
    - - `center`
    - - `radius`
    - - `color`
    - + `sample()`
  - **Rectangle**
    - - `color`
    - + `sample()`
  - **Overlay**
    - - `layers`
    - + `sample()`
To CLion!

See renderable.h.
## Vehicles: a class family varying along two axes

<table>
<thead>
<tr>
<th>Power</th>
<th>Medium</th>
<th>Land</th>
<th>Water</th>
<th>Air</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>gas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>electric</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nuclear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X-57 Maxwell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nuclear sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Nested generalization
Nested generalization

Vehicle

- LandV
  - GasLandV
  - NucLandV
  - EleLandV

- WaterV
  - EleWaterV
  - GasWaterV
  - NucWaterV

- AirV
  - GasAirV
  - NucAirV
  - EleAirV

- SpaceV
  - GasSpaceV
  - NucSpaceV
  - EleSpaceV
Multiple inheritance

Vehicle

LandV

WaterV

AirV

SpaceV

GasV

EleV

NucV

GasLandV

NucLandV

EleWaterV

GasAirV

GasSpaceV

NucSpaceV

EleLandV

GasWaterV

NucWaterV

EleAirV
Multiple inheritance
Multiple inheritance
Multiple inheritance
Bridge Pattern

Abstraction
- impl: Implementor
+ function()

Implementor
+ implementation()

RefinedAbstraction
+ refinedFunction()

ConcreateImplementor
+ implementation()
Vehicle example: Bridge Pattern

Vehicle

Engine

LandV  WaterV  AirV  SpaceV

GasE  EleE  NucE
To CLion!

See vehicle_bridge.h.