CS 247: Software Engineering Principles

Object-Oriented Design Principles

Reading: none
Today's Agenda

Object-Oriented Design Principles - characteristics, properties, and advice for making decisions that improve the modularity of the design.

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

**OO Principles**
- Program to an Interface, not an Implementation
  - aka Open Closed Principle
- Favour Composition over Inheritance
- Single Responsibility Principle
- Liskov Substitutability Principle
- Principle of Least Knowledge (Law of Demeter)
If you want to know more:

- Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
**Open Closed Principle**

**Principle:** A module should be open for extension but closed to modification.*

*AKA Program to an Interface, not an Implementation

**Idea:** Have client code depend on an abstract class (that can be extended) rather than on concrete classes.

**Example:** Dynamic polymorphism

![Diagram showing the open closed principle in action with client code, TermRec, Math TermRec, Engineering TermRec, and Science TermRec classes with the calcGPA method.](image-url)
"Default" Implementation

Alternatively, our abstract base class may provide a default implementation that the derived classes may override. This makes it easier to derive new classes.

```
class TermRec {
    public:
        virtual void print() = 0;
        virtual void calcGPA();
    protected:
        TermRec();
};
void TermRec::calcGPA() {...}
```

Abstract base class can declare and define common data and operations.
Inheriting Interface vs. Implementation

The abstract base class designer determines what parts of a member function the derived class inherits:

1. interface (declaration) of member function e.g. `printStats`
2. interface and (default) overridable implementation e.g. `calcGPA`
3. interface and non-overridable implementation e.g. `print`

```cpp
class TermRecord {
    public:
        virtual void printStats() const = 0; // template method op
        virtual float calcGPA(); // compute mean average
        void print() const; // print template transcript
    ...
};

class MathTermRecord : public TermRecord { ...
}
class EngineeringTermRecord : public TermRecord { ...
}
Problem: When defining a new class that includes attributes and capabilities of an existing class, should our new class:

- inherit from the existing class (inheritance)?
- include existing class as a complex attribute (composition)?

**Inheritance vs. Composition**

Bertrand Meyer, *Object-Oriented Software Construction*

![Diagram](https://example.com/diagram.png)
Choosing Inheritance

**Principle:** Favour inheritance when:

1. using **subtyping**—i.e., using the fact that a derived class can be used wherever the base class is accepted.
2. using the entire interface of an existing class.
Choosing Composition

Principle: Favour composition for simple (non-overriding) code reuse and extension...

...because with composition, the components' capabilities (data and functions) can change at run-time.

```
Window -> rectangle

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
</tr>
<tr>
<td>width</td>
</tr>
<tr>
<td>area()</td>
</tr>
</tbody>
</table>
```
Delegation in object composition simulates inheritance-based method reuse.

- Composite object delegates operations to component object.
- Can pass itself as parameter, to let delegated operation refer to composite object.

```cpp
Window
area()

return rectangle->area();

Rectangle
height
width
area()

return width*height;
```
The benefits of composition are maximized when the component is an abstract type—an interface (in Java) or an abstract base class (in C++)—that can be concretized in different ways.

```cpp
Window
  area() 
  return shape->area();

Shape
  area()

Rectangle
  height
  width
  area() 
  return width*height;

Circle
  radius
  area() 
  return PI * radius * radius;
```

```cpp
return shape->area();
```
The Single-Responsibility Principle

**Principle:** Encapsulate each changeable design decision in a separate module.

The **Single-Responsibility Principle** offers guidance on how to decompose our program into cohesive modules.

**Example:** Based on the names of its methods, how many design decisions are implemented by this one class?

<table>
<thead>
<tr>
<th>DeckOfCards</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNextCard() : bool</td>
</tr>
<tr>
<td>nextCard() : Card</td>
</tr>
<tr>
<td>addCard(Card)</td>
</tr>
<tr>
<td>removeCard(Card)</td>
</tr>
<tr>
<td>shuffle()</td>
</tr>
</tbody>
</table>

U Waterloo CS247 (Spring 2017) — p.12/31
Liskov Substitutability Principle (LSP)

Principle: A derived class must be substitutable for its base class.

Idea: A derived class must preserve the behaviour of its base class, so that it will work with client code that uses the base class.

- Objects accept the base class's messages.
- Methods require no more than base class methods.
- Methods promise no less than base class methods.
Example: Bounded Stack

A bounded stack is a specialized type of stack that can store only a limited number of elements.
Substitutability Rules
Liskov, Guttag, *Program Development in Java*

When overriding inherited virtual functions, three rules must be followed:

1) **Signatures:** The derived-class objects must have all of the methods of the base class, and their signatures must match.

2) **Method behaviours:** Calls to derived-class methods must behave like calls to the corresponding base-class methods.

3) **Properties:** The derived class must preserve all properties of the base class objects.
Signatures: The derived class must support all of the methods of the base class, and their signatures must match:

- Parameters of overridden virtual methods must have compatible types as the parameters of the base class's methods.
  - C++, Java: same types (otherwise redefining)

- The return type of an overridden virtual method must be compatible with the return type of the base-class's method.
  - C++, Java: same type, or subtype of

- A derived-class method raises the same or fewer exceptions than the corresponding base-class method.
Method behaviours: A derived-class method maintains or weakens the precondition and maintains or strengthens the postcondition:

- **Precondition Rule:**
  \[ \text{pre}_\text{base} \Rightarrow \text{pre}_\text{derived} \]

- **Postcondition Rule:**
  \[ \text{(pre}_\text{base \&\& post}_\text{derived}) \Rightarrow \text{post}_\text{base} \]

In other words, the specification of a derived class must be stronger-than the specification of its base type.
LSP Property Rules
Liskov, Guttag, *Program Development in Java*

**Property behaviours:** The derived class must preserve all declared (and enforced) properties of the base class objects.

- invariants (e.g., no duplicate elements in a container type)
- optimized for performance (memory requirements, timing)
Substitutability Example

class Stack {
    long *items_;  
    int tos_;  

public:
    Stack();  
    Stack(int);  
    ~Stack();  
    long top() const;  
    long pop();  
    virtual void push(long);  
};

class CountStack : public Stack {
    int count_;  

public:
    CountStack() : count_(0) {}  
    void push(long);  
    int numPushes() const;  
};

    int CountStack::numPushes() {
        return count_;  
    }

    void CountStack::push(long v) {
        Stack::push(v);  
        count_ += 1;  
    }

U Waterloo CS247 (Spring 2017) — p.19/31
Another Example

Is SubstituteEntity substitutable for Entity?
Yet Another Example

Is ChequingAccount (with a credit limit) substitutable for Account?
Encapsulation of Components

Information Hiding: Modular design should hide design and implementation details, including information about components.

<table>
<thead>
<tr>
<th>Transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPA</td>
</tr>
<tr>
<td>addTermRecord (term)</td>
</tr>
<tr>
<td>computeTermGPA (term)</td>
</tr>
<tr>
<td>printTermRecord (term)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TermRecord</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
</tr>
<tr>
<td>termGPA</td>
</tr>
<tr>
<td>academic standing</td>
</tr>
<tr>
<td>- TermRecord (term)</td>
</tr>
<tr>
<td>- computeGPA() : float</td>
</tr>
<tr>
<td>- print()</td>
</tr>
</tbody>
</table>
Composition and Data Encapsulation

### Client Code

Transcript *t;
...

```c
    t->addTermRecord("Spring2014");
    t->computeTermGPA("Spring2014");
    t->printTermRecord("Sring2014");
```

### Client Code

Transcript *t;
TermRecord *tr;
...

```c
    tr = t->getTermRecord("Spring2014");
    tr->computeGPA();
    tr->print();
```
Composition and Data Encapsulation

Composite object offers methods for accessing and manipulating component information.

Client Code

```cpp
Transcript *t;
TermRecord *tr;
...
tr = t->getTermRecord("Spring2014");
tr->computeGPA();
tr->print();
```
Another Example

void Store::settlebill(float total) {
    
    Customer->getPayPal()->getCreditCard()->charge(total);
    
}
### Component Encapsulation

#### Composite Objects

<table>
<thead>
<tr>
<th>Class</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store</td>
<td>+ settlebill (total)</td>
</tr>
<tr>
<td>Customer</td>
<td>+ pay (total)</td>
</tr>
<tr>
<td>PayPal</td>
<td></td>
</tr>
<tr>
<td>CreditCard</td>
<td></td>
</tr>
</tbody>
</table>

#### Code Example

```cpp
void Store::settlebill(float total) {
    ...  
    Customer->pay(total);
    ...
}
```
"Law" tests encapsulation: an object "talks only to its neighbours"

Method A::m1 can only call methods of

- A itself
- A’s data members
- m1’s parameters
- any object constructed by A's methods

In particular, an A object should not invoke methods of B objects returned by a method/function call.

You can play with yourself.
You can play with your own toys (but you can't take them apart),
You can play with toys that were given to you.
You can play with toys you've made yourself.

Peter VanRooijen
BasicAccount::call() creates a record of a cellphone call, recording the start date and start time of the call and the duration in minutes. BasicAccount::bill() decrements the account's balance by the monthly service fee ($100). BasicAccount::pay() increments the balance by the amount paid.

CheapAccount::call() creates a record of a cellphone call, and updates the number of minutes (numMinutes) of calls in the current month. CheapAccount::bill() decrements the account's balance by the $30 monthly service fee and by $1 for each minute of calls beyond the free 200 minutes of calls in the current month; it also resets numMinutes to 0 to start recording number of call minutes for the next month.
The above design can be inefficient if a customer frequently changes the type of his or her cellphone plan. Use the principle of *Favour Composition over Inheritance* to restructure the design to minimize how much has to change when a customer's cellphone plan changes from a BasicAccount to a CheapAccount or vice versa.
Which program statements violate the design principle *Law of Demeter*?

```cpp
#include <vector>
#include "Date.h"
#include "Time.h"

class BasicAccount {
 public:
    ... 
    void printCallRecords ( const Date &minDate, const Date &maxDate ) const;

 private:
    std::vector<CallRecord*> calls_; 
    static int monthlyFee_; // = $100/month 
    int balance_; 
};

BasicAccount::monthlyFee = 100;

void BasicAccount::printCallRecords( const Date &minDate, const Date &maxDate) const {
    for ( auto item : calls_ ) {
        if ( item->startDate() >= minDate && item->startDate() <= maxDate ) {
            std::cout << item->startDate() << " " << item->startTime();
            std::cout << " " << item->duration() << std::endl;
        }
    }
}
```
Take Aways

Recognition
• The goals underlying each design principle.
• That the principles are guidelines, not rules or laws.

Comprehension
• Detect that a design violates a principle.
• Argue (with justification) that one design is better than another.

Application
• Design an abstract base class.
• Determine whether one class is substitutable for another.
• Convert an inheritance relation into a composition relation.
• Determine whether a class encompasses more than one responsibility.