"Gang of Four"* Design Patterns

Abstract OO designs that encapsulate change, to improve the modularity and flexibility of our code.

i.e., increase cohesion, loosen coupling, improve information hiding

How patterns help:

- Improve our **efficiency** in finding a suitable design
- Improve the **predictability** of the design quality
- Offer **higher-level abstractions** than procedures or classes
- Extend developers' **vocabulary**
- Ease refactoring and evolution

*GoF = "Gang of Four", authors of original book on object-oriented Design Patterns
Problem: Duplicate Code

Consider a bicycle computer that has a mode button for switching the display of different data being collected: current time, speed, distance.

Example code:

```cpp
void Distance::showValue ()
{
    display_.setLabel ("D");
    display_.setValue (dist_);
    display_.setUnits ("km");
}

void Time::showValue ()
{
    display_.setLabel ("T");
    display_.setValue (hour_ + ":" + min_ + ":" + sec_);
    display_.setUnits (""");
}

void Speed::showValue ()
{
    display_.setLabel ("S");
    display_.setValue (speed_);
    display_.setUnits ("km/h");
}
```

Notes:

- **Approach 1: Inheritance**
  - Put common code in abstract base class’s method. Can be shared by subclasses’ methods.
  - **Downside:** subclasses may neglect to use parent behaviour.

Similar Algorithms

Examples from SE_2013:

- [http://headfirstlabs.com/books/hfdp/](http://headfirstlabs.com/books/hfdp/)

References

- Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- Code examples: [http://headfirstlabs.com/books/hfdp/](http://headfirstlabs.com/books/hfdp/)
Inheriting Common Code

The commonality among the classes' `showValue()` methods is the structure of the algorithm.
The variations are data values (but the data values could just as easily be subroutine values).

Approach 2: Template Method

A template method is a base-class method that defines common code structure, but includes primitive operations (holes) to be defined by subclass methods.

```cpp
// template method
void Function::showValue () {
    display_.setLabel (theLabel());
    display_.setValue (theValue());
    display_.setUnits (theUnits());
}
```

It is essential that:
- the template method be nonvirtual
- the primitive operations be virtual functions in the base class

Template Method Pattern

Problem: duplicate code
Multiple subclass methods have similar algorithm structures.

Solution: localize duplicate code structure in an abstract class
Abstract class defines a template method (of common code) that calls pure virtual subroutines. Subclasses override the subroutines.
Example: Assignment 1

Operation that bills a cellphone plan subscriber, based on plan type:
- monthly fee
- charges for additional calls

```
class Account {
public:
  Account ();
  void bill ();
  ...
private:
  int balance_;
  virtual int monthlyFee() = 0;
  virtual int callCharges() = 0;
};
Account::Account(): balance_{0} {}
// template method
void Account::bill() {
  balance_ += monthlyFee() + callCharges();
}
```

```
class ExpensiveAccount : public Account {
public:
  ExpensiveAccount ( );
private:
  static const int monthlyFee_ = 80;
  int monthlyFee() override;
  int callCharges() override;
};
ExpensiveAccount:ExpensiveAccount( ) {}
int ExpensiveAccount::monthlyFee ()  {
  return monthlyFee_;
}
int ExpensiveAccount::callCharges() {
  return 0;
}
```

```
class CheapAccount : public Account {
public:
  CheapAccount ( );
private:
  static const int monthlyFee_ = 25;
  static const int freeMin_ = 250;
  static const int rate_ = 1;
  int minutes_; 
  int monthlyFee() override;
  int callCharges() override;
};
CheapAccount::CheapAccount: minutes_{0} {} 
int CheapAccount::monthlyFee() {
  return monthlyFee_; 
}
int CheapAccount::callCharges() {
  if ( (minutes_ - freeMin_) > 0 )
    return (minutes_ - freeMin_) * rate_;
  return 0;
}
```

Abstract Account
Adapter Pattern Idea

Problem: Interface mismatch between two modules.
- Want to reuse an existing class, but its interface does not match what is needed.
- Or the interface of one of our modules changes (!!) and we don't want to make major changes to the existing (working!) code.

Solution: Define an Adapter class that maps one interface to another.

Example: STL Stacks

STL stacks are implemented by adapting the interface of a Container class (by default, deque).

```
// Example implementation of a stack using a deque
template<typename T, template<typename,typename> class Container = std::deque, class A = std::allocator<T>>
class stack {
  public:
    bool empty() const;  
    int size() const;  
    T& top() const;  
    void push(const T& val);  
    void pop();  
  private:  
    Container<T> contents_;  
};
```

STL Stacks

Here is a version of the implementation:

```
template<
  typename T,  
  template<typename,typename> class Container = std::deque, class A = std::allocator<T>>
class stack {
  public:
    bool empty() const;
    int size() const;
    T& top() const;
    void push(const T& val);
    void pop();
  private:
    Container<T> contents_; 
};
```

Example: STL Stacks

The Adapter class translates requests made to the Target interface into requests made of the Adaptee object.

```
// Adapter class
class Adapter {
  public:
    void request(const T& val) {
      contents_.push_back(val);
    }
};
```

---

---
Facade Design Pattern

Problem: complex interface.
Client of subsystem interacts with multiple (complex?) classes.

Solution: create a single, simplified interface (class).
Restrict, simplify client's interactions with subsystem's classes.

Another Example

Metaphor: Facade Design Pattern

Strategy Pattern

Problem: Want to vary an algorithm at run-time.

Solution: Encapsulate the algorithm decision.
Define algorithm as a component object.
Use subclassing to specialize the algorithm in different ways.
Example 1

A class that authenticates users, and uses a variety of authentication policies.

```cpp
class AgilePlayer {
private:
    BiddingStrategy *strategy_
...
public:
    AgilePlayer();
    ...,
    void bid ();
    void reassess_Bidding_Strategy( )
};
AgilePlayer::AgilePlayer() : strategy_(new Conservative) {}
void AgilePlayer::bid( ) {
    strategy_ -> bid(*this);
}
void AgilePlayer::reassess_Bidding_Strategy( ) {
    ... // decide to change bidding strategy
    delete strategy_
    strategy_ = new Aggressive;
```

Example 2

A card player whose strategy changes at runtime.

```cpp
class AgilePlayer {
private:
    BiddingStrategy *strategy_
...
public:
    AgilePlayer();
    ...,
    void bid ();
    void reassess_Bidding_Strategy( )
};
AgilePlayer::AgilePlayer() : strategy_(new Conservative) {}
void AgilePlayer::bid( ) {
    strategy_ -> bid(*this);
}
void AgilePlayer::reassess_Bidding_Strategy( ) {
    ... // decide to change bidding strategy
    delete strategy_
    strategy_ = new Aggressive;
```

Implementation

```cpp
class AgilePlayer {
private:
    BiddingStrategy *strategy_
...
public:
    AgilePlayer();
    ...,
    void bid ();
    void reassess_Bidding_Strategy( )
};
AgilePlayer::AgilePlayer() : strategy_(new Conservative) {}
void AgilePlayer::bid( ) {
    strategy_ -> bid(*this);
}
void AgilePlayer::reassess_Bidding_Strategy( ) {
    ... // decide to change bidding strategy
    delete strategy_
    strategy_ = new Aggressive;
```

Considerations

Applicability:
• Different variants of an algorithm.
• Choose/Change algorithm at runtime.

Some complications:
• Encapsulation of context data to be operated on.
• All algorithms must use the same Strategy interface.
Summary

The goal of design patterns is to encapsulate change.

**Template Method Pattern:** encapsulates that parts of an algorithm that is different for each derived class.

**Adapter Pattern:** encapsulates the interface to a class.

**Facade Pattern:** doesn’t really encapsulate the subsystem classes since they’re still available but does decouple the client from the subsystem.

**Strategy Pattern:** encapsulates the algorithm to be used, such that it can be set (and changed) at runtime.