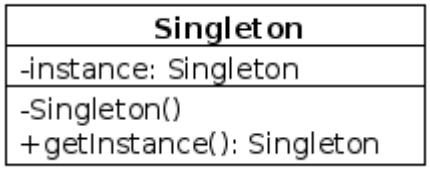
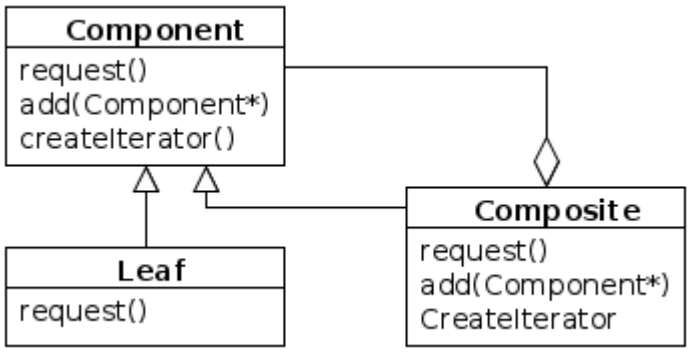
Singleton (одиночка)



Позволяет создавать единственный экземпляр некоторого типа, предоставляет к нему доступ извне и запрещает создание нескольких экземпляров того же типа.

```
template <typename T>
class Singleton
{
public:
    static T& instance()
    {
        if (!myInstance)
            myInstance = new T;
        return *myInstance;
    }
private:
    static T* myInstance;
    Singleton(const Singleton&){}
};

template <typename T>
T* Singleton::myInstance = NULL;
```

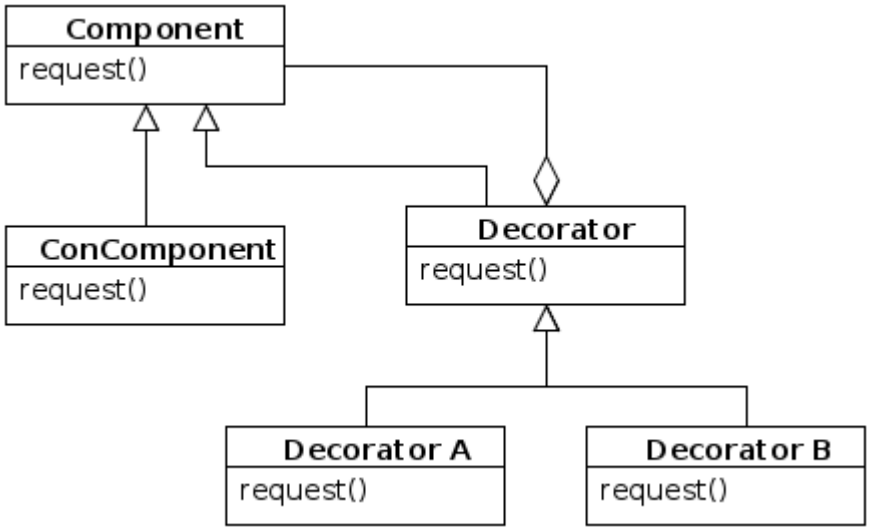
Composite (композит, компоновщик)



Объединение групп схожих объектов и управление ими. Объекты могут быть как примитивными, так и составными. Может образоваться сложная древовидная структура. Работа с примитивными и составными объектами единообразна.

```
class Unit { // component
public:
    virtual int getStrength() = 0;
    virtual void addUnit(Unit* p) {}
    virtual ~Unit() {}
};

class Archer: public Unit { // leaf

public:
    virtual int getStrength() {return 1;}
};

class Infantryman: public Unit { // leaf
public:
    virtual int getStrength() {return 2;}
};

class CompositeUnit: public Unit { // composite
public:
    int getStrength()
    {
        int total = 0;
        for (int i = 0; i < c.size(); ++i)
            total += c[i]->getStrength();
        return total;
    }

    void addUnit(Unit* p) {c.push_back(p);}

    ~CompositeUnit()
    {
        for(int i = 0; i< c.size(); ++i)
            delete c[i];
    }
private:
    std::vector<Unit*> c;
};

CompositeUnit* createLegion() {
    CompositeUnit* legion = new CompositeUnit;
    for (int i = 0; i < 3000; ++i)
        legion->addUnit(new Infantryman);
    for (int i = 0; i < 1200; ++i)
        legion->addUnit(new Archer);
    return legion; }
```

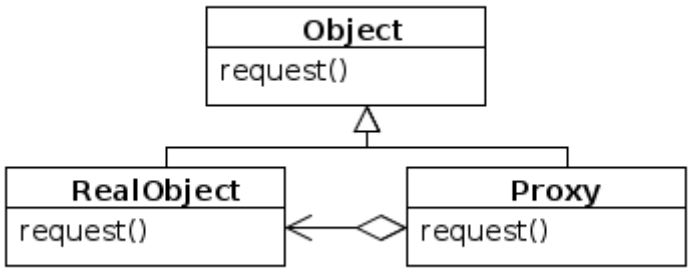
Decorator (декоратор)



Добавление функционала. Динамически добавляет новые обязанности объекту.

```
class Widget // component
{
public:
    virtual void draw() = 0;
};

class TextField: public Widget // conComponent
{
    int width, height;
public:
    TextField(int w, int h) {
        width = w;
        height = h;
    }

    void draw() { cout << "TextField: " << width << ", " << height << 'n'; }
};

class Decorator: public Widget // Decorator
{
    Widget *wid;
public:
    Decorator(Widget *w) {wid = w;}
    void draw() {wid->draw();}
};

class BorderDecorator: public Decorator // Decorator A
{
public:
    BorderDecorator(Widget *w): Decorator(w) {}
    void draw() {Decorator::draw();cout << "BorderDecorator" << 'n';}
};

Widget *aWidget = new BorderDecorator( new BorderDecorator( (new TextField(80,
24))));

aWidget->draw();
```

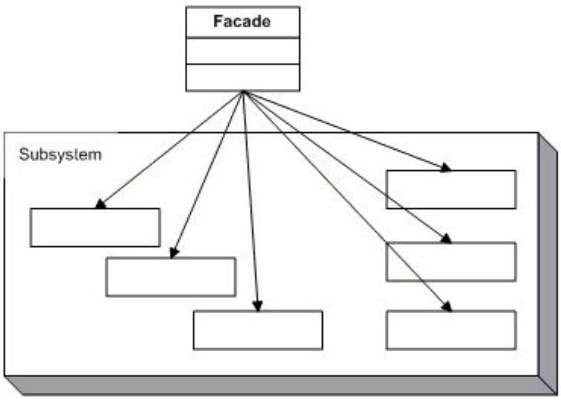
Proxy (заместитель)



Является заместителем другого объекта и контролирует доступ к нему.
По сути: вам нужно управлять ресурсоемкими объектами. Вы не хотите создавать экзмепляры таких объектов до момента их реального использования. Интерфейс взаимодействия при этом **не меняется**.

```
class Image
{
    int myId;
public:
    Image();
    ~Image();

    virtual void draw() = 0;
};

class RealImage : Image
{
    RealImage(int i) {myId = i;}
    void draw() {cout << "   drawing image " << myId << '\n';}
};

class ProxyImage : Image
{
    RealImage* realImage;
    static int next;
public:
    ProxyImage()
    {
        myId = next++;
        realImage = 0;
    }

    void draw()
    {
        if (!realImage)
        {
            realImage = new RealImage(myId);
        }
        realImage->draw();
    }
};

int ProxyImage::next = 1;
```

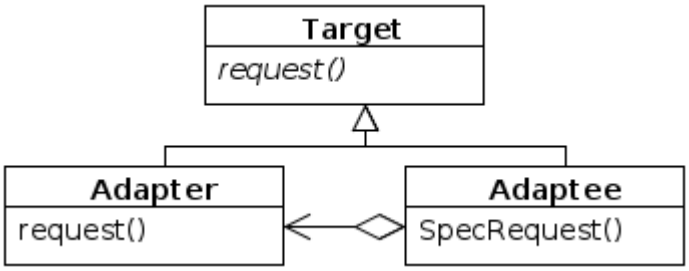
Facade (фасад)



Позволяет скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

```
class Band {
    private:
        GuitarPlayer _GuitarPlayer;
        Drummer _Drummer;
        BassPlayer _BassPlayer;
        void PlayVerse(int nVerse);
        void PlayChorus();
        void PlaySolo();
    public:
        Band(const char *guitarPlayerName, const char *drummerName,
             const char *bassPlayerName) : _GuitarPlayer(guitarPlayerName),
            _Drummer(drummerName), _BassPlayer(bassPlayerName) { }
        ~Band() { }
        void PlaySong();
};
class Musician {
    protected:
        string _Name;
    public:
        virtual ~Musician() { }
};
class GuitarPlayer: public Musician {
    public:
        GuitarPlayer(const char *name) { _Name = name; }
        ~GuitarPlayer() override { }
        void PlayVerseRiff(int nVerse);
        void PlayChorusRiff();
        void PlaySolo();
};
void GuitarPlayer::PlayVerseRiff(int nVerse)
{
    cout << _Name << " plays verse riff " << nVerse << ".\n";
}
/*...*/
class Drummer: public Musician {
    /*...Rhythm*/
};
/*...*/
class BassPlayer: public Musician {
    /*...Riff*/
};
/*...*/
void Band::PlayVerse(int nVerse)
{
    cout << "Verse " << nVerse << ":\n";
    _GuitarPlayer.PlayVerseRiff(nVerse);
    _Drummer.PlayVerseRhythm();
    _BassPlayer.PlayVerseRiff();} /*...*/
```

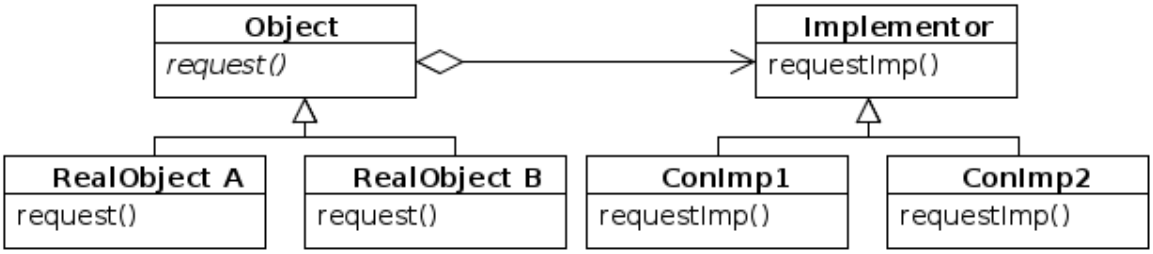
Adapter (адаптер)



Заменяет интерфейс, не изменяя функционал. Применяется, когда имеющиеся классы обладают нужным функционалом, но не подходящим интерфейсом.

```
class FahrenheitSensor  // adaptee
{
  public:
    float getFahrenheitTemp()
    {
      float t = 32.0;
      // ...
      return t;
    }
};

class Sensor  // target
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor   // adapter
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {}
   ~Adapter() {delete p_fsensor;}
    float getTemperature()
    {
      return (p_fsensor->getFahrenheitTemp() - 32.0) * 5.0 / 9.0;
    }
  private:
    FahrenheitSensor* p_fsensor;
};

int main()
{
  Sensor* p = new Adapter(new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
  delete p;
  return 0;
}
```

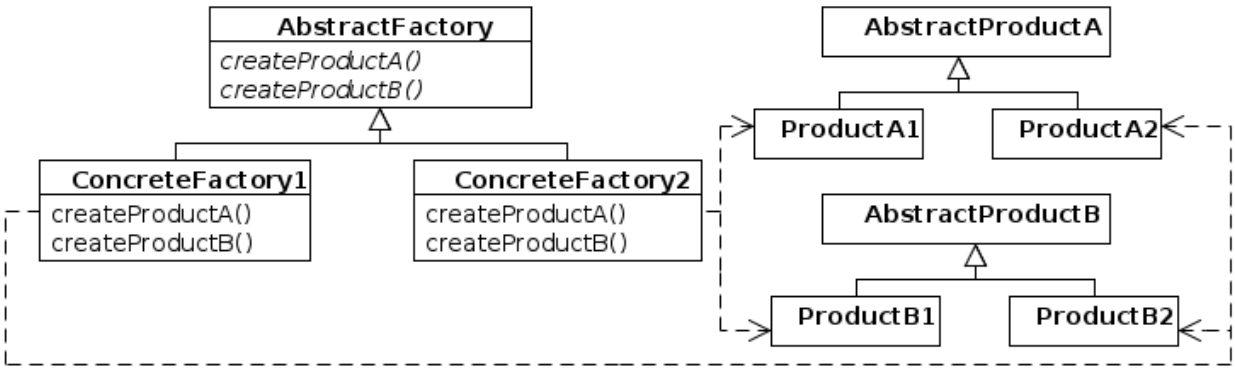
Bridge (мост)



Разделяет абстракцию и реализацию так, чтобы они могли изменяться независимо. По сути: есть базовый класс, определяющий интерфейс для производных. Обычно мы бы использовали наследование и все было бы норм, но при таком подходе мы жестко привязаны к наследованию. То есть интерфейс и реализация в производном классе не могут изменяться независимо.

```
class AlarmSystem // object
{
    public:
        virtual ~AlarmSystem() { }
        virtual void MakeSound() = 0;
};

class StereoAlarmSystem: public AlarmSystem  // real object
{
    public:
        ~StereoAlarmSystem() override { }
        void MakeSound() override {cout << "Music playes!!! \n";}
};

class AlarmClock  // implementor
{
    protected:
        AlarmSystem *_AlarmSystem;

        int _Hour;
        int _Minute;

    public:
        virtual ~AlarmClock() { delete _AlarmSystem; }

        void SetTime(int hour, int minute) { _Hour = hour; _Minute = minute; }
        virtual void Ring() = 0;
};

class StereoAlarmClock: public AlarmClock  // conImplementor
{
    public:
        StereoAlarmClock() { _AlarmSystem = new StereoAlarmSystem(); }
        ~StereoAlarmClock() override { }

        void Ring() override;
};

void StereoAlarmClock::Ring()
{
    cout << "Time: " << _Hour << ":" << _Minute << "\n";
    _AlarmSystem->MakeSound();
}
```

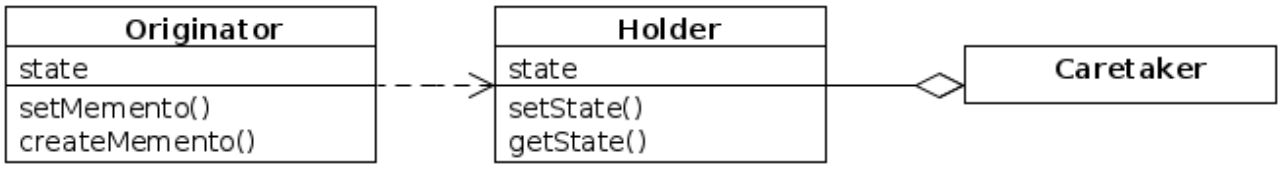
Abstract factory (абстрактная фабрика)



Предоставляет возможность создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.
По сути: просим фабрику сделать нам что-то. Она делает, но как именно – решает сама фабрика, мы этого не знаем. Таким образом мы можем менять способ «производства», но пользователь будет все так же делать запрос к фабрике, и для него ничего не изменится.

```
class Car {
    protected:
        string _Brand;

    public:
        virtual ~Car() { }
        void ShowBrand() { cout << _Brand << "\n"; } };
class Zhiguli: public Car {
    public:
        Zhiguli() { _Brand = "Zhiguli"; }
        ~Zhiguli() override { } };
class Mitsubishi: public Car {
    public:
        Mitsubishi() { _Brand = "Mitsubishi"; }
        ~Mitsubishi() override { } };
class Hammer: public Car {
    public:
        Hammer() { _Brand = "Hammer"; }
        ~Hammer() override { } };
enum class RoadType {
    CITY,
    COUNTRYSIDE,
    MOUNTAIN_AREA
};

class CarFactory {
    public:
        static Car* MakeCar(RoadType roadType)
        {
            switch (roadType) {
                case RoadType::CITY:
                    return new Zhiguli();
                case RoadType::COUNTRYSIDE:
                    return new Mitsubishi();
                case RoadType::MOUNTAIN_AREA:
                    return new Hammer();
            }
        }
};
```

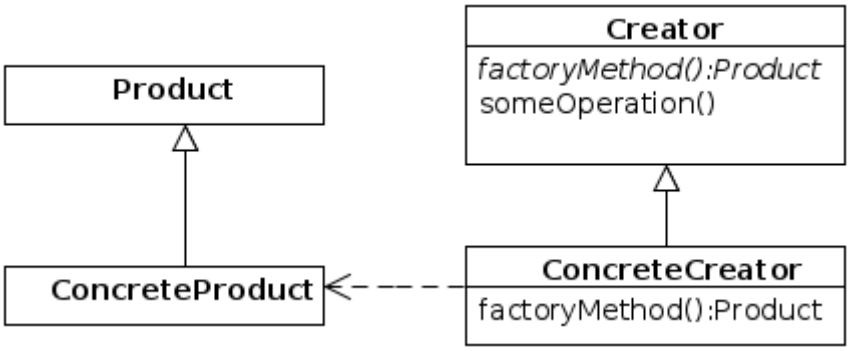
Holder (memento, хранитель)



Получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.

```
class IOriginator
{
    Memento GetMemento();
    void SetMemento(Memento memento);
};

class Memento
{
private:
    int _helth;
public:
    Memento(int helth) {_helth = helth; }
    int GetState() { return _helth; }
};

class Player : IOriginator
{
private:
    int _helth;

public:
    Player() {_helth = 100;}
    void GetHurt(int hurt) { _helth -= hurt; }

    void GetCure(int cure) { _helth += cure; }

    void PrintPulse()
    {
      if(_helth > 70)
        Console.WriteLine("Green");
      if(_helth <= 70 && _helth > 40)
        Console.WriteLine("Yellow");
      if (_helth <= 40)
        Console.WriteLine("Red");
    }

    void SetMemento(Memento memento) { _helth = memento.GetState(); }
    Memento GetMemento() { return new Memento(_helth); }
  }
```

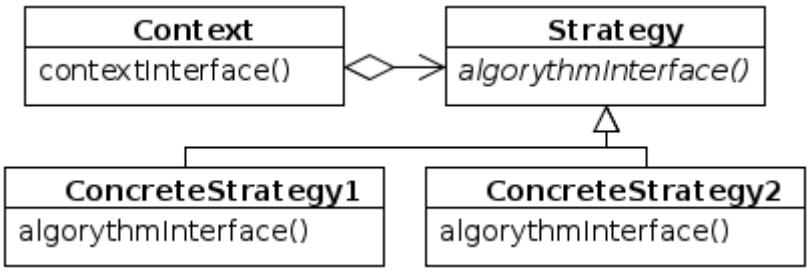
Factory method (фабричный метод)



Определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

```
enum Warrior_ID { Infantryman_ID=0, Archer_ID, Horseman_ID };
class Warrior
{
  public:
    virtual void info() = 0;
    virtual ~Warrior() {}
    static Warrior* createWarrior( Warrior_ID id );
};

class Infantryman: public Warrior
{
  public:
    void info() { cout << "Infantryman" << endl; }
};

class Archer: public Warrior
{
  public:
    void info() { cout << "Archer" << endl; }
};

class Horseman: public Warrior
{
  public:
    void info() { cout << "Horseman" << endl; }
};

Warrior* Warrior::createWarrior( Warrior_ID id )
{
    Warrior * p;
    switch (id) {
        case Infantryman_ID:
            p = new Infantryman();
            break;
        case Archer_ID:
            p = new Archer();
            break;
        case Horseman_ID:
            p = new Horseman();
            break;
        default:
            assert( false);
    }
    return p;
}
```

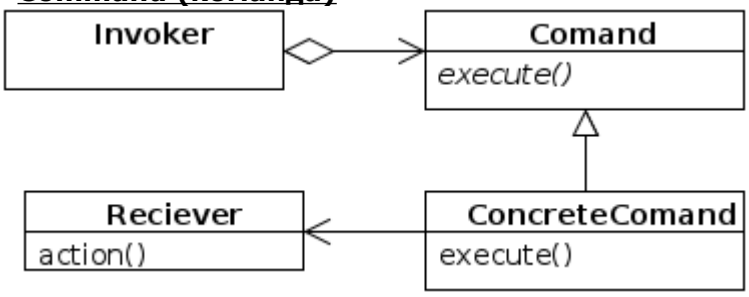
Strategy (стратегия)



Определяет группу алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Позволяет изменить алгоритм независимо от клиента.

```
class Compression  // context
{
  public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};

class Compressor  // Strategy
{
  public:
    Compressor( Compression* comp): p(comp) {}
   ~Compressor() { delete p; }
    void compress( const string & file ) { p->compress( file); }
  private:
    Compression* p;
};

class ZIP_Compression : public Compression  // conStr1
{
  public:
    void compress( const string & file ) { cout << "ZIP compression" << endl; }
};

class ARJ_Compression : public Compression  // conStr2
{
  public:
    void compress( const string & file ) { cout << "ARJ compression" << endl;}
};

class RAR_Compression : public Compression  // conStr3
{
  public:
    void compress( const string & file ) { cout << "RAR compression" << endl;}
};
```

Command (команда)



Преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

```
class Document  // invoker
{
   vector<string> data;
public:
   void Insert( int line, const string & str );
   void Remove( int line );
};

class Command   // command
{
protected:
   Document * doc;
public:
   virtual void Execute() = 0;
   virtual void unExecute() = 0;

   void setDocument( Document * _doc ) { doc = _doc; }
};

class InsertCommand : public Command    // concreteCommand
{
   int line;
   string str;
public:
   InsertCommand( int _line, const string & _str ): line( _line ), str( _str )
{}

   void Execute();
   void unExecute();
};

class Receiver        // reciever
{
   vector<Command*> DoneCommands;
   Document doc;
   Command* command;
public:
   void Insert( int line, string str )
   {
      command = new InsertCommand( line, str );
      command->setDocument( &doc );
      command->Execute();
      DoneCommands.push_back( command );
   }
};
```