11. Patrones de diseño

Julio Vega

julio.vega@urjc.es

Contenidos

- El diseño software es un proceso iterativo de afinamiento de este.
- Cualquier software complejo requiere varias etapas.
  - Al principio se identifican los módulos más abstractos.
  - Progresivamente se va concretando el diseño de cada módulo.
- En el largo proceso del diseño software aparecen problemas.
  - Y estos problemas suelen repetirse con frecuencia en un DS.
  - Esto es, suelen darse diversos *patrones* de problemas/soluciones.
- **Patrones de diseño:** formas de resolver problemas comunes de DS.
- Igual que reutilizamos código, reutilizamos soluciones de DS.

- **Nombre:** corto y autodefinido, fácil de usar por diseñadores.
- **Problema:** qué resuelve el patrón y en qué contexto.
- **Solución:** normal/ describe las clases y relaciones entre objetos.
- **Ventajas/Desventajas:** complejidad, t.º ejec., portabilidad, etc.

- **De creación:** ofrecen solución sobre construcción clases, objetos, etc.
  - E.g. *Abstract Factory*, *Builder*, etc.
- **Estructurales:** jerarquía de clases, relaciones y composiciones.
  - E.g. *Adapter*, *Facade*, *Flyweight*, etc.
- **De comporta/:** cómo organizar ejecución, mensajes entre objetos.
  - E.g. *Visitor*, *Iterator*, *Observer*, etc.

- Son la base para la definición de patrones más complejos.
  - Como los citados previamente.
- Surgen de la relación *method-method*.
  - Esto es, cuando un método A llama desde su código a un método B.
  - Dicho de otro modo, cuando el método A depende de B.
    - Ya sea directa o indirectamente, como veremos a continuación.

```
class B {
  void methodB () {
    // ...
  }
}

class A {
  B b;
  void methodA () { // methodA depends on methodB
    b.methodB ();
    // ...
  }
}

main () {
  A a;
  a.methodA ();
}
```

```
class B {
  void methodB () { /* ... */ }
}
class C {
  B b;
  void methodC () { // methodC depends on methodB
    b.methodB ();
  }
}
class A {
  C c;
  void methodA () { // methodA depends on methodC
    c.methodC ();
  }
}
main () {
  A a;
  a.methodA ();
}
```

- Son originados por la relación *method-method*.
  - Y haciendo las combinaciones posibles entre los objetos y los métodos.
- **Recursion:** objetos y métodos implicados son los mismos.
- **Conglomeration:** si objetos son los mismos pero métodos diferentes.
- **Redirection:** objeto recibe invocación a método.
  - Y este método invoca al "mismo" método de otro objeto.
  - A un método cuya funcionalidad es igual aunque su implementación no.
- **Delegation:** cuando un método delega funcionalidad en otro.

```
class GUIButtonMaker {
  GUIButton makeGUIButton {
    // make the GUI button and return it
  }
}

class GUIMaker {
  GUIButtonMaker g;
  GUI makeGUI () {
    GUIButton button1 = g.makeGUIButton ();
    // GUIMaker object delegates making button to g instance
    // continues doing more stuff...
  }
}

main () {
  GUIMaker g;
  g.makeGUI ();
}
```

```
class Printer3D {
  void print (Model m) {
    //...
  }
}

class PrinterManager {
  Printer3D p;
  void print (Model m) {
    p.print (m); // PM object redirects its job to p instance
    // continues doing more stuff...
  }
}

main() {
  Model m;
  PrinterManager pm;
  pm.print (m);
}
```

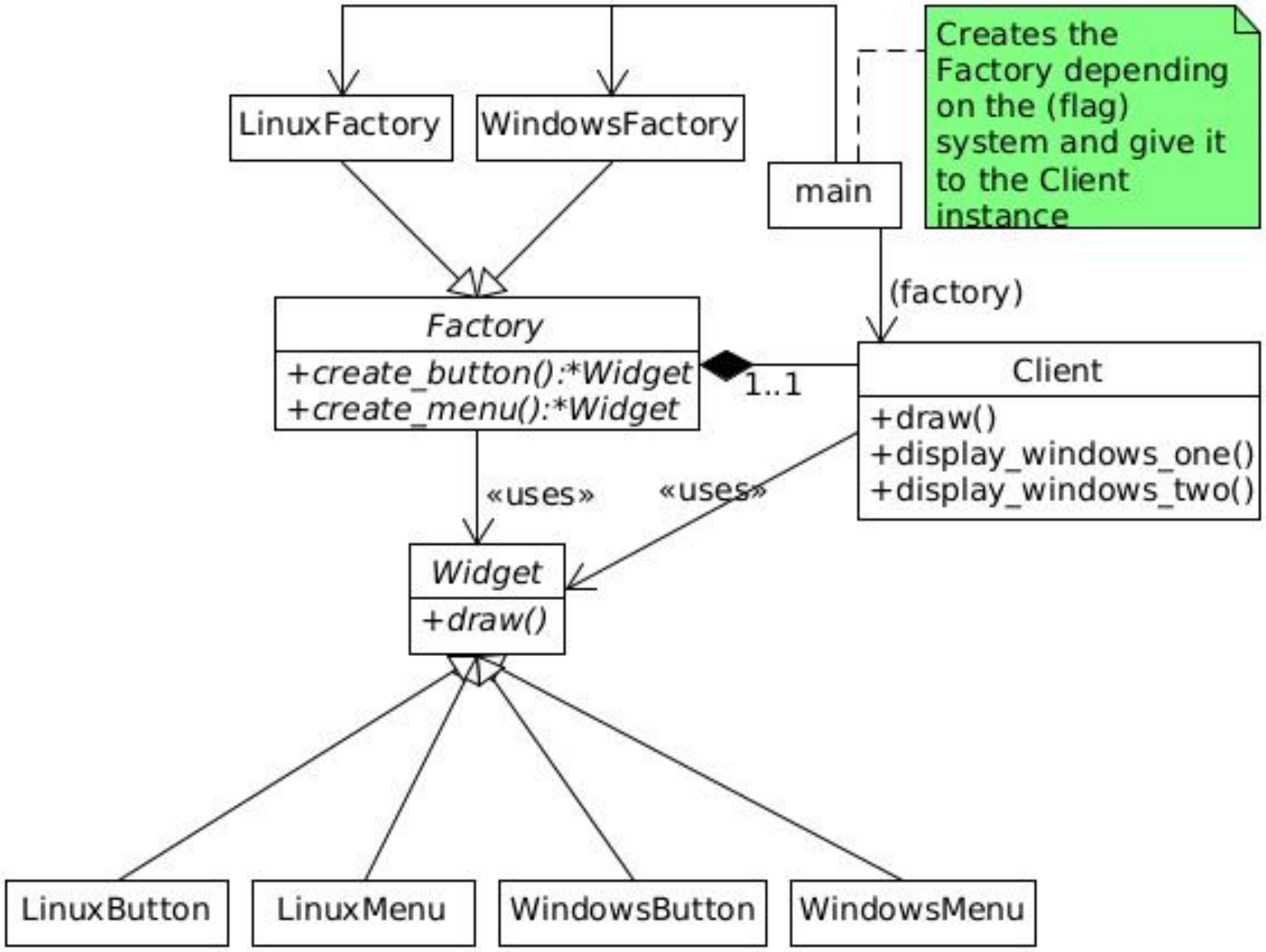
- Patrón usado cuando se requiere tener una única instancia de clase.
- Para crear instancias en C++ se puede usar el operador `new`.
  - Pero quizás necesario que no se permita crear más de una instancia.
  - E.g. el objeto `Balon` en el diseño de un juego de fútbol.
- Para asegurar una instancia $\rightarrow$ impedir clientes acceder a constructor.
  - ¿Cómo? Haciendo al constructor `private` o `protected`.
    - Si un cliente intenta crear instancia directa/ $\rightarrow$ error de compilación.
  - Y proporcionando un punto controlado por el que pedir la instancia.

```
#include <iostream>
using namespace std;

class Ball {
public:
  static Ball* getTheBall();
  void move (int x, int y);
  void showPos ();
  // singletons shoul not be cloneable or assignable:
  Ball (Ball &otherBall) = delete;
  void operator= (const Ball&) = delete;

protected: // this is the KEY: constructor is protected!
  Ball():posx(0),posy(0){}; // it will be called by getTheBall()

private:
  int posx, posy;
  static Ball* singleBall; // pointer to the single instance
};
```

```
#include "Ball.h"

Ball* Ball::singleBall = nullptr;

Ball* Ball::getTheBall () {
  if (singleBall == nullptr)
    singleBall = new Ball ();
  else
    cout << "Error: trying to get another instance of a Ball
        singleton class!\n";

  return singleBall;
}

// [...] move and showPos functions are omitted but they are
    defined as always
```

```
#include "Ball.h"

int main () {
  Ball* ball = Ball::getTheBall ();
  int posx = 0, posy = 0;

  cout << "You got the ball! Choose its position (x, y):\n";
  cout << "> ";
  cin >> posx;
  cout << "> ";
  cin >> posy;

  ball->move (posx, posy);
  ball->showPos();

  // Ball* anotherBall1 = Ball::getTheBall(); // exec time error!
  // Ball* anotherBall2 = new Ball (); // comp. time error!
}
```

- Permite crear diferentes tipos de instancias aislando al cliente de ello.
- Al crecer pograma, el n.º de clases y sus jerarquías también lo hace.
- Este patrón permite crear diferentes objetos con diferentes jerarquías.
  - E.g. al construir los diferentes personajes de un juego de rol.
    - Cada personaje tendrá determinadas restricciones y relaciones.
  - E.g. al crear el sistema de ventanas según S.O. de un programa.
    - Dependiendo del S.O. las jerarquías subyacentes de creación difieren.
    - Crear el GUI es transparente para el cliente $\rightarrow$ lo hace la factoría.


LinuxFactory
WindowsFactory
main
Creates the Factory depending on the (flag) system and give it to the Client instance
(factory)
Factory
+create_button():*Widget
+create_menu():*Widget
1..1
Client
+draw()
+display_windows_one()
+display_windows_two()
«uses»
«uses»
Widget
+draw()
LinuxButton
LinuxMenu
WindowsButton
WindowsMenu

```
#include "Client.h"
#include "LinuxFactory.h"
#include "WindowsFactory.h"
#define LINUX // set the flag to Linux platform

int main() {
  Factory *factory;
#ifdef LINUX // switch statement to create a proper factory
  factory = new LinuxFactory;
#else // a Windows platform is being used
  factory = new WindowsFactory;
#endif

  Client *c = new Client(factory); // creates a proper GUI system
  c->draw(); // client doesn't know which draw function is called
}
```

- Se basa en definir una interfaz para crear instancias de objetos.
- Permite a las subclases decidir cómo se crean tales instancias.
  - De forma transparente para el cliente (como *Abstract Factory*).
- E.g. motorización de un coche, según selección del cliente.

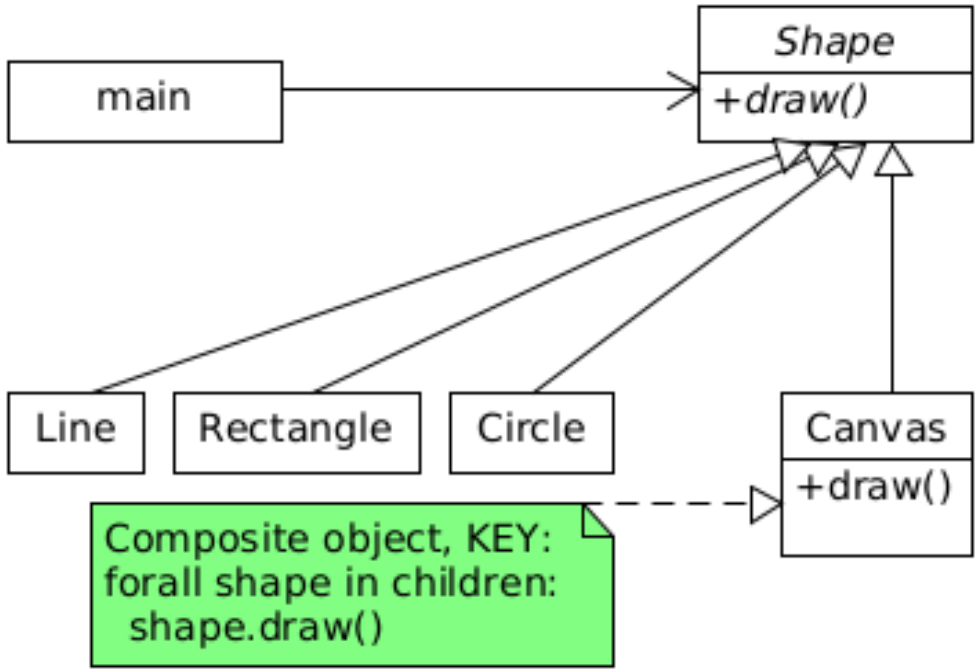
```
#include "Engine.h"

int main() {
  vector<Engine*> engines;
  int choice;
  while (true) {
    cout << "Choose engine: Gasoline=1 Diesel=2 Electric=3
        (Exit=0): ";
    cin >> choice;
    if (choice == 0) break;
    engines.push_back (Engine::makeEngine(choice));
    // factory method is called: depending on the engine, the
        object instance created will be different
  }
  for (int i = 0; i < engines.size(); i++)
    engines[i]->label();
  for (int i = 0; i < engines.size(); i++)
    delete engines[i];
}
```

- Proporciona abstracción cuando hay que crear diferentes objetos.
  - En un contexto en que se desconoce cuántos y cuáles van a ser.
- La idea ppal. es que los objetos deben poder clonarse en t.^{o} ejec.
- Los dos patrones anteriores usan herencia y mét. abstractos.
  - Implementados por subclases que construyen y definen a cada objeto.
- Esto es un problema cuando el n.^{o} de objetos es elevado o indeter/.
  - E.g. pensemos que un cliente puede configurar su motor como quiera.
    - Habría un gran n.^{o} indeter/ de tipos de motores que se podrían hacer...
    - La solución ideal aquí será el uso del patrón *Prototype*.

- La clase interfaz `Engine` sería el `Prototype`.
- La ppal. diferencia: nuevo método `clone ()` en todos los objetos.
  - ∄ *Factory Method*: `static Engine *makeEngine (int choice);`
  - Ahora tendremos: `virtual Engine* clone() const = 0;`
    - Y en cada subclase se implementa ese método devolviendo la instancia.
    - E.g. subclase Diesel: `Engine* clone() const{return new Diesel;}`
- No se necesita agente intermedio (factoría) para crear instancias.
  - La creación se realiza en la clase concreta que representa a la instancia.
  - Se puede tener factoría con los prototipos y esta invoque a `clone()`.
  - O se puede tener un gestor de prototipos. Hecho en nuestro ejemplo.
    - Que permita cargar/descargar los prototipos disponibles en t.^{o} ejec.
    - Solución interesante para diseños ampliables en t.^{o} ejec. (*plugins*).

```
#include "EngineManager.h"

Engine* EngineManager::engineTypes[] = {
  0, new Diesel, new Gasoline, new Electric
};

Engine* EngineManager::makeEngine (int choice) {
  return engineTypes[choice]->clone();
}
```

- Permite diseñar objetos como estructuras recursivas tipo árbol.
- Se puede trabajar con esta estructura como si fuera un único objeto.
- Los nodos hoja de la estructura son objetos sin hijos.
  - En nuestro ejemplo, son: `Line`, `Rectangle` y `Circle`.
- Los nodos intermedios serán objetos *Composite*.
  - En nuestro ejemplo, el único objeto *Composite* es `Canvas`.
- Una misma operación puede ser usada por ambos tipos de nodos.
  - En ejemplo, vemos cómo todos los objetos usan la función `draw()`.
- Aunque nodo hoja herede todos los métodos, no todos le serán *útil*.
  - En ejemplo, los nodos hoja heredan `add()`, `remove()` y `getChild()`.
  - Son heredados pero solo tienen sentido para el nodo *Composite*.


Shape
+draw()
main
Line
Rectangle
Circle
Canvas
+draw()
Composite object, KEY:
forall shape in children:
shape.draw()

```
int main() {
  Line l;
  l.draw(); // every leaf object can draw itself
  Rectangle r;
  r.draw();
  Circle c;
  c.draw();

  Canvas canvas; // it composes a complete structure...
  canvas.add(&l);
  canvas.add(&r);
  canvas.add(&c);
  canvas.add(&r);
  canvas.draw(); // ...and draw the structure at once!

  return 0;
}
```

- Permite modificar responsabilidad o propiedades de un obj. en t.^{o} ejec.
- Ofrece alternativas a las subclases para extender su funcionalidad.
- Ej.: supongamos que queremos añadir color a las distintas *shapes*.
  - No vamos a crear tres clases nuevas (¡¿y si fueran más?!).
    - `ColoredLine`, `ColoredRectangle` y `ColoredCircle`.
  - En su lugar, podemos crear una nueva clase `ColoredShape`.
    - Y que esta nos permita colorear las figuras que queramos.
- Tipos:
  - *Dynamic Decorator*: agrega al *decorated* obj. por ptr. o ref. en t.^{o} ejec.
    - Op. +común. Cuando no sabemos *a priori* qué objetos se decorarán.
    - Permite jerarquía de clases más flexible que la herencia estática.
  - *Static Decorator*: hereda del *decorated* object. (t.^{o} compilación).
    - Si conocemos de antemano qué objetos y cómo se decorarán.
    - Esta forma sería poco flexible y muy parecida al *Composite*.

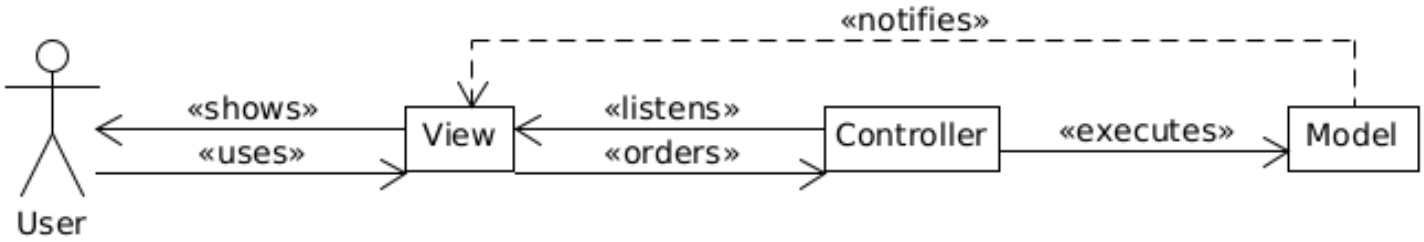
```
class ColoredShape : public Shape {
  public:
    ColoredShape (const Shape& s, const string &c) : shape{s},
        color{c} {
      cout << removeNumbers(typeid(s).name()) << " colored in "
          << c << endl;
    }
    void draw () const {
      cout << "Draw circle\n";
    }

  private:
    const Shape& shape;
    string color;
};
```

```
int main() {
  Line l; // basic objects
  l.draw();
  Rectangle r;
  r.draw();
  Circle c;
  c.draw();

  // dynamic because we decide at runtime which shape is colored
  ColoredShape blueL {l, "blue"}; // objects are colored
  ColoredShape greenR {r, "green"};
  ColoredShape redC {c, "red"};

  return 0;
}
```

- Este patrón eleva aún más el nivel de abstracción de un sistema.
- Oculta detalles de implementación para hacer más sencillo su uso.
- Proporciona un interfaz simplificado/unificado para el sistema.
  - Que oculta un complejo subsistema o subconjunto de interfaces.
- E.g. arrancar sistema complejo, como un coche, con solo girar llave.
  - Con ese gesto se activan, por debajo, otros subsistemas.
  - E.g. motor, sistemas frenos, luces, etc.

```
#include "LightSystem.h"
#include "BrakeSystem.h"

class Car { // Facade class
  public:
    void turnIgnitionOn () {
      engine.turnOn ();
      brakeSystem.turnOn ();
      lightSystem.turnOn ();
    }
  private:
    Engine engine;
    BrakeSystem brakeSystem;
    LightSystem lightSystem;
};
```

```
#include "Engine.h"
#include "Car.h"

int main() {
  Car myCar;
  myCar.turnIgnitionOn (); // very simple ignition

  return 0;
}
```



- Usado para aislar el dominio de aplicación del de presentación.
  - Dicho de otro modo: aislar la lógica del interfaz de un programa.
  - Así, una misma aplicación puede tener diferentes interfaces.
- En este patrón existen tres entidades bien diferenciadas:
  - **Modelo**: contiene únicamente los datos del programa.
  - **Vista**: interfaz de usuario que recibe órdenes de este.
    - Y también del controlador para mostrar info. o modificar interfaz.
  - **Controlador**: intermediario entre la vista y el modelo.
    - Recibe órdenes y traduce esa acción al dominio del modelo.
    - Estas órdenes son normal/ manejadas mediante *callbacks*.
    - Acciones: crear instancia obj., actualizar estados, pedir ops. al modelo...

```
// callback function: is called when data changes
typedef void (*DataChangedCallback)(string data);

class Model { // responsible for getting/setting data
public:
  Model ();
  Model (const string&);
  string getData ();
  void setData (const string&);
  void registerEvent (DataChangedCallback callback);

private:
  string data = "";
  DataChangedCallback event = nullptr;
};
```

```
class View { // responsible to show data to the user
public:
  View ();
  View (const Model&);
  void setModel (const Model&);
  void renderData ();
private:
  Model model;
};
```

```
#include "Model.h"
#include "View.h"

class Controller { // abstracts model from view and viceversa
public:
  Controller();
  Controller(const Model&, const View&);
  void setModel (const Model&);
  void setView (const View&);
  void boot();

private:
  Model model;
  View view;
};
```

```
void changeData (string data) {
  cout << "New data: " << data <<endl;
}

int main() {
  Model model("Model");
  View view(model);
  model.registerEvent (&changeData);

  Controller controller(model, view); // binds model and view

  controller.boot(); // app starts
  model.setData("Booting...");
  return 0;
}
```

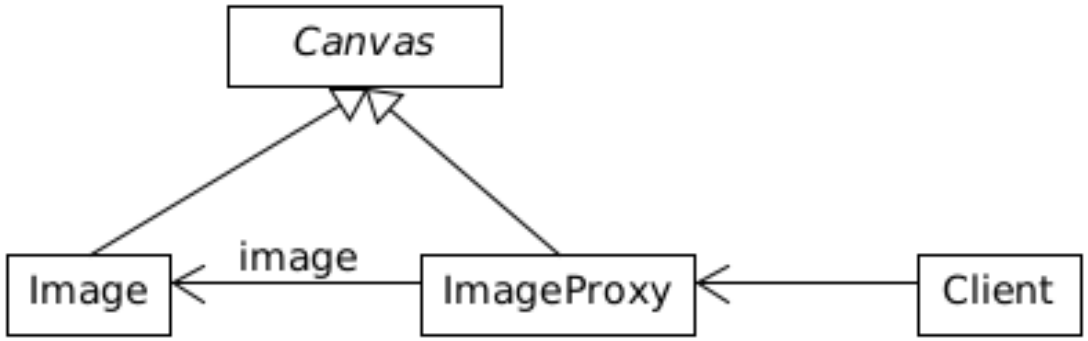
- Convierte el interfaz de una clase en otro demandado por cliente.
- Permite la comunicación entre clases con interfaces incompatibles.
- Muy útil si hay que usar necesaria/ una biblioteca externa.
  - Esta tendrá su interfaz cuya modificación supondría gran trabajo.
  - O que sea software privado sin posibilidad de modificar.
- Veamos ejemplo con adaptación de `LegacyRectangle` a `Rectangle`.

```
class Rectangle { // desidered interface (target)
  public:
    virtual void draw () = 0;
};
```

```
class LegacyRectangle { // legacy class (adaptee)
public:
  LegacyRectangle(int x1, int y1, int x2, int y2) {
    x1_ = x1;
    y1_ = y1;
    x2_ = x2;
    y2_ = y2;
    std::cout << "LegacyRectangle with [(x1,y1),(x2,y2)]
        coordinates is created\n";
  }
  void legacyDraw() {
    std::cout << "LegacyDraw rectangle is shown\n";
  }
private:
  int x1_;
  int y1_;
  int x2_;
  int y2_;
};
```

```
#include "Rectangle.h"
#include "LegacyRectangle.h"

// Adapter wrapper
class RectangleAdapter : public Rectangle, private
    LegacyRectangle {
public:
  RectangleAdapter (int x, int y, int w, int h):
    LegacyRectangle (x, y, x + w, y + h) {
      std::cout << "RectangleAdapter with [(x,y),(x+w,x+h)]
          coordinates is created\n";
    }

  void draw() {
    std::cout << "RectangleAdapter draw() method is called\n";
    legacyDraw ();
  }
};
```

```
int main() {
  int x = 10, y = 40, w = 320, h = 240;

  // I can use new interface (Rectangle) through the adapter
  Rectangle *r = new RectangleAdapter (x,y,w,h); // new coord.
      system

  r->draw(); // new drawing interface
  return 0;
}

/* Output:
LegacyRectangle with [(x1,y1),(x2,y2)] coordinates is created
RectangleAdapter with [(x,y),(x+w,x+h)] coordinates is created
RectangleAdapter draw() method is called
LegacyDraw rectangle is shown
*/
```



- Proporciona al cliente un objeto que actúa como sustituto del real.
- El obj. `Proxy` recibe petición y se la pasa al objeto real.
  - Y antes de pasarla hace algo: control de acceso, *caching*, etc.
  - El obj. real realiza casi todo el trabajo, el `Proxy`, controla acceso.
- El `Proxy` y el obj. real implementan un mismo interfaz.
  - Lo que permite al cliente tratar al `Proxy` como al obj. real.
- Uso: si se necesita añadir comporta/ a obj. sin modificar su clase.
- Ejemplo: mostrar una imagen cuya carga es costosa computacional/.
  - Sol.: `ImageProxy` que representa al obj. real `Image`.
  - Comporta/ modificado: obj. proxy puede cargar una sola vez imagen...
    - ...y la muestra tantas veces como solicite el cliente.

```
class Canvas {
public:
   virtual void request() = 0;
   virtual ~Canvas() {}
};
#endif
```

```
class Image : public Canvas {
public:
   void request() {
      cout << "Image (real subject): request()" << endl;
   }
};
```

```
#include "Canvas.h"
#include "ImageProxy.h"
#include "Image.h"

class ImageProxy : public Canvas {
private:
   Canvas* image; // ptr to the subject (canvas)
public:
   ImageProxy() : image (new Image()) {}
   ~ImageProxy() {
      delete image;
   }

   void request() { // forward calls to the (real) Image
      image->request();
   }
};
```

```
#include "ImageProxy.h"

int main() {
   ImageProxy imageProxy;
   imageProxy.request();
}
```

11. Patrones de diseño

Julio Vega

julio.vega@urjc.es