Resumen Segundo Parcial

Tomás Felipe Melli

July 2, 2025

Índice

1 Modern Software Engineering - David Farley

David Farley es un reconocido experto en desarrollo de software, especialmente en prácticas relacionadas con entrega continua (*continuous delivery*) y metodologías ágiles.

1.1 Capítulo 1: Engineering—The Practical Application of Science

David sostiene que el desarrollo de software es un proceso de descubrimiento y exploración, por lo que los ingenieros de software deben volverse expertos en aprender. Para ello, se propone adoptar los principios básicos del método científico (caracterizar, *hipotetizar*, predecir, experimentar) de forma pragmática. No se trata de aplicar ciencia con precisión extrema, sino de usar sus estrategias para reducir riesgos, mejorar la toma de decisiones y avanzar más rápido.

1.1.1 What Is Software Engineering?

La definición propuesta:

La aplicación de un enfoque empírico y científico para encontrar soluciones eficientes y económicas a problemas prácticos en software.

Este enfoque reconoce que el desarrollo es un ejercicio de descubrimiento y aprendizaje continuo. Para ser eficientes:

[noitemsep]**Aprender**: mediante *iteración, retroalimentación, incrementalismo, experimentación y empirismo.* **Gestionar la complejidad**: con *modularidad, cohesión, separación de responsabilidades, abstracción y bajo acoplamiento.* **Herramientas prácticas**: *testabilidad, posibilidad de desplegar, velocidad, control de variables y entrega continua.*

Aplicar estos principios lleva a software de mayor calidad, menos estrés y mejores resultados.

1.1.2 Reclaiming "Software Engineering"

Farley sostiene que la industria ha desvirtuado el término "ingeniería" en software. En otras disciplinas, ingeniería implica "cosas que funcionan", un proceso orientado al éxito. Si nuestras prácticas no nos permiten desarrollar mejor software más rápido, entonces no estamos haciendo verdadera ingeniería.

1.1.3 How to Make Progress

El desarrollo de software es complejo. Para avanzar como industria, debemos adoptar principios comunes y mantener la libertad intelectual para cuestionar y reemplazar ideas ineficaces. La ciencia es ese enfoque que permite mejorar con evidencia.

1.1.4 The Birth of Software Engineering

La ingeniería de software nació en los 60s, con Margaret Hamilton y la conferencia de la OTAN sobre la "crisis del software". Fred Brooks, con su artículo "No Silver Bullet", destacó que el problema no era la lentitud del software, sino la rapidez del hardware. El software no ha mejorado al mismo ritmo.

1.1.5 Shifting the Paradigm

El cambio de paradigma implica abandonar ideas obsoletas. Tratar el software como una disciplina de ingeniería basada en el método científico permite aprender más y mejor. Esta visión representa un nuevo paradigma.

1.2 Capítulo 2: What Is Engineering?

Se discute la idea de que "hacer software no es como construir puentes". Aunque es cierto, también hay mucho desconocimiento sobre lo que implica realmente construir puentes. Se confunde ingeniería de producción con ingeniería de diseño.

1.2.1 Production Is Not Our Problem

A diferencia de otras industrias, en software la producción no es el problema. Producir software debería ser casi gratuito. El reto está en el diseño, no en la producción.

1.2.2 Design Engineering, Not Production Engineering

El software se puede cambiar y probar rápidamente. No requiere simulaciones; el producto es también el modelo. Esto permite un ciclo de prueba y error mucho más eficiente.

Ejemplo clave: Margaret Hamilton y el software del Apollo 11. Su enfoque de "failing safely" salvó la misión. El sistema estaba preparado para recuperarse de errores críticos en tiempo real.

1.3 Capítulo 3: Fundamentals of an Engineering Approach

Todas las disciplinas de ingeniería se basan en el racionalismo científico. La ingeniería de software debe identificar principios duraderos que reflejen la realidad del desarrollo.

1.3.1 An Industry of Change?

Aunque la industria del software cambia constantemente, no siempre mejora. Ejemplo: usar Hibernate fue más complejo que SQL directo. A menudo adoptamos modas sin evidencia de mejora.

1.3.2 The Importance of Measurement

No mejoramos porque no medimos correctamente. Métricas como velocidad o líneas de código no sirven. Estudios como *Accelerate* proponen medir **stability** y **throughput**, correlacionadas con mejores resultados.

[noitemsep]**Stability**: tasa de fallos y tiempo de recuperación. **Throughput**: velocidad y frecuencia de despliegue.

Los equipos con alto rendimiento comparten prácticas como tests automatizados, integración continua y desarrollo basado en la rama principal.

1.3.3 Applying Stability and Throughput

Estas métricas permiten evaluar cambios con datos. Por ejemplo, las juntas de aprobación (*Change Approval Boards*) no mejoran la calidad y empeoran el rendimiento.

1.3.4 The Foundations of a Software Engineering Discipline

Las ideas fundamentales se dividen en dos ejes:

[noitemsep]Enfoque filosófico y procesos. Técnicas de diseño y arquitectura.

Debemos aprender a aprender y aceptar que desarrollamos sistemas complejos que requieren buena gestión técnica y organizacional.

1.3.5 Experts at Learning

Cinco prácticas fundamentales del enfoque científico aplicado al software:

[noitemsep]Iteración Feedback rápido Incrementalismo Experimentación Empirismo

Estas prácticas explican por qué enfoques como *waterfall* no funcionan bien hoy.

1.3.6 Experts at Managing Complexity

La complejidad se manifiesta tanto en lo técnico como en lo organizacional. Para controlarla, debemos aplicar:

[noitemsep]Modularidad Cohesión Separación de responsabilidades Abstracción / ocultamiento de información Bajo acoplamiento

Estos principios nos ayudan a evitar el desorden y a construir sistemas más sostenibles.

2 Pattern Abuser

2.1 Pat's Confession

Pat cuenta que era un programador común que trabajaba mucho para que su código funcionara, sin técnicas de diseño formales, solo modificando hasta que el código servía.

Todo cambió cuando llegó Jerry (a quien llaman JERR), un diseñador experto que les presentó el libro Gang of Four con patrones de diseño. JERR les mostró cómo aplicar patrones para mejorar el código: hacerlo más entendible, modificable, testeable y mantenible.

Pat se volvió fanático de los patrones: memorizó todos, reescribió todo su código usando patrones, y empezó a exigir a los demás que hicieran lo mismo. Organizó revisiones de código que generaban miedo, porque criticaba duramente la ausencia de patrones.

Gracias a esto, convenció a los gerentes de que más patrones significaban mejor reutilización y calidad, lo que hizo que su proyecto fuera reconocido como el mejor de la empresa. Pat pasó de programador a diseñador estrella y consultor famoso, invitado a conferencias y contratado por muchas empresas.

Sin embargo, la obsesión de Pat creció al punto que quería aplicar la mayor cantidad posible de patrones a una sola clase, sin importar las consecuencias al código. Desarrolló su propia "técnica" para aplicar muchos patrones a un mismo componente, y planea mostrar un ejemplo con clases de cuentas bancarias.

Aplicación de Patrones de Diseño a una Jerarquía de Clases (Ejemplo con Cuentas Bancarias)

- Strategy Pattern para Métodos
  - Aplicar el Strategy Pattern a métodos como calcInterest.
  - Crear una clase estrategia InterestCalculator con métodos para calcular interés.
  - Definir subclases para distintos tipos de interés (simple, compuesto, fijo, variable).
  - La clase Account (o su estado) contiene una referencia a la estrategia específica para calcular el interés.
- Chain of Responsibility para Agrupaciones
  - Usar Chain of Responsibility para decidir dinámicamente qué instancia maneja una solicitud.
  - Por ejemplo, para una AccountPortfolio que agrupa cuentas, determinar desde qué cuenta retirar dinero cuando hay sobregiro, buscando en la cadena la cuenta con saldo suficiente.
- Composite, Iterator y Visitor para Agrupaciones
  - Composite organiza cuentas en una estructura árbol (nodos grupo y hojas).
  - Iterator permite iterar sobre las cuentas en grupos.
  - Visitor separa operaciones (como imprimir estados, calcular saldos) del objeto cuenta.
  - Agrupación posible: por tipo de cuenta, tipo de interés, tipo de cliente, etc.
- Observer para Clientes Dependientes
  - Clientes se registran para observar cambios en cuentas o portafolios.
  - Cuando una cuenta cambia (ej. saldo), notifica a los observadores, quienes actualizan sus vistas o datos.
- Proxy para Acceso Controlado y Distribuido
  - Crear proxies para Account y AccountPortfolio que representen servicios (e.g., cajero automático).
  - Proxies aceptan mensajes, invocan métodos reales y devuelven resultados, facilitando control o acceso remoto.
- Facade para Simplificar Interfaces
  - Definir una fachada (Account Facade) que agrupe la complejidad de múltiples clases y patrones aplicados, exponiendo una interfaz unificada (retirar, depositar, transferir, consultar saldos).
- Bridge para Métodos Complejos
  - Separar la interfaz del cálculo de interés de su implementación, especialmente si hay varias formas de calcular el mismo tipo (e.g., diferentes formas de interés compuesto).
- Adapter para Objetos Legados
  - Adaptar objetos antiguos para que cumplan con la nueva interfaz y puedan integrarse al sistema actual sin problemas.
- Abstract Factory, Builder y Factory Method para Creación de Objetos Familiares
  - Crear familias de objetos relacionados (Customer, AccountPortfolio, Account) usando estos patrones para controlar la creación y ensamblado.
  - El cliente controla el ensamblaje (Abstract Factory) o el patrón devuelve un producto final (Builder).
- Prototype para Crear Objetos Similares
  - Clonar prototipos para crear nuevas cuentas en una cartera, facilitando la creación rápida basada en un objeto modelo.

2.2 The Dark

Pat confiesa que, aunque al principio disfrutó del prestigio, fama y dinero que le trajo aplicar patrones en sus proyectos, empezó a cuestionar si realmente mejoraba los sistemas que construía.

En el caso del banco, aunque quedaron satisfechos que Pat aplicara muchos patrones, pronto los desarrolladores tuvieron dificultades para hacer cambios sencillos en el sistema. Pat justificó esto diciendo que los desarrolladores debían "madurar" en el uso de patrones y que la solución era agregar aún más patrones, lo que le valió ser contratado de nuevo.

Sin embargo, Pat se siente cada vez más frustrado y deprimido porque se da cuenta de que abusar de los patrones genera problemas: no sabe cuándo un patrón realmente aporta valor ni cómo medir si el sistema es más reutilizable o "mejor" tras su aplicación. Se cuestiona cómo definir y determinar si una aplicación ha mejorado.

Pat pide ayuda para resolver estas dudas.

3 Adapter Pattern

3.1 Intent

El **intent** es lo que define a un patrón. Entender esto nos va a permitir poder diferenciar uno de otro. En el caso del **Adapter** la intención es **convertir la interfaz de una clase en otra que el cliente espera. Permite que clases con interfaces incompatibles trabajen juntas.**

3.2 Motivation

Una aplicación puede necesitar usar clases existentes cuya interfaz no coincide con la requerida por el sistema. Por ejemplo, integrar una clase `TextView` dentro de un sistema gráfico que trabaja con la interfaz `Shape`. Un `TextShape` actúa como adaptador, permitiendo que `TextView` se utilice como si fuera una `Shape`.

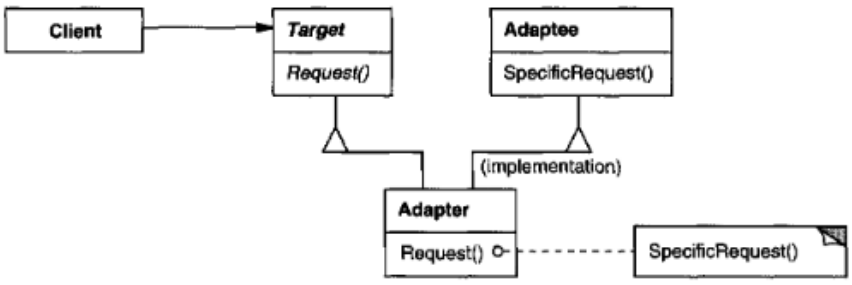
3.3 Applicability

Usar el patrón Adapter cuando:

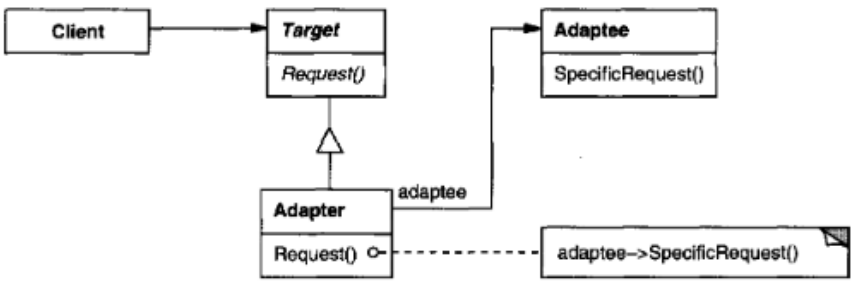
- Se quiere usar una clase existente pero su interfaz no coincide con la requerida.
- Se necesita reutilizar varias subclases existentes con interfaces incompatibles.

3.4 Structure

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



3.5 Participants

- **Target**: Interfaz esperada por el cliente (ej. `Shape`).
- **Client**: Utiliza objetos que siguen la interfaz Target (ej. `DrawingEditor`).
- **Adaptee**: Clase existente con interfaz incompatible (ej. `TextView`).
- **Adapter**: Convierte la interfaz del Adaptee en la del Target (ej. `TextShape`).

3.6 Example

Contexto del problema

El sistema tiene un editor gráfico que trabaja con figuras (`Shape`). Queremos reutilizar una clase existente llamada `TextView`, que muestra texto en pantalla, pero no implementa la interfaz requerida por `Shape`. No podemos modificar `TextView`.

Interfaz esperada: Shape

El editor espera que las figuras implementen la siguiente interfaz:

```
class Shape {
public:
    virtual void BoundingBox(Point& bottomLeft, Point& topRight) const = 0;
    virtual bool IsEmpty() const = 0;
    virtual Manipulator* CreateManipulator() const = 0;
    virtual ~Shape() = default;
};
```

`BoundingBox` devuelve el área ocupada. `IsEmpty` indica si hay contenido. `CreateManipulator` crea una herramienta para manipular la figura.

Clase existente: TextView

`TextView` tiene su propia interfaz, incompatible con `Shape`:

```
class TextView {
public:
    void GetOrigin(int& x, int& y) const;
    void GetExtent(int& width, int& height) const;
    bool IsEmpty() const;
};
```

Por ejemplo, no usa puntos ni retorna ‘BoundingBox‘ directamente.

Solución: Adapter TextShape

Creamos una clase `TextShape` que actúa como adaptador:

```
class TextShape : public Shape {
private:
    TextView* _text;

public:
    TextShape(TextView* t) : _text(t) {}

    void BoundingBox(Point& bottomLeft, Point& topRight) const override {
        int x, y, width, height;
        _text->GetOrigin(x, y);
        _text->GetExtent(width, height);
        bottomLeft = Point(x, y);
        topRight = Point(x + width, y + height);
    }

    bool IsEmpty() const override {
        return _text->IsEmpty();
    }

    Manipulator* CreateManipulator() const override {
        return new TextManipulator(this);
    }
};
```

Notas clave

- `TextShape` implementa la interfaz `Shape`.
- Usa composición: guarda internamente un puntero a `TextView`.
- Traduce llamadas desde la interfaz esperada hacia la clase adaptada.

Resultado

Ahora el editor puede trabajar con instancias de `TextShape` como si fueran figuras normales:

```
TextView* tv = new TextView();
Shape* shape = new TextShape(tv);

Point bl, tr;
shape->BoundingBox(bl, tr);
//   Funciona   como un Shape, aunque es un TextView adaptado!
```

4 Composite Pattern

4.1 Intent

El **Composite** permite componer objetos en estructuras de árbol para representar jerarquías. Permite tratar objetos individuales y composiciones de objetos de manera uniforme.

4.2 Motivation

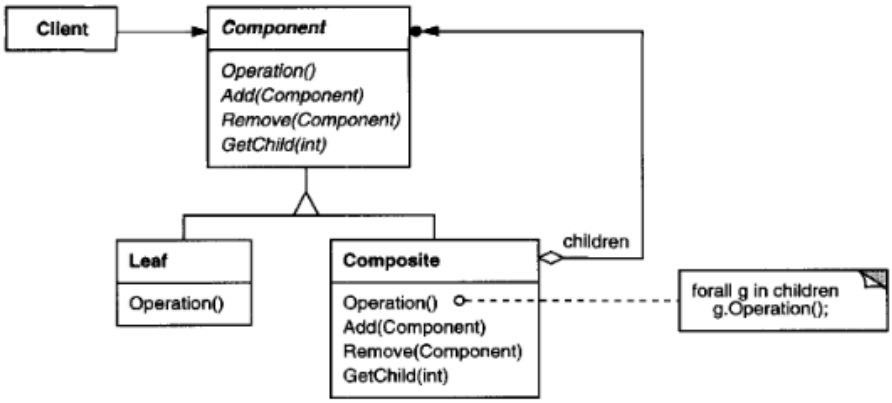
En aplicaciones que manipulan estructuras jerárquicas (como gráficos, archivos, GUI), se necesita poder tratar elementos simples (hojas) y complejos (compuestos) de forma similar. Por ejemplo, un dibujo puede estar compuesto de líneas, rectángulos o incluso de otros dibujos. El patrón Composite permite que el cliente no tenga que distinguir si está trabajando con un objeto simple o compuesto.

4.3 Applicability

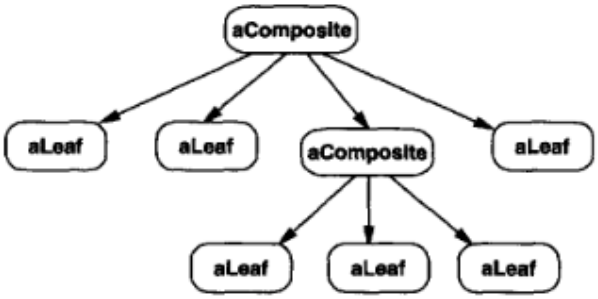
Usar Composite cuando:

- Se quiere representar jerarquías de objetos.
- Se quiere tratar objetos individuales y grupos de manera uniforme.
- Se desea que clientes ignoren diferencias entre composiciones y objetos individuales.

4.4 Structure



A typical Composite object structure might look like this:



4.5 Participants

- **Component**: Declara la interfaz para todos los objetos de la composición e implementa el comportamiento por default de la interfaz común a todas las clases.
- **Leaf**: Representa objetos simples (sin hijos). Implementa el comportamiento base.
- **Composite**: Representa objetos compuestos (con hijos). Almacena y administra los hijos. Implementa las operaciones de ‘Component‘.
- **Client**: Usa la interfaz ‘Component‘ para interactuar con objetos simples y compuestos.

4.6 Example

Component

Queremos construir un editor gráfico que permita manipular distintos elementos visuales como líneas, rectángulos o textos. Algunos de estos elementos pueden contener otros elementos (por ejemplo, un grupo de objetos). El desafío es poder tratar todos estos elementos —ya sean simples o compuestos— de la misma forma en el código.

La solución es usar el patrón **Composite** para que todos los objetos compartan una interfaz común. Así, el cliente puede trabajar con una línea individual o con un grupo de líneas sin preocuparse por la diferencia.

```
class Graphic {
public:
    virtual void Draw() = 0;
    virtual void Add(Graphic* g) {}
    virtual void Remove(Graphic* g) {}
    virtual Graphic* GetChild(int index) { return nullptr; }
    virtual ~Graphic() {}
};
```

Leaf

```
class Line : public Graphic {
public:
    void Draw() override {
        // Dibuja una l nea
    }
};
```

Composite

```
#include <vector>

class Picture : public Graphic {
private:
    std::vector<Graphic*> children;

public:
    void Draw() override {
        for (Graphic* g : children) {
            g->Draw();
        }
    }

    void Add(Graphic* g) override {
        children.push_back(g);
    }

    void Remove(Graphic* g) override {
        // L gica para eliminar g de children
    }

    Graphic* GetChild(int index) override {
        return children.at(index);
    }
};
```

Resultado

El cliente puede trabajar con cualquier objeto que implemente la interfaz ‘Graphic‘, ya sea una ‘Line‘, ‘Rectangle‘ o una composición como ‘Picture‘.

```
Picture* drawing = new Picture();
drawing->Add(new Line());
drawing->Add(new Picture()); // otra subcomposici n

drawing->Draw(); // Dibuja todo, sin importar qu  contiene
```

5 Decorator Patter

5.1 Intent

El patrón **Decorator** agrega responsabilidades adicionales a un objeto de forma dinámica. Es una alternativa flexible a la herencia para extender funcionalidad. También se conoce como **Wrapper**.

5.2 Motivation

Supongamos que tenemos una clase `TextView` que dibuja texto. Queremos poder agregarle funcionalidades como borde, scroll, sombra, etc., sin modificar su código ni crear muchas subclases como:

`BordeadoYConScrollTextView`, `ConSombraYConScrollTextView`, etc.

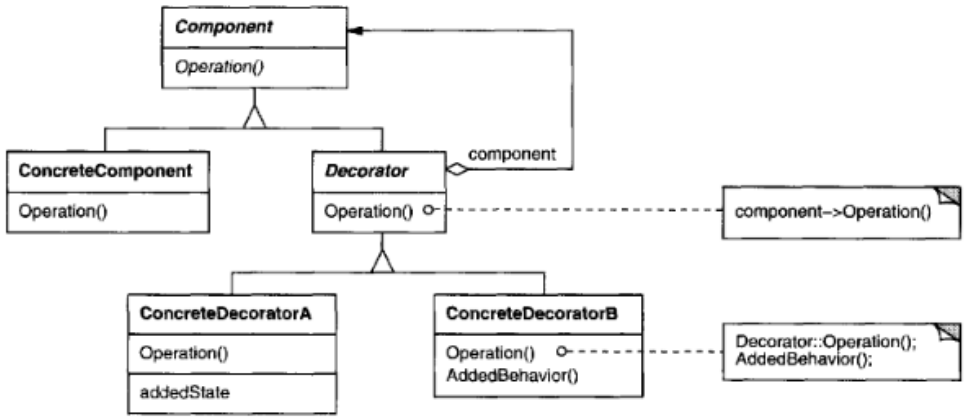
El patrón Decorator resuelve esto permitiendo componer dinámicamente decoraciones sobre un objeto base, sin alterar su clase original.

5.3 Applicability

Usar Decorator cuando:

- Querés agregar funcionalidades a objetos sin cambiar su clase.
- La herencia no es posible o adecuada.
- Necesitás combinar comportamientos de forma flexible.

5.4 Structure



5.5 Participants

- **Component**: Interfaz común para los objetos que pueden tener responsabilidades adicionales (`VisualComponent`).
- **ConcreteComponent**: Implementación básica del componente original (`TextView`).
- **Decorator**: Clase abstracta que mantiene una referencia al componente y define una interfaz compatible.
- **ConcreteDecorator**: Agrega responsabilidades concretas (`BorderDecorator`, `ScrollDecorator`).

5.6 Example

Contexto del problema

Tenemos un componente visual `TextView` que puede ser mostrado en una ventana. Queremos agregarle funcionalidades como bordes o scroll, sin modificar su clase ni usar herencia múltiple.

La solución es envolverlo con decoradores que agreguen las funcionalidades deseadas.

Componente base

```
class VisualComponent {
public:
    virtual void Draw() = 0;
    virtual ~VisualComponent() {}
};
```

Componente concreto

```
class TextView : public VisualComponent {
public:
    void Draw() override {
        // Dibuja el texto
    }
};
```

Clase Decorator

```
class Decorator : public VisualComponent {
protected:
    VisualComponent* component;

public:
    Decorator(VisualComponent* comp) : component(comp) {}

    void Draw() override {
        component->Draw(); // delega
    }
};
```

Decoradores concretos

```
class BorderDecorator : public Decorator {
private:
    int width;

public:
    BorderDecorator(VisualComponent* comp, int w)
        : Decorator(comp), width(w) {}

    void Draw() override {
        Decorator::Draw();  // dibuja el componente base
        DrawBorder(width);  // agrega borde
    }

    void DrawBorder(int w) {
        // C digo para dibujar borde
    }
};

class ScrollDecorator : public Decorator {
public:
    ScrollDecorator(VisualComponent* comp)
        : Decorator(comp) {}

    void Draw() override {
        Decorator::Draw();  // dibuja componente base
        DrawScroll();       // agrega scroll
    }

    void DrawScroll() {
        // C digo para scroll
    }
};
```

Resultado

Composición dinámica de decoradores:

```
TextView* textView = new TextView();

// Componemos decoradores
VisualComponent* component =
    new BorderDecorator(
        new ScrollDecorator(textView), 1);

component->Draw(); // Dibuja el TextView con scroll y borde
```

6 Proxy Pattern

6.1 Intent

El patrón **Proxy** proporciona un sustituto u objeto representante (proxy) de otro objeto para controlar el acceso a él. También se conoce como **Surrogate**.

6.2 Motivation

Queremos insertar imágenes en un documento de texto, pero cargarlas en memoria solo cuando sea necesario (por ejemplo, cuando se dibujan por primera vez). El objeto `ImageProxy` actúa como intermediario entre el documento y el objeto `Image` real.

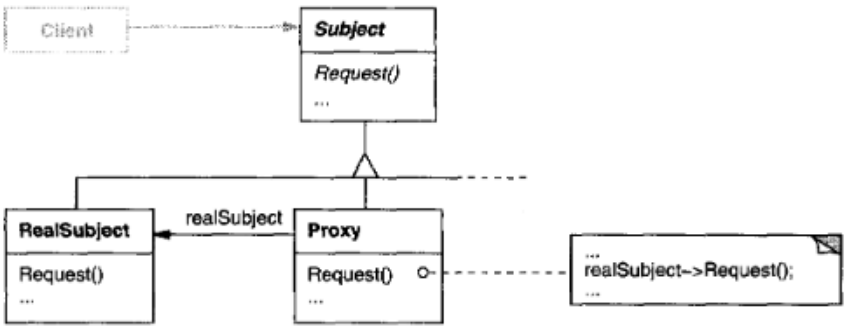
El proxy guarda la ruta al archivo, las dimensiones de la imagen, y crea la instancia real solo cuando se necesita.

6.3 Applicability

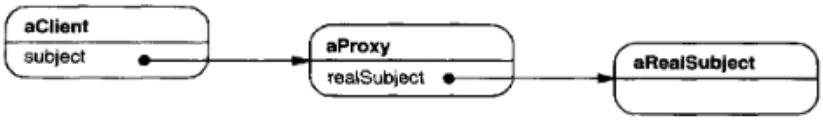
Usar Proxy cuando:

- Querés controlar el acceso a un objeto complejo o costoso de instanciar.
- Querés agregar lógica adicional al acceder al objeto (como autenticación, carga diferida, o control remoto).
- Querés proteger o encapsular el acceso a un objeto real desde el cliente.

6.4 Structure



Here's a possible object diagram of a proxy structure at run-time:



6.5 Participants

- **Proxy**: (`ImageProxy`)
  - Mantiene una referencia al `RealSubject`.
  - Implementa la interfaz `Subject` para que el cliente no note la diferencia.
  - Controla el acceso al objeto real. Por ejemplo:
    - Creando el objeto sólo cuando es necesario (proxy virtual).
    - Verificando permisos de acceso (proxy de protección).
    - Controlando acceso remoto (proxy remoto).
    - Añadiendo conteo de referencias u otras funcionalidades (proxy inteligente).
  - Puede realizar tareas antes o después de delegar la solicitud al objeto real.
- **Subject**: Declara la interfaz común entre el Proxy y el RealSubject. (`Graphic`)
- **RealSubject**: El objeto real al cual el proxy representa (`Image`).

6.6 Example

Contexto del problema

Un editor de texto permite insertar imágenes en un documento, pero muchas imágenes pueden ocupar memoria innecesariamente si se cargan todas de entrada. Queremos posponer la carga de la imagen real hasta que se necesite dibujarla.

Interfaz común (Subject)

```
class Graphic {
public:
    virtual void Draw(const Point& position) = 0;
    virtual Point GetExtent() = 0;
    virtual ~Graphic() {}
};
```

Objeto real (RealSubject)

```
class Image : public Graphic {
public:
    Image(const std::string& fileName);

    void Draw(const Point& position) override {
        // C digo para dibujar la imagen real
    }

    Point GetExtent() override {
        // Retorna el tama o real
    }
};
```

Proxy

```
class ImageProxy : public Graphic {
private:
    Image* _image;
    std::string _fileName;
    Point _extent;

public:
    ImageProxy(const std::string& fileName)
        : _image(nullptr), _fileName(fileName), _extent(100, 100) {}

    void Draw(const Point& position) override {
        LoadImage()->Draw(position);
    }

    Point GetExtent() override {
        return _extent; // Valor estimado antes de cargar
    }

private:
    Image* LoadImage() {
        if (!_image) {
            _image = new Image(_fileName);
        }
        return _image;
    }
};
```

Resultado

El cliente usa el objeto ‘Graphic‘, que puede ser un ‘Image‘ o un ‘ImageProxy‘ indistintamente:

```
Graphic* img = new ImageProxy("foto.jpg");

// No carga a n la imagen
Point size = img->GetExtent();

// Ahora s  la carga y la dibuja
img->Draw(Point(50, 100));
```

7 Builder Pattern

7.1 Intent

Separar la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear diferentes representaciones.

7.2 Motivation

Se quiere que un lector de documentos RTF pueda convertir textos RTF a múltiples formatos, como ASCII, TeX o widgets editables. Dado que la cantidad de formatos posibles es abierta, es importante poder agregar nuevas conversiones sin modificar el lector.

La solución es desacoplar el lector del formato de salida, configurándolo con un objeto TextConverter que se encargue de la conversión. Mientras el RTFReader analiza el documento, delega la conversión de los tokens a TextConverter.

Cada subclase de TextConverter maneja un formato específico (por ejemplo, ASCIIConverter, TeXConverter, TextWidgetConverter). Estas clases encapsulan la lógica de construcción de la representación final detrás de una interfaz abstracta.

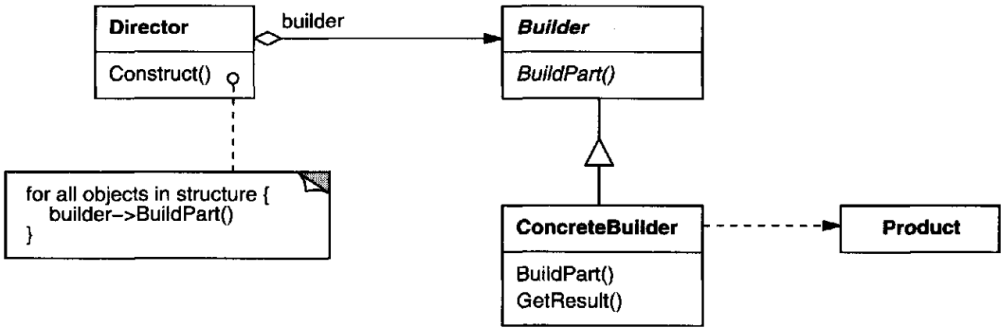
Este diseño aplica el patrón Builder, donde el lector (director) se encarga del análisis, y los convertidores (builders) se encargan de construir distintas representaciones del texto. Así, el mismo proceso de parsing puede generar múltiples salidas configurando el lector con distintos builders.

7.3 Applicability

Usar Builder cuando:

- El algoritmo para crear un objeto complejo debe ser independiente de las partes que lo componen y de cómo se ensamblan.
- El proceso de construcción debe permitir distintas representaciones del objeto construido.

7.4 Structure



7.5 Participants

- Builder (TextConverter): define una interfaz abstracta para crear partes de un objeto complejo (producto).
- ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter): implementa la interfaz del builder, construye y ensambla las partes del producto, mantiene su representación interna y ofrece una interfaz para obtener el producto final.
- Director (RTFReader): usa el builder para construir el objeto paso a paso.
- Product (ASCIIText, TeXText, TextWidget): es el objeto complejo que se está construyendo. Sus partes son definidas por el ConcreteBuilder, que controla cómo se ensamblan.

7.6 Example

Contexto del problema

Interfaz Builder (MazeBuilder)

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
    virtual Maze* GetMaze() { return nullptr; }

protected:
    MazeBuilder() = default;
};
```

Director (MazeGame)

```
class MazeGame {
public:
    Maze* CreateMaze(MazeBuilder& builder) {
        builder.BuildMaze();
        builder.BuildRoom(1);
        builder.BuildRoom(2);
        builder.BuildDoor(1, 2);
        return builder.GetMaze();
    }

    Maze* CreateComplexMaze(MazeBuilder& builder) {
        for (int i = 1; i <= 1001; ++i) {
            builder.BuildRoom(i);
        }
        return builder.GetMaze();
    }
};
```

Concrete Builder (StandardMazeBuilder)

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder() : _currentMaze(nullptr) { }

    void BuildMaze() override {
        _currentMaze = new Maze;
    }

    void BuildRoom(int n) override {
        if (!_currentMaze->RoomNo(n)) {
            Room* room = new Room(n);
            _currentMaze->AddRoom(room);
            room->SetSide(North, new Wall);
            room->SetSide(South, new Wall);
            room->SetSide(East,  new Wall);
            room->SetSide(West,  new Wall);
        }
    }

    void BuildDoor(int n1, int n2) override {
        Room* r1 = _currentMaze->RoomNo(n1);
        Room* r2 = _currentMaze->RoomNo(n2);
        Door* d = new Door(r1, r2);
        r1->SetSide(CommonWall(r1, r2), d);
        r2->SetSide(CommonWall(r2, r1), d);
    }

    Maze* GetMaze() override {
        return _currentMaze;
    }

private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

Concrete Builder Alternativo (CountingMazeBuilder)

Este no lo construye sólo cuenta las habitaciones y puertas.

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder() : _rooms(0), _doors(0) { }

    void BuildRoom(int) override {
        _rooms++;
    }

    void BuildDoor(int, int) override {
        _doors++;
    }

    void GetCounts(int& rooms, int& doors) const {
        rooms = _rooms;
        doors = _doors;
    }

private:
    int _rooms;
    int _doors;
};
```

Resultado

Uso del contador

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder); // realiza el "dibujo" virtual
builder.GetCounts(rooms, doors);

cout << "The maze has " << rooms << " rooms and "
     << doors << " doors" << endl;
```

Uso del patrón completo

```
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
Maze* maze = builder.GetMaze();
```

8 Visitor Pattern

8.1 Intent

Representar una operación que se realiza sobre los elementos de una estructura de objetos. El patrón Visitor permite definir nuevas operaciones sin modificar las clases sobre las que opera.

8.2 Motivation

Un compilador representa programas como árboles de sintaxis abstracta (AST) y debe realizar muchas operaciones sobre ellos: análisis semánticos, generación de código, optimizaciones, impresión, métricas, etc. Estas operaciones tratan distintos tipos de nodos (asignaciones, variables, expresiones) de forma diferente, por lo que cada tipo de nodo es una clase distinta.

Distribuir todas estas operaciones dentro de las clases de los nodos hace el sistema difícil de mantener, entender y extender, ya que el código de operaciones muy distintas (tipo-checking, optimización, impresión) queda mezclado. Además, agregar una nueva operación implica modificar y recompilar las clases de nodos.

La solución es encapsular cada operación en un objeto separado llamado visitor y hacer que los nodos acepten un visitor. Al aceptar el visitor, el nodo llama al método específico del visitor para su tipo (por ejemplo, VisitAssignment para nodos de asignación). Esto separa la estructura de los nodos de las operaciones que se les aplican.

Con el patrón Visitor, se mantienen dos jerarquías:

- La jerarquía de elementos (nodos del AST).
- La jerarquía de visitors (operaciones sobre nodos).

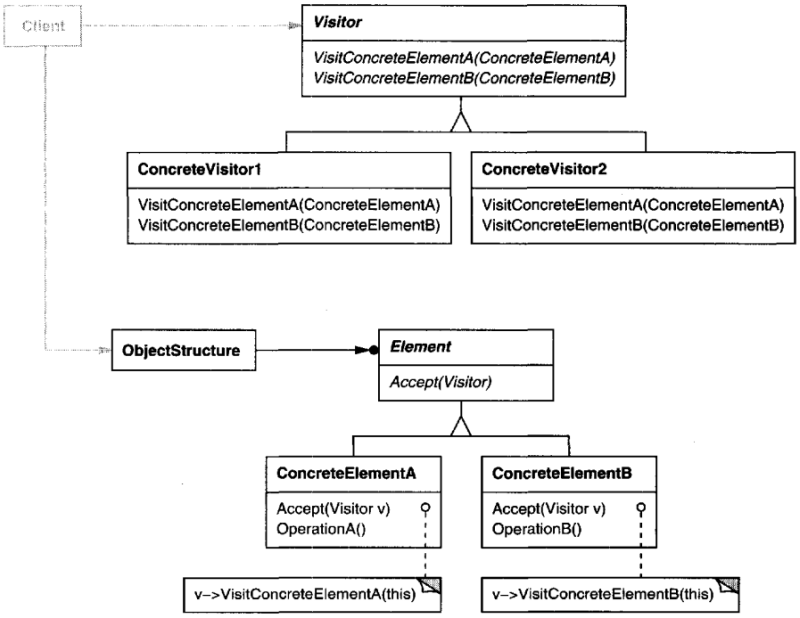
Así, para agregar una nueva operación, sólo se define un nuevo visitor, sin modificar las clases de los nodos, siempre que la estructura del AST no cambie. Esto facilita extender y mantener el compilador.

8.3 Applicability

Usar Visitor cuando:

- La estructura de objetos contiene muchas clases con interfaces diferentes y se quieren realizar operaciones que dependen de la clase concreta de cada objeto.
- Se necesitan muchas operaciones distintas y no relacionadas sobre los objetos, y se quiere evitar mezclar esas operaciones dentro de las clases de los objetos (evitar "contaminarlas"). Visitor agrupa operaciones relacionadas en una sola clase.
- Las clases de la estructura rara vez cambian, pero se suelen agregar nuevas operaciones. Cambiar la estructura implica actualizar la interfaz de todos los visitors, lo que es costoso. Si la estructura cambia frecuentemente, es mejor definir las operaciones directamente en las clases.

8.4 Structure



8.5 Participants

- Visitor (NodeVisitor) : Declara una operación Visit para cada clase concreta de elementos en la estructura. El nombre y firma de la operación identifican la clase concreta que envía la solicitud, permitiendo al visitor acceder directamente a la interfaz específica del elemento.
- Concrete Visitor (TypeCheckingVisitor) : Implementa las operaciones declaradas en Visitor. Cada operación contiene la lógica específica para el tipo de elemento visitado, manteniendo el contexto y estado local (por ejemplo, resultados acumulados durante el recorrido).
- Element (Node) : Define un método Accept que recibe un visitor como argumento.
- Concrete Element (AssignmentNode, VariableRefNode) : Implementan el método Accept que llama al visitor correspondiente.
- Object Structure (Program) : Puede enumerar sus elementos y proveer una interfaz para que el visitor los recorra. Puede ser una estructura compuesta (composite) o una colección (lista, conjunto, etc.).

8.6 Example

Contexto del problema

Tenemos una jerarquía de clases de equipamiento (Equipment) que pueden ser simples (disco floppy, tarjeta) o compuestos (chasis, bus) que contienen otros equipos. Queremos definir operaciones como calcular el costo total o hacer inventario, pero sin mezclar esas operaciones dentro de las clases de equipo.

Aquí entra Visitor: separa las operaciones en clases visitantes (EquipmentVisitor) y permite extender operaciones sin cambiar la jerarquía de equipos.

Clase Base (Equipment)

```
class Equipment {
public:
    virtual ~Equipment();
    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    // M todo clave para Visitor: acepta un visitante
    virtual void Accept(EquipmentVisitor& visitor);

protected:
    Equipment(const char* name) : _name(name) {}

private:
    const char* _name;
};
```

Clase Visitor (EquipmentVisitor)

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();

    // M todos para cada tipo concreto de Equipment
    virtual void VisitFloppyDisk(FloppyDisk* e) { }
    virtual void VisitCard(Card* e) { }
    virtual void VisitChassis(Chassis* e) { }
    virtual void VisitBus(Bus* e) { }

protected:
    EquipmentVisitor() = default;
};
```

Implementación de Accept en Equipos concretos

```
void FloppyDisk::Accept(EquipmentVisitor& visitor) {
    visitor.VisitFloppyDisk(this);
}
```

```
void Chassis::Accept(EquipmentVisitor& visitor) {
    for (auto i = _parts.begin(); i != _parts.end(); ++i) {
        (*i)->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
```

Visitor Concreto 1: PricingVisitor (Cálculo de costos)

```
class PricingVisitor : public EquipmentVisitor {
public:
    PricingVisitor() : _total(0) {}

    Currency& GetTotalPrice() { return _total; }
```

```
    void VisitFloppyDisk(FloppyDisk* e) override {
        _total += e->NetPrice();
    }

    void VisitChassis(Chassis* e) override {
        _total += e->DiscountPrice();
    }

    // Otros visitadores...

private:
    Currency _total;
};
```

Visitor Concreto 2: InventoryVisitor (Inventario)

```
class InventoryVisitor : public EquipmentVisitor {
public:
    InventoryVisitor() {}

    Inventory GetInventory() { return _inventory; }

    void VisitFloppyDisk(FloppyDisk* e) override {
        _inventory.Accumulate(e);
    }

    void VisitChassis(Chassis* e) override {
        _inventory.Accumulate(e);
    }

    // Otros visitadores...

private:
    Inventory _inventory;
};
```

Resultado

Uso del patrón

```
Equipment* equipment = /* alg n equipo o estructura compuesta */;
PricingVisitor pricingVisitor;

equipment->Accept(pricingVisitor);

std::cout << "Costo total del equipo " << equipment->Name()
          << " es " << pricingVisitor.GetTotalPrice() << std::endl;

InventoryVisitor inventoryVisitor;
equipment->Accept(inventoryVisitor);

std::cout << "Inventario de " << equipment->Name() << ": "
          << inventoryVisitor.GetInventory().Summary() << std::endl;
```