

Polimorfismo dinámico versus polimorfismo estático

Flexibilidad versus rendimiento

J. Daniel Garcia

Grupo ARCOS
Universidad Carlos III de Madrid

24 de noviembre de 2016

Aviso

- © Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivar 4.0 Internacional.
- ⓘ Debes dar crédito en la obra en la forma especificada por el autor o licenciante.
- Ⓝ El licenciante permite copiar, distribuir y comunicar públicamente la obra. A cambio, esta obra no puede ser utilizada con fines comerciales — a menos que se obtenga el permiso expreso del licenciante.
- Ⓞ El licenciante permite copiar, distribuir, transmitir y comunicar públicamente solamente copias inalteradas de la obra – no obras derivadas basadas en ella.

ARCOS@uc3m

- **UC3M:** Una universidad joven, internacional y orientada a la investigación.
- **ARCOS:** Un grupo de investigación aplicada.
 - **Líneas:** Computación de altas prestaciones, Big data, Sistemas Ciberfísicos, y **Modelos de programación para la mejora de las aplicaciones**
- **Mejorando las aplicaciones:**
 - **REPARA:** Reengineering and Enabling Performance and powerR of Applications. Financiado por Comisión Europea (FP7).
 - **RePhrase:** REactoring Parallel Heterogeneous Resource Aware Applications. Financiado por Comisión Europea (H2020).
- **Normalización:**
 - ISO/IEC JTC/SC22/WG21. Comité ISO C++.

Agradecimientos

- Algunas de las ideas de esta charla están inspiradas en la charla *Inheritances is base class of evil* de Sean Parent.
- Muchas gracias a las siguientes personas por sus comentarios:
 - Ion Gaztañaga.
 - Manu Sánchez.
 - Joaquín M. López.
- El código completo de esta charla se puede encontrar en:
 - <https://github.com/jdgarciauc3m/polyexamples>.



- 1 Introducción
- 2 Observaciones
- 3 Problema
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

Herramientas y problemas

- Un martillo es fantástico para clavar un clavo.

Herramientas y problemas

- Un martillo es fantástico para clavar un clavo.
 - ¿Y para apretar tuercas?

Herramientas y problemas

- Un martillo es fantástico para clavar un clavo.
 - ¿Y para apretar tuercas?
 - ¿Y para atornillar?

Herramientas y problemas

- Un martillo es fantástico para clavar un clavo.
 - ¿Y para apretar tuercas?
 - ¿Y para atornillar?
 - ¿Para depurar un programa?

Herramientas y problemas

- Un martillo es fantástico para clavar un clavo.
 - ¿Y para apretar tuercas?
 - ¿Y para atornillar?
 - ¿Para depurar un programa?

- ¿Será la OO un martillo para programadores?

Los orígenes

- La subrutina:
 - David Wheeler y Maurice Wilkes. 1951.

Los orígenes

- La subrutina:
 - David Wheeler y Maurice Wilkes. 1951.
- Programación funcional:
 - Lisp, Johh McCarthy. 1958.

Los orígenes

- La subrutina:
 - David Wheeler y Maurice Wilkes. 1951.
- Programación funcional:
 - Lisp, Johh McCarthy. 1958.
- Programación orientada a objetos:
 - Simula, Simula-67. Ole-Johan Dahl y Kristen Nygaard. 1960-1970.
 - Smalltalk, Smalltalk-80. Alan C. Kay. 1970-1980.
 - ...

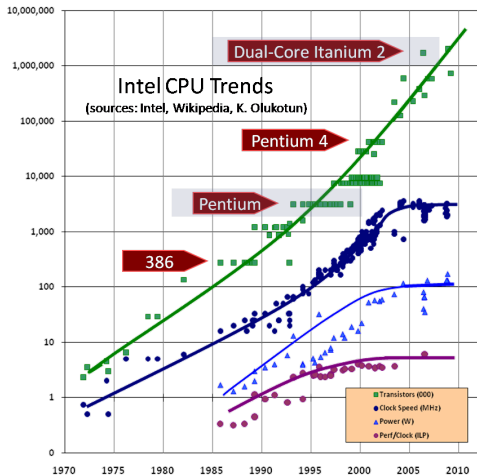
Los orígenes

- La subrutina:
 - David Wheeler y Maurice Wilkes. 1951.
- Programación funcional:
 - Lisp, Johh McCarthy. 1958.
- Programación orientada a objetos:
 - Simula, Simula-67. Ole-Johan Dahl y Kristen Nygaard. 1960-1970.
 - Smalltalk, Smalltalk-80. Alan C. Kay. 1970-1980.
 - ...
- Programación genérica:
 - CLU. Barbara Liskov. 1974.
 - Ada. Jean Ichbiah. 1983.
 - C++. Bjarne Stroustrup. 1983. Alexander Stepanov. 1998.

Pero ¿Cuándo tiene éxito?

- 1979-1983: C++.
- 1986: Object Pascal.
- 1986: Eiffel.
- 1995: Java.
- 1995: Ada95.
- 2000: C#.

¿Qué más ocurrió en los 90?





- 1 Introducción
- 2 Observaciones
- 3 Problema
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

OO: Lenguajes vs Sistema

- ¿Se puede hacer desarrollo OO sin un lenguaje OO?

OO: Lenguajes vs Sistema

- ¿Se puede hacer desarrollo OO sin un lenguaje OO?
 - En teoría: SI.

OO: Lenguajes vs Sistema

- ¿Se puede hacer desarrollo OO sin un lenguaje OO?
 - En teoría: Sí.
 - Y en la práctica también:
 - X-Windows System, OSF/Motif.

OO: Lenguajes vs Sistema

- ¿Se puede hacer desarrollo OO sin un lenguaje OO?
 - En teoría: Sí.
 - Y en la práctica también:
 - X-Windows System, OSF/Motif.
- Sin embargo: demasiado complicado y conducente a errores.
 - Fácil cometer errores.
 - ¿Por qué dejar al programador tareas que puede hacer un compilador?

Cosas que a veces olvidamos

- Principios de diseño de X-Windows (Bob Scheifler y Jim Gettys, 1984).
 - **Do not add new functionality** unless an implementor cannot complete a real application without it.
 - It is as important to decide what a system is not as to decide what it is. **Do not serve all the world's needs.**
 - The only thing worse than **generalizing** from one example is generalizing from no examples at all.
 - If a problem is not completely understood, it is probably best to provide no solution at all.
 - If you can get 90 percent of the desired effect for 10 percent of the work, use the simpler solution.
 - **Isolate complexity** as much as possible.
 - Provide mechanism rather than policy.

Todo tiene que ser un método

- ¿Por qué tenemos clases?

Todo tiene que ser un método

- ¿Por qué tenemos clases?
 - Principio de la ocultación de la información. David Parnas (1972)
 - Incrementar la flexibilidad y comprensión.
 - Reducir tiempo de desarrollo.
 - Reducir acoplamiento global.

Todo tiene que ser un método

- ¿Por qué tenemos clases?
 - Principio de la ocultación de la información. David Parnas (1972)
 - Incrementar la flexibilidad y comprensión.
 - Reducir tiempo de desarrollo.
 - Reducir acoplamiento global.
- Pero ...
 - Hemos asumido que el módulo o componente es equivalente a la clase.

Todo tiene que ser un método

- ¿Por qué tenemos clases?
 - Principio de la ocultación de la información. David Parnas (1972)
 - Incrementar la flexibilidad y comprensión.
 - Reducir tiempo de desarrollo.
 - Reducir acoplamiento global.
- Pero ...
 - Hemos asumido que el módulo o componente es equivalente a la clase.
 - Y nos empeñamos en ocultar, incluso cuando no hay nada que ocultar.

Métodos de clase

- Un método de la *metaclass* → Compartido por todas la instancias.

Métodos de clase

- Un método de la *metaclass* → Compartido por todas la instancias.

Java

```
public final class Math {  
    public static double abs(double a) { /*...*/ }  
}
```

```
// Invocación  
x = Math.abs(y);
```

Métodos de clase

- Un método de la *metaclass* → Compartido por todas la instancias.

Java

```
public final class Math {  
    public static double abs(double a) { /*...*/ }  
}
```

// Invocación

```
x = Math.abs(y);
```

C++

```
namespace std {  
    double abs(double arg);  
}
```

// Invocación

```
x = abs(y);
```

Métodos de clase

- Un método de la *metaclass* → Compartido por todas la instancias.

Java

```
public final class Math {
    public static double abs(double a) {/*...*/}
}
```

// Invocación

```
x = Math.abs(y);
```

C++

```
namespace std {
    double abs(double arg);
}
```

// Invocación

```
x = abs(y);
```

- Tendemos a *inventar* clases:
 - Porque en alguna clase tengo que meter esto.

Métodos de clase

- Un método de la *metaclass* → Compartido por todas la instancias.

Java

```
public final class Math {
    public static double abs(double a) { /*...*/ }
}
```

// Invocación

```
x = Math.abs(y);
```

C++

```
namespace std {
    double abs(double arg);
}
```

// Invocación

```
x = abs(y);
```

- Tendemos a *inventar* clases:
 - Porque en alguna clase tengo que meter esto.
- O a tener *animales* como la clase **Integer**.

Métodos de clase

- Un método de la *metaclass* → Compartido por todas la instancias.

Java

```
public final class Math {
    public static double abs(double a) { /*...*/ }
}
```

```
// Invocación
x = Math.abs(y);
```

C++

```
namespace std {
    double abs(double arg);
}
```

```
// Invocación
```

```
x = abs(y);
```

- Tendemos a *inventar* clases:
 - Porque en alguna clase tengo que meter esto.
- O a tener *animales* como la clase **Integer**.
- O a hacer que **main** sea un *método*.

Ocultación paranóica de la información

- Los atributos de una clase siempre tiene que ser privados.

Ocultación paranóica de la información

- Los atributos de una clase siempre tiene que ser privados.
 - Y luego lo llenamos todo de *getter* y *setters*.

Ocultación paranóica de la información

- Los atributos de una clase siempre tiene que ser privados.
 - Y luego lo llenamos todo de *getter* y *setters*.
 - ¿Dónde está la ocultación de la información?

Ocultación paranóica de la información

- Los atributos de una clase siempre tiene que ser privados.
 - Y luego lo llenamos todo de *getter* y *setters*.
 - ¿Dónde está la ocultación de la información?

Un par de valores

```
class pair {  
public:  
    pair(int k, const string & v);  
    void set_key(int k) { key=k; }  
    int get_key() const { return key; }  
    void set_value(const string & v) { value=v; }  
    string get_value() const { return value; }  
private:  
    int key;  
    string value;  
};
```

Ocultación paranóica de la información

- Los atributos de una clase siempre tiene que ser privados.
 - Y luego lo llenamos todo de *getter* y *setters*.
 - ¿Dónde está la ocultación de la información?

Un par de valores

```
class pair {  
public:  
    pair(int k, const string & v);  
    void set_key(int k) { key=k; }  
    int get_key() const { return key; }  
    void set_value(const string & v) { value=v; }  
    string get_value() const { return value; }  
private:  
    int key;  
    string value;  
};
```

O un par de valores

```
struct pair {  
    pair(int k, const string & v);  
    int key;  
    string value;  
};
```

Herencia

- **Herencia**: Mecanismo de reutilización de código.
 - **Herencia de interfaz** (Subtyping): Establece una relación *es-un*.
 - **Herencia de implementación**: Mecanismo de reutilización de implementación.

Herencia

- **Herencia**: Mecanismo de reutilización de código.
 - **Herencia de interfaz** (Subtyping): Establece una relación *es-un*.
 - **Herencia de implementación**: Mecanismo de reutilización de implementación.
- Aproximaciones:
 - Todo deben ser clases organizadas en una única jerarquía universal.
 - Un **Vector** extiende un **AbstractList** que extiende un **AbstractCollection** que extiende un **Object**.
 - Utilizar herencia de forma limitada cuando sea necesario.
 - Un **std::vector** no hereda nada de nadie.

Polimorfismo

- **Polimorfismo**: Capacidad de ofrecer el mismo interfaz variando la implementación.

Polimorfismo

- **Polimorfismo**: Capacidad de ofrecer el mismo interfaz variando la implementación.
- Normalmente usado como sinónimo de *polimorfismo dinámico*.

Polimorfismo

- **Polimorfismo**: Capacidad de ofrecer el mismo interfaz variando la implementación.
- Normalmente usado como sinónimo de *polimorfismo dinámico*.
- **Polimorfismo dinámico**: Se toma la decisión sobre la selección de la implementación en tiempo de ejecución.
 - Pequeño sobrecoste por invocación a función.
 - Aceptable si el uso es limitado.
 - Conduce a diseños más flexible de lo realmente necesario.
 - Y si un día ...

Hay otro polimorfismo

- **Polimorfismo estático**: Se toma la decisión sobre la selección en tiempo de compilación.
 - No siempre es posible.
 - Flexibilidad suficiente.
 - No hay sobrecoste en tiempo de ejecución.

Ordenación genérica

```
template <typename C>  
void sort(C & c) {  
    sort(c.begin(), c.end());  
}
```

Un problema adicional

- Gestión de los recursos.
 - Muchos lenguajes OO han optado por *hacerle la vida fácil* al programador con la recolección de basura.

Un problema adicional

- Gestión de los recursos.
 - Muchos lenguajes OO han optado por *hacerle la vida fácil* al programador con la recolección de basura.

- Pero seguimos con *memory leaks* por todas partes.

Un problema adicional

- Gestión de los recursos.
 - Muchos lenguajes OO han optado por *hacerle la vida fácil* al programador con la recolección de basura.
- Pero seguimos con *memory leaks* por todas partes.
- La recolección de basura:
 - Generalmente tiene impacto en el rendimiento.
 - Incremento del impacto debido a la *revolución* multi-core.
 - La mejor estrategia de gestión de la basura es no generar basura.

- 1 Introducción
- 2 Observaciones
- 3 Problema**
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

Gráficos vectoriales

- Se quiere representar una escena gráfica como un conjunto de figuras geométricas (rectángulos, círculos, ...).
 - Todos los objetos tienen una posición (x,y).
 - Por simplicidad usaremos coordenadas enteras.

- Operaciones sobre una escena:
 - Cargar en archivo.
 - Guardar en archivo.
 - Calcular la suma de todas las áreas.
 - Trasaldar todos los objetos.
 - Agrandar los objetos.

Utilidades a considerar

- Generar un archivo con figuras aleatorias.
- Determinar la suma de el área de todas las figuras de un archivo.
- Trasladar todas las figuras de un archivo.
- Agrandar todos los objetos de un archivo.



- 1 Introducción
- 2 Observaciones
- 3 Problema
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

Solución mínima

- Se considera como única figura posible el **rectángulo**.
 - Posición (**x,y**).
 - Dimensiones (**width,height**).

- Código simplificado:
 - Una única clase (**rectangle**).
 - La escena contiene un **vector<rectangle>**.

Formato de fichero

```
scene
rectangle: 37 29 7 10
rectangle: 73 100 4 5
rectangle: 31 75 9 2
rectangle: 84 28 7 3
rectangle: 5 63 2 6
end-scene
```

Un rectángulo

rectangle.h

```
class rectangle {  
public:  
    rectangle() noexcept = default;  
  
    rectangle(int x, int y, int w, int h) noexcept :  
        x_{x}, y_{y}, width_{w}, height_{h} {}  
  
    int area() const noexcept  
        { return width_ * height_; }  
  
    void translate(int dx, int dy) noexcept  
        { x_ += dx; y_ += dy; }  
  
    void enlarge(int k) noexcept  
        { width_ *= k; height_ *= k; }  
  
    // ...  
};
```

Un rectángulo

rectangle.h

```
// ...
```

```
friend std::ostream & operator<<(std::ostream & os, const rectangle & r);  
friend std::istream & operator>>(std::istream & is, rectangle & r);
```

```
private:
```

```
int x_=0;  
int y_=0;  
int width_=0;  
int height_=0;  
};
```

Entrada/salida de rectángulos

rectangle.cpp

```
std::ostream & operator<<(std::ostream & os, const rectangle & r) {  
    return os << r.x_ << " " << r.y_ << " "  
        << r.width_ << " " << r.height_;  
}  
  
std::istream & operator>>(std::istream & is, rectangle & r) {  
    return is >> r.x_ >> r.y_  
        >> r.width_ >> r.height_;  
}
```

Una escena

scene.h

```
class rectangle;  
  
class scene {  
public:  
    void add_shape(const rectangle & r);  
    int size() const noexcept { return shapes_.size(); }  
  
    long long area() const noexcept;  
    void translate(int dx, int dy) noexcept;  
    void enlarge(int k) noexcept;  
  
    friend std::ostream & operator<<(std::ostream & os, const scene & s);  
    friend std::istream & operator>>(std::istream & is, scene & s);  
  
private:  
    std::vector<rectangle> shapes_;  
};
```


Implementando una escena

scene.cpp

```
void scene::add_shape(const rectangle & r) {
    shapes_.push_back(r);
}

long long scene::area() const noexcept {
    // transform_reduce(begin(shapes_), end(shapes_),
    //   [](auto x) { return x.area(); },
    //   [](auto x, auto y) { return x+y; });
    long long r = 0;
    for (auto && s : shapes_) {
        r += s.area();
    }
    return r;
}
```

Implementando una escena

scene.cpp

```
void scene::translate(int dx, int dy) noexcept {  
    for (auto && s : shapes_) {  
        s.translate(dx,dy);  
    }  
}  
  
void scene::enlarge(int k) noexcept {  
    for (auto && s : shapes_) {  
        s.enlarge(k);  
    }  
}
```

Volcando una escena

scene.cpp

```
std::ostream & operator<<(std::ostream & os, const scene & s) {  
    os << "scene\n";  
    for (auto && s : s.shapes_) {  
        os << "rectangle: " << s << std::endl;  
    }  
    os << "end-scene";  
    return os;  
}
```

Leyendo una escena

scene.cpp

```
std::istream & operator>>(std::istream & is, scene & s) {  
    using namespace std;  
    string w;  
    is >> w;  
    if (w!="scene") return is;  
    while (is >> w) {  
        if (w=="rectangle:") {  
            rectangle r;  
            is >> r;  
            s.add_shape(r);  
        }  
        else if (w=="end-scene") {  
            return is;  
        }  
        else {  
            is.setstate(ios_base::failbit );  
            return is;  
        }  
    }  
    return is;  
}
```

Generando escenas

genscene.cpp

```
void generate_scene(const std::string & name, int n) {
    using namespace std;
    using namespace dsl;

    cout << "Writing file " << name << endl;
    cout << "sizeof(rectangle)= " << sizeof(dsl::rectangle) << endl;
    cout << "Generating " << n << " elements\n";

    random_device rd;
    scene s;
    for (int i=0; i<n; i++) {
        auto r = random_rectangle(rd);
        s.add_shape(r);
    }

    ofstream of{name};
    of << s << endl;
    if (!of) throw runtime_error{"Cannot write to file : " + name};

    cout << "Generated file " << name << " with " << n << " elements\n";
}
```

Calculando areas

area.cpp

```
void print_area(const std::string & inname) {  
    using namespace std;  
    using namespace dsl;  
  
    scene s;  
    ifstream in{inname};  
    in >> s;  
    if (!in) throw runtime_error{"Error reading scene file "};  
  
    cout << s.area() << endl;  
}
```

Trasladando figuras

translate.cpp

```
void translate_scene(const std::string & inname, const std::string & outname,
                    int dx, int dy) {
    using namespace std;
    using namespace dsl;

    cout << "Reading gfile " << inname << endl;

    scene s;
    ifstream in{inname};
    in >> s;
    if (!in) throw runtime_error{"Error reading scene file : " + inname};

    s.translate(dx,dy);

    ofstream out{outname};
    out << s;
    if (!out) throw runtime_error{"Error writing scene file " + outname};

    cout << s.size() << " shapes written to file " << outname << endl;
}
```

- 1 Introducción
- 2 Observaciones
- 3 Problema
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

Diferentes figuras

- Múltiples tipos de figuras.
 - Restringiremos el ejemplo al **rectángulo** y el círculo.
 - Después de todo es un ejemplo de libro :-)
- Solución clásica:
 - Una clase base: **shape**.
 - Una clase derivada por tipo de figura: **rectangle**, **circle**.
 - La escena mantiene una lista de figuras.
 - No se pueden almacenar objetos de distinto tipo en un contenedor.
 - Almacenaremos punteros a **shape**.
 - Usamos **std::shared_ptr** para simplificar la gestión de la memoria.

Diseñando la clase base

- Construcción y destrucción:
 - El destructor tiene que ser virtual.
 - Hace falta definir explícitamente el constructor vacío.

shape.h

```
class shape {  
public:  
    shape() noexcept = default;  
    virtual ~shape() noexcept = default;  
  
    shape(int x, int y) noexcept :  
        x_{x}, y_{y} {}  
  
private:  
    int x_=0;  
    int y_=0;  
};
```

Diseñando la clase base

- La posición se mantiene en la figura.
 - La traslación se gestiona aquí.
 - Funciones virtuales puras para funciones de clases derivadas.

shape.h

```
// ...  
  
void translate(int dx, int dy) noexcept  
{ x_ += dx; y_ += dy; }  
  
virtual int area() const noexcept = 0;  
virtual void enlarge(int k) noexcept = 0;  
  
// ...
```

Entrada/salida

- Virtualización de operadores de inserción y extracción.
 - Un operador acompañado de una función virtual pura.
 - Una función para obtener la etiqueta del tipo de objeto.
 - **rectangle**, **circle**.

shape.h

```
// ...
```

```
virtual std::string tagname() const = 0;
```

```
friend std::ostream & operator<<(std::ostream & os, const shape & r);
```

```
virtual std::ostream & insert(std::ostream & os) const;
```

```
friend std::istream & operator>>(std::istream & is, shape & r);
```

```
virtual std::istream & extract(std::istream & is);
```

```
// ...
```



Entrada/salida

shape.cpp

```
std::ostream & shape::insert(std::ostream & os) const {
    return os << x_ << ' ' << y_ << ' ';
}

std::istream & shape::extract(std::istream & is) {
    return is >> x_ >> y_;
}

std::ostream & operator<<(std::ostream & os, const shape & r) {
    return r.insert(os);
}

std::istream & operator>>(std::istream & is, shape & r) {
    return r.extract(is);
}
```

Derivando el rectángulo

rectangle.h

```
class rectangle final : public shape {  
public:  
    rectangle() noexcept = default;  
  
    rectangle(int x, int y, int w, int h) :  
        shape{x,y}, width_{w}, height_{h} {}  
  
    int area() const noexcept override  
        { return width_ * height_; }  
  
    void enlarge(int k) noexcept override  
        { width_ *= k; height_ *= k; }  
  
private:  
    int width_=0;  
    int height_=0;  
};
```

Derivando el rectángulo

rectangle.h

```
std::string tagname() const override  
{ return "rectangle"; }
```

```
friend std::ostream & operator<<(std::ostream & os, const rectangle & r);  
std::ostream & insert(std::ostream & os) const override;
```

```
friend std::istream & operator>>(std::istream & is, rectangle & r);  
std::istream & extract(std::istream & is) override;
```

Entrada/salida del rectángulo

rectangle.cpp

```
std::ostream & operator<<(std::ostream & os, const rectangle & r) {  
    return r.insert(os);  
}
```

```
std::ostream & rectangle::insert(std::ostream & os) const {  
    shape::insert(os);  
    return os << width_ << " " << height_;  
}
```

```
std::istream & operator>>(std::istream & is, rectangle & r) {  
    return r.extract(is);  
}
```

```
std::istream & rectangle::extract(std::istream & is) {  
    shape::extract(is);  
    return is >> width_ >> height_;  
}
```


Diseñando la escena

- Almacenamiento polimórfico de las figuras.
 - Mantenimiento de las figuras en un vector de punteros a la clase base.
 - Usamos **shared_ptr** para gestionar la memoria.

- Interfaz de adición de figuras a la escena.
 - La función **add_shape()** toma ahora un argumento **shared_ptr<shape>**.
 - El argumento se toma como *referencia a r-valor*.
 - Permite aprovechar semántica de movimiento.

Una escena con figuras polimórficas

scene.h

```
class shape;

class scene {
public:
    void add_shape(std::shared_ptr<shape> && s) { shapes_.push_back(s); }
    int size() const noexcept { return shapes_.size(); }

    long long area() const noexcept;
    void translate(int dx, int dy) noexcept;
    void enlarge(int k) noexcept;

    friend std::ostream & operator<<(std::ostream & os, const scene & s);
    friend std::istream & operator>>(std::istream & is, scene & s);

private:
    std::vector<std::shared_ptr<shape>> shapes_;
};
```

Implementando las escenas

scene.cpp

```
long long scene::area() const noexcept {  
    long long r = 0;  
    for (auto && s : shapes_) {  
        r += s->area();  
    }  
    return r;  
}
```

Implementando las escenas

scene.cpp

```
void scene::translate(int dx, int dy) noexcept {
    for (auto && s : shapes_) {
        s->translate(dx,dy);
    }
}

void scene::enlarge(int k) noexcept {
    for (auto && s : shapes_) {
        s->enlarge(k);
    }
}
```

Escribiendo las escenas

scene.cpp

```
std::ostream & operator<<(std::ostream & os, const scene & s) {  
    os << "scene\n";  
    for (auto && s : s.shapes_) {  
        os << s->tagname() << ": ";  
        s->insert(os); // Polymorphic write  
        os << std::endl;  
    }  
    os << "end-scene";  
    return os;  
}
```

Una factoría de figuras

scene.cpp

```
namespace { // Anonymous namespace

std::shared_ptr<shape> make_shape(const std::string & cname) {
    using namespace std;
    shared_ptr<shape> p = nullptr;
    if (cname=="rectangle:") p = make_shared<rectangle>();
    else if (cname=="circle:") p = make_shared<circle>();
    return p;
}

}
```

Leyendo figuras

scene.cpp

```
std::istream & operator>>(std::istream & is, scene & s) {  
    using namespace std;  
    string w;  
    is >> w;  
    if (w!="scene") return is;  
    while (is >> w) {  
        auto sh = make_shape(w);  
        if (sh) {  
            is >> *sh;  
            s.add_shape(std::move(sh));  
        }  
        else if (w=="end—scene") {  
            return is;  
        }  
        else {  
            is.setstate(ios_base::failbit );  
            return is;  
        }  
    }  
    return is;  
}
```

Usando las escenas

- Cambios en la interfaz:
 - La función `add_shape()` ha cambiado su interfaz.
- Efectos sobre el código:
 - Se traslada complejidad al desarrollador de las figuras.
 - Se crean objetos en memoria dinámica que hay que transferir.
- Efectos sobre el rendimiento:
 - Mayor uso de la memoria dinámica.
 - Una reserva por figura.
 - Pero comportamiento de memoria caché.
 - Las figuras dejan de estar contiguas en memoria.

- 1 Introducción
- 2 Observaciones
- 3 Problema
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

Polimorfismo dinámico versus estático

- Polimorfismo dinámico:
 - Aporta mayor flexibilidad.
 - Require incorporar nuevas clases en una jerarquía.
 - Tiene un mayor coste en tiempo de ejecución.

- Polimorfismo estático:
 - Menor flexibilidad (aunque muchas veces suficiente).
 - Los tipos usados pueden ser independientes.
 - Menor coste en tiempo de ejecución.

- **La virtud está en el justo medio.**

Ocultando el polimorfismo

- Ocultar el polimorfismo detrás de una única clase.
 - Permite usar clases independientes.
 - Oculta la herencia como mecanismo de implementación.
- Diseño de figura:
 - Un constructor genérico que acepta valores de cualquier tipo.
 - Un función de factoría para valores vacíos.
 - La clase es copiable pero no movible.
 - Una jerarquía de clases como detalle privado de implementación.
 - Un puntero a la base de la jerarquía que puede contener un objeto derivado.

Un constructor genérico

No es lo mismo una clase genérica...

```
template <typename T>
class A {
public:
    A(T x);
    // ...
};
```

- Cada instancia da lugar a una clase distinta.
- **A<int>** no tiene el mismo tipo que **A<long>**.

Un constructor genérico

No es lo mismo una clase genérica...

```

template <typename T>
class A {
public:
    A(T x);
    // ...
};

```

- Cada instancia da lugar a una clase distinta.
- **A<int>** no tiene el mismo tipo que **A<long>**.

...Que un constructor genérico

```

class A {
public:
    template <typename T>
    A(T x);
    // ...
};

```

- Todas las instancias dan lugar al mismo tipo.
- **A<int>** y **A<long>** tienen ambos el tipo **A**.

Una jerarquía privada

Jerarquía para shape

```

class shape {
public:
    template <typename T> shape(T x);

private:
    class shape_base { /*...*/ };

    template <typename T>
    concrete_shape : public shape_base {
        // ...
    private:
        T impl_;
    };

    std::unique_ptr<shape_base> self_;
};

```

- El puntero **self_** puede contener un objeto de cualquier clase derivada.
- La clase **shape_base** puede tener un número de funciones virtuales puras.
- Se pueden redefinir las funciones virtuales en la clase **concrete_shape**.
- La clase **concrete_shape** puede delegar la implementación en el objeto **impl_**.

Una función de factoría polimórfica

Función de factoría

```
class shape {  
    // ...  
    template <typename T>  
    friend shape make_shape();  
    // ...  
};  
  
template <typename T>  
shape make_shape() {  
    shape s;  
    s.self_ = make_unique<  
        shape::concrete_shape<T>>();  
    return s;  
}
```

- No es posible tener un constructor genérico sin argumentos.
- Se hace la función de factoría **friend** para que pueda acceder a la implementación.

Una figura con semántica de valor

shape.h

```
class shape {  
public:  
  
    shape() : self_{nullptr} {}  
  
    template<typename T>  
    shape(T x);  
  
    shape(const shape &) = delete;  
    shape & operator=(const shape &) = delete;  
  
    shape(shape &&) noexcept = default;  
    shape & operator=(shape &&) = default;
```


Una figura con semántica de valor

shape.h

```
std::string tagname() const { return self_ ->tagname(); }
int area() const { return self_ ->area(); }
void translate(int dx, int dy) { self_ ->translate(dx,dy); }
void enlarge(int k) { self_ ->enlarge(k); }

friend std::ostream & operator<<(std::ostream & os, const shape & s)
{ s.self_ ->insert(os); return os; }
friend std::istream & operator>>(std::istream & is, const shape & s)
{ s.self_ ->extract(is); return is; }
```

Una figura con semántica de valor

shape.h

private:

```
class shape_base {  
public:  
    shape_base() {}  
    virtual ~shape_base() = default;  
    virtual std::string tagname() const = 0;  
    virtual int area() const = 0;  
    virtual void translate(int dx, int dy) = 0;  
    virtual void enlarge(int k) = 0;  
    virtual void insert(std::ostream & os) const = 0;  
    virtual void extract(std::istream & is) = 0;  
};
```

Una figura con semántica de valor

shape.h

```

template <typename T>
class concrete_shape final : public shape_base {
public:
    concrete_shape() : impl_{ } {}
    concrete_shape(T && x) : impl_{std::forward<T>(x)} {}
    virtual ~concrete_shape() = default;
    std::string tagname() const override { return impl_.tagname(); }
    int area() const override { return impl_.area(); }
    void translate(int dx, int dy) override { impl_.translate(dx,dy); }
    void enlarge(int k) override {impl_.enlarge(k); }
    void insert(std::ostream & os) const override { os << impl_; }
    void extract(std::istream & is) override { is >> impl_; }
private:
    T impl_;
};

```

Una figura con semántica de valor

shape.h

```
std::unique_ptr<shape_base> self_ ;

template <typename U> friend shape make_shape();
};

template <typename T>
shape::shape(T x) :
    self_{std::make_unique<concrete_shape<T>>(std::forward<T>(x))}
{}

template<typename T>
shape make_shape() {
    shape s;
    s.self_ = std::make_unique<shape::concrete_shape<T>>();
    return s;
}
```

¿Qué pasa con las figuras concretas?

- Simplificación.
 - Pasan a ser clases sin dependencias.
 - No heredan de ninguna clase.
 - Todas sus funciones miembro pasan a ser no virtuales.

¿Qué pasa con la escena?

scene.h

```
class scene {  
public:  
    void add_shape(shape && s) { shapes_.push_back(std::forward<shape>(s)); }  
    int size() const { return shapes_.size(); }  
  
    long long area() const;  
    void translate(int dx, int dy);  
    void enlarge(int k);  
  
    friend std::ostream & operator<<(std::ostream & os, const scene & s);  
    friend std::istream & operator>>(std::istream & is, scene & s);  
  
private:  
    std::vector<shape> shapes_;  
  
};
```

¿Y qué pasa con el código cliente?

- Vuelve a usar semántica de valor.

Código cliente

```
shape s = make_shape<rectangle>();  
myscene.add_shape(move(s));  
// ...
```

- 1 Introducción
- 2 Observaciones
- 3 Problema
- 4 Solución simple
- 5 Solución heterogénea
- 6 Combinando polimorfismos
- 7 Optimizando los objetos pequeños

Disposición de memoria y rendimiento

- La solución con borrado de tipos simplifica la extensión pero tiene los mismos problemas de rendimiento que la versión orientada a objetos.
 - Cada objeto se asigna en memoria dinámica independientemente.
 - Multitud de reservas de memoria individuales.
 - Los objetos no se almacenan de forma contigua en memoria.
 - Baja tasa de aciertos en caché de procesador.

Optimización del objeto pequeño

- Usar un búfer de tamaño fijo para cada figura.
 - Si el objeto es muy pequeño construir sobre el búfer.
 - Si el objeto es grande construir sobre el búfer un puntero a un objeto en memoria dinámica.
- Se puede fijar el tamaño de búfer a **32** bytes.
 - Tamaño suficiente para objetos muy pequeños más un puntero **vptr**.
 - Un tamaño de **16** solamente permitiría objetos extremadamente pequeños.
 - Un tamaño de **64** reduciría excesivamente la tasa de aciertos de caché.
 - Valor con cierta dependencia de la aplicación.

Una figura optimizada

shape.h

```
class shape {  
private:  
    class shape_base;  
    constexpr static int max_shape_size = 32;  
  
    using internal_buffer =  
        typename std::aligned_storage<max_shape_size,max_shape_size>::type;
```

Jerarquía de clases

- Clases privadas como detalle de implementación.
 - **shape_base**: Base de la jerarquía.
 - Clase abstracta con funciones virtuales puras.
 - Función virtual pura para construir por movimiento en un búfer con *placement* (**moving_clone()**).
 - **local_shape<S>**: Envoltorio derivado para objetos pequeño.
 - Contiene un miembro con la figura **S**.
 - Delegan las operaciones virtuales a este miembro.
 - **dynamic_shape**: Evoltorio derivado para puntero a objetos grandes.
 - Contiene un miembro con un puntero la figura **S**.
 - Delegan las operaciones virtuales al objeto apuntado.

Jerarquía **privada** de clases

shape.h

```
class shape {  
private:  
    // ...  
    class shape_base { /* ... */ };  
  
    class local_shape final : public shape_base {  
        // ...  
    };  
  
    class dynamic_shape final : public shape_base {  
        // ...  
    };  
    // ...  
}
```

La base de todas las figuras

shape.h

```
class shape_base {  
public:  
    shape_base() noexcept {}  
    virtual ~shape_base() noexcept = default;  
  
    virtual void moving_clone(internal_buffer & buf) noexcept = 0;  
  
    virtual std::string tagname() const = 0;  
    virtual int area() const noexcept = 0;  
    virtual void translate(int dx, int dy) noexcept = 0;  
    virtual void enlarge(int k) noexcept = 0;  
    virtual void insert(std::ostream & os) const noexcept = 0;  
    virtual void extract(std::istream & is) noexcept = 0;  
};
```

Una figura local

shape.h

```

template <typename S>
class local_shape final : public shape_base {
public:
    local_shape() noexcept : impl_{} {}
    local_shape(S && x) noexcept : impl_{std::forward<S>(x)} {}
    virtual ~local_shape() noexcept = default;

    virtual void moving_clone(internal_buffer & buf) noexcept override;

    std::string tagname() const override { return impl_.tagname(); }
    int area() const noexcept override { return impl_.area(); }
    void translate(int dx, int dy) noexcept override { impl_.translate(dx,dy); }
    void enlarge(int k) noexcept override { impl_.enlarge(k); }
    void insert(std::ostream & os) const noexcept override { os << impl_; }
    void extract(std::istream & is) noexcept override { is >> impl_; }
private:
    S impl_;
};

```

Una figura dinámica

shape.h

```

template <typename S>
class dynamic_shape final : public shape_base {
public:
    dynamic_shape() : impl_{std::make_unique<S>()} {}
    dynamic_shape(S && s) : impl_{std::make_unique<S>(std::forward<S>(s))} {}
    dynamic_shape(std::unique_ptr<S> && p) noexcept : impl_{std::forward<std::unique_ptr<S>>(p)} {}
    virtual ~dynamic_shape() noexcept = default;

    virtual void moving_clone(internal_buffer & buf) noexcept override;

    std::string tagname() const noexcept override { return impl_ ->tagname(); }
    int area() const noexcept override { return impl_ ->area(); }
    void translate(int dx, int dy) noexcept override { impl_ ->translate(dx,dy); }
    void enlarge(int k) noexcept override { impl_ ->enlarge(k); }
    void insert(std::ostream & os) const noexcept override { os << *impl_; }
    void extract(std::istream & is) noexcept override { is >> *impl_; }
private:
    std::unique_ptr<S> impl_;
};

```


Moving clones

- Crean el objeto implementación a otro búfer.
- El objeto actual quedará en un estado válido pero destruible.

shape.h

```
template <typename S>
void shape::local_shape<S>::moving_clone(internal_buffer & buf) noexcept {
    new (&buf) local_shape<S>(std::move(impl_));
}

template <typename S>
void shape::dynamic_shape<S>::moving_clone(internal_buffer & buf) noexcept {
    new (&buf) dynamic_shape<S>(std::move(impl_));
}
```

Representación interna

shape.h

private:


```
internal_buffer buffer_;
```

```
shape_base * self() noexcept {  
    return reinterpret_cast<shape_base*>(&buffer_);  
}
```

```
const shape_base * self() const noexcept {  
    return reinterpret_cast<const shape_base*>(&buffer_);  
}
```

```
shape() {}
```

■ **Constructor vacío privado** → no se puede hacer genérico.

■ Más cuando veamos función amiga **make_shape()**. 

Soporte para algo parecido a conceptos

shape.h

```
template <typename T>
static constexpr bool is_small() {
    return sizeof(local_shape<T>) <= max_shape_size;
}

template <typename T>
using small_shape = typename std::enable_if<is_small<T>(), shape>::type;

template <typename T>
using large_shape = typename std::enable_if<!is_small<T>(), shape>::type;
```

- `is_small<T>()` predicado sobre tipo `T` para verificar si es *pequeño*.
- `small_shape<T>` es el tipo `shape` solamente si `T` es *pequeño*.
- `large_shape<T>` es el tipo `shape` solamente si `T` **no** es *pequeño*.

Seleccionando el movimiento de figuras

shape.h

public:

```
template <typename S,  
         small_shape<S> * = nullptr>  
shape(S && s) noexcept {  
    new (&buffer_) local_shape<S>{std::forward<S>(s)};  
}  
  
template <typename S,  
         large_shape<S> * = nullptr>  
shape(S && s) noexcept {  
    new (&buffer_) dynamic_shape<S>{std::forward<S>(s)};  
}
```

- Dependiendo del tamaño de **T** solamente uno de los dos existe.

Simulando un constructor vacío selectivo

shape.h

```
template <typename S>
friend small_shape<S> make_shape() noexcept;

template <typename S>
friend large_shape<S> make_shape() noexcept;
```

- Permite simular un constructor vacío con un argumento de plantilla.
- **make_shape()** es una función de factoría amiga.
 - Necesita acceso al constructor vacío privado.

Implementando las funciones de factoría

shape.h

```
template <typename S>
shape::small_shape<S> make_shape() noexcept {
    shape s;
    new (&s.buffer_) shape::local_shape<S>{};
    return s;
}
```

```
template <typename S>
shape::large_shape<S> make_shape() noexcept {
    shape s;
    new (&s.buffer_) shape::dynamic_shape<S>{};
    return s;
}
```

Copia, movimiento y destrucción

shape.h

```
shape(const shape &) noexcept = delete;  
shape & operator=(const shape &) noexcept = delete;  
  
shape(shape && s) noexcept {  
    s.self () -> moving_clone(buffer_);  
}  
  
shape & operator=(shape &&) noexcept = delete;  
  
~shape() noexcept {  
    self () -> ~shape_base();  
}
```

Funcionalidad delegada

shape.h

```
std::string tagname() const { return self()->tagname(); }  
int area() const noexcept { return self()->area(); }  
void translate(int dx, int dy) noexcept { self()->translate(dx,dy); }  
void enlarge(int k) noexcept { self()->enlarge(k); }  
  
friend std::ostream & operator<<(std::ostream & os, const shape & s)  
{ s.self()->insert(os); return os; }  
friend std::istream & operator>>(std::istream & is, shape & s)  
{ s.self()->extract(is); return is; }
```


Una escena simplificada (otra vez)

scene.h

```
class scene {  
public:  
    void add_shape(shape && s) { shapes_.push_back(std::move(s)); }  
    int size() const noexcept { return shapes_.size(); }  
  
    long long area() const noexcept ;  
    void translate(int dx, int dy) noexcept ;  
    void enlarge(int k) noexcept ;  
  
    friend std::ostream & operator<<(std::ostream & os, const scene & s);  
    friend std::istream & operator>>(std::istream & is, scene & s);  
  
private:  
    std::vector<shape> shapes_ ;  
};
```

Asignaciones de memoria

- **Clásico**: 25 asignaciones de memoria.

Asignaciones de memoria

- **Clásico**: 25 asignaciones de memoria.
- **Orientada a Objetos**: 1,000,025 asignaciones de memoria.

Asignaciones de memoria

- **Clásico**: 25 asignaciones de memoria.
- **Orientada a Objetos**: 1,000,025 asignaciones de memoria.
- **Genérico**: 25 asignaciones de memoria.

Asignaciones de memoria

- **Clásico**: 25 asignaciones de memoria.
- **Orientada a Objetos**: 1,000,025 asignaciones de memoria.
- **Genérico**: 25 asignaciones de memoria.
- **Type erased**: 1,000,028 asignaciones de memoria.

Asignaciones de memoria

- **Clásico**: 25 asignaciones de memoria.
- **Orientada a Objetos**: 1,000,025 asignaciones de memoria.
- **Genérico**: 25 asignaciones de memoria.
- **Type erased**: 1,000,028 asignaciones de memoria.
- **Type erased optimizado**: 28 asignaciones de memoria.

Tiempo de ejecución

Tamaño	Clásico	OO	Genérico	Type-erased	Type-erased Opt
10^3	1.908	2.011	1.713	2.036	1.550
10^4	4.187	4.720	4.112	4.610	4.885
10^5	28.227	35.818	28.979	34.959	33.797
10^6	248.014	305.680	238.878	289.829	258.668
10^7	2427.600	3038.769	2410.310	3001.076	2636.193

Código fuente disponible

<https://github.com/jdgarciauc3m/polyexamples>

Polimorfismo dinámico versus polimorfismo estático

Flexibilidad versus rendimiento

J. Daniel Garcia

Grupo ARCOS
Universidad Carlos III de Madrid

24 de noviembre de 2016