DISEÑO DE SISTEMAS SOFTWARE

INTRODUCCIÓN

Problemáticas

Variabilidad

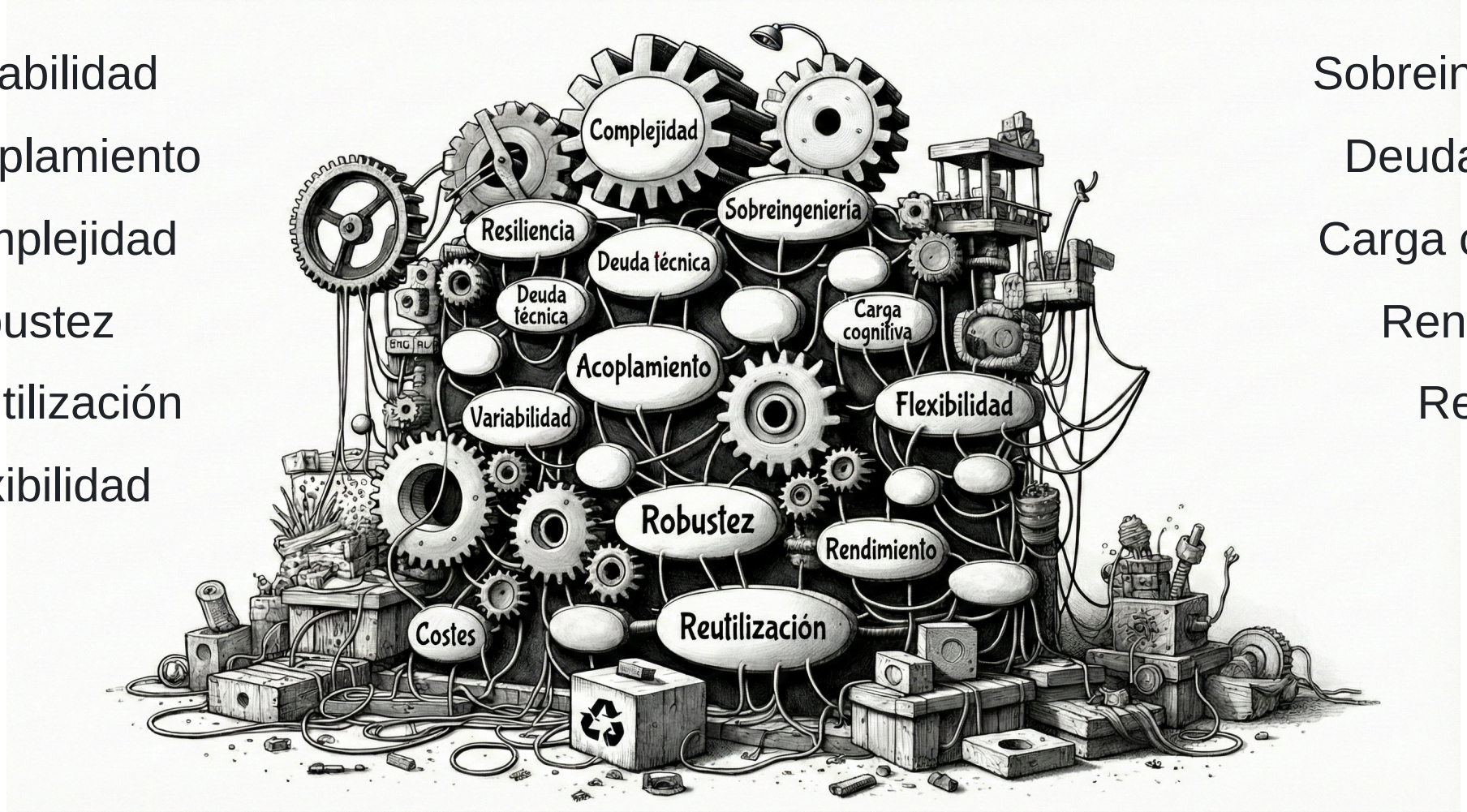
Acoplamiento

Complejidad

Robustez

Reutilización

Flexibilidad



Sobreingeniería

Deuda técnica

Carga cognitiva

Rendimiento

Resiliencia

Costes

Técnicas

- Refactoring
- Inyección de dependencias
- Contratos
- Aspectos
- Patrones
- Frameworks

Principios

- SRP: Single Responsibility Principle
- OCP: Open/Closed Principle
- ISP: Interface Segregation Principle
- LSP: Liskov Substitution Principle
- DIP: Dependency Inversion Principle

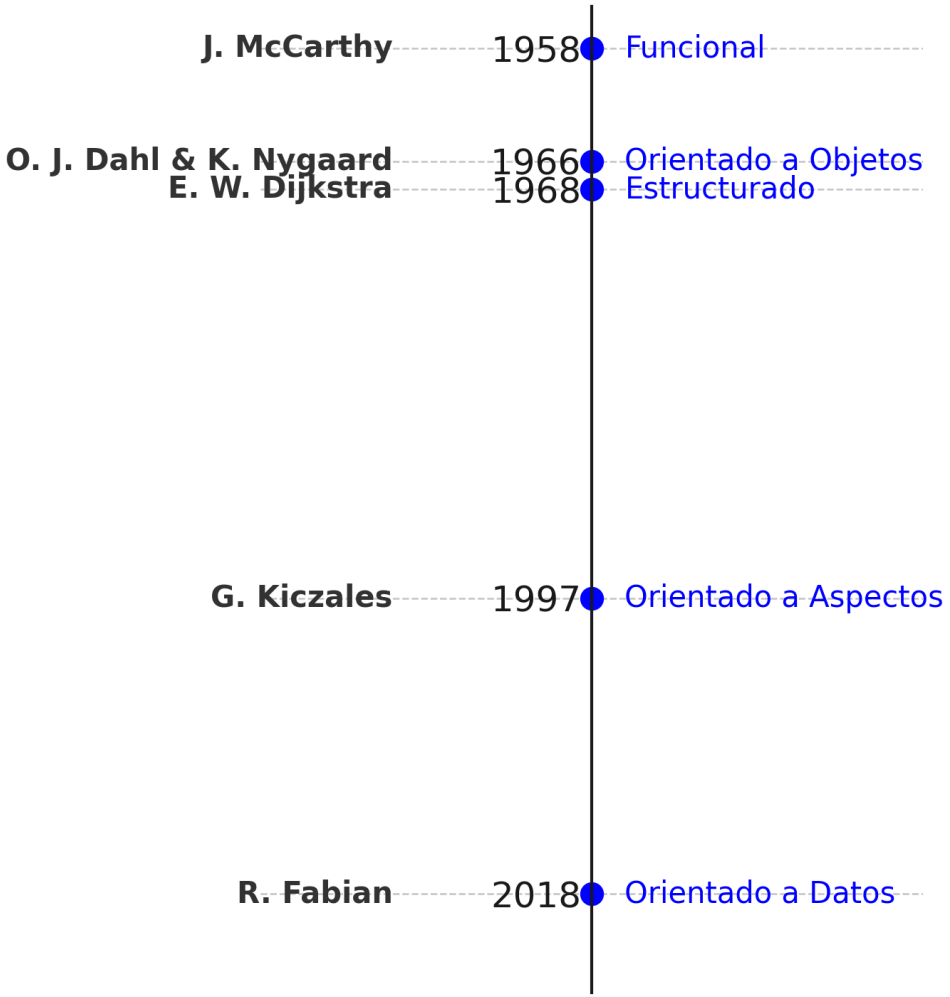
Paradigmas

- Estructurado (E. W. Dijsktra)
- Orientado a objetos (O. J. Dahl & K. Nygaard)
- Funcional (J. McCarthy)
- Orientado a aspectos (G. Kiczales)
- Orientado a datos (R. Fabian)

Preguntas

¿De qué fecha data cada paradigma?

Ordenar cronológicamente



Respuesta

¿De qué fecha data cada paradigma?

- Funcional (1958)
- Orientado a Objetos (1966)
- Estructurado (1968)
- Orientado a Aspectos (1997)
- Orientado a Datos (2018)

Bloques

- I. Fundamentos de diseño
- II. Principios de diseño OO
- III. Patrones de diseño
- IV. Arquitectura de software

FUNDAMENTOS DE DISEÑO

Casos prácticos

Caso 1. Identificadores

Caso 2. Framework de pruebas unitarias

Caso 3. Caballeros de la mesa redonda

Caso 4. Figuras geométricas

Conceptos teóricos

- Cohesión y acoplamiento
- Bibliotecas y frameworks
- Inyección de dependencias / Inversión de control
- Reutilización y flexibilidad
- Principios SOLID
- Orientación a aspectos
- Diseño por contratos

CASO PRÁCTICO 1

Identificadores

¿Cómo diseñar la identificación de los empleados de una empresa?

Versión inicial: Identificadores v0.1

```
class Empleado implements Comparable<Empleado> {
  private final int dni;

  Empleado(String dni) {
    this.dni = Integer.parseInt(dni);
  }

  public int getDni() {
    return dni;
  }

  @Override
  public String toString() {
    return Integer.toString(dni);
  }
```

```
  @Override
  public int compareTo(Empleado otro) {
    return Integer.compare(this.dni,
                           otro.getDni());
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Empleado otro))
      return false;
    return dni == otro.dni;
  }

  @Override
  public int hashCode() {
    return Integer.hashCode(dni);
  }
}
```

Nuevos requisitos:

- El Real Decreto 338/1990 regula el uso de NIFs en lugar de DNIs. ¡Hay que cambiar toda la implementación!

Implementación alternativa: Identificadores v0.2

```
class Empleado implements Comparable<Empleado> {
  private final String nif;

  Empleado(String nif) {
    this.nif = nif;
  }

  public String getNif() {
    return nif;
  }

  @Override
  public String toString() {
    return nif;
  }

  @Override
  public int compareTo(Empleado otro) {
    return this.nif.compareTo(otro.getNif());
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Empleado otro)) return false;
    return nif.equals(otro.nif);
  }

  @Override
  public int hashCode() {
    return nif.hashCode();
  }
}
```

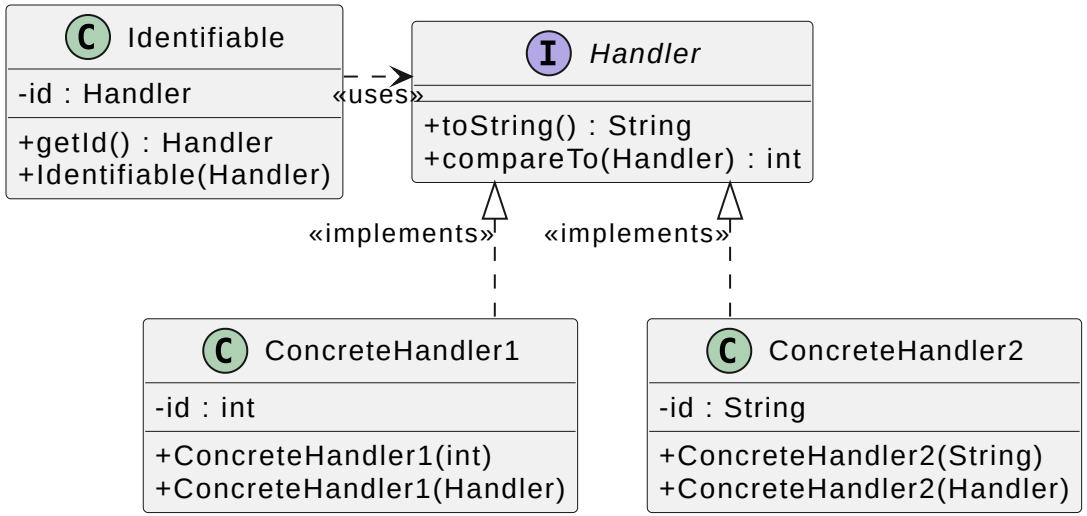
Críticas:

- **Flexibilidad** / **Reutilización**: Posiblemente haya más situaciones (algunas futuras) donde hagan falta *identificadores* que incluso pueden cambiar.
- Ejemplo: UUID, números de la seguridad social, tarjetas de identidad, números de cuenta corriente, IBAN, etc.

Hacemos refactoring: patrón *handler*

- Manejo de *identificadores* de forma independiente de la implementación del objeto identificado.
- Cambio fácil de implementación de los identificadores (`int`, `String`, etc.) hacia cualquier tipo básico o clase primitiva, sencilla o compuesta.

Diseño de un Handler



- **Identifiable**: Clase cliente que necesita identificar a sus objetos a través de algún atributo identificador
- **Handler**: Interfaz para declarar los identificadores de los objetos de la clase `Identifiable`
- **ConcreteHandler**: Implementación concreta de la interfaz `Handler`

Implementación del diseño en Java

```
interface Handler<T extends Comparable<? super T>>
    extends Comparable<Handler<T>> {
  T getId();

  @Override
  public boolean equals(Object o);

  @Override
  default int compareTo(Handler<T> otro) {
    return getId().compareTo(otro.getId());
  }
}
```

```
final class IdentificadorNumerico
    implements Handler<Integer> {
  private final Integer id;

  IdentificadorNumerico(String id) {
    this.id = Integer.valueOf(id);
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof IdentificadorNumerico otro))
      return false;
    return id.equals(otro.id);
  }

  @Override
  public Integer getId() {
    return id;
  }

  @Override
  public String toString() {
    return id.toString();
  }
}
```

Conceptos de diseño

- **Responsabilidad única**: La lógica de generación de identificadores (ej. limpiar caracteres, validar formato) debe estar aislada de la lógica de negocio principal.
- **Acoplamiento** (bajo): El sistema principal debe usar interfaces para generar IDs, sin depender de si se usa un UUID, una secuencia o un algoritmo custom.
- **Cohesión** (alta): Agrupar métodos relacionados con la manipulación de IDs en un único componente.
- **Encapsulamiento**: Ocultar los detalles algorítmicos de la construcción del identificador tras los métodos públicos.
- **Reutilización y flexibilidad**: Permitir la reutilización del componente de identificación en otros sistemas y facilitar la adaptación a futuros cambios.

Implementación en los lenguajes

Java: Identificadores con `java.lang.Comparable`

`Comparable<T>` es una interfaz que define el orden natural entre objetos del mismo tipo. La implementan `String`, `File`, `Date`, etc. y las *clases de envoltura* del JDK (i.e. `Integer`, `Long`, etc.)

Método de la interfaz:

```
  public int compareTo(T o)
```

El tipo `T` garantiza seguridad de tipos: un `Comparable<Empleado>` solo se compara con `Empleado`, evitando casts y errores en tiempo de ejecución.

Invariantes: las debe asegurar cualquier implementación de `compareTo`

`sgn(x.compareTo(y)) = -sgn(y.compareTo(x))`

`(x.compareTo(y)>0 and y.compareTo(z)>0)` $\Rightarrow$ `x.compareTo(z)>0`

`x.compareTo(y)=0` $\Rightarrow$ `sgn(x.compareTo(z))=sgn(y.compareTo(z))` $\forall$ `z`

Consistencia con `equals`: recomendable pero no exigible

`(x.compareTo(y)=0)` $\Leftrightarrow$ `(x.equals(y))`

C++: Comparación de identificadores

Cómo implementar la interfaz de comparación de un Handler en C++

¿Tienen sentido las siguientes implementaciones?

```
    static int compare(const Handler&, const Handler&);
    int compareTo(const Handler&); // member function
```

Ver **stackoverflow**

Sobrecarga de operadores en C++:

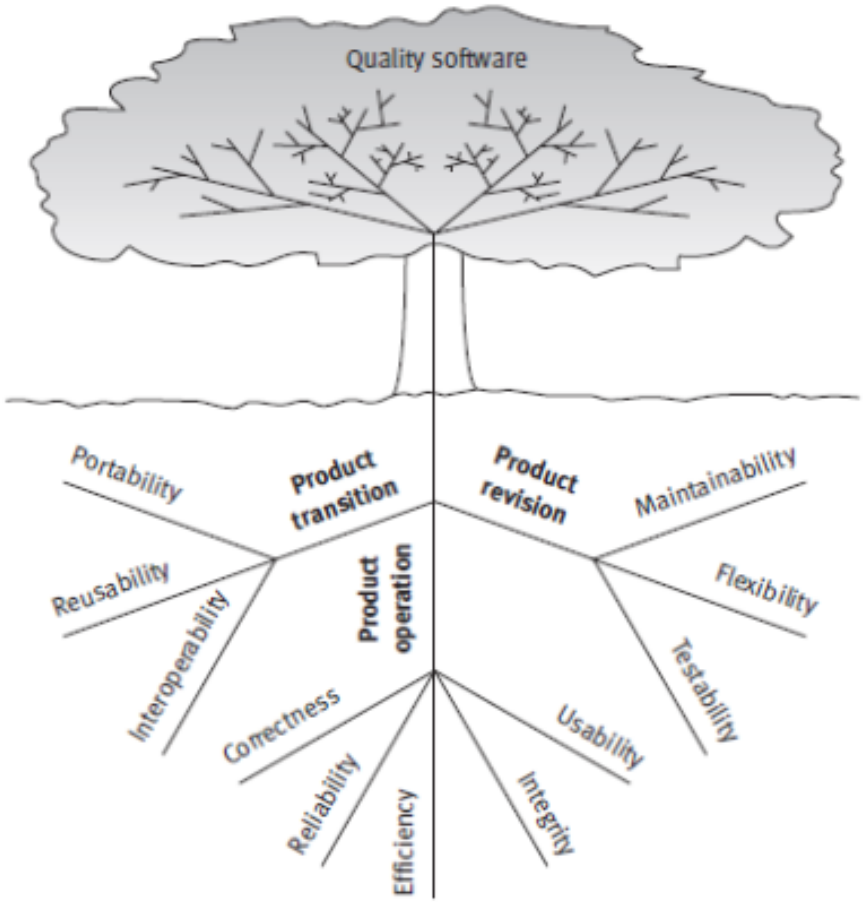
```cpp
template <typename T>
struct Handler {
    T id;
    explicit Handler(T v) : id(std::move(v)) {}
    bool operator==(const Handler& other) const = default;
    auto operator<=>(const Handler& other) const = default;
};
using NumericHandler = Handler<int>;
using StringHandler  = Handler<std::string>;
```

```cpp
int main() {
    StringHandler a{"EMP-001"};
    StringHandler b{"EMP-002"};
    auto ord = (a <=> b);  // OK
    NumericHandler c{123};
    // bool same = (a == c);  // ERROR: tipos distintos
}
```

Reutilización y flexibilidad

- **Reutilización**: Construir software fácil de reutilizar sin tener que cambiar los módulos ya escritos (afecta a la fase de **desarrollo**)
- **Flexibilidad**: Adaptarse a cambios de requisitos y construir software fácil de cambiar (afecta a la fase de **mantenimiento adaptativo**)

El árbol de la calidad del software



CASO PRÁCTICO 2

Pruebas unitarias

jUnit: Framework de pruebas unitarias

- JUnit es un framework en Java que sirve para diseñar, construir y ejecutar **pruebas unitarias**
- Una prueba unitaria comprueba la corrección de un *módulo* de software en cuanto a funcionalidades que ofrece.
- En el caso de Java, las pruebas unitarias comprueban la corrección de cada uno de los métodos de *cada clase*.

¿Por qué? ¿Cómo funciona?

¿Cómo se probaba `Saludo.java` sin bibliotecas de pruebas unitarias?

```
class Saludo {
  /**
  * Imprime "Hola Mundo!"
  */
  void saludar() {
    System.out.println("Hola Mundo!");
  }

  /**
  * Imprime un mensaje
  */
  void saludar(String mensaje) {
    System.out.println(mensaje);
  }
```

Incluir un método `main` que pruebe la funcionalidad de la clase:

```
  /**
  * Tests
  */
  public static void main( String[] args ) {
    Saludo saludo1 = new Saludo();
    saludo1.saludar();

    Saludo saludo2 = new Saludo("Hola caracola!");
    saludo2.saludar();
  }
}
```

Pegas

- Cuanto más grande sea la interfaz de la clase, mayor será el `main`
- El tamaño del código de la clase crece por las pruebas
- Poco fiable, porque `main` forma parte de la misma clase y tiene acceso a los elementos privados
- Difícil de automatizar las pruebas, incluso pasando argumentos a `main`

Ejemplo: software *cliente* del framework jUnit

Caso de prueba con jUnit 4

```
import org.junit.*;
import static org.junit.Assert.*;

public class SaludoTest {
  public static void main(String args[]) {
    junit.textui.TestRunner.run(SaludoTest.class);
  }
  @Test
  public void saludar() {
    Saludo hola = new Saludo();
    assert( hola!=null );
    assertEquals("Hola Mundo!", hola.saludar() );
  }
}
```

Ejecución de los tests:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(SaludoTest.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

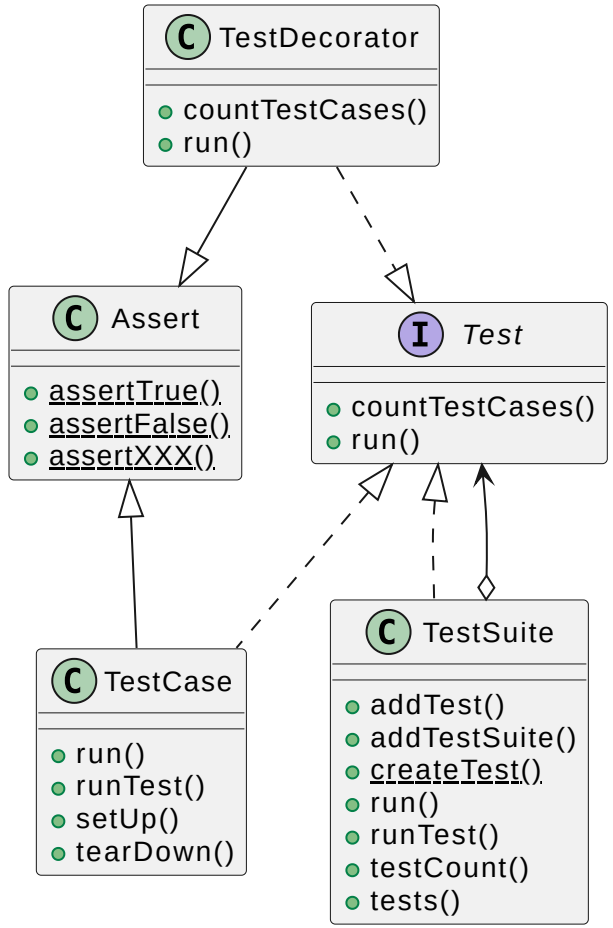
¿De qué están hechas las anotaciones como `@Test`?

Caso de prueba con jUnit 3

Veamos una versión anterior de jUnit, que expone más claramente las *tripas* del framework

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class SaludoTest extends TestCase {
    public SaludoTest(String nombre) {
      super(nombre);
    }
    public void testSaludar() {
      Saludo hola = new Saludo();
      assert( hola!=null );
      assertEquals("Hola Mundo!", hola.saludar() );
    }
}
```

Diseño del framework jUnit



Ejemplo: aplicación de comercio electrónico

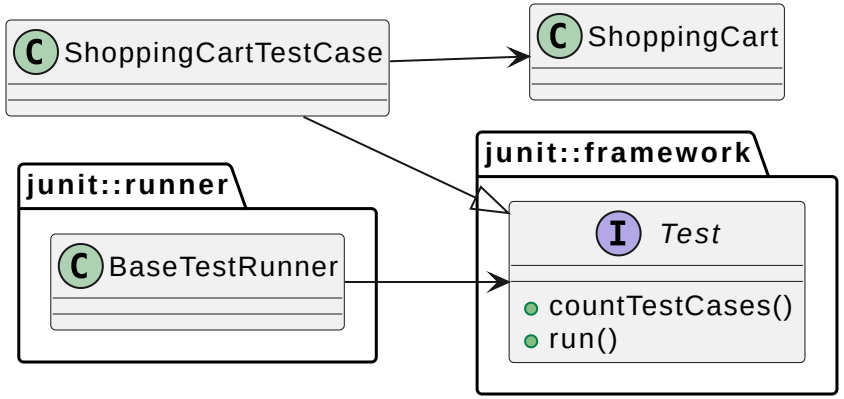
Diseño de una aplicación de comercio electrónico:

- `ShoppingCart` - carrito de la compra
- `CreditCard` - tarjeta de crédito
- `Product` - artículos
- Etc.

Diseño de pruebas unitarias de `ShoppingCart` para:

- Probar carrito de la compra (añadir/eliminar artículos)
- Probar validación de tarjetas de crédito
- Probar manejo de varias monedas
- Etc.

Utilización del framework jUnit



ShoppingCart

```
public class ShoppingCart {
  private ArrayList items;
  public ShoppingCart() { ... }
  public double getBalance() { ... }
  public void addItem(Product p) { ... }
  public void removeItem(Product p)
      throws ProductNotFoundException { ... }
  public int getItemCount() { ... }
  public void empty() { ... }
  public boolean isEmpty() { ... }
}
```

ShoppingCartTestCase con jUnit 3

```
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.framework.Assert;

public class ShoppingCartTest extends TestCase {
  private ShoppingCart bookCart;
  private Product defaultBook;
  //...
  protected void setUp() {
      bookCart = new ShoppingCart();
      defaultBook = new Product("Extreme Programming", 23.95);
      bookCart.addItem(defaultBook);
  }
  protected void tearDown() {
      bookCart = null;
  }
  public void testEmpty() {
      bookCart.empty();
      assertTrue(bookCart.isEmpty());
  }
  public void testProductAdd() {
      Product book = new Product("Refactoring", 53.95);
      bookCart.addItem(book);
      double expectedBalance = defaultBook.getPrice() + book.getPrice();
      assertEquals(expectedBalance, bookCart.getBalance(), 0.0);
      assertEquals(2, bookCart.getItemCount());
  }
```

```
  public void testProductRemove() throws ProductNotFoundException {
      bookCart.removeItem(defaultBook);
      assertEquals(0, bookCart.getItemCount());
      assertEquals(0.0, bookCart.getBalance(), 0.0);
  }
  public void testProductNotFound() {
      try {
          Product book = new Product("Ender's Game", 4.95);
          bookCart.removeItem(book);
          fail("Should raise a ProductNotFoundException");
      } catch(ProductNotFoundException success) {
          ...
      }
  }
  public static Test suite() {
      // Use reflection to add all testXXX() methods
         TestSuite suite = new TestSuite(ShoppingCartTest.class);
      // Alternatively, but prone to error when adding more
      // test case methods...
      // TestSuite suite = new TestSuite();
      // suite.addTest(new ShoppingCartTest("testProductAdd"));
      // suite.addTest(new ShoppingCartTest("testEmpty"));
      // suite.addTest(new ShoppingCartTest("testProductRemove"));
      // suite.addTest(new ShoppingCartTestCase("testProductNotFound"));
         return suite;
  }
}
```

Podemos agrupar varios casos de prueba en una misma *suite*:

```
import junit.framework.Test;
import junit.framework.TestSuite;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class EcommerceTestSuite extends TestSuite {
    //...
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(ShoppingCartTest.suite());
        return suite;
    }
}

public class MyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(EcommerceTestSuite.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

ShoppingCartTestCase con jUnit 4

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class ShoppingCartTest {
  private ShoppingCart bookCart;
  private Product defaultBook;
  //...
  @Before
  protected void setUp() {
      bookCart = new ShoppingCart();
      defaultBook = new Product("Extreme Programming", 23.95);
      bookCart.addItem(defaultBook);
  }
  @After
  protected void tearDown() {
      bookCart = null;
  }
```

```
  @Test
  public void testEmpty() {
      bookCart.empty();
      assertTrue(bookCart.isEmpty());
  }
  @Test
  public void testProductAdd() {
      Product book = new Product("Refactoring", 53.95);
      bookCart.addItem(book);
      double expectedBalance = defaultBook.getPrice() + book.getPrice();
      assertEquals(expectedBalance, bookCart.getBalance(), 0.0);
      assertEquals(2, bookCart.getItemCount());
  }
  @Test
  public void testProductRemove() {
      bookCart.removeItem(defaultBook);
      assertEquals(0, bookCart.getItemCount());
      assertEquals(0.0, bookCart.getBalance(), 0.0);
  }
  @Test(expected = ProductNotFoundException.class)
  public void testProductNotFound() {
      Product book = new Product("Ender's Game", 4.95);
      bookCart.removeItem(book);
      fail("Should raise a ProductNotFoundException");
  }
}
```

EcommerceTestSuite con jUnit 3

```
public class EcommerceTestSuite extends TestSuite {
  //...
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(ShoppingCartTest.suite());
    suite.addTest(CreditCardTest.suite());
    // etc.
    return suite;
  }
}
```

EcommerceTestSuite con jUnit 4

```
@RunWith(Suite.class)
@SuiteClasses({
    ShoppingCartTest.class,
    CreditCardTest.class
})
public class EcommerceTestSuite {
    //...
}
```

Pregunta

¿Qué hemos conseguido con las anotaciones `@Test` del JDK $\geq$ 1.5?

¿Qué hemos conseguido con las anotaciones `@Test`?

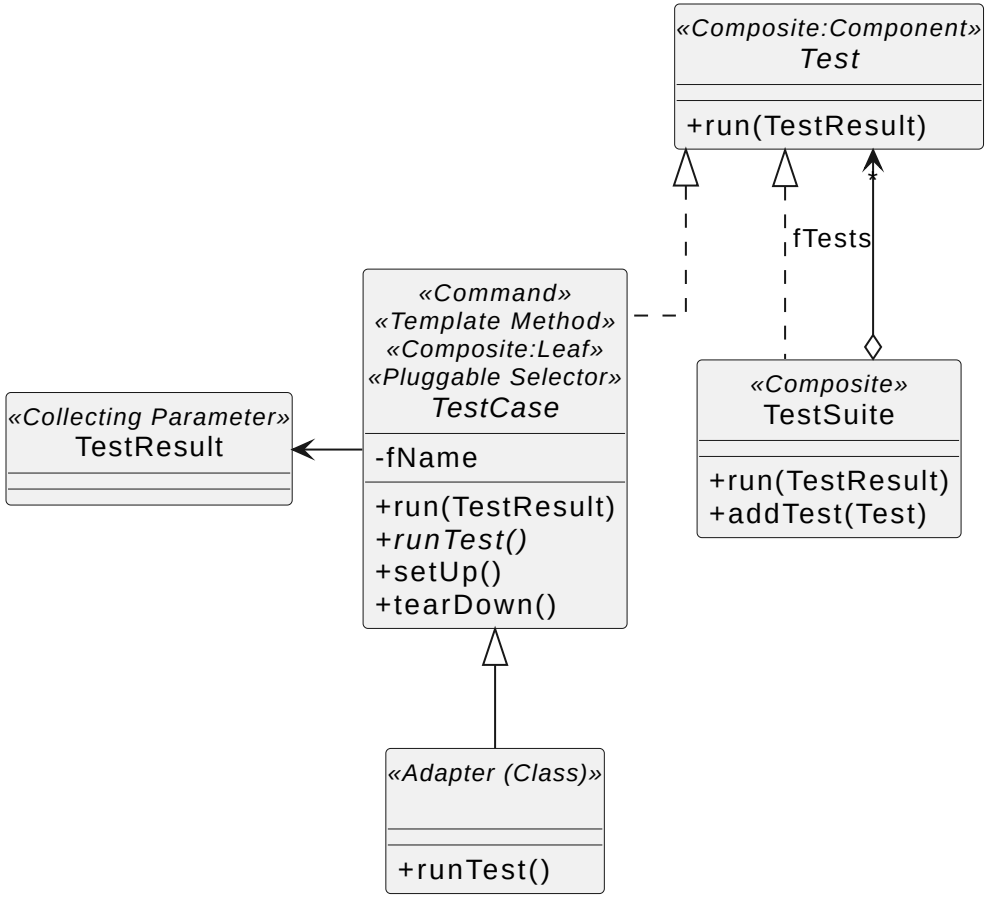
Respuesta:

- No necesitar de la incómoda herencia (i.e. mecanismo de implementación)

Ejercicio propuesto: CreditCardTest

Diseñar y codificar una suite de casos de prueba unitaria para `CreditCard` usando jUnit versión 4.

Arquitectura del framework jUnit

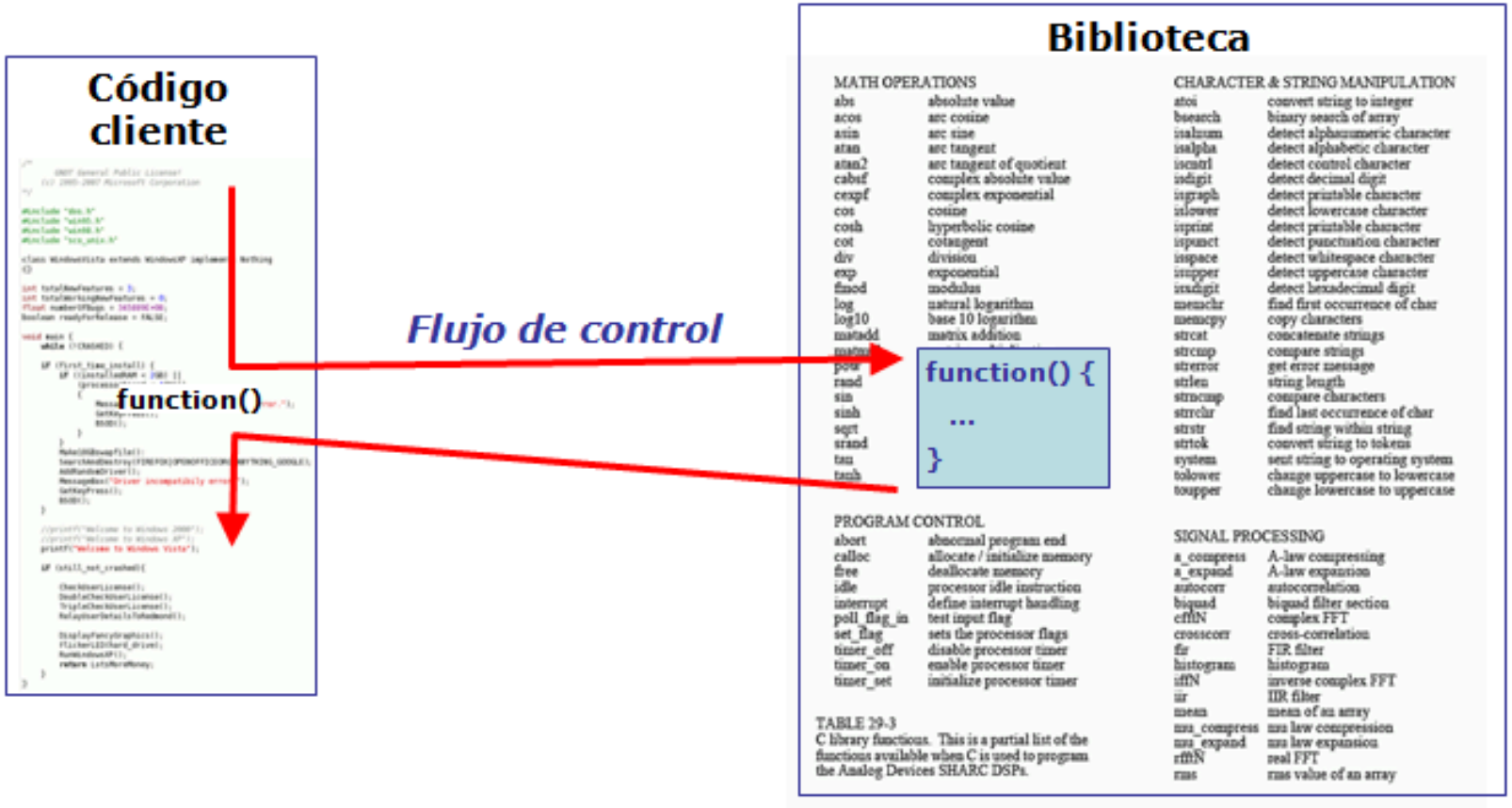


En la arquitectura del framework se observan diversos patrones:

- Composite
- Command
- Adapter
- Factory
- Decorator
- etc.

Bibliotecas y frameworks

Flujo de control en una biblioteca



MATH OPERATIONS

abs	absolute value
acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	arc tangent of quotient
cabsf	complex absolute value
cexpf	complex exponential
cos	cosine
cosh	hyperbolic cosine
cot	cotangent
div	division
exp	exponential
fmod	modulus
log	natural logarithm
log10	base 10 logarithm
matadd	matrix addition
pow	
rand	
sin	
sinh	
sqrt	
srand	
tan	
tanh	

CHARACTER & STRING MANIPULATION

atoi	convert string to integer
bsearch	binary search of array
isalnum	detect alphanumeric character
isalpha	detect alphabetic character
iscntrl	detect control character
isdigit	detect decimal digit
isgraph	detect printable character
islower	detect lowercase character
isprint	detect printable character
ispunct	detect punctuation character
isspace	detect whitespace character
isupper	detect uppercase character
isxdigit	detect hexadecimal digit
memchr	find first occurrence of char
memcpy	copy characters
strcat	concatenate strings
strcmp	compare strings
strerror	get error message
strlen	string length
strncmp	compare characters
strrchr	find last occurrence of char
strstr	find string within string
strtok	convert string to tokens
system	sent string to operating system
tolower	change uppercase to lowercase
toupper	change lowercase to uppercase

PROGRAM CONTROL

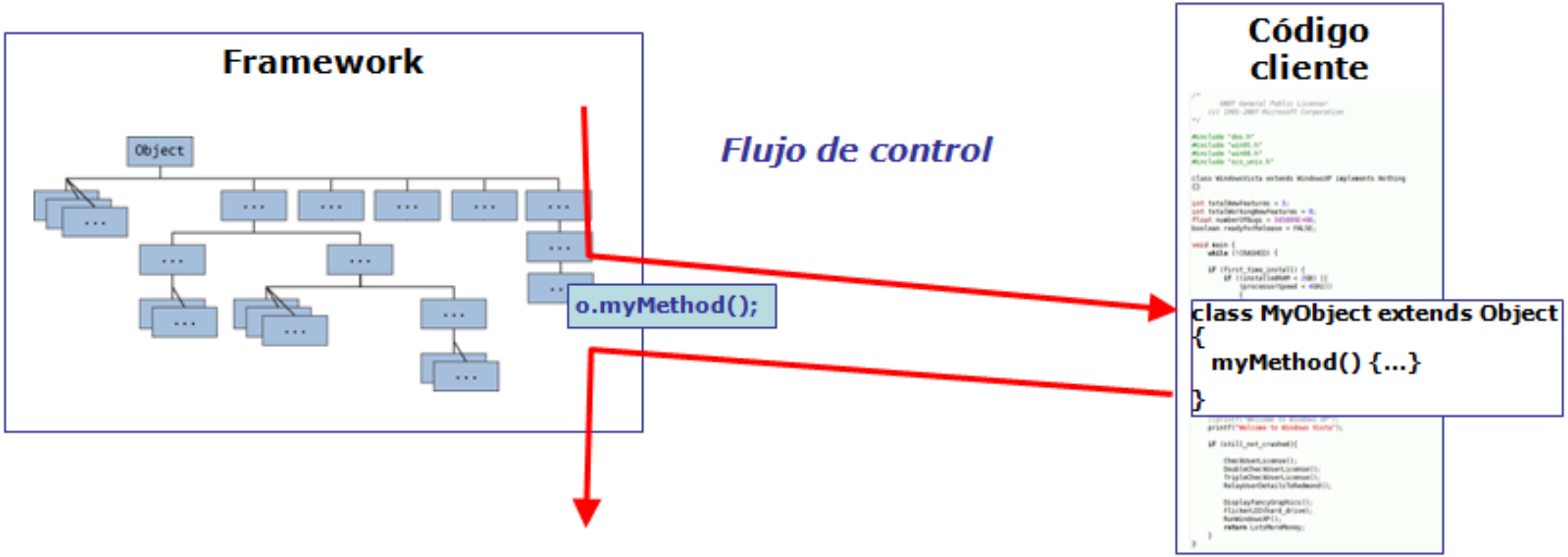
abort	abnormal program end
calloc	allocate / initialize memory
free	deallocate memory
idle	processor idle instruction
interrupt	define interrupt handling
poll_flag_in	test input flag
set_flag	sets the processor flags
timer_off	disable processor timer
timer_on	enable processor timer
timer_set	initialize processor timer

SIGNAL PROCESSING

a_compress	A-law compressing
a_expand	A-law expansion
autocorr	autocorrelation
biquad	biquad filter section
cfftN	complex FFT
crosscorr	cross-correlation
fir	FIR filter
histogram	histogram
ifftN	inverse complex FFT
iir	IIR filter
mean	mean of an array
mu_compress	mu law compression
mu_expand	mu law expansion
rfftN	real FFT
rms	rms value of an array

TABLE 29-3
C library functions. This is a partial list of the functions available when C is used to program the Analog Devices SHARC DSPs.

Flujo de control en un framework



Frameworks

Definición de *framework*

> Colección de clases e interfaces que cooperan para formar un diseño reutilizable de un tipo específico de software
>
> — E. Gamma et al.

- El framework proporciona unas guías arquitectónicas (diseño empaquetado) para dividir el diseño en clases abstractas y definir sus *responsabilidades* y *colaboraciones*.
- El framework se debe personalizar definiendo subclases y combinando instancias, o bien configurando valores que definen el comportamiento por defecto

Principios de diseño de un framework OO

- Datos encapsulados
- Interfaces y clases abstractas
- Métodos polimórficos
- Delegación

Herramientas de diseño OO

- **Patrones**: elementos reutilizables de diseño
- **Frameworks**: colecciones de patrones abstractos a aplicar

Framework vs. biblioteca

- API orientado a objetos vs. API basado en funciones (en general)
- Flujo de control invertido
- Programador *cliente* (código específico) vs. programador de *API* (código reutilizable)

Principios y técnicas de un framework

- **Abstracción**
  - Clases y componentes abstractos
  - Interfaces abiertas
  - Uso de patrones de diseño
  - Componentes de un dominio específico
- Máxima **cohesión**, mínimo **acoplamiento**
  - Minimizar dependencias: Una clase A presenta una dependencia con otra clase B ($A \rightarrow B$) si la primera usa una instancia de la segunda.
  - Cuando no se pueden eliminar las dependencias, mantener las abstractas e **inyectar** las concretas.

Dependencias

> Coupling is the enemy of change, because it links together things that must change in parallel
>
> D. Thomas & A. Hunt, The Pragmatic Programmer, 20th Anniversary Edition, 2019

- La reducción de dependencias debe ser estratégica, es decir, centrada en los puntos del sistema que cambian con distinta frecuencia.
- Si A y B cambian al mismo tiempo, no hay demasiado problema porque A $\rightarrow$ B
- Si cambian con distinta frecuencia...

Inyección de dependencias

Una clase o módulo no debería configurar sus dependencias estáticamente, sino ser configurada desde fuera

CASO PRÁCTICO 3

Caballeros de la mesa redonda

Añadir pruebas unitarias al programa siguiente:

```
public class KnightOfTheRoundTable {
  private String name;
  private HolyGrailQuest quest;

  public KnightOfTheRoundTable(String name) {
    this.name = name;
    quest = new HolyGrailQuest();
  }
  public HolyGrail embarkOnQuest()
      throws GrailNotFoundException {
    return quest.embark();
  }
}
```

```
public class HolyGrailQuest {
  public HolyGrailQuest() {
    /*...*/
  }

  public HolyGrail embark()
          throws GrailNotFoundException {
    HolyGrail grail = null;
    // Look for grail ...
    return grail;
  }

}
```

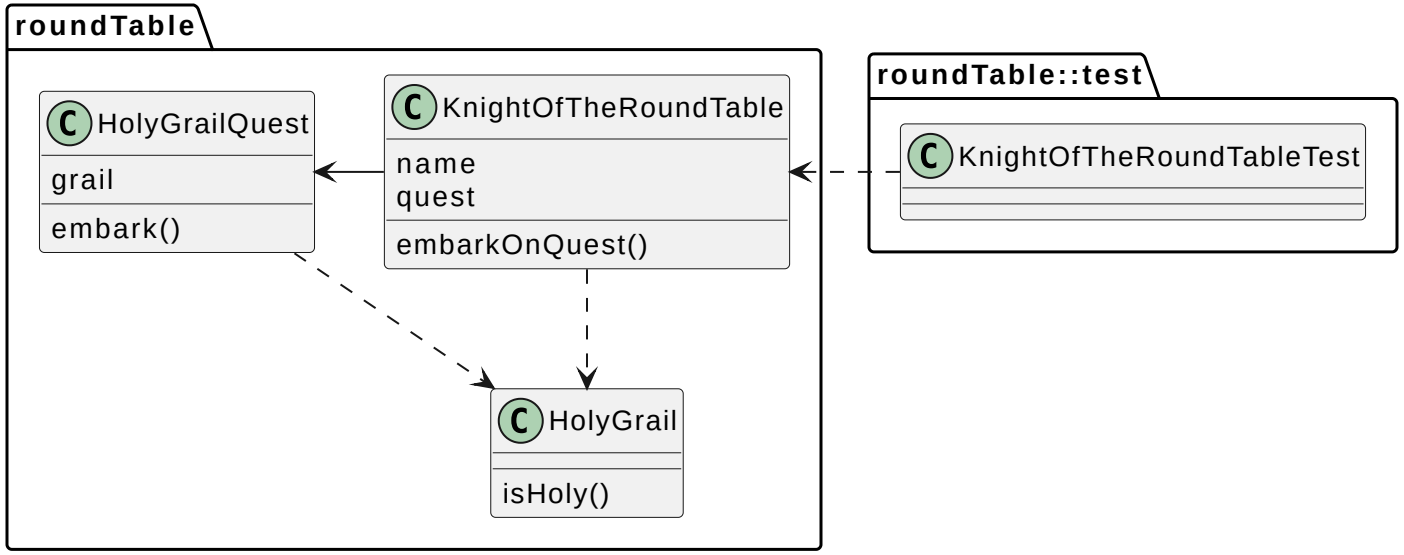
Diseño de pruebas con jUnit 3

¿Dónde está el acoplamiento?

```
import junit.framework.TestCase;

public class KnightOfTheRoundTableTest extends TestCase {
  public void testEmbarkOnQuest() throws GrailNotFoundException {
    KnightOfTheRoundTable knight =
        new KnightOfTheRoundTable("CruzadoMagico");
    HolyGrail grail = knight.embarkOnQuest();
    assertNotNull(grail);
    assertTrue(grail.isHoly());
  }
}
```

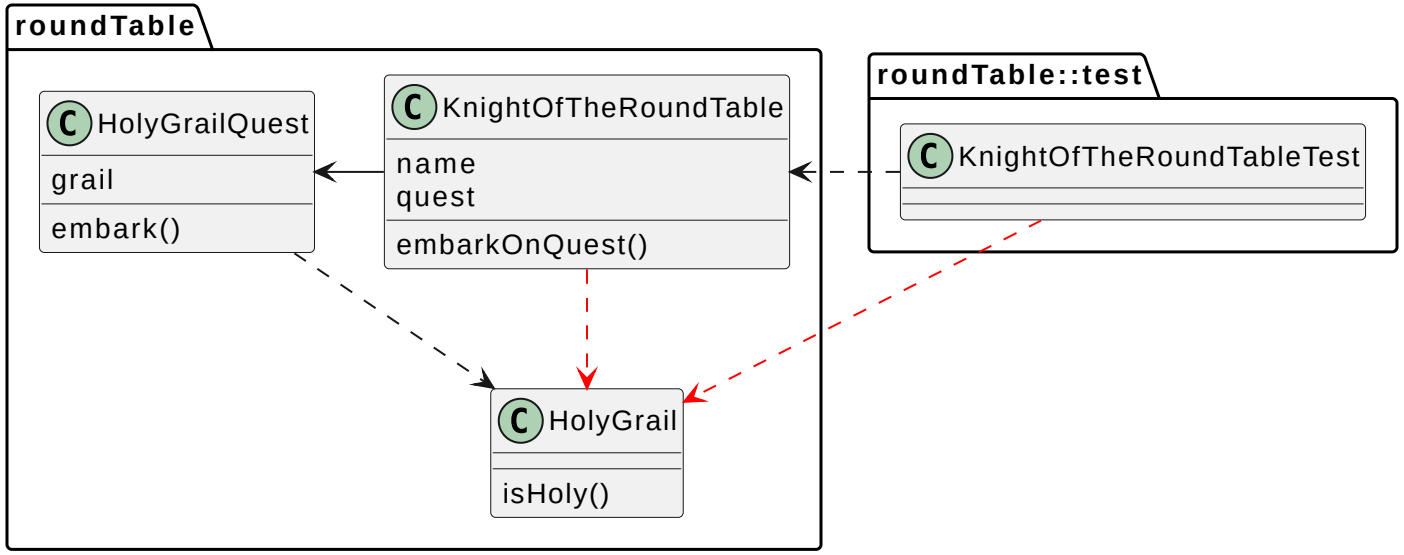
Diagrama de clases

¿Dónde está el acoplamiento?



Acoplamiento

No deseable



Pegas:

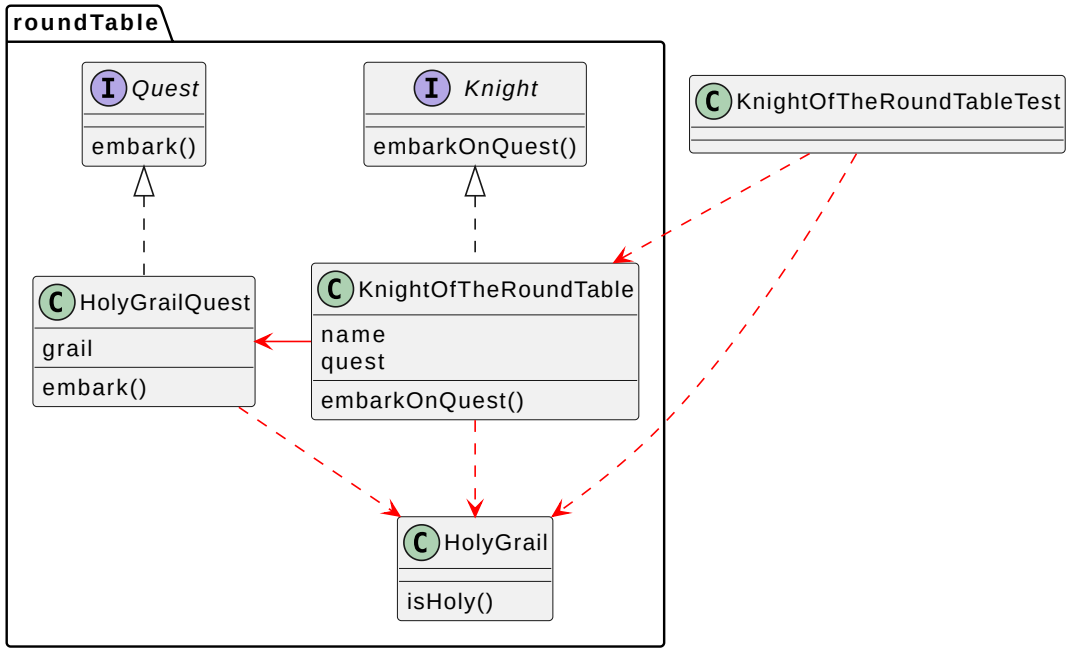
- Instanciación de `HolyGrail`
- Cada vez que se prueba `KnightOfTheRoundTable`, también se prueba `HolyGrailQuest`.
- No se puede pedir a `HolyGrailQuest` que se comporte de otra forma (v.g. devolver null o elevar una excepción)

Ocultar la implementación detrás de una interfaz:

```
public interface Knight {
  Object embarkOnQuest()
          throws QuestFailedException;
}

public class KnightOfTheRoundTable
             implements Knight {
  private String name;
  private Quest quest;

  public KnightOfTheRoundTable(String name) {
    this.name = name;
    quest = new HolyGrailQuest();
  }
  public Object embarkOnQuest()
          throws QuestFailedException {
    return quest.embark();
  }
}
```

```
public interface Quest {
  abstract Object embark()
    throws QuestFailedException;
}

public class HolyGrailQuest implements Quest {
  public HolyGrailQuest() { /*...*/ }
  public Object embark()
          throws QuestFailedException {
    // Do whatever it means
    // to embark on a quest
    return new HolyGrail();
  }
}
```

Dependencias

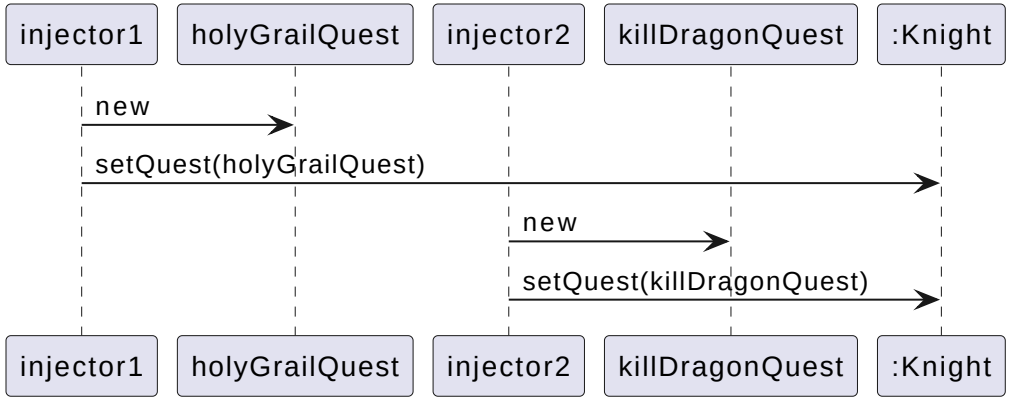


Pegas:

- El `KnightOfTheRoundTable` aún depende de un tipo específico de `Quest` (i.e. `HolyGrailQuest`) obtenido mediante `new`

¿Debe ser el caballero responsable de obtener un desafío?

```java
public class KnightOfTheRoundTable
                implements Knight {
  private String name;
  private Quest quest;

  public KnightOfTheRoundTable(String name) {
    this.name = name;
  }
  public Object embarkOnQuest()
          throws QuestFailedException {
    return quest.embark();
  }
  public void setQuest(Quest quest) {
    this.quest = quest;
  }
}
```



- El caballero sólo sabe del desafío a través de su interfaz `Quest`.
- Puede asignársele cualquier implementación de `Quest` (`HolyGrailQuest`, `KillDragonQuest`, etc.)

- Parece que no hay dependencia entre `KnightOfTheRoundTable` y `HolyGrail` porque `embark()` se ha definido como que devuelve un `Object`

  Ejercicio: Discutir el tipo de retorno `Object` de `embarkOnQuest`:

- Puede provocar `ClassCastException`
- Solución propuesta: rediseñar la interfaz `Quest`

Inversión de control

Es la base de la inyección de dependencias

> The question is: *what aspect of control are they inverting?* [...] Early **user interfaces** were controlled by the application program. You would have a sequence of commands like "Enter name", "enter address"; your program would drive the prompts and pick up a response to each one. With **graphical** (or even screen based) UIs the UI framework would contain this main loop and your program instead provided event handlers for the various fields on the screen. The main control of the program was inverted, moved away from you to the framework.
>
> — Martin Fowler, IoC containers and the DI pattern [1]

[1] http://martinfowler.com/articles/injection.html

IoC–Inversion of Control / DI–Dependency Injection

- Una aplicación está compuesta por dos o más clases que colaboran.
- Los objetos deben recibir las dependencias en su creación, por parte de una entidad externa o **contenedor** que los coordina.
- IoC = Inversión de la responsabilidad de cómo un objeto obtiene referencias a los objetos con los que colabora
- Ventaja = **bajo acoplamiento**: un objeto sólo sabe de sus dependencias por su *interfaz*, no por su *implementación*, ni por cómo fueron instanciados.
- Entonces la dependencia puede cambiarse por una implementación distinta (incluso en **tiempo de ejecución**)
- *Hollywood Principle: Don't call us, we'll call you".*

Factorías

Una factoría proporciona un mecanismo de inyección de dependencias, visto desde el lado opuesto (los clientes adquieren las dependencias, no se les inyecta)

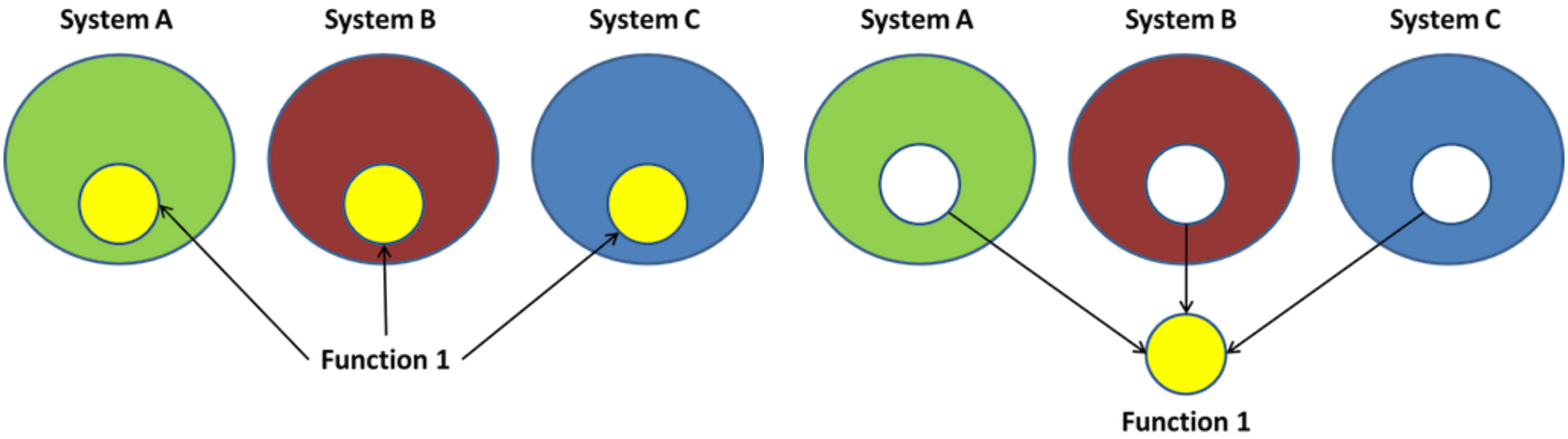
Ejemplo: Spring FactoryBean

Discusión sobre la reutilización

> We most likely would have been better off not attempting to create a reusable function in the first place
>
> — Roger Sessions, The Misuse of Reuse [2]

[2] http://simplearchitectures.blogspot.com.es/2012/07/misuse-of-reuse.html

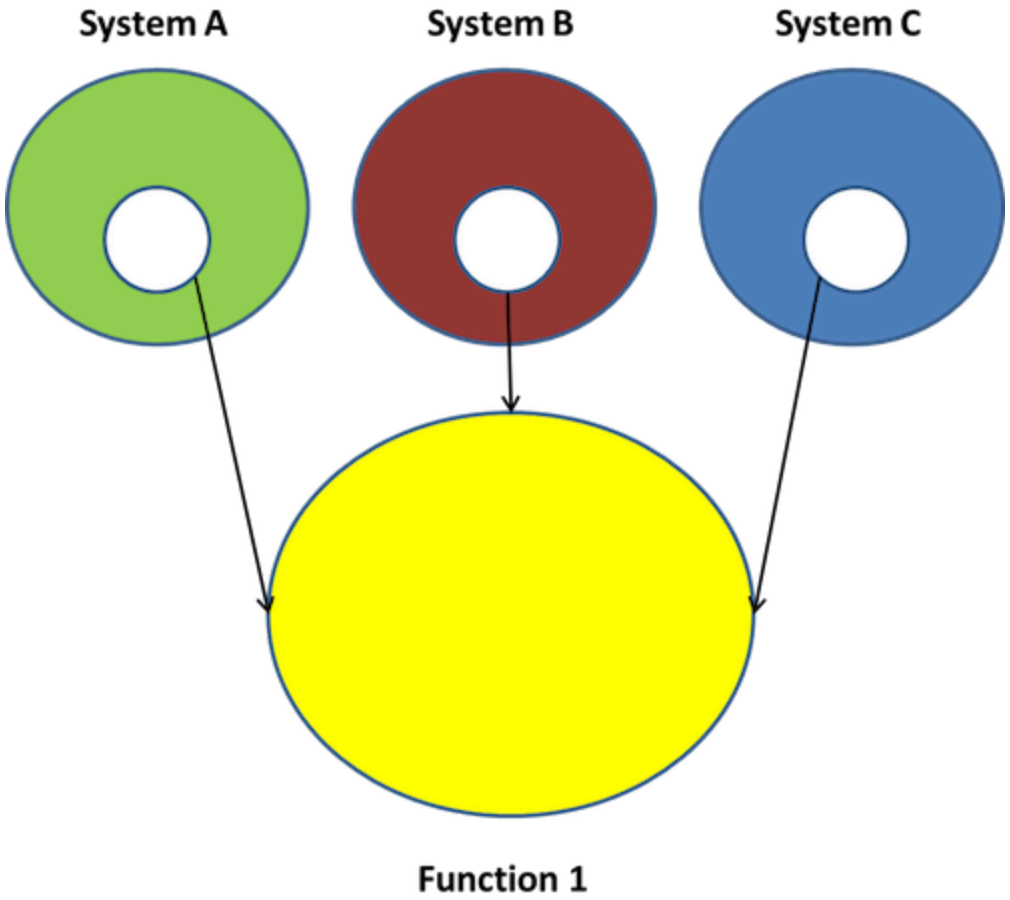
Factorizar una función



Ventajas (supuestas) de reutilizar

Ahorro: Si $\exists\ s$ sistemas $\wedge\ coste(Function\ 1) = c \Rightarrow$ ahorro = $c \times (s - 1)$

Amenazas (reales) a la reutilización



- Realmente el ahorro depende de la **complejidad** de la función, que suele estar relacionada exponencialmente con el número de sistemas.
- Con un único punto de fallo, si $Function1$ falla, todos los sistemas pueden fallar a la vez.
- La seguridad es inversamente proporcional a la complejidad del sistema.

Conclusión sobre la reutilización

- Aplicar el principio **YAGNI** (*You Ain't Gonna Need It*)
- No crear funciones reutilizables en primer lugar
- No hacer sobreingeniería
- Cuidado con las abstracciones prematuras
- Comprobar el acoplamiento semántico entre sistemas
- No aplicar patrones de diseño antes de tiempo
- Aplicar YAGNI es un pedir un *préstamo* (asumir una deuda técnica conscientemente)