Rival: un patrón de comportamiento de uso organizacional

Diego Sánchez[1], Carlos Fontela[1]

[1] Laboratorio de Métodos de Desarrollo y Mantenimiento de Software (LIMET),
Facultad de Ingeniería Universidad de Buenos Aires, Argentina
y Universidad Nacional de Tres de Febrero, Argentina
diego.sanches@gmail.com, cfontela@fi.uba.ar

Resumen. Los patrones de diseño fueron introducidos ya hace más de tres décadas, induciendo mejoras en la manera en que desarrollamos software orientado a objetos. Sin embargo, día a día se descubren nuevos patrones en situaciones concretas de proyectos de desarrollo. Este trabajo presenta un patrón de comportamiento del que hemos encontrado un uso difundido en ambientes organizacionales, y al que hemos denominado Rival.

Palabras clave: patrones de diseño, patrones de comportamiento, principios de diseño, cobertura, acoplamiento.

1 Introducción

1.1 Patrones de diseño

Los patrones de diseño, presentados por Ward Cunningham y Kent Beck en OOPSLA-87 [1] y popularizados por el libro de Gamma et al. [2], son soluciones de diseño a problemas comunes, en la forma de descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto. Luego de esas primeras publicaciones, otros autores han presentado numerosos patrones de diseño (cf. [3]) que han demostrado su aplicabilidad en la industria.

Es importante destacar que los patrones no se crean, sino que se descubren al ver su uso en muchos sistemas [2]. Por eso, se encuentran nuevos patrones todo el tiempo. El presente artículo se va a enfocar en un patrón que resuelve algunos problemas recurrentes de diseño de software en organizaciones.

De las muchas clasificaciones de patrones, la más usual los divide en patrones creacionales, de comportamiento y estructurales. En particular, los patrones de comportamiento son aquéllos que identifican situaciones de comunicación entre objetos e

incrementan la flexibilidad en esa comunicación. En general, suelen centrarse en encapsular lo que varía y desacoplar emisores y receptores [2].

1.1 Cobertura y situaciones que empeoran la cobertura

El análisis de cobertura es el proceso de hallar áreas de un programa que no están siendo recorridas por un conjunto de casos de prueba y determinar una medida cuantitativa de cobertura de código, que sea una medida indirecta de la calidad de las pruebas [4].

Las acciones condicionales (*if*, *switch*, *case*), por su propia naturaleza, reducen la cobertura de ramas [5]. Por ello, es deseable que las acciones condicionales de un programa se reduzcan para minimizar las pruebas necesarias para cubrir las distintas ramas del código.

2 Rival como patrón de comportamiento

2.1 Problema

Algunas implementaciones del paradigma de objetos descansan en la existencia de la estructura de control *if*, demandando la utilización de expresiones que deben reducirse a un valor lógico. Para lograrlo los objetos del modelo colaboran exponiendo propiedades con la intención de que haya terceros que los interpelen y decidan en función de los valores que tomen esas propiedades. Esto trae aparejadas las siguientes consecuencias:

- Acoplamiento: La exposición de propiedades de un objeto provoca un acoplamiento entre el interpelador y el interpelado. Esto viola el principio de ocultamiento de la información [6].
- Responsabilidad compartida: La existencia de la cláusula *if* manifiesta la responsabilidad compartida entre el objeto que contiene la cláusula y el objeto interpelado teniendo consecuencias negativas. Esto se conoce como “Smell” Shotgun Surgery [13].
- Cobertura: Como ya dijimos, la interpelación producida por la utilización de la cláusula *if* puede producir una baja de cobertura [5].
- Violación de principios de diseño: El objeto interpelador toma partido en función del comportamiento observado en el objeto interpelado. Esto viola el principio *Tell Don't Ask* [7]. Adicionalmente, los futuros cambios implicarán modificaciones en código existente, violando el principio *Open/Close* [8].

2.2 Caso de estudio

En el contexto de una plataforma de comercio electrónico un usuario compra un artículo pagándolo con un medio de pago y especificando una forma de envío.

A los efectos de la descripción del problema vamos a considerar la existencia de dos grandes grupos de medios de pagos, a saber: Acreditación inmediata (i.e, tarjetas de crédito) y Acreditación diferida (medios de pago offline). Además, para simplificar, vamos a considerar que sólo existe un único medio de envío y este cuenta con una fecha y hora de entrega.

Los escenarios que buscamos abordar son los siguientes:

Escenario 1: Un usuario paga con un medio de pago de acreditación inmediata y debe recibir una notificación como la que sigue:

Recibirá su producto XXX el día dd/mm/yyyy a las hh:MM.

Nota: los datos desplegados en este mensaje están contenidos en el envío.

Escenario 2: Un usuario paga con un medio de pago de acreditación diferida y debe recibir una notificación como como la que sigue:

Su producto será enviado una vez que abone el ticket Nro. NNNNNN

Nota: los datos desplegados en este mensaje están contenidos en el medio de pago.

La implementación de la solución deberá ser tal que cumpla con la especificación contenida en las siguientes pruebas:

```
public class UseCasesTest {
  @Test
  public void testUserBuysItemPayingWithOnlinePaymentAndItsShipped() {
    Purchase p = new Purchase(
        new Shipment( "Delivery date dd/mm/yyyy at hh:MM")  );
    Notification actual = p.pay(
        new OnlinePayment(1000 /* payment amount */)  );
    Notification expected = new Notification(
      "Delivery date dd/mm/yyyy at hh:MM"  );
    assertEquals(expected, actual);
  }
  @Test
  public void testUserBuysItemPayingWithDeferredPaymentAndItsShipped() {
    Purchase p = new Purchase(
        new Shipment( "Delivery date dd/mm/yyyy at hh:MM")  );
    Notification actual = p.pay(
        new DeferredPayment(1000 /* payment amount */)  );
```

```
    Notification expected = new Notification(
    "Your product will be shipped as soon as you pay ticket Nro.:
NNNNNN"
    );
    assertEquals(expected, actual);
  }
}
```

2.3 Introducción de Rival en el caso de estudio

La solución que proponemos, con la introducción del patrón Rival se basa en el diagrama de clases de la Fig. 1 y los de secuencia de las Fig. 2 y 3.

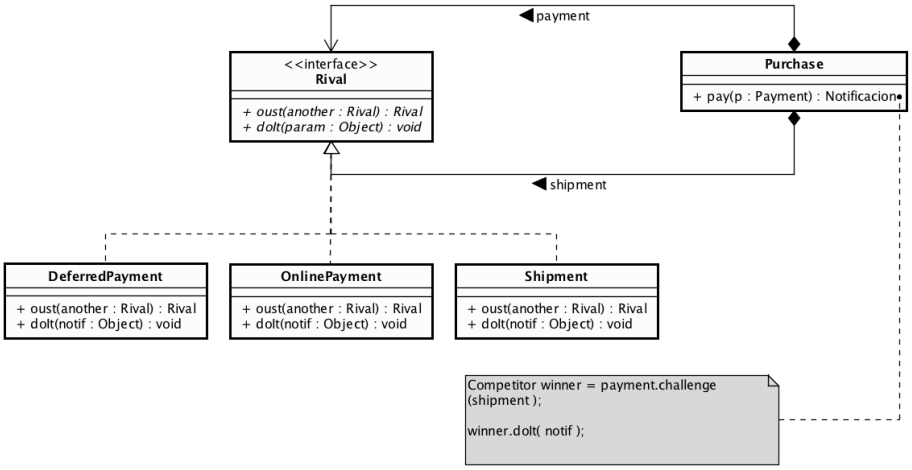

Fig. 1. Diagrama de clases del caso de estudio con aplicación del patrón Rival

En el escenario 1, *Purchase* representa la compra que realiza el usuario. El pago de acreditación inmediata es representado por *OnlinePayment* y el envío por *Shipment* (Fig. 1). Al invocar el mensaje *Purchase>>pay* debemos obtener una notificación que dependerá del envío y el pago en curso. Por esto el método *Purchase>>pay* invocará *OnlinePayment>>oust* pasando como parámetro la instancia de *Shipment:* esto provocará que *OnlinePayment* devuelva el mismo *Shipment* recibido como parámetro, ya que la notificación sólo dependerá de los datos del envío en cuestión, debido a que el pago ya fue ejecutado, por tanto se le puede enviar al cliente la promesa de entrega (Fig. 2).

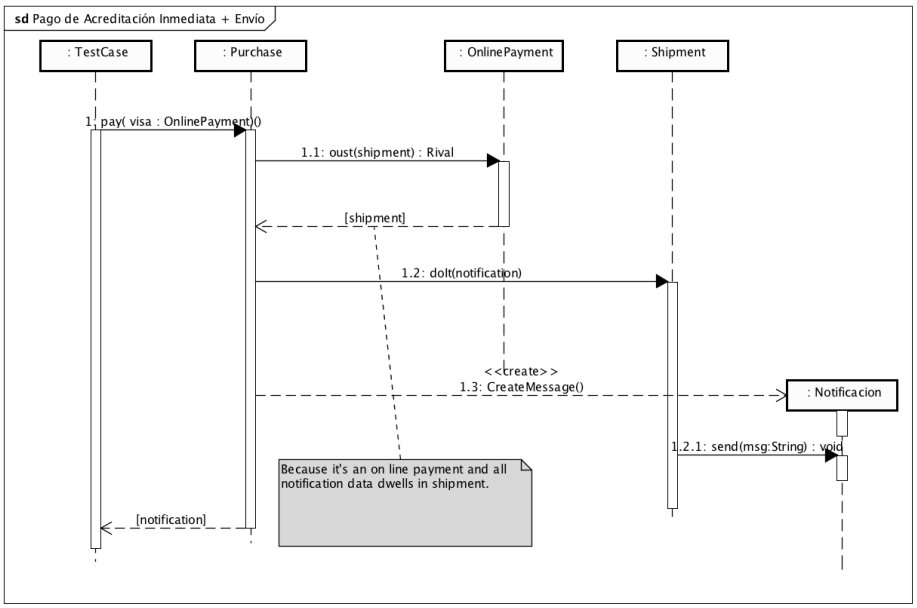

Fig. 2. Diagrama de secuencia del escenario 1 con la aplicación del patrón Rival

En el escenario 2, el *Purchase* representa la compra que realiza el usuario. El pago de acreditación diferida es representado por *DeferredPayment* y el envío por *Shipment*, tal como en el caso anterior (Fig. 1). Al invocar el mensaje *Purchase>>pay* debemos obtener una notificación que dependerá del envío y el pago en curso. Por esto el método *Purchase>>pay* invocará *DeferredPayment>>oust* pasando como parámetro la instancia de *Shipment*: esto provocará que *DeferredPayment* se devuelva a sí mismo, ignorando el *Shipment* enviado como parámetro, ya que la notificación no puede garantizar una promesa de entrega hasta tanto el cliente pague el ticket adeudado.

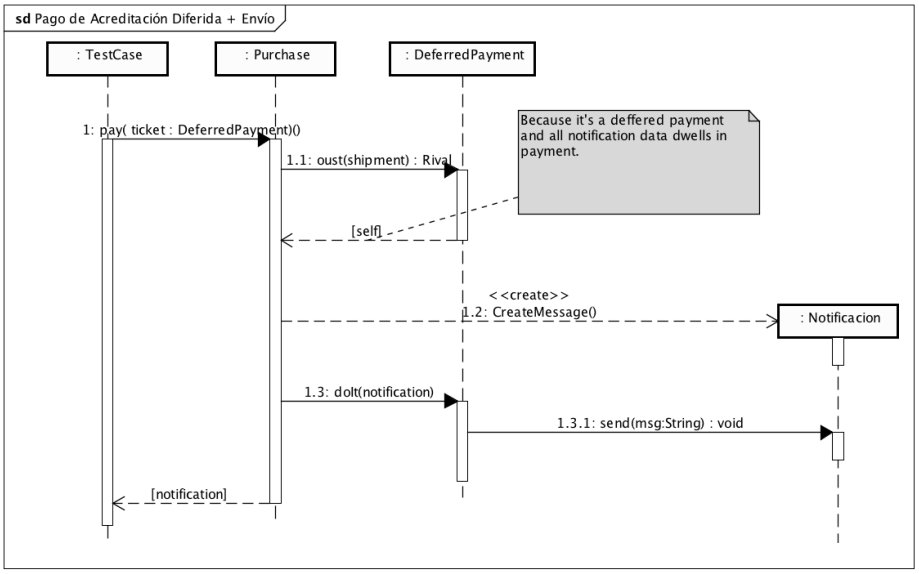

Fig. 3. Diagrama de secuencia del escenario 2 con la aplicación del patrón Rival

En ambos escenarios y dado un *Rival* que prevalece (interfaz implementada por *Shipment*, *OnlinePayment* o *DeferredPayment*) invocamos *Rival>>doIt(notification)* obteniendo la notificación con la información correspondiente, que en este caso dependerá exclusivamente de *Shipment* (Fig. 2 y Fig. 3).

A continuación, mostramos cómo sería la implementación en el caso sin patrón y con la introducción del patrón (el conjunto de pruebas es el mismo para ambas soluciones):

Con patrón	Sin Patrón

public class Purchase {
  private final Shipment shipment;
  public Purchase(Shipment shipment) {
  this.shipment = shipment;
  }
  public Notification pay(Rival payment) {
  Notification notification = new Notification();
  Rival winner =
  payment.oust(this.shipment);
  winner.doIt(notification);
  return notification;
  }
}
	public class Purchase {
	  private final Shipment shipment;
	  public Purchase(Shipment shipment) {
	  this.shipment = shipment;
	  }
	  public Notification pay(Payment payment) {
	  Worker worker = this.shipment;
	  Notification notification = new Notification();
	  if (!payment.isOnline()) {
	  worker = (Worker)payment;
	  }
	  worker.doIt(notification);
	  return notification;
	  }
	}

public class DeferredPayment
  implements Rival {
  private final Integer amount;
	public class DeferredPayment
	  implements Payment, Worker {
	  private final Integer amount;

public DeferredPayment(Integer amount) {
this.amount = amount;
}
@Override
public Rival oust(Rival another) {
return this;
}
@Override
public void doIt(Object param) {
// omited on purpose
}
}
	public DeferredPayment(Integer amount) {
	this.amount = amount;
	}
	@Override
	public boolean isOnline() {
	return false;
	}
	@Override
	public void doIt(Object param) {
	// omited on purpose
	}
	}

public class OnlinePayment implements Rival {
private final Integer amount;
public OnlinePayment(Integer amount) {
this.amount = amount;
}
@Override
public Rival oust(Rival another) {
return another;
}
public void doIt(Object notif) {
// do nothing at all
}
}
	public class OnlinePayment implements Payment,
	Worker {
	private final Integer amount;
	public OnlinePayment(Integer amount) {
	this.amount = amount;
	}
	@Override
	public boolean isOnline() {
	return true;
	}
	@Override
	public void doIt(Object notif) {
	// do nothing at all
	}
	}

public class Shipment implements Rival {
private final String msg;
public Shipment(String msg) {
this.msg = msg;
}
@Override
public Rival oust(Rival another) {
return this;
}
@Override
public void doIt(Object param) {
Notification notif = (Notification)param;
notif.send(this.msg);
}
}
	public class Shipment implements Worker {
	private final String msg;
	public Shipment(String msg) {
	this.msg = msg;
	}
	@Override
	public void doIt(Object param) {
	Notification notif = (Notification)param;
	notif.send(this.msg);
	}
	}

Payment.java: N/A	public interface Payment {
	boolean isOnline();
	}

Worker.java: N/A	public interface Worker {
	void doIt(Object param);
	}

public interface Rival {
Rival oust(Rival another);
void doIt(Object param);
}
	Rival.java: N/A

2.4 Resultados obtenidos

Como resultado, hemos logrado una solución que centraliza la responsabilidad y cumple otros principios de diseño. La lógica por la cual una instancia prevalece por sobre la otra (ya sea un medio de pago como un envío) está centralizada en aquellas clases que implementan la interfaz *Rival* y trae aparejado el cumplimiento del principio de diseño *Tell Don't Ask* [7] ya que *Purchase* no interpela a *OfflinePayment* o *DeferredPayment*, para conocer la naturaleza de los medios de pagos, con el fin de tomar partido en función del comportamiento observado. Ante un cambio de requerimiento donde la lógica de prevalecer no esté supeditada a la naturaleza de acreditación del pago implicará la implementación de esta funcionalidad en la clase que corresponda, ya sea *OfflinePayment*, *DeferredPayment*, *Shipment* o cualquier nuevo tipo de envío. No implementar este patrón violaría el principio de diseño *Open/Close* [8] ya que este cambio significará modificar *Purchase>>pay*, además de generar un alto acoplamiento [9] entre *Purchase* y los diferentes medios de pago.

2.5 Otro caso de estudio

Hay situaciones más complejas que ameritan el uso de este patrón. Veamos.

Dada una colección de elementos (*Biz*, *Qux*) buscamos reducirla [10] existiendo la posibilidad que el resultado sea ninguno de los elementos (*None*). Cada uno de estos objetos podrán ser seleccionados o no en función de su estado. Este problema tiene un correlato con un caso real pero omitimos los detalles por la complejidad intrínseca del problema. La simplificación antes propuesta es representativa a los efectos de capturar el requerimiento real.

Un posible conjunto de pruebas sería:

```
public class UseCasesTest {
    @Test
    public void testInconsistenciesOneHasToPrevail() {
        Rival[] arrayOfEntities = {
                new Biz(),
                new Qux()
        };
        EntityCollection ic = new EntityCollection(arrayOfEntities);
        Param actual = new Param();
        ic.winner().doIt(actual);
        assertEquals(new Param("I'm Biz"), actual);
    }
    @Test
    public void testInconsistenciesOneHasToPrevailRegardlessOrder() {
```

```
            Rival[] arrayOfEntities = {
                    new Qux(),
                    new Biz()
            };
            EntityCollection ic = new EntityCollection(arrayOfEntities);
            Param actual = new Param();
            ic.winner().doIt(actual);
            assertEquals(new Param("I'm Biz"), actual);
        }
    }
```

La implementación, con y sin patrón, se muestra abajo:

Con patrón	Sin Patrón

public class EntityCollection {
  List<Rival> entities;
  public EntityCollection(Rival ... entities) {
  this.entities = Arrays.asList(entities);
  }
  public Rival winner() {
  Rival result = new None();
  for(Rival i : this.entities) {
  result = result.oust(i);
  }
  return result;
  };
}
	import java.util.Arrays;
	import java.util.List;
	public class EntityCollection {
	  List<Entity> entities;
	  public EntityCollection(Entity... entities) {
	  this.entities = Arrays.asList(entities);
	  }
	  public Entity winner() {
	  Entity result = null;
	  for(Entity i : this.entities) {
	  if (i.happens()) {
	  result = i;
	  break;
	  }
	  }
	  if (result == null) {
	  result = new None();
	  }
	  return result;
	  };
	}

public class Biz implements Rival {
  @Override
  public Rival oust(Rival another) {
  return this;
  }
  @Override
  public void doIt(Object param) {
  Param p = (Param)param;
  p.setValue("I'm Biz");
  }
}
	public class Biz implements Entity, Worker {
	  @Override
	  public boolean happens() {
	  return true;
	  }
	  @Override
	  public void doIt(Object param) {
	  Param p = (Param)param;
	  p.setValue("I'm Biz");
	  }
	}

public class Qux implements Rival {
  @Override
  public Rival oust(Rival another) {
  return another;
  }
  @Override
  public void doIt(Object param) {
	public class Qux implements Entity, Worker {
	  @Override
	  public boolean happens() {
	  return false;
	  }
	  @Override
	  public void doIt(Object param) {

// Empty on purpose
}
}
	}
	}

None.java: Omitido Intencionalment.	None.java: Omitido Intencionalment.

public interface Rival {
Rival oust(Rival another);
void doIt(Object param);
}
	Rival.java: N/A

Worker.java: N/A	public interface Worker {
	void doIt(Object param);
	}

Entity.java: N/A	public interface Entity {
	boolean happens();
	}

Como resultado de implementar el patrón, logramos – además de las ventajas observadas en el caso de estudio 1 – un aumento de la cobertura.

En efecto, la implementación del patrón propone un mecanismo que por construcción favorece una alta cobertura del código debido a que minimiza la utilización de la cláusula *if*. Para observar las repercusiones de implementar el patrón exploramos la cobertura de *EntityCollection>>winner* en ambas soluciones ignorando los métodos que exceden la cuestión, y utilizando siempre el mismo conjunto de pruebas.

En ambos casos la cobertura total del proyecto tiene niveles aceptables, aunque esta métrica sea útil solo en el contexto de encontrar areas de código no exploradas por las pruebas [11].

Analizando la cobertura de las implementaciones con y sin patrón (Tablas 1 y 2, respectivamente):

Tabla 1. Solución sin patrón (SSP)

Element	Missed In-structions	Cov. %	Missed Branches	Cov. %	Cxty
Biz		100%		n/a	3
EntityCollection		88%		66%	5
None		0%		n/a	2
Param		100%		n/a	4
Qux		83%		n/a	3
Total	**10 of 83**	**87%**	**2 of 6**	**66%**	**17**

Tabla 2. Solución con patrón (SCP)

Element	Missed In-structions	Cov. %	Missed Branches	Cov. %	Cxty

Biz		100		n/a	3
EntityCollection		100		100	3
None		83%		n/a	3
Param		100%		n/a	4
Qux		83%		n/a	3
Total	**2 of 78**	**97%**	**0 of 2**	**100%**	**16**

Veamos algunos resultados:

- La cantidad de líneas de código necesarias para implementar la solución es mayor en SSP que en SCP: 83 y 78 respectivamente.
- La cobertura total es menor en la SSP (87%) que en la SCP (97%).
- La complejidad ciclomática es equivalente en ambas.
- La complejidad ciclomática de la clase *EntityCollection* es dispar, arrojando un valor de 5 en SSP y 3 SCP, lo cual merece un análisis especial.

En efecto, si analizamos la cobertura para la *EntityCollection*, observamos que los valores totales de cobertura son aceptables, si bien existe una diferencia de 12% a favor de SCP habiendo obtenido las siguientes mediciones: SSP 88% y SCP 100%. Ver tablas 3 y 4, respectivamente.

Tabla 3. Reporte de cobertura para *EntityCollection* sin patrón (SSP)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Cxty
EntityCollection (Entity[])		100%		n/a	1
winner()		85%		66%	4
Total	**4 of 35**	**88%**	**2 of 6**	**66%**	**5**

Tabla 4. Reporte de cobertura para *EntityCollection* con patrón (SCP)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Cxty
EntityCollection (Rival[])		100%		n/a	1
winner()		100%		100%	2
Total	**0 of 29**	**100%**	**0 of 2**	**100%**	**3**

Podemos inferir que el código problemático es el método *EntityCollection>>winner*. Analizando ambos códigos se observa que en SSP la cláusula *if* degrada la cobertura. En contraposición, la solución que utiliza el patrón arroja niveles de cobertura de un 100%.

3 Discusión, trabajos futuros y conclusiones

Este trabajo se ha centrado en presentar Rival, un patrón de diseño de uso organizacional que mejora el cumplimiento de principios de diseño generalmente aceptados y a la vez mejora la cobertura de código.

Cabe destacar que se intentaron implementar soluciones utilizando *Visitor* y *Chain-of-responsibility* (*CoR*) [14]. En el primer caso se observó la necesidad de modelar una clase accesoria (Visitor) produciendo Smell Shotgun Surgery. Luego si considero *CoR* y al proponer una composición entre los handler (Medio de Pago y Envio) se observó un incremento entre modelo y realidad modelada la cual propone una barrera en el entendimiento del problema.

En futuros trabajos habría que analizar más casos de estudio, además de los dos presentados en este artículo.

Otra posible línea de investigación, podría incluir el análisis de refactorizaciones que permitan introducir el patrón Rival en proyectos que no lo hayan tenido en cuenta en una primera versión, en el sentido del catálogo de patrones de Kerievsky [12].

Referencias

1. Cunningham, W., & Beck, K.: Using pattern languages for object-oriented programs. In Proceedings of OOPSLA (Vol. 87) (1987)
2. Gamma, E., Helm, R., Johnson, R., & Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Boston, Massachusetts: Addison-Wesley, 32 (1995)
3. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley (2002)
4. Cornett, S.: Code Coverage Analysis. Disponible en http://www.bullseye.com/coverage.html (visitado el 9 de abril de 2018)
5. Chilenski, J. J., & Miller, S. P.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, 9(5), (1994) 193-200
6. Parnas, D. L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), (1972) 1053-1058
7. Sharp, A.: Smalltalk By Example, McGraw-Hill (1997)
8. Meyer, B.: Object-Oriented Software Construction, Prentice Hall (1998)
9. ISO/IEC TR 19759:2005, Software Engineering — Guide to the Software Engineering Body of Knowledge (SWEBOK) (2005)
10. Hutton, G.: A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4), (1999) 355-372
11. Fowler, M.: TestCoverage. Disponible en https://martinfowler.com/bliki/TestCoverage.html (visitado el 9 de abril de 2018)
12. Kerievsky, J.: Refactoring to patterns. Addison-Wesley (2004)
13. Fowler, M., Beck, K., Brant J., Opdyke W., & Roberts d.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
14: Gamma E., Helm R., Johnson, R., & Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)