Agradecimientos:

A Sergio Gálvez Rojas por hacer posible la este tfg que ha permitido que vean la luz una serie de ideas con la esperanza de que tengan un mínimo de divulgación que permita crear discusiones para así conocer su viabilidad.

A Miguel Ángel Rodríguez Manzano por implicarse y ser el primero en añadir código al proyecto. Su código, que automatiza la generación de la plantilla basada en anotaciones, no ha podido llegar a tiempo para ser recogido en este tfg.

A Antonio César Gómez Lora por sus sugerencias que ayudaron a encaminar el diseño inicial, las ideas previas a este tfg.

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería de Computadores

Creación de una capa de persistencia abstracta en Java

Development of a persistence abstract layer in Java

Realizado por
Alberto Bellido de la Cruz
Tutorizado por
Sergio Gálvez Rojas
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Julio de 2015

Fecha defensa:
El Secretario del Tribunal

Resumen

El trabajo consiste en la implementación de una API (conjunto de métodos) que permite recopilar información individualizada de cada instancia. Esta información está relacionada con la persistencia de objetos por lo que los métodos de la API responden a preguntas relacionadas con la herencia y los atributos de la clase. Se puede preguntar de qué clase se hereda o cuáles son los atributos que se quieren hacer persistentes. Por cada atributo se recoge una serie de información como el tipo y su valor. Se da soporte a atributos no primitivos para manejar cualquier grado de composición, por ejemplo, la clase departamento tendrá un atributo que será una colección de empleados. Para los atributos no primitivos se recopila información diferente a la de los atributos primitivos, información como el tipo de colección.

La API funciona en los dos sentidos. Primero informa a la base de datos de cómo es la instancia para que pueda ser almacenada. Luego, la instancia debe ser leída y reconstruida. Para esto, la base de datos, a través de la API, coloca la información en los atributos de la instancia que corresponda.

De esta manera, el programador puede crear clases para que sean persistentes sin necesidad de conocer cómo se almacenará la información, no necesita saber nada relacionado con las base de datos relacionales ni SQL con lo que puede centrarse exclusivamente en el desarrollo de aplicaciones Java.

Este trabajo sólo ha creado la API que facilita la comunicación entre las instancias que deben ser persistentes y las bases de datos que las hacen persistentes. Aunque el objetivo de esta API también ha sido la de facilitar la creación de estas bases de datos. La curva de aprendizaje es pequeña debido a que sólo implica una clase y una serie de métodos.

Palabras claves

Java, JPA, JDO, mapeo objeto relacional, base de datos relacionales, ObjectP, BDO, persistencia, capa abstracta, reflexión, categorías, categorización, neurona aritificial, red neuronal artificial.

Abstract

The work involves the implementation of an API (set of methods) that can collect individualized information for each instance. This information is related to the persistence of objects so the API methods answer questions related to inheritance and class attributes. You may ask what kind it is inherited or are the attributes that you want to make persistent. For each attribute a series of information is collected, this information could be the type and the value. Objects support (not primitives) is given to handle any level of composition, for example, a department will have a collection of employees. For objects different information is collected, such as the type of collection.

The API works in both directions. First reports to the database how is the instance so it can be stored. Then, the instance must be read and reconstructed. For this, the database through the API, place the information on the attributes of the instance concerned.

For this, the database through the API, place the information on the attributes of the instance concerned. You not need to know anything about the relational database and SQL so that you can focus exclusively on the development of Java applications.

This work has just created the API that facilitates communication between the instances to be persistent and databases that make them persistent. Although the goal of this API has also been to facilitate the creation of databases. The learning curve is small because it only involves a class and a number of methods.

Keywords

Java, JPA, JDO, object relational mapping, relational database, ObjectP, BDO, persistence, abstract layer, reflection, categories, categorization, aritificial neuron, artificial neural network.

Índice

Capítulo 1. Introducción

1.1. Planteamientos

Actualmente la persistencia de objetos en lenguajes OO como Java o C++ suele llevarse a cabo mediante sistemas de mapeo Objeto-Relacional como Hibernate o Ibatis. El principal problema de esta aproximación es que no abstrae al desarrollador de los conceptos inherentes al modelo relacional que le permita centrarse exclusivamente en los aspectos de desarrollo y de persistencia pura. Abstraerse del modelo relacional permite ahorrar tiempo en los desarrollos a la vez que aumenta las capacidades del código creado.

Se parte de la experiencia adquirida tras la realización de un prototipo previo. Este prototipo se realizó a ciegas, sin un diseño inicial pues lo que se pretendía era experimentar, dejar que las distintas situaciones fueran las que escribiesen las especificaciones y obtener los conceptos básicos a utilizar en una siguiente aproximación. Así pues, basándose en ese prototipo se realizará una revisión completa para seleccionar y organizar las partes que se consideran más positivas. Para realizar esta selección se analizarán con detenimiento las soluciones disponibles actualmente para almacenar objetos Java en bases de datos, tomándose de cada tecnología los aspectos más ventajosos y adaptándolos en caso necesario.

1.2. Objetivos

En este proyecto se propone crear una metodología de persistencia que desacopla el CÓMO se almacenan los objetos en un sistema de persistencia particular (para el que utilizaremos el término genérico BDO – Base de Datos de Objetos) del QUÉ se quiere hacer persistente. En particular, el objetivo del proyecto se centra en dotar a cada instancia a almacenar, y de manera automática, de un conjunto de funciones, o API, que permita a cualquier BDO conocer qué

elementos de la instancia deben guardarse y sus relaciones con otras instancias. A este conjunto de funciones se le llamará ObjectP.

No se entrará en detalles de cómo se implementa una BDO. ObjectP ofrece en su API los métodos necesarios para que cualquier BDO recopile toda la información necesaria. Las pruebas, por tanto, consistirán en comprobar que las estructuras de ObjectP adquieren los valores que le correspondan en relación a la instancia que esté representando.

1.3. Estructura del trabajo

Este documento consta de ocho capítulos y un anexo con los siguientes contenidos:

2. **Tecnologías actuales de persistencia**: Se analizan herramientas actuales de persistencia para poder realizar una comparativa con ObjectP.
3. **Utilización de ObjectP**: Se muestra la interfaz de ObjectP y la manera de usarla.
4. **Ejemplos de clases ObjectP**: Los ejemplos recogen una gran variedad de casos para que resulte sencillo adaptarlos a la mayoría de las necesidades que tiene un programador.
5. **Interioridades de ObjectP**: Se describe con detalle los distintos elementos de ObjectP.
6. **Pruebas**: Se comprueba que el flujo de datos entre las instancias del desarrollador y las estructuras de ObjectP es el correcto.
7. **Resultados y discusión**: Se analiza el trabajo realizado destacando sus elementos positivos y negativos.
8. **Conclusiones y trabajo futuro**.

Anexo – Sugerencia para una BDO: No forma parte de esta documentación abordar la creación de BDOs pero a la mentalidad predominante actualmente le resulta muy complicado separar el QUÉ del CÓMO. Cualquier cuestión que esté relacionada con la gestión del almacenamiento sólo se abordará en este anexo. Es bastante probable que el lector se sienta más cómodo mirando este anexo antes y durante la lectura del trabajo.

Capítulo 2. Tecnologías actuales de persistencia

Este capítulo analiza tecnologías actuales que suministran persistencia de objetos para disponer de una base con la que comparar ObjectP. Las tecnologías que interesan son las que persiguen recoger el QUÉ se debe almacenar de cada instancia. Quedan en un segundo plano las tecnologías que abordan el CÓMO se almacenan las instancias.

En la práctica esta separación resulta muy complicada. Incluso ObjectP no adquiere todo su significado o es percibido de otra manera cuando se describe junto con alguna BDO, como la sugerida en el anexo. La realidad sugiere que una tecnología para el QUÉ necesita de una para el CÓMO que la explote al máximo.

2.1. Tecnologías para el QUÉ

Se describen dos tecnologías, JPA y JDO. Además, se realiza una introducción a ObjectP para que resulte sencillo de comparar.

2.1.1. Java Persistence API (JPA)

Las bases de datos relacionales son el repositorio más popular actualmente. A su vez, lenguajes orientados a objetos como Java también son muy populares. Esto conlleva que se busquen mecanismos que permitan usarlos en combinación. La propuesta más usada es JPA. Por tanto, se analizará esta tecnología buscando identificar las características que más relacionadas estén con el objetivo de este trabajo, la abstracción que pueda dar al desarrollador a la hora de escribir sus clases.

- **Clave primaria**: JPA permite que la clave primaria sea defina por el usuario. Puesto que la clave primaria es un elemento propio del repositorio, implicar al desarrollador en su definición le impide abstraerse del modelo relacional.
- **Relaciones entre tablas**: Le es solicitada información al desarrollador sobre la manera de interconectar las de tablas. Debe decir si quiere 'ManyToOne' por ejemplo. Al igual que antes, el desarrollador se ve obligado a implicarse en el diseño del modelo relacional.
- **Herencia**: También se le pide al desarrollador que seleccione uno de los varios mecanismos para representar la herencia en las tablas.
- **Colecciones**: Una clase puede estar compuesta de otras clases como un Departamento que tiene Empleados. JPA obliga a especificar de qué clases son las instancias de una colección. Esto es similar a usar en Java 'List<Empleado>' en vez de simplemente 'List'. Se debe considerar una restricción importante ya que las clases Java no tienen esta limitación.

Una tecnología para el QUÉ transparente no debería requerir conocimientos por parte del desarrollador del repositorio final. Debería ser suficiente con conocer el paradigma de la orientación a objetos.

JPA tiene la ventaja de proporcionar mucha eficiencia al permitir que se especifiquen toda clase de detalles. Involucrar al desarrollador en estos detalles, que son propios del modelo relacional, le impide centrarse en los objetos lo que puede dificultarle la creación de sistemas conceptualmente complejos.

2.1.2. Java Data Object (JDO)

JDO es parecido a JPA en lo que a sus anotaciones se refiere, se podría decir que es como un superconjnto de JPA aunque son proyectos independientes. Pero JDO busca añadir la transparencia que JPA no tiene lo que significa que se acerca a las ideas de ObjectP. Al disponer de la transparencia que JPA no tiene puede ser usado por repositorios que no son base de datos relacionales. JDO será descrito con más detenimiento debido a que se acerca mucho al objetivo de este trabajo.

La figura 2.1 muestra un resumen de los paquetes de JDO.

Packages	
javax.jdo	This package contains the JDO specification interfaces and classes.
javax.jdo.annotations	
javax.jdo.datastore	This package contains the JDO specification datastore interfaces.
javax.jdo.identity	This package contains the JDO specification identity interfaces and classes.
javax.jdo.listener	This package contains the JDO specification listener interfaces and classes.
javax.jdo.metadata	This package contains classes representing the different components of the JDO Metadata.
javax.jdo.spi	This package contains the interfaces and classes used by JDO implementations.

Fig. 2.1 Paquetes de JDO

La figura 2.2 muestra cómo se usa JDO: Se crea una instancia y luego la guarda llamando a un método. Pero esto se puede tener también en JPA, lo importante es cómo esté hecho 'Employee'.

```
PersistenceManager pm = PMF.get().getPersistenceManager();

Employee e = new Employee("Alfred", "Smith", new Date());

try {
    pm.makePersistent(e);
} finally {
    pm.close();
}
```

Fig. 2.2 Guardando una instancia con JDO

La figura 2.3 muestra cómo es el código de 'Employee'. Se observa que el objetivo de lograr transparencia es conseguido mucho mejor con JDO que con JPA.

```
@PersistenceCapable
public class Employee {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private ContactInfo contactInfo;

    ContactInfo getContactInfo() {
        return contactInfo;
    }
    void setContactInfo(ContactInfo contactInfo) {
        this.contactInfo = contactInfo;
    }
```

Fig. 2.3 Guardando una instancia con JDO

Se describen a continuación las características más relevantes de JDO.

- **Anotación 'PersistenceCapable'**: Con la anotación 'PersistenceCapable' se dice que la clase es JDO.

```
import javax.jdo.annotations.PersistenceCapable;

@PersistenceCapable
public class Employee {
    // ...
}
```

Fig. 2.4 PersistenceCapable indica que las instancias son manejables por JDO

- **Anotación 'Persistent'**: Los atributos que deben hacerse persistentes deben ser marcados con la anotación 'Persistent'.

```
import java.util.Date;
import javax.jdo.annotations.Persistent;

// ...
    @Persistent
    private Date hireDate;
```

Fig. 2.5 Con @Persistent se indica que el atributo debe ser almacenado

- **Clave primaria**: Se obliga a definir un atributo que haga de clave primaria. Aquí sucede como JPA, no se debería pedir. La figura 2.6 muestra un ejemplo.

```
import com.google.appengine.api.datastore.Key;

import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PrimaryKey;

// ...
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;
```

Fig. 2.6 Se obliga a especificar las claves primarias

- **Colecciones**: Una clase puede estar compuesta de otras clases como un Departamento que tiene Empleados. Cada clase tendrá su tabla y es necesario conectarlas de alguna manera para indicar a qué departamento pertenece cada empleado. JDO, a diferencia de JPA, no necesita que se indique la manera en la que conectará las diferentes tablas. Dependiendo de cómo sea la relación de composición, JDO opta por usar un mecanismo de forma automática. En cambio, mantiene la necesidad de especificar el tipo de elementos que almacenará la colección (List<Empleado>).

- **Herencia**: El manejo de la herencia es uno de los puntos débiles de JDO ya que hace referencia a cómo DataNucleous, un repositorio, maneja la herencia. Parece poco elegante que sea la clase raíz la que indique que otras clases pueden heredar de ella.

Worker.java

```
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Inheritance;
import javax.jdo.annotations.InheritanceStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable
@Inheritance(strategy = InheritanceStrategy.SUBCLASS_TABLE)
public abstract class Worker {
```

Fig. 2.7 La superclase debe indicar que otras clases pueden heredar de ella

- **Interfaz 'PersistenceManager':** La definición de JDO no se centra exclusivamente en facilitar la persistencia a las instancias del desarrollador. JDO también define la interfaz con la que se debe interactuar con el repositorio; la figura 2.2 mostraba un ejemplo.
- **Clase 'JDOImplHelper'**: Es un elemento de JDO interesante por su parecido con ObjectP. Es una clase que cumple funciones similares a ObjectP, la de registrar metadatos para no tener que usar reflexión.
- **Método 'jdoNewInstance'**: Es un método interesante de JDO que tiene su análogo en ObjectP y se usan para facilitar la creación de instancias. Con 'new' se obliga a especificar exactamente la clase que se quiere mientras que este mecanismo permite solicitar una instancia 'que sea de la misma clase que esta otra' sin necesidad de saber de qué clase es. Permite crear código más genérico para trabajar con diferentes clases.

JDO alcanza un nivel de transparencia alto pero no termina de separarse de la implementación. La clave primaria, el manejo de la herencia y las colecciones son detalles que necesitaría terminar de separar. También, el hecho de que JDO disponga de 'PersistenceManager' se puede interpretar como una forma de limitar el uso que una BDO puede hacer de JDO, se le obliga a implementar unos métodos y no otros.

2.1.3. Introducción a ObjectP

ObjectP significa 'Persistence Object' como una forma de indicar que es un objeto que facilita la persistencia. La introducción a ObjectP se inicia con una clase que podría escribir cualquier programador para añadirle a continuación el resto de elementos involucrados.

- La clase de la figura 2.8 es la clase escrita por el programador. Esta clase podría ser 'Empleado', 'Departamento' o cualquier otra.

```
public class Empleado{

        String nombre = "";
        :     :     :     :     :

}//END CLASS Empleado
```

Fig. 2.8 Clase escrita por el programador

- La clase debe heredar de ObjectP pues es donde están las estructuras y los métodos que permiten recoger toda la información necesaria. Estas estructuras persiguen un fin muy parecido a 'JDOImplHelper' de JDO. ObjectP busca simplificar tanto el trabajo que debe realizar el desarrollador de 'Empleado' como el desarrollador de la BDO.

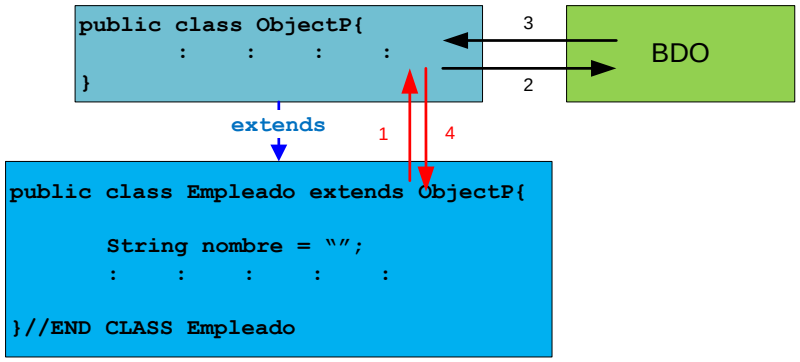

Fig. 2.9 La clase escrita por el programador debe hereda de ObjectP

- La información relevante para la persistencia llega a la BDO en dos pasos:
    - El primer paso consiste en llevar la información desde la instancia a ObjectP. Esta comunicación está representada por la flecha número 1 en la fig. 2.10.
    - El segundo paso consiste en llevar la información desde ObjectP a la BDO. La flecha número 2 representa este paso.

- Las flechas 3 y 4 representan el proceso inverso, la lectura desde la BDO para reconstruir las instancias almacenadas.

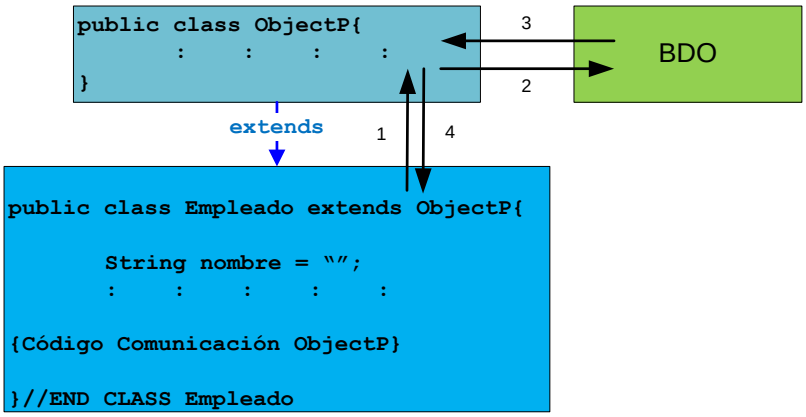

Fig. 2.10 Código de comunicación entre la clase del programador y ObjectP

- La comunicación entre la clase 'Empleado' y 'ObjectP' se realiza mediante un código que se coloca en la clase 'Empleado'. Este código y la manera de generarlo es analizado más adelante.
- En la figura 2.11, las flechas 5 y 6 representan una comunicación ajena a ObjectP, es la comunicación habitual entre una clase principal y las instancias creadas.

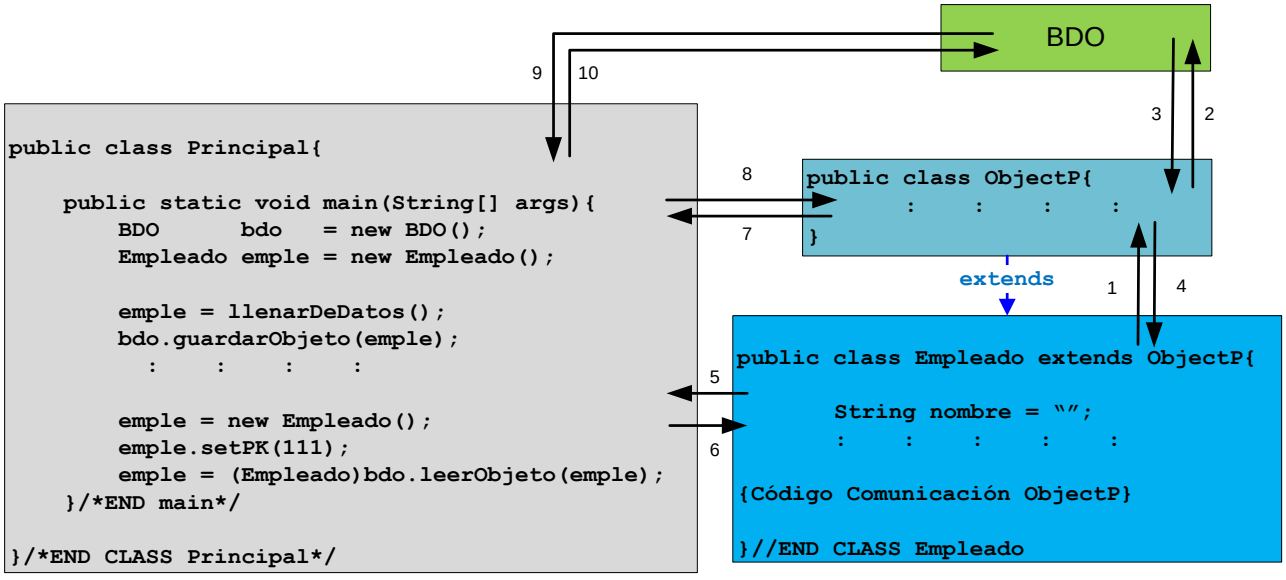

Fig. 2.11 Todos los elementos implicados en un entorno de persistencia basado en ObjectP

- Las flechas 7 y 8 representan una comunicación avanzada. Cuando se conoce con detalle ObjectP se puede optar por crear instancias directamente de ObjectP y usarlas sin que

estén asociadas a una clase. Esta es una de las diferencias con respecto a JDO aunque es una diferencia que no está relacionada con la persistencia.

- La comunicación representada por las flechas 9 y 10 es la habitual entre el código del desarrollador y una base de datos, es similar al de la figura 2.2.

A continuación se analizan los mismos puntos analizados para JPA y JDO.

- **Clave primaria**: No es un concepto que exista en los programas orientados a objetos, por tanto, no hay algo que ObjectP pueda recoger.
- **Identificadores y restricciones**: No existe clave primaria pero sí se da la opción de marcar los atributos como identificador o asignarle restricciones. Los identificadores no buscan ser la clave primaria ya que es algo opcional, es una ayuda para localizar la información. Las restricciones son condiciones que, aunque inspiradas en SQL, sí se pueden considerar como requisitos de la clase programada. Por ejemplo, 'REQUIRED' ('NOT_NULL') es la idea de que la instancia no es válida si el atributo no tiene un valor. Además, al igual que se puede especificar ahora restricciones como 'REQUIRED' Y 'UNIQUE', sería de utilidad poder indicar que los valores válidos son los que se encuentran dentro de un intervalo. Esto ahora se hace por medio de métodos de la clase pero se quiere que en un futuro ObjectP disponga de un abanico de restricciones más amplio.
- **Relaciones entre tablas**: Las relaciones entre tablas se delega en la BDO.
- **Herencia**: ObjectP se limita a recoger la información de la herencia entre clases. Cada BDO realizará la representación que considere oportuno.
- **Colecciones**: Se recoge la información tal y como está en la instancia. Si se dice 'List<Empleado>' ObjectP controla que los elementos de la colección sean de la clase 'Empleado' o de una que herede de ella. Si la colección simplemente es 'List' se asume que es 'List<ObjectP>' lo que permite almacenar cualquier cosa.

Descripción de los elementos análogos de JDO en ObjectP:

- **Anotación 'PersistenceCapable' vs Heredar de ObjectP**: Este concepto de JDO, el de indicar que la clase puede ser manejada por JDO usando una anotación, en ObjectP se realiza cuando se hereda de ObjectP.
- **Anotaciones para la persistencia**: Las flechas 1 y 4 de la figura 2.10 representa la comunicación entre ObjectP y la instancia. Esta comunicación se realiza mediante un

código en la clase escrita por el programador. Si este código lo escribe el programador no son necesarias anotaciones pero para crear este código de manera automatizada está previsto usar un conjunto de anotaciones similar al de JDO.

- **Interfaz 'PersistenceManager':** La definición de JDO define la interfaz con la que se debe interactuar con el repositorio; la figura 2.2 muestra un ejemplo. ObjectP deja en manos de la BDO la creación de esta interfaz.
- **Clase 'JDOImplHelper'**: Este elemento de JDO es ObjectP. Una manera de tener la información necesaria ya recogida para no tenerla que recopilar en tiempo de ejecución.
- **Creación de instancias**: ObjectP tiene un método análogo a 'jdoNewInstance' para crear instancias: 'objpCrearObjecto'.

Al delegar ObjectP tareas en la BDO la implementación de ésta última se puede tornar más compleja. Puesto que JPA y JDO describen la manera en la que la información se hace persistente es de esperar que el lector tenga curiosidad por saber cómo lo haría una BDO para ObjectP. Es por este motivo por el que se ha redactado una sugerencia de BDO que está al final de esta documentación en un anexo.

2.2. Comparativa entre las herramientas de persistencia

2.2.1. Características usadas para la comparativa

Las características usadas para la comparativa serán:

- **Transparencia**: Habrá transparencia si la herramienta sólo pide información sobre qué elementos quiere guardar pero no necesita conocer cómo deben guardarse.

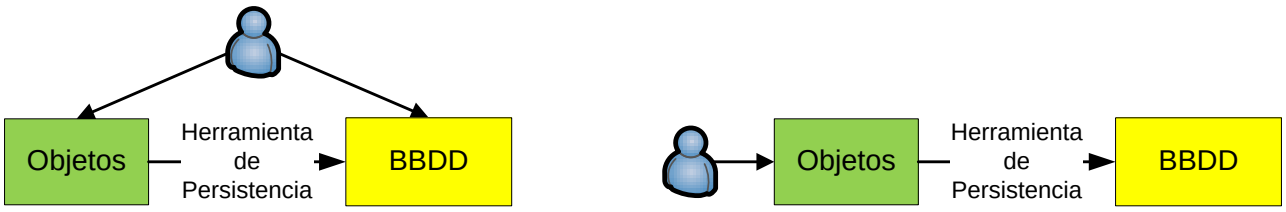

Fig. 2.12 El usuario de la izquierda conoce ambas representaciones, el de la derecha sólo los objetos.

- **Colecciones 'Raw Type'**: En memoria se puede tener una colección con referencias a cualquier objeto. En vez de tener 'List<Empleado>' se tiene simplemente 'List'.

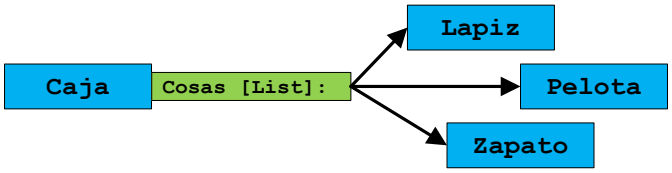

Fig. 2.13 No todas las herramientas de persistencia permiten colecciones heterogéneas

- **Dinamismo de instancia**: Permite modificar el modelo por defecto de la clase. La clase 'Caja' puede tener dos colecciones definidas pero en tiempo de ejecución se le podrían añadir otras a las instancias.

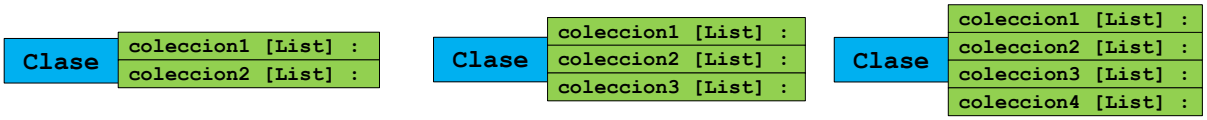

Fig. 2.14 El dinamismo consiste en modificar la estructura por defecto de la clase.

2.2.2. Comparativa

La tabla 2.1 muestra un resumen comparativo de las distintas características de los modelos más relevantes.

	JPA	JDO	ObjectP
SQL	SI	SI	NO
Transparente	NO	SI	SI
Colecciones Raw Type	NO	NO	SI
Dinamismo	NO	NO	SI

Tabla. 2.1 Tabla comparativa entre los distintos modelos

JDO tiene el mismo objetivo que ObjectP, el de abstraer al desarrollador entre el QUÉ y CÓMO. Esto hace que para ciertos desarrollos ObjectP pueda carecer de interés pero las características únicas de ObjectP junto con que JDO no termina de ser 100% transparente permiten que ObjectP deba ser considerado aún una opción.

Capítulo 3. Utilización de ObjectP

Este capítulo introduce los distintos elementos de ObjectP desde el punto de vista del desarrollador. Explica cómo usar ObjectP sin entrar en detalles de su implementación. Este uso puede servir tanto para el desarrollador que escribe la clase 'Empleado' como para el que crea la BDO ya que la API a usar es común para los dos casos.

3.1. Creación de clases

Se empieza escribiendo una clase como puede ser 'Empleado.java' y lo hacemos libremente, sin pensar en que luego debe adaptarse a ObjectP. Una vez escrita, la integración se realiza heredando de ObjectP que se puede decir que es la única gran restricción. Próximas mejoras buscarán fusionar ObjectP con java.lang.Object.

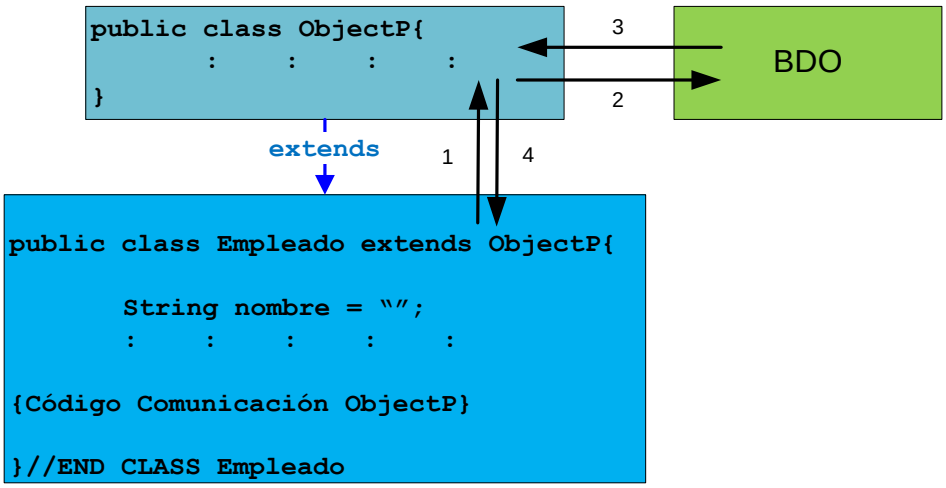

Fig. 3.1 El desarrollador escribe su clase y luego extiende de ObjectP

El siguiente paso a dar después de haber añadido '**`extends ObjectP`**' es añadir un trozo de código y personalizarlo para nuestra nueva clase. El objetivo de este código es permitir una comunicación entre ObjectP y nuestra clase de manera que ObjectP pueda atender las solicitudes de la BDO.

3.1.1. Comunicación entre ObjectP y la clase del desarrollador

Los trozos del código coloreados son los que se deben personalizar.

```
//-----------------------------------------------------------------------------
//BEGIN Plantilla ObjectP

    private final static String PROPIO_NOMBRE_CLASE = "Departamento";

    private final static ObjectP[] PROPIO_SUPER_REPRESENTANTE      =
                                                        {ObjectP.bdoCrearObjetoStatic() };

    //Atributos
    private final static String[]   PROPIO_ATRIBUTO_NOMBRE         = {"nombre "    ,
                                                                      "ubicacion" };

    private final static Boolean[]  PROPIO_ATRIBUTO_ES_ID           = {true ,
                                                                       false };

    private final static Integer[]  PROPIO_ATRIBUTO_TIPO           = {ObjectP.TIPO_DATO_STRING ,
                                                                      ObjectP.TIPO_DATO_STRING };

    private final static Boolean[]  PROPIO_ATRIBUTO_ES_REQUIRED    = {false ,
                                                                       false };

    private final static Boolean[]  PROPIO_ATRIBUTO_ES_UNIQUE      = {false ,
                                                                       false };

    private final static Boolean[]  PROPIO_ATRIBUTO_ES_INDICE      = {false ,
                                                                       false };

    private final static String[]   PROPIO_ATRIBUTO_DEFAULT_VALUE  = {" " ,
                                                                      " " };

    //Agregados
    private final static String[]     PROPIO_AGREGADO_NOMBRE =
                                                        {"jefe"       ,
                                                         "empleados" };

    private final static Integer[]    PROPIO_AGREGADO_COLECCION =
                                                        {ObjectP.AGREGADO_COLECCION_OBJECT    ,
                                                         ObjectP.AGREGADO_COLECCION_ARRAYLIST}

    private final static Integer[][]  PROPIO_AGREGADO_DIMENSIONES  =
                                                        { new Integer[0] ,
                                                          new Integer[0] };

    private final static ObjectP[]    PROPIO_AGREGADO_REPRESENTANTE_CLAVE =
                                                        {null ,
                                                         null };

    private final static ObjectP[]    PROPIO_AGREGADO_REPRESENTANTE_VALOR =
                                                        {Empleado.bdoCrearObjetoStatic() ,
                                                         Empleado.bdoCrearObjetoStatic() };


//METODOS para la Clase ------------------------------------------------------
    public Departamento (){
    }

    public static   bdoCrearObjetoStatic(){
    /*VAR*/
        ObjectP nuevo_objeto = null;
    /*BEGIN*/
        nuevo_objeto = new Departamento ();
        return nuevo_objeto;
    }
    @Override
```

```
    public ObjectP bdoCrearObjeto(){
        return Departamento.bdoCrearObjetoStatic();
    }


    public static String bdoNombreDeLaClaseStatic(){
    /*BEGIN*/
        return PROPIO_NOMBRE_CLASE;
    }


//METODOS para la Sincronizacion ------------------------------------------------

    @Override
    public void bdoAtributoSincrUserToObject(){
    /*BEGIN*/
        super.bdoAtributoSincrUserToObject();
        this.bdoAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 0], this.nombre     )
        this.bdoAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 1], this.ubicacion  )
    }


    @Override
    public void bdoAtributoSincrObjectToUser(){
    /*BEGIN*/
        super.bdoAtributoSincrObjectToUser();
        this.nombre     = (String)  this.bdoAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                              PROPIO_ATRIBUTO_NOMBRE[ 0] );
        this.ubicacion  = (String)  this.bdoAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                              PROPIO_ATRIBUTO_NOMBRE[ 1] );
    }


    @Override
    public void bdoAgregadoSincrUserToObject(){
    /*BEGIN*/
        super.bdoAgregadoSincrUserToObject();
        this.bdoAgregadoLimpiar (PROPIO_NOMBRE_CLASE, PROPIO_AGREGADO_NOMBRE[0]);
        this.bdoAgregadoLimpiar (PROPIO_NOMBRE_CLASE, PROPIO_AGREGADO_NOMBRE[1]);

        this.bdoAgregadoSetAgregadito  ( PROPIO_NOMBRE_CLASE, PROPIO_AGREGADO_NOMBRE[0],
                                         "0", this.jefe);
        this.bdoAgregadoSetAgregaditos ( PROPIO_NOMBRE_CLASE, PROPIO_AGREGADO_NOMBRE[1],
                                         this.empleados      );
    }


    @Override
    public void bdoAgregadoSincrObjectToUser(){
    /*BEGIN*/
        super.bdoAgregadoSincrObjectToUser();
        this.jefe      = (Empleado)            this.bdoAgregadoGetAgregadito      (
                                                          PROPIO_NOMBRE_CLASE,
                                                          PROPIO_AGREGADO_NOMBRE[0], "0" );
        this.empleados = (ArrayList<Empleado>) this.bdoAgregadoGetAgregaditosArrayList (
                                                          PROPIO_NOMBRE_CLASE,
                                                          PROPIO_AGREGADO_NOMBRE[1] );
    }


//Bloque inicializador ----------------------------------------------------------
    {
    /*VAR*/
        List conf_atributos = new ArrayList();
        int  i              = 0;
    /*BEGIN*/
        //Nombre de la Clase
        this.bdoNombreDeLaClase(PROPIO_NOMBRE_CLASE);


        //Herencia
        for (i=0; i<PROPIO_SUPER_REPRESENTANTE.length; i++){
            conf_atributos.clear();
            conf_atributos.add(0, PROPIO_SUPER_REPRESENTANTE[i] );
            this.bdoSupersConfiguracion(conf_atributos);
        }/*for*/



        //Atributos
        for (i=0; i<PROPIO_ATRIBUTO_NOMBRE.length; i++){
```

```
            conf_atributos.clear();
            conf_atributos.add(0, PROPIO_ATRIBUTO_NOMBRE        [i] );
            conf_atributos.add(1, PROPIO_NOMBRE_CLASE               );
            conf_atributos.add(2, PROPIO_ATRIBUTO_ES_ID         [i] );
            conf_atributos.add(3, PROPIO_ATRIBUTO_TIPO          [i] );
            conf_atributos.add(4, PROPIO_ATRIBUTO_ES_REQUIRED   [i] );
            conf_atributos.add(5, PROPIO_ATRIBUTO_ES_UNIQUE     [i] );
            conf_atributos.add(6, PROPIO_ATRIBUTO_ES_INDICE     [i] );
            conf_atributos.add(7, PROPIO_ATRIBUTO_DEFAULT_VALUE [i] );
            this.objpAtributoConfiguracion(conf_atributos);
        }/*for*/

        //Agregados
        for (i=0; i<PROPIO_AGREGADO_NOMBRE.length; i++){
            conf_atributos.clear();
            conf_atributos.add(0, PROPIO_AGREGADO_NOMBRE              [i] );
            conf_atributos.add(1, PROPIO_NOMBRE_CLASE  );
            conf_atributos.add(2, PROPIO_AGREGADO_COLECCION           [i] );
            conf_atributos.add(3, PROPIO_AGREGADO_REPRESENTANTE_CLAVE [i] );
            conf_atributos.add(4, PROPIO_AGREGADO_REPRESENTANTE_VALOR [i] );
            conf_atributos.add(5, PROPIO_AGREGADO_DIMENSION           [i] );
            this.objpAgregadoConfiguracion(conf_atributos);
        }/*for*/
    }/*END*/
//-----------------------------------------------------------------------------
//END Plantilla
//-----------------------------------------------------------------------------
```

Los elementos que se deben configurar son:

- **Nombre de la clase.**
- **Clases de las que hereda.** Java no admite herencia múltiple pero ObjectP se ha diseñado como un caso general, por eso la estructura es un array.
- **Información de los atributos.** Los atributos son los tipos primitivos y para ellos se recoge la siguiente información:
    - Nombre: El nombre de la variable usada en el código.
    - ID: No busca ser la clave primaria para interconectar objetos en la BDO. Es necesario para impedir duplicidades al guardar. Si se guarda un dpto. con un empleado y luego otro dpto. con el mismo empleado, ese empleado no debe duplicarse.
    - Tipo: Para informar si es String, Integer, etc. Se usa 'Integer' en vez de 'int'.
    - Resto de elementos: Forman parte de un conjunto básico de restricciones pendiente de ampliar en futuras implementaciones de ObjectP.
- **Información de los agregados**: Los agregados son cualquier elemento que no es un tipo primitivo. Para un 'Departamento' los empleados es una colección de agregados.
    - Nombre: El nombre de la variable usada en el código.
    - Tipo Colección: Los agregados son variables no primitivas. O es un 'Object' o es una colección HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap y Object[].
    - Dimensiones: Cuando la colección es un array se debe informar de sus dimensiones. No es necesario hacerlo al inicializar, se puede hacer justo antes de sincronizar.

- Representante Agregados: Se usan para indicar qué tipo de objetos se pueden almacenar. Es análogo List<String>. Podrán almacenarse objetos de la clase del representante o que hereden de ella. Se pondrá ObjectP si se quiere que sea cualquier objeto.

- **Métodos para crear instancias**:
  - Constructor sin parámetros: Necesario para que otros métodos puedan funcionar. Es una restricción de ObjectP que el constructor sin parámetros esté en la plantilla y vacío. El usuario tendrá que crear constructores con parámetros.
  - bdoCrearObjetoStatic. Creación de instancias.
  - bdoCrearObjeto: Creación de instancias.
  - bdoNombreDeLaClaseStatic: Para conocer el nombre de la clase.
- **Métodos para la sincronización**:
  - bdoAtributoSincrUserToObject: Para colocar la información de los atributos en las estructuras de ObjectP.
  - bdoAtributoSincrObjectToUser: Para colocar la información que hay en las estructuras de ObjectP en los atributos de la instancia.
  - bdoAgregadoSincrUserToObject: Para colocar la información de los agregados en las estructuras de ObjectP.
  - bdoAgregadoSincrObjectToUser: Para colocar la información que hay en las estructuras de ObjectP en los agregados de la instancia.

3.1.2. Limitaciones de ObjectP

Hay una serie de limitaciones que están actualmente presentes:

- Ahora mismo ObjectP sólo acepta los tipos String, Integer, Character y Boolean para los atributos.
- Los agregados deben ser o bien un ObjectP o alguna de las colecciones siguientes: HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap y Object[]. Otro tipo de colección actualmente no es soportada.
- Se debe heredar de la clase ObjectP mientras no venga integrado con el lenguaje de programación.
- Construir la plantilla a mano si no se dispone de algún mecanismo de automatización.

3.2. Sincronización

La sincronización consiste en copiar los valores de los atributos de la clase del desarrollador en las estructuras de ObjectP (y viceversa). Si se tratase de C++ este apartado carecería de sentido pues en la configuración inicial habría punteros en ObjectP apuntando a las variables de la clase.

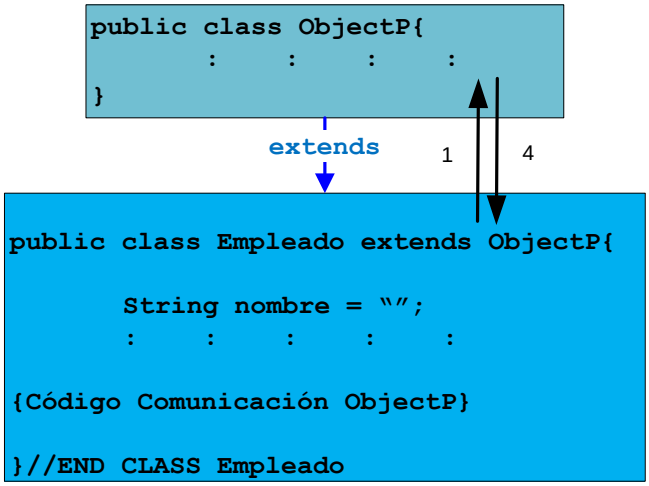

Fig. 3.2 La sincronización copia los valores de las variables.

La sincronización es un proceso que debe desencadenar la BDO cuando sea necesario. El desarrollador sólo debe preocuparse de que los métodos de sincronización de la plantilla estén creados. La sincronización se da en ambos sentidos para que la BDO pueda guardar las instancias en disco y también las pueda devolver después de leerlas de disco.

- **UserToObject**: Carga la información desde la instancia a ObjectP.
- **ObjectToUser**: Copia en los atributos de la instancia la información que haya en ObjectP.

Los ejemplos de las clases detallarán cómo es el código de los métodos de sincronización.

3.3. Agregados

Los agregados tienen su complejidad debido a la amplia variedad de casos con los que nos podemos encontrar. Los casos soportados son: ObjectP, HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap y Object[].

Recursivamente pueden contener cualquier combinación. Es decir, los elementos de un TreeSet pueden ser HashMap (sin límite en profundidad). Pero se sugiere que el contenido de las colecciones sea un ObjectP que luego tenga otro nivel de colecciones.

Aquí se introducen los conceptos, los ejemplos se verán en el capítulo correspondiente.

3.3.1. Adaptación de los agregados

El código de la sección 3.1.1 tiene un ejemplo con dos agregados. En la mayoría de los desarrollos habrá variables con cierto grado de complejidad que deberán ser manejadas por ObjectP. Estas variables requieren cierta adaptación.

- **Fase1**: Consiste en adaptar el envoltorio externo. Si tenemos un departamento con empleados entonces los empleados deben estar contenidos en algún tipo de colección. Se admiten las colecciones HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap y Object[]. En las primeras fases del desarrollo de ObjectP no se soportaba Set y sí List con lo que esta fase consistía en adaptar el Set a List. En la práctica esta fase ya no la realiza ObjectP, si hiciera falta se ampliaría el conjunto de colecciones para evitar las adaptaciones de fase1.

Fase 1 Guardar		Fase 1 Leer	
De	A	De	A
`Object`	`Object`	`Object`	`Object`
`Array[][]`	`Array[]`	`Array[]`	`Array[][]`
`HashSet`	`HashSet`	`HashSet`	`HashSet`
`TreeSet`	`TreeSet`	`TreeSet`	`TreeSet`
`ArrayList`	`ArrayList`	`ArrayList`	`ArrayList`
`LinedkList`	`LinedkList`	`LinedkList`	`LinkedList`
`HashMap`	`HashMap`	`HashMap`	`HashMap`
`TreeMap`	`TreeMap`	`TreeMap`	`TreeMap`

Tabla 3.1 Adaptación para los agregados en la fase1.

- **Fase 2**: Se adapta el contenido de las colecciones recursivamente. Si se tiene un ArrayList<String> se debe convertir a ArrayList<PString>. ArrayList no hay que tocarlo, es fase1 y es soportado por ObjectP, lo que hay que adaptar es el contenido de ArrayList. En este caso, como la clase String de Java no es un ObjectP se debe adaptar a 'PString'. Para los arrays se ha creado una clase para manejarlos como cualquier otra colección.

Fase 2 Guardar		Fase 2 Leer	
De	A	De	A
`String`	`PString`	`PString`	`String`
`ObjectP`	`ObjectP`	`ObjectP`	`ObjectP`
`Array[][]`	`PArray`	`PArray`	`Array[][]`
`HashSet`	`PHashSet`	`PHashSet`	`HashSet`
`TreeSet`	`PTreeSet`	`PTreeSet`	`TreeSet`
`ArrayList`	`PArrayList`	`PArrayList`	`ArrayList`
`LinkedList`	`PLinkedList`	`PLinkedList`	`LinkedList`

HashMap	PHashMap	PHashMap	HashMap
TreeMap	PTreeMap	PTreeMap	TreeMap

Tabla 3.2 Adaptación del contenido de los agregados.

3.4. Interfaz de ObjectP

Se muestran todos los métodos de la interfaz de ObjectP. Los métodos están clasificados entre final y no final, estos últimos marcados con asterisco para indicar que deben o pueden ser sobrescritos.

3.4.1. Métodos para crear y conocer las instancias

Este primer grupo de métodos sirve para crear instancias, copiarlas, mostrar su contenido e iniciar la sincronización. Los dos métodos de sincronización son final porque éstos no se modifican, llaman a los que deben ser sobrescritos.

```
* public                  ObjectP                         ()
* public static ObjectP   objpCrearObjetoStatic           ()
* public        ObjectP   objpCrearObjeto                 ()
* public static String    objpGetNombreDeLaClaseStatic    ()
  public final  void      objpSetNombreDeLaClase          (String nombre_clase)
  public final  String    objpGetNombreDeLaClase          ()
  public final  ObjectP   objpCopiar                      ()
  public final  void      objpMostrarObjetoCompleto       (int tab, List mostrds)
  public final  void      objpSincrObjectToUser           ()
  public final  void      objpSincrUserToObject           ()
  public final  void      objpSincrObjectToUserRecursivo (List objs_sincrs)
  public final  void      objpSincrUserToObjectRecursivo (List objs_sincrs)
```

Hay cuatro métodos que deben ser creados o sobrescritos en la plantilla.

- **ObjectP**: Crear. Es el constructor que se debe llamar igual que el nombre de la clase.
- **objpCrearObjectStatic**: Crear. Invoca al constructor para devolver una instancia de la clase.
- **objpCrearObjeto**: Sobrescribir. Otra manera de crear instancias.
- **objpGetNombreDeLaClaseStatic**: Crear. Un método static que devuelve el nombre de la clase.

3.4.2. Métodos equals y hashCode

El método equals recorre las estructuras de ObjectP buscando diferencias. Las estructuras principales de ObjectP son las relacionadas con la información de los atributos, los agregados y la herencia. La de los agregados requiere recursividad. Si no se ha realizado la sincronización los valores de los atributos no serán analizados y para el caso de los agregados no habrá agregados

que visitar. No se considera que equals deba invocar la sincronización, tal vez se quieran comparar las instancias sin tener en cuenta los valores.

Fig. 3.3 Llamadas para recorrer las estructuras de ObjectP recursivamente

Para tareas testers se creó un método que muestra con detalle las estructuras de ObjectP junto con sus valores haciendo un recorrido similar al de equals, el mostrado en la figura 3.3. Por último, el método de hashCode() usa la información anterior para crear un String e invocar al método hashCode de la clase String.

3.4.3. Métodos para la herencia

Este grupo de métodos se usan para recopilar la información de las clases de las que se hereda. Siempre se hereda de alguna clase, ObjectP es la raíz. Son métodos para ser usados desde la plantilla y por la BDO, no es necesario que el desarrollador los conozca.

```
public final Integer      objpSupersConfiguracion          (List      configuracion)
public final List[]       objpSupersTodos                  ()
public final Integer      objpSupersExisteClase            (String    clase_nombre)
public final Integer      objpSupersExisteConexion         (String    clase_subor_nombre,
                                                            String    clase_super_nombre)
public final Boolean      objpSupersHeredaDe               (String    clase_super_nombre)
public final int          objpSuperCantidad                (String    clase_subor_nombre)
public final String       objpSupersGetNombreClaseSuper    (String    clase_subor_nombre)
public final List<String> objpSupersGetNombresClaseSuper   (String    clase_subor_nombre)
public final ObjectP      objpSupersCrearObjeto            (String    clase_nombre)
```

3.4.4. Métodos para los atributos

La configuración de un atributo se hace mediante un único método. En cambio, para recibir información se debe indicar el nombre del atributo y la clase donde ha sido definido. Hay dos métodos para la sincronización que se sobrescriben en la plantilla.

```
  public final Integer    objpAtributoConfiguracion       (List   configuracion)
  public final List[]     objpAtributoTodos               ()
  public final Boolean    objpAtributoExiste              (String clase, String atributo)
  public final Integer    objpAtributoCantidadPorClase    (String clase)
  public final List<Str>  objpAtributoNombresParaUnaClase (String clase)
  public final List[]     objpAtributoIDs                 ()
  public final Boolean    objpAtributoEsID                (String clase, String atributo)
  public final Integer    objpAtributoTipo                (String clase, String atributo)
  public final Boolean    objpAtributoEsNotNull           (String clase, String atributo)
  public final Boolean    objpAtributoEsUnique            (String clase, String atributo)
  public final Boolean    objpAtributoEsIndice            (String clase, String atributo)
  public final String     objpAtributoValorPorDefecto     (String clase, String atributo)
  public final void       objpAtributoSetValor            (String clase, String atributo,
                                                                        Object valor )
  public final Object     objpAtributoGetValor            (String clase, String atributo)
  public final Boolean    objpAtributoRemove              (String clase, String atributo)
* public       void       objpAtributoSincrObjectToUser   ()
* public       void       objpAtributoSincrUserToObject   ()
```

Se definen unas constantes para establecer el tipo de dato:

```
final static Integer   TIPO_DATO_STRING    = 0;
final static Integer   TIPO_DATO_INTEGER   = 1;
final static Integer   TIPO_DATO_BOOLEAN   = 2;
final static Integer   TIPO_DATO_CHARACTER = 3;
```

A no ser que se vaya a hacer uso de los atributos dinámicos no es necesario conocer con detalle estos métodos, basta con saber realizar cambios en la plantilla.

3.4.5. Métodos para los agregados

La idea es análoga a la de los atributos pero hay más métodos al ser su casuística mayor. La palabra ‘agregado’ hace referencia al nombre de la colección y ‘agregadoItem’ a los elementos de la colección. También hay dos métodos para la sincronización que deben ser sobrescritos en la plantilla.

```
  public final void         objpPuedeHaberAgregados         (boolean puede_haber_agregados)
  public final boolean      objpPuedeHaberAgregados         ()
  public final boolean      objpAgregadoConfiguracion       (List   configuracion)
  public final Integer      objpAgregadoConfiguracionDim    (String   clase, String nombre,
                                                             Integer[]dimension)
  public final List[]       objpAgregadoTodos               ()
  public final boolean      objpAgregadoExiste              (String clase, String nombre)
  public final int          objpAgregadoCantidadAgregados   (String clase)
  public final List<String> objpAgregadoNombres             (String clase)
  public final int          objpAgregadoCantidadAgregadosItems (String clase, String nombre)

  public final boolean objpAgregadoContieneAgregadoItemClave (Str clase, Str nombre, ObjectP obj_p)
  public final boolean objpAgregadoContieneAgregadoItemValor (Str clase, Str nombre, ObjectP obj_p)

  public final ObjectP      objpAgregadoCrear               (String clase, String nombre)
  public final void         objpAgregadoBorrar              (String clase, String nombre)
  public final void         objpAgregadoLimpiar             (String clase, String nombre)
```

```
  public final Integer       objpAgregadoSetAgregadoItem (String  clase, String nombre,
                                                          ObjectP key  , ObjectP  agregadito)
  public final Map<Int, Int> objpAgregadoSetAgregadoItems(String clase, String nombre,
                                                          Object agregaditos)

  public final ObjectP    objpAgregadoGetAgregadoItem (Str clase, Str nombre, Object key)
  public final HashSet    objpAgregadoGetAgregadoItemsHashSet    (Str clase, Str nombre)
  public final TreeSet    objpAgregadoGetAgregadoItemsTreeSet    (Str clase, Str nombre)
  public final ArrayList  objpAgregadoGetAgregadoItemsArrayList  (Str clase, Str nombre)
  public final LinkedList objpAgregadoGetAgregadoItemsLinkedList (Str clase, Str nombre)
  public final HashMap    objpAgregadoGetAgregadoItemsHashMap    (Str clase, Str nombre)
  public final TreeMap    objpAgregadoGetAgregadoItemsTreeMap    (Str clase, Str nombre)
  public final Object[]   objpAgregadoGetAgregadoItemsArray      (Str clase, Str nombre)

  public final ObjectP objpAgregadoRemoveAgregadoItem (Str clase, Str nombre, Object  key       )
  public final void    objpAgregadoRemoveAgregadoItem (Str clase, Str nombre, ObjectP agregadito)

* public       void objpAgregadoSincrUserToObject ()
* public       void objpAgregadoSincrObjectToUser ()
```

Se definen unas constantes para establecer el tipo de colección que es cada agregado:

```
    public final static int      AGREGADO_COLECCION_OBJECT      = 0;
    public final static int      AGREGADO_COLECCION_HASHSET     = 1;
    public final static int      AGREGADO_COLECCION_TREESET     = 2;
    public final static int      AGREGADO_COLECCION_ARRAYLIST   = 3;
    public final static int      AGREGADO_COLECCION_LINKEDLIST  = 4;
    public final static int      AGREGADO_COLECCION_HASHMAP     = 5;
    public final static int      AGREGADO_COLECCION_TREEMAP     = 6;
    public final static int      AGREGADO_COLECCION_ARRAY       = 7;
```

3.4.6. Métodos para modificar la estructura

Cuando ya se han guardado instancias en la BDO y se modifica la clase añadiendo o quitando atributos y/o agregados se debe informar para que la BDO haga las operaciones que considere necesarias. Estas operaciones serán del tipo ALTER TABLE.

No es necesario que estos métodos aporten más información que la del nombre del elemento creado o borrado. Para los elementos creados la información estará en las estructuras anteriores y para los borrados debe ser suficiente con la información que tenga la BDO.

```
public final void         objpModificarSupersBorrarSet   (String clase, String nombre)
public final List<String> objpModificarSupersBorrarGet   (String clase              )
public final void         objpModificarSupersCrearSet    (String clase, String nombre)
public final List<String> objpModificarSupersCrearGet    (String clase              )

public final void         objpModificarAtributoBorrarSet (String clase, String nombre)
public final List<String> objpModificarAtributoBorrarGet (String clase              )
public final void         objpModificarAtributoCrearSet  (String clase, String nombre)
public final List<String> objpModificarAtributoCrearGet  (String clase              )

public final void         objpModificarAgregadoBorrarSet (String clase, String nombre)
public final List<String> objpModificarAgregadoBorrarGet (String clase              )
```

3.4.7. Métodos Callback

Estos métodos son cuestionables por ser un mecanismo similar al de los triggers. Tienen su utilidad y si la BDO sólo admite que se realicen operaciones de lectura para comprobaciones no debería haber problemas.

Devuelven un boolean para darle la opción a la BDO de cancelar la operación. Por ejemplo, si PreUpdate devuelve false la BDO debería cancelar la actualización. Y si un método Post devuelve false tal vez debería realizar un RollBack. Pero esto es una decisión que deberá tomar la implementación de la BDO.

Los métodos son llamados por la BDO en los momentos correspondientes. Estos métodos no tienen nada que ver con la plantilla por los que el desarrollador los debe sobrescribir en su código. Deben llamar al super en la primera línea para que se ejecuten en cascada cuando haya herencia.

Estos métodos en ObjectP están vacíos y sólo tienen un 'return true'.

```
* public Boolean objpPrePersist  ()
* public Boolean objpPostPersist ()
* public Boolean objpPostLoad    ()
* public Boolean objpPreUpdate   ()
* public Boolean objpPostUpdate  ()
* public Boolean objpPreRemove   ()
* public Boolean objpPostRemove  ()
```

3.5. Generación automática de la plantilla

Se trata de aplicar sucesivos refinamientos para generar la plantilla con el mínimo esfuerzo.

3.5.1. Anotaciones

Consiste en anotar los elementos necesarios para que un procesador de anotaciones genere automáticamente la plantilla. Las anotaciones serán muy parecidas a las de JDO:

```
@ObjpClass
public class Empleado extends ObejctP{

        @ObjpAtributo
        Integer atributo = 0;
        :    :    :    :    :
}
```

Fig. 3.4 Anotaciones para automatizar la generación de la plantilla de comunicación

Los detalles de cómo deben ser, si todas son necesarias y si deben ir acompañadas de parámetros queda fuera del alcance de esta documentación. Las anotaciones podrían ser las siguientes:

- **ObjpClass**: Para saber el nombre de la clase.
- **ObjpSuper**: Para informar de qué clases se hereda.
- **ObjpAtributo**: Para definir un atributo.
- **ObjpAgregado**: Para definir un agregado.

3.5.2. Reflexión

Las anotaciones que debe añadir el desarrollador siguen siendo necesarias pero la plantilla desaparecería, la información se consultaría en tiempo de ejecución en vez de inicializarla al crear la instancia.

Esta solución podría significar que gran parte de las estructuras de ObjectP dedicadas a recopilar la información no fueran necesarias. Pero sin estas estructuras ObjectP perdería parte de su identidad, la relacionada con el dinamismo de atributos, de agregados y la creación de instancias ObjectP.

No soy partidario de usar reflexión porque es un mecanismo usado por otras herramientas para consultar las anotaciones. Por tanto, será poco probable que se pueda realizar algún descubrimiento o aporte significativo. En cambio, por desconocer si existe algún proyecto similar, sí parece más interesante la idea de integrar ObjectP en java.lang.Object.

3.5.3. Compilador

La opción más avanzada consistiría en integrar ObjectP con java.lang.Object para que Java facilitara la persistencia de forma nativa. Y la plantilla sería creada por el compilador por lo que no habría código adicional en los .java escritos por el desarrollador. Tampoco serían necesarias las anotaciones ya que la información necesaria es conocida por el compilador. Las anotaciones podrían usarse para añadir información opcional como las restricciones o para indicar qué atributos no se quiere que sean persistentes.

3.6. Usando las instancias

Este apartado aborda las distintas maneras de usar las instancias de la clase escrita por el programador cuando es también un ObjctP. Lo habitual será que el desarrollador las use a través

de la interfaz que él mismo haya creado y que la API de ObjectP quede para la BDO. Pero como se ve en la figura 3.5 hay otras posibilidades:

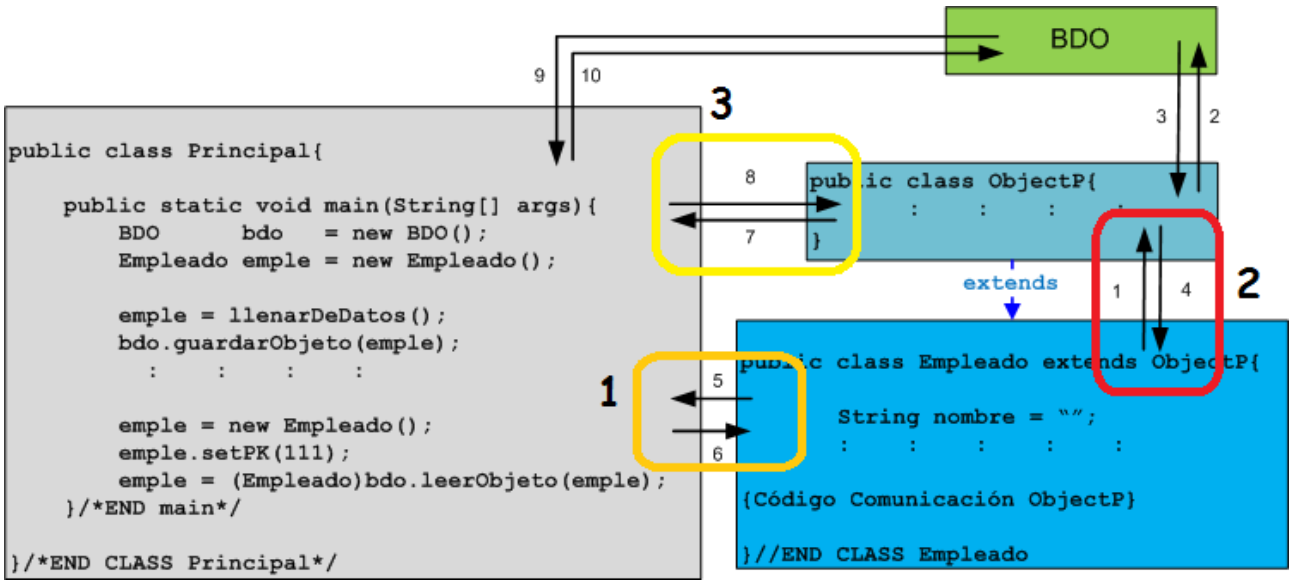

Fig. 3.5 Los tres tipos de comunicación que afectan al desarrollador.

1. **Clase Principal – Clase del desarrollador**: Esta comunicación es independiente de ObjectP. El desarrollador invoca los métodos que haya creado.
2. **Clase del Desarrollador – ObjectP**: Es la comunicación realizada con la plantilla. Las funciones principales son la inicialización y la sincronización.
3. **Clase Principal – ObjectP**: Las flechas no tocan la clase del desarrollador. Hay situaciones en las que podría resultar útil acceder directamente como, por ejemplo, manejar atributos y agregados dinámicos o usar instancias de ObjectP sin que se modele ninguna clase.

Capítulo 4. Ejemplos de clases ObjectP

Este capítulo muestra una serie ejemplos que cubren gran parte de los casos que un desarrollador puede necesitar. Un desarrollador podrá usar los ejemplos aquí mostrados para adaptar la plantilla a sus necesidades si aun no se dispone de un mecanismo que implemente la plantilla.

Se han creado las siguientes clases:

- **ClasePri**: No tiene agregados, sólo atributos.
- **ClasePriExt**: Hereda de 'ClasePri' sobrescribiendo dos atributos.
- **ClaseAgr**: Una clase que muestra una amplia variedad de agregados (no representados en la figura 4.1). La figura 4.1 sólo indica que los agregados de 'ClaseAgr' están compuestos por la clase 'ClasePri' o la clase 'ClasePriExt'.

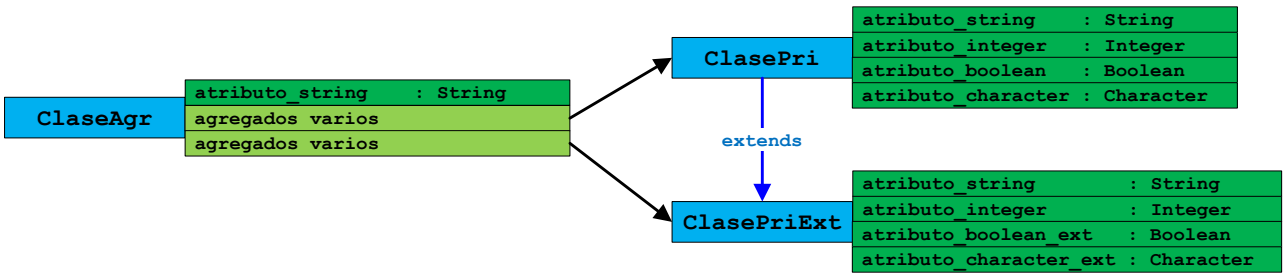

Fig. 4.1 Ejemplos de clases ObjectP

4.1. Clase primitiva: ClasePri

La clase 'ClasePri' se caracteriza por ser la clase más simple de todas. Sin agregados ni herencia.

Esta clase define los siguientes atributos:

```
// VARIABLES atributos ------------------------------------------------------
    String         atributo_string    =     "";
    Integer        atributo_integer   =      0;
    Boolean        atributo_boolean   =  false;
    Character      atributo_character =    ' ';
```

La parte de las constantes de la plantilla queda configurada de la siguiente manera:

```
    //Para poder crear el metodo bbddooNombreDeLaClaseStatic()
    private final static String PROPIO_NOMBRE_CLASE = "ClasePri";


    //Herencia
    private final static ObjectP[] PROPIO_SUPER_REPRESENTANTE      =
                                                    {ObjectP.objpCrearObjetoStatic() };
    //Atributos
    private final static String[]   PROPIO_ATRIBUTO_NOMBRE        = {"atributo_string"  ,
                                                                     "atributo_integer" ,
                                                                     "atributo_boolean" ,
                                                                     "atributo_character"};
    private final static Boolean[]  PROPIO_ATRIBUTO_ES_ID         = {true ,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Integer[]  PROPIO_ATRIBUTO_TIPO   = {ObjectP.TIPO_DATO_STRING   ,
                                                              ObjectP.TIPO_DATO_INTEGER  ,
                                                              ObjectP.TIPO_DATO_BOOLEAN  ,
                                                              ObjectP.TIPO_DATO_CHARACTER};

    private final static Boolean[]  PROPIO_ATRIBUTO_ES_REQUIRED   = {false,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Boolean[]  PROPIO_ATRIBUTO_ES_UNIQUE     = {false,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Boolean[]  PROPIO_ATRIBUTO_ES_INDICE     = {false,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Object[]   PROPIO_ATRIBUTO_DEFAULT_VALUE = {  " ",
                                                                        0,
                                                                     false,
                                                                       ' '};

    //Agregados
    private final static String[]     PROPIO_AGREGADO_NOMBRE              = { };
    private final static Integer[]    PROPIO_AGREGADO_COLECCION           = { };
    prívate final static Integer[][]  PROPIO_AGREGADO_DIMENSIONES         = { };
    private final static ObjectP[]    PROPIO_AGREGADO_REPRESENTANTE_CLAVE = { };
    private final static ObjectP[]    PROPIO_AGREGADO_REPRESENTANTE_VALOR = { };
```

Los métodos de sincronización son los siguientes:

```
    @Override
    public void objpAtributoSincrUserToObject(){
    /*BEGIN*/
        super.objpAtributoSincrUserToObject();
        this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 0],
                                   this.atributo_string   );
        this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 1],
                                   this.atributo_integer  );


        this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 2],
```

```
                                this.atributo_boolean  );
    this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 3],
                                this.atributo_character);
}/*END objpAtributoSincrUserToObject*/


@Override
public void objpAtributoSincrObjectToUser(){
/*BEGIN*/
    super.objpAtributoSincrObjectToUser();
    this.atributo_string     = (String)    this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                      PROPIO_ATRIBUTO_NOMBRE[ 0] );
    this.atributo_integer    = (Integer)   this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                      PROPIO_ATRIBUTO_NOMBRE[ 1] );
    this.atributo_boolean    = (Boolean)   this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                      PROPIO_ATRIBUTO_NOMBRE[ 2] );
    this.atributo_character  = (Character) this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                        PROPIO_ATRIBUTO_NOMBRE[ 3] );
}/*END objpAtributoSincrObjectToUser*/
```

4.2. Clase extendida: ClasePriExt

La clase 'ClasePrExti' extiende de 'ClasePri' sobrescribiendo dos atributos aunque a la hora de escribir la plantilla es indiferente ya que para ObjectP un atributo se define con el nombre de la clase donde se define y el nombre del atributo.

```
// VARIABLES atributos ----------------------------------------------------------
    String         atributo_string        =     ""; //Sobrescribe
    Integer        atributo_integer       =      0; //Sobrescribe
    Boolean        atributo_boolean_ext   =  false; //Nuevo
    Character      atributo_character_ext =    ' '; //Nuevo
```

En rojo se muestran las diferencias con 'ClasePri':

```
    //Para poder crear el metodo bbddooNombreDeLaClaseStatic()
    private final static String PROPIO_NOMBRE_CLASE = "ClasePriExt";

    //Herencia
    private final static ObjectP[] PROPIO_SUPER_REPRESENTANTE      =
                                                    {ClasePri.objpCrearObjetoStatic() };
    //Atributos
    private final static String[]   PROPIO_ATRIBUTO_NOMBRE   = {"atributo_string"      ,
                                                                "atributo_integer"     ,
                                                                "atributo_boolean_ext" ,
                                                                "atributo_character_ext"};
    private final static Boolean[]  PROPIO_ATRIBUTO_ES_ID         = {true ,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Integer[]  PROPIO_ATRIBUTO_TIPO   = {ObjectP.TIPO_DATO_STRING    ,
                                                              ObjectP.TIPO_DATO_INTEGER   ,
                                                              ObjectP.TIPO_DATO_BOOLEAN   ,
                                                              ObjectP.TIPO_DATO_CHARACTER};

    private final static Boolean[]  PROPIO_ATRIBUTO_ES_REQUIRED   = {false,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Boolean[]  PROPIO_ATRIBUTO_ES_UNIQUE     = {false,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Boolean[]  PROPIO_ATRIBUTO_ES_INDICE     = {false,
                                                                     false,
                                                                     false,
                                                                     false};
    private final static Object[]   PROPIO_ATRIBUTO_DEFAULT_VALUE = {  " ",
                                                                        0,
                                                                     false,
    //Agregados
    private final static String[]     PROPIO_AGREGADO_NOMBRE              = { };
```

```
private final static Integer[]    PROPIO_AGREGADO_COLECCION           = { };
prívate final static Integer[][]  PROPIO_AGREGADO_DIMENSIONES         = { };
private final static ObjectP[]    PROPIO_AGREGADO_REPRESENTANTE_CLAVE = { };
private final static ObjectP[]    PROPIO_AGREGADO_REPRESENTANTE_VALOR = { };
```

Los métodos de sincronización son los siguientes. Es importante la llamada a super para sincronizar los atributos de Empelado:

```
@Override
public void objpAtributoSincrUserToObject(){
/*BEGIN*/
    super.objpAtributoSincrUserToObject();
    this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 0],
                               this.atributo_string       );
    this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 1],
                               this.atributo_integer      );
    this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 2],
                               this.atributo_boolean_ext  );
    this.objpAtributoSetValor (PROPIO_NOMBRE_CLASE, PROPIO_ATRIBUTO_NOMBRE[ 3],
                               this.atributo_character_ext);
}/*END objpAtributoSincrUserToObject*/


@Override
public void objpAtributoSincrObjectToUser(){
/*BEGIN*/
    super.objpAtributoSincrObjectToUser();
    this.atributo_string         = (String)    this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                          PROPIO_ATRIBUTO_NOMBRE[ 0] );
    this.atributo_integer        = (Integer)   this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                          PROPIO_ATRIBUTO_NOMBRE[ 1] );
    this.atributo_boolean_ext    = (Boolean)   this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                          PROPIO_ATRIBUTO_NOMBRE[ 2] );
    this.atributo_character_ext  = (Character) this.objpAtributoGetValor (PROPIO_NOMBRE_CLASE,
                                                                          PROPIO_ATRIBUTO_NOMBRE[ 3] );
}/*END objpAtributoSincrObjectToUser*/
```

4.3. Clase con agregados: ClaseAgr

La clase ‘ClaseAgr’ ya es un ejemplo de una clase más real pues tiene atributos y agregados. Esta clase reúne una amplia variedad de agregados para mostrar las distintas maneras en las que se pueden usar y cómo se realiza la sincronización. Debido a que es un código extenso los detalles sólo se muestran en la clase.

Capítulo 5. Interioridades de ObjectP

5.1. Separando datos de código

Junto con la clase ObjectP se ha creado la clase ObjectDataP. Esta clase tiene todas las estructuras que ObjectP necesita para almacenar la información. Esto quiere decir que en ObjectP sólo hay definida una variable de tipo ObjectDataP. La figura 5.1 muestra la única variable definida en ObjectP.

```
public class ObjectP{

    private ObjectDataP object_basico = new ObjectDataP();
    :      :      :
```

Fig. 5.1 ObjectDataP, una clase que sólo define constantes y variables.

Esto tiene varios propósitos. El primero es por simple claridad de código pues como programador me gusta separar en ficheros trozos de código cada vez que encuentro una justificación, me resulta más fácil de mantener. El segundo motivo ya está más relacionado con las necesidades de la BDO. Una clase como es ObjectDataP contiene exactamente, ni más ni menos, lo que la BDO necesita para trabajar. Por ejemplo, consistiría en crear una clase 'EmpleadoData.java' que tuviera las variables nombre, apellido, teléfono, etc. No es algo necesario pero tal vez aplicar esta metodología ayude a adaptar nuestra mentalidad como programadores y consigamos así mejores códigos para trabajar con una BDO.

5.2. ObjectDataP

Se describen las constantes y atributos de la clase ObjectDataP. Todas son 'public' para que sean accedidas desde ObjectP de la siguiente manera:

```
private ObjectDataP object_data = new ObjectDataP();

object_data.atributo_clase = "Empleado";
```

5.2.1. Constantes

Para ayudar a construir la plantilla. Con ellas se indica el tipo de los atributos primitivos y el tipo de colección que será el agregado.

```
public final static Integer   TIPO_DATO_STRING    = 0;
public final static Integer   TIPO_DATO_INTEGER   = 1;
public final static Integer   TIPO_DATO_BOOLEAN   = 2;
public final static Integer   TIPO_DATO_CHARACTER = 3;

public final static int      AGREGADO_COLECCION_OBJECT      = 0;
public final static int      AGREGADO_COLECCION_HASHSET     = 1;
public final static int      AGREGADO_COLECCION_TREESET     = 2;
public final static int      AGREGADO_COLECCION_ARRAYLIST   = 3;
public final static int      AGREGADO_COLECCION_LINKEDLIST  = 4;
public final static int      AGREGADO_COLECCION_HASHMAP     = 5;
public final static int      AGREGADO_COLECCION_TREEMAP     = 6;
public final static int      AGREGADO_COLECCION_ARRAY       = 7;
```

5.2.2. Grupo inicial

Con 'grupo inicial' se hace referencia a unas variables que no se han enmarcado en ningún otro grupo. Una es para el nombre de la clase y otra es para indicar si es un tipo primitivo.

```
public String   nombre_de_la_clase   =    "";
public Boolean  es_tipo_primitivo    = false;
```

Las clases como String e Integer deberán ser adaptadas a PString e PInteger como se verá más adelante. Sólo estás clases tendrán '**es_tipo_primitivo**' a true y se usa en ciertos procesos recursivos para indicar el caso base.

5.2.3. Atributos

En una clase escrita por el programador como puede ser 'Empleado', se considera atributo a la variable que es de tipo primitivo como String o Integer. Para cada atributo se recoge cierta información con:

```
Map<String, List<String>> atributo_clase            //Key=NombreClase
Map<String, Integer>      atributo_tipo             //Key=KeyHerencia
Map<String, Object>       atributo_valor            //Key=KeyHerencia
List<String>              atributo_id_lista         //Valor=KeyHerencia
```

```
Map<String, Boolean>      atributo_id_es             //Key=KeyHerencia
Map<String, Object>       atributo_valor_por_defecto //Key=KeyHerencia
Map<String, Boolean>      atributo_es_REQUIRED       //Key=KeyHerencia
Map<String, Boolean>      atributo_es_unique         //Key=KeyHerencia
Map<String, Boolean>      atributo_es_indice         //Key=KeyHerencia
```

Los Maps usan dos tipos de claves. Una es el nombre de la clase donde se define el atributo y otra es el resultado de concatenar el nombre de la clase con el nombre del atributo (usando un carácter separador especial). Esta segunda clave se usa para poder mezclar en un mismo map atributos de varias clases debido a que, cuando hay herencia, pueden repetirse los nombres de los atributos.

Se describen con detalle cada elemento:

- **atributo_clase**: Para cada nombre de clase (que es la clave del map) se tiene una lista con los nombres de los atributos que hay definidos en esa clase.
- **atributo_tipo**: Para indicar el tipo del que es el atributo. Se basa en las constantes `TIPO_DATO_xxxxx`.
- **atributo_valor**: Sólo cuando se realiza la sincronización aquí habrá un dato válido. Si el atributo es la edad del empleado aquí habrá un Integer con 35, por ejemplo. Para evitar problemas será siempre mejor definir los atributos como 'Integer' en vez de cómo 'int'.
- **atributo_id_lista**: La lista de atributos que el desarrollador considera que deben ser identificadores.
- **atributo_id_es**: Tiene la finalidad de informar rápidamente si un atributo es identificador.
- **atributo_valor_por_defecto**: El valor por defecto que debe tener cada atributo.
- **atributo_es_ required**: Indica si no se debe dejar a null en la BDO.
- **atributo_es_unique**: Indica si el valor no se debe repetir.
- **atributo_es_indice**: Le sugiere a la BDO que construya un índice para este atributo.

De estos elementos sólo los tres primeros son los importantes aunque también deben mantenerse lo relacionado con los identificadores. Los demás son restricciones que una futura versión de ObjectP buscará colocar en una clase o estructura aparte donde se recojan éstas y otras restricciones. La definición del atributo es una cosa y el conjunto de restricciones otra. Además, permitiría que un atributo pudiera tener varios conjuntos de restricciones.

5.2.4. Agregados

Las variables para almacenar la información de los agregados son:

```
Map<String, List<String>>             agregado_clase
Map<String, Integer>                  agregado_coleccion
Map<String, ObjectBDO>                agregado_representante_clave
Map<String, ObjectBDO>                agregado_representante_valor
Map<String, List<Integer>>            agregado_dimension
Map<String, ObjectP>                  agregado_valor_object
Map<String, HashSet<ObjectP>>         agregado_valor_hashset
Map<String, TreeSet<ObjectP>>         agregado_valor_treeset
Map<String, ArrayList<ObjectP>>       agregado_valor_arraylist
Map<String, LinkedList<ObjectP>>      agregado_valor_linkedlist
Map<String, HashMap<ObjectP,ObjectP>> agregado_valor_hashmap
Map<String, TreeMap<ObjectP,ObjectP>> agregado_valor_treemap
Map<String, Object[]>                 agregado_valor_array
```

- **agregado_clase**: Análogo a 'atributo_clase'. Para cada nombre de clase (que es la clave del map) se tiene una lista con los nombres de los agregados que hay definidos en esa clase.
- **agregado_colección**: Similar a 'atributo_tipo'. Informa sobre la manera en la que se agrupan los elementos. Se basa en las constantes `AGREGADO_COLECCION_xxxxx`.
- **agregado_representante_xxx**: Los representantes son instancias que se usan para tener un control de lo que puede ir dentro de la colección. Es como las clases parametrizadas List<String>. Este representante sirve para compararlo con lo que se quiere insertar dentro de la colección, para hacer preguntas como '¿Sois de la misma clase?' o '¿Heredas de?' Para los maps se debe proporcionar también el representante de la clave.
- **agregado_dimension**: Cuando el agregado es un array se deben especificar las dimensiones del array mediante una lista de enteros. Para Object[10][5][7] se debe crear la lista {10,5,6} donde get(0)=10.
- **agregado_valor_xxx**: Sólo una de las ocho variables no será null. Será en función del valor de 'agregado_coleccion' y sólo tendrá valores válidos después de la sincronización.

5.2.5. Herencia

Recoge la información de las clases de las que se hereda. Por cada nombre de clase hay una lista de clases de las que se hereda (aunque Java sólo soporta herencia simple). El representante es necesario para saber de quién hereda él a su vez para poder hacer un recorrido recursivo.

```
Map<String, List<String>> supers_nombre
Map<String, ObjectBDO>    supers_representante
```

5.2.6. Modificaciones

Cuando una clase se queda antigua se le realizan modificaciones como añadirle o quitarle atributos. Esta información debe ser también recogida de alguna manera. Cuando se quita sólo hace falta informar de qué se quita dando el nombre del elemento y la clase donde fue definido. Cuando se añade se debe aportar toda la configuración pero esa información ya estará en las variables anteriores.

```
Map<String, List<String>>  modificar_borrar_super_nombre
Map<String, List<String>>  modificar_anadir_super_nombre
Map<String, List<String>>  modificar_borrar_atributo_nombre
Map<String, List<String>>  modificar_anadir_atributo_nombre
Map<String, List<String>>  modificar_borrar_agregado_nombre
Map<String, List<String>>  modificar_anadir_agregado_nombre
```

5.2.7. Observaciones

Estos comentarios sugieren modificaciones para una futura clase ObjectP.

Es interesante observar que las estructuras para la herencia son un caso particular de las estructuras de los agregados. Es decir, internamente ObjectP podría haber creado un agregado que no fuera visible al exterior donde almacenase la información de la herencia:

- ObjectP definiría el agregado 'supers'.
- Tipo de colección: HashMap<PString, List<PString>>
- Key="Empleado". Value={"ObjectP"}
- Key="Secretario". Value={"Empleado"}

En realidad, sólo las estructuras de los agregados son necesarias. Las estructuras de los atributos podrían ser reemplazadas por la de los agregados aunque requeriría ciertas adaptaciones. Se haría creando un agregado por cada nombre de atributo y el tipo colección sería el de ObjectP. El resto de información que acompaña al atributo, la de las restricciones, deberá estar aparte tal y como ya se ha sugerido. Y deberá haber al menos dos conjuntos de restricciones, uno para los tipos primitivos y otro para los no primitivos.

5.3. Clase auxiliar: ManejarColecciones

Se trata de una clase con procedimientos auxiliares que ObjectP necesita. Por claridad se han colocado en una clase aparte. El manejo de ciertas operaciones relacionadas con los arrays y las colecciones tiene su complejidad, de ahí el nombre de la clase, porque sólo tiene procedimientos auxiliares que trabajan con colecciones y arrays.

5.3.1. Copiar

Se ha creado en ObjectP un método para poder copiar objetos, no simplemente clonar. La copia consiste en duplicar todas las estructuras donde ObjectP almacena la información, las descritas en el apartado 5.2.

```
public static List  objpCopiarListObj           (List list_obj             )
public static List  objpCopiarListObjp          (List list_objp            )
public static Map   objpCopiarMapObjObj         (Map  map_obj_obj          )
public static Map   objpCopiarMapObjObjp        (Map  map_obj_objp         )
public static Map   objpCopiarMapObjpObjp       (Map  map_objp_objp        )

public static Map   objpCopiarMapObjListObj     (Map  map_obj_list_obj     )
public static Map   objpCopiarMapObjListObjp    (Map  map_obj_listobjp     )
public static Map   objpCopiarMapObjMapObjObj   (Map  map_obj_map_obj_obj  )
public static Map   objpCopiarMapObjMapObjObjp  (Map  map_obj_map_obj_objp )
public static Map   objpCopiarMapObjMapObjpObjp (Map  map_obj_map_objp_objp)
```

5.3.2. Arrays anidados

Un array multidimensional, Integer[][], puede ser manejado como Integer[Integer[]]. La ventaja de esta segunda forma es el paso de parámetros, un único método puede manejar cualquier dimensión.

```
Object[]   arrayAnidadoCrearArray    (Integer[] dimen, int    d)
Integer[]  arrayAnidadoDimensiones   (Object[]  array)
void       arrayAnidadoInicializar   (Object[]  array, Integer[] dimen, Object valor)
void       arrayAnidadoSetValor      (Object[]  array, Integer[] coord, Object valor)
Object     arrayAnidadoGetValor      (Object[]  array, Integer[] coord)
boolean    arrayAnidadoContieneValor (Object[]  array, Integer[] dimen, Object valor)
Integer[]  arrayAnidadoIncrementarCoordenadasA (Integer[] coord, Integer[] dimen)
boolean    arrayAnidadoIncrementarCoordenadasB (Integer[] coord, Integer[] dimen)
```

Estos métodos se han movido a la clase PArray pero se mantiene aquí la información para tener agrupado todo lo relacionados con el procesamiento de colecciones.

5.3.3. Adaptación de agregados fase 1

Los agregados soportan un número determinado de colecciones, las indicadas por las constantes '**AGREGADO_COLECCION_xxx**'. Si hubiera alguna colección no soportada habría que adaptarla.

Al principio, ObjectP aceptaba List pero no Set por lo que un Set se adaptaba a List. Ahora, ninguno de los métodos mostrados a continuación son ya necesarios pero se mantiene el concepto de adaptación de fase 1 por que es necesario distinguirlo de otro tipo de adaptaciones.

```
public static List         objToList        (Object   objp  )
public static Object       listToObj        (List     list  )
public static HashSet      setToHashSet     (Set      set   )
public static TreeSet      setToTreeSet     (Set      set   )
public static ArrayList    setToArrayList   (Set      set   )
public static LinkedList   setToLinkedList  (Set      set   )
public static HashSet      listToHashSet    (List     list  )
public static TreeSet      listToTreeSet    (List     list  )
public static ArrayList    listToArrayList  (List     list  )
public static LinkedList   listToLinkedList (List     list  )
public static HashMap      mapToHashMap     (Map      map   )
public static TreeMap      mapToTreeMap     (Map      map   )
public static Object[][]   arrayToArray2    (Object[] array )
public static Object[][][] arrayToArray3    (Object[] array )
```

Cuando se lee de la BDO hay que reconstruir la variable original y los arrays son los más problemáticos por ser manejados como arrays anidados. Éstos deben adaptarse a sus dimensiones y tipos correspondientes. Para escribir estos métodos se ha usado una plantilla por lo que sería fácil dar soporte a más de tres dimensiones. Estos métodos para los arrays se usan en la plantilla como se mostró en los ejemplos de clases del capítulo anterior.

```
String[]         array1objToArray1str (Object[]     array1_obj)
String[][]       array2objToArray2str (Object[][]   array2_obj)
String[][][]     array3objToArray3str (Object[][][] array3_obj)
Integer[]        array1objToArray1int (Object[]     array1_obj)
Integer[][]      array2objToArray2int (Object[][]   array2_obj)
Integer[][][]    array3objToArray3int (Object[][][] array3_obj)
Boolean[]        array1objToArray1bol (Object[]     array1_obj)
Boolean[][]      array2objToArray2bol (Object[][]   array2_obj)
Boolean[][][]    array3objToArray3bol (Object[][][] array3_obj)
Character[]      array1objToArray1chr (Object[]     array1_obj)
Character[][]    array2objToArray2chr (Object[][]   array2_obj)
Character[][][] array3objToArray3chr (Object[][][] array3_obj)
```

5.3.4. Adaptación de agregados fase 2

Sucede que el contenido de una colección pueden ser instancias que no son ObjectP como un ArrayList<String>. ArrayList es un tipo de colección soportada por ObjectP pero la clase String no es un ObjectP. La clase String debe adaptarse a PString.

Hay sólo dos métodos que se encargan de esta tarea debido a que son recursivos y usan 'instanceof'. Sólo es necesario un método público para adaptar y otro para desadaptar. Esta fase no sería necesaria si java.lang.Object fuera ObjectP pues todo sería un ObjectP.

```
public static Object   objpTransformarJavaToObjectP  (Object coleccion_obj )
public static Object   objpTransformarObjectPToJava  (Object coleccion_objp)
```

5.4. Adaptación de las clases Java

Este apartado no sería necesario si ObjectP estuviera integrado en java.lang.Object y el compilador añadiera la plantilla. Pero mientras esto no suceda, para poder aprovechar todas las capacidades de ObjectP, es necesario adaptar algunas clases de Java y convertirlas en ObjectP.

Este trabajo ha adaptado las clases asociadas a los tipos primitivos que soporta ObjectP y las colecciones principales: HashSet, TreeSet, ArrayList, LinkedList, HashMap y TreeMap además de crear una para los arrays llamada PArray.

Estas adaptaciones son transparentes al desarrollador, se realizan de manera automática en la fase 2 de la adaptación de agregados.

Hay dos maneras de adaptar las clases:

1. **Atributo público**: Creando una clase donde el atributo es público. Al crear una instancia se accede directamente a la variable. Se usa para los tipos primitivos.

   ```
   public PString extends ObjectP{
       public String valor = "";
   ```

2. **Implementando la interfaz**: Creando una clase donde el atributo es privado. Obliga a implementar la interfaz. Esta es la opción que se usa para las colecciones.

5.4.1. Adaptación de los tipos primitivos

Las clases asociadas a los tipos primitivos se adaptan usando la técnica del atributo público. Aquí se muestra sólo el caso de la adaptación de la clase Integer a PInteger. Debido a que el atributo se hace público y no se implementa la interfaz el contenido de la clase es simplemente el siguiente (aunque luego hay que añadirle la plantilla):

```
public class PInteger extends ObjectP{
    public Integer valor = 0;

    public PInteger(Integer valor){
        this.valor = valor;
    }
```

Las constantes de la plantilla quedan como se muestra a continuación. En rojo se maraca lo que cambia para String, Character y Boolean. Se indica que no puede ser null y que debe ser único. El código restante, el de la sincronización, no difiere del mostrado en las clases ejemplo, basta examinar cómo sincroniza la clase 'ClasePri' un atributo Integer.

```
String      PROPIO_NOMBRE_CLASE = "PInteger";
ObjectP[]   PROPIO_SUPER_REPRESENTANTE = {ObjectP.objpCrearObjetoStatic()};

String[]    PROPIO_ATRIBUTO_NOMBRE           = {"valor"};
Integer[]   PROPIO_ATRIBUTO_TIPO             = {TIPO_DATO_INTEGER};
Boolean[]   PROPIO_ATRIBUTO_ES_REQUIRED      = {true};
Boolean[]   PROPIO_ATRIBUTO_ES_UNIQUE        = {true};
Boolean[]   PROPIO_ATRIBUTO_ES_INDICE        = {true};
Object[]    PROPIO_ATRIBUTO_DEFAULT_VALUE    = {" "};
String[]    PROPIO_AGREGADO_NOMBRE          = { };
ObjectP[]   PROPIO_AGREGADO_REPRESENTANTE   = { };
Integer[]   PROPIO_AGREGADO_COLECCION       = { };
```

5.4.2. Adaptación de las colecciones

Para cualquier clase que no sea un tipo primitivo lo adecuado es crear una nueva clase que parezca que hereda de HashSet o la que corresponda. Digo que parezca porque en realidad no puede al tener que heredar de ObjectP.

Se ha buscado crear clases funcionales de cara al exterior y que parezca que hereda de las colecciones originales de Java. Pero si las clases quedan sólo para uso interno de ObjectP hubiera bastado con haber creado una clase sólo con los métodos de añadir y sacar.

- **PHashSet / PTreeSet:** HashSet hereda de otras clases pero PHashSet debe heredar de ObjectP. Se debe cumplir lo siguiente si se quiere mantener una clase funcional de cara al exterior:
    - PHashSet debe heredar e ObjectP. Esto implica que si, en algún momento HashSet o alguna de las clases de las que hereda cambia PHashSet deberá cambiar.
    - PHashSet debe implementar todas las interfaces que implementa HashSet.
    - PHashSet debe soportar los tipos parametrizados pues así sucede con Set y HashSet.
    - Los tipos parametrizado deben heredar de ObjectP pues no tiene sentido que un PHashSet almacene elementos que no sean ObjectP.

La definición de HashSet es la siguiente, (no coincide exactamente con la de TreeSet):

```
public class       PHashSet<T extends ObjectP>
       extends     ObjectP
       implements Serializable, Cloneable, Iterable<E>, Collection<E>, Set<E>{

       private HashSet<T> valor = new HashSet<T>();
```

Un ejemplo de cómo se adaptan los métodos de HashSet es el siguiente. Se pone Override porque la mayoría de los métodos son especificados por las interfaces. Los métodos propios que HashSet pueda definir fuera de las interfaces implementadas no llevarán 'Override' porque no se heredan (al heredar de ObjectP no se puede heredar de HashSet).

```
@Override
public boolean contains(Object o){
    return this.valor.contains(o);
}
```

Por último, se muestran las constantes de la plantilla. Debido a los tipos parametrizados una de ellas debe dejar de ser 'static'. El resto del código no es necesario mostrarlo, la sincronización es la misma que la mostrada en los ejemplos de la clase 'ClaseAgr'.

```
static String PROPIO_NOMBRE_CLASE = "PHashSet";

static ObjectP[] PROPIO_SUPER_REPRESENTANTE      = {ObjectP.objpCrearObjetoStatic() };

static String[]   PROPIO_ATRIBUTO_NOMBRE           = { };
static Integer[]  PROPIO_ATRIBUTO_TIPO             = { };
static Boolean[]  PROPIO_ATRIBUTO_ES_REQUIRED      = { };
static Boolean[]  PROPIO_ATRIBUTO_ES_UNIQUE        = { };
static Boolean[]  PROPIO_ATRIBUTO_ES_INDICE        = { };
static String[]   PROPIO_ATRIBUTO_DEFAULT_VALUE    = { };

static String[]    PROPIO_AGREGADO_NOMBRE        = {"valor"}
static Integer[]   PROPIO_AGREGADO_COLECCION     = {ObjectP.AGREGADO_COLECCION_HASHSET}
static Integer[][] PROPIO_AGREGADO_DIMENSIONES   = {new Integer[0]                    }
       ObjectP[]   PROPIO_AGREGADO_REPRESENTANTE = {E.objpCrearObjetoStatic()         }
```

- **PArrayList / PLinkedList:** Adaptar la clase ArrayList a PArrayList es similar a PHashSet.

```
public class       PArrayList<E extends ObjectP>
       extends     ObjectP
       implements  Serializable , Cloneable, Iterable<E>   ,
                   Collection<E>, List<E>   , RandomAccess {

       private ArrayList<E> valor = new ArrayList<E>();
```

- **PHashMap / PTreeMap:** Adaptar un HashMap y TreeMap requiere tener en cuenta que ahora hay clave y valor. Pero no cambia mucho respecto a las adaptaciones anteriores.

```
public class       PHashMap<K extends ObjectP, V extends ObjectP>
       extends     ObjectP
       implements  Serializable, Cloneable, Map<K,V>{

       private Map<K, V> valor = new HashMap<K, V>();
```

En este caso las constantes de la plantilla deben tener en cuenta la clave.

```
static String[]    PROPIO_AGREGADO_NOMBRE              = {"valor"                            }
static Integer[]   PROPIO_AGREGADO_COLECCION           = {ObjectP.AGREGADO_COLECCION_HASHMAP}
static Integer[][] PROPIO_AGREGADO_DIMENSIONES         = {new Integer[0]                     }
       ObjectP[]   PROPIO_AGREGADO_REPRESENTANTE_CLAVE = {K.objpCrearObjetoStatic()          }
       ObjectP[]   PROPIO_AGREGADO_REPRESENTANTE_VALOR = {V.objpCrearObjetoStatic()          }
```

- **PArray:** Para poder manejar los array correctamente resulta conveniente crearles una clase para ellos. Se puede dar el caso de tener algo como 'ArrayList< Integer[][] > con lo que la fase 2 de la adaptación de agregados hace uso de PArray.

```
// No static ----------------------------
Iterator  iterator()
Integer[] dimensiones    ()
Integer   size           ()
void      inicializar    (Object   valor)
Integer[] contieneValorA (Object valor)
boolean   contieneValorB (Object valor)
void      setArray (Object[] array)
Object[]  getArray ()
void      setValor (Integer[] coordenadas, Object valor)
Object    getValor (Integer[] coordenadas)

// static -----------------------------
Object[]     arrayAnidadoCrearArray (Integer[] dimensiones )
Object[]     arrayAnidadoCrearArray (Integer[] dimensiones, Object valor_ini)
Integer[]    arrayAnidadoCoordenadasIni    (Integer   dim)
Integer      arrayAnidadoCantidadElementos (Object[]  array)
Integer[]    arrayAnidadoDimensiones       (Object[]  array)
void         arrayAnidadoInicializar       (Object[]  array, Object valor)
Integer[]    arrayAnidadoContieneValorA    (Object[]  array, Object valor)
boolean      arrayAnidadoContieneValorB    (Object[]  array, Object valor)
void         arrayAnidadoSetValor (Object[]  array, Int[] coord, Object valor)
Object       arrayAnidadoGetValor (Object[]  array, Int[] coord)
Integer[]    arrayAnidadoIncrementarCoordenadasA (Int[] coord, Int[] dim)
boolean      arrayAnidadoIncrementarCoordenadasB (Int[] coord, Int[] dim)
Object[][]   arrayToArray2    (Object[] array )
Object[][][] arrayToArray3    (Object[] array )
```

Capítulo 6. Pruebas

6.1. Descripción de las pruebas realizadas

Las pruebas a realizar se organizan en baterías de pruebas. Cada una de estas baterías realiza un test a una parte del código. Un resumen de las pruebas realizadas es el siguiente:

- **PArray**: Se han adaptado varias clases Java para que sean ObjectP. PArray no tiene su correspondiente clase en Java pero se ha tenido que crear para trabajar más fácilmente con los arrays. Tiene una amplia variedad de métodos que hay que probar.
- **Adaptación de agregados**: Se prueba la fase 2 de la adaptación de los agregados. En esta misma batería de pruebas se realizan los test a todas las PClases, las clases Java que han sido adaptadas.
- **Pruebas con objetos ObjectP**: Por último, se realizan las pruebas a las clases ejemplo que reúnen a todos los elementos.

Para realizar estas pruebas se han creado clases en un paquete aparte pero dentro del mismo proyecto de ObjectP.

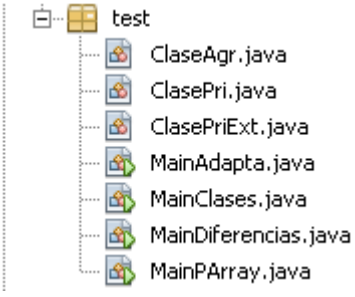

Fig. 6.1 Paquete de pruebas.

Debido a la complejidad para interpretar los resultados no se ha usado ningún tipo de metodología. ObjectP implementa el método 'objpMostrarObjetoCompleto()' que muestra detalladamente las estructuras de ObjectP. La mayoría de las pruebas complejas consistían en crear, por ejemplo, una instancia de PString, de PArrayList o de ClaseAgr y mostrarla por pantalla con este método. Tal vez no sea muy elegante pero sí bastante fiable porque involucra muchos elementos.

6.2. Pruebas a PArray

Las pruebas se realizan desde test.MainPArray. Se trata de crear instancias de PArray para luego probar sus métodos principales:

```
Object[]   arrayAnidadoCrearArray     (Integer[] dim                  )
Object[]   arrayAnidadoCrearArray     (Integer[] dim, Object valor_ini)
Integer[]  arrayAnidadoCoordenadasIni (Integer   dim)
Integer[]  arrayAnidadoDimensiones    (Object[]  array)
void       arrayAnidadoInicializar    (Object[]  array, Object valor)
Integer[]  arrayAnidadoContieneValorA (Object[]  array, Object valor)
boolean    arrayAnidadoContieneValorB (Object[]  array, Object valor)
void       arrayAnidadoSetValor       (Object[]  array, Integer[] coord, Object valor)
Object     arrayAnidadoGetValor       (Object[]  array, Integer[] coord)
Integer[]  arrayAnidadoIncrementarCoordenadasA (Integer[] coord, Integer[] dimensiones)
boolean    arrayAnidadoIncrementarCoordenadasB (Integer[] coord, Integer[] dimensiones))
```

La clase 'PArray.java' va acompañada de 'PArrayIterator.java' pues se implementa la interfaz Iterator. La batería de pruebas se realiza con los siguientes procedimientos.

```
prueba_arrayAnidadoCrearArrayVacio          ();
prueba_arrayAnidadoInicializar              ();
prueba_arrayAnidadoCrearArrayIni            ();
prueba_arrayAnidadoDimensiones              ();
prueba_arrayAnidadoSetValor                 ();
prueba_arrayAnidadoGetValor                 ();
prueba_arrayAnidadoIncrementarCoordenadasA1 ();
prueba_arrayAnidadoIncrementarCoordenadasA2 ();
prueba_arrayAnidadoIncrementarCoordenadasB1 ();
prueba_arrayAnidadoIncrementarCoordenadasB2 ();
prueba_arrayAnidadoContieneValorA           ();
prueba_arrayAnidadoContieneValorB           ();
prueba_PArrayIterator();
```

Además, la mayoría de las pruebas incluyen dos líneas adicionales para probar equals() y hashCode(). Cuando se han creado dos instancias diferentes se comparan entre ellas y si no se compara consigo mismo:

```
System.out.println("hashCode: " + parray.hashCode()     );
System.out.println("equals:   " + parray.equals(parray) );
```

Todas estas pruebas resultaron satisfactorias.

6.3. Pruebas de adpatación de agregados fase2.

Esta sección prueba los métodos de la clase ManejarColecciones relacionados con la adaptación de los agregados. Estas pruebas están en 'test.MainAdapta'. En esta batería de pruebas se están probando todas las clases Java adaptadas como las que se ven en la tabla 6.1.

La fase 2 consiste en adaptar el contenido de un ObjectP para que todo sea ObjectP. Se debe hacer un recorrido recursivo para analizar las colecciones que haya dentro de colecciones adaptando tanto las colecciones como los tripos primitivos. La tabla 6.1 muestra las adaptaciones realizadas.

Fase 2 Guardar		Fase 2 Leer	
De	A	De	A
`String`	`PString`	`PString`	`String`
`ObjectP`	`ObjectP`	`ObjectP`	`ObjectP`
`Array[][]`	`PArray`	`PArray`	`Array[][]`
`HashSet`	`PHashSet`	`PHashSet`	`HashSet`
`TreeSet`	`PTreeSet`	`PTreeSet`	`TreeSet`
`ArrayList`	`PArrayList`	`PArrayList`	`ArrayList`
`LinkedList`	`PLinkedList`	`PLinkedList`	`LinkedList`
`HashMap`	`PHashMap`	`PHashMap`	`HashMap`
`TreeMap`	`PTreeMap`	`PTreeMap`	`TreeMap`

Tabla 6.1 Adaptación avanzada para el contenido de los agregados.

La adaptación se realiza en varios pasos como intenta mostrar la figura 6.2. Las pruebas buscan recorrer, si no todas las combinaciones, sí todos los procedimientos.

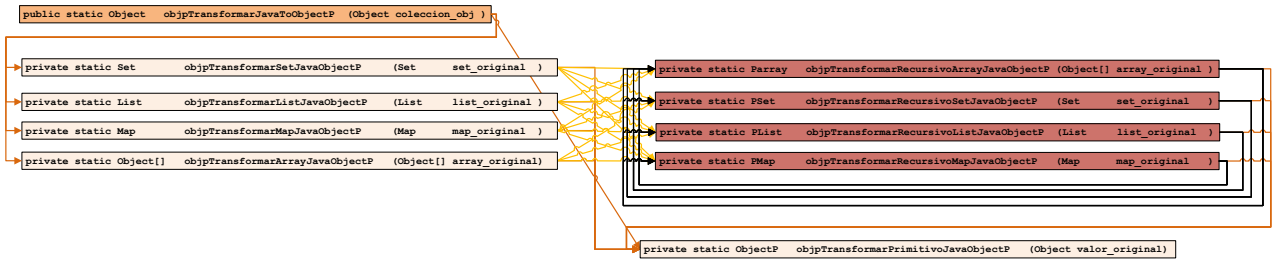

Fig. 6.2 Esquema de llamadas entre métodos.

A continuación se describen todas las pruebas realizadas en 'test.MainAdapta'. Cada prueba realiza las dos adaptaciones, la de guardar y la de leer que es la inversa.

- **De String a PString:** Se trata de adaptar una variable de tipo primitivo. Aunque los tipos primitivos generalmente irán en atributos, cuando se tenga ArrayList<String>, los String habrá que adaptarlos, serán manejadas como un nivel más de agregados y no como atributos.

```
    private static void prueba_StrToStrp(){
    /*VAR*/
        String          string          = "aaa";
        PString         pstring         = null;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_StrToStrp ***");
FASE2   pstring = (PString)ManejarColecciones.objpTransformarJavaToObjectP(string);
        System.out.println("Valor PString: " + pstring.valor);
        System.out.println("*** FIN GUARDAR prueba_StrToStrp ***");
        System.out.println("");

        System.out.println("");
        System.out.println("*** INI LEER prueba_StrToStrp ***");
        pstring = new PString("bbb");
FASE2   string  = (String)  ManejarColecciones.objpTransformarObjectPToJava(pstring);
        System.out.println("Valor String: " + string);
        System.out.println("*** FIN LEER prueba_StrToStrp ***");
    }/*END prueba_StrToStrp*/
```

- **De ObjectP a ObjectP:** En realidad no hay nada que hacer pero el método que realiza la transformación debe ser capaz de manejar este caso devolviendo lo mismo que recibe.

```
    private static void prueba_ObjpToObjp(){
    /*VAR*/
        ClasePri    clase_objp     = new ClasePri("str", 3, true, 'c');
        ClasePri    clase_objp_aux = null;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_ObjpToObjp ***");
FASE2   clase_objp_aux = (ClasePri)ManjrColec.objpTransformarJavaToObjectP(clase_objp)
        clase_objp_aux.mostrar();
        System.out.println("*** FIN GUARDAR prueba_ObjpToObjp ***");
        System.out.println("");

        System.out.println("");
        System.out.println("*** INI LEER prueba_ObjpToObjp ***");
FASE2   clase_objp_aux = (ClasePri)ManejarColecciones.objpTransformarObjectPToJava(clase_objp)
        clase_objp_aux.mostrar();
        System.out.println("*** FIN LEER prueba_ObjpToObjp ***");
    }/*END prueba_ObjpToObjp*/
```

- **De String[] a PString[]:** Un array de una dimensión cuyo contenido se deba adaptar. En la lectura hay una fase extra, que llamaremos fase 0, debido a que al leer lo que proporciona la fase 2 es un Object[] que hay que adaptarlo a String[].

```
    private static void prueba_array1strToArrayObjp (){
    /*VAR*/
        Object[]      array_1d_obj     = null;
        String[]      array_1d_string  = new String[3];
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_array1strToArrayObjp ***");
        for (i=0; i<3; i++){
            array_1d_string[i] = "A"+i;
        }/*for*/
        mostrarArrayObj(array_1d_string);
FASE2   array_1d_obj = (Object[])ManjrColec.objpTransformarJavaToObjectP(array_1d_string)
        mostrarArrayObjp(array_1d_obj);
        System.out.println("*** INI GUARDAR prueba_array1strToArrayObjp ***");

        System.out.println("*** INI LEER prueba_array1strToArrayObjp ***");
        for (i=0; i<3; i++){
            array_1d_obj[i] =  new PString("A"+i);
        }/*for*/
FASE2   array_1d_obj    = (Object[])ManjrColec.objpTransformarObjectPToJava(array_1d_obj);
```

```
FASE0   array_1d_string = ManjrColec.array1objToArray1str(array_1d_obj);
        mostrarArrayObj(array_1d_string);
        System.out.println("*** FIN prueba_array1strToArrayObjp ***");
    }/*END prueba_array1strToArrayObjp */
}
```

- **De Integer[][] a PInteger[][]:** El caso anterior pero con un array de dos dimensiones.

```
    private static void prueba_array2intToListObjp(){
    /*VAR*/
//Guar  Integer[][]           array_2d_integer  = new Integer[3][3];
        Object[]              array_anidado     = null;
        int                   i,j;
//Leer  PInteger[][]          array_2d_pinteger = new PInteger[3][3];
        Object[][]            array_2d_obj      = null;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_array2intToListObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                array_2d_integer[i][j] = i*j;
            }/*for*/
        }/*for*/
FASE2    array_anidado = (Object[])ManjrColec.objpTransformarJavaToObjectP(array_2d_integer);
        mostrarArrayObjp(array_anidado);
        System.out.println("*** FIN GUARDAR prueba_array2intToListObjp ***");

        System.out.println("*** INI LEER prueba_array2intToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                array_2d_pinteger[i][j] = new PInteger(i*j);
            }/*for*/
        }/*for*/
FASE2    array_anidado =(Object[]) ManjrColec.objpTransformarObjectPToJava(array_2d_pinteger)
FASE0   array_2d_obj  = PArray.arrayToArray2(array_anidado);
FASE0   array_2d_integer = ManejarColecciones.array2objToArray2int(array_2d_obj);
        mostrarArrayObj(array_2d_integer);
        System.out.println("*** FIN LEER prueba_array2intToArrayObjp ***");
    }/*END prueba_array2intToListObjp*/
```

- **De Character[][][] a PCharacter[][][]:** Y el caso de tres dimensiones. Para soportar más de tres dimensiones es necesario ampliar la colección de métodos asociados a la fase 0.

```
    private static void prueba_array3chrToArrayObjp(){
    /*VAR*/
//Guar  Character[][][]       array_3d_character = new Character[3][3][3];
        Object[]              array_anidado      = null;
        char[]                array_char_aux     = null;
        int                   i,j,k;
//Leer  PCharacter[][][]      array_3d_pcharacter = new PCharacter[3][3][3];
        Object[][][]          array_3d_obj      = null;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_array3chrToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                for (k=0; k<3; k++){
                    array_char_aux              = Character.toChars( 65 + (i*j*k) );
                    array_3d_character[i][j][k] = array_char_aux[0];
                }/*for*/
            }/*for*/
        }/*for*/
FASE2    array_anidado = (Object[])ManjrColec.objpTransformarJavaToObjectP(array_3d_character)
        mostrarArrayObjp(array_anidado);
        System.out.println("*** FIN GUARDAR prueba_array3chrToArrayObjp ***");

        System.out.println("*** INI LEER prueba_array2intToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                for (k=0; k<3; k++){
                    array_char_aux               = Character.toChars( 65 + (i*j*k) );
                    array_3d_pcharacter[i][j][k] = new PCharacter( array_char_aux[0] );
                }/*for*/
            }/*for*/
        }/*for*/
FASE2    array_anidado = (Object[]) ManjrColec.objpTransformarObjectPToJava(array_3d_pcharacter)
```

```
FASE0   array_3d_obj = PArray.arrayToArray3(array_anidado);
FASE0   array_3d_character = ManejarColecciones.array3objToArray3chr(array_3d_obj);
        mostrarArrayObj(array_3d_character);
        System.out.println("*** FIN LEER prueba_array2intToArrayObjp ***");
    }/*END prueba_array3chrToArrayObjp*//
```

- **De HashSet<String> a HashSet<PString>:** La variable del desarrollador, el agregado, ahora es un conjunto de elementos primitivos.

```
    private static void prueba_hashsetstrToHashSetObjp (){
    /*VAR*/
        HashSet<String>   hashset_string   = new HashSet<String> ();
        HashSet<PString>  hashset_pstring  = new HashSet<PString>();
        int           i;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_hashsetstrToHashSetObjp ***");
        for (i=0; i<3; i++){
            hashset_string.add("A"+i);
        }/*for*/
FASE2   hashset_pstring = (HashSet)ManjrColec.objpTransformarJavaToObjectP(hashset_string);
        mostrarSetObjp(hashset_pstring);
        System.out.println("*** FIN GUARDAR prueba_hashsetstrToHashSetObjp ***");

        System.out.println("*** INI LEER prueba_hashsetstrToHashSetObjp ***");
        for (i=0; i<3; i++){
            hashset_pstring.add( new PString("A"+i) );
        }/*for*/
FASE2    hashset_string = (HashSet) ManjrColec.objpTransformarObjectPToJava(hashset_pstring);
        mostrarSetObj(hashset_string);
        System.out.println("*** FIN LEER prueba_hashsetstrToHashSetObjp ***");
    }/*END prueba_hashsetstrToHashSetObjp*/
```

- **De TreeSet<Integer> a TreeSet<PInteger>:** Igual que el caso de HashSet.

```
    private static void prueba_liststrToListObjp(){
    /*VAR*/
        TreeSet<Integer>   treeset_integer   = new TreeSet<Integer> ();
        TreeSet<PInteger>  treeset_pinteger  = new TreeSet<PInteger>();
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_treesetintToTreeSetObjp ***");
        for (i=0; i<3; i++){
            treeset_integer.add(i);
        }/*for*/
FASE2    treeset_pinteger = (TreeSet)ManjrColec.objpTransformarJavaToObjectP(treeset_integer);
        mostrarSetObjp(treeset_pinteger);
        System.out.println("*** FIN GUARDAR prueba_treesetintToTreeSetObjp ***");

        System.out.println("*** INI LEER prueba_treesetintToTreeSetObjp ***");
        for (i=0; i<3; i++){
            treeset_pinteger.add( new PInteger(i) );
        }/*for*/
FASE2    treeset_integer = (TreeSet) ManjrColec.objpTransformarObjectPToJava(treeset_pinteger);
        mostrarSetObj(treeset_integer);
        System.out.println("*** FIN LEER prueba_treesetintToTreeSetObjp ***");
    }/*END prueba_liststrToListObjp*/
```

- **De ArrayList<String> a ArrayList<PStirng>:** Igual que los casos de HashSet y TreeSet.

```
    public static void prueba_arrayliststrToArrayListObjp (){
    /*VAR*/
        ArrayList<String>  list_string      = new ArrayList<String>();
        ArrayList<PString> list_1d_pstring  = null;
        int           i;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_arrayliststrToArrayListObjp ***");
        for (i=0; i<3; i++){
            list_string.add("A"+i);
        }/*for*/
FASE2   list_1d_pstring = (ArrayList)ManjrColec.objpTransformarJavaToObjectP(list_string);
        mostrarListaObjp(list_1d_pstring);
        System.out.println("*** FIN GUARDAR prueba_arrayliststrToArrayListObjp ***");
```

```
        System.out.println("*** INI LEER prueba_arrayliststrToArrayListObjp ***");
        list_1d_pstring = new ArrayList();
        for (i=0; i<3; i++){
            list_1d_pstring.add( new PString("A"+i) );
        }/*for*/
FASE2   list_string  = (ArrayList) ManjrColec.objpTransformarObjectPToJava(list_1d_pstring);
        mostrarListaObj(list_string);
        System.out.println("*** FIN LEER prueba_arrayliststrToArrayListObjp ***");
    }/*END prueba_arrayliststrToArrayListObjp*/
```

- **De LinkedList<Integer> a LinekdList<PInteger>:** Igual que los casos anteriores de Set y List.

```
    private static void prueba_linkedlistintToLinkedListObjp (){
    /*VAR*/
        LinkedList<Integer>  list_integer      = new LinkedList<Integer>();
        LinkedList<PInteger> list_1d_pinteger  = null;
        int           i;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_linkedlistintToLinkedListObjp ***");
        for (i=0; i<3; i++){
            list_integer.add(i);
        }/*for*/
FASE2   list_1d_pinteger = (LinkedList)ManjrCole.objpTransformarJavaToObjectP(list_integer)
        mostrarListaObjp(list_1d_pinteger);
        System.out.println("*** FIN GUARDAR prueba_linkedlistintToLinkedListObjp ***");

        System.out.println("*** INI LEER prueba_linkedlistintToLinkedListObjp ***");
        list_1d_pinteger = new LinkedList();
        for (i=0; i<3; i++){
            list_1d_pinteger.add( new PInteger(i) );
        }/*for*/
FASE2   list_integer = (LinkedList) ManjrCole.objpTransformarObjectPToJava(list_1d_pinteger)
        mostrarListaObj(list_integer);
        System.out.println("*** FIN LEER prueba_linkedlistintToLinkedListObjp ***");
    }/*END prueba_linkedlistintToLinkedListObjp*/
```

- **De HashMap<Chr,Chr> a HashMap<PChar, PChar>:** Igual que antes, no hay diferencias aunque ahora la colección tenga clave y valor.

```
    private static void prueba_hashmapchrToHashMapObjp(){
    /*VAR*/
        HashMap<Character ,Character > map_chrchr  = new HashMap<Character ,Character >();
        HashMap<PCharacter,PCharacter> map_pchrchr = new HashMap<PCharacter,PCharacter>();
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_hashmapchrToHashMapObjp ***");
        for (i=0; i<3; i++){
            array_char_aux = Character.toChars( 65 + (i) );
            map_chrchr.put(array_char_aux[0],  array_char_aux[0]);
        }/*for*/
FASE2   map_pchrchr = (HashMap)ManjrColec.objpTransformarJavaToObjectP(map_chrchr);
        mostrarMapObjp(map_pchrchr);
        System.out.println("*** FIN GUARDAR prueba_hashmapchrToHashMapObjp ***");

        System.out.println("*** INI LEER prueba_hashmapchrToHashMapObjp ***");
        for (i=0; i<3; i++){
            array_char_aux = Character.toChars( 65 + (i) );
            map_pchrchr.put( new PCharacter(array_char_aux[0]), new PCharacter(array_char_aux[0]) );
        }/*for*/
FASE2   map_chrchr  = (HashMap) ManjrColec.objpTransformarObjectPToJava(map_pchrchr);
        mostrarMapObj(map_chrchr);
        System.out.println("*** FIN LEER prueba_hashmapchrToHashMapObjp ***");
    }/*END prueba_hashmapchrToHashMapObjp*/
```

- **De TreeMap<String, String> a TreeMap<PString, PStirng>:** Igual que HashMap.

```
    private static void prueba_treemapstrToTreeMapObjp(){
    /*VAR*/
```

```
        TreeMap<String , String >  map_strstr     = new TreeMap<String , String >();
        TreeMap<PString, PString>  map_pstrstr    = new TreeMap<PString, PString>();
        int                        i;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_treemapstrToTreeMapObjp ***");
        for (i=0; i<3; i++){
            map_strstr.put("A"+i, "A"+(i+10) );
        }/*for*/
FASE2   map_pstrstr = (TreeMap)ManejarColec.objpTransformarJavaToObjectP(map_strstr);
        mostrarMapObjp(map_pstrstr);
        System.out.println("*** FIN GUARDAR prueba_treemapstrToTreeMapObjp ***");

        System.out.println("*** INI LEER prueba_treemapstrToTreeMapObjp ***");
        for (i=0; i<3; i++){
            map_pstrstr.put( new PString("A"+i), new PString("A"+(i+10)) );
        }/*for*/
FASE2   map_strstr  = (TreeMap) ManejarColec.objpTransformarObjectPToJava(map_pstrstr);
        mostrarMapObj(map_strstr);
        System.out.println("*** FIN LEER prueba_treemapstrToTreeMapObjp ***");
    }/*END prueba_treemapstrToTreeMapObjp*/
```

- **De ClasePri[] a ClasePri[]:** En este caso se podría pensar que no hay nada que hacer porque tanto envolvente (el array) como contenido (ClasePri) son elementos que ObjectP sabe manejar. Pero debido a que el contenido del array es un objeto no primitivo no se sabe a priori si tiene o no agregados con lo que debe ser procesado. Esta prueba es muy similar a la prueba String[], de hecho es igual al guardar pero al leer se debe pasar de Object[] a ClasePri[] y esto no es tan directo en Java, no se puede hacer un cast del array entero, se debe recorrer elemento a elemento.

```
    private static void prueba_array1objpToArrayObjp(){
    /*VAR*/
        Object[]         array_1d_obj       = new Object[3];
        ClasePrimitiva[] array_1d_primitiva = new ClasePrimitiva[3]; //Podría no usarse
        int              i;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_array1objpToArrayObjp ***");
        for (i=0; i<3; i++){
            array_1d_primitiva[i] = new ClasePrimitiva("A"+i,i,true,'a');
        }/*for*/
FASE2   array_1d_obj = (Object[])ManjrColec.objpTransformarJavaToObjectP(array_1d_primitiva)
        mostrarArrayObjp(array_1d_obj);
        System.out.println("*** FIN GUARDAR prueba_array1objpToArrayObjp ***");

        System.out.println("*** INI LEER prueba_array1objpToArrayObjp ***");
        for (i=0; i<3; i++){
            array_1d_obj[i] = new ClasePrimitiva("A"+i,i,true,'a');
        }/*for*/
FASE2   array_1d_obj = (Object[]) ManjrColec.objpTransformarObjectPToJava(array_1d_obj)
//      array_1d_primitiva = (ClasePrimitiva[])array_1d_obj;         //No soportado
FASE0   array_1d_primitiva = new ClasePrimitiva[array_1d_obj.length];
FASE0   for (i=0; i<array_1d_obj.length; i++){
FASE0       array_1d_primitiva[i] = (ClasePrimitiva)array_1d_obj[i];
FASE0   }/*for*/
        mostrarArrayObjp(array_1d_primitiva);
        System.out.println("*** FIN LEER prueba_array1objpToArrayObjp ***");
    }/*END prueba_array1objpToArrayObjp*/
```

- **De ObjectP[][] a ObjectP[][]:** El código es muy similar al caso anterior, simplemente se amplía una dimensión más.

```
    private static void prueba_array2objpToArrayObjp(){
    /*VAR*/
        Integer[]              dim                = new Integer[] {3,3};
        Object[]               array_anidado      = PArray.arrayAnidadoCrearArray(dim)
        ClasePrimitiva[][]     array_2d_primitiva = new ClasePrimitiva[3][3];
```

```
        ClasePrimitiva         valor              = null;
        int                    i,j;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_array2objpToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                array_2d_primitiva[i][j] = new ClasePrimitiva("A"+(i*j),i*j,true,'a');
            }/*for*/
        }/*for*/
FASE2   array_anidado = (Object[])ManjrCole.objpTransformarJavaToObjectP(array_2d_primitiva)
        mostrarArrayObjp(array_anidado);
        System.out.println("*** FIN GUARDAR prueba_array2intToArrayObjp ***");

        System.out.println("*** INI LEER prueba_array2objpToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                Integer[] coordenadas    = new Integer[] {i,j};
                valor                    = new ClasePrimitiva("A"+(i*j),i*j,true,'a');
                PArray.arrayAnidadoSetValor(array_anidado, coordenadas, valor);
            }/*for*/
        }/*for*/
FASE2   array_anidado = (Object[]) ManjrColec.objpTransformarObjectPToJava(array_anidado)
FASE0   for (i=0; i<3; i++){
FASE0       for (j=0; j<3; j++){
FASE0           Integer[] coordenadas    = new Integer[] {i,j};
FASE0           array_2d_primitiva[i][j] =
FASE0                (ClasePrimitiva)PArray.arrayAnidadoGetValor(array_anidado, coordenadas)
FASE0       }/*for*/
FASE0   }/*for*/
        mostrarArrayObjp(array_2d_primitiva);
        System.out.println("*** FIN LEER prueba_array2objpToArrayObjp ***");
    }/*END prueba_array2objpToArrayObjp*/
```

- **De ObjectP[][][] a ObjectP[][][]:** La única diferencia es un nivel más de bucle.

```
    private static void prueba_array3objpToArrayObjp(){
    /*VAR*/
        Integer[]            dimensiones        = new Integer[] {3,3,3};
        Object[]                array_anidado      = PArray.arrayAnidadoCrearArray(dimensiones)
        ClasePrimitiva[][][] array_3d_primitiva = new ClasePrimitiva[3][3][3];
        ClasePrimitiva       valor              = null;
        int                  i, j, k;
    /*BEGIN*/
        System.out.println("*** INI GUARDAR prueba_array3objpToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                for (k=0; k<3; k++){
                    array_3d_primitiva[i][j][k] = new ClasePrimitiva("A"+(i*j*k),i*j*k,true,'a')
                }/*for*/
            }/*for*/
        }/*for*/
FASE2   array_anidado =
             (Object[])ManejarColecciones.objpTransformarJavaToObjectP(array_3d_primitiva);
        mostrarArrayObjp(array_anidado);
        System.out.println("*** FIN GUARDAR prueba_array3objpToArrayObjp ***");
        System.out.println("");

        System.out.println("");
        System.out.println("*** INI LEER prueba_array3objpToArrayObjp ***");
        for (i=0; i<3; i++){
            for (j=0; j<3; j++){
                for (k=0; k<3; k++){
                    Integer[] coordenadas    = new Integer[] {i,j,k};
                    valor                    = new ClasePrimitiva("A"+(i*j*k),i*j*k,true,'a');
                    PArray.arrayAnidadoSetValor(array_anidado, coordenadas, valor);
                }/*for*/
            }/*for*/
        }/*for*/
FASE2   array_anidado = (Object[])ManejarColecciones.objpTransformarObjectPToJava(array_anidado);
FASE0   for (i=0; i<3; i++){
FASE0       for (j=0; j<3; j++){
FASE0           for (k=0; k<3; k++){
FASE0               Integer[] coordenadas       = new Integer[] {i,j,k};
FASE0               array_3d_primitiva[i][j][k] =
FASE0               (ClasePrimitiva)PArray.arrayAnidadoGetValor(array_anidado, coordenadas);
FASE0           }/*for*/
FASE0       }/*for*/
FASE0   }/*for*/
        mostrarArrayObjp(array_3d_primitiva);
```

```
    System.out.println("*** FIN LEER prueba_array3objpToArrayObjp ***");
}/*END prueba_array3objpToArrayObjp*/
```

Para ser minuciosos habría que seguir realizando más pruebas para abarcar más casos aunque en este punto se tiene un alto grado de seguridad de que está bien construido.

6.4. Pruebas con clases ObjectP

Estas baterías de pruebas son para las clases 'ClasePri', 'ClasePriExt' y 'ClaseAgr'. Las dos primeras, al no tener agregados, son sencillas. En cambio, con la clase 'ClaseAgr' sucede lo mismo que ha pasado con las pruebas de adaptación de agregados, no se han podido probar todos los casos.

6.4.1. ClasePri

Esta primera prueba consiste en crear una instancia de una clase que sólo tiene atributos primitivos. Se trata de darle valores y comprobar que las sincronizaciones funcionan correctamente. Los tres métodos de las pruebas son los siguientes y los resultados han sido satisfactorios.

```
private static void pruebaClasePriAntesDeSincronizar(){
/*VAR*/
    ClasePri hoja = null;
/*BEGIN*/
    hoja = new ClasePri("str", 1, true, 'a');
    hoja.objpMostrarObjetoCompleto(0, null, false);
    hoja.mostrar();
}/*END pruebaClasePriAntesDeSincronizar*/


private static void pruebaClasePriDespuesDeSincronizar(){
/*VAR*/
    ClasePri hoja = null;
/*BEGIN*/
    hoja = new ClasePri("str", 1, true, 'a');
    hoja.objpSincrUserToObject();
    hoja.objpMostrarObjetoCompleto(0, null, false);
    hoja.mostrar();
}/*END pruebaClasePriDespuesDeSincronizar*/


private static void pruebaClasePriSincronizarGet(){
/*VAR*/
    ClasePri hoja = null;
/*BEGIN*/
    hoja = new ClasePri("str", 1, true, 'a');
    hoja.objpAtributoSetValor ("ClasePri",  "atributo_string"    ,  "txt");
    hoja.objpAtributoSetValor ("ClasePri",  "atributo_integer"   ,      3);
    hoja.objpAtributoSetValor ("ClasePri",  "atributo_boolean"   ,  false);
    hoja.objpAtributoSetValor ("ClasePri",  "atributo_character",    'z');
    hoja.mostrar();
    hoja.objpMostrarObjetoCompleto(0, null, false);
    hoja.objpSincrObjectToUser();
    hoja.mostrar();
}/*END pruebaClasePriSincronizarGet*/
```

El resultado fue positivo. Después de la sincronización se llama al método 'objpMostrarObjeto' que muestra el contenido de las estructuras de ObjectP. Y por otro lado se invoca a un método propio de la clase 'ClasePri' que muestra su contenido y así poder asegurar que lo que ve ObjectP y lo que ve la instancia de 'ClasePri' es lo mismo (o no cuando no se realiza la sincronización).

6.4.2. ClasePriExt

La clase 'ClasePriExt' extiende de 'ClasePri'. Los atributos sin el sufijo '_ext' son los que se sobrescriben ya que tienen el mismo nombre. Se busca ver como maneja ObjectP la herencia.

El constructor asigna valores a los atributos supers por comodidad y así tener valores para todos los atributos fácilmente.

La definición de 'ClasePriExt' es la siguiente.

```
// VARIABLES atributos ------------------------------------------------------
    String         atributo_string        =     "";    //Sobrescritura
    Integer        atributo_integer       =      0;    //Sobrescritura
    Boolean        atributo_boolean_ext   =  false;
    Character      atributo_character_ext =    ' ';

// CONSTRUCTORES ------------------------------------------------------------
    public ClasePriExt(String  atributo_string,  Integer   atributo_integer,
                       Boolean atributo_boolean, Character atributo_character){
    /*BEGIN*/
        this.atributo_string        = atributo_string;
        this.atributo_integer       = atributo_integer;
        this.atributo_boolean_ext   = atributo_boolean;
        this.atributo_character_ext = atributo_character;
        super.atributo_string       = atributo_string  + atributo_string;
        super.atributo_integer      = atributo_integer + atributo_integer;
        super.atributo_boolean      = !atributo_boolean;
    }/*END ClasePriExt*/
```

Se realizan las tres pruebas anteriores. En este caso hay ocho atributos, los cuatro de 'ClasePriExt' y los cuatro que se heredan.

```
    private static void pruebaClasePriExtAntesDeSincronizar(){
    /*VAR*/
        ClasePriExt hoja_ext = null;
    /*BEGIN*/
        hoja_ext = new ClasePriExt("str", 1, true, 'a');
        hoja_ext.objpMostrarObjetoCompleto(0, null, false);
        hoja_ext.mostrar();
    }/*END pruebaClasePriExtAntesDeSincronizar*/


    private static void pruebaClasePriExtDespuesDeSincronizar(){
    /*VAR*/
        ClasePriExt hoja_ext = null;
    /*BEGIN*/
        hoja_ext = new ClasePriExt("str", 1, true, 'a');
        hoja_ext.objpSincrUserToObject();
        hoja_ext.objpMostrarObjetoCompleto(0, null, false);
        hoja_ext.mostrar();
    }/*END pruebaClasePriExtDespuesDeSincronizar*/


    private static void pruebaClasePriExtSincronizarGet(){
    /*VAR*/
```

```
        ClasePriExt hoja_ext = null;
    /*BEGIN*/
        hoja_ext = new ClasePriExt("str", 1, true, 'a');
        hoja_ext.objpAtributoSetValor ("ClasePriExt",  "atributo_string"       ,  "txt");
        hoja_ext.objpAtributoSetValor ("ClasePriExt",  "atributo_integer"      ,      3);
        hoja_ext.objpAtributoSetValor ("ClasePriExt",  "atributo_boolean_ext"  ,  false);
        hoja_ext.objpAtributoSetValor ("ClasePriExt",  "atributo_character_ext",    'z');
        hoja_ext.mostrar();
        hoja_ext.objpMostrarObjetoCompleto(0, null, false);
        hoja_ext.objpSincrObjectToUser();
        hoja_ext.mostrar();
    }/*END pruebaClasePriExtSincronizarGet*/
```

Aquí es interesante visualizar lo mostrado por 'MostrarObjectCompeto' pues da información por separado de la herencia:

```
Puntero     : test.ClasePriExt@785d65
NombreClase : ClasePriExt
..........
Supers:
ClasePriExt hereda de: ClasePri,
ClasePri hereda de: ObjectP,
..........

Atributos:
ClasePriExt define: atributo_string, atributo_integer, atributo_boolean_ext, atributo_ch
ClasePri define: atributo_string, atributo_integer, atributo_boolean, atributo_character
..........

Agregados:
NO HAY
..........

Representantes de los Supers
SupersRepresentanteClase : ClasePri
SupersRepresentanteClase : ObjectP
..........

Atributos
IDs : [ClasePri%atributo_string, ClasePriExt%atributo_string]
-
Key             : ClasePri%atributo_boolean
ID es           : false
Valor           : null
Tipo            : TIPO_DATO_BOOLEAN
ValorPorDefecto : false
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
Key             : ClasePri%atributo_string
ID es           : true
Valor           : null
Tipo            : TIPO_DATO_STRING
ValorPorDefecto :
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
Key             : ClasePriExt%atributo_character_ext
ID es           : false
Valor           : null
Tipo            : TIPO_DATO_CHARACTER
ValorPorDefecto :
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
Key             : ClasePriExt%atributo_integer
ID es           : false
Valor           : null
Tipo            : TIPO_DATO_INTEGER
ValorPorDefecto : 0
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
```

```
Key             : ClasePriExt%atributo_boolean_ext
ID es           : false
Valor           : null
Tipo            : TIPO_DATO_BOOLEAN
ValorPorDefecto : false
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
Key             : ClasePri%atributo_integer
ID es           : false
Valor           : null
Tipo            : TIPO_DATO_INTEGER
ValorPorDefecto : 0
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
Key             : ClasePri%atributo_character
ID es           : false
Valor           : null
Tipo            : TIPO_DATO_CHARACTER
ValorPorDefecto :
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
Key             : ClasePriExt%atributo_string
ID es           : true
Valor           : null
Tipo            : TIPO_DATO_STRING
ValorPorDefecto :
EsNotNull       : false
EsUnique        : false
EsIndice        : false
-
..........

Agregados
Puede haber agregados: true
-
```

6.4.3. ClaseAgr

Estas pruebas se realizan después de haber realizado las pruebas de adaptación de agregados. Estas adaptaciones son invocadas en los métodos de sincronización de los agregados. En estas pruebas lo que se analiza son los métodos de sincronización en sí asumiendo que las adaptaciones no van a dar problemas. Lo que se analiza entonces es la prellamada y la postllamada a los métodos de adaptaciones.

Las pruebas tienen una fase de escritura y una de lectura (igual que se ha hecho en las pruebas anteriores de adaptaciones). Ambas han sido forzadas en el sentido de que las acciones de sincronización deben ser invocadas por la BDO y aquí se hacen en test.MainClases.

- **Escritura**: La fase de escritura consiste en preparar las estructuras de ObjectP con la información de las variables del desarrollador.

  La BDO debe llamar a 'objpSincrUserToObject()' pero aquí se hace en test.MainClases.

- **Lectura**: La fase de lectura consiste en llevar la información desde las estructuras de ObjectP a las variables de la instancia del desarrollador.

La BDO debe llamar a ‘objpSincrObjectToUser()’ pero aquí se hace en test.MainClases.

Esta sección informa de las pruebas realizadas y de sus resultados. También se añaden comentarios (en cursiva) sobre lo que se ha aprendido haciendo las pruebas. Es la experiencia que se va adquiriendo usando ObjectP.

Las pruebas son muy similares a la adaptación de los agregados.

- **agregado_str**: OK. El agregado es un tipo primitivo. Puede parecer absurdo ya que este caso suele manejarse con los atributos pero una HashSet<String> se adapta a HashSet<PString> convirtiendo el tipo primitivo en agregado.

  Se aprecian los típicos problemas de sincronización en cascada. Trabajar con agregados es trabajar con un conjunto de instancias. Cada una necesita sincronizarse. Hay que seguir una metodología para no perder información.

- **agregado_objp**: OK. No se realizan adaptaciones pero hay que observar que el flujo del código funciona bien devolviendo lo mismo que lo enviado.

  En la clase ‘ClaseAgr’ los atributos se pueden inicializar con un valor pero los agregados se deben inicializar a null pues de lo contrario se creará una relación de composición innecesaria.

- **agregado_array_1d_character:** OK. El agregado es un array de una dimensión, un array de Character. El contenido del array, los Character, se deben adaptar a PCharacter.

  Al trabajar con arrays se deben inicializar las dimensiones de alguna manera. Si se usa el método ‘objpAgregadoSetAgregadoItems’ que recibe el array está inicialización es automática. Pero si se usa el método ‘objpAgregadoSetAgregadoItem’ (sin la ‘s’ al final) lo que se está haciendo es dar un único elemento para ser colocado en una posición del array. Aquí la inicialización debe realizarse manualmente con ‘objpAgregadoConfiguracionDimension’.

- **agregado_array_2d_string:** OK. El agregado es un array de dos dimensiones. El contenido del array, los String, se deben adaptar a PString.

- **agregado_array_3d_integer:** OK. El agregado es un array de tres dimensiones. El contenido del array, los Integer, se deben adaptar a PInteger.

- **agregado_hashset_string:** OK. El agregado es un hashset. El contenido del hashset, los String, se deben adaptar a PString.
- **agregado_treeset_integer:** OK. El agregado es un treeset. El contenido del treeset, los Integer, se deben adaptar a PInteger.
- **agregado_arraylist_character:** OK. El agregado es un arraylist. El contenido del arraylist, los Character, se deben adaptar a PCharacter.
- **agregado_linkedlist_integer:** OK. El agregado es un linkedlist. El contenido del linkedlist, los Integer, se deben adaptar a PInteger.
- **agregado_integer_string:** OK. El agregado es un hashmap. El contenido del hashmap, los Integer y los String, se deben adaptar a PInteger y PString.
- **agregado_string_integer:** OK. El agregado es un treemap. El contenido del treemap, los Integer y los String, se deben adaptar a PInteger y PString.
- **agregado_array_1d_objp:** OK. El agregado es un array de una dimensión, concretamente un array de ClasePri. El contenido del array, los ClasePri, no se tienen que adaptar.

  El código en la plantilla para leer es más largo como ya sucedió en las pruebas de adaptación. Esto es así porque no se admite el cast desde Object[][][] a ClasePri[][][].
- **agregado_array_2d_objp:** OK. El agregado es un array de dos dimensiones, concretamente un array de ClasePri. El contenido del array, los ClasePri, no se tienen que adaptar al guardar pero sí realizar un cast en la lectura.
- **agregado_array_3d_objp:** OK. El agregado es un array de tres dimensiones, concretamente un array de ClasePri. El contenido del array, los ClasePri, no se tienen que adaptar al guardar pero sí realizar un cast en la lectura.

Capítulo 7. Resultados y discusión

Este capítulo busca poner de relieve los puntos fuertes y débiles de ObjectP para conocer a qué tareas resultaría interesante aplicarlo y para qué tareas no aporta ninguna mejora.

El objetivo de ObjectP es ser una alternativa como herramienta de persistencia aunque tiene una serie de características que no tienen relación directa con la persistencia.

7.1. ObjectP para la persistencia

7.1.1. Consecución de los objetivos

ObjectP consigue el objetivo de abstraer al programador para que pueda escribir clases que resulten fáciles de hacer persistentes sin tenga que pensar en la manera en la que se realiza la persistencia.

JDO también consigue el objetivo aunque como se comentó en el capítulo de 'Tecnologías actuales' la separación no llega a ser total entre el QUÉ y el CÓMO aunque lo que falta es muy poco. También es cierto que a ObjectP, al mirar la plantilla, hay detalles que recuerdan a tablas relacionales y es que, como ya se mencionó, la separación cien por cien puede ser muy difícil de conseguir por lo que el objetivo más realista debería consistir en facilitar todo lo que se pueda la abstracción.

7.1.2. Facilidad de uso con respecto a otras herramientas

Otra cuestión a analizar son las mejoras que aporta ObjectP con respecto a JDO. Ahora mismo se podría decir que ObjectP puede llegar a ser algo más sencillo que JDO debido a que tiene menos elementos pero las necesidades a cubrir, ya sea con ObjectP o con JDO serán las mismas con lo que si ObjectP tiene menos elementos es porque está delegando tareas en la BDO así que lo que falta en ObjectP tal vez esté en la BDO.

7.1.3. Inconvenientes de ObjectP

La manera en la que ObjectP es llevado a la práctica tiene una serie de situaciones incómodas o poco elegantes

- **Heredar de ObjectP**: Tener que heredar de ObjectP es algo que en la práctica puede limitar mucho su uso generalizado.
- **Adaptación de agregados y arrays**: Es algo que queda reservado a la plantilla y que es invisible si se dispone de un mecanismo para generarla de manera automática. Aun así se es poco elegante

7.1.4. Conclusiones

La abstracción que proporciona ObjectP es muy similar a la que proporciona JDO. Además, JDO tiene a su favor el llevar años en el mercado por lo que ObjectP tendrá que ofrecer algo más que simple abstracción. Ese extra que ObjectP podría llegar a ofrecer es su capacidad de fusionarse con java.lang.Object. No se está diciendo que sea el ObjectP aquí creado el que deba fusionarse, es la idea de que sea Java de forma nativa quien facilite lo que facilita ObjectP. Actualmente Java ofrece la reflexión como una aproximación, tal vez sólo consista en potenciar este mecanismo para que resulte más sencillo de usar.

7.2. Características propias de ObjectP

Este trabajo se inició con la idea de ofrecer un mecanismo alternativo para la persistencia. En su desarrollo se apreció que ObjectP podría ser usado para tareas que nada tenían que ver con la persistencia. Por tanto, estas características no entran en competición con JDO. Es más, estas características, al estar basadas en la idea de que sólo se usarán instancias de ObjectP, elimina

toda la problemática relacionada con heredar de ObjectP, adaptar agregados o integrar ObjectP con java.lang.ObjectP. Ahora ObjectP es una clase más para ser usada como cualquier otra clase cuya principal funcionalidad es la de ser capaz de representar cualquier estructura de datos.

El análisis de estas características requiere de un nuevo trabajo, aquí se muestran las ideas de manera preliminar.

7.2.1. Atributos y agregados dinámicos

La clase Empleado puede venir con los atributos nombre, apellido y teléfono. Pero tal vez para una necesidad concreta se necesite un atributo adicional, como teléfono2. ObjectP permite añadirle ese atributo sin necesidad de tener que crear una clase nueva.

La idea anterior, llevada al extremo, consiste en crear una instancia de ObjectP y configurarla individualmente. Tal vez haya situaciones en las que se considere una molestia tener que programar clases, sobre todo clases en las que sólo se tienen métodos set/get.

7.2.2. Tipos de ObjectP

Aunque sólo exista una clase de ObjectP las instancias podrán tener significados y comportamientos diferentes. Un flag, por ejemplo, servirá para indicar si cierta instancia de ObjectP está representando a un tipo primitivo, que no puede tener agregados, o está representando una instancia que sí puede tener agregados. La figura 7.1 muestra un ejemplo de cómo las diferentes instancias de ObjectP se adaptan para representar a un tipo primitivo o a un tipo no primitivo.

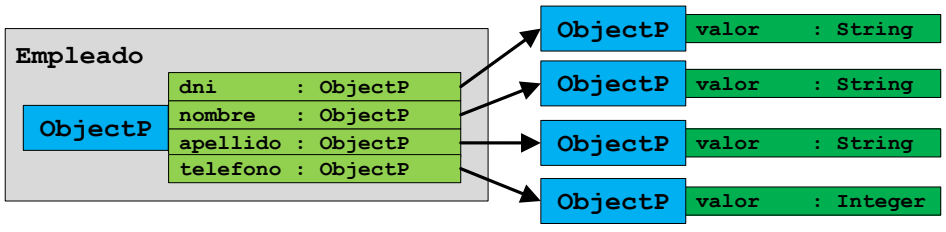

Fig. 7.1 Las instancias de ObjectP pueden tener comportamientos diferentes.

7.2.3. ObjectP para categorizar

Como no existe la clase Empleado y Departamento es necesario un mecanismo para poder hacer referencia a las instancias que son Empleado y diferenciarlas de las que son Departamento. Para esto es necesario adaptar ObjectP para que tenga un flag llamado 'es_categoría' de manera que se pueda crear un ObjectP, activarlo como categoría y llamarlo Empleado para que haga referencia a todas las instancias que son Empleado. Estos ObjectP para representar categorías pueden ser de tres tipos:

- Categorías 1: Son las más sencillas donde todo son procesos manuales. Se pide la creación de una categoría con cierto nombre y cuando se guarda una instancia se pide que sea referenciada por cierta categoría.

Creación de categorías	Manual
Representante	No existe
Asignación de instancias a categorías	Manual

- Categorías 2: Se definen representantes para las categorías. Un representante se construye como una condición (no se explica ahora cómo se construyen estas condiciones). De esta manera, al guardar una instancia, ésta se compara con todos los representantes de categorías y se crean las referencias de manera automática.

Creación de categorías	Manual
Representante	Manual
Asignación de instancias a categorías	Automático

- Categorías 3: En este caso la creación de los representantes es una tarea automática. Primero entra la información en la BDO, luego procesos automatizados crearían las categorías y luego las categorías harían referencias a las instancias que la han creado.

Creación de categorías	Automático
Representante	Automático
Asignación de instancias a categorías	Automático

Las categorías que más interesan son las número 3 para situaciones donde no se conoce a priori la forma que pueden tener las estructuras de datos.

- Supongamos un robot con sensores. Cada sensor tiene una rutina que crea instancias de ObjectP con lo percibido. Estas rutinas se puede decir que son como Empleado o Departamento, crean estructuras de datos prefijadas como puede ser línea, superficie o volumen. Pero a partir de aquí, la combinación de los volúmenes, para crear formas geométricas, es indeterminada y depende de la experiencia percibida.
- Habrá otras rutinas que cojan las estructuras de volúmenes para analizarlas y sacar conclusiones. Estas conclusiones son las categorías y es lo que permite diferenciar un árbol de una mesa.

No debe resultar muy complicado aplicar sobre ObjectP algoritmos de categorización o minería de datos cuyas conclusiones son a su vez otros ObjectP. ObjectP ofrece la posibilidad de realizar los procesos de categorización de otra manera.

Desconozco los algoritmos de categorización y los tipos de categorías que existen pero a continuación muestro una sugerencia de jerarquía de categorías por si pudiera ser aprovechada alguna de sus ideas. Lo que se persigue es representar los distintos conceptos que suele manejar la mente humana. El primer nivel es el que permite crear el concepto de 'silla' desde un punto de vista geométrico pero crear 'silla' como algo que sirva para sentarse requiere de categorías más evolucionadas.

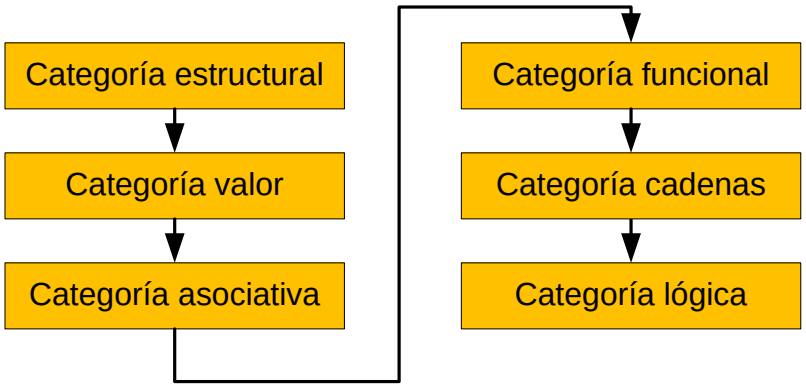

Fig. 7.2 Ejemplo de jerarquía de categorías.

Descripción breve de cada tipo de categoría:

- **Estructural o geométrica**: Sólo se fija en la forma. Es que la instancia de ObjectP tenga ciertos nombres de atributos o de agregados.
- **Valor**: Es crear restricciones a la estructura. Es asignar restricciones a los valores que pueden tomar los atributos o agregados.
- **Asociativa**: Buscan modelar 'A está a 10 metros de B'. También, si primero 'A está a 10 metros de B' y luego 'A está a 5 metros de B', se crear 'A está acercándose a B'.
- **Funcional**: Las categorías estructurales modelan sustantivos básicos como silla. Las asociativas modelan los verbos como acercare. Las funcionales generan un sustantivo en base a una categoría asociativa: de golpear crea golpeador o martillo.
- **Cadenas**: Las categorías de cadenas enlazan hechos con lo que es aquí cuando se crea el concepto de tiempo. Es un paso previo para crear la lógica.
- **Lógica**: La lógica natural es una lógica estadística: 'Si ocurre esto tantas veces entonces es probable que ocurra esto otro'. El 'SI ENTONCES' de programación es una estructura de datos como lo es 'silla', simplemente hay que recorrer un proceso concreto de captación y construcción de información.

7.2.4. ObjectP para crear un tipo diferente de red neuronal

Se parte de la estructura mostrada en la figura 7.3 donde se aprecian las instancias de ObjectP que son tipos primitivos y las que no son tipos primitivos. Los tipos primitivos están asociados a las neuronas receptoras.

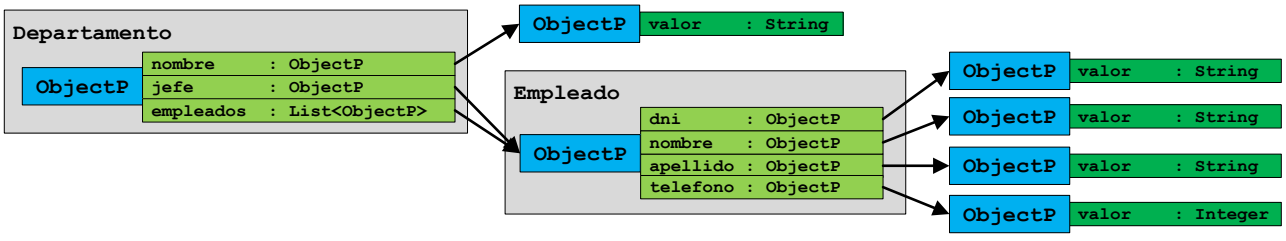

Fig. 7.3 Las instancias de ObjectP de tipo primitivo representan neuronas receptoras.

La característica de una neurona artificial, la del perceptrón por ejemplo, es su función de activación y sus pesos. La función de activación es el código y la información que procesa son números. Esto quiere decir que en este tipo de neuronas artificiales tiene más importancia el

código que los datos. Además, la estructura de datos queda difusa en el conjunto de la red neuronal, algo que suele resultar incómodo para el humano al gusta tener bien localizada la información.

En una posible red neuronal basada en ObjectP la estructura de datos no está difusa puesto que ObjectP está diseñado más para representar datos que código. De hecho, lo que le falta a ObjectP es el código, la función de activación. Analizar los parecidos, las diferencias y las posibles equivalencias entre estas redes neuronales podría ser un trabajo bastante interesante. Aquí se muestra cómo sería un posible comportamiento de la red neuronal basada en ObjectP.

- Para añadirle código a ObjectP se implementa Runable. Lo que se tiene ahora es una BDO viva o activa, no es un simple repositorio de información pasivo. Por su puesto, esta BDO sólo funciona en memoria.
- Cada nueva información modificará la BDO creando nueva información pues los algoritmos de categorización estarían programados en los hilos. Aunque dependerá del tipo de instancia de ObjectP para ejecutar un código u otro.
- Ahora imaginemos que tenemos información guardada en la BDO. Que ni siguiera existe la clase String como clase primitiva, que la clase primitiva es Character y cada cadena de texto es una lista de caracteres.
- La BDO tiene referencias en los dos sentidos (el dpto. apunta a sus empleados pero cada empleado sabe a qué dpto. pertenece). La búsqueda de una cadena, o incluso subcadena, es similar a cómo el cerebro identifica una melodía. Según se van activando las neuronas receptoras, los caracteres en este caso, cada instancia de ObjectP que representa un carácter informa a las instancias que le referencian.
- Las instancias de String que referencian a los caracteres van cogiendo la secuencia de caracteres que se van produciendo. Si se completa la cadena a la que representa se activa la función de activación y transmite el impulso a las instancias que la referencian. Una de esas instancias es un observador, una instancia que apunta a todas las instancias, algo así como el gestor de la BDO que necesita conocer lo que sucede en todo momento.
- Es un mecanismo de búsqueda masiva en paralelo. Se lanza la secuencia de caracteres ‘Margarita’ lo que hace que se active la instancia de string que la recoge. Y como este string es a su vez referenciado por un Empleado y una Planta éstas también se activa aunque de otra manera. No produce la misma activación si se recibe toda la información que si se recibe parte. ¿Qué departamentos tienen a alguien que se llama ‘Margarita’?

- Hará falta que la información introducida por las neuronas receptoras lleve una marca temporal así como un toquen. La marca temporal es para no acumular indefinidamente información percibida, que caduque y se deseche. El toquen es para hacer búsquedas en paralelo: los caracteres asociados a la cadena 'Margarita' llevarán un toquen diferente a la cadena 'Violeta' para no mezclar. Se propaga el toquen de manera que el observador sabe en base a qué toquen (consulta realizada) se activa cada hilo de las instancias ObjectP.
- Será interesante analizar cómo se debe adaptar ObjectP para ser ejecutado en una GPU ya sí aprovechar su altísimo paralelismo.

7.2.5. Conclusión

ObjectP como mecanismo de persistencia puede resultar poco interesante por tener competidores como JDO además de que su implementación se ve dificultada debido a que para que resultase práctica debería integrarse con java.lang.Object. En cambio, las características propias de ObjectP permiten experimentar probando a realizar de otra manera ciertas tareas actuales.

Capítulo 8. Conclusiones y trabajo futuro

El trabajo realizado, la implementación de ObjectP, debería ser analizado por la idea que representa más que por el desarrollo realizado. La idea es la de una clase que facilita la persistencia al hacer de intermediaria. La limitación que es la obligación de tener que heredar de esta clase se convierte en virtud si se fusiona con java.lang.Object.

Es necesario que ObjectP ofrezca características diferenciadoras, ahora mismo es bastante similar a JDO. Se deben continuar los desarrollos para analizar si ciertas características de ObjectP pueden llegar a ser elementos claramente diferenciadores.

Primero se describen los posibles pasos a dar para convertir el conjunto ObjectP+BDO en un producto real. Luego comenta como seguir evolucionando ObjectP para añadirle funcionalidades con el objetivo de trabajar sólo con instancias ObjectP.

8.1. Desarrollo de ObjectP como herramienta de persistencia

La figura 8.1 ilustra una línea de desarrollos que buscan convertir a ObjectP en una herramienta alternativa para la persistencia. Hay dos líneas de desarrollo independientes.

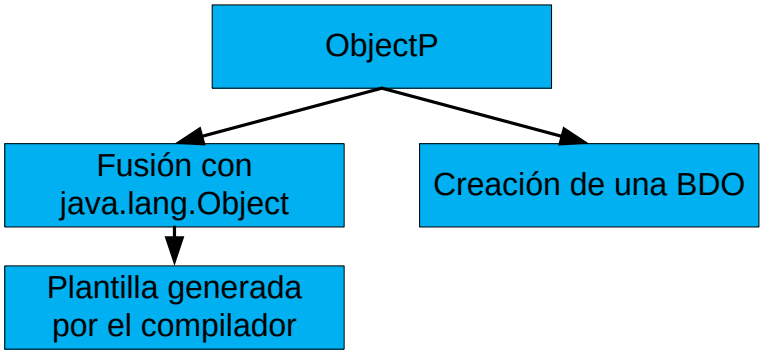

Fig. 8.1 Desarrollos futuros para que ObjectP sea una herramienta para la persistencia.

1. Buscar la máxima integración con Java fusionando ObjectP con java.lang.Object y creando un compilador que construya la plantilla. Tener que heredar de ObjectP crea limitaciones que se deben solucionar.
2. Desarrollar una BDO que permita trabajar con ObjectP.
3. También sería interesante analizar cómo puede ser ObjectP en C++ donde se permite la herencia múltiple.

8.2. Desarrollo de ObjectP para representar estructura de datos

Otra línea de desarrollos busca potenciar el uso de sólo instancias de ObjectP (`objp = new ObjectP()`). Esta línea tiene diferentes fases siendo la primera la creación de categorías debido a que si se trabaja sólo con instancias ObjectP no existe la clase Empleado ni la clase Departamento, tan sólo instancias de ObjectP. Algunas instancias de ObjectP representarán a empleados y otras a departamentos por lo que es necesario un mecanismo para hacer referencia a los distintos conjuntos de instancias para recuperar lo perdido al trabajar sólo con instancias de ObjectP.

Estos elementos, las categorías, vienen a decir: 'las instancias que tengan tales características'. Esto es lo mismo que las consultas, otra tarea pendiente, la de la búsqueda de información en la BDO.

Capítulo 9. Referencias bibliográficas

1. Keith M, Schincariol M. (2009). Pro JPA 2. Mastering the Java Persistence API. Apress.
2. Linwood J, Minter D. (2010). Beginning Hibernate. An introduction to persistence using Hibernate 3.5. Apress.
3. Guruzu S, Mak G. (2010). Hibernate Recipes. A problem-solution approach. Apress.
4. Russell C. (2013). Java Data Objects 3.1. Sitio web: https://db.apache.org/jdo
5. Google. (2013). Google Cloud Platform (de JDO). Sitio web: https://cloud.google.com/appengine/docs/java/datastore/jdo/overview-dn2
6. Siegel E, Retter A. (2015). eXist. A NoSQL document database and application platform.
7. ObjectDB Software. (2013). ObjectDB 2.5 Developer's Guide. Sitio web: http://www.objectdb.com
8. DataNucleus. (2015). DataNucleus AccessPlatform. v.4.0. Sitio web: http://www.datanucleus.org
9. Bellido A. (2012). Primera implementación de ObjectP+BDO. Sitio web: http://bbddoorr.blogspot.com.es/

Anexo 1. Sugerencia para una BDO

A1.1. La BDO debe personalizar las instancias de ObjecP

Una instancia de ObjectP asociada a la clase 'Empleado' sólo contiene información sobre los atributos y agregados de la clase Empleado. Pero la BDO podría necesitar elementos adicionales para marcar cada instancia como considere oportuno. Uno de estos elementos podría ser 'alfa', la clave primaria que la BDO asignaría a cada instancia.

Para que la BDO pueda personalizar las instancias y mantenga un control sobre ellas, el programador ya no podrá escribir '`new Empleado()`', deberá pedirle la instancia a la BDO para que la inicialice como considere oportuno. Algunos de los atributos que la BDO podría necesitar para gestiones internas podrían ser:

- **Alfa**: Es la clave primaria que usa la BDO, el usuario no puede definirla. Esta clave es la que se usa para relacionar tablas entre ellas. En la plantilla existe la opción de marcar un atributo como ID pero al ser opcional la BDO no puede contar con él. Además, cada ID puede ser de un tipo diferente mientras que resulta más fácil usar siempre el mismo.
- **Versión**: Para entornos donde hay varias máquinas conectadas a una misma BDO. La BDO puede devolver la misma instancia a varias máquinas. Luego, una de las máquinas realiza modificaciones y la guardar lo que incrementa la versión. Si luego otra máquina intenta hacer lo mismo la BDO deberá avisar del riesgo que supone.

A1.2. Tablas para objetos

Las primeras tablas son para las clases creadas por el usuario. Estas tablas almacenarán los atributos, las variables de tipo primitivo. Unos ejemplos se muestran en la figura A1.1.

Empresa		
Alfa	Nombre	CP
m01	Medusa Systems	28020

Departamento		
Alfa	Nombre	Ubicación
d01	Sistemas	1º Planta
d02	Desarrollo	Sótano

Empleado			
Alfa	DNI	Nombre	Apellidos
e01	11111	Azrael	Álvarez
e02	22222	Benita	Bonita
e03	33333	Carlos	Criado
e04	44444	Diana	Díaz
e05	55555	Ernesto	Elías
e06	666666	Fabiola	Fernández
e07	77777	Gabriel	Gómez

Libro	
Alfa	Título
L01	Cinco semanas en globo
L02	Canticos de la lejana tierra

Dar
Alfa
a01
a02
a03

Secretario		
Alfa	Extensión	ppm
e02	22	220
e04	44	440
e06	66	660

Bar	
Alfa	Nombre
b01	Manoletes

Fig. A1.1 Ejemplo de tablas para objetos de usuario sólo con atributos primitivos

Un ejemplo significativo es la clase 'Dar' que no tiene atributos de tipo primitivo por lo que sólo tiene la columna de 'alfa'. Las referencias a otros objetos están en otras tablas.

Es interesante la manera en la que se maneja la herencia. Cuando una instancia está implicada en la herencia, como 'e02' que es una instancia de 'Secretario' que hereda de 'Empleado', se usa el mismo alfa para indicar que todos los atributos forman parte de una misma instancia. Es la manera más sencilla de manejar la herencia. Esta BDO no usa claves externas.

A1.3. Tablas de referencias.

Las tablas de referencias mantienen las relaciones entre clases. Por ejemplo, dicen qué empleados pertenece a cada departamento. Son tablas del tipo M-M con lo que las dos columnas principales son el alfa del apuntador y el alfa del apuntado; apuntador es el departamento y apuntado es el empleado.

El nombre de la tabla será: 'NombreClaseApuntador_NombreAgregadoEnClaseApuntador'.

Estas tablas son diferentes dependiendo del tipo de colección que sea el agregado. ObjectP da soporte para Object, HashSet, TreeSet, ArrayList, LinkedList, HashMap y TreeMap.

- **Object**: Se maneja con la tabla básica M-M.

Departamento_jefe	
Apuntador	Apuntado
d01	e02
d02	e03

Fig. A1.2 Tabla de referencias para un agregado que no tiene colección.

- **Set**: Las dos implementaciones de Set se manejan con la tabla M-M. Esta vez se puede repetir el apuntador pero no la pareja apuntador-apuntado ya que un conjunto no puede tener elementos repetidos.

Apuntador_agregado	
Apuntador	Apuntado
d01	a1
d01	a2
d02	b1
d02	b2

Fig. A1.3 Tabla de referencias para Sets.

- **ArrayList**: Es necesario una columna adicional para almacenar el número del índice. La clave sería el alfa apuntador y el índice. Se permite repetir elementos apuntados.

Departamento_empleados		
Apuntador	Idx	Apuntado
d01	0	e04
d01	1	e05
d02	0	e06
d02	1	e07

Fig. A1.4 Tabla de referencias para ArrayList.

- **LinkedList**: La implementación realizada usa cinco columnas. Nodo, NodoAnterior, NodoSiguiente, Apuntado y Apuntado. La clave primaria es Nodo y Apuntador. Se permite repetir elementos apuntados.

Departamento_empleados				
NodoAnt	Nodo	NodoSig	Apuntador	Apuntado
NULL	nodo01	nodo02	d01	e04
nodo01	nodo02	nodo03	d01	e05
nodo02	nodo03	nodo04	d02	e06
nodo03	nodo02	NULL	d02	e07

Fig. A1.5 Tabla de referencias para LinkedList.

- **Map**: Por sencillez las dos implementaciones se manejan igual. La tabla binaria inicial de M-M aquí se amplía a una ternaria M-M-M, el alfa del apuntador, el alfa de la clave apuntada y el alfa del valor apuntado. Esto es así porque ObjectP generaliza y la clave del map es un

ObjectP y no un tipo primitivo. La clave primaria sería el alfa del apuntador y el alfa de la clave del map.

Departamento_empleados		
Apuntador	KeyStr	Apuntado
d01	e01	e03
d01	e05	e07
d02	e02	e04

Fig. A1.6 Tabla de referencias para los Maps.

- **Array**: Es necesario añadir tantas columnas como dimensiones tenga el array. Otra opción es una única columna donde las coordenadas sea una lista de números separados por comas.

Clase_agregado				
Apuntador	X_0	X_1	X_2	Apuntado
	0	0	0	
	0	0	1	
	0	0	2	
	0	1	0	
	0	1	1	
	0	1	2	
	0	2	2	

Fig. A1.7 Tabla de referencia para un array de tres dimensiones.

A1.4. Tablas para lectura de instancias

Las tablas anteriores son las básicas para guardar la información de las instancias pero para recuperarla son necesarias tablas adicionales. El proceso de lectura sugerido es el siguiente:

- **Crear la instancia**: Es necesario conocer el nombre de la clase para poder crear una instancia usando '`Class.forName()`' donde luego se vuelca la información que lea.
- **Personalizar Atributos**: ObjectP proporciona dinamismo a nivel de atributos. Esto quiere decir que la configuración por defecto descrita por la clase puede ser modificada, se pueden añadir o quitar atributos. Esto obliga a tratar cada instancia de manera individualizada.
- **Personalizar Agregados**: Igual que con los atributos sucede con los agregados. Es una tabla diferente porque la información para los agregados no es la misma que para los atributos.

NOTA: El dinamismo a nivel de atributos invalida las tablas para los objetos vistas en A1.1. Más adelante se describe cómo se modifican para soportar dinamismo a nivel de atributos.

A1.4.1. Tabla para el nombre de la clase de cada instancia

Para cada instancia se tiene el nombre de su clase. Las columnas que indican la cantidad de atributos y la cantidad de agregados ayudan a leer las tablas siguientes ya que si el valor no es cero se deberán mirar otras tablas.

IndicePrincipal			
Alfa	Clase	CantidadAtributos	CantidadAgregados
m01	Empresa	2	2
d01	Departamento	2	3
d02	Departamento	2	3
e01	Empleado	3	0
e02	Secretario	5	0
e03	Empleado	3	0

Fig. A1.8 Tabla que informa sobre el nombre de la instancia

A1.4.2. Tabla para reconstruir los atributos

Por cada atributo de una instancia habrá una fila. Cada fila informa de:

- El nombre de la clase donde se definió. Por ejemplo, ‘e02’ es Secretario pues así lo dice la tabla anterior pero tiene atributos definidos en Empleado y en Secretario.
- Debido a que los nombres de los atributos se pueden repetir cuando hay herencia es necesario añadir una columna adicional, Idx, para completar la clave primaria.
- Además del nombre del atributo hará falta almacenar otras columnas adicionales. Aquí sólo se ha mostrado ‘Tipo’ pero podrían hacer falta más.

Después de que la BDO cree la instancia debe comprobar si los atributos por defecto son como los que se especifican en la tabla A1.9. Si no lo son deberá realizar correcciones.

AtributosDeCadaInstancia				
Alfa	Clase	Idx	Atributo	Tipo
m01	Empresa	0	CP	Integer
m01	Empresa	0	Nombre	String
d01	Departamento	0	Nombre	String
d02	Departamento	0	Nombre	String
e01	Empleado	0	DNI	String
e01	Empleado	1	Nombre	String
e01	Empleado	2	Apellidos	String
e02	Secretario	0	Extensión	Integer
e02	Secretario	1	ppm	Integer
e02	Empleado	2	DNI	String
e02	Empleado	3	Nombre	String
e02	Empleado	4	Apellidos	String

Fig. A1.9 Tabla que informa sobre los atributos de cada instancia

A1.4.3. Tabla para reconstruir los agregados

Igual que para los atributos, por cada agregado de la instancia habrá una fila. Las columnas de la clave primaria es la misma pero el resto de las columnas cambian porque la información para los agregados es diferente que para los atributos. El nombre de la tabla de referencia se construye con el nombre de la clase y el nombre del agregado.

AgregadosDeCadaInstancia					
Alfa	Clase	Idx	Agregado	Colección	Dimensiones
m01	Empresa	0	departamentos	Lst	
m01	Empresa	1	jefe	Lst	
d01	Departamento	0	jefe	Lst	
d01	Departamento	1	empleados	Lst	
d01	Departamento	1	empleados2	Map	
d02	Departamento	0	jefe	Lst	
d02	Departamento	1	empleados	Lst	
d02	Departamento	1	empleado2	Map	
				Array	20,20,5

Fig. A1.10 Tabla que informa sobre los agregados de cada instancia

A1.5. Tablas para borrado de instancias (referencias inversas)

Cuando se borran instancias se deben actualizar referencias. Si se borra un empleado, el departamento que le apunta para indicar que le pertenece debe dejar de apuntarle. Por tanto, cada empleado debe saber quién le está apuntado.

Borrar de la BDO puede hacerse borrando instancias directamente o sólo referencias.

- **Referencias**: Se puede borrar la referencia entre un departamento y un empleado para que el empleado deje de pertenecer al departamento pero no se borra al empleado.
- **Instancias**: Borrar la instancia de un empleado implica borrar todas las referencias que le apunten.

A1.5.1. Borrado de referencias

Es un proceso sencillo ya que sólo hay que eliminar filas de las tablas de referencias. El resto de tablas no se ven afectadas. No hacen falta tablas adicionales para borrar referencias.

A1.5.2. Borrado de instancias

Para poder borrar una instancia primero hay que borrar todas las referencias que le llegan y para saber qué referencias le llegan hacen falta tablas adicionales. Hay dos tipos de tablas:

- **ReferenciadoPor_Cantidad**: Indica cuántas referencias le llegan a cada instancia. Tal vez esta tabla no sea estrictamente necesaria pero ayuda. El nombre de la tabla es 'NombreClaseApuntada_ReferenciadoPorCantidad'.

Empleado_ReferenciadoPorCantidad	
Apuntado	Cantidad
e01	3
e02	3
e03	3
e04	3
e05	3
e06	1
e07	2

Departamento_ReferenciadoPorCantidad	
Apuntado	Cantidad
d01	1
d02	1

Fig. A1.11 Tablas ReferenciadoPor_Cantidad

- **ReferenciadoPor**: Esta tabla se puede considerar que es la tabla de referencias vista en el punto A1.3 pero a la inversa. Cuando la colección del agregado es un maps es necesario decir desde qué columna se apunta, si es desde la clave o desde el valor. El nombre de la tabla es 'NombreClaseApuntada_ReferenciadoPor'.

  La clave primaria la compone el alfa del apuntado junto con un número ya que una instancia puede ser apuntada desde múltiples sitios. Luego está el alfa apuntador, la tabla de referencia desde la que apunta y el campo de la tabla desde que el que apunta; esto último sólo es de interés cuando el agregado apuntador es un map.

La tabla de la figura A1.12 contiene Empleados que son apuntados desde varias tablas. Por ejemplo, ‘e06’ sólo es apuntado por ‘d02’ desde la tabla ‘Departamento_empleados’ (el carácter separador ya decidirá la BDO si usa ‘_’, ‘$’ u otro).

Empleado_ReferenciadoPor				
Apuntado	Idx	Apuntador	Agregado / TablaRef	Desde
e01	0	m01	Empresa$presidente	Apuntado
e01	2	d01	Departamento$empleados2	KeyStr
e02	0	d01	Departamento$jefe	Apuntado
e02	2	d02	Departamento$empleados2	KeyStr
e05	0	d01	Departamento$empleados	Apuntado
e05	2	d01	Departamento$empleados2	KeyStr
e06	0	d02	Departamento$empleados	Apuntado
e07	0	d02	Departamento$empleados	Apuntado
e07	0	d01	Departamento$empleados2	Apuntado

Fig. A1.12 Tablas ReferenciadoPor

El proceso de borrar una instancia:

- Usando la tabla de ReferenciadoPor se borran las filas de las tablas de referencias.
- Luego se borran filas de las tablas de ReferenciadoPor.
- Por último se borra la instancia borrando la fila correspondiente de la tabla de su clase.

Se debe tener especial cuidado cuando hay elementos repetidos en la colección. Por ejemplo, que un ‘Departamento’ tuviera al mismo empleado varias veces en su lista de ‘empleados’ y que sólo se quisiera borrar una de las referencias.

A1.5.3. Recolector de basura / Objetos empotrados

Esta BDO no reconoce el concepto de objeto empotrado. Por ejemplo, una dirección asociada a un empleado es una instancia de la clase ‘Direccion’ y podría ser referenciada por otros empleados. Borrar al empleado no implicará borrar la dirección. Cuando se borra una instancia no se borra información asociada a ella de manera automática, se deberá indicar explícitamente.

A1.6. Atributos dinámicos

ObjectP permite que los atributos definidos por defecto por la clase puedan ser modificados. Las primeras tablas que se vieron fueron las tablas que tienen una columna por atributo. Pero esto sólo funciona si todas las instancias de la clase tienen siempre los mismos atributos. Como este ya no es el caso se debe crear una solución. Hay dos alternativas para dar soporte a los atributos dinámicos.

A1.6.1. Tablas independientes para los atributos

Lo que se hace es crear tablas de dos columnas, una para 'alfa' y otra para el atributo. El nombre de la tabla es 'NombreClase_NombreAtributo' como muestra la figura A1.13.

Empleado_DNI	
Alfa	DNI
e01	11111
e02	22222
e03	33333
e04	44444
e05	55555
e06	666666
e07	77777

Empleado_Nombre	
Alfa	Nombre
e01	Azrael
e02	Benita
e03	Carlos
e04	Diana
e05	Ernesto
e06	Fabiola
e07	Gabriel

Empleado_Apellidos	
Alfa	Apellidos
e01	Álvarez
e02	Bonita
e03	Criado
e04	Díaz
e05	Elías
e06	Fernández
e07	Gómez

Fig. A1.13 Tablas para atributos dinámicos.

A1.6.2. Atributos como agregados

Los atributos se manejan como si fueran agregados. Esto implica que no hay columnas en las tablas de los objetos a excepción de la columna de alfa tal y como se muestra en la figura A1.14

Empresa
Alfa
m01

Departamento
Alfa
d01
d02

Empleado
Alfa
e01
e02
e03
e04
e05
e06
e07

Secretario
Alfa
e02
e04
e06

Fig. A1.14 Las tablas de los objetos quedan reducidas sólo la columna del alfa.

Esto obliga a que los tipos primitivos se manejen como objetos, que tengan su propia tabla como la tiene Empleado. Sólo que en este caso sí hay una segunda columna para el valor. La figura A1.15 muestra un ejemplo de tablas con los valores primitivos.

PSstring	
Alfa	valor
str10	Carlos
str11	Diana
str13	Fabiola
str14	Gabriel
str18	Díaz
str19	Elías
str22	Sistemas
str23	Desarrollo
str25	Sótano
str26	Medusa

PInteger	
Alfa	valor
int01	22
int02	44
int03	66
int04	220
int05	440
int06	660
int07	28020

Fig. A1.15 Las tablas de los valores primitivos son las únicas con valores.

Entiendo que esta es la manera donde la base de datos aporta el mayor conocimiento. Permite que la cadena 'Margarita' nunca se duplique y que sea referenciada desde Persona y desde Flor, por ejemplo.

En realidad, String tampoco debería ser considerado tipo primitivo, debería ser Character y String tener un agregado de tipo ArrayList<Character>.

A1.10. Implementación y eficiencia

Lo más llamativo de esta arquitectura es la cantidad de tablas que son necesarias para almacenar poca información. Las bases de datos tradicionales buscan realizar pocos accesos y traer la mayor cantidad de información mientras que aquí sucede al revés, muchos accesos que traen muy poca información sobre todo si se usa el extremo de manejar los atributos como agregados.

Muchos accesos que mueven poca información es algo que recuerda al cerebro: *Cada pequeña tabla es una neurona. Las conexiones entre tablas con las conexiones neuronales. Los tipos primitivos son las neuronas receptoras.*

Una característica de las redes neuronales es su alto grado de paralelismo. Los accesos a disco son muy secuenciales con lo que esta arquitectura debe potenciar CPUs+RAM ya que permite un grado mucho mayor de paralelismo. Las bases de datos actuales tienen el disco duro como su repositorio principal pero usan de forma auxiliar la memoria. Esta BDO sugiere hacerlo al revés, usar la memoria como elemento principal y el disco duro como auxiliar.