Genericidad

La programación orientada a objetos tiene como principal objetivo favorecer la confiabilidad, reusabilidad y extensibilidad del software. Adoptar el enfoque propuesto por la programación orientada a objetos implica:

- En la etapa de diseño reducir la complejidad en base a la descomposición del problema en piezas más simples, a partir de un conjunto de clases relacionadas entre sí.
- En la etapa de implementación utilizar un lenguaje que permita retener las relaciones entre las clases y encapsular su representación interna.

La abstracción de datos y el encapsulamiento permiten que una o más clases usen los servicios provistos por otra clase considerando solo su funcionalidad, sin tener en cuenta cómo la implementa. La herencia permite aumentar el nivel de abstracción mediante un proceso de clasificación en niveles.

La extensibilidad reduce el impacto de los cambios. Las modificaciones con frecuencia pueden resolverse definiendo nuevas clases específicas, sin necesidad de cambiar las que ya han sido desarrolladas y verificadas. La reusabilidad evita escribir el mismo código repetidamente acelerando el proceso de desarrollo.

La genericidad favorece la extensibilidad y reusabilidad. Los datos de aplicaciones muy diferentes pueden modelarse con estructuras que serán creadas, modificadas y procesadas sin considerar el tipo de las componentes.

La genericidad puede modelarse de dos maneras diferentes: usando **polimorfismo paramétrico** o usando **herencia**. El polimorfismo paramétrico en Java es limitado y tiene varias restricciones, pero es posible definir clases genéricas usando herencia.

Clases Genéricas

> Una **clase genérica** es aquella que encapsula a una estructura cuyo comportamiento es independiente del tipo de las componentes.

Por ejemplo, es posible definir una clase `Fila` que encapsula un arreglo de componentes y brinda un método `esCreciente()` que retorna verdadero sí y solo sí las componentes están ordenados de forma creciente. Esto es, dada la clase:

```
class Fila {
private Elemento [] f;
public Fila(int n){
   f = new Elemento[n];}
…
}
```

La implementación del método puede ser:

```
public boolean esCreciente (){
/*Requiere que todas las posiciones del arreglo estén ligadas y que
el arreglo tenga al menos dos componentes*/
  int i=0; boolean es=true;
  while (i<f.length-1 && es)
    es = f[i].menor(f[i+1]); i++;
  return es;}
```

El código del método `esCreciente` es independiente del tipo de las componentes de la fila `f`, en tanto la clase `Elemento` brinde el servicio `menor`. Si la clase `Elemento` brinda además el servicio `igual`, es posible definir un método genérico en la clase `Fila` para contar la cantidad de elementos del arreglo equivalentes a `e`.

```
public int contar (Elemento e){
/*Requiere que todas las posiciones del arreglo estén ligadas*/
  int cont =0;
  for (int i=0;i<f.length;i++)
   if (f[i].igual(e)) cont++;
  return cont;}
```

El método `contar` es independiente del tipo de las componentes, siempre que la clase `Elemento` brinde el servicio `igual`.

La clase `Fila` puede definirse entonces como:

```
class Fila {
private Elemento [] f;
public Fila(int n){
   f = new Elemento[n];}
public void insertar (Elemento e, int p){
//Asume que p es una posición válida
  f[p] = e;}
public boolean esCreciente (){
/*Requiere que todas las posiciones del arreglo estén ligadas y
contiene al menos dos componentes*/
int i=0; boolean es=true;
while (i<f.length-1 && es){
  es = f[i].menor(f[i+1]); i++;  }
return es;}
public int contarElementos (Elemento e){
/*Requiere que todas las posiciones del arreglo estén ligadas*/
  int cont =0;
  for (int i=0;i<f.length;i++)
   if (f[i].igual(e)) cont++;
  return cont;}
}
```

Un objeto de clase `Fila` brinda servicios `insertar`, `esCreciente` y `contar`. Ninguno de los tres servicios depende del tipo de las componentes. De hecho es posible usar `Fila` para crear estructuras con componentes de clase `PresionArterial` o `Racional`. Sin embargo, es necesario definir una clase `Elemento` con servicios `igual` y `menor`.

La implementación de estos servicios no se conoce en el momento de crear la clase `Elemento`. Son métodos abstractos y por lo tanto la clase `Elemento` es abstracta.

```
abstract class Elemento {
abstract boolean igual(Elemento e);
abstract boolean menor (Elemento e);}
```

Las clases `PresionArterial` y `Racional` extienden a `Elemento` e implementan los métodos abstractos, de manera consistente con la aplicación.

Por ejemplo, en `PresionArterial`, igual retorna `true` sí y solo sí los dos objetos que se comparan computan el mismo valor para el pulso.

```
class PresionArterial extends Elemento {
//Valores representados el milímetros de mercurio
//Atributos de clase
private static final int umbralMax=120;
private static final int umbralMin=80;
//Atributos de instancia
private int maxima;
private int minima;
//Constructores
public PresionArterial(int ma,int mi){
//Requiere ma > mi
  maxima = ma;
  minima = mi;}
//Consultas
…
public int obtenerPulso(){
  return maxima-minima;}
public boolean igual(Elemento e){
    return obtenerPulso() == ((PresionArterial) e).obtenerPulso();}
public boolean menor(Elemento e){
    return obtenerPulso() < ((PresionArterial) e).obtenerPulso ();}
}
```

La clase `PresionArterial` extiende a la clase abstracta `Elemento` e implementa los métodos `igual` y `menor`, considerando que la comparación se realizada de acuerdo al pulso. Para que los servicios `igual` y `menor` queden redefinidos en la clase derivada, el parámetro debe ser de clase `Elemento`.

Si el parámetro de uno de estas consultas fuera de clase `Racional`, obviamente en la clase `Racional`, el servicio quedaría sobrecargado y no redefinido. El compilador reportaría entonces un error porque las clases `Racional` y `PresionArterial` no implementan el método abstracto.

En un método de una clase `Hospital` que modele el registro de mediciones de presión arterial de un paciente, la clase `Fila` puede usarse como sigue:

```
class Hospital{
…
Fila registro = new Fila (3);
registro.insertar(new PresionArterial (7,13));
registro.insertar(new PresionArterial (7,14));
registro.insertar(new PresionArterial (7,15));
if (registro.esCreciente())
 …
}
```

Los tres elementos de la estructura `registro` son de clase `PresionArterial`. Dos objetos de clase `PresionArterial` son **comparables**, de modo que es perfectamente válido comparar el primero valor con el segundo, el segundo con el tercero y así siguiendo. La comparación se realiza usando el servicio `menor` provisto por la clase `PresionArterial`. Como `menor` redefine al método definido en la clase `Elemento`, el parámetro tiene que ser de tipo `Elemento`, aun cuando lo que se van a comparar son dos objetos de clase `PresionArterial`. El casting justamente asegura que el mensaje `obtenerPulso()` va a poder ligarse a un método provisto para `e`.

La implementación parcial de `Racional` es:

```
class Racional extends Elemento{
//El denominador es siempre mayor a 0
private int numerador,denominador;
…
public Racional(int n,int d){
//Requiere d > 0
  numerador = n;
  denominador = d;}
public int obtenerNumerador(){
  return numerador;}
public int obtenerDenominador(){
  return denominador;}
public boolean igual(Elemento e){
    Racional r = (Racional) e;
    return numerador*r.obtenerDenominador() ==
           r.obtenerNumerador()*denominador ; }
public boolean menor(Elemento e){
    Racional r = (Racional) e;
    return numerador*r.obtenerDenominador() <
           r.obtenerNumerador()*denominador ; }
}
```

Nuevamente, el tipo del parámetro en los servicios `igual` y `menor` debe ser `Elemento`, de modo que el método quede redefinido. Como el método `obtenerDenominador()` no está definido para la clase `Elemento`, es necesario garantizar que va a estar provisto para el parámetro `r`.

La clase `Fila` puede ser usada entonces, por ejemplo, en una aplicación matemática:

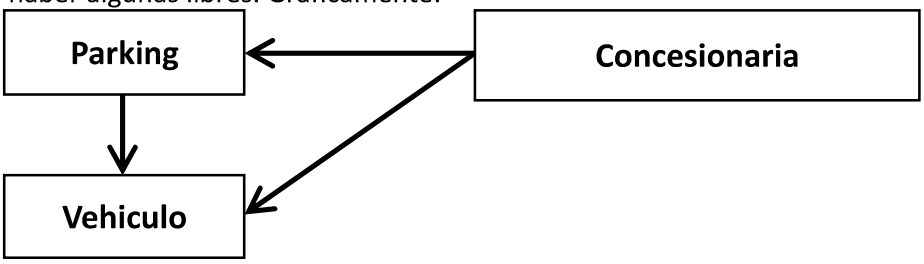
```
class Simulador{
…
Racional r1,r2,r3,r4;
…
Fila f = new Fila (4);
f.insertar(r1);
f.insertar(r2);
f.insertar(r3);
f.insertar(r4);
if (f.esCreciente())
 …
}
```

Aunque `f` es una estructura genérica, el casting relaja el control sin provocar riesgos; en la aplicación `Simulador` los elementos de un objeto de clase `Fila` son todos de clase `Racional`. Análogamente en la aplicación `Hospital`, todos los elementos del objeto `registro` son de clase `PresiónArterial`.

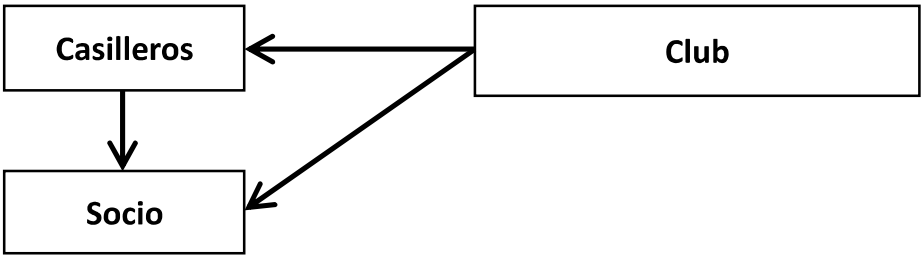
Generalización y Reuso

En los capítulos anteriores se han presentado patrones de algoritmos que podían reusarse en aplicaciones muy diferentes. Al analizar el diagrama de clases de distintos sistemas es posible observar que un conjunto de servicios brindan una funcionalidad equivalente, de modo que se favorece el reuso definiendo una clase genérica que los incluya.

A continuación se retoma el caso de estudio referido a una concesionaria de autos que dispone de un estacionamiento en el cual se ubican los vehículos en venta. La clase `Parking` encapsula una **tabla** de elementos de tipo `Vehiculo`, representada a través de un arreglo parcialmente ocupado. Cada vehículo se asigna a una plaza particular del estacionamiento, aunque puede haber algunas libres. Gráficamente:



También es posible retomar el caso de un club que dispone de un conjunto de casilleros que alquila a sus socios para guardar sus pertenencias. Cada casillero se asigna a un socio particular, aunque puede haber algunos libres. Cada socio puede alquilar varios casilleros. Gráficamente:



Los diagramas de las clases `Parking` y `Casilleros` son:

Parking
<<Atributos de instancia>> T [] Vehiculo
<<constructor>> Parking (max : entero) <<comandos>> estacionar(v:Vehiculo, p:entero) estacionar(v:Vehiculo) salir (p : entero) salir (v : Vehiculo) <<consultas>> maxPlazas() : entero plazasOcupadas ():entero plazaOcupada(p:entero):boolean hayPlazaOcupada():boolean estaLleno() : boolean estaVehiculo(v:Vehiculo):boolean existePlaza(p:entero):boolean recuperaVehiculo(p:entero): Vehiculo valuacionTotal():real menorKm():Vehiculo cantModelo(mod:entero):entero

Casilleros
<<Atributos de instancia>> T [] Socio
<<constructor>> Casilleros (max : entero) <<comandos>> alquilar(s:Socio,p:entero) alquilar(s:Socio) liberar(p : entero) liberar(s : Socio) <<consultas>> maximo() : entero ocupados ():entero estaOcupado(p:entero):boolean hayCasilleroOcupado():boolean todoLleno() : boolean tieneCasillero(s:Socio):boolean existeCasillero(p:entero):boolean recuperaSocio(p:entero): Socio cantCasilleros(s:Socio):entero

En la clase `Parking` la funcionalidad de los servicios es:

`estacionar(v:Vehiculo, p:entero)` asigna el vehículo `v` a la plaza `p` que asume válida y libre.

`estacionar(v:Vehiculo)` asigna el vehículo `v` a la primera plaza libre.

`salir (p : entero)` asigna nulo a la plaza `p` que asume válida.

`salir (v : Vehiculo)` busca una plaza ligada al vehículo `v` y le asigna nulo.

`maxPlazas() : entero` computa la cantidad de plazas del estacionamiento .

`plazasOcupadas ():entero` computa la cantidad de plazas ocupadas del estacionamiento.

`plazaOcupada(p:entero):boolean` computa true si en la plaza `p` está estacionado un vehículo, asume `p` válida.

`hayPlazaOcupada():boolean` computa true si al menos una plaza está ocupada.

`estaLleno() : boolean` computa true si todas las plazas están ocupadas.

`estaVehiculo(v:Vehiculo):boolean` computa true si el vehículo está estacionado en una plaza.

`existePlaza(p:entero):boolean` computa true si `p` es una plaza del estacionamiento.

`recuperaVehiculo(p:entero): Vehiculo` retorna el vehículo estacionado en la plaza `p`, que asume una plaza válida.

`valuacionTotal():real` computa el valor que resulta de computar la suma de los valores de todos los vehículos.

`menorKm():Vehiculo` retorna el vehículo estacionado con menor kilometraje

`cantModelo(mod:entero):entero` retona la cantidad de vehículos estacionados del modelo `mod`.

La clase `Consesionaria` define un atributo de instancia de clase `Parking` gestionar el estacionamiento:

```
class Consesionaria {
//Atributos de instancia
private Parking e;
…
public void gestionarPlazas(){

Vehiculo v;
int p;
…
if (e.existePlaza(p) && !e.plazaOcupada(p))
  e.estacionar(v,p);
…
Vehiculo menor= e.menorKm();
}
…
}
```

En la clase `Casilleros` la funcionalidad de los servicios es:

`asignar(s:Socio, p:entero)` asigna el socio `s` al casillero `p` que asume válido y libre.

`asignar(s:Socio)` asigna el socio `s` al primer casillero libre.

`salir (p : entero)` asigna nulo al casillero `p` que asume válido.

`salir (s : Socio)` busca los casilleros asignados al socio `s` y les asigna nulo.

`maximo() : entero` computa la cantidad total de casilleros.

`casillerosOcupados():entero` computa la cantidad de casilleros ocupados.

`casilleroOcupado(p:entero):boolean` computa true si el casillero `p` está asignado a un socio, asume `p` válido.

`hayCasilleroOcupado():boolean` computa true si al menos un casillero está ocupado.

`estaLleno() : boolean` computa true si todos los casilleros están ocupados.

`tieneCasillero(s:Socio):boolean` computa true si el socio `s` tiene al menos un casillero asignado.

`existeCasillero(p:entero):boolean` computa true si `p` denota un casillero válido.

`recuperaSocio(p:entero): Socio` retorna el socio que tiene asignado el casillero `p`, asumiendo `p` válido.

`cantCasilleros (s:Socio): entero` computa la cantidad de casilleros ligados al socio `s`.

Una implementación parcial de la administración de los casilleros en la clase `Club` puede ser:

```
class Club{
//Atributos de instancia
private Casilleros c;
…
public void administrar(){

Socio s;
int p;
…
if (c.existeCasillero(p) && !e.casilleroOcupado(p))
  c.alquilar(s,p);
…
int cant = c.cantCasilleros(s);
}
…
}
```

Se puede observar que algunos de los servicios de la clase `Casillero` tienen una funcionalidad equivalente a un servicio de `Parking`.

Al analizar la implementación del comando `estacionar`:

```
public void estacionar (Vehiculo v, int p) {
/*Asigna el vehiculo v a la plaza p, requiere que exista la plaza y
esté libre*/
  T[p] = v;   }
```

se observa que aunque se trata de aplicaciones diferentes, el comando `alquilar` brinda la misma funcionalidad:

```
public void alquilar (Socio s, int p) {
/*Asigna el socio s al casillero p, requiere que exista el casillero
y esté libre*/
  T[p] = s;   }
```

En Java toda clase, en particular `Socio` y `Vehiculo`, hereda implícitamente de la clase `Object`. Como el código de `estacionar` y de `alquilar` no depende del tipo de las componentes, puede generalizarse definiendo un comando que reciba un parámetro de clase `Object`:

```
public void insertar (Object e, int p) {
/*Asigna el elemento e a la posición p, requiere que exista la
posición y esté libre*/
  T[p] = e;   }
```

Análogamente, el método `estacionar` inserta un vehículo en la primera plaza libre:

```
public void estacionar (Vehiculo v) {
/*Busca la primera plaza libre y asigna v. La clase Cliente es
responsable  de controlar que la tabla no esté llena y v no sea
nulo*/
  int i = 0;
  while (T[i] != null)
    i++;
  T[i] = v;}
```

Tiene una funcionalidad equivalente al servicio `alquilar` que asigna el socio `s` al primer casillero libre:

```
public void alquilar (Socio s) {
/*Busca el primer casillero libre y asigna el socio s. La clase
Cliente es responsable  de controlar que la tabla no esté llena y s
no sea nulo*/
  int i = 0;
  while (T[i] != null)
    i++;
  T[i] = s;}
```

La solución puede generalizarse para cualquier parámetro de clase `Object` ya que el código no depende del tipo de las componentes:

```
public void insertar (Object e) {
/*Busca la primera posición libre y asigna e. La clase Cliente es
responsable  de controlar que la tabla no esté llena y e no sea
nulo*/
  int i = 0;
  while (T[i] != null)
    i++;
  T[i] = e;}
```

Si las búsquedas comparan la **identidad de los elementos** se puede generalizar el comportamiento de la consulta `estaVehiculo` en la clase `Parking`:

```
public boolean estaVehiculo (Vehiculo v){
/*Decide si el vehículo v está estacionado en alguna plaza */
  int i = 0; boolean esta = false;
  while (i < T.length && !esta ){
    esta = T[i] != null && T[i] == v ;
    i++; }
```

```
  return esta;}
```

También se puede generalizar la consulta `tieneCasillero` en la clase `Casilleros`.

```
public boolean tieneCasillero (Socio s){
/*Decide si el socio s tiene asignado algún casillero  */
  int i = 0; boolean esta = false;
  while (i < T.length && !esta ){
    esta = T[i] != null && T[i] == s ;
    i++; }
  return esta;}
```

Los métodos `estaVehiculo` y `tieneCasillero` brindan una funcionalidad equivalente.

Es importante considerar cuidadosamente la funcionalidad porque puede haber variaciones. En la concesionaria cada vehículo puede estar asociado a lo sumo a una plaza, en el club en cambio un socio puede estar asignado a varios casilleros. El código:

```
public void salir(Vehiculo v){
/*Busca una plaza asignada al vehículo y si existe asigna nulo.
Asume que v está ligado*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)esta = T[i] == v ;
   else  i++; }
 if (esta) T[i] = null;}
```

Es análogo a la consulta `tieneCasillero` en la clase `Casilleros`:

```
public void liberar(Socio s){
/*Busca todos los casilleros asignados al socio s y les asigna
nulo. Asume que v está ligado*/
int i = 0; boolean esta=false;
while (i < T.length ){
  if (T[i] == s) T[i] = null;
  i++;}
}
```

De modo que estos comandos no son generalizables.

Servicios Generales de una clase

Una clase genérica brinda un conjunto de servicios generales cuya implementación es independiente del tipo de las componentes de la estructura. La clase genérica puede ser usada para modelar diferentes aplicaciones.

A partir de las dos aplicaciones presentadas en la sección anterior, se propone el diseño de una clase genérica `Tabla` de acuerdo al siguiente diagrama:

Tabla
T [] Object
<<constructor>> Tabla (max : entero) <<comandos>> insertar (elem:Object,p:entero) insertar (elem:Object) eliminar (p : entero) <<consultas>> maxElem() : entero

```
cantElem():entero
estaLlena () : boolean
hayElem():boolean
estaElem (elem : Object) : boolean
posicionOcupada(p:entero):boolean
existePosicion(p:entero):boolean
recuperaElem(p:entero): Object
```

Con la siguiente funcionalidad:

`insertar(elem:Object, p:entero)` asigna el elemento `elem` a la posición `p` que asume válida y libre.

`insertar(elem:Object)` asigna el elemento `elem` a la primera posición libre.

`eliminar (p : entero)` asigna nulo a la posición `p` que asume válida.

`maxElem() : entero` computa la cantidad de posiciones de la tabla.

`cantElem(p:entero):entero` computa la cantidad de posiciones ligadas.

`posicionOcupada(p:entero):boolean` computa true si la posición `p` está ligada, asume `p` válida.

`hayPosicionesOcupadas():boolean` computa true si al menos una posición está ocupada.

`estaLlena() : boolean` computa true si todas las posiciones de la tabla están ocupadas.

`existeElem(elem:Object):boolean` computa true si el elemento `elem` está asignado al menos a una posición.

`existePosicion(p:entero):boolean` computa true si `p` denota una posición válida en la tabla.

`recuperaElem(p:entero): Object` retorna el elemento asignado a la posición `p`, que asume válida.

Y la implementación es:

```
class  Tabla {
protected Object[] T;
//Constructor
public Tabla(int max){
   T = new Object[max];}
//Comandos
public void insertar(Object elem, int p) {
/*Asigna elem a la posicion p, requiere que p sea válida y esté
libre */
    T[p] = elem;}
public void eliminar(int p){
/*Elimina el elemento de la posicion p, requiere que p sea una
posición válida*/
    T[p] = null;}
//Consultas
public int maxElem(){
  return T.length;}
public int cantElem(){
//Retorna la cantidad de referencias ligadas
  int cant=0;
```

```
  for (int i=0; i<=T.length;i++)
    if (T[i] != null)cant++;
  return cant;}
public boolean posicionOcupada(int i){
   return T[i] != null;}
public boolean hayPosicionOcupads(){
/*Retorna true sí y solo sí al menos una referencia está ligada*/
int i = 0; boolean hay=false;
while (i < T.length && !hay){
   if (T[i] != null)hay = true ;
   i++;}
return hay;}
public boolean estaLlena(){
int i = 0; boolean hay=false;
while (i < T.length && !hay){
   if (T[i] == null) hay = true ;
   i++;}
return !hay;}
public boolean estaElemento(Object elem){
/*Retorna true sí y solo si existe una posición ligada a elem*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)
     esta = elem == T[i] ;
   i++;}
return esta;}
public boolean existePosicion (int p) {
/*Retorna true sí y solo sí p denota una posición válida de la
tabla*/
    return p >= 0 && p < T.length;}
public Object recuperarElemento(int p) {
/* Retorna el elemento que ocupa la posición p en la tabla Requiere
que p sea una posición válida*/
    return T[p];}
}
```

El conjunto de servicios provistos por la clase `Tabla` es independiente del tipo de las componentes.

Servicios Específicos de una clase

En muchas aplicaciones es posible reusar una clase genérica, pero es necesario extenderla con otra que brinde el comportamiento específico.

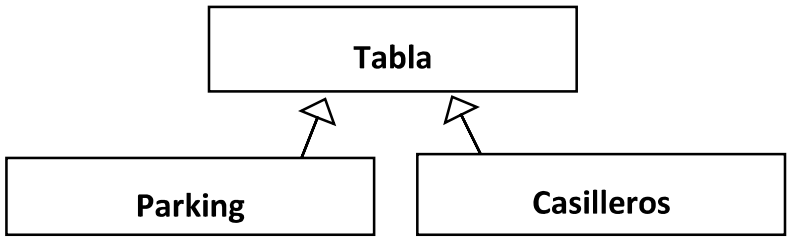
Algunos de los servicios provistos por la clase `Parking` son específicos de la aplicación:

- La salida de un vehículo del estacinamiento.
- El cómputo de cuántos vehículos mantiene el estacionamiento.
- El cómputo de la valuación total de los vehículos del estacionamiento.
- La determinación del vehículo con menor kilometraje.

De manera análoga, la clase `Casilleros` ofrece algunos servicios específicos, cuyo código depende del tipo de las componentes:

- La liberación de todos los casilleros de un socio.
- El cómputo de la cantidad de casilleros asignados a un socio.

Las clases `Parking` y `Casilleros` extienden a la clase `Tabla`:



Las clases derivadas definen los servicios que son específicos de cada aplicación y heredan los servicios generales provistos por la clase base `Tabla`:

Parking
<<constructor>> Parking (max : entero) <<comandos>> salir (v : Vehiculo) <<consultas>> valuacionTotal():real menorKm():Vehiculo cantModelo(mod:entero):entero

Casilleros
<<constructor>> Casilleros (max : entero) <<comandos>> liberar(s : Socio) <<consultas>> cantCasilleros(s:Socio):entero

En este caso, ninguna de las clases especializadas define atributos. Cada uno de los servicios brinda la funcionalidad descripta anteriormente a partir del acceso y recorrido de la estructura definida en la clase genérica.

La implementación de la clase `Casilleros` es entonces:

```
class Casilleros extends Tabla {
public Casilleros (int max){
  super(max);}
public void liberar (Socio s){
/*Asigna nulo a todos los casilleros ligados al socio s*/
int i = 0;
while (i < T.length){
   if (T[i] == s)
     T[i] = null ;
   i++;}}
public int cantCasilleros (Socio s){
```

```
/*Computa la cantidad de casilleros asociados al socio s*/
int i = 0; int cont = 0;
while (i < T.length){
   if (T[i] != null && T[i] == s)
     cont++ ;
   i++;}
return cont;}
}
```

Como antes, la clase `Club` define un atributo de instancia de clase `Casilleros` y modelará la administración de casilleros.

```
class Club{
//Atributos de instancia
private Casilleros c;
…
public void administrar(){

Socio s;
int p;
…
if (c.existeCasillero(p) && !c.casilleroOcupado(p))
  c.alquilar(s,p);
…
int cant = c.cantCasilleros(s);
}
…
}
```

El objeto ligado a la variable `c` recibe los mensaje `alquilar` y `cantCasilleros`, el primero definido en la clase base `Tabla` y el segundo en la clase derivada `Casilleros`. El cambio en el diseño de la clase `Casilleros` es transparente para la clase `Club`. La definición de una clase genérica permite reusar el código en diferentes aplicaciones, en este caso la implementación de una tabla.

Problemas Propuestos

1. Implemente el siguiente diagrama de clases que modela una colección genérica de elementos:

Coleccion
T [] TipoElemento cant:entero
<< Constructores>> Coleccion (max:entero) <<Comandos>> insertar(c:TipoElemento) eliminar(c:TipoElemento) quickSort() <<Consultas>> estaLlena():boolean hayElementos():boolean cantElementos():entero existePosición(p:entero):boolean

*TipoElemento
*equals(t:TipoElemento):boolean *greater(t:TipoElemento):boolean

recuperarElemento(p:entero):TipoElemento estaElemento(c:TipoElemento):boolean	

De acuerdo a la siguiente especificación:

`Coleccion (max:entero)` Crea una colección para max elementos.

`insertar(c:TipoElemento)` inserta el elemento c en la primera posición libre e incrementa cant. Asume que la estructura no está llena y c está ligada y no hay un elemento equivalente en la colección.

`eliminar(c:TipoElemento)` busca un elemento equivalente a c. Si existe y ocupa la posición p, asigna cada elemento de la posición i, con p<i<cant a la posición i-1 y decrementa cant.

`quickSort()` ordena los elementos de la estructura aplicando el método recursivo quicksort.

`estaLlena():boolean` retorna true si cant es igual a la cantidad de elementos del arreglo.

`hayElementos():boolean` retorna true si cant es mayor a 0.

`cantElementos():entero` retorna cant.

`existePosición(p:entero):boolean` retorna true si 0<=p<cant.

`recuperarElemento(p:entero):TipoElemento` retorna el elemento asignado a la posición p, si p no es válida retorna nulo.

`estaElemento(c:TipoElemento):boolean` retorna true si algún elemento de la colección es equivalente a c, asume c ligada.

2. Dadas las clases Coleccion y TipoElemento definidas antes

a) Implemente la clase Empleado extendiendo TipoElemento y la clase NominaEmpleados extendiendo a Coleccion a partir del siguiente diagrama:

Empleado
<<atributos de instancia>> legajo: entero nombre:String cantHoras: entero valorHora: real
<<Constructor>> Empleado(leg:entero, nombre:String,canth:entero, valorh:real) <<Consultas>> obtenerLegajo():entero obtenerSueldo():real obtenerCantHoras():entero obtenerValorHoras():real obtenerNombre():String equals(t:TipoElemento):boolean greater(t:TipoElemento):boolean

NominaEmpleados
contarSupHoras(h:entero): entero sumarSueldos (): real

contarSupHoras(h:entero):Cuenta la cantidad de empleados que trabajan más de h horas.

obtenerSueldo(): Computa la cantidad de horas por el valor de la hora.

Los métodos equals y greater comparan el legajo.

b) *Implemente la clase Técnico extendiendo Empleado y redefiniendo el método obtenerSueldo() computando el sueldo de cualquier empleado más un 15%.*
c) *Defina una clase tester que cree una nómina de empleados, inserte 12 Empleados, algunos de clase Tecnico y compute la suma de sus sueldos.*

3. *Implemente el siguiente diagrama de clases:*

ColeccionOrdenada
T [] Elemento cant:entero
<< Constructores>> ColeccionOrdenada (max:entero) <<Comandos>> insertar(c:Elemento) eliminar(c: Elemento) <<Consultas>> estaLlena():boolean hayElementos():boolean cantElementos():entero existePosición(p:entero):boolean recuperarElemento(p:entero):Elemento estaElemento(c: Elemento):boolean intercarlar(c:ColeccionOrdenada):ColeccionOrdenada

*Elemento
*equals(t: Elemento):boolean *greater(t: Elemento):Boolean

De acuerdo a la siguiente especificación:

`Coleccion (max:entero)` Crea una colección para `max` elementos.

`insertar(c:TipoElemento)` inserta el elemento `c` manteniendo el orden la estructura e incrementa `cant`. Asume que la estructura no está llena y `c` está ligada y no hay un elemento equivalente en la colección.

`eliminar(c:TipoElemento)` busca un elemento equivalente a `c`. Si existe y ocupa la posición p, asigna cada elemento de la posición i, con p<i<`cant` a la posición i-1 y decrementa cant.

`estaLlena():boolean` retorna true si `cant` es igual a `cantElementos()`.

`hayElementos():boolean` retorna true si `cant` es mayor a 0.

`cantElementos():entero` retorna `cant`.

`existePosición(p:entero):boolean` retorna true si 0<=p<`cant`.

`recuperarElemento(p:entero):TipoElemento` retorna el elemento asignado a la posición p, si p no es válida retorna nulo.

`estaElemento(c:TipoElemento):boolean` retorna true si algún elemento de la colección es equivalente a `c`, asume `c` ligada y usa búsqueda binaria.

`intercarlar(c:ColeccionOrdenada):ColeccionOrdenada` retorna una colección generada intercalando ordenadamente la colección que recibe el mensaje y la que pasa como parámetro.

4. Dado el siguiente diagrama de clase:

Matriz
<<atributos de instancia>> M: TipoElemento[][]
<<constructor>> Matriz(nf,nc: entero) <<Comandos>> establecer (f,c:entero,e: TipoElemento) <<Consultas>> obtener (f,c:entero):TipoElemento pertenece(e: TipoElemento):boolean contarElem(e:TipoElemento):entero equals(m: Matriz) : boolean

a. *Implemente la clase genérica Matriz usando un arreglo.*
b. *Implemente una clase MatrizRacionales que extienda a Matriz y agregue un servicio* `esIdentidad()`. *El constructor de la clase recibe como parámetro el nombre de un archivo e inicializa la matriz de racionales con los valores leídos del archivo.*
c. *Implemente una clase MatrizPixels que extienda a la clase Matriz y agregue un servicio* `todosGrises()` *que decida si los pixels de la matriz representan algún matiz del color gris. El constructor de la clase recibe como parámetro el nombre de un archivo e inicializa la matriz de pixels con los valores leídos del archivo.*
d. *Implemente una clase tester para cada clase.*