Herencia y Polimorfismo

En un modelo basado en clases las entidades se agrupan a partir de sus semejanzas y diferencias, establecidas en función de algún criterio. Una vez establecido el criterio es posible decidir si una entidad **pertenece** a una clase.
En una reserva ecológica un criterio de clasificación natural para la fauna y la flora es por especie. Podemos identificar así, las clases Puma, Águila, Hornero, Tero, Liebre, Piquillín y Algarrobo, cada una de las cuales agrupa a un conjunto de entidades semejantes entre sí.
Tanto un puma como una liebre son mamíferos y como tales comparten algunos atributos físicos y parte de su comportamiento, en particular la manera de reproducirse. La definición de clases independientes Puma y Liebre no permite modelar las semejanzas. La definición de una única clase Mamíferos que agrupe a los pumas y las liebres, no permite modelar las diferencias.
Un modelo más natural es entonces definir un segundo nivel de clasificación, agrupando clases a partir de sus semejanzas y diferencias. El mismo criterio que se aplica para decidir si una entidad pertenece a una clase, se puede usar para determinar si una clase está **incluida** dentro de otra.
Las clases Puma y Liebre pueden agruparse en una clase más general llamada Mamífero. El criterio de clasificación es la forma de reproducción. Las clases Águila, Hornero y Tero se agrupan de acuerdo al mismo criterio en la clase Ave. Las clases Mamífero y Ave pueden agruparse en una clase más general llamada Vertebrados.
La clase Puma está incluida en la clase Mamífero, toda entidad que pertenece a la clase Puma, pertenece también a la clase Mamífero. Análogamente, la clase Liebre está incluida en la clase Mamífero y toda instancia de Liebre es también instancia de Mamífero. Claramente, una instancia de Liebre NO es instancia de la clase hermana Puma.
Observemos que la relación de pertenencia vincula a una entidad con una clase. La relación de inclusión vincula a una clase con otra. Una entidad que pertenece a una clase, pertenece también a cualquier clase que la incluya.
La clasificación en niveles es una habilidad natural para el ser humano y permite modelar la relación de **herencia** entre clases. Una clase específica hereda las propiedades de las clases más generales, en las cuales está incluida.
El modelo de la reserva puede incluir, entre otras, las clases Ser Vivo, Animal, Planta, Vertebrado, Mamífero, Ave, Árbol, Arbusto, etc. Las clases Animal y Planta heredan de la clase Ser Vivo. Las clases Mamífero y Ave heredan de Vertebrado, que hereda de Animal y por lo tanto también de Ser Vivo.
Todo objeto de clase Mamífero es también una instancia de la clase Animal y también de Ser Vivo y como tal hereda sus atributos y comportamiento. Así la clase Ser Vivo define solo lo que es compartido por todos los seres vivos.

En la programación orientada a objetos la **herencia** es el mecanismo que permite crear clasificaciones que modelan una relación de **generalización-especialización** entre clases. Las clases especializadas heredan atributos y comportamiento de las clases generales y agregan atributos y comportamiento específico. Una clase especializada puede también redefinir el comportamiento establecido por su clase más general.

El concepto de herencia está fuertemente ligado al de polimorfismo. El **polimorfismo** es la capacidad que tiene una entidad para diferenciarse de otras que pertenecen a su misma clase, exhibiendo propiedades o comportamiento específico.

En la reserva natural la clase Animal establece que cada una de sus instancias se puede *trasladar*, pero la manera de trasladarse puede variar según la clase específica. Una entidad de clase Ave puede volar para trasladarse y está habilidad no es propia de una instancia de la clase Mamífero. Así, es posible establecer que todas las instancias de la clase Animal exhiben el comportamiento trasladar, pero especificar diferentes *formas* de trasladarse, según la clase específica.

En la programación orientada a objetos el polimorfismo permite que un mismo nombre pueda quedar ligado a objetos de diferentes clases y que objetos de distintas clases puedan recibir un mismo mensaje y cada uno actúe de acuerdo al comportamiento establecido por su clase.

En el desarrollo de un sistema de software la **herencia** es un recurso importante porque favorece la **reusabilidad** y la **extensibilidad**. El polimorfismo también favorece la productividad porque permite que un mismo nombre pueda asociarse a abstracciones diferentes, dependiendo del contexto.

Atributos y comportamiento compartido

La organización de un sistema en clases permite agrupar objetos a partir de los atributos y el comportamiento compartido. Con frecuencia algunas entidades de un problema comparten algunos atributos y comportamiento y difieren en otros. La definición de una colección de clases en la cual cada objeto pertenece exclusivamente a una clase, resulta insuficiente en estos casos, como ilustra el siguiente caso de estudio.

Caso de Estudio: Máquina Expendedora

Una fábrica produce dos tipos diferentes de máquinas expendedoras de infusiones, M111 y R101. Cada máquina tiene un número de serie que la identifica. Las máquinas del tipo M111 preparan ***café****,* ***café con leche****,* ***té****,* ***té con leche*** *y* ***submarino****. Tienen depósitos para los siguientes ingredientes:* ***café****,* ***té****,* ***leche*** *y* ***cacao****. Las máquinas de tipo R101 preparan* ***café*** *y* ***café carioca****. Tienen depósitos para* ***café****,* ***crema*** *y* ***cacao****.*

Los depósitos tienen las siguientes capacidades máximas:

Café	**1500**
Té	**1000**
Leche	**600**
Cacao	**600**
Crema	**600**

Además de la capacidad máxima de cada ingrediente, cada máquina mantiene el registro de la cantidad disponible.

Cuando se habilita una máquina se establece su número de serie y las cantidades disponibles comienzan con el valor máximo de cada ingrediente. La cantidad disponible aumenta cuando se carga el depósito con un ingrediente específico y disminuye cada vez que se prepara una infusión. El aumento es variable, aunque nunca se puede superar la capacidad máxima de cada depósito. Si el valor que se intenta cargar, sumado al disponible, supera al máximo, se completa hasta el máximo y retorna el sobrante. Cada máquina recibe un mantenimiento mensual de modo que se guarda el mes y el año del último mantenimiento.

Cada vez que se solicita una infusión se reducen los ingredientes de acuerdo a la siguiente tabla:

	Café	Café con leche	Té	Submarino	Té con leche	Café carioca
Café	40	30				30
Cacao				40		10
Té			35		20	
Leche		20		50	20	
Crema						30

Observemos que las máquinas de los modelos M111 y R101 comparten algunos atributos y difieren en otros. Si el diseñador agrupa las entidades en clases el diagrama sería:

M111	R101
<<atributos de clase>> maxCafé : entero maxTe : entero maxCacao : entero maxLeche : entero <<atributos de instancia>> nroSerie:entero ultMnt:MesAnio cantCafé : entero cantTe : entero cantCacao : entero cantLeche : entero	<<atributos de clase>> maxCafé : entero maxCacao : entero maxCrema : entero <<atributos de instancia>> nroSerie:entero ultMnt:MesAnio cantCafé : entero cantCacao : entero cantCrema : entero
<<constructor>> M111(n:entero) <<comandos>> cargarCafe(grs: entero) :entero cargarCacao(grs: entero): entero cargarTe(grs: entero): entero cargarLeche (grs : entero) : entero cafe() te() cafeConLeche() teConLeche() submarino() mnt(ma:MesAnio) <<consultas>> obtenerNroSerie():entero obtenerUltMnt():MesAnio obtenerCantCafe(): entero obtenerCantTe(): entero obtenerCantCacao(): entero obtenerCantLeche() : entero obtenerMaxCafe(): entero obtenerMaxTe(): entero obtenerMaxCacao(): entero obtenerMaxLeche() : entero vasosCafe() : entero vasosCafeConLeche() : entero vasosTe() : entero vasosTeConLeche() : entero vasosSubmarino () : entero masCafe(e:MaquinaExpendedora): MaquinaExpedendora	<<constructor>> R101(n:entero) <<comandos>> cargarCafe(grs: entero) :entero cargarCacao(grs: entero): entero cargarCrema (grs : entero) : entero cafe() carioca () mnt(ma:MesAnio) <<consultas>> obtenerNroSerie():entero obtenerUltMnt():MesAnio obtenerCantCafe(): entero obtenerCantCacao(): entero obtenerCantCrema() : entero obtenerMaxCafe(): entero obtenerMaxCacao(): entero obtenerMaxCrema() : entero vasosCafe() : entero vasosCarioca() : entero masCafe(e: MaquinaExpendedora): MaquinaExpedendora
La preparación de una infusion requiere que el depósito tenga los ingredientes necesarios	La preparación de una infusion requiere que el depósito tenga los ingredientes necesarios

Observamos que los atributos y el comportamiento compartido, se especifica en las dos clases. En la implementación, se duplicará una parte considerable del código. Si la fábrica produce 100 tipos diferentes de máquinas expendedoras y todas ofrecen café, parte del código se va a repetir en las 100 clases. Más aun, si se produce una modificación en los atributos o comportamiento común a todas las máquinas, es necesario realizar el cambio en cada clase. Por lo tanto, este modelo no es adecuado.

Un modelo alternativo puede ser:

```
MaquinaExpendedora
<<atributos de clase>>
maxCafé : entero
maxTe : entero
maxCacao : entero
maxLeche : entero
maxCrema:entero
<<atributos de  instancia>>
nroSerie:entero
ultMnt:MesAnio
cantCafé : entero
cantTe : entero
cantCacao : entero
cantLeche : entero
cantCrema: entero
<<constructor>>
MaquinaExpendedora(n:entero)
<<comandos>>
cargarCafe(grs: entero) :entero
cargarCacao(grs: entero): entero
cargarTe(grs: entero): entero
cargarLeche (grs : entero) : entero
cargarCrema (grs : entero) : entero
cafe()
te()
cafeConLeche()
teConLeche()
submarino()
carioca()
mnt(ma:MesAnio)
<<consultas>>
obtenerNroSerie():entero
obtenerUltMnt():MesAnio
obtenerCantCafe(): entero
obtenerCantTe(): entero
obtenerCantCacao(): entero
obtenerCantLeche() : entero
obtenerCantCrema() : entero
obtenerMaxCafe(): entero
obtenerMaxTe(): entero
obtenerMaxCacao(): entero
obtenerMaxLeche() : entero
obtenerMaxCrema() : entero
vasosCafe() : entero
vasosCafeConLeche() : entero
vasosTe() : entero
vasosTeConLeche() : entero
vasosSubmarino () : entero
vasosCarioca():entero
masCafe(e:MaquinaExpendedora):
MaquinaExpedendora
```

```
La preparación de una infusion requiere
que el depósito tenga los ingredientes
necesarios
```

En este caso la clase `MaquinaExpendedora` no modela a ningún objeto del problema porque incluye todos los atributos y servicios, los que corresponden a los modelos M111 y a R101. Cuando se crea un objeto del modelo M111 el atributo `cantCrema()` tendrá el valor 0 y no debería recibir el mensaje `carioca()`, aunque la clase brinda ese servicio. El modelo tampoco es adecuado.

Herencia

Una clase es un patrón que define los atributos y comportamiento de un conjunto de entidades. Dada una clase es posible definir otras más específicas que heredan los atributos y comportamiento de la clase general y agregan atributos y comportamiento especializado.

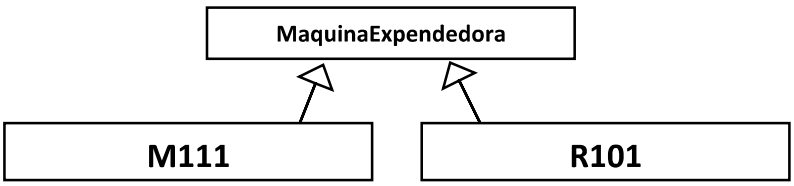
La **herencia** es un mecanismo que permite organizar la colección de clases de un sistema, estableciendo una relación de **generalización-especialización**.

Cuando el diseño propone dos o más clases que comparten algunos atributos y servicios y difieren en otros, es posible definir una **clase base** y una o más **clases derivadas.** La clase base específica los atributos y servicios compartidos. Las clases derivadas o subclases especializan a la clase base, heredan atributos y comportamiento de las clases generales y agregan atributos y comportamiento específico. Una clase derivada puede también redefinir el comportamiento establecido por su clase más general.

Un objeto pertenece a una clase si puede ser caracterizado por sus atributos y comportamiento. Una clase es **derivada** de una clase **base**, si todas sus instancias pertenecen también a la clase base.

La herencia es un recurso poderoso porque favorece la extensibilidad. Con frecuencia los cambios en la especificación del problema se resuelven incorporando nuevas clases especializadas, sin necesidad de modificar las que ya han sido implementadas, verificadas e integradas al sistema. La herencia facilita la reusabilidad porque no solo se reutilizan clases, sino colecciones de clases relacionadas a través de herencia.

En el caso de estudio propuesto una alternativa de diseño es factorizar los atributos y comportamiento compartidos de los modelos M111 y R101 en una clase general y retener los atributos y servicios específicos en clases especializadas.



La clase más general incluye atributos y comportamiento compartido:

```
MaquinaExpendedora
<<atributos de clase>>
maxCafé : entero
maxCacao : entero
<<atributos de  instancia>>
```

nroSerie:entero ultMnt:MesAnio cantCafé : entero cantCacao : entero
<<constructor>> MaquinaExpendedora(n:entero) <<comandos>> cargarCafe(grs: entero) :entero cargarCacao(grs: entero): entero cafe() mnt(ma:MesAnio) <<consultas>> obtenerNroSerie():entero obtenerUltMnt():MesAnio obtenerCantCafe(): entero obtenerCantCacao(): entero obtenerMaxCafe(): entero obtenerMaxCacao(): entero obtenerNroSerie():entero vasosCafe() : entero masCafe(e:MaquinaExpendedora): MaquinaExpedendora

Las clases `M111` y `R101` especifican los atributos y servicios específicos de estas máquinas:

M111
<<atributos de clase>> maxTe : entero maxLeche : entero <<atributos de instancia>> cantTe : entero cantLeche : entero
<<constructor>> M111(n:entero) <<comandos>> cargarTe(grs: entero): entero cargarLeche (grs : entero) : entero te() cafeConLeche() teConLeche() submarino() <<consultas>> obtenerCantTe(): entero obtenerCantLeche() : entero obtenerMaxTe(): entero obtenerMaxLeche() : entero vasosCafeConLeche() : entero vasosTe() : entero vasosTeConLeche() : entero vasosSubmarino () : entero masCafe(e:MaquinaExpendedora): MaquinaExpedendora

R101
<<atributos de clase>> maxCrema:entero <<atributos de instancia>> cantCrema: entero
<<constructor>> R101(n:entero) <<comandos>> cargarCrema (grs : entero) : entero carioca() <<consultas>> obtenerCantCrema() : entero obtenerMaxCrema() : entero vasosCarioca() : entero

Las clases `M111` y `R101` están vinculadas a la clase `MaquinaExpendedora` por una relación de **herencia**. La clase `MaquinaExpendedora` está asociada a `MesAnio`.

Diseño orientado a objetos

El diseño orientado a objetos consiste en definir una colección de clases relacionadas entre sí. Las clases pueden relacionarse a través de:

Asociación: permite modelar la relación ***tieneUn***, esto es, un atributo de instancia de una clase corresponde a otra clase.

Dependencia: permite modelar la relación ***usaUn***, es decir, los métodos de una clase reciben parámetros o declaran variables locales de otra clase.

Herencia: permite modelar la relación de generalización-especialización de tipo ***esUn***, las instancias de una clase derivada son también instancias de las clases de las cuales hereda.

La abstracción de datos permite reconocer aspectos comunes y relevantes en un conjunto de objetos para agruparlos en una clase general que los incluye a todos. La herencia aumenta el nivel de abstracción porque las clases son a su vez clasificadas a partir de un proceso de **generalización** o **especialización**. Si hablamos de **abstracción** cuando agrupamos objetos en clases, podemos llamar **superabstracción** al proceso de clasificar clases.

El proceso de clasificación puede hacerse partiendo de una clase muy general y descomponiéndola en otras más específicas identificando las diferencias entre los objetos. Si el proceso continúa hasta alcanzar subclases homogéneas, hablamos de **especialización**.

Alternativamente es posible partir del conjunto de todos los objetos y agruparlos en clases según sus atributos y comportamiento. Estas clases serán a su vez agrupadas en otras de mayor nivel hasta alcanzar la clase más general. Hablamos entonces de **generalización**.

Como en todos los casos de estudio propuestos en este libro, el diagrama de clases ya está elaborado, el objetivo es interpretarlo e implementarlo.

Herencia simple y herencia múltiple

Los lenguajes soportan el mecanismo de herencia de manera diferente, algunos de manera más compleja y flexible, otros brindan alternativas más simples pero menos poderosas.

Cuando la **herencia es simple** la clasificación es **jerárquica** y queda representada por un **árbol**. En este caso el proceso de clasificación se realiza de manera tal que cada subclase corresponde a una única clase base. Cada clase puede **derivar** entonces en una o varias **subclases** o **clases derivadas**, pero sólo puede llegar a tener una única **clase base**. La **raíz** del árbol es la clase más general, las hojas son las clases más específicas. El término **superclase** en ocasiones se usa para referirse a la raíz y otros autores lo utilizan como sinónimo de clase base.

Las clases **descendientes** de una clase son las que heredan de ella directa o indirectamente, incluyéndola a ella misma. Los **descendientes propios** de una clase son todos sus descendientes, excepto ella misma.

El conjunto de clases **ancestro** de una clase, incluye a dicha clase y a todas la que ocupan los niveles superiores en la misma rama del árbol que grafica la estructura de herencia. Los **ancestros propios** de una clase son todos sus ancestros, excepto ella misma.

Las **instancias** de una clase son los objetos que son instancia de alguna clase descendiente de dicha clase. Las **instancias propias** de una clase son los objetos de dicha clase.

La **herencia múltiple** permite que una clase derivada pueda heredar de dos o más clases más generales. Es una alternativa poderosa pero más compleja. Nuevamente las clases de los niveles superiores son más generales que las clases de los niveles inferiores. En los casos propuestos en este libro el diseño utiliza únicamente herencia simple.

En el diseño propuesto para modelar las máquinas expendedoras se establece una relación de herencia simple. La clase `MaquinaExpendedora` es la **clase base** de `M111` y `R101`, que son sus clases **derivadas**.

El vínculo entre la clase `M111` y `MaquinaExpendedora` es de tipo ***esUn***. Todo objeto de clase `M111` es también un objeto de clase `MaquinaExpendedora`.

En Java la palabra `extend` establece una relación de **herencia simple** entre clases. Así dada la implementación de la clase `MaquinaExpendedora`:

```
class MaquinaExpendedora {
…
}
```

Es posible definir la clase `M111` que extiende a `MaquinaExpendedora`:

```
class M111 extends MaquinaExpendedora {
…
}
```

Un objeto de clase `M111` estará caracterizado por todos los atributos y el comportamiento propio de la clase, pero además por todos los atributos y el comportamiento de la clase `MaquinaExpendedora`.

Análogamente:

```
class R101 extends MaquinaExpendedora {
…
}
```

Un objeto de clase `R101` es también una instancia de la clase `MaquinaExpendedora`.

Cuando el programador define una clase sin establecer su clase base, la nueva clase extiende implícitamente a la clase `Object`, provista por Java. Es decir, la clase `Object` es la raíz en la jerarquía de herencia. En este ejemplo `MaquinaExpendedora` extiende a `Object`.

Herencia y Encapsulamiento

La clase `MaquinaExpendedora` define los atributos compartidos por todas las máquinas:

```
class MaquinaExpendedora {
//atributos de clase
protected static final int maxCafe = 1500;
protected static final int maxCacao = 600;
//atributos de instancia
protected int nroSerie;
protected int cantCafe;
protected int cantCacao;
protected MesAnio ultMnt;
…
}
```

La clase `M111` define los atributos específicos de las máquinas que corresponden a ese modelo:

```
class M111 extends MaquinaExpendedora {
//atributo de clase
protected static final int maxTe = 1000;
protected static final int maxLeche = 600;
//atributos de instancia
protected int cantTe;
protected int cantLeche;
public M111 (int n){
…}
}
```

Como los atributos de la clase `MaquinaExpendedora` se declaran *protegidos*, sus clases derivadas tienen acceso a ellos. En particular el comando `cafeConLeche` en la clase `M111` puede acceder a sus propios atributos y también a los definidos en `MaquinaExpendedora`:

```
public void cafeConLeche(){
    cantLeche = cantLeche - 20;
    cantCafe = cantCafe – 30; }
```

Si el atributo `cantCafe` se hubiera declarado como privado, los métodos de la clase `M111` deberían acceder a él a través de las operaciones provistas por la clase `MaquinaExpendedora`, que debería incluir un método `retirarCafe(n)`, público o protegido.

```
public void cafeConLeche(){
    cantLeche = cantLeche - 20;
    this.retirarCafe(30); }
```

Existen diferentes criterios referidos al nivel de encapsulamiento que debería ligar a clases vinculadas por una relación de herencia.

Un argumento a favor de que las clases derivadas accedan a todos sus atributos, aun los que corresponden a las clases superiores en la jerarquía, es que una instancia de una clase específica, es también una instancia de las clases más generales, de modo que debería poder acceder y modificar su estado interno. El modificador de acceso `protected` permite que las clases derivadas accedan a los atributos de sus clases ancestro directamente.

El argumento en contra es que si se modifica la clase base, el cambio afectará a todas las clases derivadas que accedan directamente a la representación. Si alguno de los métodos de la clase `M111` accede al atributo `ultMnt` definido en la clase `MaquinaExpendedora` y el diseñador decide modificar la representación de modo tal que el tipo de `ultMnt` sea la clase `Date` provista por Java, el cambio probablemente impacte en la clase `M111`.

Herencia y Constructores

Una clase derivada hereda de la clase base todos sus atributos y métodos, pero no los constructores. Cada clase derivada tendrá sus propios constructores, que pueden invocar a los constructores de las clases más generales.

```
//Constructor
public MaquinaExpendedora(int n) {
//Cada depósito se carga completo
  nroSerie = n;
  ultMnt = new MesAnio(1,2010);
  cantCafe = maxCafe;
  cantCacao = maxCacao;}
```

Para invocar al constructor de la clase base se usa la palabra clave `super`. En la clase `M111`:

```
public M111 (int n){
//Los depósitos comienzan completos
  super(n);
  cantTe = maxTe;
  cantLeche = maxLeche;}
```

En la clase `R101`:

```
public R101 (int n){
//Los depósitos comienzan completos
  super(n);
  cantCrema = maxCrema;}
```

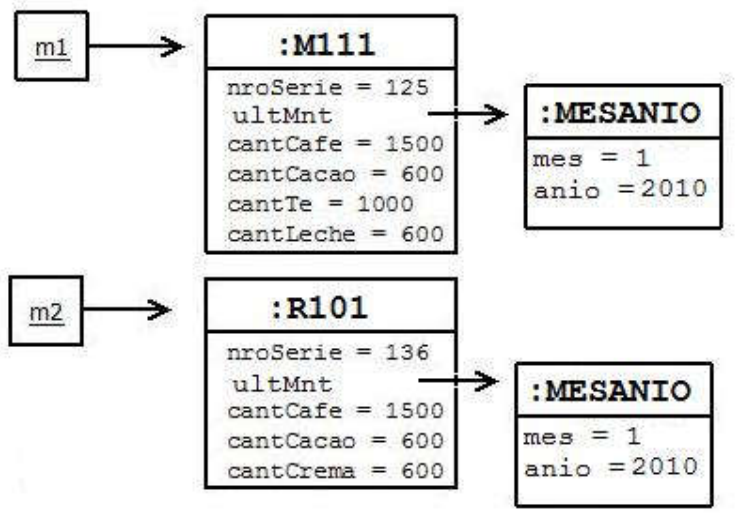
Si se invoca al constructor de una clase base mediante `super`, **siempre tiene que ser en la primera línea** de la definición del constructor de la clase derivada**.**

En ejecución, el estado interno de un objeto de una clase derivada mantiene los atributos propios de la clase y todos los atributos heredados de las clases ancestro. Así, el estado interno de un objeto de clase `M111` estará constituido por todos los atributos de su clase, más los atributos de la clase `MaquinaExpendedora`.

La ejecución de las instrucciones:

```
M111 m1 = new M111(125);
R101 m2 = new R101(136);
```

invoca a los constructores de `M111` y `R101` y la ejecución puede graficarse mediante el siguiente diagrama de objetos:



Y serán válidas las siguientes instrucciones:

```
m1.cafe();
m1.submarino();
m2.cafe();
int v = m2.vasosCarioca();
```

En cambio, en los siguientes casos se produce un error de compilación:

```
m1.carioca();
int c = m1.obtenerCantCrema();
m2.submarino();
```

debido a que `m1` es una instancia de `M111` y ni esa clase, ni sus ancestros, brindan servicios para atender los mensajes `carioca()` y `obtenerCantCrema()`. Análogamente, `m2` es una instancia de `R101`, que no brinda un servicio `submarino()` como tampoco lo brindan sus ancestros.

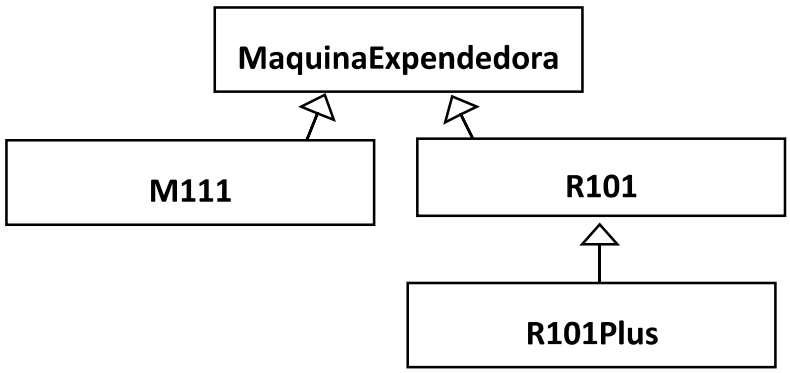
Herencia y extensibilidad

La extensibilidad es una cualidad fundamental para lograr productividad. La herencia permite que, con frecuencia, los cambios en los requerimientos puedan resolverse definiendo nuevas clases, sin modificar las clases que ya han sido desarrolladas y verificadas.

Caso de Estudio: Máquina Expendedora extendida

La fábrica de máquinas expendedoras incorpora un nuevo modelo R101 Plus que tiene la funcionalidad de R101 pero prepara un café más fuerte usando 45 grs., agrega como ingrediente a la canela con capacidad máxima de 600 grs. y prepara café bahiano. El café bahiano requiere la preparación de un café carioca al cual se le agregan 10 gramos de canela.

La jerarquía de clases incluye una nueva clase `R101Plus`, con los atributos y comportamiento específico. Gráficamente:



El diagrama de modelado se extiende ahora con la clase R101Plus:

R101Plus
<<atributos de clase>> maxCanela: entero <<atributos de instancia>> cantCanela : entero
<<constructor>> R101Plus(n:entero) <<comandos>> cargarCanela(grs: entero): entero bahiano() <<consultas>> obtenerCanela(): entero obtenerMaxCanela() : entero vasosBahiano():entero

```
Preparar una infusion requiere que los
depósitos tengan la cantidad necesaria
de ingredientes.
```

La clase `R101Plus` está vinculada a la clase `R101` por una relación de herencia. Todo objeto de la clase `R101Plus` es también un objeto de la clase R101 y además un objeto de la clase `MaquinaExpendedora`. La relación de herencia es **transitiva**. La clase `R101` es una clase derivada de la clase `MaquinaExpendedora`, pero a su vez es una clase base para la clase `R101Plus`.

La modificación en la especificación del problema no requiere modificar las clases ya implementadas, sino agregar una nueva clase. La implementación parcial de `R101Plus` es:

```
class R101Plus extends R101 {
protected static final int maxCanela = 600;
protected int cantCanela;
public R101Plus (int n){
  super(n);
  cantCanela = maxCanela;}
…
public void bahiano() {
  carioca ();
  cantCanela = cantCanela-10;}
}
```

Dada la declaración:

```
R101Plus m3 = new R101Plus(369);
```

Serán válidas las siguientes instrucciones:

```
m3.carioca();
m3.bahiano();
```

En cambio el compilador reporta un error en el siguiente caso:

```
m3.teConLeche();
```

Porque el objeto ligado a `m3` no reconoce el mensaje `teConLeche()`.

Polimorfismo

Polimorfismo significa muchas formas y en Ciencias de la Computación en particular se refiere a la capacidad de asociar diferentes definiciones a un mismo nombre, de modo que el contexto determine cuál corresponde usar.

El concepto de polimorfismo es central en la programación orientada a objetos y se complementa con el mecanismo de herencia.

> El polimorfismo es el mecanismo que permite que objetos de distintas clases puedan recibir un mismo mensaje y cada uno actúe de acuerdo al comportamiento establecido por su clase.

En el contexto de la programación orientada a objetos el polimorfismo está relacionado con variables, asignaciones y métodos.

- Una **variable polimórfica** puede quedar asociada a objetos de diferentes clases.
- Una **asignación polimórfica** liga un objeto de una clase a una variable declarada de otra clase

- En un **método polimórfico** al menos uno de los parámetros formales es una variable polimórfica.

Dadas las siguientes instrucciones:

```
MaquinaExpendedora m;
m= new M111(147);
```

La variable `m` es polimórfica y queda asociada a un objeto de clase `M111` a través de una asignación polimórfica.

Dado que una variable puede estar asociada a objetos de diferentes tipos, es posible distinguir entre:

- El **tipo estático** de una variable, es el tipo que aparece en la declaración.
- El **tipo dinámico** de una variable es la clase a la que pertenece el objeto referenciado.

El tipo estático lo determina el compilador, el tipo dinámico se establece en ejecución.

El método `masCafe(MaquinaExpendedora m)` definido en la clase `MaquinaExpendedora` recibe como parámetro una variable polimórfica con tipo estático `MaquinaExpendedora`. La consulta `masCafe` retorna un objeto de clase `MaquinaExpendedora`.

```
public MaquinaExpendedora masCafe (MaquinaExpendedora m) {
 if (cantCafe > m.obtenerCantCafe())
   return this;
   else return m;}
```

La consulta `masCafe` es un **método polimórfico** porque la asignación que vincula al parámetro efectivo con el parámetro formal es una asignación polimórfica.

```
MaquinaExpendedora mc, r, m;
r = new R101(131);
m = new M111(135);
mc = m.masCafe(r);
```

Las variables `mc`, `r` y `m` son polimórficas, el tipo estático es diferente a su tipo dinámico.

Redefinición de métodos

Un lenguaje que soporta el concepto de polimorfismo permite establecer que todos los objetos de una clase base van a brindar cierta *funcionalidad*, aunque la forma de hacerlo va a depender de su clase específica.

En Java un mismo nombre puede utilizarse para definir un método en la clase base y otro en la clase derivada. Si en la clase derivada se define un método con el mismo nombre, número y tipo de parámetros que un método definido en la clase base, el método de la clase base queda **derogado**. La definición del método en la clase derivada **redefine** el método de la clase base.

En el caso de estudio propuesto todo objeto de clase `MaquinaExpendedora()` puede preparar la infusión café y computar cuántos vasos de café puede preparar con la cantidad de ingrediente café que tiene en depósito.

Los métodos `cafe()` y `vasosCafe()` se implementan como sigue:

```
public void cafe() {
  cantCafe = cantCafe - 40;}
public int vasosCafe(){
```

```
  return (int) cantCafe/40;}
```

La especificación de la clase `R101Plus` establece que las instancias de esta clase preparan un café más fuerte, consumiendo 45 grs. de café en polvo. La implementación de `R101Plus` incluye entonces los siguientes métodos:

```
public void cafe() {
  cantCafe = cantCafe - 45;}
public int vasosCafe(){
  return (int) cantCafe/45}
```

El comando `cafe()` y la consulta `vasosCafe()` de la clase `MaquinaExpendedora` quedan **derogados** para los objetos de la clase `R101Plus`.

Luego de la ejecución de:

```
M111 m1 = new M111(111);
R101 m2 = new R101(325);
R101Plus m3 = new R101Plus(258);
```

Los siguientes mensajes se ligarán al método definido en `MaquinaExpendedora`:

```
m1.cafe();
m2.cafe();
```

En cambio:

```
m3.cafe();
```

quedará ligado al comando `cafe()` definido (redefinido) en la clase `R101Plus`.

El método `toString()` puede definirse en `MaquinaExpendedora` como:

```
public String toString(){
  return nroSerie+" "+
         cantCafe+" "+cantCacao+" "+ultMnt.toString();}
```

El mensaje `ultMnt.toString()` queda ligado al método `toString` definido en `MesAnio`.

La clase `M111` redefine `toString`:

```
public String toString(){
  return super.toString()+" "+cantLeche+" "+cantTe;}
```

En este caso `super.toString()` provoca la ejecución del método `toString` definido en la clase padre de `M111`, es decir, `MaquinaExpendedora`.

La clase `R101` también redefine `toString`:

```
public String toString(){
  return super.toString()+" "+cantCrema;}
```

Luego de la ejecución de:

```
M111 m1 = new M111(111);
R101 m2 = new R101(325);
```

Los siguientes mensajes se ligarán a los métodos definidos en `M111` y `R101` respectivamente:

```
System.out.println(m1.toString());
System.out.println(m2.toString());
```

De modo que la salida será:

```
111 1500 600 600 1000
325 1500 600 600
```

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida. En Java se utiliza entonces el **modificador**

`final`, que tiene significados levemente distintos según se aplique a una variable, a un método o a una clase.

- Para una **clase**, `final` significa que la clase no puede extenderse. Es, por tanto, una hoja en el árbol que modela la jerarquía de clases.
- Para un **método**, el modificador `final` establece que no puede redefinirse en una clase derivada.
- Para un atributo, `final` establece que no puede ser redefinido en una clase derivada, pero además su valor no puede ser modificado.

Cuando en una clase se define un método con el mismo nombre que otro de la misma clase o de alguna clase ancestro, pero con diferente número o tipo de parámetros, el método queda **sobrecargado**.

Ligadura Dinámica de código

La ligadura dinámica de código es la vinculación en ejecución de un mensaje con un método. Esto es, cuando un método definido en una clase, queda redefinido en una clase derivada, el tipo dinámico de la variable determina qué método va a ejecutarse en respuesta a un mensaje.

Dada la instrucción:

```
R101 r1 = new R101Plus(121);
```

El mensaje:

```
r1.cafe();
```

Provoca la ejecución del método definido en la clase `R101Plus`, ya que el objeto ligado a `r1` es de dicha clase.

Es decir, si una variable declarada de clase `R101`, referencia a un objeto de clase `R101Plus` y recibe un mensaje que corresponde a un método redefinido en `R101Plus`, la ligadura se establece con el método redefinido. El compilador no puede establecer la ligadura, es en ejecución que se resuelve qué mensaje corresponde ejecutar.

Polimorfismo, redefinición de métodos y ligadura dinámica de código, son conceptos fuertemente relacionados. La posibilidad de que una variable pueda referenciar a objetos de diferentes clases y que existan varias definiciones para una misma signatura, brinda flexibilidad al lenguaje, siempre que además exista ligadura dinámica de código.

Chequeo de tipos

El polimorfismo es un mecanismo que favorece la **reusabilidad** pero debe restringirse para brindar **confiabilidad**. Los chequeos de tipos en compilación garantizan que no van a producirse errores de tipo en ejecución.

El chequeo de tipos establece restricciones sobre:

- Las asignaciones polimórficas
- Los mensajes que un objeto puede recibir

En una asignación polimórfica, la clase del objeto que aparece a la derecha del operador de asignación, debe ser de la misma clase o de una clase descendiente de la clase de la variable

que aparece a la izquierda del operador. Así, el tipo estático de una variable determina el conjunto de tipos dinámicos.

Dadas las siguientes instrucciones:

```
M111 m1 = new M111(951);
R101 r1 = new R101Plus(357);
R101Plus r2;
```

No son válidas las asignaciones:

```
r2=r1;
m1=r1;
```

Si la primera de las asignaciones anteriores fuera válida, la ejecución de:

```
r2.bahiano();
```

provocaría un error de ejecución, ya que la máquina ligada a `r2` no tiene canela y por lo tanto no puede preparar esa infusión. El chequeo de tipo pretende justamente establecer controles en compilación que eviten errores de ejecución.

Con respecto a restricciones sobre los mensajes, un objeto solo puede recibir mensajes para los cuales existe un método definido en la clase que corresponde a la declaración de la variable, o en sus clases ancestro. Por lo tanto, la siguiente instrucción provoca un error de compilación:

```
r1.bahiano();
```

El objeto referenciado por una variable de clase `R101` sólo podrá recibir los mensajes que corresponden al comportamiento de la clase `R101`. El compilador no tiene modo de saber que `r1` está ligada a un objeto de clase `R101Plus` y por lo tanto que, en ejecución, será capaz de responder al mensaje `bahiano()`. Esta situación es evidente en el siguiente segmento de código:

```
if (i==0)
  r1 = new R101(125);
else
  r1 = new R101Plus(125);
r1.bahiano();
```

Si Java no estableciera el chequeo, el mensaje `bahiano()` podría ligarse al método provisto en la clase `R101Plus`, cuando se ejecute la instrucción ligada al `else` del condicional. Cuando la condición computa `true`, `r1` estará vinculada a un objeto de clase `R101` y no dispondrá de un servicio que le permita atender el mensaje. El resultado sería un error en ejecución. Java busca prevenir este tipo de situaciones y establece restricciones justamente para evitarlas. Si existe la posibilidad de que un mensaje no pueda ligarse a un servicio, se lo rechaza.

El tipo estático de la **variable** determina los mensajes que un objeto puede recibir, pero el tipo dinámico determina la implementación específica del comportamiento que se ejecuta en respuesta a los mensajes. Es decir, el compilador chequea la validez de un mensaje considerando el tipo estático de la variable. En ejecución, el mensaje se liga con el método, considerando el tipo dinámico.

Casting

Casting es un mecanismo provisto por Java para relajar el control del compilador. El programador se hace responsable de garantizar que una asignación va a ser válida o un mensaje va a poder ligarse.

```
R101 r1 = new R101Plus(987);
```

```
R101Plus r2;
```

El casting provoca que el compilador relaje el chequeo:

```
r2=(R101Plus) r1;
```

La asignación es válida, pero como contrapartida, riesgosa. Si en ejecución `r1` está ligada a un objeto de clase `R101`, se producirá un **error en ejecución**, esto es, una terminación anormal, ligada al manejo de excepciones.

Igualdad y Equivalencia en una jerarquía de clases

El diseño de una clase debe establecer si para decidir si dos objetos son iguales se va a exigir que tengan la misma identidad o basta con que sean equivalentes. La implementación del método `equals` debe verificar las propiedades:

- Reflexividad: `x.equals(x)` retorna `true`
- Simetría: si `x.equals(y)` retorna `true` si y solo si `y.equals(x)` es `true`
- Transitividad: si `x.equals(y)` y `y.equals(z)` retornan `true` entonces `x.equals(z)` retorna `true`

Además `x.equals(y)` retorna falso si `y` es nulo.

El método `equals` en `MaquinaExpendedora` se define como:

```
  public boolean equals(MaquinaExpendedora e) {
  boolean ig;
  if (this == e)
    ig = true;
  else if (e == null)
    ig = false;
   else if(getClass()!= e.getClass())
     ig = false;
    else
      ig= nroSerie==(e.obtenerNroSerie()) &&
          cantCafe==(e.obtenerCantCafe()) &&
          cantCacao==(e.obtenerCantCacao());
  return ig;}
```

La implementación de `equals` en `R101` es:

```
class R101 extends MaquinaExpendedora{
…
  public boolean equals(MaquinaExpendedora e) {
  boolean ig;
  if (this == e)
    ig = true;
  else if (e == null)
    ig = false;
   else if(getClass()!= e.getClass())
     ig = false;
    else {
     R101 r = (R101) e;
     ig=super.equals(e) &&
        cantCrema == r.obtenerCantCrema();}
  return ig;}
}
```

El casting es necesario ya que `e.obtenerCantCrema()` no es una mensaje válido. En cambio sí lo es `r.obtenerCantCrema()` porque tanto el objeto que recibe el mensaje como `r`, son instancias de la clase `R101`.

Como la consulta `equals` está definida en la clase `MaquinaExpendedora` y redefinida en cada clase derivada, la clase del objeto que recibe el mensaje determina la ligadura entre el mensaje y el método:

```
esta = unaME.equals(otraME);
```

Si la consulta `equals` tuviera una signatura diferente en cada clase, la ligadura se establece estáticamente.

```
class R101 extends MaquinaExpendedora{
…
  public boolean equals(R101 e) {
  boolean ig;
  if (this == e)
    ig = true;
  else if (e == null)
    ig = false;
   else if(getClass()!= e.getClass())
     ig = false;
    else {
     R101 r = (R101) e;
     ig=super.equals(e) &&
        cantCrema == r.obtenerCantCrema();}
  return ig;}
}
```

En este caso, el método `equals` queda sobrecargado, no redefinido, de modo que la ligadura la determina el compilador.

Herencia y Asociación

En la secciones anteriores se han definido y relacionado los conceptos de herencia, polimorfismo y vinculación dinámica, esenciales en la programación orientada a objetos. Como ilustra el caso de estudio propuesto, la herencia es un mecanismo adecuado para modelar problemas en los cuales las clases pueden organizarse de acuerdo a una estructura jerárquica. Sin embargo, hasta el momento no se mostraron los beneficios del polimorfismo y la vinculación dinámica.

Caso de Estudio Tabla de Máquinas Expendedoras

En un hospital cada pasillo está numerado y en algunos de ellos se ha instalado una máquina expendedora. El conjunto de máquinas expendedoras se mantiene en una tabla en la cual es posible instalar, retirar y buscar una máquina. La tabla se representa con un arreglo, la cantidad de componentes corresponde a la cantidad de pasillos en el hospital. La clase brinda también métodos específicos de la aplicación como computar la cantidad total de café del conjunto de máquinas.

MEHospital
<<atributos de instancia>> T [] MaquinaExpendedora
<<constructor>> MEHospital(max:entero)

```
<<Comandos>>
instalar(unaME:MaquinaExpendedora,p:entero)
retirar (p:entero)
retirar (unaMe:MaquinaExpendedora)
<<consultas>>
estaLibre(p:entero):boolean
cantPasillos():entero
cantME():entero
hayPasilloLibre():boolean
estaME (unaME : MaquinaExpendedora):boolean
totalCafe () : entero
cantMasVasos(v:entero) : entero
```

`instalar(unaME:MaquinaExpendedora,p:entero)` asigna la máquina `unaME` al pasillo `p`. Requiere 0<=p<`cantPasillos()`, `T[p]` no ligado y `unaME` no está asignada previamente a un pasillo.

`retirar (p:entero)` asigna nulo al pasillo `p`. Requiere 0<=p<`cantPasillos()`

`retirar (unaMe:MaquinaExpendedora)` busca la máquina `unaME` y si la encuentra asigna nulo al pasillo.

`cantPasillos():entero` retorna la cantidad total de pasillos, esto es, la cantidad de elementos del arreglo.

`cantME():entero` retorna la cantidad de pasillos que tienen instalada una máquina, esto es, la cantidad de elementos del arreglo que mantienen referencias ligadas.

`hayPasilloLibre():boolean` retorna true sí y solo sí, al menos uno de los pasillos no tiene instalada una máquina, es decir, una de las componentes del arreglo mantiene una referencia libre.

`estaME (unaME : MaquinaExpendedora) : boolean` retorna `true` si la máquina `unaME` está instalada en un pasillo

`totalCafe():entero` computa el total de ingrediente café almacenado entre todas las máquinas expendedoras instaladas en los pasillos.

`cantMasVasosCafe(v:entero):entero` computa la cantidad de máquinas expendedoras con capacidad para preparar más de `v` vasos de café.

Una implementación para la clase `MEHospital` es:

```
class  MEHospital{
//Atributos de instancia
protected MaquinaExpendedora[] T;
//Constructor
public MEHospital(int max){
   T = new MaquinaExpendedora[max];}
//Comandos
public void instalar(MaquinaExpendedora unaME, int p) {
/* asigna la máquina unaME al pasillo p.
Requiere 0<=p<cantPasillos() y T[p] no ligado y unaME no está
asignada previamente a un pasillo.  */
    T[p] = unaME;}
public void retirar(int p){
/* asigna nulo al pasillo p. Requiere 0<=p<cantPasillos()*/
```

```
   T[p] = null;}
public void retirar(MaquinaExpendedora unaME){
/* busca la máquina unaME y si la encuentra asigna nulo al
pasillo.*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)
     esta = unaME == T[i] ;
   i++;}
 if (esta)
   T[i--] = null;}
//Consultas
public int cantPasillos (){
/* retorna la cantidad total de pasillos*/
  return T.length;}
public int cantME(){
/*Retorna la cantidad de pasillos que tienen instalada una máquina*/
  int cant = 0;
  for (int i=0; i<cantPasillos();i++)
    if (T[i] != null) cant++;
  return cant;}
public boolean hayPasilloLibre(){
/*Retorna true si al menos un pasillo está libre*/
int i = 0; boolean libre=false;
while (i < cantPasillos() && !libre){
   if (T[i] == null)
     libre = true ;
   i++;}
return libre;}
public boolean estaME(MaquinaExpendedora unaME){
/*Retorna true si la máquina unaME está asignada a un pasillo*/
int i = 0; boolean esta=false;
while (i < cantPasillos() && !esta){
   if (T[i] != null)
     esta = unaME ==T[i] ;
   i++;}
return esta;}
public int totalCafe(){
/*Computa la cantidad total de café disponible en las máquinas
asignadas a pasillos */
  int cant = 0;
  for (int i=0; i<cantPasillos();i++)
    if (T[i] != null) cant += T[i].obtenerCantCafe();
  return cant;
}
public  int cantMasVasosCafe (int v){
/* Computa la cantidad de máquinas expendedoras que tienen
  capacidad para preparar al menos v vasos de cafe*/
  int c = 0;
  for (int i = 0; i<cantPasillos();i++)
    if (T[i] != null && T[i].vasosCafe() > v)
      c++;
  return c;
 }
}
```

Los pasillos del hospital pueden tener instaladas máquinas expendedoras de diferentes tipos. Esta característica puede ser modelada naturalmente a través de un arreglo de variables polimórficas.

El arreglo sigue siendo una estructura de componentes homogéneas, todos los elementos son instancias de `MaquinaExpendedora`. Sin embargo, también es posible argumentar que se trata de una estructura heterogénea, ya que los objetos pertenecen a distintas especializaciones de `MaquinaExpendedora` y la estructura del estado interno depende de la clase específica.

Si una variable `lME` se declara de clase `MEHospital`, el objeto ligado a esa variable, puede recibir cualquiera de los mensajes provistos por esa clase. Los métodos `asignar`, `retirar`, `cantPasillos`, `cantME`, `estaME` tienen una funcionalidad similar a otros planteados para soluciones propuestas en los capítulos anteriores.

En en la expresión `T[i] != null && T[i].vasosCafe() > v` el orden de los factores es relevante, como también lo es la evaluación en cortocircuito. Si la primera subexpresión computa `true`, la segunda no se ejecuta.

El método `obtenerCantCafe()` está definido para todo objeto de clase `MaquinaExpendedora`. Al ejecutarse `totalCafe()`, cada componente de `T` recibirá dicho mensaje y ejecutará el método definido en `MaquinaExpendedora`.

El método `vasosCafe()` está definido en `MaquinaExpendedora` y redefinido en `R101Plus`. La ligadura entre el mensaje recibido por `T[i]` y el método `vasosCafe`, va a depender del tipo dinámico de `T[i]`. Si `T[i]` es de clase `R101Plus` se ejecutará el método redefinido en dicha clase, en caso contrario se ejecuta el método definido en `MaquinaExpendedora`.

Si el hospital incorpora un nuevo tipo de máquina expendedora, con capacidad para preparar nuevos tipos de infusiones o incluso una manera diferente de preparar café, la clase `MEHospital` no se modifica. El cambio provocará seguramente la definición de una nueva clase, descendiente de `MaquinaExpendedora`, pero no implicará cambios en las clases que están desarrolladas y verificadas.

Así, los mecanismos de herencia, polimorfismo y vinculación dinámica favorecen la extensibilidad porque reducen el impacto de los cambios. Si una estructura de datos está conformada por objetos que pertenecen a una colección de clases, la colección puede incorporar a nuevas clases, sin afectar a la estructura de datos.

Esto no significa que una clase una vez desarrollada no va a tener cambios. A la clase `MEHospital` se le puede agregar funcionalidad, por ejemplo para computar cuántos pasillos tienen asignadas máquinas con capacidad para preparar más de n vasos de café. También es posible que la fábrica que produce las máquinas modifique un modelo, por ejemplo M111 y el diseñador decida cambiar la representación o funcionalidad de la clase `M111` para reflejar la modificación.

Clases Abstractas

En el diseño de una aplicación es posible definir una clase que factoriza propiedades de otras clases más específicas, sin que existan en el problema objetos concretos vinculados a esta

clase más general. En este caso la clase se dice **abstracta** porque fue creada para lograr un modelo más adecuado. En ejecución no va a haber objetos de software de una clase abstracta.

Una clase abstracta puede incluir uno, varios, todos o ningún **método abstracto**. Un método abstracto es aquel que no puede implementarse de manera general para todas las instancias de la clase. Una clase que incluye un método abstracto define solo su signatura, sin bloque ejecutable. Esto es, todos los objetos de la clase van a ofrecer una misma funcionalidad, pero la implementación concreta no puede generalizarse.

Si una clase hereda de una clase abstracta y no implementa todos los métodos abstractos, también debe ser definida como abstracta. El constructor de una clase abstracta sólo va a ser invocado desde los constructores de las clases derivadas. Una clase concreta debe implementar todos los métodos abstractos de sus clases ancestro.

En el caso de estudio propuesto la clase `MaquinaExpendedora` es abstracta porque fue creada artificialmente para factorizar los atributos y el comportamiento común a todas las máquinas. En ejecución no va a haber objetos de software de clase `MaquinaExpendedora`.

Podemos declarar variables de clase `MaquinaExpendedora` pero no crear objetos. El constructor de la clase `MaquinaExpendedora` solo va a ser invocado desde los constructores de las clases derivadas.

En el siguiente enunciado proponemos una variación del problema. Observemos que no es un diseño alternativo, sino un problema diferente, las máquinas expendedoras brindan una infusión adicional a las requeridas anteriormente.

*Una fábrica produce dos tipos diferentes de máquinas expendedoras de infusiones, M111 y R101. Cada máquina tiene un número de serie que la identifica. Las máquinas del tipo M111 preparan **café**, **mate cocido**, **café con leche**, **té**, **té con leche** y **submarino**. Tienen depósitos para los siguientes ingredientes: **café**, **té**, **yerba**, **leche** y **cacao**. Las máquinas de tipo R101 preparan **café**, **mate cocido** y **café carioca**. Tienen depósitos para **café**, **yerba**, **crema** y **cacao**.*

Los depósitos tienen las siguientes capacidades máximas:

Café	**1500**
Yerba	**1000**
Té	**1000**
Leche	**600**
Cacao	**600**
Crema	**600**

Además de la capacidad máxima de cada depósito, cada máquina mantiene registro de la cantidad disponible.

Cuando se habilita una máquina se establece su número de serie y las cantidades disponibles comienzan con el valor máximo de cada ingrediente. La cantidad disponible aumenta cuando se carga el depósito con un ingrediente específico y disminuye cada vez que se prepara una infusión. El aumento es variable, aunque nunca se puede superar la capacidad máxima de cada ingrediente. Si el valor que se intenta cargar, sumado al disponible, supera al máximo, se completa hasta el máximo y retorna el sobrante.

Cada vez que se solicita una infusión se reducen los ingredientes de acuerdo a la siguiente tabla:

	Café	Café con leche	Té	Submarino	Té con leche	Café carioca	Mate Cocido
Café	40	30				30	
Cacao				40		10	
Té			35		20		
Leche		20		50	20		
Crema						30	
Yerba							

La cantidad de gramos que demanda preparar un mate cocido depende de la máquina. En el modelo M111 se utilizan 25 grs., en el modelo R101 se usan 35 grs.

En este caso la clase `MaquinaExpendedora` incluye un **método abstracto**, la implementación de `mateCocido()` depende del tipo de máquina. Todas brindan ese servicio, pero de manera diferente.

El diagrama de clase de `MaquinaExpendedora` es ahora:

```
*MaquinaExpendedora
<<atributos de clase>>
maxCafé : entero
maxCacao : entero
maxYerba : entero
<<atributos de  instancia>>
NroSerie: entero
cantCafe : entero
cantCacao : entero
cantYerba: entero
<<constructor>>
MaquinaExpendedora(n:entero)
<<comandos>>
cargarCafe(grs: entero)
:entero
cargarCacao(grs: entero):
entero
cargarYerba(grs: entero):
entero
cafe()
*mateCocido()
<<consultas>>
obtenerNroSerie(): entero
obtenerCantCafe(): entero
obtenerCantCacao(): entero
obtenerCantYerba() : entero
obtenerMaxCafe(): entero
obtenerMaxCacao(): entero
obtenerMaxYerba() : entero
vasosCafe() : entero
*vasosMateCocido():entero
masCafe(MaquinaExpendedora e):
MaquinaExpedendora
```

El modificador **abstract** permite declarar una clase abstracta:

```
abstract class   MaquinaExpendedora {
…
```

```
}
```

El modificador **abstract** permite declarar métodos abstractos:

```
public abstract void mateCocido() ;
public abstract int vasosMateCocido();
```

Un método abstracto no tiene implementación, inmediatamente después de la lista de parámetros se escribe el símbolo ; como terminador.

```
class M111 extends MaquinaExpendedora {
     public void mateCocido() {
       yerba = yerba - 25;
}
class R101 extends MaquinaExpendedora{
     public void mateCocido() {
       yerba = yerba - 35;
}
```

La clase `MEHospital` puede incluir un método `totalVasosMateCocido()` que computa cuántas infusiones pueden prepararse con la yerba disponible entre todas las máquinas. La implementación de este método es:

```
public int totalVasosMateCocido(){
/*Retorna la cantidad vasos de mate cocido que pueden prepararse con
la yerba de todas las máquinas*/
  int cant = 0;
  for (int i=0; i< maxElementos();i++)
    if (T[i] != null) cant += T[i].vasosMateCocido();
  return cant;
}
```

La definición del método abstracto `vasosMateCocido()` permite que `T[i]` pueda recibir el mensaje `vasosMateCocido()` aun cuando el método no está implementado en `MaquinaExpendedora`. Cada objeto de `T` va a ser de alguna subclase de `MaquinaExpendedora`, de modo que existirá una implementación que se ejecutará en respuesta al mensaje.

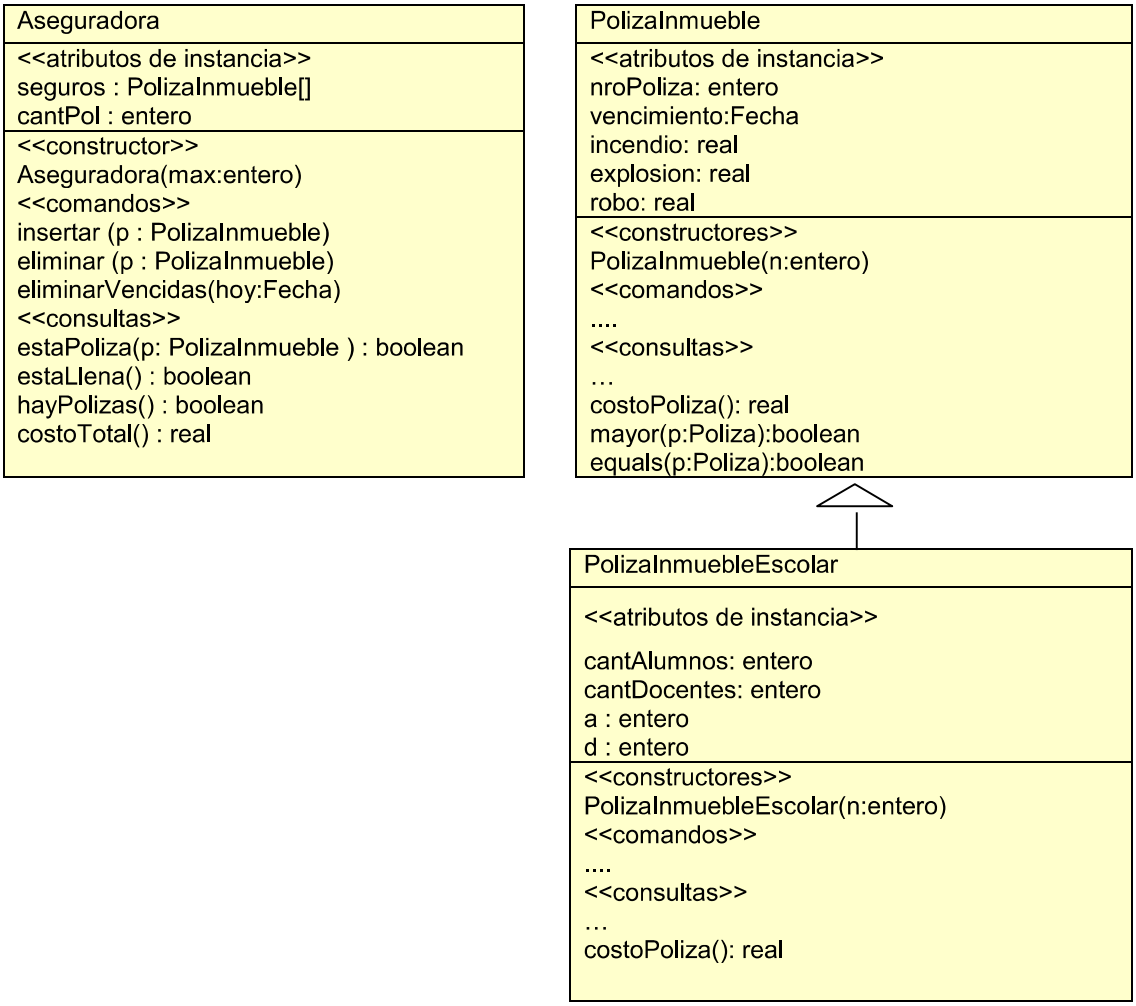
Problemas Propuestos

1. Una empresa de seguros mantiene información referida a las pólizas, de acuerdo al diagrama de clases que sigue a las siguientes consignas.

a) Implemente el diagrama en Java incluyendo los métodos triviales

- *Insertar(p:PolizaInmueble) agrega una nueva componente a la estructura, ordenada de acuerdo al criterio establecido por el método mayor, en la clase PolizaInmueble. Requiere que la estructura no esté llena y estaPoliza(p) sea falso.*
- *costoPoliza() en PolizaInmueble se computa como la suma de las coberturas por robo, explosión e incendio.*
- *costoPoliza() en PolizaInmuebleEscolar se computa como poliza de cualquier inmueble, más a por la cantidad de alumnos, más d por la cantidad de docentes.*

b) Implemente una clase tester que verifique los servicios provistos por la clase Aseguradora.



c) Considere que el diseño se amplia para agregar una clase PolizaInmuebleEquipado que extiende a Poliza Inmueble e incluye como atributos cantPersonas, montoEquipamiento, montoMobiliario y montoPersona. El costo de la poliza es el de cualquier póliza más, un monto por cada persona, más el 90% del monto del equipamiento, más el 50% del monto del mobiliario. Implemente la clase PolizaInmuebleEquipado. Analice qué cambios tiene que implementar en la clase Aseguradora cuando se extiende el diseño.