Motivación

Un sistema de software se desarrolla con el objetivo de resolver un problema que puede surgir de una necesidad o de una idea. El desarrollo de software es entonces un proceso de abstracción que permite construir un modelo para la solución de un problema.

La construcción de un modelo implica un esfuerzo intelectual, cuya complejidad depende de las características del problema y por supuesto del conocimiento del equipo de desarrollo responsable de hallar una solución. A esa complejidad se suma la necesidad de escribir código utilizando un lenguaje de programación.

Sin embargo, escribir código es una tarea relativamente sencilla respecto al resto de las etapas involucradas en el proceso. Especificar los requerimientos del problema y diseñar una solución, son etapas previas a escribir el código y por lo general son las que demandan mayor nivel de abstracción. Verificar que el código es una solución para el problema especificado por los requerimientos, es una etapa posterior, que también implica esfuerzo, excepto en problemas muy simples. Desarrollar sistemas de software de calidad para ambientes dinámicos, requiere la aplicación de principios y metodologías que guíen cada etapa del proceso.

En este contexto, enseñar a desarrollar software o, como contrapartida, aprender a desarrollar software, no es una tarea sencilla. Aun en los niveles iniciales, demanda un enfoque que de manera sistemática y disciplinada introduzca principios y metodologías adecuados para desarrollar sistemas de software de mediana y gran escala, que satisfagan criterios de calidad.

La programación orientada a objetos brinda justamente un paradigma que guía el proceso de desarrollo de software desde su concepción. En la actualidad se aplica tanto en el desarrollo de aplicaciones de pequeña escala, como para la construcción de sistemas complejos. Sin embargo, no es sencillo ilustrar la potencia del paradigma y sobre todo enfatizar la importancia de adoptar un estilo disciplinado y sistemático, planteando problemas sencillos, adecuados para enseñar a desarrollar software.

El objetivo de este libro es introducir los conceptos fundamentales del paradigma orientado a objetos y desarrollar competencias para interpretar un modelo orientado a objetos, implementarlo en Java y verificarlo para un conjunto de casos de prueba. Nuestra propuesta es presentar estos conceptos y desarrollar estas competencias a partir de distintos casos de estudio especificados a través de un texto y un diagrama de clases. Así, las etapas de desarrollo de requerimientos y diseño se asumen realizadas previamente.

Los contenidos conceptuales se desarrollan de manera transversal a la presentación del lenguaje. Esto es, para cada uno de los conceptos abordados se describe el soporte en Java, se plantean casos de estudio en cuya resolución se aplican los conceptos y las facilidades del lenguaje y se analiza la calidad de la solución en términos de legibilidad, eficiencia y correctitud.

En esta propuesta el lenguaje de programación es una herramienta que se irá presentando a través de la resolución de problemas concretos, sin una descripción exhaustiva de cada uno de los recursos. Nuestra expectativa es que el conocimiento y las habilidades desarrolladas a partir de la resolución de casos de estudio abordados de acuerdo a esta propuesta, sirvan de

base para bibliografías más avanzadas en las que se presenten contenidos específicos de ingeniería de software y se profundice el aprendizaje de recursos de programación, en particular en lo que se refiere a polimorfismo paramétrico, manejo de excepciones y concurrencia. Aunque todos estos temas puede abordarse utilizando Java, es posible llegar a un nivel de comprensión más profundo, si se presentan proponiendo diferentes lenguajes de programación.

Sobre el lector de este libro

Se asume que los lectores han adquirido previamente competencias para:

- Interpretar enunciados de problemas incluyendo definiciones usando la notación matemática.
- Diseñar algoritmos aplicando los criterios de programación estructurada, diseño top down y refinamiento paso a paso.
- Implementar soluciones en un lenguaje de programación utilizando adecuadamente las estructuras de control a propuestas por la programación estructurada.
- Verificar la solución implementada en lenguaje de programación respecto al enunciado para un conjunto significativo de casos de prueba.
- Documentar el código para favorecer la legibilidad.

A través de la resolución de problemas planteados como casos de estudio se aspira a desarrollar las siguientes competencias:

- Interpretar un diagrama de clases definido usando un lenguaje de modelado e implementarlo en Java considerando la funcionalidad y las responsabilidades especificadas.
- Diseñar algoritmos eficientes para modelar servicios que permitan procesar y actualizar estructuras de datos.
- Verificar la implementación de un método, clase o colección de clases, respecto a su especificación, utilizando casos de prueba adecuados.

El libro como material para un curso introductorio de programación

Este libro puede ser utilizado como material auxiliar a un curso introductorio a la programación orientada a objetos. Durante el desarrollo del curso en el cual se utilice el libro, los docentes podrán graduar el nivel de detalle con el que describen cada uno de los recursos provistos por Java, dependiendo del conocimiento que aspiren desarrollar en los alumnos y el grado de autonomía en el aprendizaje que esperen alcanzar.

El Proceso de Desarrollo de Software

Los sistemas de software actuales suelen resolver problemas complejos que requieren soluciones confiables, eficientes y capaces de adaptarse dinámicamente a cambios en las necesidades de los **clientes** o **usuarios**. El desarrollo de un sistema de software de estas características es un **proceso** que tiene un **ciclo de vida** conformado por etapas que pueden organizarse de diferentes formas. El **producto** final de este proceso es un sistema de software.

El proceso se realiza en el marco de un **proyecto** que establece un **cronograma** y un **presupuesto**. El éxito del proyecto está ligado a la **calidad** del **producto** final, pero también es fundamental que se complete con los recursos previstos en el presupuesto y los tiempos establecidos en el cronograma.

Aunque el desarrollo de software es una actividad creativa, requiere de la aplicación de un **paradigma** que guíe, oriente y sistematice cada **etapa**. Un paradigma de programación brinda:

- **Un principio** que describe propiedades generales que se aplican a todo el proceso de desarrollo.
- **Una metodología** que consta de un conjunto integrado de métodos, estrategias y técnicas que aseguran la aplicación del principio.
- **Un conjunto de herramientas** que soportan y facilitan la aplicación de la metodología.

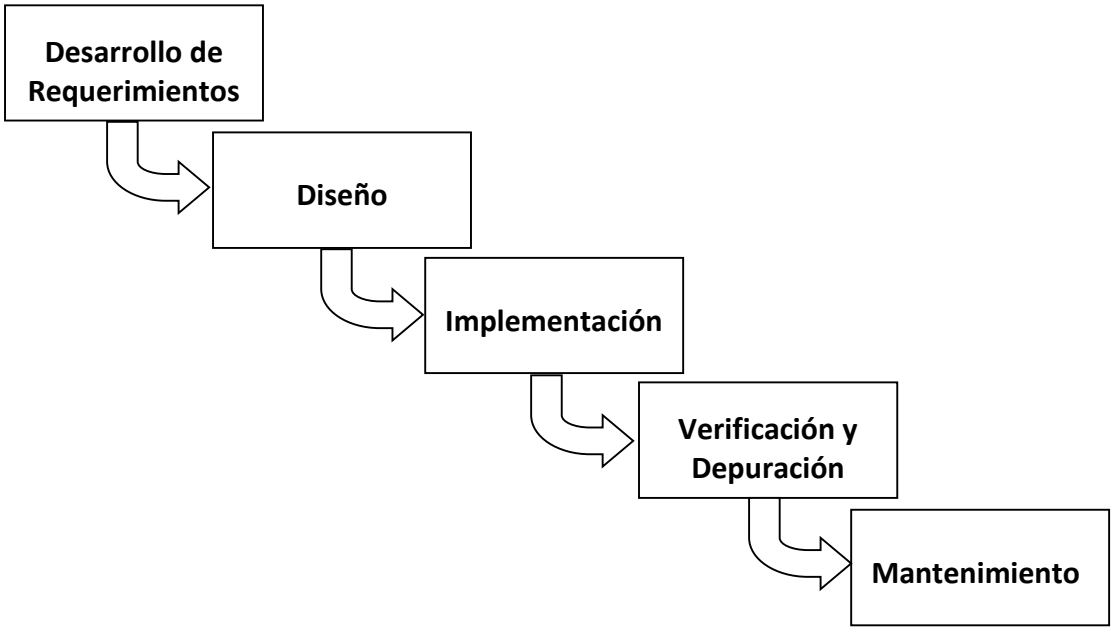
Es posible clasificar los sistemas de software de acuerdo a diferentes criterios. Proponemos la definición de tres categorías, sistemas de **pequeña, mediana y gran escala**. Aunque no hay límite precisos que separen a estas categorías, el tiempo y el costo de desarrollo son los principales factores que determinan la escala de un sistema. En los proyectos de mediana y gran escala participa un **equipo de desarrollo** conformado por profesionales con diferentes capacidades, cada uno de los cuales tiene un rol protagónico en alguna de las etapas del proceso.

Se adopta el **ciclo de vida en cascada** para estructurar las etapas y el **paradigma de programación orientada a objetos** para guiar todo el proceso. El ciclo de vida en cascada es adecuado para problemas de pequeña y mediana escala. Para problemas más complejos es conveniente utilizar un enfoque iterativo, no secuencial. El paradigma de programación orientada a objetos en cambio se aplica tanto en la resolución de problemas de pequeña o mediana escala como en el desarrollo de grandes sistemas.

Ciclo de Vida en Cascada

El **ciclo de vida en cascada** propone un enfoque específico para establecer y ordenar las etapas que conforman el proceso de desarrollo del software.

Las etapas conforman una **secuencia**, dentro de la cual el **resultado** de una etapa es el **insumo** para la siguiente. Durante el proceso intervienen diferentes participantes, entre ellos el cliente que demanda el producto, el equipo de profesionales que lo desarrolla y los usuarios que en definitiva van a interactuar con el sistema.



Existen otros modelos de ciclo de vida, algunos de ellos con distintas etapas, otros con las mismas etapas pero organizadas de manera diferente. Este modelo es adecuado para introducir conceptos de programación orientada a objetos.

Desarrollo de los Requerimientos

Un **sistema de software** se desarrolla para satisfacer una **demanda** que puede surgir de una **necesidad**, una **oportunidad** o una **idea**. Un proyecto será exitoso si el sistema satisface la demanda, para ello deben definirse con precisión los **requerimientos**.

El **resultado** de esta etapa es un **documento de especificación de requerimientos** elaborado por uno o más **analistas**, que establecen:

- **Qué** problema tiene que ser resuelto
- **Qué** características debe incluir la solución y **qué** restricciones debe cumplir
- **Por qué** es un problema y por lo tanto requiere solución
- **Quiénes** tienen la responsabilidad de participar en la construcción de la solución

Un problema puede surgir dentro de una organización que desarrolla un sistema para uso propio, ya sea a partir de una demanda del entorno interno o del contexto externo. Alternativamente el problema puede surgir dentro de una organización para crear y comercializar un sistema de software en su contexto externo.

Diseño

A partir de los requerimientos se **diseña** una **solución** para el problema especificado. El **resultado** de esta etapa es un **documento** elaborado por uno o más **diseñadores,** que describen los módulos que integrarán el sistema y el modo en que se relacionan entre sí. El documento puede establecer también **casos de prueba** o **tests** que se aplicarán en la verificación.

Implementación

A partir del documento producido durante la etapa de diseño, los **desarrolladores** o **programadores** generan el **programa** escrito en un lenguaje de programación y toda la documentación referida al código. Es importante que el programa implementado mantenga

la estructura de la solución especificada en la etapa de diseño. Cada **módulo de diseño** debería corresponderse con una **unidad de código** implementada, con cierta independencia del sistema completo.

Verificación y Depuración

La verificación evalúa si el sistema satisface la especificación de requerimientos. Durante la depuración se corrigen los errores que se detectan.

Cuando el sistema está dividido en módulos, cada unidad de código se verifica y depura por separado y luego se verifica y depura la integración de los módulos. Es importante que al menos una parte de la verificación la lleven a cabo personas ajenas a la implementación. Los casos de prueba pueden haberse establecido en el diseño o pueden ser definidos en esta misma etapa.

El término validación se utiliza para referirse a la evaluación de un sistema respecto a las necesidades reales del usuario, que en ocasiones no corresponden a los requerimientos especificados. Considerando que el principal objetivo de este libro es introducir conceptos de programación orientada a objetos, no se distinguirá entre validación y verificación, asumiendo que los requerimientos de los problemas propuestos, corresponden a las necesidades de los usuarios que los plantean.

Mantenimiento

Durante el ciclo de vida de un sistema de software las necesidades del usuario cambian y normalmente crecen. El mantenimiento involucra entonces todos los cambios en el software que resultan de modificaciones en la especificación de requerimientos. El diseño modular es fundamental para controlar el impacto de los cambios. En un sistema bien modulado los cambios menores impactan sobre un conjunto reducido de módulos o incluso pueden llegar a provocar la necesidad de agregar nuevos módulos, sin afectar a los que ya están implementados y verificados.

Dos de las herramientas fundamentales del proceso de desarrollo de software, son el **lenguaje de modelado** y el **lenguaje de programación**. El **lenguaje de modelado** comienza a utilizarse durante el desarrollo de requerimientos y permite elaborar diferentes tipos de **diagramas**. El lenguaje de programación afecta fundamentalmente a la etapa de implementación.

Si el proyecto culmina exitosamente el producto final es un sistema de software desarrollado en los tiempos establecidos en el cronograma, con los costos previstos en el presupuesto y que cumple con ciertos **criterios de calidad**. El costo y el tiempo de desarrollo están ligados a **criterios de productividad**.

Calidad

En un sentido estricto la calidad de un sistema de software se evalúa considerando el nivel de satisfacción que alcanza el usuario o cliente a partir del momento que comienza a utilizarlo. Un mecanismo menos exigente evalúa la calidad del sistema con relación a los requerimientos acordados.

En cualquier caso la calidad se evalúa a partir de diferentes **factores**, en particular:

Correctitud: Un producto de software correcto actúa de acuerdo a los requerimientos especificados.

Eficiencia: Un producto de software es eficiente si tiene una baja demanda de recursos de hardware, en particular tiempo de CPU, espacio de memoria y ancho de banda.

Portabilidad: Un producto de software es portable si puede ejecutarse sobre diferentes plataforma de hardware y de software.

Simplicidad: Un producto de software es simple si es fácil de usar, su interfaz es amigable y no requiere demasiado entrenamiento ni capacitación por parte del usuario.

Robustez: Un producto de software es robusto si reacciona adecuadamente aun en circunstancias imprevisibles.

Usabilidad: Un producto de software es usable si está disponible en el momento que el usuario lo necesita y el rendimiento está dentro de parámetros establecidos.

Los criterios de calidad son temas esenciales de la ingeniería de software, este libro se concentra en aquellos que son fundamentales en las etapas formativas de un programador y por lo tanto se utilizarán para evaluar las soluciones de los casos de estudio propuestos.

Productividad

En los últimos años se ha destinado mucho esfuerzo a desarrollar lenguajes y metodologías que permitan aumentar la **calidad** y la **productividad** del software.

La calidad de un producto de software puede definirse como su capacidad para satisfacer las necesidades del usuario establecidas durante el desarrollo de requerimientos. La calidad puede evaluarse de acuerdo a distintos factores, algunos de ellos son percibidos por el usuario o cliente, otros lo afectan indirectamente.

La productividad implica reducir **tiempos** y **costos** en la concepción y construcción de un sistema y está fuertemente ligada a dos cualidades fundamentales, **extensibilidad** y **reusabilidad**.

La **extensibilidad** se refiere a la flexibilidad de un sistema de software para adaptarse a cambios en los requerimientos. La **reusabilidad** es la capacidad de aprovechar en la construcción de un nuevo sistema, componentes desarrolladas previamente.

La productividad está ligada al **costo** y al **tiempo** que demanda cada etapa del desarrollo de un sistema de software. Si una etapa se saltea o no se completa adecuadamente, las etapas siguientes sufrirán las consecuencias, el proyecto probablemente termine demandando más tiempo y el costo global será mayor al presupuestado. Por el contrario, si cada etapa se realiza de manera adecuada, el proyecto se completará de acuerdo al cronograma y al presupuesto establecido. Esto no garantiza el éxito, existen otros factores, en general externos al proyecto, que inciden en la aceptación del producto.

Extensibilidad: Un producto de software es extensible si es fácil adaptarlo a cambios en la especificación.

Reusabilidad: Un módulo de software o una colección de módulos es reusable si puede utilizarse para la construcción de diferentes aplicaciones.

La extensibilidad afecta fundamentalmente al mantenimiento pero depende de las etapas anteriores. Un sistema con un diseño adecuado será más fácil de extender que una solución con un diseño pobre.

La reusabilidad afecta a todas las etapas. Cuando el equipo de desarrollo resuelve un problema puede reusar requerimientos, módulos de diseño o código de sistemas anteriores.

Ambas cualidades están ligadas a la **legibilidad**, un producto de software es legible si un desarrollador o incluso otros miembros del equipo de desarrollo, puede leerlo e interpretar su estructura y contenido fácilmente.

Programación Orientada a Objetos

El **principio** fundamental del **paradigma de programación orientada a objetos** es desarrollar un sistema de software en base a las entidades relevantes del problema que le da origen.

Una **metodología orientada a objetos** parte del reconocimiento de los **objetos** del problema. Cada objeto del problema es una entidad que puede caracterizarse a través de sus **atributos** y su **comportamiento**. Los atributos y el comportamiento permiten agrupar a los objetos en **clases**. Una metodología orientada a objetos abarca el ciclo de vida completo de un producto de software y brinda herramientas adecuadas para cada etapa.

El **conjunto de herramientas** incluye a un **lenguaje de modelado** y un **lenguaje de programación**. Un lenguaje de modelado es una notación artificial que permite describir la estructura y las componentes de un sistema de software. Un lenguaje de modelado consistente con la programación orientada a objetos, permite representar a la colección de clases y sus relaciones a través de un **diagrama de clases**. Un lenguaje de programación orientado a objetos brinda mecanismos que facilitan la implementación del sistema diseñado.

En la etapa de **desarrollo de requerimientos** la especificación de:

- **Qué** problema tiene que ser resuelto
- **Qué** características debe incluir la solución y **qué** restricciones debe cumplir

se aborda a partir de la identificación de los **objetos** relevantes del problema y su clasificación. Una **clase** define los atributos que caracterizan a un conjunto de objetos. Las clases se vinculan entre sí a través de diferentes mecanismos de relación. El documento producido en esta etapa incluye un **diagrama de clases** que modela la colección de clases y sus relaciones.

En la etapa de **diseño** se establece el comportamiento de los objetos y se completa la especificación de las clases y relaciones.

En la actualidad UML es el lenguaje de modelado más utilizado cuando se aplica la programación orientada a objetos. UML permite elaborar diferentes tipos de diagramas, en particular diagramas de clases.

Existen numerosos lenguajes de programación orientados a objetos que facilitan la implementación del sistema diseñado. Java es un lenguaje que adopta los lineamientos de la programación orientada a objetos y se utiliza tanto en el ámbito académico como comercial.

Es importante considerar que una metodología favorece el proceso de desarrollo, pero no es una receta infalible y completa que garantice la calidad y productividad. Los lenguajes de modelado y los lenguajes de programación orientados a objetos brindan mecanismos consistentes con la metodología, pero no garantizan que la implementación sea una representación consistente con el diseño.

En los casos de estudio que se presentan en los capítulos que siguen, se utiliza una versión simplificada de UML para los diagramas de clases y Java como lenguaje de programación. Los

enunciados propondrán una especificación de requerimientos acompañada de un diagrama de clases. Se explicarán las principales características de Java, asumiendo cierto nivel de autonomía para el aprendizaje de los aspectos básicos de un lenguaje de programación.

Objetos y Clases

El desarrollo de un sistema de software comienza con la elaboración de un **modelo del problema** a resolver. A medida que se avanza en el proceso de desarrollo se obtiene un **modelo de la solución del problema.**

Un **modelo** es una representación que permite describir, explicar, analizar o simular un sistema o proceso a partir de sus entidades relevantes y el modo en que se relacionan. Existen distintos **tipos de modelos**. Una maqueta de un estadio de fútbol es un modelo físico. Un sistema de ecuaciones que describe el equilibrio en la resistencia de los materiales con los cuales se construyen las plateas, es un modelo matemático. Un plano catastral que muestra un terreno y sus límites, es un modelo gráfico. También son representaciones gráficas los planos del sistema de iluminación y el plano de alzada de los vestuarios.

El mecanismo fundamental que se utiliza para elaborar un modelo es la **abstracción**. Una abstracción se genera a partir de una serie de operaciones mentales que permiten **identificar** entidades y caracterizarlas de acuerdo a sus propiedades y al modo en que interactúan con las demás. Una misma entidad puede ser caracterizada de diferentes maneras según qué propiedades se consideren esenciales y qué nivel de detalle se pretenda en la representación.

En un modelo del sistema solar, por ejemplo, algunas de las entidades relevantes son el Sol, la Luna, Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano, Neptuno, Plutón, Ceres. Algunas propiedades de la Tierra son el radio, la masa, el período orbital, el período de rotación y la atmósfera. La Tierra puede rotar sobre sí misma y trasladarse siguiendo una órbita alrededor del Sol.

Otro mecanismo fundamental para la construcción de modelos es la **clasificación**. Una clasificación se genera a partir de una serie de operaciones mentales que permiten **agrupar** entidades en función de sus **semejanzas** y **diferencias**. Las semejanzas o diferencias se establecen en función de diferentes **criterios**. El criterio de clasificación va a estar ligado a las propiedades que se identificaron como relevantes.

En el modelo del sistema solar, Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano y Neptuno pueden agruparse en la clase Planeta. Todas las entidades de la clase Planeta se caracterizan por las mismas propiedades, aunque los valores para cada propiedad van a variar de una entidad a otra. La clase Planeta es un patrón para las entidades Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano y Neptuno.

La capacidad para clasificar entidades se desarrolla de manera natural en la infancia a través del contacto con objetos concretos. Se refuerza progresivamente y se especializa a medida que aumenta el conocimiento en un determinado dominio.

> El **principio fundamental del paradigma de programación orientada a objetos** es **construir un sistema de software** en base a las entidades de un **modelo** elaborado a partir de un proceso de abstracción y clasificación.

El concepto de objeto

El término **objeto** es central en la programación orientada a objetos y se refiere a dos conceptos relacionados pero diferentes**.**

Durante el **desarrollo de requerimientos** de un sistema de software el **analista** es responsable de identificar los **objetos del problema** y caracterizarlos a través de sus **atributos**. En la etapa de diseño se completa la representación modelando también el **comportamiento** de los objetos.

Un **objeto del problema** es una entidad, física o conceptual, caracterizada a través de **atributos** y **comportamiento**. El comportamiento queda determinado por un conjunto de **servicios** que el objeto puede brindar y un conjunto de **responsabilidades** que debe asumir.

Cuando el sistema de software está en ejecución, se crean objetos de software.

Un **objeto de software** es un **modelo**, una representación de un objeto del problema. Un objeto de software tiene una **identidad** y un **estado interno** y recibe **mensajes** a los que responde ejecutando un **servicio**. El estado interno mantiene los valores de los atributos.

Los objetos se comunican a través de mensajes para solicitar y brindar servicios**.** La ejecución de un servicio puede modificar el estado interno del objeto que recibió el mensaje y/o computar un valor.

El **modelo computacional** propuesto por la programación orientada a objetos es un mundo poblado de objetos comunicándose a través de mensajes.

La ejecución se inicia cuando un objeto recibe un **mensaje** y en respuesta a él envía mensajes a otros objetos.

Caso de Estudio: Identificación de los objetos en un Sistema de Gestión Hospitalaria

En un hospital se mantiene la historia clínica de cada paciente, incluyendo nombre, sexo, fecha de nacimiento, obra social, etc. Las historias clínicas registran además información referida a cada práctica de diagnóstico, por ejemplo las radiografías. Cada radiografía incluye una imagen, que puede representarse como un conjunto de pixels. Un pixel es la menor unidad homogénea en color en una imagen digital.

Durante el desarrollo del sistema de Gestión Hospitalia, se identifican como objetos del problema a las historias clínicas. Cada historia clínica tiene varios atributos, entre ellos el paciente. Cada paciente es a su vez un objeto, cuyos atributos pueden ser nombre, sexo, fecha de nacimiento, obra social, etc. La fecha de nacimiento es un objeto, cuyos atributos son día, mes y año.

Una radiografía realizada a un paciente es un tipo particular de procedimiento de diagnóstico, con un atributo imagen. Una imagen es en sí misma un objeto, cuyo principal atributo es un conjunto de pixels. Cada pixel es un objeto, cuyos atributos son tres números enteros. El aparato con el que se realiza el procedimiento es un objeto conceptual, como también lo son el médico que ordena la realización del procedimiento, la enfermera que la registra en la historia clínica y el técnico que la hace.

La medición de la presión arterial es otro tipo de procedimiento de diagnóstico, con atributos para registrar el valor máximo y mínimo. El conjunto de registros de presión arterial de un paciente es un objeto formado por otros objetos.

En un momento determinado de la ejecución del sistema de Gestión del Hospital, se mantendrá en memoria un objeto de software que modelará a una historia clínica en particular. Algunos de los atributos serán a su vez objetos.

La palabra **objeto** se utiliza entonces para referirse a:

- Los **objetos del problema**, es decir, las entidades identificadas durante el desarrollo de requerimientos o el diseño: cada historia clínica, paciente, radiografía, imagen, pixel, registro de presión arterial, etc.
- Los **objetos de software**, esto es, cada representación que modela en ejecución a una entidad del problema: cada historia clínica, paciente, radiografía, imagen, pixel, registro de presión arterial, etc.

Una vez que se han identificado los objetos del problema es posible concentrarse en la especificación de algunos de ellos en particular.

Caso de Estudio: Especificación de Requerimientos de Medición de la presión arterial

*La **presión arterial** es la fuerza de presión ejercida por la sangre circulante sobre las arterias y constituye uno de los principales **signos vitales** de un paciente. Se mide por medio de un esfigmomanómetro, que usa la altura de una columna de mercurio para reflejar la presión de circulación. Los valores de la presión sanguínea se expresan en kilopascales (kPa) o en milímetros del mercurio (mmHg). Para convertir de milímetro de mercurio a kilopascales el valor se multiplica por 0,13.*

*La presión **sistólica** se define como el **máximo** de la curva de presión en las arterias y ocurre cerca del principio del ciclo cardíaco durante la sístole o contracción ventricular; la presión **diastólica** es el valor **mínimo** de la curva de presión en la fase de diástole o relajación ventricular del ciclo cardíaco. La **presión de pulso** refleja la diferencia entre las presiones máxima y mínima medidas. Estas medidas de presión no son estáticas, experimentan variaciones naturales entre un latido del corazón a otro a través del día y tienen grandes variaciones de un individuo a otro.*

La hipertensión se refiere a la presión sanguínea que es anormalmente alta, y se puede establecer un umbral para la máxima y otro para la mínima que permitan considerar una situación de alarma.

Desarrollar el sistema completo para la Gestión del Hospital de manera integrada es una tarea compleja, que probablemente demandará la participación de un equipo de profesionales con diferentes roles. La complejidad se reduce si durante la etapa de diseño se divide el problema en subproblemas más simples que el problema original. La especificación de la presión arterial de un paciente es un problema de pequeña escala, con relación al problema global de desarrollar un sistema de Gestión para el Hospital. Sin embargo, establecer adecuadamente los requerimientos es una tarea fundamental, aun para problemas simples.

El concepto de clase

Durante el desarrollo de requerimientos de un sistema de software se identifican los objetos relevantes del **problema**, se los caracteriza a través de sus atributos y se los agrupa en **clases**. Todos los objetos que son instancias de una clase van a estar caracterizados por los mismos atributos. En la etapa de diseño se especifica el comportamiento de los objetos y probablemente se agregan nuevas clases significativas para la **solución** del problema.

> Desde el punto de vista del diseño de un sistema de software, una **clase** es un *patrón* que establece los atributos y el comportamiento de un conjunto de objetos.

Durante la implementación, el programador escribe el código de cada clase en un lenguaje de programación.

En la implementación de un sistema de software una **clase** es un **módulo de software** que puede construirse, verificarse y depurarse con cierta independencia a los demás.

Así, un **sistema de software orientado a objetos** es una **colección de módulos de código**, cada uno de los cuales implementa a una de las clases modeladas en la etapa de diseño. En la ejecución del sistema se crean **objetos de software**, cada uno de ellos es instancia de una clase.

Desde el punto de vista **estático** un **sistema de software orientado a objetos** es una **colección de clases** relacionadas. Desde el punto de vista **dinámico** un **sistema de software orientado a objetos** es un **conjunto de objetos** comunicándose.

Cada objeto brinda el conjunto de servicios que define su clase.

El diseño de una clase

Durante el desarrollo de requerimientos el **analista** construye un **diagrama de clases** que modela el **problema**. El **diseñador** del sistema completa el diagrama de clases para modelar la **solución**.

Un **diagrama de clases** es una representación visual que permite modelar la estructura de un sistema de software a partir de las clases que lo componen.

El diagrama de clases se construye usando un **lenguaje de modelado**. El diagrama de cada clase en el lenguaje de modelado tiene la siguiente forma:

<Nombre>
<<Atributos>> …
<<Constructores>> … <<Comandos>> … <<Consultas>> …
<<Responsabilidades>> …

El **nombre** de una clase representa la abstracción del conjunto de instancias. Por lo general es un sustantivo y debe elegirse cuidadosamente para representar al conjunto completo.

Un **atributo** es una propiedad o cualidad relevante que caracteriza a todos los objetos de una clase.

Es posible distinguir los **atributos de clase** de los **atributos de instancia**. En el primer caso, el valor es compartido por todos los objetos que son instancias de la clase. Los valores de los atributos de instancia varían en cada objeto.

Un **servicio** es una operación que todas las instancias de una clase pueden realizar.

Los **servicios** pueden ser **métodos** o **constructores**. Los métodos pueden ser **comandos** o **consultas**.

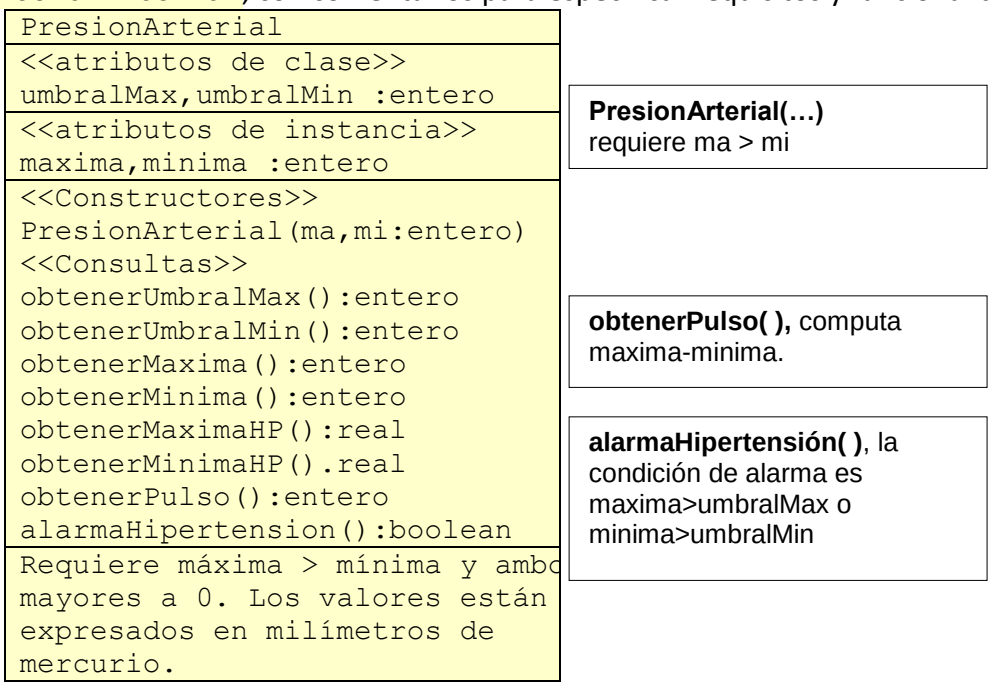
Una **responsabilidad** representa un compromiso para la clase o un requerimiento.

El conjunto de servicios y las responsabilidades determinan el **comportamiento** de las instancias de una clase.

Al diagrama de una clase se le pueden agregar **notas** o **comentarios** que describan **requisitos** o la **funcionalidad**. Pueden dibujarse mediante un rectángulo y pueden tener la esquina superior derecha plegada, como si fuera una hoja. Se puede unir con una línea al elemento de la clase al cual corresponde la nota. Alternativamente los requisitos y la funcionalidad de cada servicio puede especificarse como parte de la descripción del problema o a continuación del diagrama.

Caso de Estudio: Diagrama de la clase PresionArterial

Retomando el caso de estudio propuesto, el siguiente diagrama modela la clase `PresionArterial`, con comentarios para especificar requisitos y funcionalidad.



Los comentarios podrían reemplazarse por especificaciones como:

PresionArterial(ma,mi:entero): Inicializa los valores de los atributos con los parámetros. Requiere `ma` > `mi`.
obtenerPulso( ): computa`maxima`-`minima`. Los valores están representados en milímetros de mercurio.

alarmaHipertensión( ): retorna true si `maxima`>`umbralMax` o `minima`>`umbralMin`.

La clase `PresionArterial` es un **patrón** para cada medición que se va a representar. El diseñador decidió que es relevante representar la máxima y la mínima de cada medición. En la caracterización interesa también conocer el pulso, pero el diseñador resolvió que este valor no se mantenga como un atributo, sino que se calcule como la diferencia entre la máxima y la mínima.

De acuerdo a esta propuesta, en el momento que se crea una medición se establecen los valores de los dos atributos de instancia. El diseñador decidió que la clase `PresionArterial` requiera que la clase que la usa se haga responsable de garantizar que los valores cumplen ciertas restricciones, en este caso que la máxima sea mayor que la mínima.

El caso de estudio presentado forma parte de una aplicación de gran escala. La implementación de la clase `PresionArterial` es una solución para un subproblema, que luego va a integrarse con otras para resolver el problema global. Así, los beneficios de la programación orientada a objetos, que pueden pasar desapercibidos cuando se abordan problemas de pequeña escala, quedarán mejor ilustrados si se considera cada solución en el contexto de una aplicación de mayor envergadura.

La implementación de una clase

En la etapa de implementación de un sistema de software el programador parte del documento elaborado por el diseñador y escribe el código de cada clase del diagrama. La implementación requiere elegir un **lenguaje de programación**. Un lenguaje orientado a objetos permite retener durante la implementación la organización de las clases modeladas durante el desarrollo de requerimientos y la etapa de diseño.

En este libro se ha elegido **Java** como lenguaje de programación y se adoptaron algunas convenciones para el código. En particular el código de cada clase mantiene la estructura de cada diagrama:

<Nombre>
<<Atributos>> …
<<Constructores>> … <<Comandos>> … <<Consultas>> …
<<Responsabilidades>> …

```
class <Nombre>{
//Atributos de clase
…
//Atributos de instancia
…
//Constructores
…
//Comandos
…
//Consultas
…
}
```

La palabra reservada `class` está seguida por un identificador que es el nombre de la clase. Java es libre de la línea y sensible a las minúsculas y mayúsculas. Las { } son los delimitadores de cada unidad de código. Existen otros delimitadores como los corchetes y los paréntesis. El símbolo // precede a un comentario de una línea. Los símbolos /* */ delimitan a un comentario de varias líneas. El símbolo ; termina cada instrucción.

Los **miembros** de una clase son **atributos** y **servicios**. Manteniendo la misma clasificación que la propuesta por el lenguaje de modelado, los atributos pueden ser de clase o de instancia. Los servicios pueden ser constructores, comandos o consultas. En todos los casos están formados por un encabezamiento y un bloque de instrucciones delimitado por llaves.

Los atributos se definen a través de la declaración de variables. En la declaración de una variable, el tipo precede al nombre de la variable. Un **tipo elemental** determina un conjunto de valores y un conjunto de operaciones que se aplican sobre estos valores. En ejecución, una **variable de un tipo elemental** mantiene un **valor** que corresponde al tipo y participa en las operaciones establecidas por su tipo.

Java brinda siete tipos de datos elementales. Una variable de un tipo elemental se utiliza como operando en cualquier **expresión** en la que se apliquen **operadores** con los cuales su

tipo sea compatible. El mecanismo de **conversión implícito** entre tipos elementales permite escribir **expresiones mixtas**, esto es, con operandos de distinto tipo.

Un **constructor** es un **servicio** provisto por la clase y se caracteriza porque recibe el mismo nombre que la clase. El constructor se invoca cuando se crea un objeto y habitualmente se usa para inicializar los valores de los atributos de instancia. Una clase puede brindar varios constructores, siempre que tengan diferente número o tipo de parámetros.

Si en una clase no se define explícitamente un constructor, el compilador crea automáticamente uno, en ese caso los atributos son inicializados **por omisión**. Si la clase incluye uno o más constructores, el compilador no agrega ningún otro.

Los **comandos** son servicios que modifican los valores de uno o más de los atributos del objeto que recibe el mensaje. Las **consultas** son servicios que no modifican el estado interno del objeto que recibe el mensaje y por lo general retornan un valor. Un comando puede retornar también un valor. Los comandos y consultas conforman los **métodos** de una clase.

Cada método está precedido por el **tipo del resultado**. Si el tipo es `void` el método es un comando. Si el tipo no es `void` el método debe incluir una instrucción de retorno. La instrucción de retorno comienza con la palabra `return` y sigue con una expresión que debe ser **compatible** con el tipo del resultado.

Una implementación para la clase `PresionArterial`, cuyo diseño se presentó en la sección anterior, puede ser:

Caso de Estudio: El código de la clase PresionArterial

```
class PresionArterial {
//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;
//Constructor
public PresionArterial(int ma,int mi){
//Requiere ma > mi
  maxima = ma;
  minima = mi;}
//Consultas
public int obtenerUmbralMax(){
  return umbralMax;}
public int obtenerUmbralMin(){
  return umbralMin;}
public int obtenerMaxima(){
  return maxima;}
public int obtenerMinima(){
  return minima;}
public double obtenerMaximaHP(){
//Convierte a hectopascales
  return maxima*0.13;}
public double obtenerMinimaHP(){
//Convierte a hectopascales
  return minima*0.13;}
public int obtenerPulso(){
```

```
  return maxima-minima;}
public boolean armaHipertension(){
 return maxima > umbralMax || minima > umbralMin;}}
```

Algunas convenciones adoptadas en este libro para favorecer la legibilidad:

- La primera letra del nombre de la clase se escribe en mayúscula y por lo tanto también el nombre del constructor.
- La primera letra del nombre de atributos y métodos se escribe con minúscula. Se separan las palabras de un identificador escribiendo la inicial de todas las palabras, excepto la primera, con mayúscula. Por ejemplo, `umbralMax` u `obtenerUmbralMax`
- Se escriben como comentarios las notas y responsabilidades del diagrama de clases, excepto cuando resultan evidentes del código como por ejemplo el cálculo del pulso.
- Se usan comentarios para identificar las secciones en las que se definen los atributos, constructores, comandos y consultas.
- Los nombres de las consultas que retornan el valor de un atributo comienzan con la palabra `obtener`. En el caso de que la clase brinde comandos para modificar el valor de un atributo, sus nombres comenzarán con la palabra `establecer`. Otra posibilidad es mantener la convención en inglés usando los prefijos `get` y `set`.

Cada atributo queda ligado a una variable. En este caso de estudio, ambos atributos se declaran de un tipo elemental. La palabra reservada `private` es un **modificador**, indica que las variables `umbralMax`, `umbralMin`, `maxima` y `minima` solo son **visibles** y pueden ser usadas dentro de la clase. Fuera de la clase los atributos privados no son visibles.

El modificador `static` establece que todas las instancias de la clase `PresionArterial` comparten el mismo valor para cada umbral. El modificador `final` indica que se trata de valores constantes, establecidos en la declaración.

En este caso de estudio, la clase brinda un único constructor que establece los valores de los atributos de instancia de acuerdo a los parámetros. La clase `PresionArterial` no ofrece comandos para modificar los atributos de instancia.

Aunque la clase no mantiene un atributo de instancia para la presión del pulso, las clases que usan los servicios provistos por `PresionArterial` acceden de manera uniforme a la presión máxima, mínima y a la presión del pulso. Asimismo puede acceder a los valores que corresponden a la presión máxima y mínima expresados en kilopascales, aunque no están ligados a variables, sino que se computan.

Los tipos `int`, `float`, `double` y `boolean` son tipos elementales. La expresión:

```
maxima*0.13
```

recibe un operando de tipo `int` y otro de tipo `float`, el valor del atributo `maxima` se convierte implícitamente antes de computarse el resultado. Como el resultado de `obtenerMaximaHP` es de tipo `double`, el valor computado se convierte antes de retornar.

La instrucción:

```
 return maxima > umbralMax || minima > umbralMinima;
```

retorna un valor de verdad que resulta de computar la expresión booleana. El operador `||` se evalúa en **cortocircuito**, es decir, si la primera expresión computa `true`, no computa la

segunda. En la siguiente instrucción la evaluación es **completa**, esto es, se computan las dos subexpresiones:

```
return maxima > umbralMax | minima > umbralMinima;
```

Ejercicio: Consulte un tutorial de Java para averiguar: los siete tipos elementales provistos, el rango de valores para cada uno, el valor por omisión para cada uno de ellos, las reglas de precedencia de los operadores, el significado de los operadores en cortocircuito, el significado de los operadores aritméticos unarios y combinados (asignación y aritméticos), cuáles son las reglas de conversión implícita y compatibilidad para tipos elementales.

Edición, compilación y ejecución

La edición del código de cada clase se realiza dentro de un **entorno de desarrollo** que integra también recursos para compilar, depurar, ejecutar código y crear aplicaciones autónomas. La sigla IDE se utiliza justamente para referirse a un entorno integrado de desarrollo. Algunas IDEs brindan otros recursos que favorecen el proceso de desarrollo.

El desarrollador edita el **código fuente** de cada clase y luego las **compila**. Una de las clases debe incluir un método con la siguiente signatura:

```
public static void main (String a[])
```

En Java la ejecución comienza cuando se invoca el método `main` de una clase específica. Todas las clases que conforman el sistema deben haberse compilado antes de que se invoque a `main`. La ejecución provocará la creación de algunos objetos de otras clases que recibirán mensajes y probablemente crearán objetos de otras clases.

La verificación y depuración de una clase

Un sistema de software orientado a objetos está conformado por una colección de clases. Cada clase implementada debe ser testeada individualmente antes de integrarse con el resto de las clases que conforman la colección. La verificación de una clase aspira a detectar y depurar errores de **ejecución** o de **aplicación**. Los errores sintácticos o de compilación ya han sido corregidos antes de comenzar la verificación.

Los errores de ejecución provocan la terminación anormal del sistema. Para detectarlos es necesario anticipar distintos **flujos de ejecución**, defiendo **casos de prueba** que conducen a una terminación anormal. Java previene muchos errores de ejecución imponiendo chequeos durante la compilación, sin embargo no controla situaciones como por ejemplo división por 0. En los capítulos que siguen se describen otros tipos de errores de ejecución que es importante detectar y depurar.

Los errores de aplicación, también llamados errores lógicos, son probablemente los más difíciles de detectar, el sistema no termina en forma anormal, pero los resultados no son correctos. La causa puede ser que no se hayan especificado correctamente los requisitos, las funcionalidades o las responsabilidades o bien que el diseñador haya elaborado correctamente la especificación, pero la implementación no sea consistente con ella.

En un sistema de mediana o alta complejidad probablemente los roles de analista, diseñador, desarrollador y responsable de verificación y depuración sean ocupados por diferentes miembros del equipo de desarrollo. Sin embargo, con frecuencia el diseñador define un conjunto de **casos de prueba** que el responsable de la verificación debe utilizar.

En todos los casos el objetivo es detectar y depurar los errores, considerando las responsabilidades asignadas a la clase y la funcionalidad establecida para cada servicio en la especificación de requerimientos. Es importante considerar que la verificación muestra la presencia de errores, no garantiza la ausencia.

Para cada uno de los casos de estudio presentados en este libro, se propone una clase **tester** que verifica los servicios de una o más clases para un conjunto de casos de prueba. Los casos de prueba pueden ser valores:

- Fijos establecidos en el código de la clase tester
- Leídos de un archivo, por consola o a través una interface gráfica
- Generados al azar

Los casos de prueba propuestos aspiran detectar errores internos en el código, no fallas externas al sistema o situaciones que no fueron previstas en el diseño.

La siguiente clase tester utiliza valores fijos e incluye el método `main` que inicia la ejecución:

Caso de Estudio: La clase tester para la clase PresionArterial

```
class testPresion {
public static void main (String a[]){
  PresionArterial mDia;
  PresionArterial mTarde;
  mDia = new PresionArterial (115,60);
  mTarde = new PresionArterial (110,62);
  int p1 = mDia.obtenerPulso();
  int p2 = mTarde.obtenerPulso();

  System.out.println (mDia.obtenerMaxima()+"-"+
    mDia.obtenerMinima()+" pulso "+p1);
  System.out.println (mTarde.obtenerMaxima()+"-"+
    mTarde.obtenerMinima()+" pulso "+p2);}}
```

La compilación de la clase tester exige que la clase `PresionArterial` compile también. Una vez que el compilador no detecta errores es posible invocar el método `main` a partir del cual se inicia la ejecución. Las instrucciones:

```
  PresionArterial mDia;
  PresionArterial mTarde;
```

Declaran dos variables de clase `PresionArterial` y son equivalentes a:

```
  PresionArterial mDia,mTarde;
```

Las instrucciones:

```
  mDia = new PresionArterial (115,60);
  mTarde = new PresionArterial (110,62);
```

Crean dos objetos, cada uno de los cuales queda **ligado** a una variable. La instrucción:

```
  int p1 = mDia.obtenerPulso();
```

envía el mensaje `obtenerPulso()` al objeto ligado a la variable `mDia`. El mensaje provoca la ejecución del método provisto por la clase y retorna un valor de tipo `int` que se asigna a la variable `p1`.

Las instrucciones:

```
  System.out.println (mDia.obtenerMaxima()+"-"+
    mDia.obtenerMinima()+" pulso "+p1);
  System.out.println (mTarde.obtenerMaxima()+"-"+
```

```
    mTarde.obtenerMinima()+" pulso "+p2);
```

provocan una salida por consola. `System` es un objeto que recibe el **mensaje** `out.println`. El parámetro de este mensaje es una **cadena de caracteres,** esto es un objeto de la clase `String`, provista por Java. La cadena se genera **concatenando cadenas de caracteres**.

La expresión `mDia.obtenerMaxima()` envía el mensaje `obtenerMaxima()` al objeto ligado a la variable `mDia`. El mensaje provoca la ejecución del método que tiene ese mismo nombre y retorna un resultado de tipo `int`, que se convierte automáticamente a una cadena de caracteres, antes de concatenarse para formar otra cadena.

Objetos y Referencias

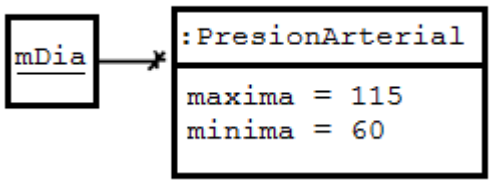
En Java los objetos son **referenciados** a través de variables. La ejecución de la instrucción:

```
PresionArterial mDia = new PresionArterial(115,60);
```

tiene el siguiente significado:

- Declara la variable `mDia`
- Crea un objeto de clase `PresionArterial`
- Invoca al constructor de la clase `PresionArterial`
- Liga el objeto a la variable `mDia`

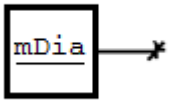
Una variable ligada mantiene una referencia al estado interno de un objeto. El **diagrama de objetos** que modela la ejecución de la instrucción anterior es:



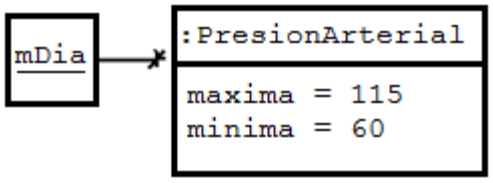
La declaración de la variable puede separarse de la creación del objeto:

```
PresionArterial mDia;
mDia = new PresionArterial(115,60);
```

La primera instrucción crea una **referencia nula**, se dice entonces que la variable `mDia` **no está ligada** y puede graficarse como sigue:



La segunda instrucción crea el objeto, invoca al constructor y liga el objeto a la variable. El diagrama es nuevamente:



La variable `mDia` está ligada a un objeto de clase `PresionArterial`, esto es, mantiene una referencia al estado interno del objeto.

En este caso de estudio, el estado interno del objeto ligado a la variable `mDia` está formado por los atributos `maxima` y `minima`. La estructura del estado interno depende de las variables de instancia de la clase, no de los servicios provistos.

Clases y Tipos de Datos

Como se mencionó antes, cuando el analista o el diseñador de un sistema orientado a objetos especifica una clase, establece sus atributos y servicios. Los servicios incluyen constructores y métodos, estos últimos conformados por comandos y consultas. En la implementación, una clase con esta estructura define un **tipo de dato** a partir del cual se pueden **declarar variables**.

Como todo tipo, el tipo definido por una clase establece un **conjunto de valores** y un **conjunto de operaciones**. El conjunto de valores queda determinado por los valores de los atributos, el conjunto de operaciones lo definen los servicios provistos por la clase. Sin embargo, **una variable declarada de un tipo definido por una clase, no mantiene un valor dentro del tipo, sino una referencia a un objeto cuyo estado interno mantiene un valor del tipo.**

Caso de Estudio: Cuenta Corriente Bancaria

Un banco ofrece cajeros automáticos a través de los cuales los clientes pueden realizar depósitos, extracciones y consultar el saldo de su cuenta corriente. En el momento que se crea una cuenta corriente se establece su código y el saldo se inicializa en 0. También es posible crear una cuenta corriente estableciendo su código y saldo inicial. Una cuenta bancaria puede tener un saldo negativo hasta un máximo establecido por el banco, en ese caso la cuenta está "en descubierto". En el momento de la creación el saldo debe ser mayor o igual a cero. El código no se modifica, el saldo cambia con cada depósito o extracción. La clase brinda servicios para decidir si una cuenta está en descubierto, determinar el código de la cuenta con menor saldo entre dos cuentas y determinar cuál es la cuenta con menor saldo, entre dos cuentas.

En el diseño de la solución cada cuenta corriente bancaria se modela mediante el siguiente diagrama:

CuentaBancaria
<<atributos de clase>> maxDescubierto:real <<atributos de instancia>> codigo:entero saldo:real
<<constructores>> CuentaBancaria(c:entero) CuentaBancaria(c:entero,s:float) <<comandos>> depositar(mto:real) extraer(mto:real):boolean <<consultas>> obtenerCodigo():entero obtenerSaldo():entero obtenerMaxDescubierto():entero enDescubierto():boolean toString():String
Para crear una cuenta corriente bancaria codigo > 0 y saldo>=0.

depositar(mto:real)
Requiere mto> 0

extraer(mto:real):boolean
Requiere mto>0
Si mto > saldo+maxDescubierto retorna false y la extracción no se realiza.

La implementación de la clase que modela a una cuenta corriente bancaria es:

```
class CuentaBancaria{
//Atributos de clase
/*Define el monto máximo que se puede extraer sin fondos en la
cuenta*/
private static final int maxDescubierto=1000;
//Atributos de Instancia
/*El codigo se establece al crear la cuenta corriente bancaria  y no
cambia
El saldo va a ser siempre mayor a -maxDescubierto*/
private int codigo;
private float saldo;
// Constructores
public CuentaBancaria(int cod) {
      codigo = cod; saldo = 0;}
public CuentaBancaria(int cod, float sal) {
//Requiere sal>-maxDescubierto
      codigo = cod; saldo = sal;}
// Comandos
public void depositar(float mto){
//Requiere mto > 0
   saldo+=mto;}
public boolean extraer(float mto){
/*si el mto es mayor a saldo+maxDescubierto retorna false y la
extracción no se realiza*/
  boolean puede = true;
  if (saldo+maxDescubierto >= mto)
    saldo=saldo-mto;
  else
    puede = false;
  return puede;}
// Consultas
public int obtenerCodigo(){
```

```
  return codigo;}
public float obtenerSaldo(){
  return saldo;}
public float obtenerMaxDescubierto(){
  return maxDescubierto;}
public boolean enDescubierto(){
  return saldo < 0;}
public String toString(){
    return codigo+" "+saldo;}       }
```

El identificador `CuentaBancaria` está **sobrecargado**, el tipo o número de parámetros debe ser diferente en cada definición del constructor. El primero recibe como parámetro una variable cuyo valor se utiliza para inicializar el código de la cuenta corriente en el momento de la creación. El segundo constructor recibe dos parámetros, cada uno de los cuales inicializa a un atributo de instancia.

La clase `CuentaBancaria` define un tipo de dato a partir del cual es posible declarar variables. Los servicios provistos por la clase conforman el conjunto de operaciones del tipo. El conjunto del valores son todos los pares <c,s> con c>0 y s>-maxDescubierto que pueden mantenerse en el estado interno de cada objeto de clase `CuentaBancaria`.

El siguiente segmento está incluido en el método `main` de una clase tester, responsable de verificar los servicios de `CuentaBancaria`:

```
  CuentaBancaria cb, sm;
  cb =  new CuentaBancaria(111,1000);
  sm = new CuentaBancaria(112);
```

La instrucción:

```
  CuentaBancaria cb, sm;
```

declara dos variables de tipo clase `CuentaBancaria`. El **valor** de las variables `cb` y `sm` es `null`, es decir, ambas mantienen **referencias nulas**.

Las instrucciones:

```
  cb = new CuentaBancaria(111,1000);
  sm = new CuentaBancaria(112);
```

crean dos objetos de clase `CuentaBancaria`. Las variables `cb` y `sm` están ahora ligadas, es decir, el valor de cada variable es una **referencia ligada**.

En la primera instrucción el constructor está seguido de dos parámetros, se invoca entonces el constructor con dos parámetros definido en la clase `CuentaBancaria`. La segunda instrucción provoca la ejecución del primero constructor provisto por la clase.

Cualquiera sea el constructor que se ejecute, el **estado interno** de cada objeto de clase `CuentaBancaria` mantiene dos atributos, `codigo` y `saldo`. Como establece la especificación de requerimientos, el código se inicializa cuando se crea la cuenta corriente bancaria y no cambia.

Clases y Paquetes

Un **paquete** en Java es un contenedor que agrupa un conjunto de clases con características comunes. El uso de paquete favorece en primer lugar la reusabilidad porque permite utilizar clases que ya han sido diseñadas, implementadas y verificadas, fuera del contexto del sistema

que se está desarrollando. También favorece la eficiencia porque el lenguaje brinda un repertorio reducido de recursos, pero permite agregar otros **importando paquetes**.

En el caso de estudio de la Cuenta Corriente Bancaria la clase tester puede implementarse generando valores al azar, dentro de cierto rango:

```
import java.util.Random;
class Simulacion_CuentaBancaria{
public static void main (String[] args) {
  Random gen;
  gen = new Random();
  CuentaBancaria cta =   new CuentaBancaria(111,1000);
  int i = 0; int monto;
  int tipoMov; boolean desc=false;
  while (i<50 && !desc)  {
    i++;
    monto = gen.nextInt(500)+10;
    tipoMov = gen.nextInt(2)+1;
    if (tipoMov==1){
       cta.depositar(monto);
       System.out.println (i+" deposito " +
       monto+" Cuenta "+cta.obtenerCodigo()+
       " "+cta.obtenerSaldo());}
   else
      if (cta.extraer(monto))
           System.out.println (i+" extrajo "+ monto+ " Saldo "
+cta.obtenerSaldo());
      else
        System.out.println (i+" NO pudo extraer "+ monto+ " Saldo "
+cta.obtenerSaldo());}} }
```

La definición de la clase `Simulacion_CuentaBancaria` está precedida por una instrucción que **importa** al paquete `java.util.Random` que ofrece a la clase `Random`. Así, el lenguaje brinda algunos recursos básicos y permite utilizar otros importando los paquetes que los ofrecen.

La clase `Random` define un tipo de dato a partir del cual es posible crear instancias, desconociendo la representación de los datos y la implementación de las operaciones, por ejemplo `nextInt`. Solo se conoce su **signatura**, esto es, el tipo del resultado y el número y tipo de los parámetros.

En el simulador propuesto los valores de las variables `monto` y `tipoMov` se generan al azar. El primero toma valores entre 10 y 510 y el segundo toma los valores 1 o 2. Si `tipoMov` es 1 se realiza un depósito, sino una extracción. La cantidad de movimiento va a ser como máximo 50, aunque puede ocurrir que la variable `desc` tome el valor `true` antes de que `i` sea 50 y en ese caso el bloque iterativo termina.

En cada caso de estudio que requiera de un nuevo paquete, se describen las clases que ofrece. La clase `String`, provista por Java, está incluida dentro de los recursos básicos del lenguaje.

Variables y Alcance

La **declaración** de una variable establece su nombre, su tipo y su **alcance**. Java impone algunos requisitos para los nombres. El tipo puede ser elemental o una clase. El alcance de una

variable determina su **visibilidad**, es decir, el segmento del programa en el cual puede ser nombrada.

En Java una **variable declarada en una clase como atributo de clase o de instancia**, es visible en toda la clase. Si se declara como **privada**, solo puede ser usada dentro de la clase, esto es, el alcance es la clase completa.

En los casos de estudio propuestos en este capítulo los atributos, de instancia y de clase se declaran como privados y solo se usan **atributos de clase** para representar valores **constantes**. En particular, en la clase `CuentaBancaria` las variables `codigo` y `saldo` son los atributos de instancia de la clase y pueden ser usadas en cualquiera de los servicios provistos por la clase. Como se declaran privados, desde el exterior sus valores sólo pueden ser accedidos por los servicios públicos que brinda la clase. La variable `maxDescubierto` es un atributo de clase, que también solo es visible en la clase.

En Java una **variable declarada local a un bloque** se crea en el momento que se ejecuta la instrucción de declaración y se destruye cuando termina el bloque que corresponde a la declaración. El alcance de una variable local es entonces el bloque en el que se declara, de modo que solo es **visible** en ese bloque.

En la clase `CuentaBancaria` la variable `puede`, declarada e inicializada en el método `extraer`, solo es visible y puede ser accedida en el bloque de código de ese método.

```
public boolean extraer(float mto){
/*si mto es mayor a saldo+maxdescubierto retorna false y la
extracción no se realiza*/
  boolean puede = true;
  if (saldo+maxDescubierto >= mto)
    saldo=saldo-mto;
  else
    puede = false;
  return puede;}
```

Un **parámetro formal de un servicio** se trata como una variable local que se crea en el momento que comienza la ejecución del servicio y se destruye cuando termina. Se inicializa con el valor del argumento o parámetro real. El pasaje de parámetros en Java es entonces **por valor**.

En el método `extraer` la variable `mto` es un parámetro, su alcance es el bloque del comando. Fuera del comando, la variable no es visible.

Cada bloque crea un nuevo **ambiente de referencia**, formado por todos los identificadores que pueden ser usados. Los operandos de una **expresión** pueden ser constantes, atributos, variables locales y parámetros visibles en el ambiente de referencia en el que aparece la expresión.

Mensajes y Métodos

Cuando un objeto recibe un mensaje, su clase determina el método que se va a ejecutar en respuesta a ese mensaje. En todos los casos de estudio propuestos, cuando un objeto recibe un mensaje, el **flujo de control** se interrumpe y el control pasa al método que se liga al mensaje. Al terminar la ejecución del método, el control vuelve a la instrucción que contiene al mensaje. Java permite modelar también procesamiento paralelo, esto es, ejecutar varios métodos simultáneamente. En este libro no se presentan conceptos de paralelismo.

Si un método retorna un valor, el tipo de la expresión que sigue a la palabra `return` debe ser compatible con el tipo que precede al nombre del método en el encabezamiento.

La clase `CuentaBancaria` brinda dos comandos para modificar el saldo: `depositar` y `extraer`. Ambos reciben como parámetro una variable `mto` que se requiere no negativa.

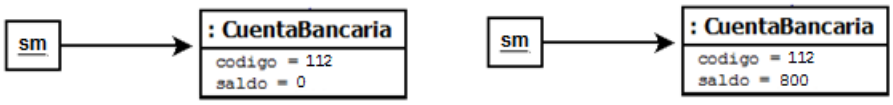
El siguiente segmento forma parte del método `main` en una clase tester de `CuentaBancaria`:

```
  int monto = 800;
  CuentaBancaria sm = new CuentaBancaria(112);
  sm.depositar (monto);
  if (!sm.extraer(100))
   System.out.println ("No se pudo realizar la extracción");
  System.out.println ("El saldo de la cuenta es "+sm.toString());
```

En la instrucción `sm.depositar(monto)` la cuenta bancaria ligada a la variable `sm` recibe el mensaje `depositar` que provoca la ejecución del método:

```
public void depositar(float mto){
//Requiere mto > 0
   saldo+=mto;}
```

El diagrama de objetos antes y después de la ejecución de `sm.depositar(monto)` es:



La clase tester no controla que la variable `monto` contenga un valor positivo porque la verificación se realiza con **valores fijos**. Si los valores los ingresa el usuario, es necesario **validar la entrada**.

El comando `depositar` se declara de tipo `void`, para indicar que no retorna un resultado. El valor del parámetro real `monto` se utiliza para inicializar el valor del parámetro formal `mto`. La asignación modifica el valor del atributo de instancia `saldo` incrementando su valor de acuerdo al valor del parámetro `mto`, en este ejemplo el valor del saldo de la cuenta bancaria era `0` y pasó a ser `800`.

Si el valor de la variable `mto` se modificara en el bloque que corresponde al método, el cambio no sería visible al terminar la ejecución de `depositar`, porque el **pasaje de parámetros es por valor**. Es decir, cuando el control retorna al método `main`, el parámetro real conserva el valor, aun cuando el parámetro formal se modifique.

En el método `main` la instrucción condicional `if (!sm.extraer(100))` envía el mensaje `extraer` al objeto ligado a la variable `sm`. Cuando el objeto recibe el mensaje se ejecuta el comando:

```
public boolean extraer(float mto){
/*si mto es mayor a saldo+maxDescubierto retorna false y la
extracción no se realiza*/
  boolean puede = true;
  if (saldo+maxDescubierto >= mto)
    saldo=saldo-mto;
  else
    puede = false;
  return puede;}
```

El parámetro formal `mto` se inicializa con el valor del parámetro real que en este caso es la constante `100`. La variable local `puede` se inicializa en `true`. En este ejemplo el condicional `(saldo+maxDescubierto >= mto)` computa `true`, de modo que la extracción puede realizarse, `saldo` se decrementa de acuerdo al valor del parámetro y la consulta retorna el valor `true`.

Cuando el comando `extraer` termina de ejecutarse, el control retorna a la instrucción condicional del método `main`. El resultado es justamente el valor de la expresión lógica del condicional.

Si en la clase tester la instrucción condicional fuera `if (!sm.extraer(10000))` nuevamente el objeto ligado a la variable `sm` recibe el mensaje `extraer`. Sin embargo, la expresión `(saldo+maxDescubierto >= mto)` en este caso computa `false`, de modo que la variable `puede` toma el valor `false` y el saldo no se modifica.

La variable local `puede` se declara del tipo que corresponde al resultado, ya que su valor es justamente el que retorna al terminar la ejecución del comando. Un método que modifica el valor de uno o más atributos del objeto que recibe el mensaje, es un comando, retorne o no un valor.

La última instrucción del segmento del método `main`:

```
System.out.println ("El saldo de la cuenta es "+sm.toString());
```

Envía el mensaje `toString()` al objeto ligado a la variable `sm`, que provoca la ejecución de la consulta:

```
public String toString(){
    return codigo+" "+saldo;}
```

La consulta retorna un resultado de tipo `String`, la expresión que sigue a la palabra `return` es la cadena de caracteres `112 400.0` que se genera concatenando los valores de los atributos `codigo` y `saldo`, separados por un espacio. Cuando la ejecución del método `toString()` termina, el control vuelve a la instrucción que contiene el mensaje `toString()`. La cadena que retorna se concatena con un cartel y se muestra en consola:

```
El saldo de la cuenta es 112 400.0
```

En este libro muchas clases brindan un método `toString()` que retorna la concatenación de los valores de los atributos del objeto que recibe el mensaje.

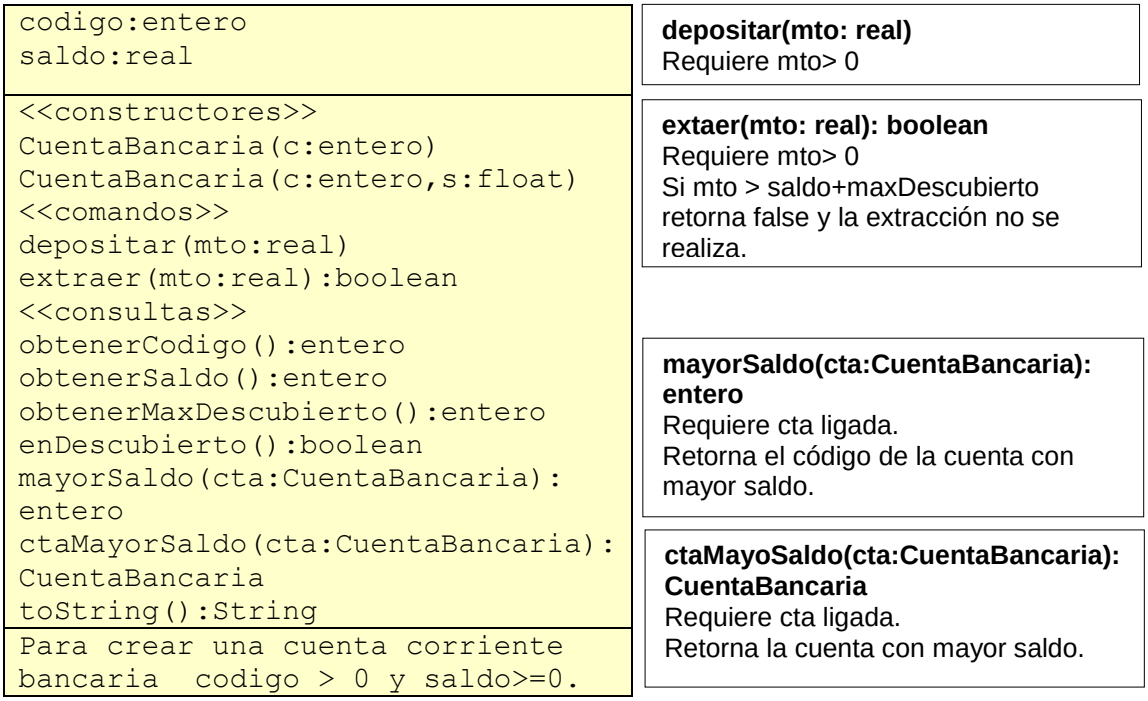
Parámetros y resultados de tipo clase

Un método puede recibir como parámetro o retornar como resultado a un objeto. En ambos casos, la variable que se recibe como parámetro o se retorna como resultado es de tipo clase.

Cuando en ejecución un mensaje se vincula a un método, el número de parámetros formales y reales es el mismo y los tipos son consistentes. Como se mencionó antes, en Java el pasaje de parámetros es por valor. Por cada parámetro formal se crea una variable y se le **asigna** el valor del parámetro real. Si el parámetro es de tipo clase, se asigna una referencia, de modo que el parámetro formal y el parámetro real mantienen una referencia a un mismo objeto.

Dada la siguiente ampliación para la especificación de la clase:

CuentaBancaria
<<atributos de clase>> maxDescubierto:real <<atributos de instancia>>



La implementación de los dos nuevos servicios es:

```
public int mayorSaldo(CuentaBancaria cta){
/*Retorna el código de la cuenta corriente bancaria  que tiene mayor
saldo.
Requiere cta ligada*/
  if (saldo > cta.obtenerSaldo())
    return codigo;
  else
    return cta.obtenerCodigo();}
public CuentaBancaria ctaMayorSaldo(CuentaBancaria cta){
/*Retorna la cuenta corriente bancaria que tiene mayor saldo.
Requiere cta ligada*/
   if (saldo > cta.obtenerSaldo())
    return this;
   else return cta;}
```

Dado el siguiente segmento del método `main` en una clase tester:

```
  CuentaBancaria cb1,cb2,cb3;
  cb1 =  new CuentaBancaria(21,1000);
  cb2 = new CuentaBancaria(22,800);
  System.out.println ("Codigo Cuenta Mayor Saldo "+
cb1.mayorSaldo(cb2));
  cb3 = cb1.ctaMayorSaldo(cb2);
  System.out.println ("Cuenta Mayor Saldo "+ cb3.toString());
```

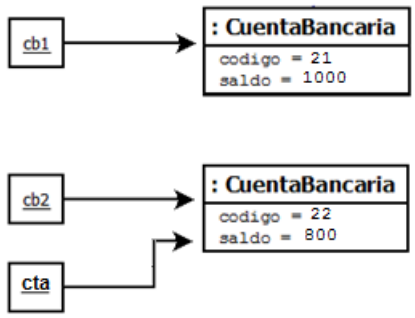
La instrucción que envía el mensaje `mayorSaldo` al objeto ligado a la variable `cb1`, provoca la ejecución del servicio `mayorSaldo` provisto por `CuentaBancaria`.

```
public int mayorSaldo(CuentaBancaria cta){
/*Compara el saldo de la cuenta corriente bancaria con el saldo de
cta y
retorna el CODIGO de la cuenta corriente bancaria que tiene mayor
saldo. Requiere cta ligada*/
  if (saldo > cta.obtenerSaldo())
    return codigo;
  else
```

```
    return cta.obtenerCodigo(); }
```

La consulta `mayorSaldo` recibe una cuenta corriente como parámetro y retorna el código de la cuenta con mayor saldo, comparando el saldo de la cuenta corriente que recibe el mensaje con el saldo de la cuenta corriente recibida como parámetro.

El diagrama de objetos cuando comienza la ejecución de `mayorSaldo` es:



El parámetro formal `cta` está ligada a un objeto de clase `CuentaBancaria`, que en este caso también está referenciado por el parámetro real `cb2`. Las variables `cb1` y `cb2` no son visibles en el bloque de código del método `mayorSaldo`. La variable `cta` solo es visible y puede ser usada en el bloque de código de `mayorSaldo`. Cuando el método termina, la variable `cta` se destruye, aun cuando el objeto sigue existiendo.

La ejecución de `cta.obtenerSaldo()` retorna el valor `800` que corresponde al saldo de la cuenta `cb2` en la clase tester. La expresión `(saldo > cta.obtenerSaldo())` computa `true` al comparar `1000` y `800` de modo que se ejecuta `return codigo` y el resultado es `21`, el código de la cuenta corriente con mayor saldo entre el objeto que recibió el mensaje y el que pasó como parámetro.

La consulta `mayorSaldo` es una operación binaria, los operandos son el objeto que recibe el mensaje y el parámetro. Los atributos del objeto que recibe el mensaje se acceden directamente, los atributos del objeto que se recibe como parámetro se acceden a través de las operaciones `obtenerCodigo()` y `obtenerSaldo()`.

En este libro, siguiendo los lineamientos de la programación estructurada, se adopta la convención de incluir una única instrucción de retorno al final del método, o dos instrucciones, únicamente en el caso de que el método incluya a una instrucción `if-else` con una instrucción simple en cada alternativa.

La instrucción:

```
  System.out.println ("Codigo Cuenta Mayor Saldo
"+cb1.mayorSaldo(cb2));
```

Muestra en consola:

```
Codigo Cuenta Mayor Saldo 21
```

La instrucción:

```
  cb3 = cb1.ctaMayorSaldo(cb2);
```

Envía el mensaje `ctaMayorSaldo` al objeto ligado a la variable `cb1`. El mensaje provoca la ejecución del método con el mismo nombre.

```
public CuentaBancaria ctaMayorSaldo(CuentaBancaria cta){
/*Retorna la cuenta corriente bancaria que tiene mayor saldo
Requiere cta ligada*/
```

```
   if (saldo > cta.obtenerSaldo())
    return this;
   else return cta;}
```

La consulta `ctaMayorSaldo` recibe una cuenta corriente como parámetro y retorna la cuenta corriente con mayor saldo, comparando el saldo de la cuenta que recibe el mensaje, con el saldo de la cuenta recibida como parámetro. El alcance de la variable de `cta` es el bloque del método `ctaMayorSaldo` que la recibe como parámetro. Solo se accede a los atributos de `cta` a partir de los servicios provistos por su clase.

La instrucción condicional `if (saldo > cta.obtenerSaldo())` compara nuevamente el saldo del objeto que recibió el mensaje con el saldo del objeto que pasó como parámetro. La palabra reservada `this` permite hacer referencia al objeto que recibe el mensaje, como la expresión condicional computa `true`, se ejecuta `return this` y retorna al método `main` el objeto que recibió el mensaje, que se liga a la variable `cb3`.

Si el método `main` de la clase tester incluye la instrucción:

```
  cb3 = cb2.ctaMayorSaldo(cb1);
```

El objeto ligado a la variable `cb2` recibe el mensaje `ctaMayorSaldo` con `cb1` como parámetro. En este caso la expresión `(saldo > cta.obtenerSaldo())` computa `false` de modo que se ejecuta el bloque del `else`, esto es `return cta`. Al terminar la ejecución del método `ctaMayorSaldo` y retornar el control al método `main`, el resultado se asigna a `cb3`.

El método `ctaMayorSaldo` recibe como parámetro a un objeto y retorna como resultado a un objeto. El tipo de la expresión que sigue a la palabra `return` se corresponde con el tipo que precede al nombre de la consulta en el encabezamiento.

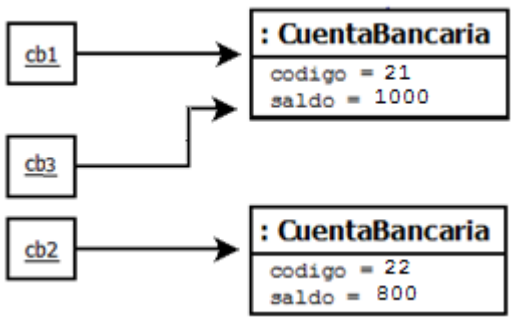
Tanto al terminar la ejecución de la instrucción:

```
  cb3 = cb1.ctaMayorSaldo(cb2);
```

como luego de ejecutarse:

```
  cb3 = cb2.ctaMayorSaldo(cb1);
```

el diagrama de objetos será:



Es decir, en ambos casos las variables `cb1` y `cb3` quedan ligadas a un mismo objeto.

La ejecución de:

```
  System.out.println ("Cuenta Mayor Saldo "+ cb3.toString());
```

Envía el mensaje `toString` al objeto ligado a `cb3`. El método `toString()` retorna un resultado de tipo `String`, esto es una cadena de caracteres. Expresado con mayor precisión, el método `toString()` retorna una referencia a un objeto de clase `String`. El mensaje

`System.out.println` requiere como parámetro un objeto de clase `String`. Cuando un parámetro es de tipo clase, el método recibe una referencia.

Ejercicio: Proponga un ejemplo en el cual las instrucciones

```
cb3 = cb1.ctaMayorSaldo(cb2);
```

y:

```
cb3 = cb2.ctaMayorSaldo(cb1);
```

generen diagramas de objetos diferentes.

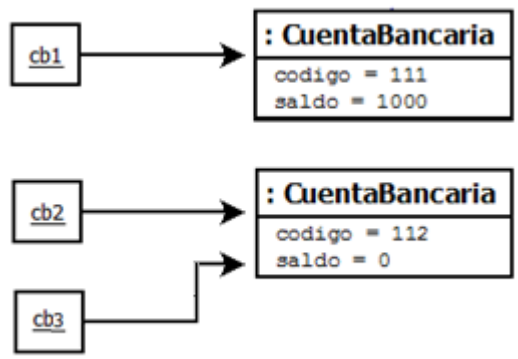
Identidad, igualdad y equivalencia

Cada objeto de software tiene una **identidad**, una propiedad que lo distingue de los demás. La **referencia** a un objeto puede ser usada como propiedad para identificarlo.

Si varias variables están ligadas a un mismo objeto, todas mantienen una misma referencia, esto es, comparten una misma identidad. La siguiente secuencia:

```
CuentaBancaria cb1 = new CuentaBancaria(111,1000);
CuentaBancaria cb2,cb3;
cb2 = CuentaBancaria(112);
cb3 = cb2;
```

Se representa gráficamente mediante el siguiente diagrama de objetos:



Las variables `cb2` y `cb3` tienen la misma identidad, mantienen referencias a un mismo objeto. Si el objeto recibe un mensaje que modifica su estado interno, como por ejemplo en la primera instrucción de:

```
cb2.depositar (200);
System.out.println (cb3.toString());
```

Se modifica el estado interno del objeto ligado a las variables `cb2` y `cb3`. La segunda instrucción muestra en consola:

```
111 200.0
```

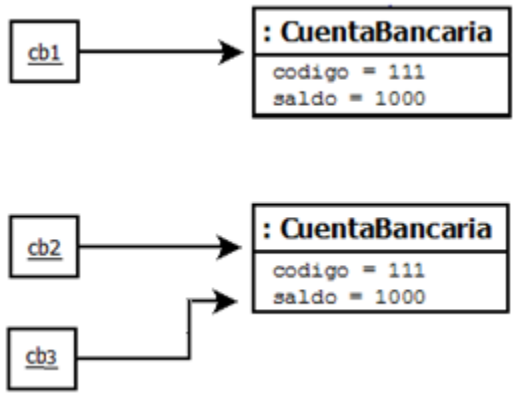
Como las dos variables están ligadas a un mismo objeto, el mensaje `depositar` modifica el atributo de instancia y el cambio es visible al ejecutarse el método `toString()`.

Los **operadores relacionales** comparan los valores de las variables. Cuando se aplica el operador de igualdad sobre variables de tipo clase, se comparan los valores de las variables; el valor de cada variable es una referencia nula o ligada a un objeto. Por ejemplo, en el siguiente segmento de código:

```
CuentaBancaria cb1,cb2,cb3;
```

```
cb1 = new CuentaBancaria(111,1000);
cb2 = new CuentaBancaria(111,1000);
cb3 = cb2;
```

Las variables `cb2` y `cb3` referencian a un mismo objeto, tienen entonces una misma identidad. El diagrama de objetos es:



Si a continuación se ejecutan las siguientes instrucciones:

```
boolean b1,b2;
b1 = cb2 == cb1;
b2 = cb2 == cb3;
```

La variable lógica `b1` toma el valor `false` porque `cb1` y `cb2` mantienen referencias a distintos objetos. La variable `b2` toma el valor `true` porque `cb2` y `cb3` mantienen un mismo valor, esto es, referencian a un mismo objeto.

Para comparar el **estado interno** de los objetos, se comparan los valores de los atributos de instancia. El método `main` en la clase tester puede incluir por ejemplo la siguiente instrucción condicional:

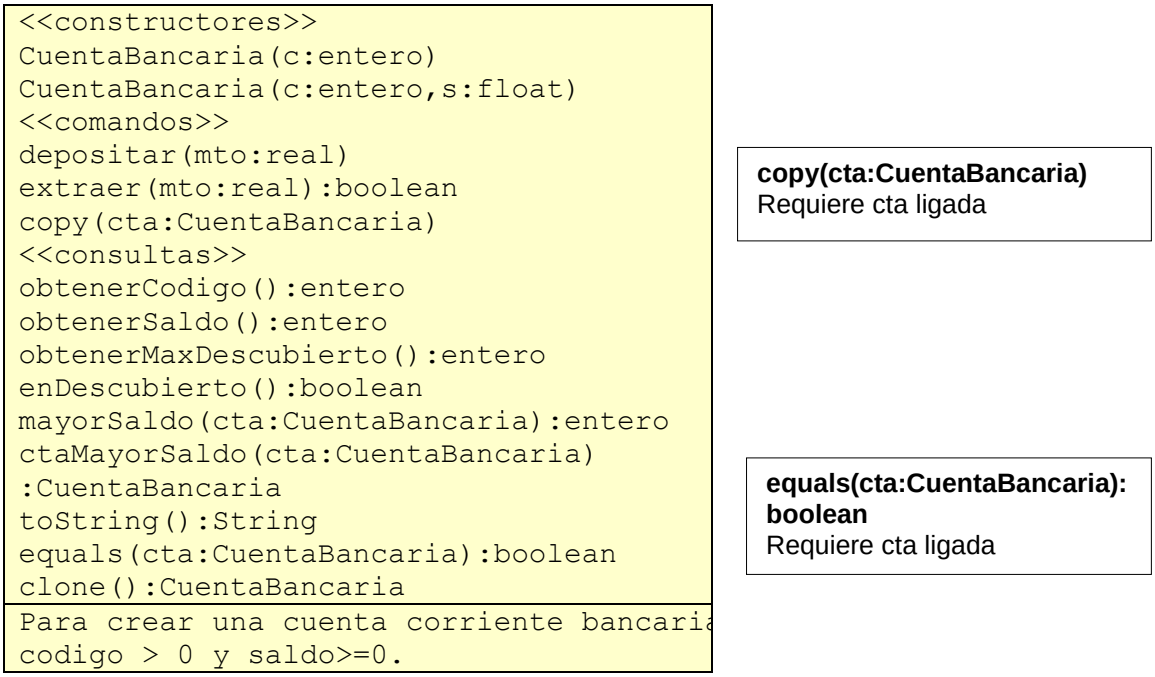
```
if (cb1.obtenerCodigo()==cb2.obtenerCodigo()&&
    cb1.obtenerSaldo() == cb2.obtenerSaldo())
```

En este caso la comparación de los estados internos puede resolverse con una expresión con dos subexpresiones. Si una clase tiene 10 atributos, cada vez que se comparan dos objetos es necesario comparar los 10 atributos. Más aun, si la representación interna de la cuenta bancaria cambia, es necesario modificar todas las clases que usan los servicios provistos por `CuentaBancaria` y comparan el estado interno.

Cuando una clase define un tipo de dato por lo general va a incluir un método que decide si dos objetos que son instancias de esa clase, son iguales. Por convención se utiliza el nombre `equals` para este método. También es habitual que una clase incluya métodos para copiar y clonar. El primero copia el estado interno de un objeto en otro. El segundo crea un clon del objeto que recibe el mensaje; el nuevo objeto puede evolucionar luego de manera autónoma al original. En este libro se adopta la convención de llamar `copy` y `clone` a estos métodos.

Dada la siguiente ampliación para la especificación de la clase `CuentaBancaria`:

CuentaBancaria
<<atributos de clase>> maxDescubierto:real <<atributos de instancia>> codigo:entero saldo:real



El método `equals` compara el estado interno del objeto que recibe el mensaje, con el estado interno del objeto que pasa como parámetro. El resultado debe ser `true` si el `codigo` del objeto que recibe el mensaje es igual al `codigo` del objeto que se liga a la variable `cta` y el `saldo` del objeto que recibe el mensaje es igual al `saldo` del objeto ligado a `cta`.

```
public boolean equals(CuentaBancaria cta){
//Requiere cta ligada
    return codigo == cta.obtenerCodigo() &&
           saldo == cta.obtenerSaldo();}
```

El estado del objeto que recibe el mensaje se accede directamente a través de las variables de instancia. El estado del objeto que pasa como parámetro se accede indirectamente a través de las operaciones provistas por la clase.

La consulta requiere que el parámetro esté ligado. Si no fuera así, esto es, si la variable `cta` mantiene una referencia nula, la expresión `cta.obtenerCodigo()` provoca la terminación anormal del programa.

Un diseño alternativo podría asignar al método equals la responsabilidad de controlar si el parámetro está ligado. En este caso la implementación sería:

```
public boolean equals(CuentaBancaria cta){
//Controla si cta está ligada
   boolean e = false;
   if (cta != null)
      e = codigo == cta.obtenerCodigo() && saldo ==
cta.obtenerSaldo();
   return e;}
```

Cualquiera sea el diseño y la implementación de `equals`, la clase tester puede usarlo para comparar el estado interno de los objetos:

```
if (cb1.equals(cb2))
```

El método `equals` es una **operación binaria**, un operando es el objeto que recibe el mensaje, el otro operando es el objeto que pasa como parámetro.

Los objetos ligados `cb1` y `cb2` tienen distinta **identidad** pero pueden considerarse **equivalentes**, en el sentido de que modelan a un mismo objeto del problema.

El operador relacional == decide si dos objetos tienen la misma identidad. El servicio `equals` provisto por la clase `CuentaBancaria` decide si dos objetos son equivalentes.

El método `copy` modifica el estado interno del objeto que recibe el mensaje, con el estado interno del objeto que pasa como parámetro. El estado del objeto que recibe el mensaje se accede directamente a través de las variables de instancia.

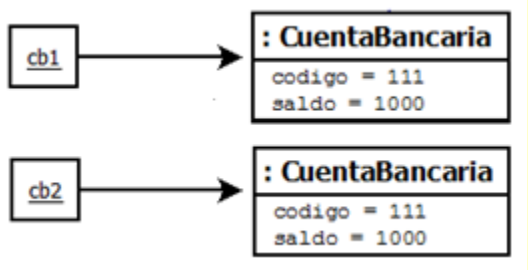
```
public void copy (CuentaBancaria cta) {
//Requiere cta ligada
  codigo = cta.obtenerCodigo();
  saldo = cta.obtenerSaldo();}
```

El estado del objeto que pasa como parámetro se accede indirectamente a través de las operaciones obtener provistas por la clase. Nuevamente el método requiere que la variable `cta` esté ligada.

Dado el siguiente segmento de código en la clase tester:

```
CuentaBancaria cb1,cb2;
cb1 = new CuentaBancaria(101,1000);
cb2 = new CuentaBancaria(102,500);
cb2.copy(cb1);
```

Cuando el objeto ligado a la variable `cb2` recibe el mensaje `copy` su estado interno se modifica con los valores del objeto ligado a `cb1`, el diagrama de objetos al terminar la ejecución del segmento, es entonces:



Si a continuación se ejecuta el siguiente segmento en la clase tester:

```
boolean b1 cb1==cb2;
boolean b2 cb1.equals(cb2);
```

Los valores de las variables `b1` y `b2` serán `false` y `true` respectivamente. Los objetos no tienen la misma identidad, pero son equivalentes. Si el objeto ligado a `cb1` recibe en mensaje que modifica su estado interno, el objeto ligado a `cb2` no modifica su estado interno.

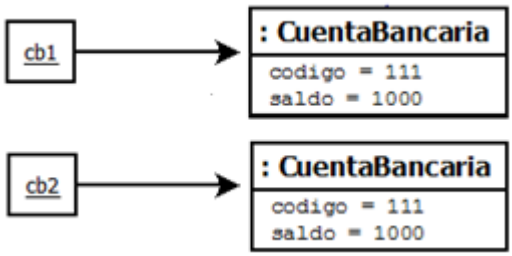
La consulta `clone` retorna la referencia a un objeto con el mismo estado interno que el objeto que recibe el mensaje.

```
public CuentaBancaria clone() {
  CuentaBancaria aux = new CuentaBancaria (codigo,saldo);
  return aux;}
```

Después de la ejecución del siguiente segmento:

```
CuentaBancaria cb1,cb2;
cb1 = new CuentaBancaria(111,1000);
cb2 = cb1.clone();
```

El diagrama de objetos será:



Es importante observar que `cb2` no estaba ligada en el momento en que se invocó al método `clone`. Si a continuación se ejecutan las siguientes instrucciones:

```
boolean b1 cb1==cb2;
boolean b2 cb1.equals(cb2);
```

las expresiones computan `false` y `true` para `b1` y `b2` respectivamente. Los objetos no tienen la misma identidad, pero son equivalentes. Si el objeto ligado a `cb1` recibe un mensaje que modifica su estado interno, el objeto ligado a `cb2` no modifica su estado interno.

Durante la ejecución de la consulta `clone()` las variables `cb1` y `cb2` **no son visibles**, están fuera del **alcance** del método. La variable `aux` es local a la consulta, cuando termina la ejecución de este método, la variable se destruye.

Representación en memoria

La memoria de una computadora es el dispositivo en el que se almacenan datos e instrucciones. Toda la información se almacena utilizando el sistema de numeración binario. Texto, números, imágenes, sonido y casi cualquier otra forma de información puede ser transformada en una sucesión de bits, o dígitos binarios, cada uno de los cuales tiene un valor de 1 o 0. La unidad de almacenamiento más común es el byte, igual a 8 bits.

La estructura de la memoria es así sumamente simple y puede graficarse como:

Dirección	Contenido
00000000	11011111
00000001	10101010
00000010	11110000
00000011	**00000001**

Cada celda de memoria tiene una **dirección** y un **contenido**. Un conjunto de celdas consecutivas pueden agruparse para definir un **bloque de memoria** que contenga a una **unidad de información**, por ejemplo, una cadena de caracteres. El contenido de una celda puede ser una dirección en memoria, en ese caso la celda contiene una referencia a otro bloque de memoria. En el diagrama de memoria propuesto, el contenido de la cuarta celda es la dirección de memoria de la segunda celda.

En general, en este libro mantiene una visión mucho más abstracta de la memoria, no menciona direcciones o la representación binaria de los valores almacenados. Los conceptos de variable y tipo de dato nos permiten justamente mantener una visión de alto nivel con abstracción.

En Java el tipo de una variable puede ser elemental o una clase. La **representación interna en memoria** es diferente en cada caso. Una variable de tipo elemental almacena un valor de su tipo. Una variable de tipo clase almacena una **referencia** a un objeto de software de su clase.

Cuando en ejecución se alcanza una declaración de una variable local de tipo elemental se reserva un **bloque de memoria**. Por ejemplo:

```
int i=3;
```

reserva un bloque de memoria ligado a la variable `i` y se almacena el valor 3, o mejor dicho, la representación binaria del valor 3. Así, el bloque de memoria ligado a la variable mantiene un valor dentro del conjunto de valores que determina el tipo de dato. La asignación:

```
i=i+1;
```

Modifica el valor almacenado en el bloque de memoria ligado a `i`, almacenando el valor que resulta de computar la expresión `i+1`. La instrucción:

```
if (i==3)
```

Compara el valor almacenado en la celda de memoria ligada a la variable `i` con la constante 3. Se comparan las representaciones binarias de los valores, aun cuando se mantiene una visión más abstracta.

Cuando en ejecución se alcanza una **declaración de variable** cuyo tipo es una clase, la variable es de **tipo clase**, se reserva también un **bloque de memoria** que mantendrá inicialmente una **referencia nula o referencia no ligada**.

```
CuentaBancaria cb;
```

La **creación de un objeto** reserva un nuevo bloque de memoria para mantener el estado interno del objeto. El valor de la variable no pertenece al conjunto de valores que determina el tipo, sino que es una **dirección al bloque de memoria** en el que se almacena el estado interno del objeto.

```
cb = new CuentaBancaria (111,1000);
```

La variable `cb` mantiene la referencia al bloque de memoria en el cual reside efectivamente el objeto de la clase `CuentaBancaria`.

La variable `cb` está **ligada** al objeto que le fue asignado, esto es, es una referencia al estado interno del objeto en el cual se almacenan los valores de los atributos. Los atributos de instancia determinan la estructura del estado interno del objeto, no de la variable.

La ejecución de la secuencia:

```
CuentaBancaria cb;
cb = new CuentaBancaria (111,1000);
```

es equivalente a:

```
CuentaBancaria cb = new CuentaBancaria (111,1000);
```

En términos de la memoria, el significado es:

- Reservar un bloque de memoria para almacenar el valor de la variable `cb`.
- Reservar un bloque en memoria para almacenar el estado interno de un objeto de software de clase `CuentaBancaria`.
- Almacenar en la variable `cb` la dirección del bloque de memoria que almacena el estado interno del objeto creado.
- Invocar al constructor con dos parámetros de clase `CuentaBancaria`.

El diagrama de objetos es un modelo abstracto de la representación en memoria, no grafica la memoria o las direcciones en memoria, sino el estado interno de los objetos y las referencias.

Dada la siguiente secuencia de instrucciones:

```
CuentaBancaria cb1,cb2;
cb1 = new CuentaBancaria(13,450);
cb2 = cb1;
```

La segunda asignación almacena en el bloque de memoria ligado a la variable `cb2`, el mismo valor que almacena el bloque de memoria ligado a la variable `cb1`. De modo que las dos variables mantienen la referencia a un mismo objeto.

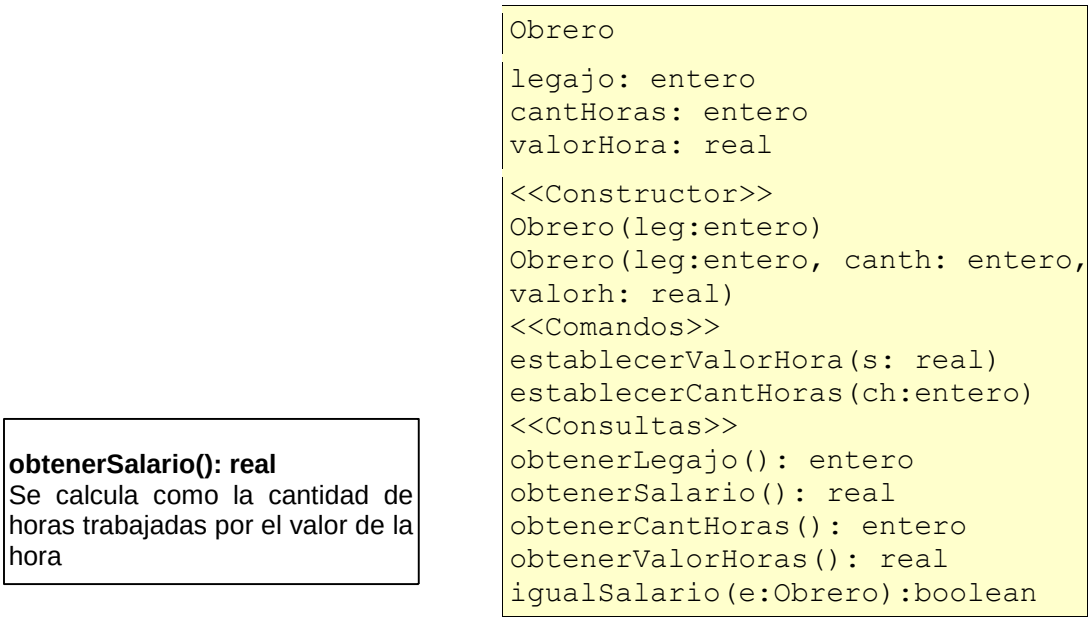
La instrucción:

```
if (cb1 ==cb2)
```

Compara el valor almacenado en el bloque de memoria ligado a la variable `cb1`, con el valor almacenado en el bloque de memoria ligado a `cb2`.

Problemas Propuestos

1. En una empresa constructora se desea almacenar y procesar toda la información requerida para computar el costo de cada obra. Entre los costos se incluyen los salarios de los obreros, que se computan dependiendo de la cantidad de horas trabajadas y el valor de la hora acordado en el contrato. El siguiente diagrama modela parcialmente el problema:



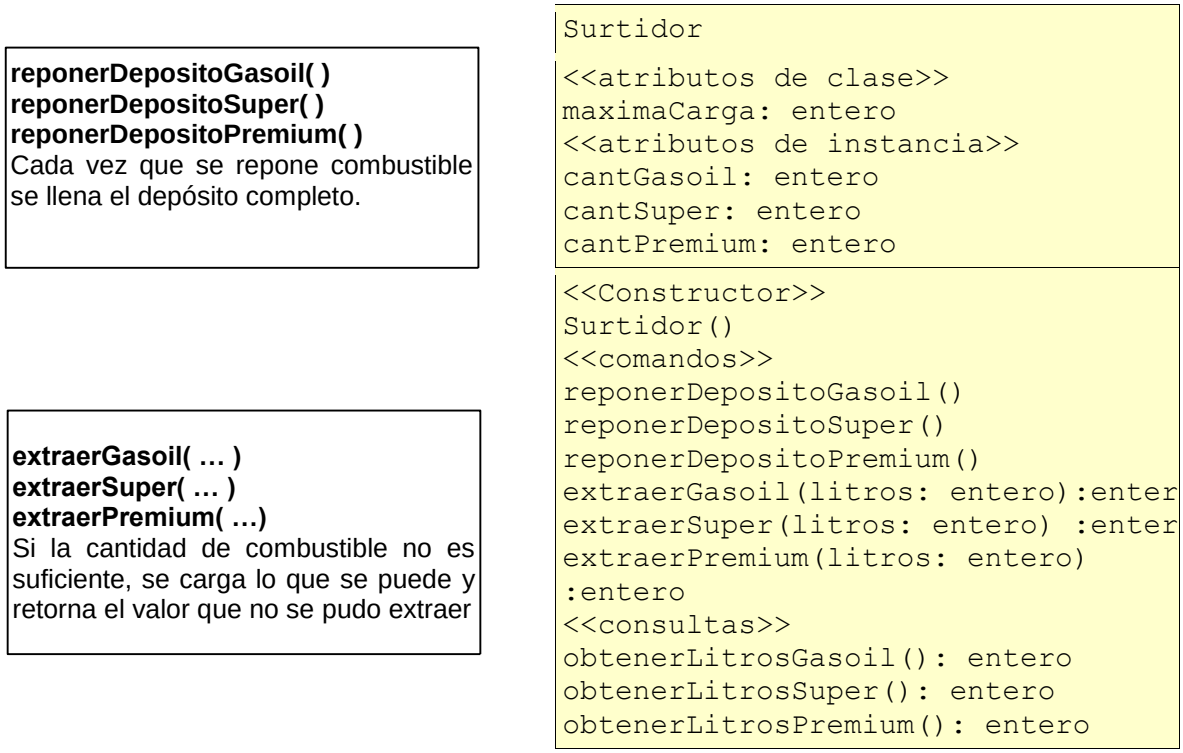
a. *Implemente en Java la clase Obrero modelada por el diagrama.*

b. *Escriba una clase Tester con un método main() que:*

- ✓ *Solicite al usuario los datos de un Obrero (legajo, cantidad de horas trabajadas y valor de la hora), cree un objeto ligado a una variable emp1 de la clase Obrero, usando el constructor con tres parámetros.*
- ✓ *Cree un objeto ligado a una variable emp2 de clase Obrero, con Legajo 111 y luego establezca la cantidad de horas en 40 y el valor de la hora en $48.*
- ✓ *Cree un objeto ligado a una variable emp3 de clase Obrero, con Legajo 112 y luego establezca la cantidad de horas en 20 y el valor de la hora en $96.*

- ✓ *Muestre en consola un mensaje indicando si los objetos ligados a las variables emp1 y emp2 tienen el mismo Salario.*
- ✓ *Muestre en consola un mensaje indicando si los objetos ligados a las variables emp2 y emp3 tienen el mismo Salario.*

2. Una estación de servicio cuenta con surtidores de combustible capaces de proveer Gasoil, Nafta Super y Nafta Premium. Todos los surtidores tienen capacidad para almacenar un máximo de 20000 litros de cada combustible. En cada surtidor se mantiene registro de la cantidad de litros disponibles en depósito de cada tipo de combustible, esta cantidad se inicializa en el momento de crearse un surtidor con la cantidad máxima. En cada surtidor es posible extraer o reponer combustible. En ocasiones la cantidad de un tipo de combustible particular en un surtidor específico puede no ser suficiente para completar una extracción, en ese caso se carga lo que se puede, el surtidor queda vacío y retorna el valor que no se pudo cargar. Cuando se repone un combustible en el surtidor, se llena el depósito completo de ese combustible. Cada surtidor puede modelarse con el siguiente diagrama:



a. *Implemente en Java la clase descripta.*
b. *Escriba una clase SimulacionSurtidor con un método main() que permita verificar los servicios provistos por la clase Surtidor de acuerdo al siguiente algoritmo:*

Algoritmo simulador
n = leer cantidad de iteraciones
repetir n veces testSurtidor

Algoritmo testSurtidor
mostrar la cantidad actual en el depósito de cada combustible

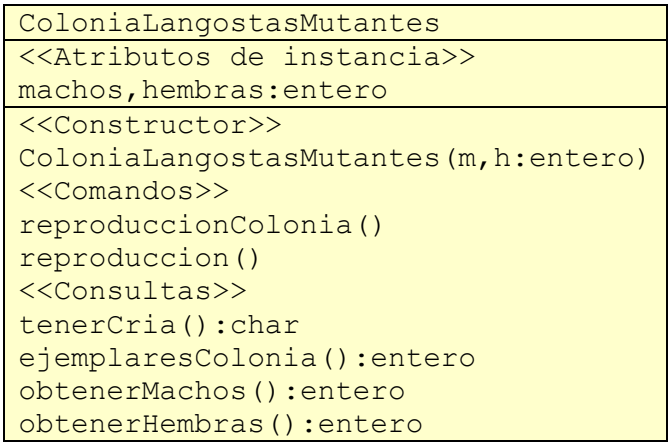
opción= número al azar entre 0 y 32 según opción sea:
Entre 0 y 9: leer y validar litros a cargar y cargar Gasoil
Entre 10 y 19: leer y validar litros a cargar y cargar Super
Entre 20 y 29: leer y validar litros a cargar y cargar Premium
30: reponer Deposito Gasoil
31: reponer Deposito Super
32: reponer Deposito Premium

La validación controla que el número leído sea mayor a 5 y menor a 50.

3. Se conoce como Langostas Mutantes a una variedad de las langostas que viven cerca de las centrales nucleares. Este tipo de langosta tiene dos características que la diferencia del resto de la especie: Por un lado, como todo animal que vive cerca de centrales nucleares, tiene tres ojos; además, su ciclo de reproducción es realmente extraño. La reproducción de la Langosta Mutante se puede describir de la siguiente manera:

La hembra langosta tiene una cría. Sí la cría es hembra entonces la langosta madre finaliza su ciclo de reproducción actual. Sí la cría es macho entonces la langosta automáticamente se reproducirá una vez más siguiendo el comportamiento explicado.

El siguiente diagrama modela la clase ColoniaLangostasMutantes:



reproducción() computa la reproducción de una hembra.

reproduccionColonia() modifica el número de machos y hembras a partir de la reproducción de una hembra

tenerCria() retorna, en forma aleatoria, un carácter indicando el género de la cría

ejemplaresColonia() retorna la suma de machos y hembras.

a. Implemente la clase ColoniaLangostasMutantes Tenga en cuenta que las hembras recién nacidas necesitan cierto proceso de maduración antes de reproducirse, es por eso que si nacen en una ejecución de reproducciónColonia() recién se reproducen en la siguiente.

b. Implemente una clase Tester que simule 5, 10 y 15 reproducciones en la colonia, generando en cada caso 4 pares de enteros al azar que representen los valores iniciales de machos y hembras y mostrando en consola el estado inicial y final.

4. Gran parte de las aplicaciones de software requieren representar y manipular imágenes. Una imagen digital puede representarse a través de una matriz de píxels. Un píxel es la menor unidad de color en una imagen digital.

Cuando observamos una imagen en la computadora no percibimos cada pixel particular, pero si aumentamos la imagen, por ejemplo un 600%, cada pixel va a ser distinguible. La percepción humana de la luz también es muy limitada y depende de nuestros sensores del color.

Una manera de representar todo el espectro de colores es a través de la combinación de tres colores primarios: azul, verde y rojo. De modo que cada color puede codificarse como una terna de números, el primero representa la cantidad de color rojo, el segundo la cantidad de color verde y el tercero la cantidad de color azul. El rango para cada valor es 0..255. Combinando el máximo de los tres se obtiene el blanco (255, 255, 255). La ausencia de los tres produce el negro (0, 0, 0). El rojo es claramente (255, 0, 0). Cuando se mantiene el mismo valor para las tres componentes se obtiene gris. La terna (50, 50, 50) representa un gris oscuro, (150, 150, 150) es un gris más claro. Esta representación se llama modelo RGB.

```
Color
<<Atributos de instancia>>
rojo: entero
azul: entero
verde : entero
<<Constructor>>
Color()
<<Comandos>>
variar(val:entero)
variarRojo(val:entero)
variarAzul(val:entero)
variarVerde(val:entero)
establecerRojo(val:entero)
establecerAzul(val:entero)
establecerVerde(val:entero)
copy(p:Color)
<<Consultas>>
obtenerRojo():entero
obtenerAzul():entero
obtenerVerde():entero
esRojo():boolean
esGris():boolean
esNegro():boolean
complemento():Color
equals(p:Color):boolean
clone():Color
```

Color( ) inicializa la representación en blanco

variar(val:entero), varia el color en un valor fijo, modifica cada componente de color sumándole si es posible, el valor val. Si sumándole el valor dado a una o varias componentes se supera el valor 255, dicha componente queda en 255. Si el argumento es negativo la operación es la misma pero en ese caso el mínimo valor que puede tomar una componente, es 0.

variarRojo, variarVerde y variarAzul son similares a **variar(val:entero)** pero solo modifican una componente de color

esRojo, esGris, esNegro retornar verdadero si el pixel representa el color que indica el mensaje.

complemento() retorna un objeto de clase Color que es el complemento para alcanzar el color blanco.

a. *Implemente la clase Color modelada por el diagrama*

b. *Defina un conjunto de casos de prueba fijos e implemente una clase tester para estos valores. Para visualizar la conformación de colores en el modelo RGB podemos usar el programa graficador de paleta.*

c. *Dada la siguiente secuencia de instrucciones:*

```
Color S1,S2,S3,S4,S5;
S1 = new Color(100,90,120);
S2 = new Color(55,110,100);
S3 = new Color(110,110,100);
S4 = S1;
S1 = S3;
S2 = S1;
```

```
S1 = new Color(80,80,80);
```

Elabore un diagrama de objetos que muestre la evolución de las referencias y continúelo a partir de cada secuencia:

`S2 = S1.clone();`
`boolean b1,b2;`
`b1 = S1==S2;`
`b2 = S1.equals(S2);`
	`S3.copy(S1);`
	`boolean b1,b2;`
	`b1 = S1==S3;`
	`b2 = S1.equals(S3);`
		`S4 = S1;`
		`boolean b1,b2;`
		`b1 = S1==S4;`
		`b2 = S1.equals(S2);`

Dependencia y Asociación entre Clases

Los procesos de abstracción y clasificación permiten identificar, representar y agrupar a entidades que son semejantes de acuerdo a algún criterio. El criterio adoptado en una clasificación permite decidir si una entidad **pertenece** a una clase o no.

En la construcción de un modelo para una aerolínea seguramente se agrupen en una clase los vuelos que se realizan, en otra las ciudades a las que llegan los vuelos, otra clase estará conformada por los pilotos habilitados para comandar una nave y otra por las tarjetas de embarque.

Las entidades de una clase son semejantes de acuerdo al criterio elegido y además se **relacionan** de la misma manera con las entidades de otras clases. Una forma de relación es la **asociación** y se produce cuando el modelo de un objeto del problema **contiene** o **puede contener** al modelo de otro objeto del problema. Por ejemplo, todo vuelo **tiene** una ciudad como destino final y **puede tener** una o más ciudades en las que hace escala.

Otra forma de relación es la **dependencia** y se produce cuando el modelo de un objeto del problema **usa** al modelo de otro objeto. Por ejemplo, un pasajero usa su tarjeta de embarque para despachar su equipaje.

En la programación orientada a objetos el punto de partida para la construcción de un sistema es un proceso de abstracción y clasificación. **Los objetos de una clase** se caracterizan por los mismos atributos y comportamiento, pero además **comparten entre sí el mismo modo de relacionarse con objetos de otras clases**.

Un objeto asociado a otro objeto, **tiene un** atributo de su clase. Por ejemplo, un objeto de la clase *Vuelo* **tiene un** atributo *destino* de la clase *Ciudad*. La relación entre los objetos provoca una relación entre las clases, que se dicen **asociadas**.

Un objeto depende de un objeto de otra clase, si **usa un** atributo de esta clase. Por ejemplo, un objeto de la clase *Pasajero* **usa un** atributo de la clase *TarjetaEmbarque*. La relación entre los objetos provoca una relación de dependencia entre las clases.

Cuando una clase está asociada a otra clase, los cambios en la segunda pueden tener un impacto en la primera. El mismo impacto se produce si se modifica una clase de la cual depende otra. Uno de los objetivos de la programación orientada a objetos es reducir el impacto de estos cambios.

Dependencia entre clases

Cuando una clase declara una variable local o un parámetro de otra clase, decimos que existe una **dependencia** entre la primera y la segunda y surge de una relación del tipo **usaUn** entre los objetos.

Caso de Estudio: Fábrica de Juguetes

*En una fábrica de autos de juguete una parte de la producción la realizan **robots**. Cada robot tiene una carga de energía que se va consumiendo a medida que ejecuta las órdenes que recibe. Cada robot es capaz de conectarse de modo tal que se recargue su energía hasta su capacidad máxima de 5000 unidades. Esta acción puede ejecutarse ante una orden externa o puede iniciarla el robot mismo cuando su energía está por debajo de las 100 unidades.*

*Cada robot cuenta con piezas de diferentes tipos: ruedas, ópticas y chasis. La cantidad de piezas se incrementa cuando un robot recibe una orden de abrir una **caja de piezas** y se decrementa cuando arma un vehículo. Cada caja tiene piezas de todos los tipos. Desarmar una caja cualquiera demanda 50 unidades de energía. Armar un auto consume 70 unidades de energía, 4 ruedas, 6 ópticas y 1 chasis. Es responsabilidad de cada servicio que consuma energía recargarla cuando corresponda.*

Robot
<<atributos de clase>> energiaMaxima : 5000 energiaMinima : 100 <<atributos de instancia>> nroSerie:entero energia: entero ruedas: entero opticas: entero chasis: entero
<<constructor>> Robot(nro:entero,caja:Caja) <<comandos>> abrirCaja (caja: Caja) recargar() armarAuto() <<consultas>> obtenerEnergia (): entero obtenerChasis () : entero obtenerRuedas () : entero obtenerOpticas () : entero cantAutos() : entero clone():Robot equals(Robot r):boolean
<<Responsabilidades>> El constructor establece la energía en el valor máximo y las cantidades de piezas en 100. Todos los servicios que consumen energía deciden recargar cuando energía es menor que la mínima.

Caja
<<atributos de instancia>> ruedas: entero opticas: entero chasis : entero
<<constructor>> Caja (r,o,ch: entero) <<comandos>> establecerRuedas(n:entero) establecerOpticas(n:entero) establecerChasis(n:entero) vaciar() <<consultas>> obtenerChasis () : entero obtenerRuedas () : entero obtenerOpticas () : entero equals(c:Caja):boolean

Requiere que haya piezas disponibles para armar un auto	

`recargar()` recarga la energía del robot hasta llegar al máximo.

`abrirCaja (caja:Caja)` aumenta las piezas disponibles de acuerdo a las cantidades de la caja y la vacía. Requiere `caja` ligada.

`armarAuto()` decrementa las piezas disponibles, requiere que se haya controlado si hay piezas disponibles antes de enviar el mensaje `armarAuto` a un robot.

`cantAutos():entero` retorna la cantidad de autos que puede armar el robot con las piezas que tiene disponibles, sin desarmar una caja.

En este caso, los valores de los atributos de instancia se establecen en la creación del objeto y se modifican cuando se arman autos o se abren cajas.

La implementación parcial de `Robot` será:

```
class Robot {
//atributos de clase
private static final int energiaMaxima = 5000;
private static final int energiaMinima = 100;
//atributos de instancia
private int nroSerie;
private int energia;
private int ruedas;
private int opticas;
private int chasis;
//Constructor
public Robot (int nro,Caja caja){
  nroSerie = nro;
  energia=energiaMaxima;
  ruedas = caja.obtenerRuedas();
  opticas = caja.obtenerOpticas();
  chasis = caja.obtenerChasis();
  caja.vaciar();}
//Comandos
public void recargar(){
  energia=energiaMaxima;}
public void armarAuto () {
/*Requiere que se haya controlado si hay piezas  disponibles*/
    ruedas -= 4 ;
    opticas -=6;
    energia -= 70;
    chasis --;
//Controla  si es necesario recargar  energía
    if (energia < energiaMinima)
      this.recargar(); }
public void abrirCaja (Caja caja) {
/*Aumenta sus cantidades según las de la caja y la vacía. Requiere
caja ligada*/
  ruedas += caja.obtenerRuedas();
  opticas += caja.obtenerOpticas();
  chasis += caja.obtenerChasis();
  energia -= 50;
```

```
  caja.vaciar();
/*Controla  si es necesario recargar energía*/
 if (energia < energiaMinima)
    this.recargar();}
//Consultas
public int obtenerRuedas(){
  return ruedas;}
public int obtenerOpticas(){
  return opticas;}
public int obtenerChasis(){
  return chasis;}
public int obtenerNroSerie(){
  return nroSerie;}
public int obtenerEnergia(){
  return energia;}
public String toString(){
  return nroSerie+" "+ruedas+" "+opticas+" "+chasis;}}
```

Entre las clases `Robot` y `Caja` se establece una relación de **dependencia**. Un parámetro de un servicio provisto por la clase `Robot`, es de tipo `Caja`.

El comando `abrirCaja(Caja caja)` provoca un **efecto colateral** sobre el parámetro. Es decir, no solo modifica el estado interno del robot que recibe el mensaje, sino también sobre la caja que pasa como parámetro. Si se modifica la signatura de los servicios provistos por la clase `Caja`, los cambios impactan en la clase `Robot`.
Ejercicio: Implemente la clase Caja y complete la clase Robot de acuerdo al diseño.

La clase `FabricaJuguetes` usa los servicios provistos por Robot:

```
class FabricaJuguetes{
…
public void producir() {
  Caja caja = new Caja(100,150,25);
  Robot unRobot;
  unRobot = new Robot(111,caja);
  …
  System.out.println(caja.obtenerRuedas());
  …
  if (unRobot.cantAutos() == 0)
    unRobot.abrirCaja(caja);

  unRobot.armarAuto();
  …}}
```

Los atributos, los comandos y las consultas de la clase `Caja` tienen los mismos nombres que algunos atributos y servicios de la clase `Robot`, cuando un objeto reciba un mensaje, su clase determina el método que va a ejecutarse.

Por ejemplo, en `caja.obtenerRuedas()` el mensaje `obtenerRuedas()` lo recibe un objeto de clase `Caja`, de modo que se ejecuta el método provisto por esa clase. Si en el método `producir()` de la clase `FabricaJuguetes`, el objeto `unRobot` recibe el mensaje `obtenerRuedas()` se ejecuta el método provisto por la clase `Robot`, porque `unRobot` es de clase `Robot`.

La clase `FabricaJuguetes` **depende** de las clases `Robot` y `Caja` porque declara variables locales de esas clases.

Asociación entre clases

Cuando una clase **tiene un atributo** de otra clase, ambas clases están **asociadas** y la relación es de tipo **tieneUn**. En general, entre dos clases asociadas también hay una relación de dependencia, algunos de los servicios provistos por una clase recibirán parámetros o retornarán un resultado de la clase asociada. Así, una relación de tipo **tieneUn** en general provoca una relación de tipo **usaUn**.

Caso de Estudio: Signos Vitales

Los signos vitales son medidas de variaciones fisiológicas que permiten valorar las funciones corporales básicas. Dos de los principales signos vitales son la temperatura corporal y la presión arterial. Se considera que existe un principio de alarma cuando estos valores escapan de los umbrales establecidos. Los valores de la presión sanguínea se expresan en kilopascales (kPa) o en milímetros del mercurio (mmHg). Para convertir de mmHg a kPa el valor se multiplica por 0,13. La temperatura se mide en centígrados.

PresionArterial
<<atributos de clase>> umbralMax,umbralMin :real <<atributos de instancia>> maxima,minima :real
<<Constructores>> PresionArterial(ma,mi:real) <<Consultas>> obtenerMaximaMM():real obtenerMinimaMM():real obtenerMaximaHP():real obtenerMinimaHP().real obtenerPresionPulsoMM():real obtenerPresionPulsoHP():real alarmaHipertension():boolean
Requiere máxima > mínima y ambo mayores a 0. Los valores están expresados en milímetros de mercurio

Signos Vitales
<<atributos de clase>> umbralTemp:real <<atributos de instancia>> temperatura: real presion :PresionArterial
<<Constructores>> SignosVitales(t:real, p:PresionArterial) <<Consultas>> obtenerTemperatura():real obtenerPresion():PresionArterial alarma ():boolean
Requiere temperatura > 0, expresada en grados.

El **estado interno** de cualquier objeto de clase `PresionArterial` mantiene los valores de dos atributos de instancia, `maxima` y `minima`, representados en milímetros de mercurio. Los valores de `maxima` y `mínima` expresados en hectopascales no se mantienen en el estado interno, se computan cuando se invocan los métodos `obtenerMaximaHP()` y `obtenerMinimaHP()`.

La clase `PresionArterial` fue implementada en el capítulo anterior. La implementación de `SignosVitales` es:

```
class SignosVitales{
//Atributos de clase
private static final float umbralTemp=38.0;
//Atributos de instancia
private float temperatura;
private PresionArterial presion ;
//Constructor
```

```
public SignosVitales (float t, PresionArterial p){
//Requiere t > 0, expresada en grados
  temperatura = t;
  presion = p;}
//Consultas
public float obtenerTemperatura (){
  return temperatura;}
public PresionArterial obtenerPresion (){
  return presion ;}
public boolean alarma(){
  return temperatura>umbralTemp || presion.alarmaHipertension();}
}
```

Las clases `SignosVitales` y `PresionArterial` están asociadas. La clase `SignosVitales` **tieneUn** atributo de clase `PresionArterial`. Además, `SignosVitales` **usaA** `PresionArterial`, su constructor recibe un parámetro de clase `PresionArterial` y la consulta `obtenerPresion()` retorna un resultado de esa misma clase.

La clase `SignosVitales` puede acceder a cualquiera de los miembros públicos de la clase `PresionArterial`. Los atributos han sido declarados como privados, por lo tanto no son accesibles.

La clase `Control` usa los servicios provistos por `SignosVitales` y `PresionArterial`:

```
class Control{
…
public void controlDiario(int mi1,int ma1,
                          int mi2, int ma2,
                          float t1,float t2){
PresionArterial p6Hs, p12Hs;
SignosVitales sv6Hs,sv12Hs;
p6Hs = new PresionArterial(mi1,ma1);
p12Hs= new PresionArterial(ma1,ma2);
sv6Hs = new SignosVitales(t1,p6Hs);
sv12Hs = new SignosVitales(t2,p12Hs);
if (sv6Hs.obtenerTemperatura() ==sv12Hs.obtenerTemperatura())
  System.out.println("Temperatura Estable");}
}
```

Existe en este caso una relación de dependencia entre `Control` y las clases `SignosVitales` y `PresionArterial`.

Representación por referencia de clases asociadas

La representación a través de referencias surge para modelar la asociación entre clases, como se ilustra con el siguiente caso de estudio:

Caso de Estudio: Aliens y Naves

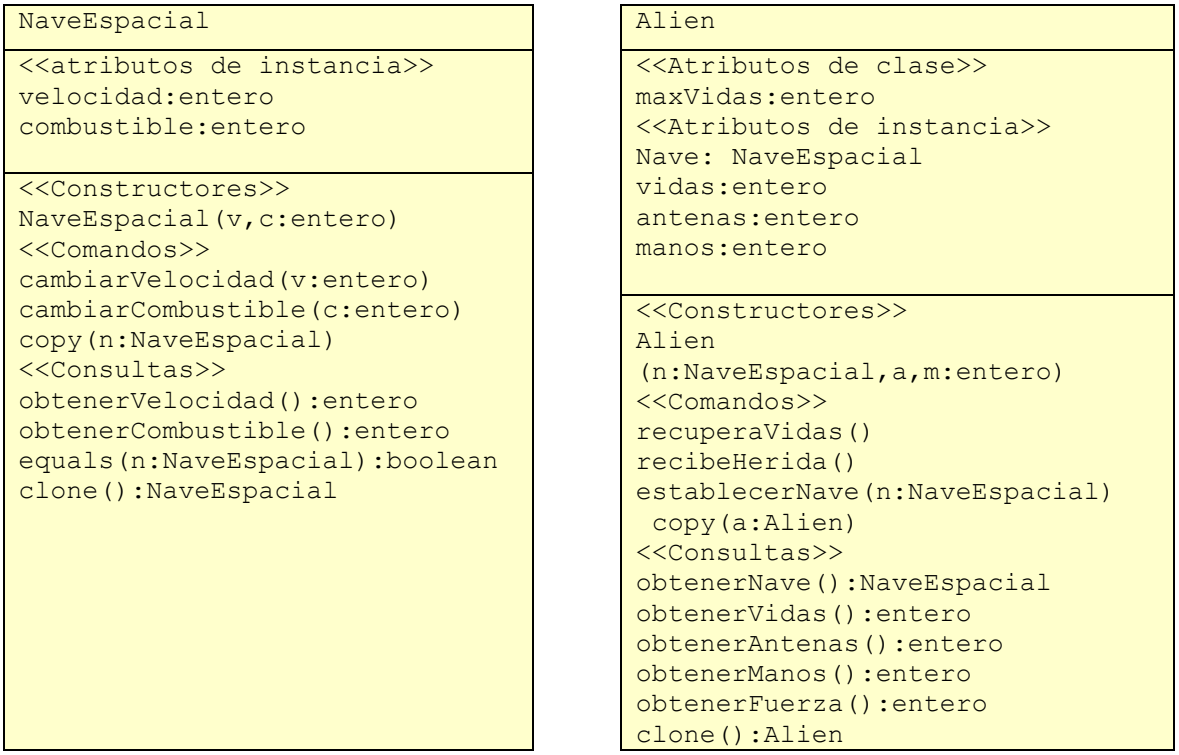
En un videojuego algunos de los personajes son aliens. Los aliens tienen cierta cantidad de antenas y de manos, que determinan su capacidad sensora y su capacidad de lucha respectivamente.

Cada alien tiene un nombre y una cantidad de vidas que inicialmente es 5 y se van reduciendo cada vez que recibe una herida. Cuando está muerto ya no tienen efecto las heridas. Cuando un alien logra llegar a la base recupera 2 vidas, sin superar nunca el valor máximo de vidas,

esto es, 5. El comando establecerVidas requiere un parámetro menor o igual al máximo de vidas.

Cada alien está asociado a una nave, cada nave tiene una velocidad y una cantidad de combustible en el tanque. Ambos atributos pueden aumentar o disminuir de acuerdo a un parámetro que puede ser positivo o negativo.

La fuerza de un alien se calcula como la capacidad sensora, más su capacidad de lucha, todo multiplicado por el número de vidas.



Las clases `Alien` y `NaveEspacial` están asociadas, la clase `Alien` **tiene un** atributo de clase `NaveEspacial`. En el **problema**, dos o más aliens pueden estar asociados a una misma nave espacial. En la **solución**, dos o más objetos de clase `Alien` pueden mantener referencias a un mismo objeto de clase `Nave`. También es posible que dos naves tengan el mismo estado interno, pero distinta identidad.

La implementación de `NaveEspacial` es:

```
class NaveEspacial {
//Atributos de instancia
private int velocidad;
private int combustible;
//Constructor
public
  NaveEspacial(int v,int c){
  velocidad= v;
  combustible= c;}
//Comandos
```

```
public void cambiarVelocidad(int v){
  velocidad =+v;}
public void cambiarCombustible(int c){
  combustible =+c;}
public void copy (NaveEspacial n){
velocidad=n.obtenerVelocidad();
combustible=n.obtenerCombustible();}
//Consultas
public int obtenerVelocidad(){
  return velocidad;}
public int obtenerCombustible (){
  return combustible;}
public boolean equals (NaveEspacial n){
  return velocidad ==n.obtenerVelocidad()&&
       combustible==n.obtenerCombustible();}
public NaveEspacial clone(){
  return new NaveEspacial(velocidad, combustible);}}
```

La implementación de `Alien` de acuerdo al diseño propuesto:

```
class Alien {
//Atributos de clase
private static final int maxVidas=5;
//Atributos de instancia
private NaveEspacial Nave;
private int vidas;
private int antenas;
private int manos;
//Constructor
public Alien (NaveEspacial n,int a,int m){
  Nave = n;
  vidas = maxVidas;
  antenas = a;
  manos = m;}
//Comandos
public void establecerVidas(int v){
//Requiere v < maxVidas
  vidas = v;}
public void recuperaVidas(){
  if (vidas+2 > maxVidas)
    vidas = maxVidas;
  else
    vidas = vidas+2;}
public void recibeHerida(){
  if (vidas > 0) vidas--;}
public void copy (Alien a){
  Nave = a.obtenerNave();
  vidas = a.obtenerVidas();
  antenas = a.obtenerAntenas();
  manos = a.obtenerManos();}
public void establecerNave(NaveEspacial n){
Nave = n;}
//Consultas
public NaveEspacial obtenerNave(){
  return Nave;}
public int obtenerVidas(){
  return vidas;}
```

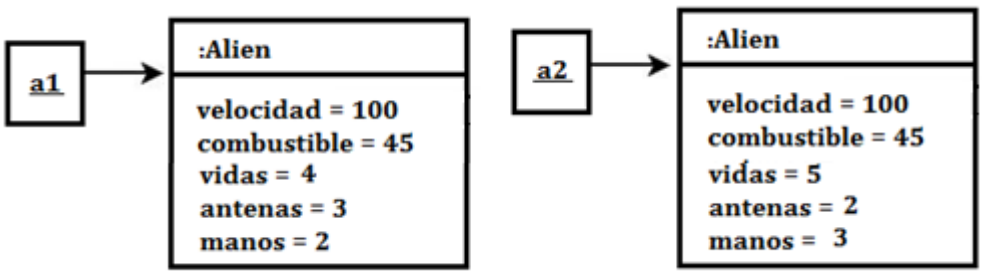
```
public int obtenerAntenas(){
  return antenas;}
public int obtenerManos (){
  return manos;}
public int obtenerFuerza(){
  return (antenas+manos)*vidas;}
public Alien clone(){
  NaveEspacial n = Nave;
  int m = manos;
  int a = antenas;
  int v = vidas;
  Alien c = new Alien(n,a,m);
  c.establecerVidas(v);
  return c;}
public boolean equals(Alien a){
  return Nave == a.obtenerNave() &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}}
```

El constructor y el comando `establecerNave` de la clase `Alien` reciben como parámetro una variable de la clase asociada `NaveEspacial`. La consulta `obtenerNave` retorna una referencia a un objeto de la clase `NaveEspacial`. Se crea entonces una dependencia entre `Alien` y `NaveEspacial`, consecuencia de la asociación entre clases. Si la signatura de los métodos de la clase `NaveEspacial` cambia, la modificación puede afectar a la clase `Alien`.

A continuación *se asume* una **representación extendida** para objetos asociados y se analizan sus inconvenientes. Al ejecutarse el siguiente bloque de instrucciones del método `main`:

```
class testVideoJuego{
public static void main (String s[]){
  NaveEspacial n = new NaveEspacial(100,45);
  Alien a1,a2;
  a1 = new Alien(n,3,2);
  a2 = new Alien(n,2,3);
  a1.recibeHerida();
```

El diagrama de objetos para la representación extendida sería:



Es decir, el estado interno de un objeto de clase `Alien` mantiene los atributos de esa clase, más los atributos de la clase asociada `NaveEspacial`. Si dos o más objetos de la clase `Alien` están asociados a una misma instancia de la clase `NaveEspacial`, la estructura completa de cada objeto de la clase `NaveEspacial` se mantendrá en cada objeto de clase `Alien`.

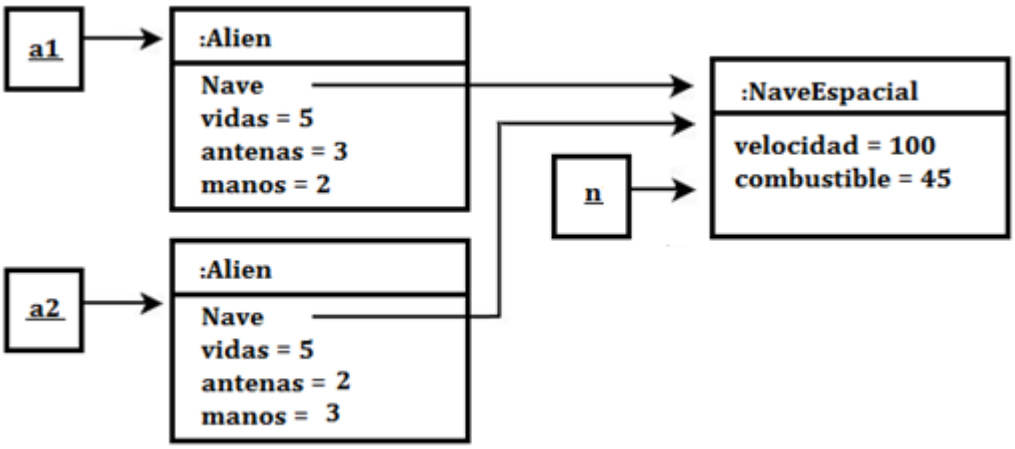
La representación extendida provoca varios inconvenientes. En primer lugar, es necesario agregar un atributo a cada nave que permita identificarla, por ejemplo un código. En caso contrario, si los estados internos de dos objetos de clase `Alien` coinciden, no es posible decidir si están ligados a la misma nave o a naves diferentes que coinciden en la velocidad y en la cantidad de combustible.

En segundo lugar, cada vez que se modifica el valor de uno de los atributos que caracterizan a un objeto, es necesario modificar el estado interno de todos los objetos a los cuales está asociado. En este caso de estudio, cada vez que cambie la `velocidad` de una nave se debería modificar el estado interno de todos los aliens asociados a esa nave.

Otro inconveniente importante de la representación extendida es que si se modifica la representación de la clase `NaveEspacial`, por ejemplo, porque se agrega un atributo de instancia `motores`, este cambio va a afectar a la representación de todas las clases que usan los servicios de `NaveEspacial`. Los cambios en una clase provocan un impacto en la representación de los objetos de las clases asociadas.

Estos inconvenientes se solucionan usando una **representación por referencia**. El estado interno de un objeto contiene referencias a los objetos de las clases asociadas, de modo que un sistema complejo puede modelarse a partir de objetos simples. La modificación de una clase, no afecta la representación de los objetos de las clases asociadas.

Luego de la ejecución del mismo bloque de instrucciones, el diagrama de objetos considerando una representación por referencia es:



En este caso, si cambia la estructura del estado interno de un objeto de clase `NaveEspacial`, por ejemplo, porque se agregan nuevos atributos, no cambia la estructura del estado interno de los objetos de clase `Alien`. Si se modifican los valores de los atributos de una nave, por ejemplo, porque se reduce el combustible, no se modifican los valores de los atributos de los aliens que referencian a esa nave.

Igualdad y equivalencia entre objetos de clases asociadas

La representación por referencia afecta al diseño y la implementación de algunos servicios, en particular a los métodos `equals`, `copy` y `clone`. Cuando una clase está asociada a otra, la igualdad, copia y clonación se puede hacer en forma **superficial** o en **profundidad**.

Para decidir si dos objetos de una clase son iguales en forma superficial, se evalúa si están ligados a una misma instancia de la clase asociada. De manera análoga, la copia superficial modifica el estado interno del objeto que recibe el mensaje, con los valores almacenados en el objeto que pasa como parámetro. La clonación superficial retorna como resultado un objeto que tiene exactamente el mismo estado interno que el objeto que recibió el mensaje, incluyendo las referencias a objetos de las clases asociadas.

Cuando la igualdad es en profundidad la exigencia es menor, se requiere que los estados internos de los objetos asociados sean equivalentes, aunque las referencias sean distintas. En el caso de la copia en profundidad, el estado interno del objeto que recibe el mensaje se modifica con los valores de los atributos del objeto que se recibe como parámetro, excepto las referencias, que no se modifican, pero sí se modifica el estado interno de los objetos asociados. La clonación en profundidad retorna un nuevo objeto con el mismo estado interno que el objeto que recibe el mensaje, excepto las referencias que estarán ligadas a clones de los objetos asociados.

Para ilustrar la diferencia, se retoma el análisis del caso de estudio que asocia a las clases `Alien` y `NaveEspacial`. Los siguientes métodos implementan entonces **copia, clonación e igualdad superficial**:

```
public void copy (Alien a){
//Requiere a ligada
  Nave = a.obtenerNave();
  vidas = a.obtenerVidas();
  antenas = a.obtenerAntenas();
  manos = a.obtenerManos();}
public Alien clone(){
  NaveEspacial n = Nave;
  int m = manos;
  int a = antenas;
  int v = vidas;
  Alien c = new Alien(n,a,m);
  c.establecerVidas(v);
  return c;}
public boolean equals(Alien a){
//Requiere a ligada
  return Nave == a.obtenerNave() &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}
```

El comando `copy` copia el estado interno del objeto de clase `Alien` que se recibe como parámetro, en el estado interno del objeto de clase `Alien` que recibe el mensaje. Así, el valor del atributo `Nave` del objeto ligado al parámetro `a`, se asigna al atributo `Nave` del objeto que recibe el mensaje `copy`.

La consulta `clone` retorna un nuevo objeto de clase `Alien` que referencia a la misma nave que el objeto que recibió el mensaje. La siguiente implementación de `clone()` es equivalente a la anterior:

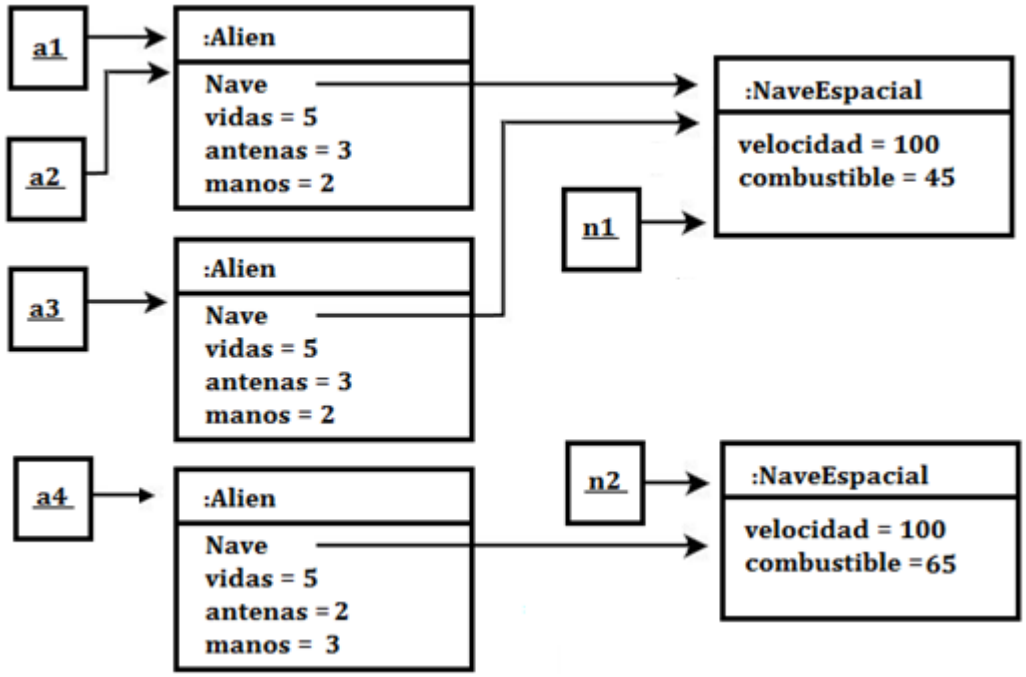
```
public Alien clone(){
  Alien c = new Alien(Nave,antenas,manos);
  c.establecerVidas(v);
  return c;}
```

La consulta `equals` retorna `true` si los valores de los atributos de los objetos que se comparan son iguales, esto implica que mantienen referencias a una misma nave.

Dada la siguiente clase tester:

```
class testVideoJuego{
public static void main (String s[]){
//Primer bloque
  NaveEspacial n1 = new NaveEspacial (100,45);
  NaveEspacial n2 = new NaveEspacial (100,65);
  Alien a1,a2,a3,a4;
  a1 = new Alien(n1,3,2);
  a2 = a1;
  a3 = a1.clone();
  a4 = new Alien(n2,2,3);
//Segundo bloque
  a4.copy(a1);
  System.out.println(a1==a2);
  System.out.println(a1==a3);
  System.out.println(a1==a4);
  System.out.println(a1.equals((a2)));
  System.out.println(a1.equals((a3)));
  System.out.println(a1.equals((a4)));
//Tercer bloque
  a1.recibeHerida();
  System.out.println(a1==a2);
  System.out.println(a1==a3);
  System.out.println(a1==a4);
  System.out.println(a1.equals((a2)));
  System.out.println(a1.equals((a3)));
  System.out.println(a1.equals((a4)));
//Cuarto bloque
  a1.recibeHerida();
  a4.copy(a1);
  System.out.println(a1.equals((a2)));
  System.out.println(a1.equals((a3)));
  System.out.println(a1.equals((a4)));
//Quinto bloque
NaveEspacial n3 = n1.clone();
Alien a5=a1.clone();}
}
```

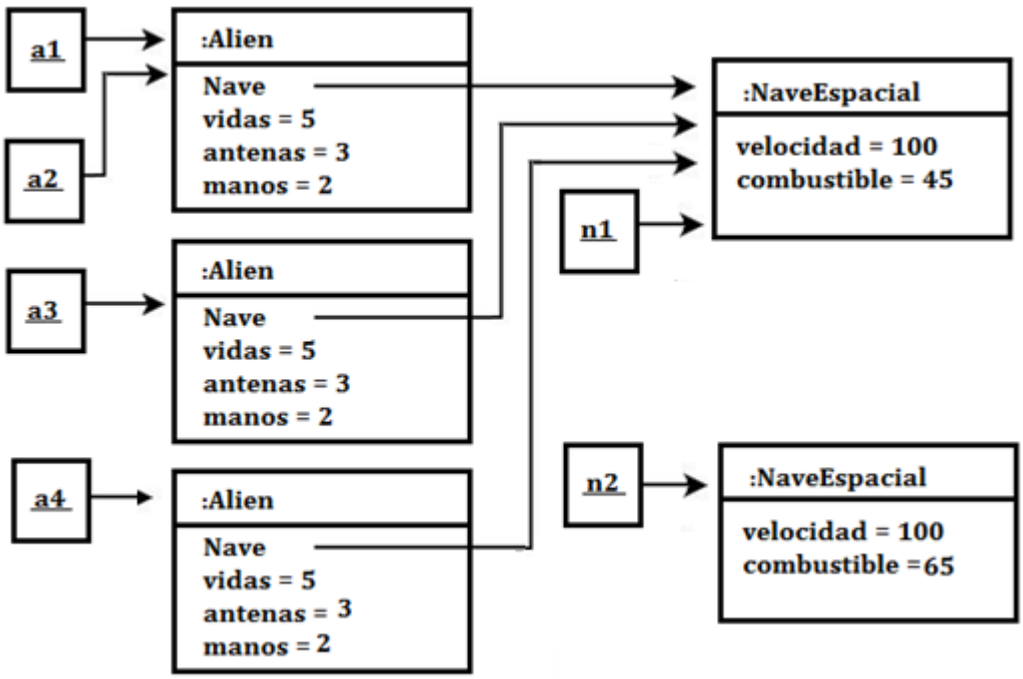
Después de la ejecución de las instrucciones que siguen al comentario “Primer bloque”, el diagrama de objetos es:



Cuando a continuación se ejecuta:

```
a4.copy(a1);
```

El diagrama de objetos se modifica como sigue:



De modo que el operador relacional en las instrucciones:

```
System.out.println(a1==a2);
System.out.println(a1==a3);
System.out.println(a1==a4);
```

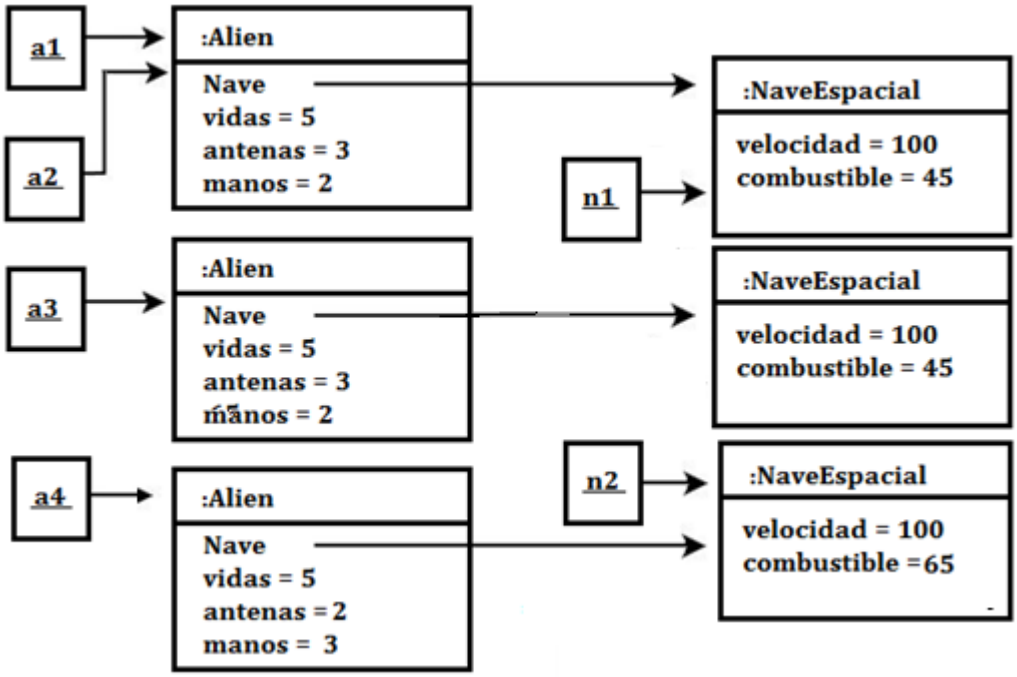
Computa `true`, `false` y `false`.

Ejercicio: Muestre la salida por consola del segmento completo de código y dibuje el diagrama de objetos inmediatamente antes de terminar la ejecución.

El diseñador del videojuego puede especificar que los métodos `copy`, `clone` y `equals` se implementen **en profundidad**. El código de **clone en profundidad** es:

```
public Alien clone(){
//Requiere Nave ligada
  NaveEspacial n = Nave.clone();
  int m = manos;
  int a = antenas;
  int v = vidas;
  Alien c = new Alien(n,a,m);
  c.establecerVidas(v);
  return c;}
```

En esta implementación, se crea un clone del alien que recibe el mensaje, vinculado a un clone de la nave asociada al alien que recibe el mensaje. Dada la misma clase tester, después de la ejecución del primer bloque, el diagrama de objetos es:



El objeto ligado a la variable `a3` mantiene una referencia a una nave que es un clone de la nave asociada al alien ligado a la variable `a1`.

La consulta `clone` definida en la clase `Alien` envía el mensaje `clone` al objeto ligado al atributo de instancia `Nave`. Las clases `Alien` y `NaveEspacial` brindan métodos con el mismo nombre. **Es la clase del objeto que recibe el mensaje la que determina cuál de los métodos debe ejecutarse** en cada caso**.**

Dada la secuencia de instrucciones del último bloque del código:

```
NaveEspacial n3 = n1.clone();
Alien a5 = a1.clone();
```

El primer mensaje `clone` se liga al método clone definido en la clase `NaveEspacial`. El segundo mensaje `clone` se liga al método clone definido en la clase `Alien`.

La implementación del comando **copy en profundidad** es:

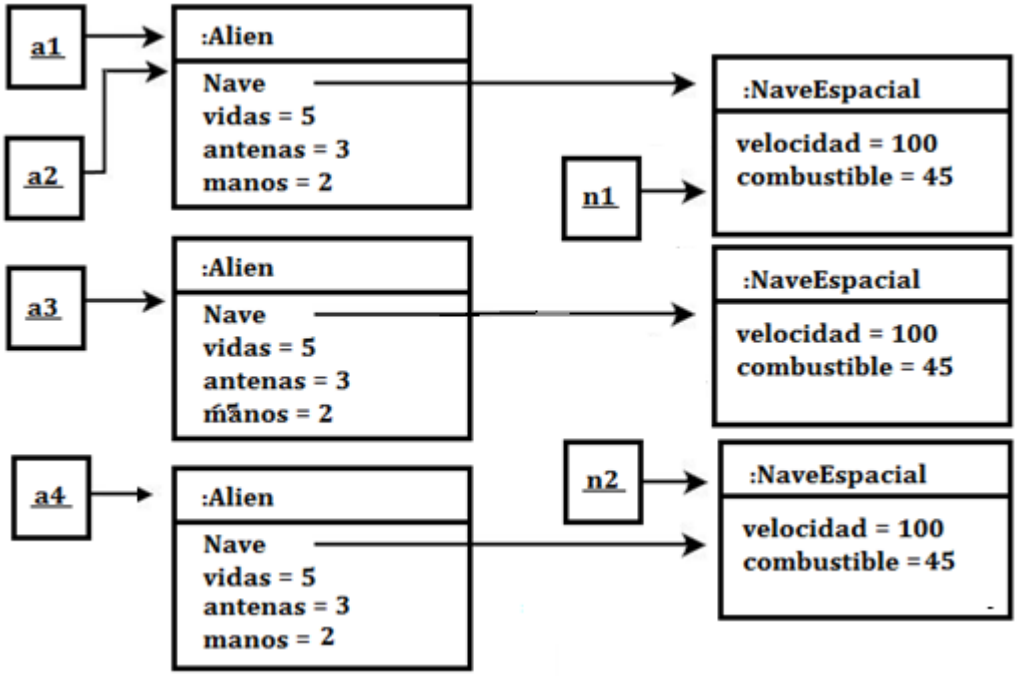
```
public void copy (Alien a){
//Requiere a ligada y la nave de a ligada
  Nave.copy(a.obtenerNave());
```

```
  vidas = a.obtenerVidas();
  antenas = a.obtenerAntenas();
  manos = a.obtenerManos();}
```

Después de la ejecución de la primera instrucción del segundo bloque:

```
  a4.copy(a1);
```

Se copia en el estado interno de la nave asociada al alien que recibe el mensaje, el estado interno de la nave del alien `a`, recibido como parámetro forma. El diagrama de objetos es:



Es decir, la copia en profundidad modifica los valores de los atributos `vidas`, `antenas` y `manos`, y el estado interno de la nave asociada al alien que recibe el mensaje, pero no se modifica el valor del atributo instancia `Nave` de la clase `Alien`.

La implementación de **equals en profundidad** es:

```
public boolean equals(Alien a){
  return Nave.equals(a.obtenerNave()) &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}
```

La consulta `equals` definida en la clase `Alien` envía el mensaje `equals` al objeto ligado al atributo de instancia `Nave`. Nuevamente, es la clase del objeto que recibe el mensaje la que determina cuál de los métodos debe ejecutarse en cada caso.

Para esta implementación de la consulta `equals` y considerando el último diagrama de objetos, las instrucciones:

```
  System.out.println(a1.equals((a3));
  System.out.println(a1.equals((a4));
```

Muestran en consola:

```
true
true
```

Es posible que el diseñador especifique en el diagrama que una clase debe brindar, por ejemplo, el método `equals` implementado en forma superficial y también en profundidad. En este caso es necesario elegir un nombre diferente para cada consulta.

```
public boolean equalsS(Alien a){
  return Nave == a.obtenerNave() &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}
public boolean equalsP(Alien a){
  return Nave.equals(a.obtenerNave()) &&
         manos == a.obtenerManos() &&
         antenas == a.obtenerAntenas() &&
         vidas == a.obtenerVidas();}
```

En este caso la clase `Alien` brinda dos métodos alternativos para decidir si dos objetos son equivalentes. El primero, más estricto, decide que dos aliens son equivalentes si están asociados a una misma nave. El segundo, considera que dos aliens son equivalentes si están asociados a naves equivalentes.

Clientes y proveedores de servicios

En el conjunto de clases que modela una aplicación, algunas clases son **clientes** de los servicios provistos por otras clases **proveedoras**. Con frecuencia una misma clase puede cumplir ambos roles, es decir, ser cliente y proveedora a la vez.

Una clase que **usa** los servicios provistos por otra clase, es cliente de la clase que provee dichos servicios. Una clase que **tiene** atributos de otra clase, es cliente de las clases a las que corresponden a esos atributos; estas últimas son proveedoras de servicios.

En particular una clase tester es una clase cliente de los servicios que brindan las clases que deben ser verificadas. Al implementar una clase tester se establece una dependencia entre esta clase y las que van a ser verificadas.

La clase cliente accede a la clase proveedora a través de su **interfaz**. La programación orientada a objetos propone **minimizar** la interfaz a través de la cual se comunican una clase cliente y una clase proveedora, de modo que se minimice también el impacto de los cambios.

Caso de Estudio: Plan Teléfono Móvil

Una empresa de telefonía celular ofrece distintos planes a sus abonados. Un plan tiene un código, un costo mensual y establece un tope para el número de mensajes de texto y un tope de créditos que los abonados consumen con sus llamadas a números dentro de la comunidad y a otros móviles fuera de la comunidad. Una línea tiene un número asociado, un plan y una cantidad de consumos a líneas dentro de la comunidad y a líneas móviles fuera de la comunidad. La cantidad de créditos de un plan se consume de modo diferente según la llamada se realice dentro de la comunidad o fuera de ella. Un minuto (o fracción) de llamada dentro de la comunidad consume 1 crédito, un minuto a una línea móvil fuera de la comunidad consume 2 créditos.

Las clases Linea y Plan están asociadas tal como modela el siguiente diagrama.

Linea
<<Atributos de instancia>>

Plan
<<Atributos de instancia>>

Linea	Plan
nro: String plan : Plan consumosSms, consumosAComunidad, consumosAMoviles : entero	codigo:entero sms,credito:entero costo:entero
<<Constructor>> Linea(nro:String) <<Comandos>> establecerPlan(p:Plan) aumentarSms(c:entero) aumentarACom(c:entero) aumentarAMov(c:entero) <<Consultas>> obtenerNro():String obtenerPlan():Plan obtenerConsumosSms():entero obtenerConsumosAComunidad(): entero obtenerConsumosAMoviles(): entero consumoCredito():entero smsDisponibles():entero creditoDisponible():entero	<<Constructor>> Plan(c:entero) <<Comandos>> establecerSms(n:entero) establecerCredito(n:entero) establecerCosto(n:entero) <<Consultas>> obtenerSms():entero obtenerCredito():entero obtenerCosto():entero
Requiere que se establezca el plan antes de aumentar los consumos o ejecutar las consultas	

`consumoCredito()`: se computa como `consumosAComunidad+ consumosAMoviles*2.`
`smsDisponibles()`: se computa como el crédito que corresponde al plan, menos el consumo de crédito de la línea .

La implementación de `Linea` es:

```
class Linea {
//Atributos de Instancia
private String nro;
private Plan plan ;
private int consumosSMS;
private int consumosAComunidad;
private int consumosAMoviles;
//Constructor
public Linea (String n){
 nro = n;}
//Comandos
public void establecerPlan (Plan p){
  plan = p;}
public void aumentarSMS (int n){
 consumosSMS=+ n;}
public void aumentarAComunidad (int n){
 consumosSMS=+ n;}
public void aumentarAMoviles (int n){
 consumosSMS=+ n;}
//Consultas
```

```
public String obtenerNro (){
 return nro ;}
public Plan obtenerPlan(){
  return plan;}
public int obtenerConsumosSMS (){
 return consumosSMS ;}
public int obtenerConsumosAComunidad (){
 return consumosAComunidad ;}
public int obtenerConsumosAMoviles (){
 return consumosAMoviles ;}
public int consumoCredito(){
  return consumosAComunidad+consumosAMoviles*2;}
public int smsDisponibles(){
//Requiere el plan ligado
  return plan.obtenerSMS()-consumosSMS;}
public int creditoDisponible(){
//Requiere el plan ligado
  return plan.obtenerCredito()-consumoCredito();}
}
```

Cada objeto de clase `Linea` está asociado a una instancia de clase `Plan`. La clase `Linea` **tieneUn** atributo de clase **`Plan.`** La clase `Linea` también está asociada a la clase `String`.

Los métodos `establecerPlan` y `obtenerPlan` de la clase `Linea`, generan también una relación de dependencia entre `Plan` y `Linea`. La clase `Linea` es **cliente** de la clase **proveedora** `Plan`. La interfaz entre las clases `Linea` y `Plan` está formada por la signatura de los servicios públicos.

La clase cliente `Linea` puede implementarse conociendo únicamente la interfaz de la clase proveedora `Plan`. La clase `Plan` se implementa desconociendo quienes van a ser sus clientes.

El atributo `plan` en la clase `Linea` mantiene una **referencia** a un objeto de clase `Plan`. En la realidad a modelar, probablemente varias líneas correspondan a un mismo plan, en ejecución varias instancias de `Linea` referenciarán entonces a un mismo objeto de clase `Plan`. La clase `Linea` asume que cuando un objeto reciba el mensaje `creditoDisponible()` su atributo de instancia `plan` estará ligado.
Ejercicio: Implemente la clase `Plan`.

Ejercicio: Implemente una clase `testTelefónica` *que* ***use*** *a las clases* `Linea`, `Plan` *y* `String`*, creando objetos de esa clase y enviándoles mensajes para verificar los servicios provistos por* `Linea`.

El contrato entre la clase Proveedora y la clase Cliente

Entre una clase cliente y una clase proveedora de servicios, se establece un **contrato** que determina las **responsabilidades** de cada una. Las condiciones del contrato se especifican en la etapa de **diseño** del sistema, en particular la **funcionalidad** que debe brindar cada servicio. En la **implementación** es necesario interpretar las responsabilidades y reflejarlas en el código. Parte de la **verificación** consiste en analizar si cada clase cumple con las responsabilidades establecidas en el contrato.

Cuando una clase no cumple con su contrato pueden producirse errores de ejecución o aplicación que es necesario detectar y depurar. También puede ocurrir que el contrato no se

haya especificado adecuadamente. En cualquier caso es fundamental diseñar una batería de casos de prueba que permitan detectar errores. El diseño de los casos de prueba puede hacerlo el diseñador del sistema o el responsable de la etapa de testeado.

Caso de Estudio: Facturas en Cuenta Corriente

*En un negocio se desea mantener información referida a las **facturas** imputadas a las cuentas corrientes de los **clientes**. De cada **factura** se mantiene el **número**, el **monto** y el **cliente**. De cada **cuenta corriente** se mantiene el **nombre** del cliente y el **saldo**. Cuando se crea un objeto de clase `Factura` el constructor inicializa todos los atributos de instancia de acuerdo a los parámetros y actualiza el saldo de la cuenta corriente del cliente con el monto de la factura.*

Factura
<<atributos de instancia>> nroFact : String montoFact : real clienteFact : Cta_Cte
<< Constructores>> Factura (nro:String,m:float,cli: Cta_Cte) <<Comandos>> establecerClienteFact(cli:Cta_Cte) establecerMontoFact(m:real) <<Consultas>> obtenerNroFact():String obtenerClienteFact():Cta_Cte obtenerMontoFact():real
<<Responsabilidades>> Requiere que nro y cli sean referencias ligadas.

Cta_Cte
<<atributos de instancia>> nombre : String saldo : float
<<Constructor>> Cta_Cte (nom:String) <<Comandos>> establecerSaldo(m:real) actualizarSaldo (m:real) <<Consultas>> obtenerNombre():String obtenerSaldo():real
<<Responsabilidades>> Requiere que nom sea una referencia ligada.

Las clases `Factura`, `String` y `Cta_Cte` están asociadas, la relación es de tipo **tieneUn**.

El comando `establecerClienteFact` de la clase `Factura` recibe como parámetro una variable de la `Cta_Cte`. La consulta `obtenerClienteFact` retorna una referencia a un objeto de la clase `Cta_Cte`. La asociación entre las clases `Factura` y `Cta_Cte` provoca también una relación de dependencia.

La clase `Factura` es la clase **cliente** de las clases `Cta_Cte` y `String`. La clase `Cta_Cte` es **proveedora** de servicios respecto a `Factura`, pero es también **cliente** de la clase `String`.

Las clases `Factura` y `Cta_Cte` son **proveedoras** de los servicios que van a ser usados por las clases que declaren variables de los tipos `Factura` y `Cta_Cte`. Estas clases deben asumir la responsabilidad de asegurar que una factura se crea ligada a un número y a una cuenta corriente. Análogamente se requiere que las clases cliente que envíen el mensaje `establecerClienteFact` a un objeto de clase `Factura`, lo hagan con una variable ligada como parámetro.

La clase `Factura` no controla que `nroFact` y `clienteFact` mantengan referencias ligadas, porque el contrato establece que esta no es su responsabilidad. En un sistema **robusto** cada clase puede establecer controles que prevengan errores provocados por otras clases, cuando estas no cumplen con sus responsabilidades. En este caso se utilizan los mecanismos provistos por el lenguaje para **manejar excepciones**.

En este diseño la clase `Factura` asume la responsabilidad de modificar el saldo de la cuenta corriente cuando se crea un objeto.

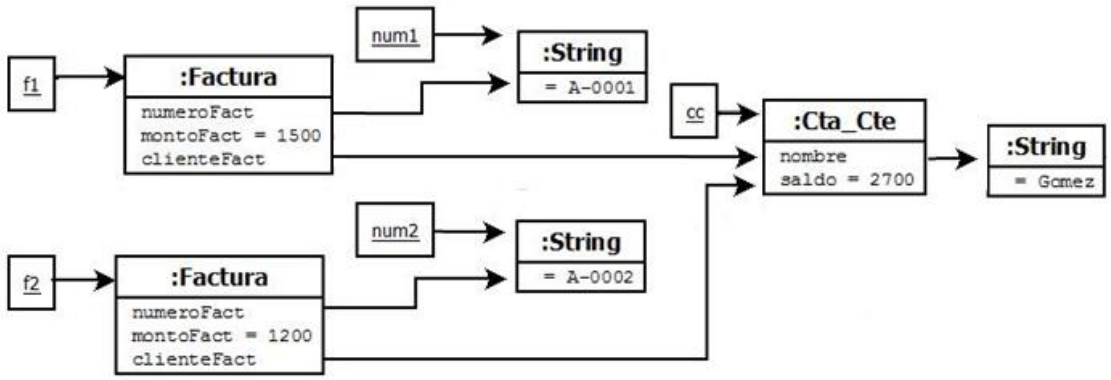
```
class Factura {
private String nroFact;
private float montoFact;
private Cta_Cte clienteFact;
public  Factura  (String n, float m,Cta_Cte cli){
/*Crea una factura, guarda número, monto y cliente, y actualiza el
saldo de la cuenta corriente del cliente con el mismo monto. Requiere
que n y cli estén ligadas */
 nroFact = n;
 montoFact = m;
 clienteFact = cli;
 clienteFact.actualizarSaldo (m);}
```

El constructor envía el mensaje `actualizarSaldo` al objeto ligado a `clienteFact`, asumiendo que es una referencia ligada, porque así lo establece el requerimiento en el diagrama.

Después de la tercera asignación, las variables `cli` y `clienteFact` están ligadas a un mismo objeto. Observemos que aunque el valor de la variable `cli` no cambia, sí se modifica el estado interno de la variable referenciada por `cli`. Decimos que la ejecución del constructor tiene un **efecto colateral**, no solo inicializa el estado interno del objeto que se crea sino que modifica también el estado interno de uno de los objetos que se recibe como parámetro.

```
class Ventas {
…
public static void test(){
String num1 = new String("A-0001");
String num2 = new String("A-0002");
Cta_Cte cc = new Cta_Cte ("Gomez");
…
Factura f1 = new Factura (num1,1500,cc);
Factura f2 = new Factura (num2,1200,cc);}
…}
```

La clase `Ventas` es cliente de la clase `Factura` y es responsable de asegurar que el tercer parámetro del constructor es una variable ligada, pero no de actualizar `saldo`. El diagrama de objetos después de la ejecución del segmento anterior es:



Consideremos el siguiente diseño alternativo:
*En un negocio se desea mantener información referida a las **facturas** imputadas a las cuentas corrientes de los **clientes**. De cada **factura** se mantiene el **número**, el **monto** y el **cliente**. De*

*cada **cuenta corriente** se mantiene el **nombre** del cliente y el **saldo**. Cuando se crea una factura, inmediatamente después se actualiza el saldo de la cuenta corriente.*

En esta alternativa de diseño cada vez que se crea un objeto de clase `Factura` la responsabilidad de actualizar el saldo es de la clase cliente. El código para la segunda alternativa de diseño de `Factura` es:

```
class Factura {
private String nroFact;
private float montoFact;
private Cta_Cte clienteFact;
public  Factura  (String n, float m,Cliente cli){
/*Crea una factura, guarda número, monto  y cliente. Requiere que n y
cli ligadas */
 nroFact = n;
 montoFact = m;
 clienteFact = cli;}
…
}
```

Nuevamente la clase `Ventas` **usa** los servicios provistos por `Factura`, pero además asume la responsabilidad de actualizar el saldo de la cuenta corriente del cliente.

```
class Ventas {
…
public static void test(){
  String num1 = new String("A-0001");
  String num2 = new String("A-0002");
  Cta_Cte cc = new Cta_Cte ("Gomez");
  Factura f1 = new Factura (num1,1500,cc);
  cc.actualizarSaldo (1500);
  Factura f2 = new Factura (num2,1200,cc);
  cc.actualizarSaldo (1200);}
…}
```

El diagrama de objetos al completarse la ejecución del método test, es el mismo cualquiera sea la alternativa elegida.

El diseñador del sistema establece la **responsabilidad** de cada clase. El implementador debe generar código adecuado para garantizar que cada clase cumple con sus responsabilidades. El responsable del testeado busca errores de aplicación en base al contrato.

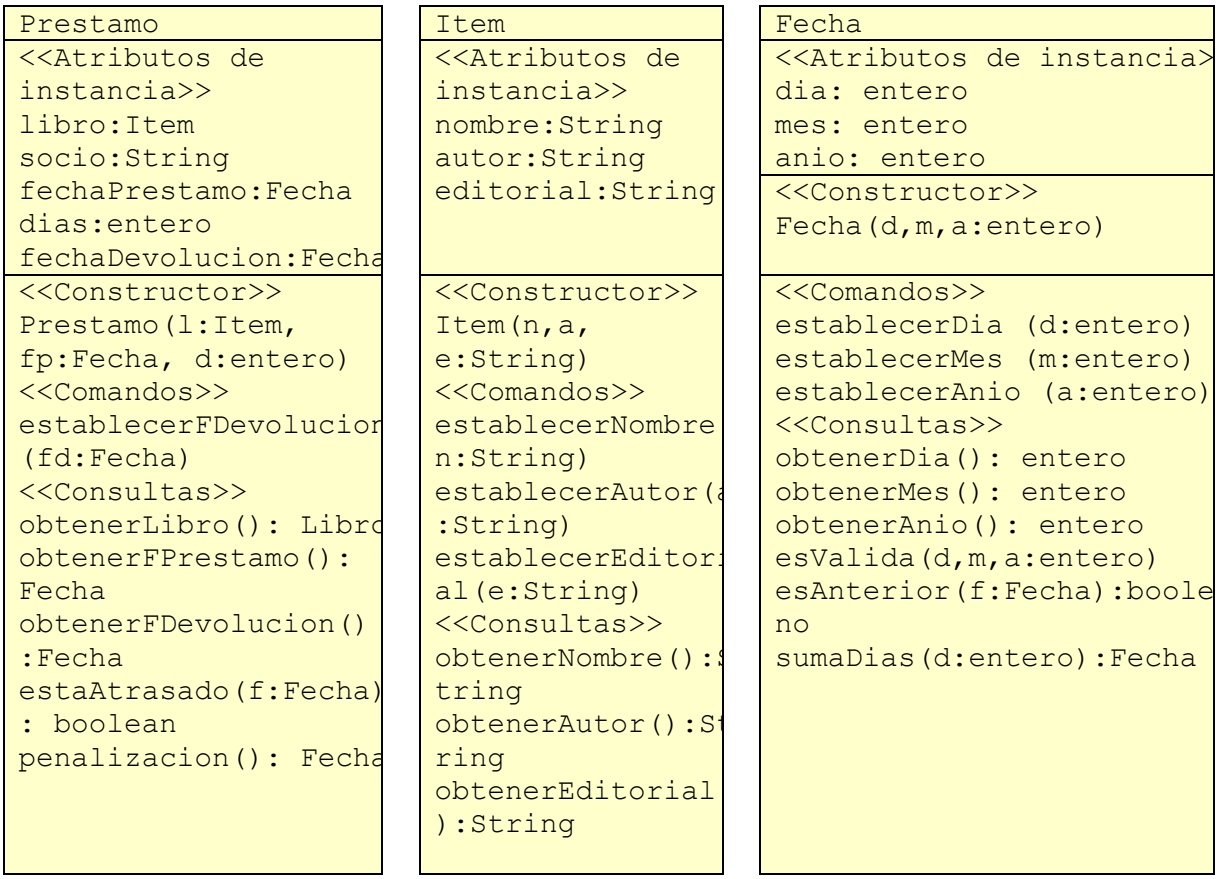
Cualquiera sea la decisión de diseño el código parcial de la clase `Cta_Cte` es:

```
class Cta_Cte {
//Atributos de Instancia
private String nombre;
private float saldo;
//Constructores
public Cta_Cte (String n){
  nombre =n;}
//Comandos
public void actualizarSaldo (float s) {
 saldo += s;}
…
}
```

Observemos que si el diseñador elige una de las alternativas y cambia de decisión una vez que las clases están implementadas, el cambio va a requerir modificar tanto la clase proveedora, como todas las clases que la usan. Una modificación de diseño que cambia las responsabilidades, afecta a la colección de clases asociadas. Si solo se modifica una de las clases, por ejemplo la clase cliente, va a producirse un **error de aplicación**, que pasa desapercibido para el compilador.

Problemas Propuestos

1. En una Biblioteca se registran los datos que permiten hacer un seguimiento de los préstamos de distintos tipos de ítems bibliográficos, en particular libros:



La clase `Item` *brinda los comandos y consultas triviales. El constructor requiere los tres parámetros ligados.*

En la clase `Fecha`*:*

El constructor requiere que `d`*,* `m` *y* `a` *representen una fecha válida.*

`esAnterior(f:Fecha)` *retorna verdadero si la fecha que recibe el mensaje es menor que el parámetro* `f`*.*

`sumaDias(d:entero)` *retorna la fecha que resulta de sumar* `d` *días a la fecha que recibe el mensaje.*

En la clase `Prestamo`*:*

El constructor requiere l y fp ligados y d mayor que 0.

`estaAtrasado(f:Fecha)` *recibe como parámetro la fecha actual y retorna verdadero si el libro no se devolvió y ya debería haberse devuelto de acuerdo a la fecha de préstamo y la cantidad de días.*

`penalizacion` *calcula la cantidad de días de penalización y devuelve la fecha hasta la que corresponde aplicar la penalización, a partir de la fecha de devolución, que tiene que estar ligada. La penalización es de 3 días si se atrasó menos de una semana, 5 días si se atrasó menos de 30 días y 10 días si se atrasó 30 días o más. Si el libro tiene categoría A los días de penalización se duplican. Requiere que la fecha de devolución esté ligada.*

a. *Implemente el diagrama completo*
b. *Implemente una clase tester para cada especificación.*

2. En un campeonato de hockey por cada partido ganado se obtienen 3 puntos y por cada empate se logra 1. Cada equipo tiene un nombre, un capitán, una cantidad de partidos ganados, otra de empatados y otra de perdidos, una cantidad de goles a favor y otra de goles en contra. El capitán es un jugador que tiene un nombre, un número de camiseta, un número que representa la posición en la que juega, la cantidad de partidos jugados y la cantidad de goles convertidos en el campeonato.

Para un jugador se desea calcular el promedio de goles por partido y dado otro jugador, cuál es el que hizo más goles. Para un equipo se desea calcular los partidos jugados y los puntos obtenidos. Además para otro equipo dado es necesario decidir cuál es el equipo con mayor puntaje y cuál es el capitán con más goles. Si dos equipos tienen en los mismos puntos, se devuelve el que tiene mayor cantidad de goles a favor y si también hay coincidencia se consideran los goles en contra. Si hay coincidencia se devuelve uno cualquiera.

El siguiente diagrama modela las clases Jugador y Equipo:

Jugador
<<Atributos de instancia>> nombre: String nroCamiseta: entero posicion: entero golesConvertidos: entero partidosJugados: entero
<<Constructor>> Jugador(nom:String) <<Comandos>> aumentarGoles(n:entero) aumentarUnPartido() <<Consultas>> promedioGolesXPart(): entero jugConMasGoles(j: Jugador): Jugador masGoles(j:Jugador): boolean toString(): String

Equipo
<<Atributos de instancia>> nombre: String capitan: Jugador pG,pE,pP: entero gFavor, gContra: entero
<<Constructor>> Equipo(nom:String, cap: Jugador) <<Comandos>> incrementarPG(jugoElCap: boolean) incrementarPE(jugoElCap: boolean) incrementarPP(jugoElCap:boolean) incrementarGfavor (total, delCap: entero) incrementarGContra(total: entero) <<Consultas>> partidos(): entero puntos(): entero mejorPuntaje(e: Equipos): Equipo

	capitanConMasGoles(e: Equipo): Jugador

a. Implemente las clases modeladas en el diagrama agregando los métodos triviales para obtener y establecer atributos y considerando la siguiente especificación:

masGoles() devuelve true si el jugador que recibe el mensaje tiene más goles que el jugador que corresponde al parámetro.

incrementarPG(jugoElCap: boolean) , incrementarPE(jugoElCap: boolean),

incrementarPP(jugoElCap: boolean) *Aumentan en 1 los partidos del equipo y si corresponde envía un mensaje al capitán para que incremente en 1 sus partidos.*

incrementarGFavor(total, delCap: entero): *Aumenta los goles convertidos por el equipo y si corresponde envía un mensaje al capitán para actualizar sus goles.*

incrementarGContra(total: entero): *Aumenta los goles en contra del equipo*

b. Considerando el mismo diseño para la clase Jugador implemente la case Equipo de acuerdo a la siguiente especificación:

incrementarPG() incrementarPE() incrementarPP(): *Aumenta en 1 los partidos del equipo. Requiere que la clase cliente aumente, si corresponde la cantidad de partidos jugados por el capitán.*

incrementarGFavor(total: entero): *Aumenta los goles del equipo. Requiere que la clase cliente aumente, si corresponde la cantidad de goles convertidos por el capitán.*

incrementarGContra(total: entero): *Aumenta los goles en contra del equipo.*

c. Implemente una clase tester para cada especificación.

Dada la clase Color implementada en prácticos anteriores y el siguiente diagrama para la clase Borde, asociada a la clase Color:

Borde
grosor : entero color : Color
<<constructor>> Borde(g:entero,c:Color) <<comandos>> copy(b:Borde) <<consultas>> obtenerGrosor():entero obtenerColor():Color clone () : Borde equals(b:Borde):boolean

Copia, Igualdad y Clonación Superficial

a. Implemente la clase Borde y verifique para un conjunto de casos de prueba fijos.

b. Dado el siguiente segmento de código:

```
Color C1,C2;
Borde B1,B2;

C1 = new Color(110, 110, 110);
```

```
C2 = new Color(110, 110, 110);
B1 = new Borde(1,C1);
B2 = new Borde(1,C2);

System.out.println( C1 == C2);
System.out.println( B1 == B2);
System.out.println( C1.equals(C2));
System.out.println(B1.equals(B2));
```

i. Dibuje el diagrama de objetos al terminar el bloque de asignaciones.

ii. Muestre la salida

c. Dado el siguiente segmento de código:

```
Color C1,C2;
Borde B1,B2,B3,B4;

C1 = new Color(110, 110, 110);
C2 = new Color(110, 110, 110);
B1 = new Borde(1,C1);
B2 = new Borde(1,C2);
B3 = B2.clone();
B4 = new Borde (B2.obtenerGrosor(),B2.obtenerColor());
B1.copy(B2);

System.out.println( B2 == B1);
System.out.println( B2 == B3);
System.out.println( B2 == B4);
System.out.println(B2.equals(B1));
System.out.println(B2.equals(B3));
System.out.println(B2.equals(B4));
System.out.println(B2.obtenerGrosor() == B1.obtenerGrosor() &
                   B2.obtenerColor() == B1.obtenerColor());
System.out.println(B2.obtenerGrosor() == B3.obtenerGrosor() &
                   B2.obtenerColor() == B3.obtenerColor());
System.out.println(B2.obtenerGrosor() == B4.obtenerGrosor() &
                   B2.obtenerColor() == B4.obtenerColor());
System.out.println(B2.obtenerGrosor() == B1.obtenerGrosor() &
                   B2.obtenerColor().equals(B1.obtenerColor()));
System.out.println(B2.obtenerGrosor() == B3.obtenerGrosor() &
                   B2.obtenerColor().equals(B3.obtenerColor()));
System.out.println(B2.obtenerGrosor() == B4.obtenerGrosor() &
                   B2.obtenerColor().equals(B4.obtenerColor()));
```

i. Dibuje el diagrama de objetos al terminar el bloque de asignaciones.

ii. Muestre la salida

d. Dibuje el diagrama de objetos y muestre la salida del ejercicio anterior considerando que la clase Borde implementa:

i. clonación en profundidad, igualdad superficial , copia superficial

ii. clonación superficial, igualdad en profundidad, copia superficial

iii. clonación en profundidad, igualdad en profundidad, copia superficial

iv. clonación en profundidad, igualdad en profundidad, copia en profundidad

Encapsulamiento y Abstracción

Un sistema es una colección de componentes que interactúan entre sí con un objetivo común. En la construcción de sistemas complejos el **encapsulamiento** es un mecanismo fundamental porque permite utilizar cada componente como una "caja negra", esto es, sabiendo **qué** hace sin saber **cómo** lo hace. Se reducen así las dependencias entre las diferentes componentes, cada una de las cuales es más fácil de entender, probar y modificar.

El concepto de encapsulamiento está ligado a toda actividad vinculada a la producción. Una máquina compuesta se construye conectando varias máquinas simples, de modo que la salida de cada una constituye la entrada de otra y en conjunto alcanzan un propósito. Un motor, por ejemplo, se fabrica a partir de dispositivos mecánicos y energéticos. Estos dispositivos pueden usarse para construir otro tipo de máquinas. Un motor a su vez, va a ser usado como una componente en la construcción de otras máquinas.

Cada componente es un **módulo** que puede considerarse individualmente o como parte del conjunto. Cuando un módulo se analiza como parte de un todo, es posible concentrarse en qué función realiza, sin considerar cómo fue construido. Cuando un módulo se analiza individualmente es posible hacer abstracción de los detalles del contexto en el que va a ser utilizado, para enfocarse en cómo lograr su propósito.

En el desarrollo de sistemas de software el concepto de encapsulamiento está fuertemente ligado al de **abstracción**. Analistas y diseñadoras elaboran un modelo a partir de un proceso de abstracción que especifica una colección de módulos relacionados entre sí. Los desarrolladores retienen en el código la estructura modular especificada en el diseño. Cada módulo puede ser verificado independientemente de los demás, antes de integrarse al sistema. Durante el mantenimiento, el encapsulamiento permite reducir el impacto de los cambios.

La construcción de cada módulo de software puede realizarse sin conocer el sistema al cual va a integrarse. El sistema completo se construye a partir de una colección de módulos, sin considerar cómo se implementó cada uno en particular.

Encapsulamiento en la programación orientada a objetos

La **estructura estática** de un sistema orientado a objetos queda establecida por una colección de clases relacionadas entre sí. El **modelo dinámico** es un entorno poblado de objetos comunicándose a través de mensajes. El comportamiento de cada objeto en respuesta a un mensaje, depende de la clase a la que pertenece.

El concepto de encapsulamiento está ligado tanto a las clases como a los objetos. Cada objeto de software interactúa con otros objetos del entorno a través de su **interfaz** y **oculta** su estado interno y los detalles que describen cómo actúa cuando recibe un mensaje. Así, si la estructura del estado interno se modifica, el cambio no afecta a los otros miembros del entorno. Solo las modificaciones en la interfaz pueden afectar a los demás objetos.

Para que un objeto de software oculte su estado interno, la clase que define sus atributos y comportamiento debe **esconder** la implementación. Cada clase es una caja negra, es decir, conoce **qué** hacen las demás clases del sistema, pero no **cómo** lo hacen.

En la programación orientada a objetos el **encapsulamiento**[1] es el mecanismo que permite **agrupar** en una clase los atributos y el comportamiento que caracterizan a sus instancias y **esconder** la representación de los datos y los algoritmos que modelan al comportamiento, de modo que solo sean visibles dentro de la clase.

El encapsulamiento brinda cierto nivel de independencia, cada clase puede ser construida y verificada antes de integrarse al sistema completo. Una clase puede modificar la representación de los datos o los algoritmos, sin afectar a sus clases clientes, en tanto mantenga su funcionalidad y sus responsabilidades.

Estructura de datos

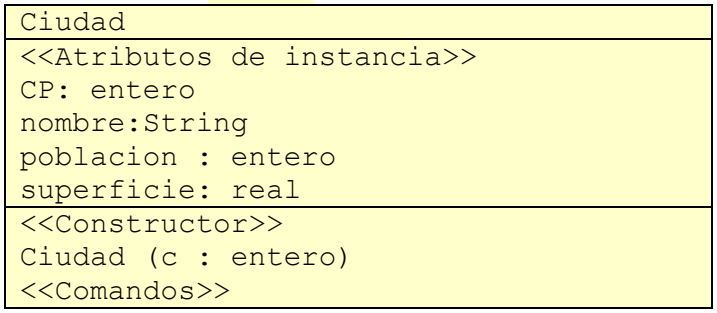
Los atributos de una clase definen la **estructura de datos** que permite representar el estado interno de los objetos de esa clase en ejecución. La programación orientada a objetos propone esconder la representación de los datos, de modo que no sean visibles desde el exterior de la clase.

En Java los **modificadores de acceso** permiten establecer la visibilidad de los miembros de una clase. Los atributos se declaran privados y quedan así escondidos dentro de la clase.

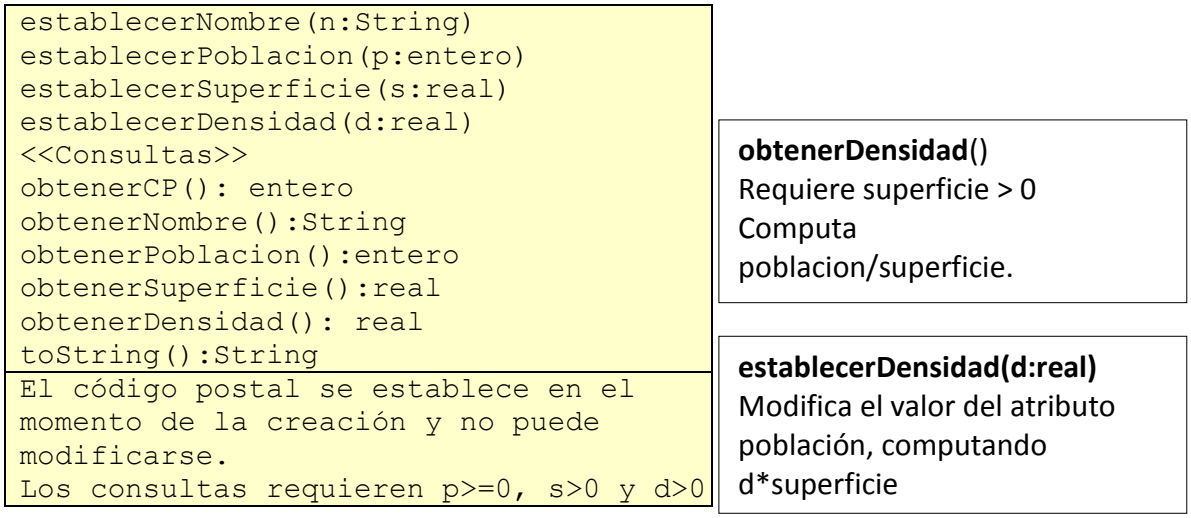
Caso de Estudio: Ciudad

*El Ministerio de Educación de la Provincia ofrece un Programa de Capacitación Docente para maestros de nivel Primario. Consideremos el problema de desarrollar un sistema para la gestión del Programa de Capacitación. El programa ofrece diferentes talleres para maestros. En cada **taller** se proponen **actividades** que luego los **maestros** desarrollan en la **escuela** en la que trabajan. El sistema de gestión requiere administrar la **inscripción** de los maestros en los talleres, el control de **asistencia**, la asignación de **capacitadores**, el registro de las **evaluaciones** de los maestros y las escuelas. Al Ministerio le interesa realizar un análisis comparativo del Programa de Capacitación en cada **ciudad** y en cada **barrio** en particular. El análisis considera la **población** de cada barrio y cada ciudad, la **densidad** de la ciudad, la **cantidad de alumnos** en escuelas públicas y en escuelas privadas, en cada barrio. El sistema de gestión para el Programa de Capacitación docente requiere entonces modelar al conjunto de ciudades, barrios, maestros, talleres, capacitadores, inscripciones, etc.*

El modelo completo va a incluir una clase para cada conjunto de entidades. A partir del modelo completo, es posible implementar cada clase en particular. El siguiente diagrama modela a la clase `Ciudad`:



[1] Algunos autores utilizan los términos *encapsulamiento* y *oscurecimiento de información* para referenciar a dos conceptos relacionados pero diferentes. El primero agrupa atributos y servicios, el segundo esconde la implementación.



El diseñador decidió que es relevante incluir atributos para el código postal, el nombre, la población y la superficie. En la caracterización interesa también conocer la densidad de cada ciudad, pero el diseñador resolvió que la densidad no se mantenga como un atributo, sino que se calcule como el cociente entre la población y la superficie.

De acuerdo a esta propuesta, en el momento que se crea una ciudad se establece únicamente el código, que no puede ser luego modificado. La clase brinda una consulta para obtener cada atributo y una más para obtener la densidad. Para las clases cliente, una ciudad tiene una densidad, más allá de que se mantenga en el estado interno o se compute.

La implementación de la clase `Ciudad` es:

```
public class Ciudad {
//Atributos de instancia
private int CP;
private int poblacion;
private float superficie;
private String nombre;
//Constructor
public  Ciudad (int cod )
{ CP = cod; poblacion = 0 ; superficie = 0 ;}
//Comandos
public void establecerNombre (String n)
{  nombre = n ; }
public void establecerPoblacion (int p )
{  poblacion = p ; }
public void establecerSuperficie ( float s)
{  superficie = s ;}
public void establecerDensidad ( float d)
{  poblacion = (int) (d*superficie) ;}
//Consultas
public String obtenerNombre()
{ return nombre;}
public int obtenerCP ()
{ return CP; }
public int obtenerPoblacion ()
{ return poblacion; }
public float obtenerSuperficie ()
{ return superficie; }
```

```
public double densidad ()
{ return poblacion/superficie; }
}
```

Los atributos se declaran como privados, quedan así escondidos dentro de la clase. Las clases clientes de `Ciudad` no pueden acceder directamente al estado interno de sus instancias.

La clase brinda cuatro comandos que permiten modificar el estado interno de un objeto de clase `Ciudad`. Si la clase cliente usa el servicio `establecerDensidad()` no se modifica el valor de este atributo, porque de hecho no se mantiene explícitamente, sino que se computa un nuevo valor para el atributo `poblacion`. El cómputo requiere conversión de tipos.

La clase `Ciudad` asume que sus clientes cumplen con sus responsabilidades. Si se envía el mensaje `obtenerDensidad()` a un objeto, antes de asignarse un valor a superficie, la ejecución terminará anormalmente. Una clase **confiable**, debería prevenir situaciones de error, más allá de sus propias responsabilidades. El **manejo de excepciones** permite justamente considerar estos casos.

Consideremos que una vez implementado el sistema se resuelve modificar la representación interna de la clase `Ciudad`.

Ciudad
<<Atributos de instancia>> CP: entero nombre:String superficie : real densidad: real
<<Constructor>> Ciudad (c : entero) <<Comandos>> establecerNombre(n:String) establecerPoblacion(p:entero) establecerSuperficie(s:real) establecerDensidad(d:real) <<Consultas>> obtenerCP(): entero obtenerNombre():String obtenerPoblacion():entero obtenerSuperficie():real obtenerDensidad(): real toString():String
El código postal se establece en el momento de la creación y no puede modificarse. Las consultas requieren p>=0, s>0 y d>0

obtenerPoblación()
Computa la parte entera de la expresión densidad*superficie

establecerPoblación(p:real)
Requiere superficie > 0.
Modifica el valor del atributo densidad, computando p/superficie.

La implementación de la clase `Ciudad` es ahora:

```
public class Ciudad {
//Atributos de instancia
private int CP;
private float densidad;
private float superficie;
private String nombre;
//Constructor
public  Ciudad (int cod )
```

```
{ CP = cod; poblacion = 0 ; superficie = 0 ;}
//Comandos
public void establecerNombre (String n)
{  nombre = n ; }
public void establecerPoblacion (int p )
{/*Requiere superficie > 0 */
densidad = superficie / p ; }
public void establecerSuperficie ( float s)
{  superficie = s ;}
public void establecerDensidad ( float d)
{  poblacion = d ;}
//Consultas
public String obtenerNombre()
{ return nombre;}
public int obtenerCP ()
{ return CP; }
public int obtenerPoblacion ()
{ return (int) (superficie*densidad); }
public float obtenerSuperficie ()
{ return superficie; }
public double densidad ()
{ return densidad; }
}
```

Si los atributos fueran visibles fuera de la clase `Ciudad`, sus clientes podrían haber accedido directamente al atributo `poblacion`. Al cambiar el diseño de la clase proveedora, tendría que modificarse también cada una de las clases cliente. Siguiendo los lineamientos de la programación orientada a objetos, las clases clientes de `Ciudad` solo acceden a su interfaz, que no cambió aun cuando se modificaron los atributos.

Representaciones alternativas para estructuras lineales y homogéneas

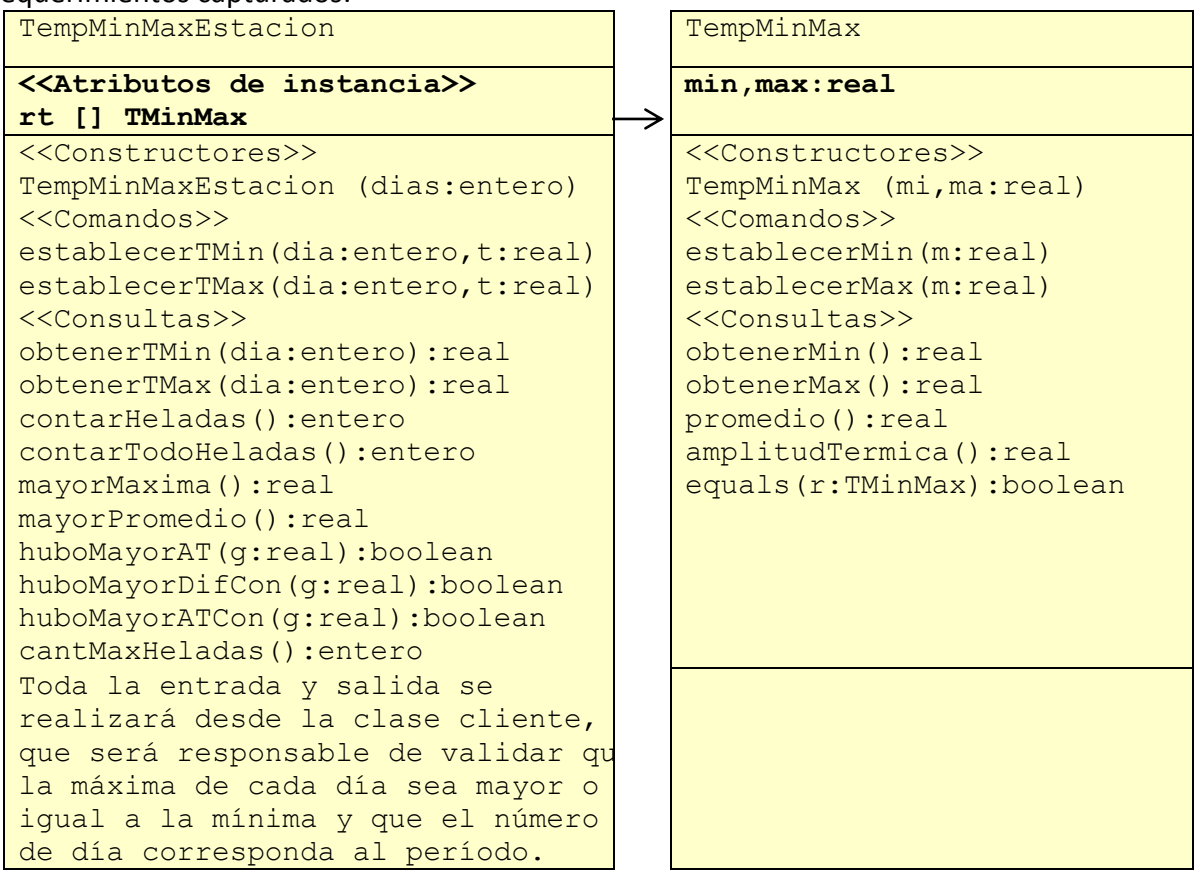
Cuando una clase modela una colección de elementos, por lo general existen diferentes estructuras alternativas para representar tanto a los elementos como a la colección misma. Si cada servicio de la clase mantiene la signatura y funcionalidad, los cambios en la representación de los datos no modifican la interfaz, aunque sí va a ser necesario modificar el código.

En este libro los casos de estudio que requieren representar colecciones de elementos pueden modelarse con estructuras lineales, homogéneas y ordenadas. Una estructura es homogénea si todos los elementos son del mismo tipo. En una estructura lineal cada elemento tiene un predecesor, excepto el primero, y un sucesor, excepto el último. La mayoría de los lenguajes de alto nivel permiten crear arreglos para definir estructuras homogéneas, lineales y ordenadas. En una estructura ordenada cada una de elemento puede accederse de acuerdo a su posición.

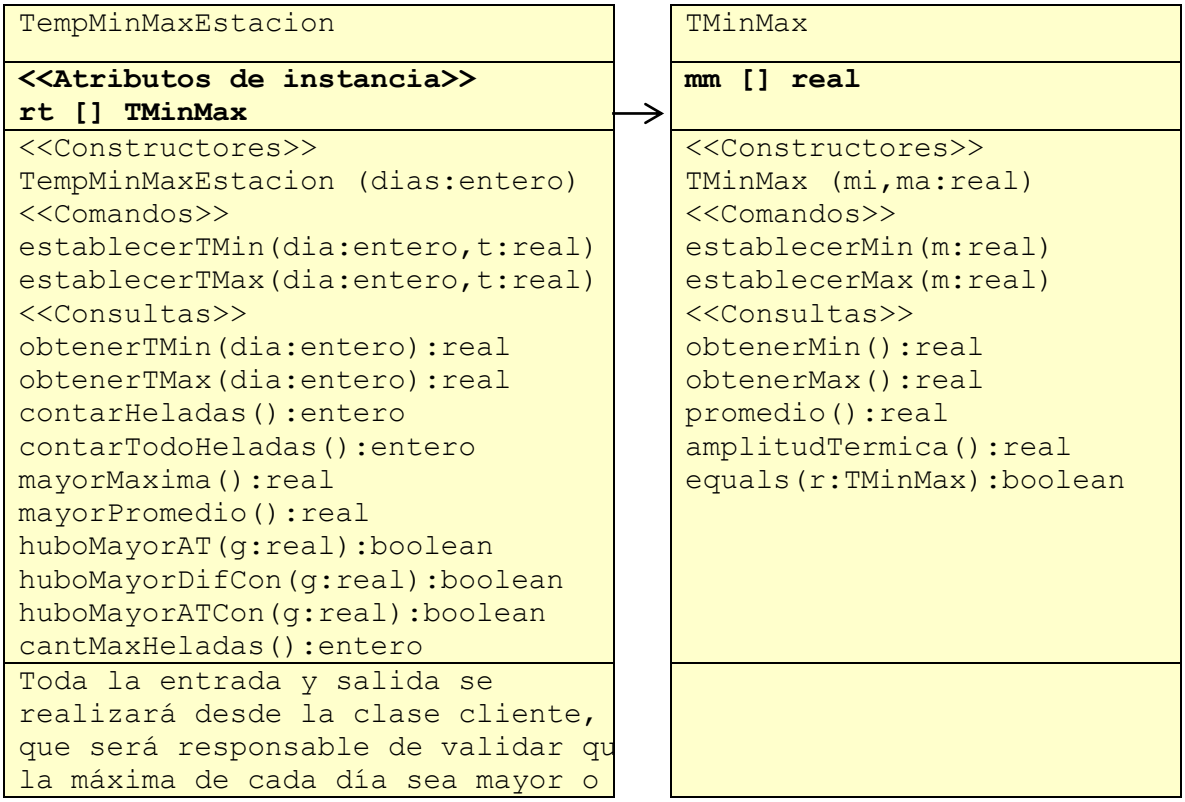
Caso de Estudio Estación Meteorológica

En una estación meteorológica se almacenan las temperaturas mínimas y máximas de cada día de un período y se computan algunos valores estadísticos. Antes de registrarse los valores se conoce la cantidad de días del período y todos los valores han sido almacenados en el momento que se computan estadísticas. El primer día del período se denota con el número 1 y así correlativamente, el último día del período corresponde al número n.

El siguiente diagrama especifica una **primera alternativa** de diseño, consistente con los requerimientos capturados:



Una **segunda alternativa** de diseño, para la misma especificación de requerimientos, puede ser:



igual a la mínima y que el número de día corresponda al período.	

Cada clase conforma un módulo o unidad de programación, a partir de los cuales se crean objetos y se les envían mensajes, sin conocer su implementación. El cambio en `TMinMax` no afecta `TempMinMaxEstacion`.

El **tercer diseño** alternativo es:

TempMinMaxEstacion
<<Atributos de Instancia>> **min [] real** **max [] real**
<<Constructores>> TempMinMaxEstacion (dias:entero) <<Comandos>> establecerTMin(dia:entero,t: real) establecerTMax(dia:entero,t: real) <<Consultas>> obtenerTMin(dia:entero):real obtenerTMax(dia:entero):real contarHeladas():entero contarTodoHeladas():entero mayorPromedio():real huboMayorAT(g:real):boolean huboMayorDifCon(g:real):boolean huboMayorATCon(g:real):boolean cantMaxHeladas():entero
Toda la entrada y salida se realizar desde la clase cliente, que será responsable de validar que la máxima de cada día sea mayor o igual a la mínima y que el número de día corresponda al período.

En las dos primeras alternativas las componentes de la colección son objetos de una clase especificada por el diseñador. Es decir, las componentes de la estructura de datos son también estructuras de datos. En la tercera alternativa, la clase `TempMinMaxEstación` esconde a dos estructuras de datos, cuyas componentes son valores elementales.

Cualquiera sea el diseño, las responsabilidades de la clase `TempMinMaxEstacion` son:

```
El primer día se referencia con 1.
Los servicios que reciben un parámetro que corresponde al día
requieren que este valor sea válido
La estructura está completa,  la cantidad de elementos se define en el
momento de la creación y es siempre mayor a 0
```

El propósito de cada servicio de la clase `TempMinMaxEstacion` es:

`contarHeladas()`: cuenta la cantidad de días del período en los que heló en algún momento del día.

`contarTodoHeladas()`:cuenta la cantidad de días del período en los que heló todo el día.

`mayorMaxima()`: computa la mayor temperatura máxima

`mayorPromedio()`: computa el mayor promedio entre la mínima y la máxima

`huboMayorAT(g:real)` : retorna true si y solo sí hubo al menos un día con amplitud térmica mayor a g.

`huboMayorATCon(g:real)`: retorna true sí y solo sí hubo al menos dos días consecutivos con amplitud térmica mayor a g.

`huboMayorATnCon(g:real,n:entero)`: retorna true sí y solo sí hubo al menos n días consecutivos con amplitud térmica mayor a g.

`huboMayorATNCon(g:real,n:entero)`: retorna true sí y solo sí hubo exactamente n días consecutivos con amplitud térmica mayor a g.

`huboMayorDifCon(g:real)`: retorna true sí y solo sí hubo al menos dos días consecutivos entre los cuales la diferencia absoluta entre las amplitudes térmicas fue mayor que g.

`cantMaxHeladas()`: retorna la duración en días del período más largo con heladas en días consecutivos.

La implementación de cada método requiere diseñar un algoritmo que va a quedar escondido en la clase. En el diseño de cada uno seguimos los lineamientos de la programación estructurada.

La representación de los datos es transparente para las clases relacionadas con `TempMinMaxEstacion`. De hecho, es posible elegir una de las tres alternativas propuestas y luego cambiarla por otra de las representaciones, sin afectar a las clases cliente, ya que no cambia la interfaz ni las responsabilidades.

Como el primer día del período se referencia con 1, si las temperaturas se almacenan en un arreglo va a ser necesario establecer un mapeo entre el día y el subíndice. Así, el día uno se almacena en la componente del arreglo que corresponde al subíndice 0, el día 2 en el subíndice 1 y así siguiendo.

Patrones de diseño de algoritmos

Con frecuencia los recorridos sobre las estructuras de datos responden a **patrones de algoritmos**, independientes del tipo de los elementos de la estructura. Por ejemplo, es habitual recorrer una estructura de datos lineal y homogénea para contar la cantidad de elementos que satisfacen una propiedad, el algoritmo puede ser entonces:

```
Algoritmo contar
DS contador
contador se inicializa en 0
para cada elemento de la estructura
  si el elemento cumple la propiedad
    incrementar el contador
```

El algoritmo `contar` describe el **comportamiento abstracto** de los servicios `contarHeladas` y `contarTodoHeladas`. El algoritmo puede **refinarse** en versiones más específicas:

```
Algoritmo cantHeladas
DS contador
contador se inicializa en 0
para cada día del período
  si temperatura minima <= 0
    incrementar el contador
```

```
Algoritmo cantTodoHeladas
DS contador
contador se inicializa en 0
para cada día del período
  si temperatura maxima <= 0
    incrementar el contador
```

Los algoritmos que implementan los servicios provistos por una clase, están **escondidos** dentro de la clase. Si se modifican, en tanto la signatura no cambie, la modificación es **transparente** para las clases cliente. Por ejemplo, el cómputo de la cantidad de componentes que verifican una propiedad dentro de un período, puede plantearse recursivamente:

Caso trivial: En un período vacío la cantidad de componentes que verifican una propiedad es 0.

Caso recursivo: En un período no vacío, la cantidad de componentes que verifican una propiedad es la cantidad de componentes que verifican la propiedad en el período que resulta de no considerar el último día, más 1 si el último día verifica la propiedad.

Caso recursivo: En un período no vacío, la cantidad de componentes que verifican una propiedad es la cantidad de componentes que verifican la propiedad en el período que resulta de no considerar el último día, si el último día NO verifica la propiedad.

Las clases relacionadas con `TempMinMaxEstacion` no *saben* si `contarHeladas` se implementa a partir de un algoritmo iterativo o un planteo recursivo.

Otro recorrido habitual sobre una estructura consiste en hallar el mayor elemento, de acuerdo a una relación de orden establecida. En este caso el algoritmo es:

```
Algoritmo mayor
DS mayor
mayor ← primer elemento
para cada elemento de la estructura a partir del segundo
  si elemento > mayor
    mayor ← elemento
```

El algoritmo propone un recorrido exhaustivo, todos los elementos del arreglo deben ser considerados para computar el mayor. En cada iteración se accede a un elemento particular, comenzando por el segundo. Nuevamente este patrón general es una **abstracción**, que puede refinarse para cada problema específico:

```
Algoritmo mayorPromedio
DS mayor
mayor ←promedio del primer día
para cada día del período
  si promedio del día > mayor
    mayor ← promedio del día
```

```
Algoritmo mayorMaxima
DS mayor
mayor ← máxima del primer día
para cada día del período
  si la máxima del día > mayor
    mayor ←máxima del día
```

Observemos que estamos utilizando una notación informal, en contraposición a la sintaxis rigurosa de un lenguaje de programación. Podemos usar el símbolo ← para denotar asignación o utilizar una notación verbal como “inicializar mayor con el promedio del primer día” o “asignar el promedio del primer día a mayor”

Para decidir si algún elemento de la estructura verifica una propiedad no es necesario hacer un recorrido exhaustivo:

```
Algoritmo verifica
```

```
DS verifica
verifica ← falso
para cada elemento de la estructura y mientras no verifica
  si el elemento actual verifica la propiedad
    verifica ← verdadero
```

Para el caso específico de decidir si la amplitud térmica fue mayor a `g` en algún día del período, el patrón se refina:

```
Algoritmo huboMayorAG
DE g
DS verifica
verifica ← falso
para cada día del período y mientras no verifica
  si la amplitud térmica del día > g
    verifica ← verdadero
```

El mismo patrón puede refinarse un poco más para modelar la solución de decidir si hubo dos días seguidos con amplitud térmica mayor a un valor g.

```
Algoritmo huboMayorATCon
DE g
DS verifica
verifica ← falso
para cada día del período excepto el último y mientras no verifica
  si la amplitud térmica del día > g y
     la amplitud térmica del día siguiente > g
    verifica ← verdadero
```

El algoritmo para decidir si hubo al menos n días consecutivos con temperaturas mayores a `g` implica nuevamente un recorrido no exhaustivo:

```
Algoritmo huboMayorATnCon
DE g,n
DS verifica
inicializa contador en 0
para cada día del período y mientras contador < n
  si la amplitud térmica del día > g
    incrementa el contador
  sino
    inicializa contador en 0
verifica es verdadero si contador = n
```

Si el problema plantea decidir si hubo exactamente `n` días que verifiquen la propiedad:

```
Algoritmo huboMayorATNCon
DE g,n
DS verifica
inicializa contador en 0
para cada día del período y mientras contador <= n
  si la amplitud térmica del día > g
    incrementa el contador
  sino
    inicializa contador en 0
verifica es verdadero si contador = n
```

Ejercicio: Diseñe algoritmos para `huboMayorDifCon(g:real)` y

`cantMaxHeladas()`.

Implementaciones alternativas

En los dos primeros diseños alternativos propuestos, las clases `TempMinMaxEstacion` y `TMinMax` están **asociadas**. En ambos casos la implementación parcial de `TempMinMaxEstacion` a partir de los algoritmos propuestos, será:

```
class TempMinMaxEstacion {
//Atributo de instancia
  private TMinMax [] rt;
//constructor
public TempMinMaxEstacion (int dias){
/*Cada elemento del arreglo representa un día del período*/
  rt = new TMinMax [dias];
  for (int dia = 0;dia < dias;dia++)
    rt[dia] = new TMinMax(0,0);}
//Comandos y Consultas triviales
/*Requiere que la clase Cliente haya controlado que max > min*/
public void establecerTempMin (int dia, float t){
  rt[dia-1].establecerMin(t);}
public void establecerTempMax (int dia,float t){
  rt[dia-1].establecerMax(t);}
public float obtenerTempMin (int dia){
  return rt[dia-1].obtenerMin();}
public float obtenerTempMax (int dia){
  return rt[dia-1].obtenerMax();}
// Consultas
public int cantDias(){
  return rt.length;}
public int cantHeladas(){
/*Calcula la cantidad de días con temperatura mínima menor a 0*/
  int cant=0;
  for (int dia=0;dia<cantDias();dia++)
    if (rt[dia].obtenerMin()<0) cant++;
  return cant;}
public float mayorPromedio(){
/*Computa el mayor promedio entre la maxima y la minima
Requiere que el periodo tenga al menos 1 día*/
  float mayor = rt[0].promedio();
  float m;
  for (int dia=1;dia<cantDias();dia++){
    m = rt[dia].promedio();
    if (m > mayor)
      mayor = m;}
  return mayor;}
}
```

Ejercicio: Complete la implementación de TempMinMaxEstacion.

La clase `TempMinMaxEstacion` se implementa sin conocer el código de `TMinMax`, solo conocemos la interfaz de esta clase, especificada en el diagrama. Para el primer diagrama de `TMinMax:`

TMinMax
min,max:real

La implementación de puede ser:

```
class TMinMax{
//Atributos de instancia
private float minima;
```

```
private float maxima;
//Constructor
public TMinMax (float mi,float ma){
//Requiere ma >= mi
  minima = mi;
  maxima = ma;}
//Comandos
public void establecerMin(float m){
  minima = m;}
public void establecerMax(float m){
  maxima = m;}
//Consultas
public float obtenerMin(){
  return minima;}
public float obtenerMax(){
  return maxima;}
public float promedio(){
  return (minima+maxima)/2;}
public float amplitudTermina(){
  return maxima-minima;}
public boolean equals(TMinMax r){
  return minima == r.obtenerMin() &&
         maxima == r.obtenerMax();}}
```

Consideremos ahora el **segundo diseño**, en el cual se modifica la clase `TMinMax`:

TMinMax
mm [] real

La implementación es:

```
class TMinMax{
//Atributos de instancia
private float [] mm;
//Constructor
public TMinMax (float mi,float ma){
//Requiere ma >= mi
  mm = new float [2];
  mm[0] = mi;
  mm[1] = ma;}
//Comandos
public void establecerMin(float m){
  mm[0] = m;}
public void establecerMax(float m){
  mm[1] = m;}
//Consultas
public float obtenerMin(){
  return mm[0];}
public float obtenerMax(){
  return mm[1];}
public float promedio(){
  return (mm[0]+mm[1])/2;}
public float amplitudTermina(){
  return mm[1]-mm[0];}
public boolean equals(TMinMax r){
  return mm[0] == r.obtenerMin() &&
         mm[1] == r.obtenerMax();}
}
```

Notemos que el cambio no tendría impacto en la clase `TempMinMaxEstacion`, ya que esta clase no accede directamente a los atributos de `TMinMax`.

Los atributos de instancia en la tercera alternativa de diseño son:

TempMinMaxEstacion
min [] real max [] real

En este caso sí es necesario modificar el código de `TempMinMaxEstacion`. La implementación parcial es entonces:

```
class TempMinMaxEstacion {
//Atributo de instancia
  private float [] min;
  private float [] max;
//constructor
public TempMinMaxEstacion (int dias){
//Cada elemento del arreglo representa un día del período
  min = new float [dias];
  max = new float [dias];}
//Comandos y Consultas triviales
// Requiere que la clase Cliente haya controlado que max > min
public void establecerTempMin (int dia, float t){
  min[dia-1] = t;}
public void establecerTempMax (int dia,float t){
  max [dia-1] = t;}
public float obtenerTempMin (int dia){
  return min[dia-1];}
public float obtenerTempMax (int dia){
  return max[dia-1];}
// Consultas
public int cantDias(){
  return min.length;}
public int cantHeladas(){
//Calcula la cantidad de días con temperatura mínima menor a 0
  int cant=0;
  for (int dia=0;dia<cantDias();dia++)
    if (min[dia]<0) cant++;
  return cant;}
public float mayorPromedio(){
/*Computa el mayor promedio entre la maxima y la minima
Requiere que el periodo tenga al menos 1 día*/
  float mayor = (min[0]+max[0])/2;
  float m;
  for (int dia=1;dia<cantDias();dia++){
    m = (min[dia]+max[dia])/2;
    if (m > mayor)
      mayor = m;}
  return mayor;}
}
```

Ejercicio: Completar la implementación de esta versión de `TempMinMaxEstación`.

A pesar del cambio de representación de los datos, las dos implementaciones corresponden a los mismos patrones de diseño para los algoritmos.

Encapsulamiento y Clases relacionadas

La modificación de la representación de los datos en una clase implica reescribir el código de algunos o incluso todos los servicios. Si los atributos están escondidos, el cambio en la representación es transparente para las clases asociadas, en tanto no cambie la función de cada servicio y las responsabilidades.

Las clases clientes de `TempMinMaxEstacion` solo pueden acceder a los atributos a través de los servicios provistos en la interfaz. Si cambia la representación, el cambio no impacta sobre las clases relacionadas.

En particular, es posible implementar una misma clase tester para verificar las dos versiones de `TempMinMaxEstación`. La clase `TestTempMinMaxEstacion` verifica el método `mayorPromedio` para tres casos de prueba establecidos con valores fijos.

```
class TestTempMinMaxEstacion {
public static void main(String[] args) {
/* Verifica el método mayorPromedio para tres casos de prueba
significativos*/
 TempMinMaxEstacion est;
 int cantD =7;
 est =genTempMinMaxEst() ;
 System.out.println("Muestra la estación  ");
 mostrarTempMinMaxEst(est) ;
 System.out.println(" El mayor promedio es "+est.mayorPromedio());
 // Caso de prueba: El mayor promedio se produjo en el primer día
  est.establecerTempMin(1,15);
  est.establecerTempMax(1,25);
  mostrarTempMinMaxEst(est) ;
  System.out.println(" El mayor promedio es "+est.mayorPromedio());
 // Caso de prueba: El mayor promedio se produjo el último día
  est.establecerTempMin(est.cantDias(),20);
  est.establecerTempMax(est.cantDias(),30);
  mostrarTempMinMaxEst(est) ;
  System.out.println(" El mayor promedio es "+est.mayorPromedio());}
public static TempMinMaxEstacion genTempMinMaxEst( ){
  TempMinMaxEstacion e;
  int cantD =7;
  e=new TempMinMaxEstacion(cantD);
  for (int dia= 1; dia<e.cantDias()-1; dia++){
    e.establecerTempMin(dia,-1);
    e.establecerTempMax(dia,dia+8);}
  e.establecerTempMin(3,5);
  e.establecerTempMax(3,15);
  e.establecerTempMin(e.cantDias(),5);
  e.establecerTempMax(e.cantDias(),11);
  return e;}
public static void mostrarTempMinMaxEst( TempMinMaxEstacion est){
 for (int dia=1; dia<=est.cantDias(); dia++)
     System.out.println(" "+est.obtenerTempMin(dia)+
                        " "+est.obtenerTempMax(dia) );}
}
```

Tanto si los valores están establecidos en la clase tester, son leídos por consola o desde un archivo, todos los datos están almacenados en el arreglo antes de que se realice cualquier procesamiento.

Ejercicio: Complete la clase tester con mensajes adecuados para probar los servicios provistos por `TempMinMaxEstacion`

Ejercicio: Modifique la clase tester para que los valores se lean de un archivo secuencial.

Ejercicio: Implemente el siguiente modelo que propone otra alternativa para modelar el mismo problema y verifique la clase con el mismo tester. La interfaz de la clase y las responsabilidades no se modifican.

TempMinMaxEstacion
minmax [] [] real

En la estación meteorológica la cantidad de componentes de la estructura de datos corresponde a la cantidad de días del período y se conoce en el momento que se crea el objeto de clase `TempMinMaxEstacion`. Las temperaturas de cada día se almacenan antes de que comience el procesamiento, es decir la estructura está completa. En muchas aplicaciones el procesamiento se realiza aun cuando la estructura no está completa, esto es, las consultas y modificaciones de los datos se entrelazan, como veremos más adelante, utilizando otro caso de estudio.

Abstracción y programación orientada a objetos

El análisis y diseño orientado a objetos demanda un proceso de abstracción a partir del cual se identifican los objetos del problema y se agrupan en clases. Una clase es una abstracción, un patrón que caracteriza los atributos y el comportamiento de un conjunto de objetos.

En la implementación, una clase que establece atributos y servicios define un **tipo de dato** a partir del cual es posible crear instancias. El conjunto de valores queda determinado por el tipo de los atributos, el conjunto de operaciones corresponde a los servicios provistos por la clase. Si la representación de los datos está escondida, la clase define un **tipo de dato abstracto (TDA)**.

La definición de tipos de datos abstractos es un recurso importante porque favorece la reusabilidad y permite reducir el impacto de los cambios. La modificación de una clase que define un tipo de dato abstracto, puede realizarse sin afectar a las clases que crean instancias del tipo.

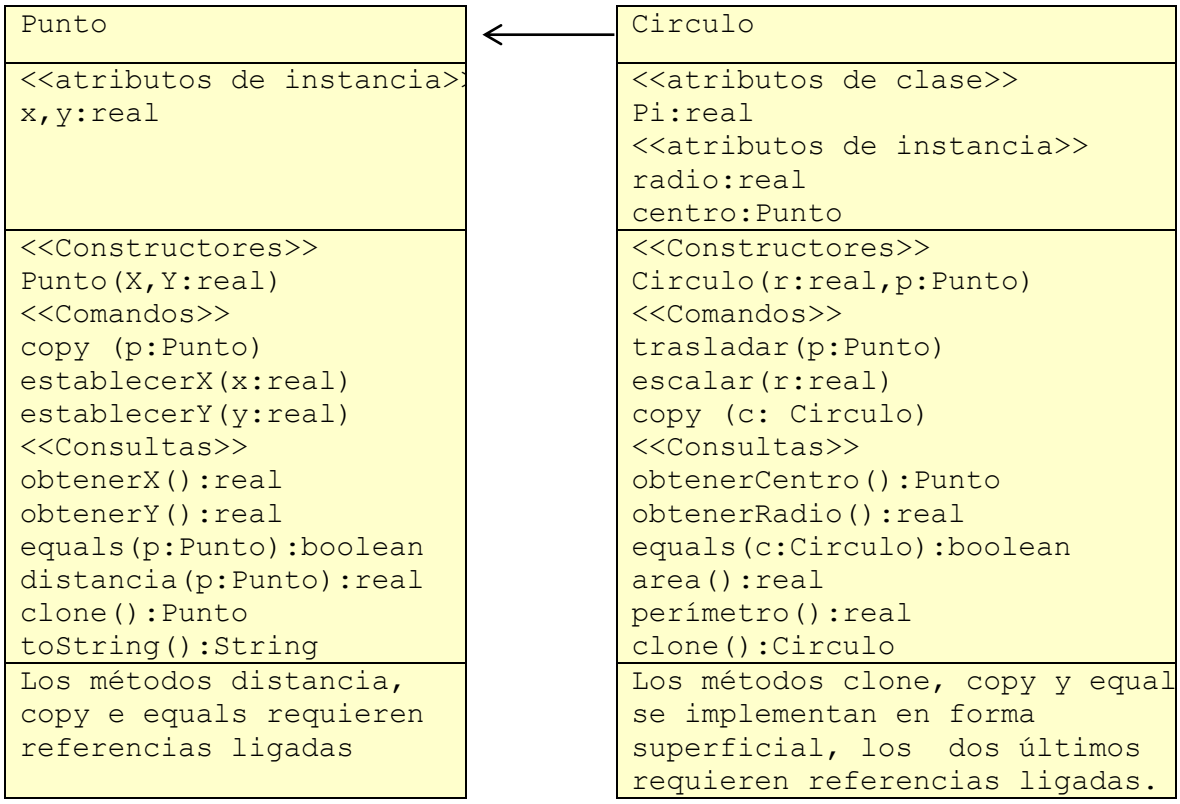
En Java la definición de clases y los modificadores de accesos, permiten que el programador defina nuevos tipos de datos abstractos a partir de los cuales se crean instancias. Todas las clases implementadas en los casos de estudio propuestos hasta el momento definen tipos de datos abstractos. A continuación presentamos varios casos de estudio, cada uno de los cuales define TDA asociados.

TDA y clases relacionadas

El siguiente caso de estudio ilustra como un cambio en la implementación del TDA `Circulo`, es transparente para las clases relacionadas por asociación o dependencia.

Caso de Estudio: Círculo y Punto

Un sistema de software didáctico de geometría para prescolar brinda facilidades para que los niños identifiquen y clasifiquen figuras geométricas. En el desarrollo del sistema el diseñador ha elaborado un diagrama con más de 50 clases, entre ellas Circulo y Punto, definidos como TDA. Implemente los TDA Circulo y Punto modelados en el siguiente diagrama:



Las clases `Circulo` y `Punto` están relacionadas por asociación. La clase `Circulo` tiene un atributo de clase `Punto`. El constructor de `Circulo` y el método `trasladar` reciben un parámetro de clase `Punto`. El método `obtenerCentro` retorna como resultado un objeto de clase `Punto`. Existe también una relación de dependencia entre `Circulo` y `Punto`.

La implementación de la clase `Punto` es:

```
class Punto{
//Atributos de Instancia
private float x,y;
//Constructores
public Punto (float x,float y){
  this.x = x;  this.y = y;}
public void establecerX (float x){
  this.x = x;}
public void establecerY (float y){
  this.y = y;}
public void copy (Punto p){
//Require p ligada
  x = p.obtenerX();
  y = p.obtenerY();}
//Consultas
public float obtenerX(){
  return x;}
public float obtenerY(){
```

```
  return y;}
public double distancia(Punto p) {
//Require p ligada
    float dx= x - p.obtenerX();
    float dy= y - p.obtenerY();
    return Math.sqrt(dx*dx+dy*dy);}
public  boolean equals (Punto p){
//Require p ligada
  return x==p.obtenerX() && y==p.obtenerY();}
public Punto clone(){
 return new Punto(x,y);}
public String toString(){
  return "("+x+","+y+")";}
}
```

La clase `Punto` define un tipo de dato abstracto a partir del cual es posible crear instancias. El conjunto de valores es el conjunto de pares de valores de tipo `float`. Cada servicio provisto por la clase corresponde a una operación provista por el tipo.

La clase `Circulo` mantiene un atributo de instancia del TDA `Punto`, como ilustra la siguiente implementación:

```
class Circulo {
//Atributos de clase
private static final float Pi= (float) 3.1415;
//Atributos de Instancia
private float radio;
private Punto centro;
//Constructor
public Circulo(float r, Punto p){
//Requiere p ligada
  radio = r;
  centro = p;}
//Comandos
public void trasladar(Punto p){
//Require p ligada
  centro = p;}
public void escalar(float r){
  radio = r;}
public void copy (Circulo c){
//Require c ligada
  radio = c.obtenerRadio();
  centro = c.obtenerCentro();}
//Consultas
public Punto obtenerCentro(){
  return centro;}
public float obtenerRadio(){
  return radio;}
public float perimetro(){
  return obtenerRadio()*2*Pi;}
public double area(){
  return Pi * obtenerRadio()* obtenerRadio();}
public boolean equals(Circulo c){
//Require c ligada
return obtenerRadio()== c.obtenerRadio() &&
```

```
       obtenerCentro() == c.obtenerCentro();}
public Circulo clone (){
  return new Circulo (obtenerRadio(),obtenerCentro());}
}
```

La clase `Circulo` también define un tipo de dato abstracto a partir del cual es posible crear instancias. La clase `Circulo` es cliente de la clase `Punto` y a su vez es proveedora de servicios para las clases que la usan.
Consideremos ahora el siguiente diseño alternativo para la clase *`Circulo`*:

Circulo
<<atributos de instancia>> centro:Punto punto:Punto
<<Constructores>> Circulo(r:real,p:Punto) <<Comandos>> trasladar(p:Punto) escalar(r:real) copy (c: Circulo) <<Consultas>> obtenerCentro():Punto obtenerRadio():real equals(c:Circulo):boolean area():real perímetro():real clone():Circulo

En este diseño, un objeto de clase `Circulo` se modela a través de dos atributos de instancia que corresponden al TDA `Punto`. Uno corresponde al centro y otro a un punto cualquiera de la circunferencia.

Observemos que el cambio en la representación no modifica la signatura de ninguno de los servicios. No se modificó la interfaz de la clase, aunque sí es necesario implementar de manera diferente varios de los servicios. Como los atributos están escondidos, el cambio no afecta a la clase `Punto`, proveedora de la clase `Circulo`, ni a las clases clientes de la clase `Circulo`.

El constructor de `Circulo` se implementa ahora como:

```
public Circulo (float r, Punto p){
//Require p ligada
  centro = p;
  punto = new Punto (centro.obtenerX(), centro.obtenerY()+r);}
```

El método `obtenerRadio()` se modifica porque el radio ya no es un atributo de la clase `Circulo`.

```
public float obtenerRadio(){
  return centro.distancia(punto);}
```

Las clases dependientes o asociadas a la clase `Circulo` no perciben el cambio de diseño ni de la implementación porque solo acceden a la interfaz de la clase proveedora.
Ejercicio

Complete la clase Circulo para este diseño alternativo, analizando qué métodos es necesario modificar.

Defina una única clase tester que permita verificar los servicios de las dos implementaciones.

TDA y estructuras homogéneas, lineales y con acceso directo

En la resolución de problemas de mediana y gran escala el diseño de las estructuras de datos es un tema relevante. Una estructura de datos es una colección de datos organizados de alguna manera. La organización está ligada a cómo van accederse las componentes.

Una estructura es homogénea si todas las componentes son del mismo tipo. En una estructura lineal cada componente tiene un predecesor, excepto el primero, y un sucesor, excepto el último. Una estructura tiene acceso directo si cada una de sus componentes puede accederse sin necesidad de acceder a las que la preceden o suceden.

La mayoría de los lenguajes de alto nivel permiten crear arreglos para definir estructuras homogéneas, lineales y con acceso directo.

Caso de Estudio TDA Matriz-Vector

Un sistema de software didáctico de álgebra elemental brinda facilidades para que los jóvenes operen con matrices y vectores. En el desarrollo del sistema el diseñador ha elaborado un diagrama con más de 50 clases, entre ellas Matriz y Vector, definidos como TDA. Implementar los TDA `Matriz` **y** `Vector` a partir del siguiente diagrama:

Matriz
mr [][] real
<<Constructor>> Matriz(nf,nc:entero) <<Comandos>> establecerElem (f,c:entero, elem:real) copy(m:Matriz) establecerIdentidad () invertirFilas(f1,f2:entero) xEscalar(r:real) <<Consultas>> existePos(f,c:entero):boolean obtenerNFil ():entero obtenerNCol ():entero obtenerElem (f,c:entero):real clone():Matriz equals(m:Matriz): boolean esCuadrada ():boolean esIdentidad():boolean esTriangularSuperior():boolean eSimetrica():boolean esRala():boolean cantElem (elem:real):entero mayorElemento ():real filaMayorElemento ():entero vectorMayores ():Vector suma (m:Matriz):Matriz producto(m:Matriz):Matriz transpuesta():Matriz

Vector
v [] real
<<Constructor>> Vector(n:entero) <<Comandos>> establecerElem (f:entero, elem:real) copy(m:Vector) xEscalar(r:real) <<Consultas>> existePos(f:entero):boolean obtenerN ():entero obtenerElem (f:entero):real clone():Vector equals(m:Vector): boolean mayorElemento ():real productoEscalar (m:Vector): real productoVectorial(m:Vector): Vector

Todos los servicios que reciben objetos como parámetros requieren referencias ligadas.	Todos los servicios que reciben objetos como parámetros requieren referencias ligadas.

La clase `Matriz` depende de la clase `Vector` porque el servicio `VectorMayores()` retorna un objeto de clase `Vector`.

`establecerElem (f,c:entero, elem:real):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`establecerIdentidad ():` requiere `obtenerNCol()` igual a `obtenerNCol()`

`invertirFilas(f1,f2:entero):` requiere `0<=f1,f2 <obtenerNFil()`

`obtenerElem (f,c:entero):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`esRala():` retorna verdadero si más de la mitad de los elementos son igual a 0

`vectorMayores ():` genera un objeto de clase `Vector` en el cual el elemento i es el mayor de la fila i de la matriz.

El método `xEscalar` implementa el siguiente algoritmo:

```
Algoritmo xEscalar
DE r
para cada posición i,j
   mr_{i,j} = mr_{i,j}*r
```

Observe que en el método `copy`, `m` es un objeto de clase `Matriz` y `mr` es un arreglo de dos dimensiones. Para acceder a los elementos de `m` se usan los servicios provistos por la clase, mientras que los elementos de `mr` se accedan con subíndices.

Un planteo recursivo para decidir si una matriz es la matriz identidad puede ser:

Caso trivial

Una matriz **mr** de 1x1 es la matriz identidad si su único elemento **mr**[0,0] es 1

Caso recursivo

Una matriz **mr** de nxn con n > 1 es la matriz identidad si

mr[i,n-1] = 0 para 0<= i < n-1

mr[n-1,j] = 0 para 0<= j < n-1

mr[n-1,n-1] = 1

y la matriz **mr'** de (n-1)x(n-1) es la matriz identidad

Propondremos la implementación de algunos de los servicios de la clase. Queda pendiente como ejercicio completar el código de `Matriz` y desarrollar completa la clase `Vector`.

```
class Matriz {
private float[][] mr;
// Constructor
public Matriz(int nfil,int ncol){
  mr=new float[nfil][ncol];}
//Consultas
public int obtenerNFil ()    {
  return mr.length; }
public int obtenerNCol ()    {
  return mr[0].length; }
public boolean existePos(int f, int c) {
  return (f>=0 && f < obtenerNFil()) &&
          (c>=0 && c < obtenerNCol());}
public float obtenerElem(int f,int c){
//Requiere f y c consistentes
  return mr[f][c];}
//Comandos
public void establecerElemento (int f,int c,
                  float x) {
//Requiere f y c consistentes
     mr[f][c]= x;}
public void establecerIdentidad () {
//Requiere que el número de filas sea igual al de columnas
    iniMatriz();
    for (int j=0;j<obtenerNCbol();j++)
      mr[j][j] = 1;    }
public void iniMatriz () {
  for (int i=0;i<obtenerNFil();i++)
    for (int j=0;j<obtenerNCol();j++)
       mr[i][j] = 0;    }
public void xEscalar (float r) {
//Multiplica cada elemento de la matriz por r
 for (int i=0;i<obtenerNFil();i++)
   for (int j=0;j<obtenerNCol();j++)
      mr[i][j] = mr[i][j] * r;}
public void copy (Matriz m) {
/*Crea un nuevo objeto para el atributo de instancia con el nro de
filas y columnas de m */
 mr = new float[m.obtenerNFil()][m.obtenerNCol()];
 for (int i=0;i<obtenerNFil();i++)  {
   for (int j=0;j<obtenerNCol();j++)
         mr[i][j] = m.obtenerElem(i,j); } }
public int cantElem (float ele)  {
    int cant = 0;
     for (int i=0;i< obtenerNFil();i++)
       for (int j=0;j< obtenerNCol();j++)
         if (mr[i][j] == ele) cant++ ;
    return cant;}
public boolean esCuadrada () {
   return (obtenerNFil() == obtenerNCol()) ; }
```

```
public boolean esIdent () {
  if (!esCuadrada())return false;
  else return esIdentidad(obtenerNFil());}
private boolean esIdentidad (int n) {
 if (n == 1)return (mr[0][0] == 1);
 else
   return (mr[n-1][n-1] == 1 &&
           esCeroFila(n-1) &&
           esCeroColumna(n-1)&&
           esIdentidad(n-1));}
}
```

La verificación de los TDA `Matriz` y `Vector` requiere como siempre establecer un conjunto de casos de prueba para cada operación provista por el tipo. Los servicios de la clase `Matriz` asumen que cada clase cliente cumple con los compromisos establecidos en el contrato. Cuando no basta que una solución sea correcta sino que se pretende que el sistema sea confiable, cada clase puede prevenir errores provocados por el incumplimiento de los compromisos de las clases clientes. Una manera de implementar el mecanismo de prevención es el manejo de excepciones.

Caso de Estudio TDA Racional MatrizRacional y **VectorRacional**

Implementar los TDA `Racional`, `MatrizRacional` **y** `VectorRacional` definidos a partir del siguiente diagrama:

MatrizRacional
mr [][] Racional
<<Constructor>> MatrizRacional(nf,nc:entero) <<Comandos>> establecerElem (f,c:entero, elem:Racional) copy(m:MatrizRacional) establecerIdentidad () invertirFilas(f1,f2:entero) xEscalar(r:Racional) <<Consultas>> existePos(f,c:entero):boolean obtenerNFil ():entero obtenerNCol ():entero obtenerElem (f,c:entero):Racional clone():MatrizRacional equals(m:MatrizRacional): boolean esCuadrada ():boolean esIdentidad():boolean esTriangularSuperior():boolean eSimetrica():boolean esRala():boolean cantElem (elem:Racional):entero mayorElemento ():Racional filaMayorElemento ():entero vectorMayores ():VectorRacional suma (m:MatrizRacional):MatrizRacional producto(m:MatrizRacional):

VectorRacional
v []Racional
<<Constructor>> VectorRacional(n:entero) <<Comandos>> establecerElem (f:entero, elem:Racional) copy(m:Racional) xRacional(r:Racional) <<Consultas>> existePos(f:entero): boolean obtenerN ():entero obtenerElem (f:entero):Racional clone():VectorRacional equals(m:VectorRacional): boolean mayorElemento ():Racional productoEscalar (m:VectorRacional): Racional productoVectorial(m:Vector): Racional

MatrizRacional transpuesta():MatrizRacional
Todos los servicios que reciben objetos como parámetros requieren referencias ligadas. Las búsqueda y comparaciones se hacen por equivalencia

Todos los servicios que reciben objetos como parámetros requieren referencias ligadas.

Racional
numerador:entero denominador:entero
<<Constructor>> Racional (n,d:entero) <<Comandos>> establecerNumerador(n:entero) establecerDenominador(d:entero) copy(r:Racional) <<Consultas>> obtenerNumerador ():entero obtenerDenominador ():entero equals(r:Racional):boolean suma(r:Racional):Racional resta(r:Racional):Racional producto(r:Racional):Racional cociente(r:Racional):Racional
El denominador es siempre positivo E constructor y establecerDenominador requieren d>0 Dos números racionales son iguales s representan un mismo valor.

equals(…)
Computa verdadero si el objeto que recibe el mensaje representa un número racional equivalente al parámetro

En la clase `MatrizRacional`:

`establecerElem (f,c:entero, elem:real):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`establecerIdentidad ():` requiere `obtenerNCol()` igual a `obtenerNCol()`

`invertirFilas(f1,f2:entero):` requiere `0<=f1,f2 <obtenerNFil()`

`obtenerElem (f,c:entero):` requiere `0<=f <obtenerNFil()` y `0<=c <obtenerNCol()`

`esRala():` retorna verdadero si más de la mitad de los elementos representan el valor cero (con cualquier denominador positivo)

`vectorMayores ():` genera un objeto de clase `VectorRacional` en el cual el elemento i es el mayor de la fila i de la matriz.

La implementación parcial de `Racional` es:

```
class Racional {
//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 equals(Racional r){
    return numerador*r.obtenerDenominador()==
           denominador*r.obtenerNumerador(); }
public Racional suma (Racional r){
  int n = numerador*r.obtenerDenominador()+
          r.obtenerNumerador()* denominador;
  int d = denominador*r.obtenerDenominador();
  Racional s = new Racional(n,d);
  return s;}
}
```

La implementación parcial de la clase `MatrizRacional` es:

```
class MatrizRacional {
private Racional[][] mr;
// Constructor
public MatrizRacional(int nfil,int ncol){
  mr=new Racional[nfil][ncol];}
//Consultas
public int obtenerNFil ()    {
  return mr.length; }
public int obtenerNCol ()    {
  return mr[0].length; }
public boolean existePos(int f, int c) {
  return (f>=0 && f < obtenerNFil()) &&
          (c>=0 && c < obtenerNCol());}
public Racional obtenerElem(int f,int c){
//Requiere f y c consistentes
  return mr[f][c];}
// Comandos
public void establecerElemento (int f,int c,
                  Racional x) {
//Requiere f y c consistentes
     mr[f][c]= x;}
public void establecerIdentidad () {
//Requiere que el número de filas sea igual al de columnas
    iniMatrizRacional();
    for (int j=0;j<obtenerNCol();j++)
      mr[j][j] = new Racional(1,1);    }
public void iniMatrizRacional () {
  for (int i=0;i<obtenerNFil();i++)
    for (int j=0;j<obtenerNCol();j++)
       mr[i][j] = new Racional(0,0);    }
public void xEscalar (Racional r) {
//Multiplica cada elemento por r
 for (int i=0;i<obtenerNFil();i++)
   for (int j=0;j<obtenerNCol();j++)
      mr[i][j] = mr[i][j].producto(r);}
public void copy (MatrizRacional m) {
/*Crea un nuevo objeto con el nro de filas y columnas de m y
referencias a los mismos elementos */
```

```
 mr = new Racional[m.obtenerNFil()][m.obtenerNCol()];
 for (int i=0;i<obtenerNFil();i++)  {
   for (int j=0;j<obtenerNCol();j++)
         mr[i][j] = m.obtenerElem(i,j); }}
public int cantElem (Racional ele)    {
/*Cuenta la cantidad de elementos equivalentes a ele que mantiene la
matriz */
    int cant = 0;
     for (int i=0;i< obtenerNFil();i++)
       for (int j=0;j< obtenerNCol();j++)
         if (mr[i][j].equals(ele)) cant++ ;
    return cant;}
public boolean esCuadrada () {
   return (obtenerNFil() == obtenerNCol()) ; }
public boolean esIdent () {
  if (!esCuadrada())
    return false;
  else return esIdentidad(obtenerNFil());}
private boolean esIdentidad (int n) {
 if (n == 1)
   return (mr[0][0].equals(new Racional(1,1)));
 else
   return (mr[n-1][n-1].equals(new Racional(1,1)) &&
           esCeroFila(n-1) &&
           esCeroColumna(n-1)&&
           esIdentidad(n-1));}
}
```

Ejercicios:

Completar el código de `MatrizRacional` *y desarrollar completa la clase* `VectorRacional`. *Complete la implementación de la clase Racional.*

Implemente nuevamente la clase Racional considerando que la sección de responsabilidades establece:

```
El denominador es siempre positivo El
constructor y establecerDenominador
requieren d>0
El constructor almacena la
representación irreductible de cada
número racional.
```

Implemente una única clase tester que permita verificar los servicios de las dos implementaciones de Racional.

TDA y estructuras parcialmente ocupadas

La abstracción permite proponer soluciones similares para problemas en apariencia muy diferentes. Esas similitudes van a permitir reusar diseño y código, favoreciendo la productividad.

En los casos de estudio que presentamos a continuación se define una clase que encapsula una estructura homogénea, lineal, con acceso directo y parcialmente ocupada. Es decir, el constructor crea una estructura para mantener cierta cantidad máxima de componentes. Desde la clase cliente, se insertan y eliminan componentes de acuerdo a funcionalidad

especificada en el diseño. Toda la entrada y salida se hace desde la clase cliente, de la clase que encapsula a la estructura de datos.

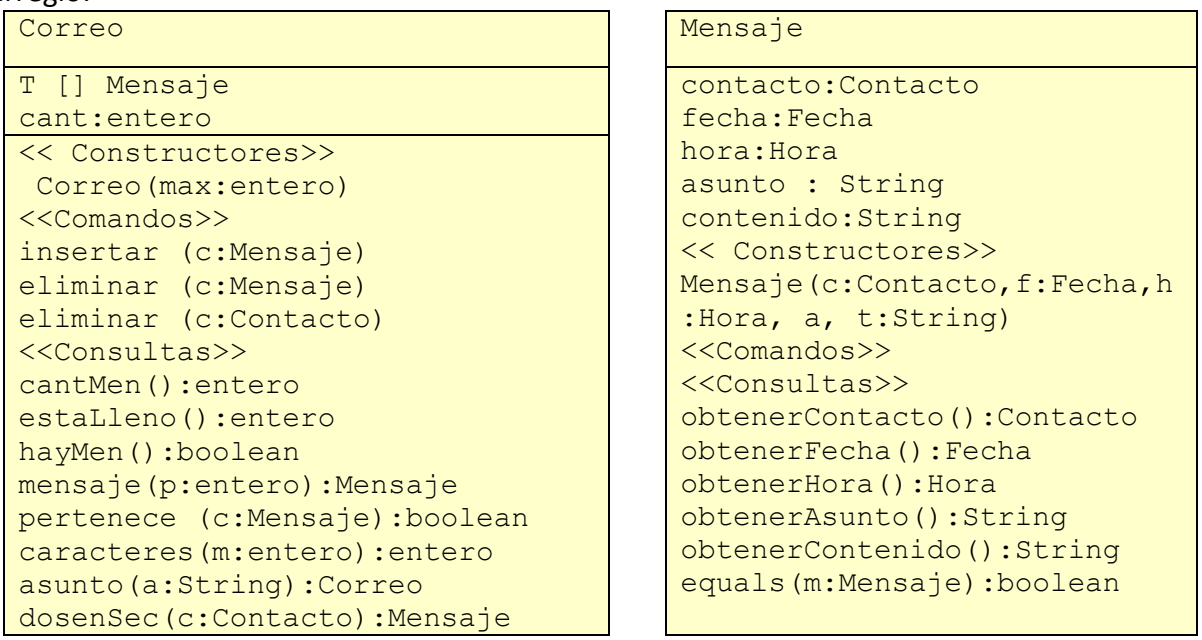
Utilizaremos el término **colección** para referirnos a una estructura con capacidad para mantener max elementos de un **tipo base** TB. En cada momento determinado solo n de los max elementos almacenan a un objeto de clase TB; más precisamente, referencian a un objeto del tipo base. Los n elementos ligados ocupan las primeras n posiciones de la estructura. De modo que las max-n componentes nulas, están comprimidas en las últimas posiciones del arreglo. La posición de cada elemento en una colección no tiene relevancia, de modo que una componente puede cambiar de posición. Por ejemplo, si un mensaje se asigna a una colección de mensajes de correo, pero su posición en la estructura no tiene significado en la aplicación.

Utilizaremos el término **tabla** para referirnos a una estructura con n elementos, cada uno de los cuales ocupa una posición que es significativa en la estructura. Por ejemplo, un micro se asigna a una unidad específica de un estacionamiento o un socio de un club tiene asignado un casillero particular que se identifica por su posición en la estructura que modela al conjunto de casilleros. Es decir las max-n componentes nulas pueden estar intercaladas con las componentes ligadas.

Caso de Estudio: Correo

Se desea mantener una secuencia de mensajes en la que pueden insertarse y eliminarse elementos, manteniendo siempre el orden de acuerdo en el cual se insertaron. También es posible recuperar un mensaje dada su posición en la secuencia, eliminar todos los mensajes de un contacto, contar cuántos mensajes tienen más de m caracteres en el contenido, generar una secuencia con todos los mensajes que corresponden a un mismo asunto y ordenar la colección de mensajes.

El siguiente diagrama modela a la colección de mensajes de correo almacenados en un arreglo:



La estructura es homogénea, todas las componentes son objetos de clase `Mensaje`. Las clases `Correo` y `Mensaje` están asociadas. La clase `Correo` encapsula a un arreglo de elementos

de clase `Mensaje` y un entero que representa la cantidad de mensajes, es decir la cantidad de elementos del arreglo que mantienen referencias ligadas. Los elementos están *comprimidos*, todos los elementos nulos están al final.

Cuando se invoca el constructor de `Correo` se crea un arreglo de `max` elementos, inicialmente nulos.

`insertar (c:Mensaje)` Asigna el objeto `c` a la posición `cant` y aumenta el valor de `cant`, que mantiene en cada momento la cantidad de elementos ocupados en el arreglo y también la posición en la que se va a insertar el próximo mensaje de correo. Requiere que la clase cliente controle que la estructura no esté llena y que el mensaje no pertenezca al correo.

`eliminar (c:Mensaje)` Busca el mensaje con la misma identidad que `c` y si existe, arrastra los que siguen una posición, de modo que no queda una referencia nula entre otras ligadas y se conserva el orden en el que se insertaron los mensajes, actualiza `cant`. Si no existe el mensaje `c`, la colección no se modifica.

`eliminar (c:Contacto)` Elimina todos los mensajes del contacto `c` y comprime los que quedan. Actualiza `cant`. Si no existen mensajes del contacto `c`, la colección no se modifica.

`pertenece (c:Mensaje):boolean` retorna verdadero si existe un mensaje con la misma identidad que c.

`caracteres(m:entero):entero` computa la cantidad de mensajes con c caracteres en el contenido.

`asunto(a:String):Correo` genera una estructura con los mensajes que corresponden al asunto a

`dosenSec(c:Contacto):Mensaje` Dada una secuencia de mensajes $S=s_1,s_2,...,s_{cant-1},s_{cant}$ la consulta `dosenSec` retorna $\mathbf{s_k}$ tal que s_k y s_{k-1} corresponden al mismo contacto c y no existe un j, j > k, tal que s_j y s_{j-1} corresponden al mismo contacto c, con 0<=k<`cant` y 0<=j<`cant`.

Observemos que los elementos pueden ser procesados, por ejemplo para calcular cuántos mensajes tienen más de `c` caracteres en el contenido, aun cuando la estructura no esté completa. Luego se insertan nuevos mensajes, se eliminan otros y se vuelve a procesar la estructura completa. La situación es diferente en otros problemas, como por ejemplo la estación meteorológica, en la cual se conoce la cantidad de elementos y la estructura está completa en el momento que se procesa.

La implementación parcial en Java de `Correo` es:

```
class  Correo {
//Atributos de Instancia
   private Mensaje[] T;
   private int cant;
/*Constructor
Crea una estructura con capacidad para max mensajes*/
public  Correo(int max) {
     T= new Mensaje [max];
     cant = 0;}
//Comandos
public void insertar  (Mensaje m) {
/*Asigna el mensaje m a la primera posición libre del arreglo, es
decir, cant. Aumenta el valor de cant. Requiere que la clase cliente
```

```
haya verificado que la colección no esté llena, m esté ligad y no
pertenezca a la colección*/
  T[cant++] = m;    }
public void eliminar  ( Mensaje m){
/* Busca un mensaje con la misma identidad que m en la secuencia de
mensajes, si lo encuentra arrastra los mensajes que siguen una
posición*/
  boolean esta = false; int i= 0;
  while (!esta && i <  cantMen())//Busca a m
     if (T [i] == m)
       esta = true;
     else
       i++;
  if (esta) {
    cant--;
    arrastrar(i);}}
private void arrastrar( int i){
/* arrastra todos los elementos una posición hacia arriba*/
   while (i < cantMen()){
     T[i] = T[i+1];
     i++;}
  T[i]=null;}
//Consultas
public int  cantMen () {
     return cant;}
public boolean estaLleno() {
     return cant == T.length;}
public boolean hayMen() {
     return cant > 0;}
public Mensaje mensaje(int p){
/*Retorna el mensaje que corresponde a la posición p, si p no es una
posición válida retorna nulo*/
   Mensaje m = null;
   if (p>=0 && p < cantMen()) m = T[p];
   return m;}
public boolean pertenece (Mensaje m){
/*Decide si algún elemento de la colección tiene la misma identidad
que m*/
  boolean esta = false;
  for (int i = 0; !esta && i <  cantMen() ; i++){
     esta = T[i] == m;}
  return esta;}
public int caracteres(int m){
/*Cuenta la cantidad de mensajes con más de m caracteres*/
int car =0;
String content;
for (int i = 0; i <  cantMen(); i++) {
    content = T[i].obtenerContenido();
    if (content.length() > m) car++;}
return car;
}
public Correo asunto(String a){
/*Genera un objeto de clase Correo solo con los objetos que
corresponden al asunto a*/
Correo n = new Correo(cantMen());
```

```
for (int i = 0; i <  cantMen() ; i++)
   if(T[i].obtenerAsunto().equals(a)){
     n.insertar(T[i]);}
return n;     }
}
```

Las clases que usan los servicios provistos por `Correo` no conocen su representación interna. La clase `Correo` usa los servicios provistos por la clase `Mensaje`, sin conocer su implementación.

La clase tester debería verificar cada servicio considerando casos significativos. Por ejemplo, para verificar el servicio `eliminar`, los casos de prueba pueden ser: eliminar el primer mensaje, el segundo, el anteúltimo, el último. También hay que considerar que el elemento a eliminar no pertenezca a la colección. La verificación, como siempre, asume que la clase cliente cumple con sus responsabilidades.

Dada un método en la clase tester que incluye las siguientes declaraciones:

```
Mensaje sms;
Correo col;
sms = new Mensaje(…);
col = new Correo(1000);
```

En la clase cliente, cada mensaje `insertar` debería controlar previamente que el mensaje no fue insertado previamente y que la estructura no está llena:

```
if (!col.pertenece(sms) && !col.estaLleno())
  col.insertar(sms);
```

Observemos que el control de pertenencia requiere recorrer la estructura completa para retornar false, que probablemente sea el caso más frecuente.
Ejercicio: Complete la implementación de la clase Correo e implementa las clases Mensaje y Contacto, considerando que un contacto tiene un nombre y una dirección de correo.

La clase `Correo` modela la colección de mensajes. El diseño puede ser parcialmente reusado para modelar otras colecciones, aunque la implementación va a ser diferente.

Caso de Estudio Inventario

Una dependencia municipal mantiene un inventario permanente de los bienes de uso, modelado de acuerdo al siguiente diagrama:

Inventario
T [] Articulo cant:entero
<< Constructores>> Inventario(max:entero) <<Comandos>> alta (c:Articulo):boolean baja (c:Articulo) depreciarRubro (r:entero,p:real) <<Consultas>> cantArt():entero estaLleno():entero recuperar (c:entero):Articulo pertenece (c:entero):boolean

Articulo
codigo:entero rubro:entero valor:real anio:entero
<<Constructor>> Articulo (c:entero,r:entero, v:entero,a:entero) <<Comandos>> depreciar(p:real) <<Consultas>> obtenerCodigo():entero obtenerRubro():entero obtenerValor():real obtenerAnio():entero equals(Articulo a):boolean

`pertenece (c:Articulo):boolean` `unRubro(r:entero):Inventario`	

Las clases `Inventario` y `Articulo` están asociadas. La clase `Inventario` encapsula a un arreglo de elementos de clase `Articulo` y un entero que representa la cantidad actual de artículos en el inventario.

Cuando se crea un objeto de clase `Inventario` se crea un arreglo de `max` elementos, inicialmente nulos. Cada vez que se utiliza el servicio `alta` aumenta el valor de `cant`, que mantiene en cada momento la cantidad de elementos ocupados en el arreglo y también la posición en la que se va a insertar el próximo elemento. Requiere que la clase cliente haya verificado que no existe un artículo con el mismo código. La clase asume la responsabilidad de controlar que la estructura no esté llena, si todos los elementos están ligados retorna false. Los elementos están "comprimidos", todos los elementos nulos están al final. Cuando se da de baja un elemento, el último se copia en su lugar, de modo que no se conserva el orden en el que se insertaron.

La implementación en Java de `Inventario` es:

```
class  Inventario {
//Atributos de Instancia
   private Articulo[] T;
   private int cant;
/*Constructor
Crea una Coleccion con capacidad para
max elementos*/
public  Inventario(int max) {
      T= new Articulo [max];
     cant = 0;}
//Comandos
public boolean alta  (Articulo elem) {
/*Inserta un elemento al final de la Coleccion, requiere que no exista
un artículo con el mismo código*/
  boolean existe=false;
  if (cant<T.length) {
       T[cant++] = elem;
       existe = true;}
  return existe;}
public void baja  ( Articulo c){
/*Busca un artículo equivalente a c y si existe copia el último en esa
posición*/
  boolean esta = false; int i= 0;
  while (!esta && i <  cantArt())
     if (T [i].equals(c)) esta = true;
     else i++;
  if (esta) {
    cant--;
    T[i] = T[cant];
    T [cant] = null;}}
public void depreciarRubro(int r, float p){
/*modifica el valor de cada artículo del rubro r decrementándolo de
acuerdo al porcentaje p.*/
float v;
for (int i = 0; i <  cantArt() ; i++)
```

```
   if(T[i].obtenerRubro() == r){
     v = T[i].obtenerValor();
     T[i].establecerValor(v*(1-p));}
}
//Consultas
public int  cantArt () {
     return cant;}
public boolean estaLleno() {
     return cant == T.length;}
public Articulo recuperar (int c){
//Retorna el artículo con el código dado
  boolean esta = false; int i;
  for (i = 0; !esta && i <  cantArt() ; i++){
     esta = T[i].obtenerCodigo() == c; }
  if (esta) return T[i];
  else return null;}
public boolean pertenece (int c){
/*Retorna verdadero si un elemento de la Coleccion tiene el código c*/
  boolean esta = false;
  for (int i = 0; !esta && i <  cantArt() ; i++)
     esta = T[i].obtenerCodigo() == c;
  return esta;}
public boolean pertenece (Articulo c){
/*Decide si algún elemento de la colección es equivalente a c*/
  boolean esta = false;
  for (int i = 0; !esta && i <  cantArt() ; i++)
     esta = T[i].equals(c);
  return esta;}
public Inventario unAnio(int a){
/*genera un objeto de clase Inventario solo con los objetos que
corresponden al año a. */
Inventario n = new Inventario(cantArt());
for (int i = 0; i <  cantArt() ; i++)
   if(T[i].obtenerAnio() == a){
     n.alta(T[i]);}}
return n;
}
```

Observemos que algunos servicios de `Inventario` tienen la misma funcionalidad y responsabilidades que los de `Correo`, aunque cambian los nombres. Ambas clases modelan a una colección de elementos homogéneos y brindan un comando para agregar un elemento o eliminar un elemento, aunque el comando para agregar tiene diferente responsabilidad en cada caso. Los servicios que realizan alguna forma de procesamiento son diferentes, porque dependen de la aplicación. Por supuesto podemos definir otras clases que compartan la funcionalidad de algunos servicios.

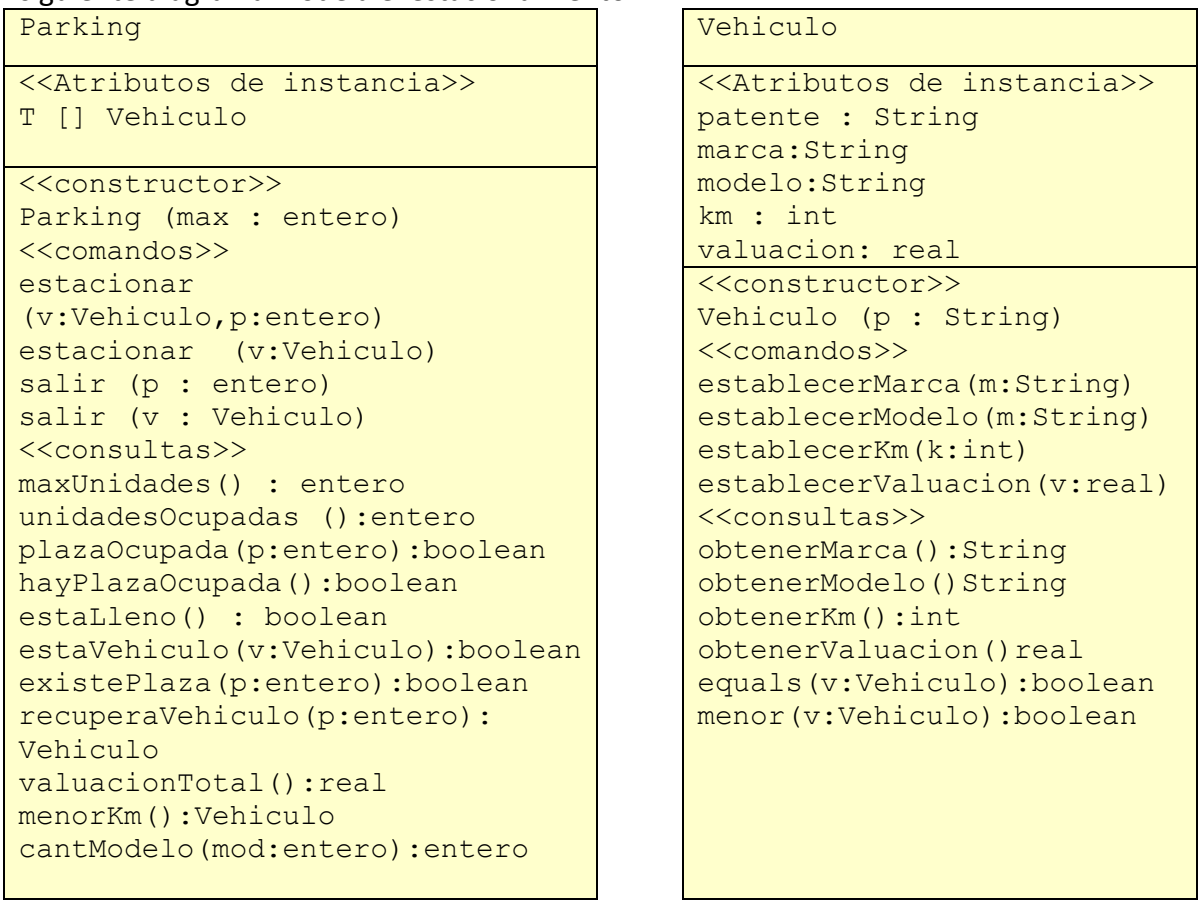
El siguiente caso de estudio define una clase que modela una tabla de elementos homogéneos. Mientras que las dos colecciones propuestas se caracterizan porque las componentes están comprimidas, utilizamos el nombre tabla para una estructura en la cual las componentes nulas y ligada pueden estar intercaladas.

Caso de Estudio Parking

Una concesionaria dispone de un estacionamiento tiene un conjunto de unidades. En un momento dado una plaza puede estar ocupada por un vehículo o libre. Las unidades libres y ocupadas están intercaladas.

El estacionamiento puede ser modelado por un arreglo en la cual el subíndice indica el número de plaza. Cuando llega un vehículo se inserta un vehículo en la tabla. La eliminación puede hacerse a partir del vehículo o del número de plaza.

El siguiente diagrama modela el estacionamiento:



`estacionar(unVeh:Vehiculo, p:entero)` asigna el vehículo `unVeh` a la plaza `p`, que requiere libre. Requiere además `0<=p<maxUnidades()`.

`salir(unVeh:Vehiculo)` busca un vehículo equivalente a `unVeh` y si está libera la plaza en la que está dicho vehículo

`salir(p: entero)` libera la plaza, requiere `0<=p<maxUnidades()`

`maxUnidades()`: retorna la cantidad de unidades del estacionamiento

`unidadesOcupadas()`: retorna la cantidad de unidades ocupadas

`plazaOcupada(int p)`: retorna true si la plaza p está ocupada, requiere `0<=p<maxUnidades()`

`hayPlazaOcupada()`: retorna true si hay al menos un vehículo en el estacionamiento

`estaLleno()`: retorna true si el estacionamiento no tiene unidades libres

`estaVehiculo(Vehiculo unVeh):` retorna true si una plaza del estacionamiento mantiene un vehículo equivalente a unVeh

`existePlaza (int p):` retorna true si `0<=p<maxUnidades()`

`recuperarVehiculo(int p):` retorna el vehículo de la plaza p, requiere `0<=p<maxUnidades()`

`valuacionTotal():` retorna la valuación total de todos los vehículos

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

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

Observemos que si la clase cliente envía el mensaje estacionar a un objeto de clase Parking y no se cumple `0<=p<maxUnidades()`, se produce un error de ejecución. Si en cambio `p` es válido pero no corresponde a una plaza libre, se produce un error de aplicación.

Las clases `Parking` y `Vehiculo` están asociadas. La clase `Parking` encapsula un arreglo de elementos de clase `Vehiculo`. La implementación en Java de `Parking` es:

```
class  Parking{
private Vehiculo[] T;
//Constructor
public Parking(int max){
   T = new Vehiculo[max];}
//Comandos
public void estacionar(Vehiculo unVeh, int p) {
/*Asigna el vehiculo a la plaza, requiere que la plaza exista*/
    T[p] = unVeh;}
public void salir(Vehiculo unVeh){
/*Busca en el estacionamiento un vehículo equivalente a unVeh y libera
la plaza*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)
     esta = unVeh.equals(T[i]) ;
   else
     i++;}
 if (esta)
   T[i] = null;}
public void salir(int p){
/* Elimina el elemento de la plaza p, requiere que p sea un plaza
válida*/
    T[p] = null;}
//Consultas
public int maxUnidades(){
  return T.length;}
public int unidadesOcupadas(){
/*Retorna la cantidad de unidades en las que hay un vehículo
asignado*/
  int cant=0;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null)
      cant++;
  return cant;}
public boolean plazaOcupada(int p){
//Requiere p válida
```

```
   return T[p] != null;}
public boolean hayPlazaOcupada(){
int i = 0; boolean hay=false;
while (i < T.length && !hay){
   if (T[i] != null)
     hay = true ;
   i++;}
return hay;}
public boolean estaLleno(){
int i = 0; boolean hay=false;
while (i < T.length && !hay){
   if (T[i] == null)
     hay = true ;
   i++;}
return !hay;}
public boolean estaVehiculo(Vehiculo unVeh){
/*Decide si una plaza mantiene un vehículo equivalente a unVeh que
asume ligado*/
int i = 0; boolean esta=false;
while (i < T.length && !esta){
   if (T[i] != null)
     esta = unVeh.equals(T[i]) ;
   i++;}
return esta;}
public boolean existePlaza (int p) {
    return p >= 0 && p < T.length;}
public Vehiculo recuperarVehiculo(int p) {
// Requiere que p sea una posición válida
    return T[p];}
public Vehiculo menorKm() {
/*Asume que no hay vehículos con más de 500000 km
Puede retornar nulo si la tabla está vacía*/
  Vehiculo v=null; int menor = 500000;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null &&  T[i].obtenerKm() < menor){
        v = T[i]; menor = v.obtenerKm();}
  return v;}
public int cantModelo(int mod) {
  int cant=0;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null &&  T[i].obtenerModelo() == mod)
          cant++;
  return cant;}
public float valuacionTotal() {
  float v=0;
  for (int i=0; i<=T.length;i++)
    if (T[i] != null)
          v = v+T[i].obtenerValuacion();
  return v;}}
```

Observemos que los operandos no son conmutativos en la expresión:

```
(T[i] != null &&  T[i].obtenerModelo() = mod)
```

Y es necesario utilizar el operador con cortocircuito. Es decir, las siguientes expresiones computan error en ejecución si `T[i] == null`:

```
(T[i] != null &  T[i].obtenerModelo() = mod)
```

```
(T[i].obtenerModelo() = mod && T[i] != null)
```

Notemos también que no se ha asignado a la clase `Parking` ni a sus clientes la responsabilidad de controlar que la unidad está disponible antes de que se ejecute el comando `estacionar`. Si la componente está ligada, la nueva referencia se sobrescribe sobre la anterior.

La clase `Parking` utiliza un arreglo parcialmente ocupado para representar una tabla de elementos cada uno de los cuales ocupa una posición particular. Podemos definir una clase similar para modelar el conjunto de casilleros asignados a los socios de un club, la asignación de mozos a las mesas de un bar o cualquier otro problema en el cual cada elemento se asigna a una posición específica.

En algunos de estos problemas cada elemento puede estar asignado a una única posición. Por ejemplo, un vehículo solo estará asignado a una plaza del estacionamiento. En otros casos un mismo elemento puede estar ligado a varias posiciones de la tabla. Por ejemplo, un mismo mozo puede estar asignado a n mesas.

El caso de estudio que sigue se modela mediante una colección, pero los elementos están ordenados de acuerdo a un atributo.

Caso de Estudio Libreta de Contactos

Una libreta de contactos mantiene el nombre, número de teléfono móvil, número de teléfono fijo y email de un conjunto de personas u organizaciones.

La clase `Libreta_Contactos` encapsula una colección de elementos de clase Contacto, representada con un arreglo parcialmente ocupado. Los elementos se mantienen ordenados alfabéticamente por nombre y están comprimidos de modo que todas las posiciones libres están al final.

Libreta_Contactos
T [] Contacto cant:entero
<< Constructores>> LibretaContactos(max:entero) <<Comandos>> insertar(con:Contacto) eliminar(con:Contacto) <<Consultas>> cantContactos():entero estaLleno():entero pertenece(c:Contacto):boolean

Contacto
nombre:String nroMovil: String nroFijo:String email:String
<<Constructor>> Contacto (n:String) <<Comandos>> <<Consultas>> igualNombre(c: Contacto):boolean mayorNombre(c: Contacto):boolean

El comando insertar requiere que la clase cliente controle que la estructura no esté llena y no exista un contacto con el mismo nombre que el parámetro `con`. Si la libreta de contactos se mantiene ordenada por nombre, el servicio `insertar` no puede implementarse asignando el nuevo contacto a la primera posición libre. Para comprender cómo implementar el servicio insertar comencemos visualizando la libreta de contactos a través de una grilla con capacidad para 6 contactos:

Nombre	Número de Móvil	Número Fijo	Email

Consideremos que el primer contacto que se inserta corresponde a Davini Laura. Notemos que no interesan los otros atributos porque la libreta se ordena por nombre. Como la grilla está vacía el primer contacto ocupa la primera posición en la grilla:

Nombre	Número de Móvil	Número Fijo	email
Davini Laura	…	…	…

Si luego vamos a insertar Castro Ramón, para que la Libreta quede ordenada, debemos *arrastrar* a Davini Laura desde la posición 0 a la posición 1 y luego asignar el nuevo contacto a la posición 0:

Nombre	Número de Móvil	Número Fijo	email
Castro Ramón	…	…	…
Davini Laura	…	…	…

Si luego insertamos a Álvarez María, tenemos que arrastrar a los dos contactos anteriores una posición y luego asignar el nuevo contacto a 0:

Nombre	Número de Móvil	Número Fijo	email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Davini Laura	…	…	…

Si luego insertamos a Funez María, como no hay ningún contacto mayor al nuevo, insertamos directamente en la última posición:

Nombre	Número de Móvil	Número Fijo	email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Davini Laura	…	…	…
Funez María	…	…	…

Si luego insertamos un nuevo contacto con nombre Cepeda Pedro, arrastramos una posición a los dos contactos con nombres mayores al nuevo:

Nombre	Número de Móvil	Número Fijo	email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Cepeda Pedro	…	…	…
Davini Laura	…	…	…
Funez María	…	…	…

La grilla nos permite visualizar la libreta de contactos de manera abstracta para mostrar cómo se inserta cada nuevo contacto. El diagrama de objetos, que también es una abstracción, es útil para graficar cómo se administra la memoria, pero no es una buena herramienta para diseñar el algoritmo insertar. El algoritmo para el servicio insertar puede ser entonces:

```
Algoritmo insertar
DE nuevo
   Buscar la posición del primer elemento mayor al nuevo
   Si existe
      Arrastrar todos los elementos desde esa posición hasta la última
   Asignar nuevo contacto a la posición encontrada
   Incrementar la cantidad de contactos
```

Si el nuevo elemento es mayor a todos, en particular si el nuevo elemento es el primero que se inserta, la posición encontrada es la primera libre.

El código del servicio `eliminar` también tiene que ser diseñado convenientemente, porque los elementos tienen que quedar ordenados y comprimidos. Si vamos a eliminar el contacto Cepeda Pedro, arrastramos una posición a los dos contactos con nombres mayores. En este caso la cantidad de contactos se decrementa en 1.

Nombre	Número de Móvil	Número Fijo	Email
Alvarez María	…	…	…
Castro Ramón	…	…	…
Davini Laura	…	…	…
Funez María	…	…	…

El algoritmo para el servicio eliminar puede ser entonces:

```
Algoritmo eliminar
DE contacto
   Buscar la posición del contacto
   Si existe
      Arrastrar todos los elementos desde la última posición hasta la
      encontrada
     Decrementar la cantidad de contactos
```

Si el contacto no existe el comando eliminar no modifica a la estructura.

```
class Libreta_Contactos{
/*Mantiene una colección de contactos ordenada por nombre. */
//Atributos de instancia
private Contacto [] T;
private int cant;
//Constructor
public Libreta_Contactos(int max){
    T = new Contacto [max];}
//Comandos
public void insertar (Contacto nuevo){
//Requiere que la colección no esté llena
    int pos = posInsercion(nuevo,cant);
    arrastrarDsp (pos,cant-pos);
    T[pos] = nuevo;
    cant++; }
private void arrastrarDsp (int pos,int n){
    if (n > 0){
       T[pos+n] = T[pos+n-1];
       arrastrarDsp(pos,--n);       } }
public void eliminar(Contacto con){
/*Busca la posición de un contacto con el mismo nombre que con, si
existe lo elimina arrastrando los que le siguen una posición  */
    int pos = posElemento (con,cant);
    if (pos < cant){
      arrastrarAnt(pos,cant-pos-1);
      cant--;} }
private void arrastrarAnt(int pos,int n){
 if (n > 0){
       T[pos] = T[pos+1];
       arrastrarAnt(++pos,--n);        }}
//Consultas
public int cantContactos (){
  return cant;}
```

```
public boolean estaLleno (){
  return cant == T.length;}
public boolean pertenece(Contacto con){
return posElemento(con,cant) < cant;}
private int  posInsercion (Contacto con,int n){
/* Retornar la posición del primer elemento mayor a con, o 0
Caso trivial: Si la colección está vacía la posición de inserción es 0
Caso trivial: Si el nombre del último contacto de la colección es
menor al buscado, la posición de inserción es n
Caso Recursivo: Si el nombre del último contacto de la colección es
mayor al buscado, busca la posición entre los primeros n-1 contactos
de la colección*/
    int pos = 0;
    if (n > 0)
      if (con.mayorNombre(T[n-1].obtenerNombre()) ||
          con.igualNombre(T[n-1].obtenerNombre()))
        pos = n;
      else
        if (T[n-1].mayorNombre(con.obtenerNombre()))
          pos  = posInsercion (con,--n);
    return pos;}
private int  posElemento(Contacto con,int n){
/* retorna la posición del elemento, o cant
Caso trivial: Si la colección está vacía el elemento no está y retorna
cant
Caso trivial: Si el último elemento de la colección es menor al
buscado, el elemento no está y retorna cant
Caso trivial: Si el último elemento de la colección es el buscado, la
posición es n- 1
Caso Recursivo: Si el último elemento de la colección es mayor al
buscado, busca la posición en los primeros n-1 elementos de la
colección*/
    int pos=cant;
    if (n > 0)
      if (T[n-1].igualNombre(con.obtenerNombre()))
         pos = n-1;
      else
        if (T[n-1].mayorNombre(con.obtenerNombre()))
          pos  = posElemento (con,--n);
    return pos;}
public void test(){
  for (int i=0;i<cant;i++)
    System.out.println(T[i].obtenerNombre());}
}
```

El arreglo mantiene una colección homogénea de elementos. La colección se crea con capacidad para mantener `max` elementos pero inicialmente está vacía. En la colección se puede insertar un elemento en forma ordenada, eliminar un elemento manteniendo el orden y decidir si un elemento pertenece a la colección.

Mantener la estructura ordenada tiene un *costo* en ejecución, cada inserción requiere buscar la posición y arrastrar, en promedio, la mitad de los elementos. Como contrapartida las búsquedas son más eficientes, ya que si se busca un elemento y este no pertenece a la colección, no es necesario recorrerla todas. Si las búsquedas son más frecuentes que las inserciones, mantener la estructura ordenada puede ser una decisión acertada.

La eficiencia del algoritmo insertar puede mejorar si al mismo tiempo se busca la posición de inserción y se arrastran los elementos. Esta estrategia es posible dado que la clase cliente controla que hay al menos una posición libre y no existe un elemento con el mismo nombre que el nombre del contacto que se agrega.

```
Algoritmo insertar
DE nuevo
   Buscar desde la última posición cant-1 hasta 0, la posición pos del
   primer elemento menor al nuevo y arrastrar cada elemento de la
   posición i a la posición i+1
   Si existe Asignar nuevo contacto en pos+1
   Sino Asignar nuevo contacto en cant
   Incrementar cant
```

Si la responsabilidad de controlar que no haya dos contactos con el mismo nombre recae en la clase cliente, es necesario recorrer dos veces la estructura, aunque sea parcialmente, para insertar un nuevo elemento. La primera cuando la clase cliente envía el mensaje `pertenece`, la segunda cuando el comando `insertar` busca la posición de inserción. Una alternativa más eficiente es que la clase `Libreta_Contactos` asuma la responsabilidad de controlar que no haya repetidos y el comando `insertar` busque el contacto, si existe no lo inserta, si no existe, encontró un contacto con nombre mayor al parámetro y esa es la posición de inserción.

Ejercicios

- *Escriba una clase tester que verifique los servicios insertar y eliminar para una colección de 10 elementos y considerando casos de prueba significativos.*
- *Modifique la implementación de los métodos posInsertar y posElemento proponiendo soluciones iterativas*
- *Agregue un método intercalar (nueva:Libreta_Contactos) a la clase Libreta_Contactos que modifique el estado interno de la Libreta de contactos que recibe el mensaje intercalando ordenadamente los contactos de nueva. Puede utilizar un objeto auxiliar.*
- *Agregue un atributo de clase String llamado ciudad a la clase Contacto y analice cómo afecta el cambio a la clase Libreta_Contactos*
- *Agregue un método contactosCiudad (c:String):entero a la clase Libreta_Contactos que compute la cantidad de contactos de la ciudad c.*
- *Implemente el método insertar considerando que la responsabilidad de controlar la pertenencia es de la clase Libreta_Contactos.*

Cuando una colección de elementos está ordenada de acuerdo a un atributo y el necesario decidir si un elemento en particular pertenece a la colección, es posible aplicar una estrategia conocida como **búsqueda binaria**. La estrategia consiste en partir la estructura en mitades, considerando que el **elemento buscado** puede ser:

- Igual al que está en el medio
- Menor que el que está en el medio
- Mayor que el que está en el medio

De modo que podemos definir un algoritmo:

```
Algoritmo Búsqueda Binaria
```

```
DE elem
  si el elemento que está en el medio es el buscado
    EXISTE
  si hay un solo elemento y no es el buscado
     NO EXISTE
  sino
    si el elemento que está en el medio es menor al buscado
      Descartar la primera  mitad
      Buscar en la segunda mitad
    sino
      Descartar la segunda mitad
      Buscar en la primera mitad
```

Para nuestro caso de estudio específico podemos refinar el algoritmo:

```
Algoritmo Búsqueda Binaria
DE contacto
  si el contacto que está en el medio de la colección
     es el buscado
    EXISTE
  si hay un solo contacto y no es el buscado
     NO EXISTE
  sino
    si el contacto que está en el medio es menor al buscado
      Descartar la primera  mitad de la Libreta de contactos
      Buscar en la segunda mitad de la Libreta de contactos
    sino
      Descartar la segunda mitad de la Libreta de contactos
      Buscar en la primera mitad de la Libreta de contactos
```

El algoritmo puede refinarse todavía más considerando que los datos están en un arreglo:

```
Algoritmo BúsquedaBinaria
DE ini,fin,contacto
Mitad←(ini+fin)/2
si T_mitad = contacto
  EXISTE
sino
  si ini >= fin
     NO EXISTE
  sino
    si T_mitad < contacto
       BuscarBinaria mitad+1,fin,contacto
   sino
      BusquedaBinaria ini,mitad-1,contacto
```

Aunque esta versión sigue siendo abstracta, la traducción a un lenguaje de programación es más directa que en el caso anterior.

Ejercicio

- *Aplicar la estrategia descripta por el algoritmo a una grilla con 10 contactos considerando los siguientes casos: se busca el primer elemento, un elemento menor al primer elemento, el último elemento, un elemento mayor al último, un elemento que está en la estructura*

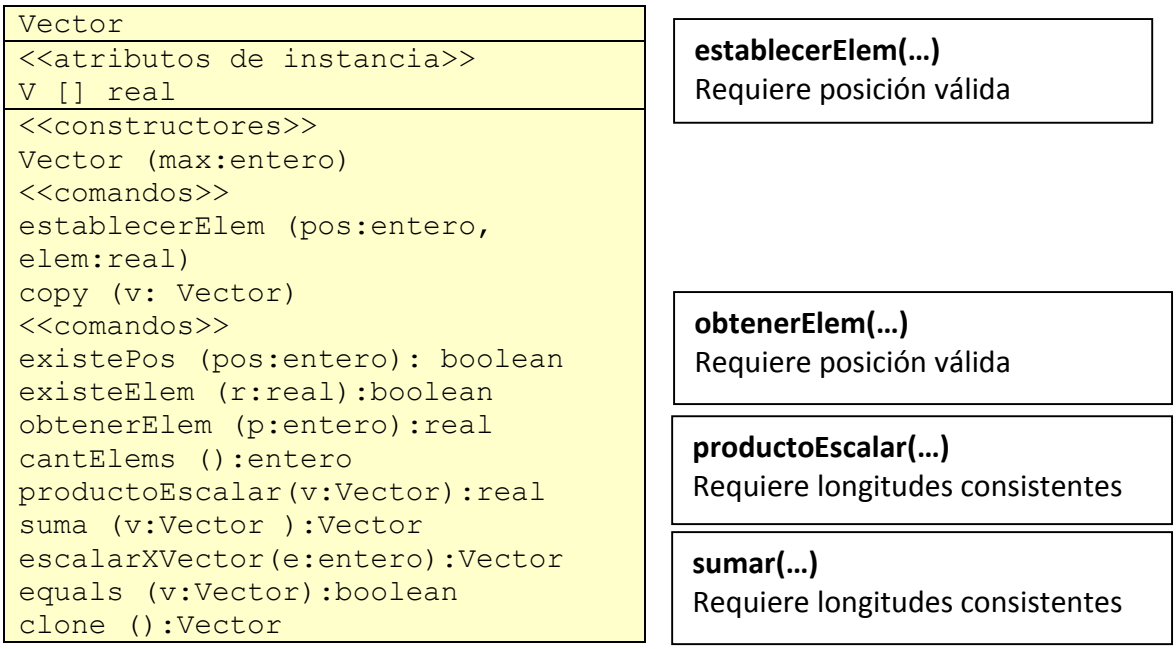
mayor al primero y menor al último, un elemento que NO está en la estructura y es mayor al primero y menor al último.

- *Implementar el método pertenece aplicando la estrategia de búsqueda binaria.*

Observemos que la recursividad permite modelar naturalmente esta estrategia de búsqueda, porque está especificada en forma recursiva.

Problemas Propuestos

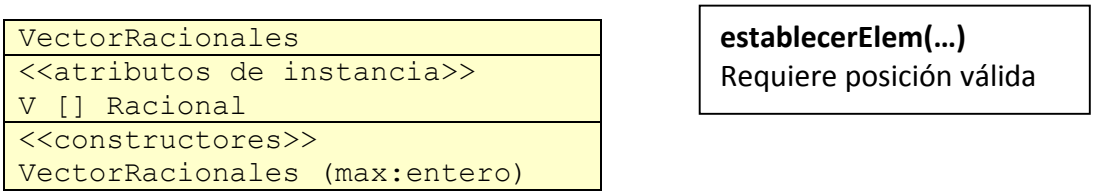
1. *Implemente el TDA Racional modelado en el siguiente diagrama:*
2. *Dado el siguiente diagrama que modela un TDA Vector*



a. *Implemente la clase Vector considerando que el cliente asume las posiciones dentro del vector de 1 a la cantidad de elementos del vector.*
b. *Implemente un Tester para la case vector, definiendo adecuadamente los casos de prueba.*
c. *Dibuje el diagrama de objetos para el siguiente segmento de instrucciones:*

```
Vector v ;
v= new Vector (7);
```

3. *Dado el siguiente diagrama que modela un TDA VectorRacionales*



```
<<comandos>>
establecerElem (pos:entero,
elem:Racional)
copy (v: Vector)
<<comandos>>
existePos (p:entero): boolean
existeElem (r:Racional):boolean
obtenerElem
(pos:entero):Racional
cantElems ():entero
productoEscalar(v:Vector):Racion
al
suma (v:Vector ):Vector
escalarXVector(e:entero):Vector
equals (v:Vector):boolean
clone ():Vector
```

obtenerElem(…)
Requiere posición válida

a. *Implemente la clase VectorRacionales en el diagrama considerando que **el cliente asume las posiciones dentro del vector de 1 a la cantidad de elementos del vector**. Implemente equals, copy y clone en profundidad.*

b. *Implemente un Tester para la case VectorRacionales, definiendo adecuadamente los casos de prueba.*

c. *Dibuje el diagrama de objetos para el siguiente segmento de instrucciones:*

```
Vector v ;
v= new Vector (7);
```

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

R101
<<atributos de clase>> maxCafé : entero maxCacao : entero maxCrema : entero <<atributos de instancia>> nroSerie:entero ultMnt:MesAnio cantCafé : entero cantCacao : entero cantCrema : entero
<<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

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.

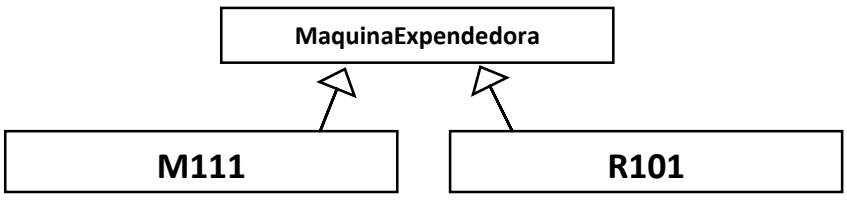
La **herencia** es un mecanismo que permite organizar la colección de clases de un sistema, estableciendo una relación de **generalización-especialización**.

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;}
```

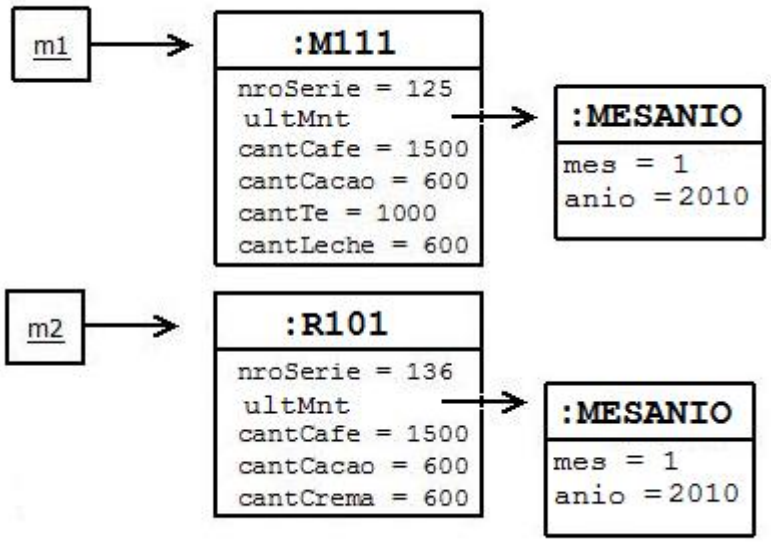
Si se invoca al constructor de una clase base mediante `super`, **siempre tiene que ser en la primera línea** de la definición del constructor de la clase derivada**.**

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.

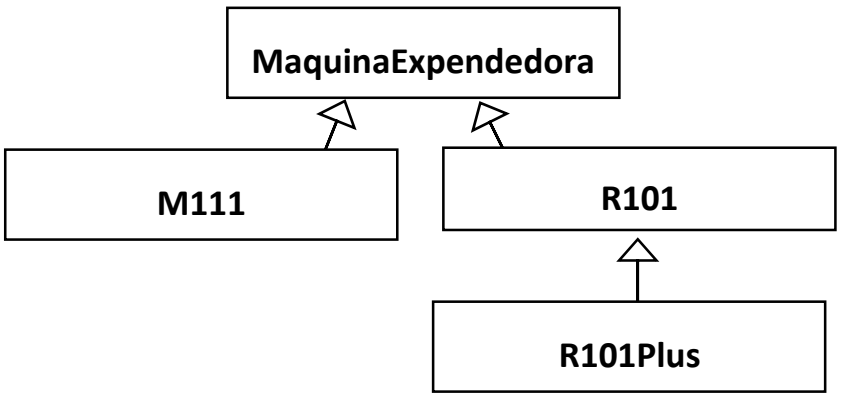
Herencia y extensibilidad

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.

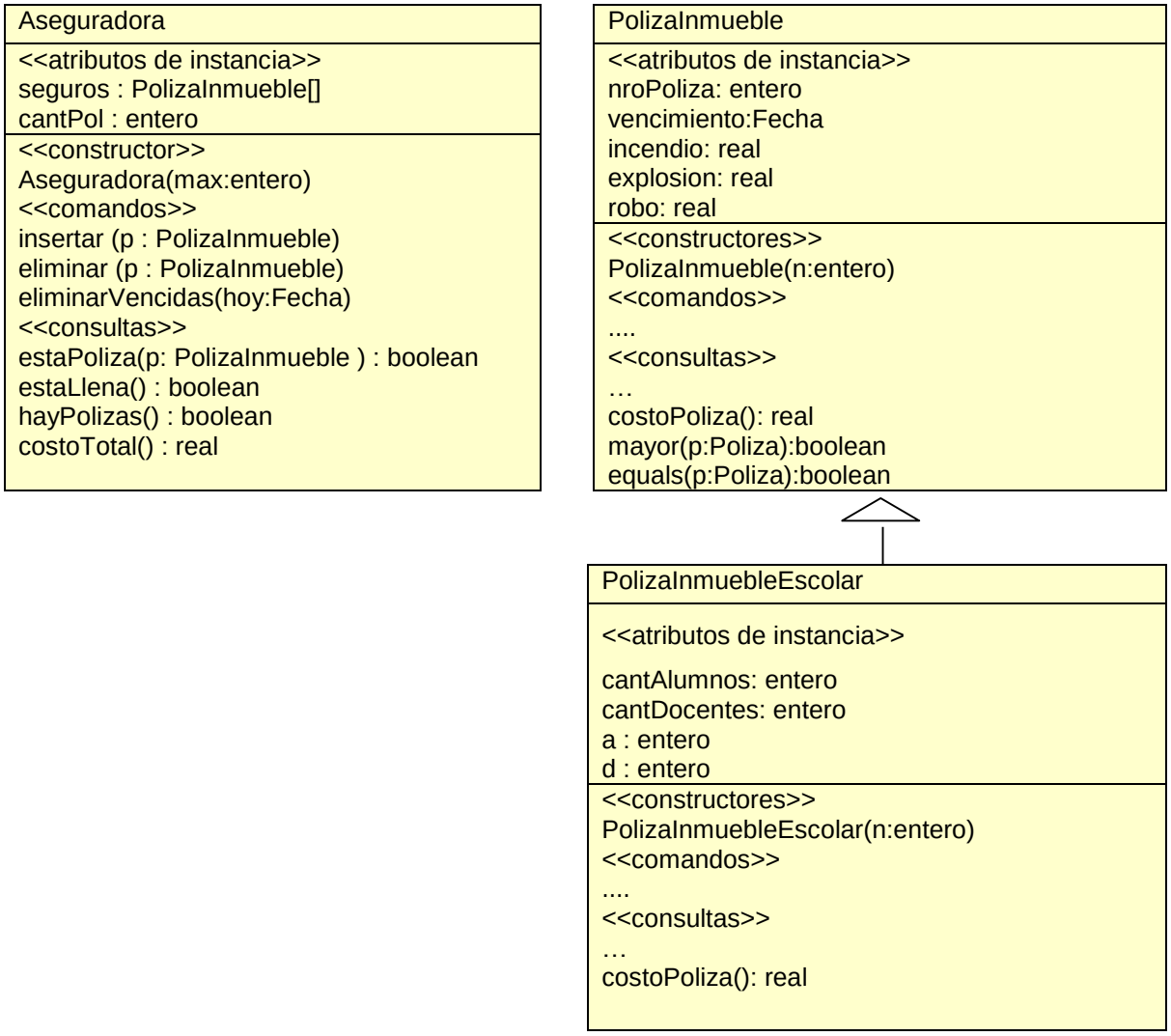
Problemas Propuestos

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.

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:

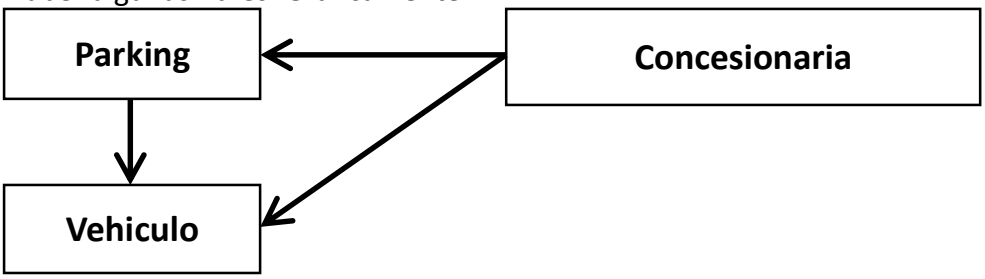
```
class Simulador{
…
Racional r1,r2,r3,r4;
…
Fila f = new Fila (4);
f.insertar(r1);
f.insertar(r2);
f.insertar(r3);
f.insertar(r4);
if (f.esCreciente())
 …
}
```

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`.

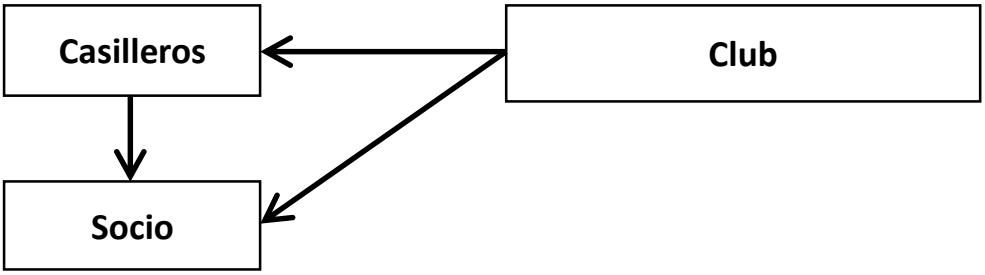
Generalización y Reuso

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.

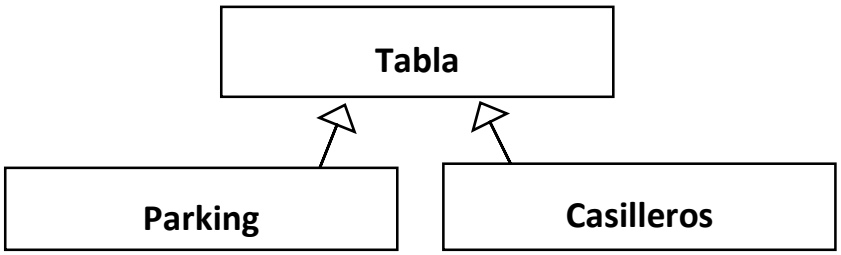
Algunos de los servicios provistos por la clase `Parking` son específicos de la aplicación:

- 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.*

Interfaces Gráficas de Usuario

La programación orientada a objetos favorece la construcción y mantenimiento de sistemas de software complejos desarrollados para ambientes dinámicos. El encapsulamiento, la herencia y el polimorfismo permiten producir componentes reusables y fáciles de extender, como las que se requieren cuando se desarrolla una **interfaz gráfica de usuario**.

Una **interfaz** es un medio que permite que dos entidades se comuniquen. En una **interfaz de usuario** una de las entidades es una **persona** y la otra es un **sistema** que el usuario intenta controlar. El sistema puede ser una herramienta, un automóvil o cualquier dispositivo electrónico, en particular una computadora.

En un automóvil el conductor visualiza el estado a través del velocímetro o indicadores de combustible y controla el vehículo a través del embriage, el acelerador, la palanca de cambios, el volante, etc. En una computadora el usuario ejecuta acciones como oprimir teclas o el botón del ratón y estas acciones son percibidas por la interfaz del sistema.

Las **interfaces gráficas de usuario (GUI)** explotan la capacidad de las computadoras para reproducir imágenes y gráficos en pantalla y brindan un ambiente amigable, simple e intuitivo. Como su nombre lo indica, los tres elementos que definen a un GUI son:

- **Interfaz**: medio de comunicación entre entidades
- **Gráfica**: incluye ventanas, menúes, botones, texto, imágenes, etc.
- **Usuario**: persona que usa la interfaz para controlar un sistema

Las GUI reemplazaron a las interfaces de **líneas de comandos** en la cuales el usuario interactuaba con el sistema a través de una consola. En una GUI la interacción se realiza a través de:

- Iconos
- Ventanas
- Menúes desplegables
- Hipertexto
- Manipulación basada en *arrastrar y soltar*

Un **ícono** es un pictograma que representa a una entidad como un archivo, una carpeta, una acción o una aplicación.

Una **ventana** es una porción de la pantalla que sirve como una pantalla más pequeña. Un **menú** es una lista de opciones alternativas ofrecidas al usuario. Un menú desplegable es aquel que “cuelga” de una opción de otro menú.

Un **hipertexto** es un texto organizado mediante una red. Si el texto se organiza en red y además incluye imágenes, sonido, animación, se llama **hipermedio**.

La **manipulación basada en arrastrar y soltar** presupone el uso del ratón como dispositivo de entrada asociado a un cursor.

Todos estos elementos fueron determinantes para el desarrollo de GUI, aunque en el momento que fueron concebidos tuvieron poco éxito comercial porque la tecnología resultaba insuficiente para lograr implementaciones eficientes.

Una GUI se construye a partir de una **colección de componentes** con una **representación gráfica**. Por ejemplo, el botón para cerrar una ventana, la barra de desplazamiento de una ventana y la ventana misma.

Algunas componentes tienen la capacidad para **percibir eventos** generados por las acciones del usuario. Un usuario realiza una **acción** que genera un **evento** que es detectado por una **componente** lo cual provoca una **reacción**. La creación de componentes reactivas requiere que el lenguaje brinde algún mecanismo para el **manejo de eventos**. El **flujo de ejecución** va a quedar determinado por los eventos que generan los usuarios sobre las componentes de la interfaz.

La construcción de GUI en Java requiere integrar los conceptos centrales de la POO: **herencia, polimorfismo, ligadura dinámica y encapsulamiento**.

Objetos gráficos y Clases gráficas

Las componentes son las partes individuales a partir de las cuales se conforma una interfaz gráfica. Cada componente de una GUI está asociada a un **objeto gráfico**. Un objeto gráfico es una instancia de una **clase gráfica**. Algunos de los atributos de un objeto gráfico son **atributos gráficos** y parte del comportamiento ofrece **servicios gráficos**.

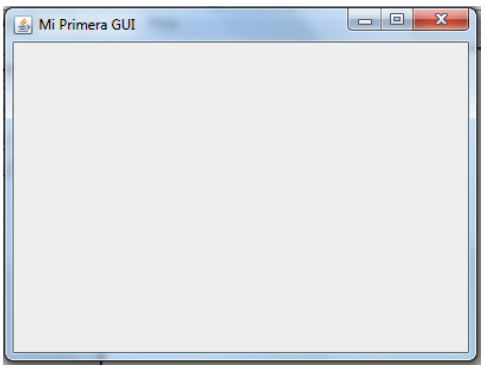
Algunos componentes son **contenedores** de otros componentes. Un contenedor tiene atributos especiales como por ejemplo el **diagramado** de acuerdo al cual se organizan las componentes contenidas.

Una **ventana** es un contenedor de alto nivel. Un **frame** es un tipo especial de ventana sobre el que puede ejecutarse una aplicación. Un frame tiene atributos gráficos como tamaño, barra de título, marco, panel de contenido y tres botones. Cada una de estas componentes tiene también sus atributos gráficos.

La ejecución del siguiente segmento de código:

```
import javax.swing.*;
class PrimerEjemplo {
 public static void main(String args[ ]) {
   JFrame f = new JFrame("Mi Primera GUI");
   f.setSize(400, 300) ;
   f.setVisible(true) ;     }}
```

Crea un frame como el que sigue:



La creación de un objeto de la clase `JFrame` requiere importar el paquete Swing. La instrucción:

```
   JFrame f = new JFrame("Mi Primera GUI");
```

Crea un objeto de la clase `JFrame` usando el constructor que recibe una cadena de caracteres con la que se inicializa el título de la barra de título. La clase `JFrame` brinda servicios para modificar los atributos gráficos. La variable `f` mantiene una referencia a un objeto de clase `JFrame` y puede recibir cualquiera de los mensajes provistos en esa clase o en sus clases ancestro. Las instrucciones:

```
   f.setSize(400, 300) ;
   f.setVisible(true) ;
```

Envían mensajes al frame para establecer su tamaño y hacerlo visible. Un efecto equivalente para el usuario, se consigue definiendo una clase que extiende a `JFrame` y creando un objeto de esta clase derivada.

```
import java.awt.*;
import javax.swing.*;
class MiVentana extends JFrame{
    public MiVentana() {
      super("Mi ventana");
      setSize(400, 300);
      getContentPane().setBackground(Color.BLUE);
      setDefaultCloseOperation(EXIT_ON_CLOSE);}}
```

La instrucción `super("Mi ventana")` invoca al constructor provisto por `JFrame` con la cadena que aparecerá en la barra de título como parámetro.

La instrucción `setSize(400, 300)` establece el ancho y la altura del frame en pixels.

El mensaje `getContentPane()` obtiene el panel de contenido del frame, el objeto que retorna es de clase `Container`. Este objeto recibe a continuación el mensaje `setBackground(Color.BLUE)` cuyo efecto es establecer el color del fondo.

El mensaje `setDefaultCloseOperation` se va a ligar a un método provisto por `JFrame`. El parámetro indica qué hacer cuando la ventana se cierra, en este caso terminar la aplicación.

El constructor envía tres mensajes al mismo objeto que se está creando, la instancia de `MiVentana`, que es también instancia de `JFrame`.

El método `main` crea ahora una instancia de `MiVentana`:

```
import javax.swing.*;
class SegundoEjemplo {
  public static void main(String args[ ])    {
    MiVentana f= new MiVentana();
    f.setVisible(true);    }
}
```

Como antes, la creación del objeto ejecuta el constructor y activa el frame. El objeto ligado a `f` puede recibir cualquiera de los mensajes provistos por la clase `JFrame`. La barra de título y los botones son componentes **reactivas**, reaccionan con un click del mouse.

Construcción de una GUI

La construcción de una GUI requiere:

- **Diseñar** la interfaz de acuerdo a las especificaciones
- **Implementar** la interfaz usando las facilidades provistas por el lenguaje

El **diseño** de una interfaz gráfica abarca tres aspectos:

- Definir las componentes
- Organizar las componentes estableciendo el diagramado de las componentes contenedoras
- Decidir cómo reacciona cada componente ante las acciones realizadas por el usuario

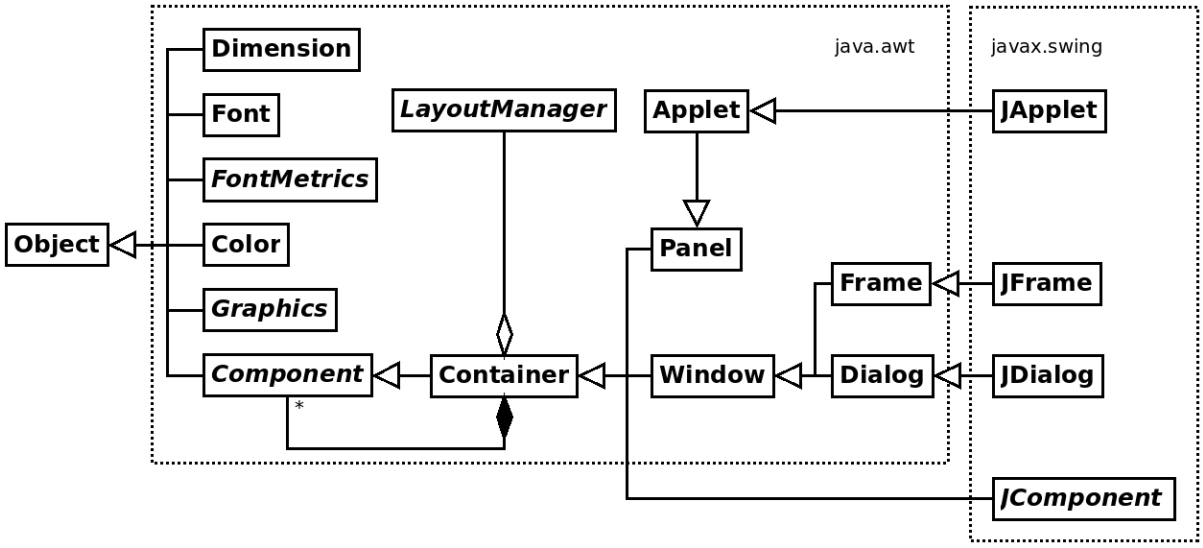
La **implementación** de una interfaz gráfica consiste en:

- **Crear** un objeto gráfico para cada componente de la GUI e **insertarlo** en otras componentes contenedoras
- **Definir el comportamiento** de las componentes reactivas en respuesta a las acciones del usuario

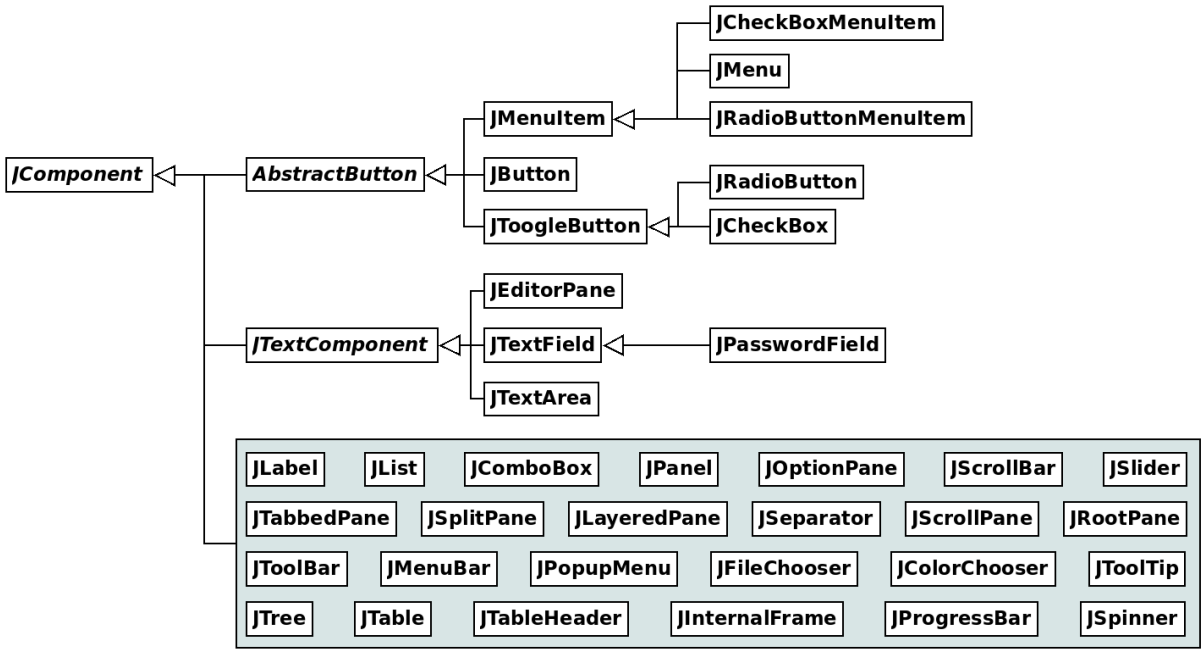
El diseño es una etapa fundamental, pero el objetivo en este libro no es diseñar interfaces gráficas, sino implementar el diseño dado para una GUI.

Tipos de Componentes

Las **componentes** de las GUI que se desarrollan en los ejemplos que siguen, son instancias de alguna de las clases provistas por los paquetes gráficos AWT (Abstract Window Toolkit) o Swing o de una clase derivada de ellas. AWT y Swing son paquetes gráficos independientes de la plataforma, que permiten desarrollar interfaces gráficas. Swing no reemplaza a AWT sino que lo usa y agrega nuevas clases, como muestra la figura.



Ambos paquetes brindan una colección de clases que pueden usarse para crear distintos **tipos de componentes**, como por ejemplo, botones, cajas de texto, menúes, etc. La jerarquía de clases que modela los diferentes tipos de componentes es:



Cada clase provista por AWT o Swing puede ser usada para:

- Crear objetos gráficos asociados a las componentes de la interfaz
- Definir clases más específicas a partir de las cuales se crearán componentes

Cada componente tiene una apariencia estándar pero también puede configurarse de acuerdo a las pautas de diseño que se adopten. El tamaño de cada componente se mide en pixels. Un **pixel** es la unidad de espacio más pequeña que puede mostrarse en pantalla. La **resolución** de una pantalla se mide de acuerdo a la cantidad de pixels que puede mostrar. Cuanto mayor sea la cantidad de pixels en la ventana, mayor será la resolución.

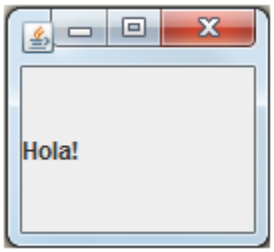
En los ejemplos que siguen se construyen GUI definiendo clases que extienden a `JFrame`, las cuales incluyen como atributos de instancia a objetos gráficos de alguna clase derivada de la clase `JComponent`.

Etiquetas

Una **etiqueta** es un objeto gráfico **pasivo** que permite mostrar un texto o una imagen. En Java se puede crear una etiqueta definiendo un objeto de clase `JLabel`. Los constructores provistos por la clase permiten establecer texto, imagen y/o alineación horizontal de la imagen:

```
JLabel()
JLabel (Icon image)
JLabel (Icon image, int horAlig)
JLabel (String text)
JLabel (String text, Icon icon, int horAlig)
```

Por ejemplo, la siguiente GUI:



Se activa al crear un objeto de clase `FrameConEtiqueta` definida a continuación:

```
import javax.swing.*;
class FrameConEtiqueta extends JFrame{
//Objeto gráfico
    JLabel etiqueta;
//Constructor
    public FrameConEtiqueta (String tit) {
      super(tit);
      setSize(100, 120);
      etiqueta= new JLabel("Hola!");
      getContentPane().add(etiqueta);
      setDefaultCloseOperation(EXIT_ON_CLOSE);}
}
```

La clase define un único atributo de instancia que es un objeto de la clase `JLabel`. Cada una de las instrucciones del constructor tiene el siguiente efecto:

- Invoca al constructor de `JFrame` con el texto de la barra de título.
- Establece el tamaño del frame.
- Crea una etiqueta estableciendo su texto.
- Recupera el panel del contenido del frame e inserta en su interior la etiqueta.
- Establece terminar la aplicación cuando el usuario cierre la ventana.

El método `main` en la clase `VentanaEtiqueta` crea un objeto de clase `FrameConEtiqueta`:

```
import javax.swing.*;
class VentanaEtiqueta {
 public static void main(String args[ ])    {
  FrameConEtiqueta f= new FrameConEtiqueta("Ventana con Etiqueta");
  f.setVisible(true);    }
}
```

El objeto ligado a la variable `f` tiene todos los atributos y servicios de `FrameConEtiqueta`, `JFrame` y de sus ancestros en la jerarquía de herencia. La segunda instrucción envía el mensaje `setVisible` al frame con el valor `true` como parámetro, su efecto es obviamente hacer visible el frame. Muchos otros objetos gráficos pueden recibir este mensaje, de modo que el parámetro permite ocultarlos o hacerlos visibles.

La etiqueta tiene asociado un texto y también es posible establecer un ícono:

```
import javax.swing.*;
class UnaImagen extends JFrame{
//Objeto gráfico
    JLabel etiquetaRobot;
public UnaImagen (String tit) {
   super(tit);
   setSize(400, 300);
   etiquetaRobot = new JLabel("Junior");
```

```
   etiquetaRobot.setIcon(new ImageIcon("junior.gif"));
   getContentPane().add(etiquetaRobot);
   setDefaultCloseOperation(EXIT_ON_CLOSE); }
}
```

En este caso se establece un ícono para la etiqueta usando la imagen almacenada en el archivo junior.gif. Como antes, para que la etiqueta sea visible es necesario insertarla en el panel de contenido del frame. El panel de contenido es un objeto contenedor, en el que se insertan componentes gráficas usando el mensaje `add`.

En el siguiente ejemplo se establece la alineación del texto y de la imagen:

```
import java.awt.*;
import javax.swing.*;
class EtiquetaCentro extends JFrame{
JLabel etiquetaRobot;
//Constructor
    public EtiquetaCentro (String tit) {
      super(tit);
      setSize(600, 300);
/*Crea la etiqueta y establece la imagen y la alineación del texto y
de la imagen*/
      etiquetaRobot = new JLabel("Junior");
      etiquetaRobot.setIcon(new ImageIcon("junior.gif"));
      etiquetaRobot.setHorizontalAlignment(JLabel.CENTER);
      etiquetaRobot.setHorizontalTextPosition(JLabel.CENTER);
      etiquetaRobot.setVerticalTextPosition(JLabel.BOTTOM);
/*Establece el color e inserta la etiqueta en el panel de contenido
*/
      getContentPane().setBackground(Color.YELLOW);
      getContentPane().add(etiquetaRobot);
      setDefaultCloseOperation(EXIT_ON_CLOSE);}
}
```

El mensaje:

```
getContentPane()
```

Obtiene el panel de contenido del frame. El panel de contenido es un atributo de instancia del frame y tiene a su vez atributos que pueden modificarse, como por ejemplo el color.

Cuando varias componentes se insertan en el panel de contenido, una queda superpuesta sobre la otra. Para distribuirlas de modo que todas queden visibles es necesario establecer el **diagramado** y también definir un tamaño adecuado para el frame.

El **organizador de diagramado** es un atributo de todos los objetos gráficos contenedores que determina como se distribuyen las componentes contenidas. Algunas de las clases provistas para crear organizadores son `BorderLayout`, `FlowLayout`, `GridLayout`.

`FlowLayout`: Distribuye los componentes uno al lado del otro comenzando en la parte superior. Por omisión provee una alineación centrada, pero también puede alinear a la izquierda o derecha.

`BorderLayout`: Divide el contenedor en cinco regiones: NORTH, SOUTH, EAST, WEST y CENTER, admite un único componente por región.

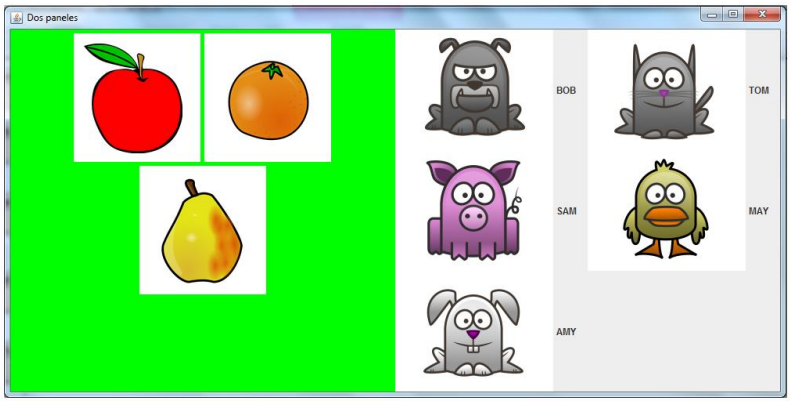
`GridLayout`: Divide el contenedor en una grilla de n filas por m columnas, con todas las celdas de igual tamaño.

Paneles

Aunque es posible insertar las componentes gráficas directamente sobre el panel de contenido del frame, resulta más sencillo organizarlas en paneles, esto es, instancias de la clase `JPanel()`.

Un **panel** es un contenedor de otras componentes gráficas. Los paneles se organizan en forma jerárquica. El principal atributo de un panel es el organizador de diagramado que establece cómo se distribuyen las componentes en su interior, tal como sucede con el panel de contenido.

Así, es posible establecer un organizador de diagramado para el panel de contenido y luego insertar en él dos o más paneles. Cada uno de estos paneles puede tener un diagramado diferente. En cada uno de ellos se insertan componentes que se distribuyen de acuerdo al diagramado establecido. El siguiente ejemplo inserta dos paneles sobre el panel de contenido, cada uno con un diagramado diferente y conteniendo un número distinto de etiquetas.



La GUI incluye dos arreglos de etiquetas. Los elementos de uno de los arreglos se insertan en uno de los paneles y quedan distribuidos de acuerdo a su organizador de diagramado. Los elementos del segundo arreglo se insertan en el otro panel, distribuidos de acuerdo a otro organizador.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class DosPaneles extends JFrame {
//Objetos gráficos
    JLabel [] mascota;
    JLabel [] fruta;
    JPanel panelIzquierdo,panelDerecho;
    Container contenedor;
    public DosPaneles (String tit){
      //Establece los atributos del frame
      super(tit);
      setSize(1000,500);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
      //Establece los atributos del panel de contenido
      contenedor = getContentPane();
      contenedor.setLayout(new GridLayout(1,2));
      contenedor.setBackground(Color.GREEN);
      //Crea los objetos gráficos
      panelIzquierdo = new JPanel();
      panelDerecho = new JPanel();
      mascota = new JLabel[5];
      fruta = new JLabel[3];
//Establece los atributos de los paneles
      panelIzquierdo.setLayout(new FlowLayout());
      panelIzquierdo.setBackground(Color.GREEN);
      panelDerecho.setLayout(new GridLayout(3,2));
//Crea las etiquetas
      mascota[0] = new JLabel("BOB");
      mascota[0].setIcon(new ImageIcon("Perro.gif"));
      mascota[1] = new JLabel("TOM");
      mascota[1].setIcon(new ImageIcon("Gato.gif"));
      mascota[2] = new JLabel("SAM");
      mascota[2].setIcon(new ImageIcon("Cerdo.gif"));
      mascota[3] = new JLabel("MAY");
      mascota[3].setIcon(new ImageIcon("Pato.gif"));
      mascota[4] = new JLabel("AMY");
      mascota[4].setIcon(new ImageIcon("Conejo.gif"));
```

```
      fruta[0] = new JLabel("");
      fruta[0].setIcon(new ImageIcon("manzana.gif"));
      fruta[1] = new JLabel("");
      fruta[1].setIcon(new ImageIcon("naranja.gif"));
      fruta[2] = new JLabel("");
      fruta[2].setIcon(new ImageIcon("pera.gif"));
      /*Inserta las componentes en los paneles
      y los paneles en el panel de contenido*/
      for (int i=0;i<5;i++)
        panelDerecho.add(mascota[i]);
      for (int i=0;i<3;i++)
        panelIzquierdo.add(fruta[i]);
      contenedor.add(panelIzquierdo);
      contenedor.add(panelDerecho); }
}
```

Cada componente de la GUI, en este caso las etiquetas y los paneles, está asociada a un objeto gráfico. Los atributos de la clase DosPaneles mantienen justamente referencias a cada uno de estos objetos.

El comportamiento de estas componentes está establecido en las clases provistas por Swing y AWT. La ventaja de trabajar con distintos paneles va a ser más evidente cuanto mayor sea el número de componentes y cuanto más compleja sea su organización.

Botones

Un **botón** es una componente **reactiva**, esto es, puede percibir la acción del usuario y reaccionar de acuerdo al comportamiento establecido por el programador. Un botón se crea como una instancia de la clase `JButton`.

Como las etiquetas, sus atributos principales son texto e imagen. El texto y la imagen tienen atributos alineación vertical y horizontal. Cada botón tiene también un **color**, un **borde**, una **letra mnemónica** y una **forma**, y además puede estar habilitado o no.

Algunos de los constructores provistos para `JButton` son:

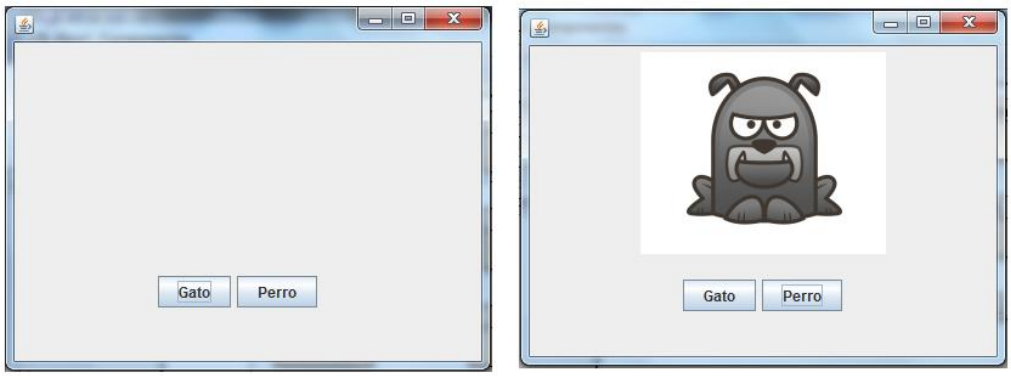
```
JButton ()
JButton (Icon image)
JButton (String text)
JButton (String text, Icon icon)
```

La clase `Dos_Botones` que se muestra a continuación permite crear un frame con dos paneles. En el primer panel se inserta una etiqueta y en el segundo se colocan dos botones. Cuando el usuario oprime un botón cambia la imagen de la etiqueta según la opción elegida.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class DosBotones extends JFrame {
//Componentes graficas
   private Container contenedor;
   private JButton botonGato, botonPerro;
   private JLabel imagen;
   private JPanel panelImagen,panelBotones;
   public DosBotones() {
//Obtiene el panel de contenido y crea los objetos gráficos
    contenedor = getContentPane();
    botonGato = new JButton("Gato");
```

```
    botonPerro = new JButton("Perro");
    imagen = new JLabel();
    panelImagen = new JPanel();
    panelBotones = new JPanel();
//Establece los atributos del frame
    setSize(400, 300);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    armaGUI();}
private void armaGUI(){
//Establece los atributos de los paneles
     contenedor.setLayout (new GridLayout(2,1));
     panelImagen.setLayout (new FlowLayout());
     panelImagen.setPreferredSize(new Dimension(380,200));
     panelBotones.setLayout (new FlowLayout());
     panelBotones.setPreferredSize(new Dimension(380,80));
//Crea un oyente para cada boton y lo registra
     OyenteP oyenteP = new OyenteP();
     OyenteG oyenteG = new OyenteG();
     botonPerro.addActionListener(oyenteP);
     botonGato.addActionListener(oyenteG);
//Inserta la etiqueta y los botones en los paneles
     panelImagen.add(imagen);
     panelBotones.add(botonGato);
     panelBotones.add(botonPerro);
//Inserta los paneles en el panel de contenido
     contenedor.add(panelImagen);
     contenedor.add(panelBotones);
}
private class OyenteP implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        imagen.setIcon(new ImageIcon("Perro.gif"));      }}
private class OyenteG implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        imagen.setIcon(new ImageIcon("Gato.gif"));      }}
}
```

La siguiente figura muestra el frame antes y después de que el usuario oprima el botón con rótulo Perro:



La definición de métodos internos y los comentarios muestran la estructura del código y favorecen la legibilidad.

```
//Obtiene el panel de contenido y crea los objetos gráficos
//Establece los atributos del frame
//Establece los atributos de los paneles
```

```
//Crea un oyente para cada boton y lo registra
//Inserta la etiqueta y los botones en los paneles
//Inserta los paneles en el panel de contenido
```

El panel de contenido del frame se organiza como una grilla de dos filas y una única columna. Los objetos ligados a `panelImagen` y `panelBotones` son instancias de la clase `JPanel` y en ambos el diagramado es un objeto de clase `FlowLayout`. Las componentes se disponen en una fila de un tamaño preestablecido. Si el tamaño es mayor que el requerido, por omisión las componentes quedan centradas.

Una **fuente de evento** es un objeto que está asociado a una componente gráfica y puede percibir una acción del usuario.

En el ejemplo se definen dos objetos fuentes de evento específicos de la aplicación y se ligan a las variables `botonGato` y `botonPerro`.

Un **oyente** es un objeto que espera que ocurra un evento y **reacciona** cuando esto sucede.

Cada botón definido en el ejemplo tiene que reaccionar ante la acción del usuario sobre él; se crean dos **objetos oyente** y se registran a **objetos fuente de evento**. El oyente de `botonPerro` se llama `oyenteP` y es de clase `OyenteP`. En forma análoga, el objeto gráfico `botonGato` se asocia a un objeto oyente `oyenteG` que va detectar las acciones del usuario sobre el botón.

Las clases `OyenteP` y `OyenteG` se definen como internas a la clase `DosPaneles`, de modo que tienen acceso a todos sus atributos de instancia. Cada una implementa a la interface `ActionListerner` provista por Java. La implementación consiste en redefinir el método `actionPerformed` de acuerdo al comportamiento que se requiera en cada botón. En este caso, la imagen de la etiqueta se establece usando la figura almacenada en un archivo.

Java brinda otras clases para crear botones, como por ejemplo `JRadioButton` que permite agrupar botones de modo que solo uno puede estar seleccionado. Los atributos de instancia de la clase son: la cadena, el ícono y el estado.

Los constructores de un objeto de clase `JRadioButton` son:

```
JRadioButton(String s)
JRadioButton(String s, boolean b)
JRadioButton(Icon i)
JRadioButton(Icon i, boolean b)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean b)
JRadioButton()
```

El parámetro de tipo `boolean` establece si el botón está seleccionado.

La clase `RadioButtonDemo` permite crear una GUI como la que muestra la siguiente figura:

La implementación es ahora:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class RadioButtonDemo extends JFrame {
    //Componentes gráficas
    private Container contenedor;
    private JPanel radioPanel,panelImagen ;
    private JLabel  imagen;
    private ButtonGroup group;
    private JRadioButton pato;
    private JRadioButton gato ;
    private JRadioButton perro ;
    private JRadioButton conejo ;
    private JRadioButton cerdo ;
//Constructor
    public RadioButtonDemo() {
        contenedor = getContentPane();
        radioPanel = new JPanel();
        panelImagen = new JPanel();
//Crea el grupo y cada botón
        group = new ButtonGroup();
        pato = new JRadioButton("pato");
        gato = new JRadioButton("gato");
        perro = new JRadioButton("perro");
        conejo = new JRadioButton("conejo");
        cerdo = new JRadioButton("cerdo");
        setSize(400, 320);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
        armaRadioButton();
}
private void armaRadioButton(){
//Establece la imagen inicial para la etiqueta y el tamaño
        imagen = new JLabel(new ImageIcon("pato.gif"));
        imagen.setPreferredSize(new Dimension(180, 180));
//Establece el diagramado de los paneles
        contenedor.setLayout (new BorderLayout());
        radioPanel.setLayout(new GridLayout(0, 1));
//Crea y registra un mismo oyente a todos los botones
        Oyente oyente = new Oyente();
        pato.addActionListener(oyente);
        gato.addActionListener(oyente);
```

```
        perro.addActionListener(oyente);
        conejo.addActionListener(oyente);
        cerdo.addActionListener(oyente);
// Agrupa los botones y activa el boton del pato
        pato.setSelected(true);
        group.add(pato);
        group.add(gato);
        group.add(perro);
        group.add(conejo);
        group.add(cerdo);
//Inserta los botones y la imagen en el panel
        radioPanel.add(pato);
        radioPanel.add(gato);
        radioPanel.add(perro);
        radioPanel.add(conejo);
        radioPanel.add(cerdo);
        panelImagen.add( imagen);
//Inserta los paneles en el contenedor
        contenedor.add(radioPanel, BorderLayout.WEST);
        contenedor.add(panelImagen, BorderLayout.CENTER);    }
private class Oyente implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String  m = (String)e.getActionCommand();
         imagen.setIcon(new ImageIcon(m + ".gif"));
    }
}
}
```

Luego de crear los objetos de clase `JRadioButton` es necesario insertarlos en un objeto de clase `ButtonGroup`. Solo uno de los objetos del grupo puede estar seleccionado.

En este caso se define una sola clase para implementar la interface `ActionListener`. El método `actionPerformed` recibe como parámetro un objeto de clase `ActionEvent`. El objeto mantiene información referida a la acción del usuario sobre la GUI. Cuando se envía el mensaje `getActionCommand` al objeto `e`, el resultado es el comando que corresponde a la acción del usuario. El resultado es en este caso una cadena de caracteres, que se usa para establecer la imagen de la etiqueta.

Cajas de opciones

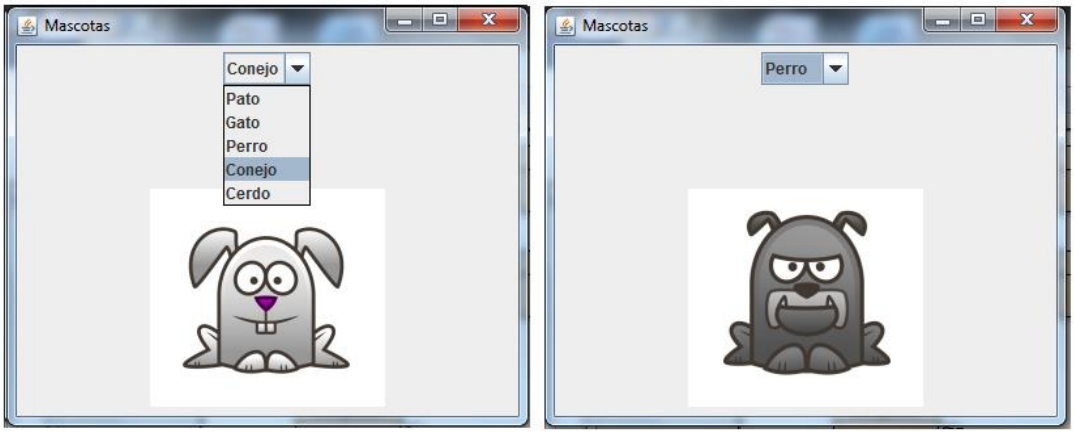
Una **caja de opciones** puede crearse como una instancia de la clase `JComboBox`, editable o no editable. Una caja de opciones no editable consta de una lista de valores drop-down y un botón. Una caja de opciones editable tiene además un campo de texto con un pequeño botón. El usuario puede elegir una opción de la lista o tipear un valor en el campo de texto. El constructor recibe como parámetro un arreglo de objetos de clase `String`.

En el siguiente ejemplo el usuario elige la mascota dentro de la una lista de opciones:

```
import java.awt.*;
import java.awt.Container;
import java.awt.event.*;
import javax.swing.*;
class ComboBoxDemo extends JFrame {
//Opciones de la caja
private String[] mascotas = { "Pato", "Gato", "Perro",   "Conejo",
"Cerdo" };
```

```
//Objetos gráficos
    private Container contenedor;
    private JPanel panelComboBox,panelImagen ;
    private JLabel imagen;
    private JComboBox listaMascotas;
public ComboBoxDemo (String tit) {
//Establece los valores de los atributos del frame
    super(tit);
    setSize(400, 320);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    armaComboBox();}
private void armaComboBox(){
//Obtiene panel de contenido y crea los paneles y la etiqueta
    contenedor = getContentPane();
    panelComboBox = new JPanel();
    panelImagen = new JPanel();
    imagen = new JLabel(new ImageIcon( "conejo.gif"));
/*Crea la caja de opciones y selecciona la opción que corresponde al
ícono establecido en la etiqueta*/
    listaMascotas = new JComboBox(mascotas);
    listaMascotas.setSelectedIndex(3);
//Establece el diagramado y la apariencia de la etiqueta
   contenedor.setLayout (new BorderLayout());
//Crea y registra el oyente
   Oyente oyente = new Oyente ();
   listaMascotas.addActionListener(oyente);
// Inserta los paneles en el panel de contenido
   panelComboBox.add (listaMascotas);
   panelImagen.add(imagen);
   contenedor.add(panelComboBox, BorderLayout.NORTH);
   contenedor.add(panelImagen, BorderLayout.SOUTH);
}
private class Oyente implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String m = (String)listaMascotas.getSelectedItem();
        imagen.setIcon(new ImageIcon(m + ".gif"));}}
}
```

La siguiente figura muestra el frame en el momento en el que el usuario oprime la flecha de la caja, inmediatamente después de crearse el frame y luego de que elige la opción perro:



El objeto `listaMascotas` se registra a un objeto oyente cuya clase implementa el método `ActionPerformed` de la interface `ActionListener`. En este caso, el mensaje `getSelectedItem` se envía al objeto asociado a `listaMascotas` y el valor que retorna se convierte en un objeto de clase `String`. Esta cadena se utiliza para crear una imagen y establecerla en la etiqueta.

Campos de Texto

Un **campo de texto** es una caja que permite ingresar una línea de texto por teclado. Un objeto de la clase `JTextField` permite mantener un campo de texto. Los atributos son una cadena de caracteres, la cantidad de caracteres que se visualizan en la caja y el modelo a utilizar. Cada vez que el usuario tipea una tecla se crean objetos de clase `KeyEvent` o de la clase `ActionEvent`.

Los constructores provistos por la clase son:

```
JTextField()
JTextField(Document doc, String text, int col)
JTextField(int col)
JTextField(String text)
JTextField(String text, int col)
```

En el siguiente ejemplo el oyente detecta la acción del usuario cuando oprime la tecla Intro después de tipear una cadena.

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
class CajaTexto  extends JFrame {
//Componentes Gráficas
      private Container contenedor;
      private JPanel panelImagen,panelCaja;
      private JLabel imagen;
      private JTextField cTexto;
      public CajaTexto () {
 //Establece los atributos del frame
       super("Mi Mascota");
       setSize(400, 320);
       setDefaultCloseOperation(EXIT_ON_CLOSE);
       setLocationRelativeTo(null);
       armaCaja();}
private void armaCaja(){
//Obtiene el panel de contenido
       contenedor =  getContentPane();
//Crea las componentes gráficas
       panelImagen = new JPanel();
       panelCaja = new JPanel();
       imagen = new JLabel();
       cTexto = new JTextField (8);
//Establece los atributos de los objetos gráficos
       contenedor.setLayout(new BorderLayout());
       TitledBorder borde = new TitledBorder (new   LineBorder
(Color.black,5,true),"");
       imagen.setHorizontalAlignment(JLabel.CENTER);
```

```
        imagen.setPreferredSize(new Dimension(180, 180));
        imagen.setBorder(borde);
//Crea y registra el oyente de la caja
        OyenteCaja oyente = new OyenteCaja();
        cTexto.addActionListener(oyente);
//Inserta las componentes en los paneles
        panelImagen.add(imagen);
        panelCaja.add(cTexto);
        contenedor.add(panelImagen,BorderLayout.NORTH);
        contenedor.add(panelCaja,BorderLayout.SOUTH);
}
private class OyenteCaja implements ActionListener {
     public void  actionPerformed(ActionEvent event){
             String m = cTexto.getText();
             imagen.setIcon(new ImageIcon(m + ".gif")); }   }
}
```

En el momento que se crea la caja de texto se establece el tamaño de la componente, sin embargo, el usuario puede tipear más caracteres que los que se visualizan en la caja. Si la clase del oyente implementa `ActionListener` el método `actionPerformed` recibe como siempre un objeto de clase `ActionEvent`. En este caso, se obtiene la cadena de caracteres que corresponde a parte del nombre del archivo, directamente de la caja de texto.

En este ejemplo se ha establecido la preferencia de tamaño y un borde para la etiqueta.

Paneles de diálogo

Un **panel de diálogo** se usa para leer un valor simple y/o mostrar un mensaje. Los atributos incluyen uno o más botones. El mensaje puede ser un error o una advertencia y puede estar acompañado de una imagen o algún otro elemento. Para definir un diálogo estandar se usa la clase `JOptionPane` que brinda los siguientes servicios:

showMessageDialog: Muestra un diálogo modal con un botón, etiquetado "OK". Se puede especificar el icono y el texto del mensaje y del título. Por omisión el ícono es de información

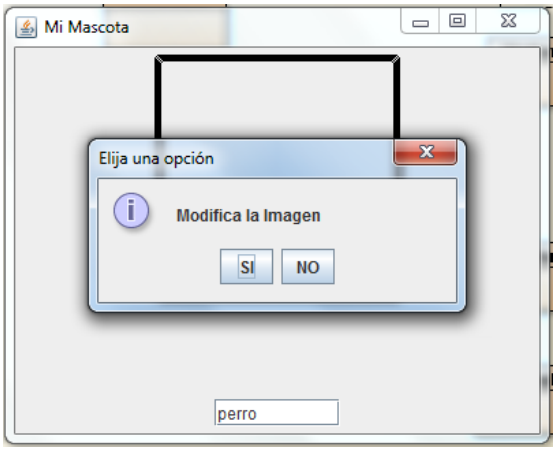
showConfirmDialog: Muestra un diálogo modal con dos botones, etiquetados "Yes" y "No". Por omisión aparece el ícono question.

showOptionDialog: Requiere especificar el texto de los botones.

showInputDialog: Muestra un diálogo modal que obtiene una cadena del usuario. La cadena puede ser ingresada por el usuario en un campo de texto o elegida de un ComboBox no editable.

En cada caso se puede utilizar un icono personalizado, no utilizar ninguno, o utilizar uno de los cuatro iconos estandar de `JOptionPane` (question, information, warning, y error). Por omisión aparece el ícono *question*.

En la clase `CajaTexto` propuesta antes se puede agregar un panel de diálogo con opciones "si" y "no" para consultar al usuario si confirma la modificación de la imagen de la etiqueta.



El código de `actionPerformed` se modifica como sigue:

```
public void  actionPerformed(ActionEvent event){
  String m = cTexto.getText();
  String [] op = {"SI","NO"};
  JOptionPane pd = new JOptionPane();
  int opSel = pd.showOptionDialog(null,
              "Modifica la Imagen",
              "Elija una opción",
               JOptionPane.YES_NO_OPTION,
               JOptionPane.INFORMATION_MESSAGE,
               null, //Icon icon,
               op,
               "Si");//Valor inicial
  if(opSel == 0 )
    imagen.setIcon(new ImageIcon(m + ".gif"));
  cTexto.setText("");
}
```

Las clases `Printing`, `JColorChooser` y `JFileChooser` permiten ofrecer diálogos más específicos.

Todo diálogo es dependiente de un frame. Cuando un frame se destruye, también se destruyen los diálogos que dependen de él. Un diálogo es modal cuando bloquea la entrada de datos del usuario a través de todas las demás ventanas.

Los cuadros de diálogo creados con `JOptionPane` son modales. Para crear un diálogo no modal es posible usar la clase `JDialog`.

Investigue de manera autónoma cuál es la utilidad y cómo se usan las clases `JCheckBox` y `JTextArea`.

Lectura y escritura

Toda la lectura y escritura realizada por componentes de la clase Swing se realiza usando objetos de clase `String`. La entrada y salida de números requiere entonces convertir valores.

La conversión de un valor de tipo `int` a `String` puede implementarse concatenando dos cadenas:

```
String s = ""+42;
```

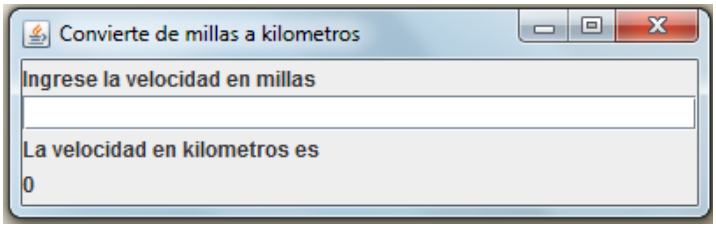
La conversión de una cadena a un entero requiere usar el método `parseInt` de la clase estática `Integer`:

```
String s = "42";
int num = Integer.parseInt(s);
```

En forma análoga es posible convertir una cadena de caracteres al tipo double:

```
double val = Double.parseDouble("42.5");
```

El siguiente ejemplo propone una GUI para convertir un valor representando una velocidad expresada en millas por horas a kilómetros por horas.



La velocidad en millas se lee en una caja de texto y la velocidad en kilómetros se establece en una etiqueta.

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class CajaTextoMllKm extends JFrame {
//Atributos de instancia
  private JPanel panel;
  private JLabel cartelEntrada,cartelSalida,solucion;
  private JTextField caja;
public CajaTextoMllKm (String tit) {
//Establece los atributos del frame
  super(tit);
  setSize(400, 120);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  crearComponentes();
  establecerAtributos();
//Crea y registra el oyente de la caja
  OyenteCaja oyente = new OyenteCaja();
  caja.addActionListener(oyente);
  armarPaneles();}
private void crearComponentes(){
//Crea las components gráficas
  panel = new JPanel();
  cartelEntrada = new JLabel();
  cartelSalida = new JLabel();
  velKm = new JLabel("0");
  caja = new JTextField ("",5);}
private void establecerAtributos(){
//Establece los atributos de las componentes gráficas
  panel.setLayout(new GridLayout(4, 1));
  getContentPane().setLayout(new BorderLayout());
  cartelEntrada.setText("Ingrese la velocidad en millas");
  cartelSalida.setText("La velocidad en kilometros es");}
```

```
private void armarPaneles(){
//Inserta las componentes en los paneles
  panel.add(cartelEntrada);
  panel.add(caja);
  panel.add(cartelSalida);
  panel.add(velKm);
  getContentPane().add(panel,BorderLayout.CENTER);}
private class OyenteCaja implements ActionListener {
  public void  actionPerformed(ActionEvent event){
    String v = caja.getText();
    if (!v.equals("")) {
      double mph = Double.parseDouble (v);
      velKm.setText(""+mph*1.621);}}  }
}
```

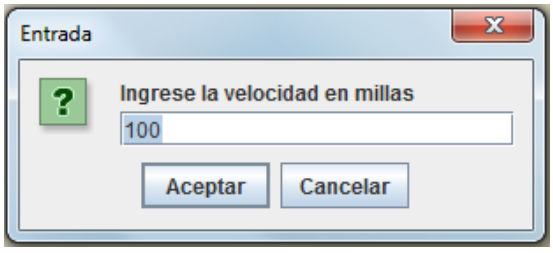
El condicional en el método `actionPerformed` permite evitar una terminación anormal si el usuario no ingresa un valor en la caja de texto.

La clase `DialogoMllKm` propuesta a continuación implementa una GUI similar a la anterior, usando un panel de diálogo:

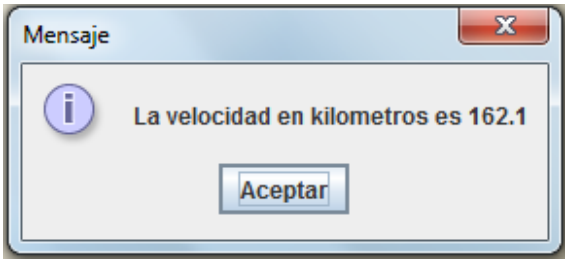
```
import javax.swing.border.*;
import javax.swing.*;
class DialogoMllKm extends JFrame {
  private JOptionPane dialogo;
  public DialogoMllKm() {
        double mph,kph;
        String cad ;
        dialogo = new JOptionPane();
        cad = dialogo.showInputDialog(null,
              "Ingrese la velocidad en millas",100 );
        if (!cad.equals("")){
          mph = Double.parseDouble(cad);
          kph = 1.621 * mph;
         dialogo.showMessageDialog (null,
         "La velocidad en kilometros es "+kph);}
    System.exit(0);}
}
```

En este caso el constructor crea el panel de diálogo y le envía el mensaje `dialogo.showInputDialog` que permite ingresar un valor, inicializado en 100. El valor ingresado es una cadena de caracteres, que puede estar vacía. Si no está vacía, se convierte en un valor de tipo `double` y luego se computa el cambio de unidad.

La creación de un objeto de clase `DialogoMllKm` genera el siguiente panel:



Si el usuario oprime Aceptar aparece:



La solución propuesta asume que cuando el usuario oprime el botón Aceptar la caja de texto del cuadro de diálogo no está vacía.

La estructura de una GUI

La estructura de una GUI debería favorecer la legibilidad de modo que el código sea fácil de verificar y mantener. El código de cada una de las clases que implementan una GUI en este libro incluye:

- Instrucciones para importar paquetes gráficos
- Declaraciones de atributos de instancia
- Definición de un constructor y algunos métodos internos invocados por el constructor
- Definición de clases internas que implementan interfaces y permiten crear oyentes

Java permite definir clases oyentes anónimas e implementar el método `actionPerformed` directamente en el parámetro real del mensaje `addActionListener`. Esta alternativa compromete la legibilidad del código.

También es posible implementar las clases oyentes a través de clases externas. Esta es una alternativa adecuada para definir interfaces gráficas complejas.

En este libro se adopta la convención de modular la solución definiendo clases oyentes que modelan el comportamiento reactivo de una o más componentes gráficas. Para los casos de estudio planteados, las clases internas son un recurso adecuado.

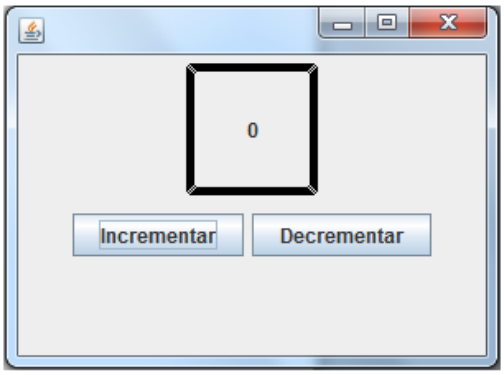
El constructor o los métodos internos invocados por el constructor incluyen instrucciones para:

- Establecer los valores de los atributos gráficos del frame
- Crear objetos gráficos y establecer los valores de sus atributos, algunos de estos objetos serán paneles contenedores
- Crear objetos oyentes y registrarlos a los objetos asociados a componentes reactivas
- Insertar los objetos gráficos en los paneles y los paneles en el panel de contenido

Como ilustra el ejemplo propuesto en esta sección, la interfaz puede incluir también atributos propios de la aplicación, que no corresponden a compontentes gráficas. Como el resto de los atributos de la clase, se declaran privados y solo pueden ser accedidos por los servicios propios y las clases internas.

Es muy importante estructurar el código de la GUI para favorecer la legibilidad. En una interfaz muy simple los comentarios pueden ser suficientes. Cuando el constructor requiere varias líneas de código es aconsejable utilizar métodos auxiliares como se propuso en los ejemplos anteriores. En cualquier caso existen varias maneras de modular.

Dada una GUI como la que muestra la siguiente figura, con dos botones que permiten incrementar y decrementar el número de una etiqueta:



La interfaz requiere crear:

- Una clase que extiende a `JFrame`, llamada `GUI_Contador`.
- Una clase llamada `Contador` con un método `main` que inicia la ejecución al crear una instancia de `GUI_Contador`

La implementación de `Contador` puede ser:

```
class Contador{
public static void main(String[] args) {
  GUI_Contador cuadro = new GUI_Contador();
  cuadro.setVisible(true);}
}
```

La clase `GUI_Contador` crea cinco componentes gráficas

- Dos botones
- Dos paneles
- Una etiqueta

Los atributos de instancia de la clase mantienen justamente referencias a estos objetos. Los dos botones son components reactivas, cada uno queda asociado a un objeto **oyente** que establece su comportamiento.

La estructura del constructor de la clase `GUI_Contador` puede ser:

- Inicializar el contador
- Establecer los valores de los atributos gráficos heredados de la clase `JFrame`.
- Crear los paneles
- Crear la etiqueta para el número, establecer sus atributos e insertar en el panel del contador.
- Crear el botón incrementar, su oyente, registrarlo e insertar en el panel de control.
- Crear el botón decrementar, su oyente, registrarlo e insertar en el panel de control.
- Insertar los paneles en el panel de contenido.

Una implementación de `GUI_Contador` es:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import javax.swing.*;
class GUI_Contador extends JFrame {
//Atributos de instancia incluyendo objetos gráficos
  private int numero;
```

```
  private JPanel pContador,pControl;
  private JLabel numeroEtiqueta;
  private JButton botonIncrementar,botonDecrementar;
  public GUI_Contador() {
//Inicializa el contador
  numero = 0;
//Establece los valores de los atributos del frame
  setLayout(new FlowLayout());
  setSize(300, 320);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  getContentPane().setLayout(new GridLayout(2,1));
//Crea los paneles
  pContador = new JPanel();
  pControl = new JPanel();
/*Crea la etiqueta,  establece los valores de los atributos y la
inserta en el panel del contador*/
  numeroEtiqueta = new JLabel("" + numero);
  TitledBorder borde = new TitledBorder (new   LineBorder
(Color.black,5,true),"");
  numeroEtiqueta.setHorizontalAlignment(JLabel.CENTER);
  numeroEtiqueta.setPreferredSize(new Dimension(80, 80));
  numeroEtiqueta.setBorder(borde);
  pContador.add(numeroEtiqueta);
/*Crea el boton incrementar, el oyente, lo registra e inserta el
botón en el panel de control*/
  botonIncrementar = new JButton("Incrementar");
  OyenteBotonI incrementar = new OyenteBotonI();
  botonIncrementar.addActionListener(incrementar);
  pControl.add(botonIncrementar);
/*Crea el boton decrementar, el oyente, lo registra e inserta el
botón en el panel de control*/
  botonDecrementar = new JButton("Decrementar");
  OyenteBotonD decrementar = new OyenteBotonD();
  botonDecrementar.addActionListener(decrementar);
  pControl.add(botonDecrementar);
/*Inserta el panel del contador y el panel de control en el panel de
contenido*/
  getContentPane().add(pContador);
  getContentPane().add(pControl);}
 private class OyenteBotonI implements ActionListener {
     public void actionPerformed(ActionEvent event) {
      numero++;
      numeroEtiqueta.setText("" + numero);}
    }
 private class OyenteBotonD implements ActionListener {
     public void actionPerformed(ActionEvent event) {
      numero--;
      numeroEtiqueta.setText("" + numero);}
    }
}
```

Por supuesto el código puede estructurarse siguiendo un criterio diferente.

- Establecer los valores de los atributos gráficos heredados de la clase `JFrame`.
- Crear los paneles.
- Crear la etiqueta.

- Crear el botón incrementar y el botón decrementar.
- Crear el oyente del botón incrementar y el oyente del botón decrementar.
- Registrar cada uno de los dos oyentes al botón.
- Insertar la etiqueta y los botones en el panel que corresponda.
- Insertar los paneles en el panel de contenido.

En la siguiente implementación se definen métodos auxiliares, internos a la clase `GUI_Contador`, para modular la solución.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import javax.swing.*;
class GUI_Contador extends JFrame {
//Atributos de instancia incluyendo objetos gráficos
  private int numero=0;
  private JPanel pContador,pControl;
  private JLabel numeroEtiqueta;
  private JButton botonIncrementar,botonDecrementar;
  public GUI_Contador() {
//Establece los valores de los atributos del frame
  setLayout(new FlowLayout());
  setSize(300, 220);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  getContentPane().setLayout(new GridLayout(2,1));
//Crea los paneles
  pContador = new JPanel();
  pControl = new JPanel();

  armarEtiqueta();
  armarBotonesyOyentes();
  armarPaneles();}
private void armarEtiqueta(){
/*Crea la etiqueta y establece los valores de los atributos*/
  numeroEtiqueta = new JLabel("" + numero);
  TitledBorder borde = new TitledBorder (new   LineBorder
(Color.black,5,true),"");
  numeroEtiqueta.setHorizontalAlignment(JLabel.CENTER);
  numeroEtiqueta.setPreferredSize(new Dimension(80, 80));
  numeroEtiqueta.setBorder(borde);}
private void armarBotonesyOyentes(){
/*Crea los dos botones, los dos oyentes y los registra*/
  botonIncrementar = new JButton("Incrementar");
  botonDecrementar = new JButton("Decrementar");
  OyenteBotonI incrementar = new OyenteBotonI();
  OyenteBotonD decrementar = new OyenteBotonD();
  botonIncrementar.addActionListener(incrementar);
  botonDecrementar.addActionListener(decrementar);}
private void armarPaneles(){
/*Inserta la etiqueta y los botones en los paneles correspondientes
y los dos paneles en el panel de contenido*/
  pContador.add(numeroEtiqueta);
  pControl.add(botonIncrementar);
  pControl.add(botonDecrementar);
  getContentPane().add(pContador);
  getContentPane().add(pControl); }
```

```
private class OyenteBotonI implements ActionListener {
     public void actionPerformed(ActionEvent event) {
      numero++;
      numeroEtiqueta.setText("" + numero);}
    }
 private class OyenteBotonD implements ActionListener {
     public void actionPerformed(ActionEvent event) {
      numero--;
      numeroEtiqueta.setText("" + numero);}
    }
}
```

En este caso el atributo de instancia `numero` se inicializa en la declaración. Los comentarios no describen la semántica de cada instrucción, sino que resaltan la estructura elegida para organizar el código.

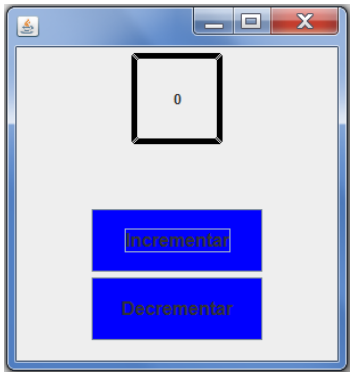
La herencia también permite modular la solución favoreciendo la legibilidad. Los botones se pueden crear con las instrucciones:

```
  botonIncrementar = new BotonContador("Incrementar");
  botonDecrementar = new BotonContador("Decrementar");
```

La clase `BotonContador` extiende a `JButton` estableciendo la misma apariencia para los dos botones de la GUI, excepto el rótulo del botón que varía en cada uno, según se especifica a través de un parámetro:

```
import java.awt.Dimension;
import java.awt.Font;
import javax.swing.*;
class BotonContador extends JButton {
    public BotonContador(String rotulo){
        setPreferredSize(new Dimension(306, 55));
        setFont(new Font("Arial",1,22));
        setText(rotulo);}
}
```

La interfaz es ahora:



Así, el programador puede especializar cualquier clase provista por `JComponent` para obtener soluciones moduladas, estableciendo una única vez los valores de los atributos de todas las componentes gráficas que tengan una misma apariencia.

Programación basada en Eventos

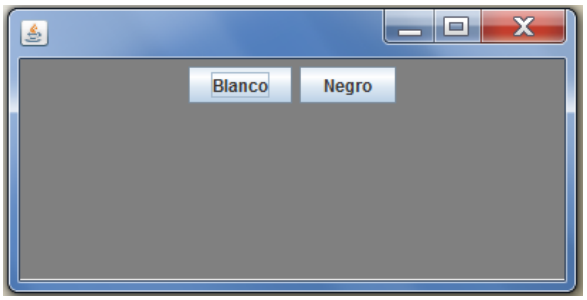
La construcción de una GUI utiliza un modelo de **programación basado en eventos**. En este modelo el orden en el cual se ejecutan las instrucciones de un programa queda determinado por **eventos.** Un evento es una **señal** de que algo ha ocurrido. Así, el flujo de ejecución no lo establece el programador, sino las acciones del usario.

Existen diferentes tipos de eventos, este libro se concentra exclusicamente en aquellos que son generados por acciones del usuario al interactuar con la GUI. Algunas componentes de una GUI son **reactivas**, pueden **percibir** las acciones del usuario y **reaccionar** en respuesta a ellas. Una componente reactiva están asociada a un **objeto fuente del evento** creado por el programador.

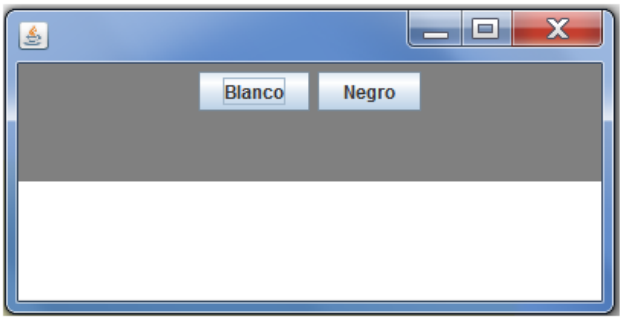
La reacción del sistema en respuesta a la acción del usuario va a quedar determinada por la clase a la que pertenece un **objeto oyente.** El objeto oyente está ligado al objeto fuente de evento a través de una instrucción de **registración**.

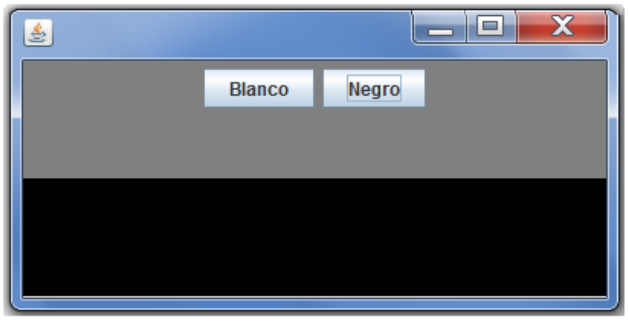
Caso de Estudio: Color panel

Implementar una GUI que permita seleccionar el color de un panel. La ventana inicialmente debe aparecer así:



Una vez que el usuario aprieta un botón, el panel se pinta de blanco o de negro como muestra la figura:





El código de la GUI puede ser:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class  GUIFondoWB extends JFrame{
//Objetos gráficos
   private JPanel panelColor, panelBotones;
   private JButton botonBlanco, botonNegro;
public GUIFondoWB() {
   setSize(400, 200);
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   armaBotones();
   armaPaneles(); }
```

```
private void armaBotones(){
//Crea un botón Blanco, su oyente y lo registra
    botonBlanco = new JButton("Blanco");
    OyenteBotonB ponerBlanco= new OyenteBotonB();
    botonBlanco.addActionListener(ponerBlanco);
//Crea el botón Negro, su oyente  y lo registra
    botonNegro = new JButton("Negro");
    OyenteBotonN ponerNegro= new OyenteBotonN();
    botonNegro.addActionListener(ponerNegro);}
private void armaPaneles(){
/*Se establece el diagramado de los Paneles*/
    Container contenedor = getContentPane();
    contenedor.setLayout(new GridLayout(2,1));
    panelBotones = new JPanel();
    panelColor = new JPanel();
    panelColor.setBackground(Color.GRAY);
    panelBotones.setBackground(Color.GRAY);
//Se agregan los botones al panel de botones
    panelBotones.add(botonBlanco);
    panelBotones.add(botonNegro);
/*Se agregan los paneles al panel de contenido*/
    contenedor.add(panelBotones);
    contenedor.add(panelColor);}
private class OyenteBotonB implements ActionListener {
  public void actionPerformed( ActionEvent event) {
     panelColor.setBackground(Color.white); }
}
private class OyenteBotonN implements ActionListener {
  public void actionPerformed( ActionEvent event) {
      panelColor.setBackground(Color.black); }
}
}
```

Para que se produzca una reacción el programador *registra* **objeto oyente** `ponerBlanco` al **objeto fuente de evento** `botonBlanco`. La clase `OyenteBotonB` *implementa* a la interface `ActionListener` provista por Java. La implementación *redefine* el método `actionPerformed`.

Cuando el **objeto fuente de evento** `botonBlanco` de clase `JButton`, detecta la **acción** del usuario sobre el botón, crea implícitamente un **objeto evento** `e` de clase `ActionEvent`. El método `actionPerformed` redefinido en la implementación de la clase `OyenteBotonB`, recibe como parámetro el **objeto evento** creado implícitamente.

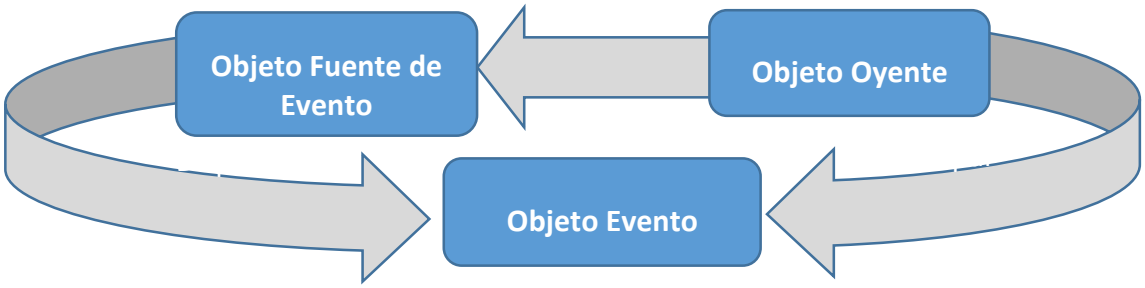
¡Importante!

El programador redefine el método `actionPerformed` que no va a ser invocado explícitamente por él mismo, sino que el mensaje que provoca su ejecución está incluido en algún método definido en una clase provista por Java. Esto es, el programador redefine un método que él mismo NUNCA VA A INVOCAR al menos explícitamente. El método recibe como argumento un **objeto evento** que tampoco ha sido creado explícitamente por el programador, sino que fue generado implícitamente en el momento que el objeto fuente de evento detectó la acción del usuario.

Objetos y Eventos

Algunos de las componentes de una GUI son **reactivas**, están asociadas a **objetos fuente de evento** que ***perciben* eventos externos** y ***disparan* eventos internos**. Cuando se dispara un evento interno se *crea implícitamente* un **objeto evento de software** que *pasa como parámetro* en un mensaje enviado a un **objeto oyente**.

El objeto oyente *se registra* al objeto fuente de evento asociado a la componente reactiva. En respuesta al mensaje el oyente ejecuta un método que corresponde a la acción del usuario.



Diferentes tipos de componentes requieren implementar diferentes interfaces para manejar los eventos internos que disparan los objetos ligados a las componentes.

Un botón dispara eventos llamados **eventos de acción** que son manejados por **oyentes de acción**. Es decir, un objeto fuente de evento de clase `JButton` detecta un **evento externo** provocado por la **acción** del usuario sobre el botón, *dispara* un **evento interno** que *crea* un objeto evento `e` de clase `ActionEvent`. Los objetos de la clase `ActionEvent` requieren implementar la interface `ActionListener`, escribiendo el código del método `actionPerformed` que recibe al objeto evento como parámetro.

La clase de un **objeto fuente de evento** determina así las clases de los **objetos evento** que se crearán implícitamente. La clase del **objeto evento** determina las interfaces de los **objetos oyente** que se deben implementar para establecer el comportamiento asociado a la componente gráfica sobre la que el usuario realizó una acción.

El **objeto oyente** es instancia de una clase que implementa una interface y redefine el método responsable de reaccionar ante la acción del usuario. El **objeto evento** es un parámetro para el método manejador del evento.

La acción del usuario

A partir del objeto evento generado implícitamente, puede obtenerse la acción del usuario asociada al objeto fuente de evento. Así el mensaje `getActionCommand()` permite definir un único oyente para dos o más componentes.

```
//Crea un oyente, los botones y registra el mismo oyente a ambos
  OyenteBoton o = new OyenteBoton();
  botonBlanco= new JButton("Blanco");
  botonBlanco.addActionListener(o);
  botonNegro = new JButton("Negro");
  botonNegro.addActionListener(o);
```

El **objeto evento** `e` mantiene la información del evento externo producido por la acción del usuario, que permite decidir cómo reaccionar:

```
private class OyenteBoton implements ActionListener {
     public void actionPerformed(ActionEvent e) {
      String op = (String) e.getActionCommand();
      if (op.equals("Blanco"))
         panelColor.setBackGround(Color.RED);
      else
         panelColor.setBackGround(Color.GREEN); }
  }
```

La instrucción

```
        String op = (String)e.getActionCommand();
```

Envía el mensaje `getActionCommand` al parámetro `e` para obtener información referida a la componente de la GUI que detecto la acción del usuario. En el caso de un objeto de clase `JButton`, por omisión la acción es el texto establecido en el botón. En un objeto de clase `JTextField` el mensaje `getActionCommand()` retorna el texto tipeado en la caja.

La fuente del evento

A partir del objeto evento generado implícitamente puede obtenerse el objeto fuente de evento enviando el mensaje `getSource()`.

Caso de Estudio: Botonera

Implementar una GUI que muestre cinco botones, inicialmente verdes. Cada vez que se oprime un botón, el color de ese botón pasa a ser rojo. Si un botón se oprime dos veces no se provoca ningún cambio, esto es, permanece rojo.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Botonera extends JFrame {
//Declara un arreglo de botones
   private JButton []botones;
   private JPanel panelBotones;
   public Botonera() {
      setSize(400, 200);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      panelBotones = new JPanel();
     armarBotonera();
     getContentPane().add(panelBotones);}
private void armarBotonera (){
/*Crea cada botón, registra su oyente y lo inserta en el panel*/
  botones = new JButton  [5];
  Oyente oyente = new Oyente();
  for (int i=0;i<5;i++){
    botones[i] = new JButton();
    botones [i].setText(i+"");
    botones [i].addActionListener(oyente);
    panelBotones.add(botones [i]); }
}
private class Oyente implements ActionListener {
  public void actionPerformed(ActionEvent e) {
     JButton b = (JButton) e.getSource();
     b.setBackground(Color.RED);
     b.setText(" ");}
  }
```

```
}
```

En este caso se envía el mensaje `getSource()` al objeto evento `e` para obtener la referencia al botón que recibió el mensaje. Luego, esta referencia se usa para enviarle el mensaje `setBackground` al botón que recibió la acción del usuario de modo que se pinte de rojo.

Los botones están agrupados en un arreglo cuyas componentes son de clase `JButton`. Cada botón está rotulado con un número entre 0 y 4. Cada componente del arreglo se registra a un mismo oyente y se inserta en el panel.

Eventos del Mouse

Cuando un usuario interactúa con una interface gráfica realiza acciones que generan eventos de **alto nivel** y de **bajo nivel**. Estos eventos pueden ser percibidos por la aplicación o no. En los ejemplos anteriores se han percibido y manejado eventos de alto nivel.

Si el usuario oprime el mouse sobre un botón y lo suelta sobre el mismo botón, aunque antes de soltarlo lo arrastre fuera de esta componente gráfica, el objeto fuente de evento captura el evento externo y dispara un **evento interno de alto nivel**, esto es, crea un **objeto evento** de clase `ActionEvent`.

Si en cambio el usuario oprime el mouse sobre un botón, lo arrastra y suelta fuera del botón, no se dispara un evento interno de alto nivel y por lo tanto no se crea un objeto evento.

Sin embargo, en ambos casos, se disparan **eventos internos de bajo nivel**, relacionados con el mouse e independientes de la componente gráfica. Estos eventos están asociados a objetos de clase `MouseEvent`, uno creado como consecuencia del click sobre una componente, otro al arrastrar y otro al soltar el botón. Otros eventos de bajo nivel se producen cuando se oprimen teclas o se manipulan las ventanas.

La clase de un **objeto evento** determina las interfaces que se deben implementar para establecer el comportamiento de los **objetos oyentes** asociados a la componente gráfica sobre la que el usuario realizó una acción:

Objeto Evento	Interface de oyente	Manejador
`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent)`
`ItemEvent`	`ItemListener`	`itemStateChanged(ItemEvent)`

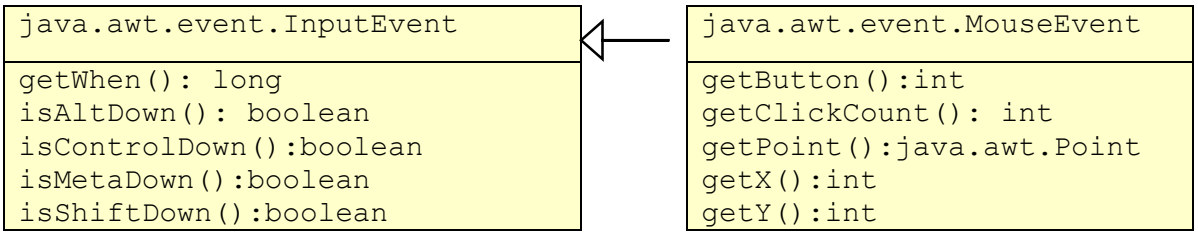
`MouseEvent`	`MouseListener`
	`MouseMotionListener`
		`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`

`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent)`
		`keyRealesed(KeyEvent)`
		`keyTyped(KeyEvent)`

Los atributos del objeto evento brindan información referida a la ubicación del cursor del mouse, cuántos clicks se hicieron, qué botón que se apretó, etc.

La clase del objeto oyente asociada a una componente que detecta acciones del mouse, tiene que implementar los métodos provistos por la interface `MouseListener` o `MouseMotionListener`. El método `addMouseListener` permite registrar el objeto oyente a la componente que va a percibir los eventos del mouse.

Dado que la clase `MouseEvent` hereda de `InputEvent`, sobre un objeto de la clase `MouseEvent` también pueden usarse los métodos definidos en la clase `InputEvent`.



En el siguiente ejemplo el color de un panel cambiará según los diferentes eventos del mouse que se produzcan como consecuencia de las acciones del usuario. En otro panel se muestra la acción del usuario y la posición del indicador del mouse sobre la componente.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class ControlMouse extends JFrame{
//Atributos de instancia
   private JPanel panelMouse, panelEtiqueta;
   private JLabel eventoMouse;
   private Container contenedor;
//Constructor
public ControlMouse() {
  setSize(700,700);
//Establece los atributos del panel de contenido
  contenedor = getContentPane();
  contenedor.setLayout(new GridLayout(2,0));
//Crea y establece los valores de los atributos de los paneles
  panelMouse = new JPanel();
  panelMouse.setBackground(Color.WHITE);
  panelMouse.setPreferredSize
    (new java.awt.Dimension(300, 150));
  panelEtiqueta = new JPanel();
/*Crea una etiqueta, establece los valores de sus atributos y la
inserta en el panel*/
  eventoMouse = new JLabel();
  eventoMouse.setPreferredSize
   (new    java.awt.Dimension(300,70));
  panelEtiqueta.add(eventoMouse);
//Crea el oyente del mouse y lo registra al panel
  OyenteMouse escuchaMouse = new OyenteMouse();
  panelMouse.addMouseListener(escuchaMouse);
```

```
//Inserta los paneles en el panel de contenido
  contenedor.add(panelMouse);
  contenedor.add(panelEtiqueta);}
private class OyenteMouse implements MouseListener{
public void mouseClicked (MouseEvent e)      {
    String s = new String();
    eventoMouse.setText(s.format("Cliqueó en [%d,%d]",
    e.getX(), e.getY()));
    panelMouse.setBackground(Color.BLUE);}
public void mouseEntered (MouseEvent e)      {
    String s = new String();
    eventoMouse.setText(s.format("Entró en[%d,%d]",
    e.getX(), e.getY()));
    panelMouse.setBackground(Color.RED);}
public void mouseExited (MouseEvent e) {
     String s = new String();
      eventoMouse.setText(s.format(" Salió en [%d,%d]",
     e.getX(), e.getY()));
      panelMouse.setBackground(Color.GREEN);}
public void mouseReleased (MouseEvent e)     {
    String s = new String();
      eventoMouse.setText(s.format(" Soltó en [%d,%d]",
     e.getX(), e.getY()));
      panelMouse.setBackground(Color.MAGENTA);}
public void mousePressed (MouseEvent e)      {
    String s = new String();
      eventoMouse.setText(s.format(" Presionó en [%d,%d]",
     e.getX(), e.getY()));
      panelMouse.setBackground(Color.YELLOW);}}
}
```

La implementación de la interface `MouseListener` exige redefinir cinco métodos para controlar cinco eventos sobre el ratón:

`getButton():int` `getClickCount(): int`

`java.awt.event.MouseListener`
`mouseClicked(e:MouseEvent): void`
`mousePressed(e:MouseEvent): void`
`mouseReleased(e:MouseEvent): void`
`mouseEntered(e:MouseEvent): void`
`mouseExited(e:MouseEvent): void`

La clase oyente puede implementar las dos interfaces provistas para detectar acciones sobre el ratón:

```
private class OyenteMouse implements MouseListener ,
                      MouseMotionListener {
…
public void mouseDragged (MouseEvent e) {
     String s = new String();
     eventoMouse.setText(s.format(" Arrastró a [%d,%d]",
      e.getX(),e.getY()));}
public void mouseMoved(MouseEvent e)  {
    String s = new String();
    eventoMouse.setText(s.format(" Se movió a [%d,%d]",
     e.getX(),e.getY()));}
}
```

En este caso la clase `OyenteMouse` redefine los cinco métodos anteriores y además los dos que corresponden al ratón en movimiento. La estructura de la clase es entonces:

```
private class OyenteMouse implements MouseListener ,
                      MouseMotionListener {
 public void mouseClicked (MouseEvent e)     { … }
 public void mousePressed(MouseEvent e)      { … }
 public void mouseReleased (MouseEvent e)    { … }
 public void mouseEntered (MouseEvent e)     { … }
 public void mouseExited (MouseEvent e)     { … }
 public void mouseDragged (MouseEvent e)     { … }
 public void mouseMoved(MouseEvent e)       { … }
}
```

Con frecuencia una interface de eventos brinda más servicios que los que necesitamos. Por ejemplo, la interface `MouseListener` ofrece cinco servicios, si la GUI solo va a reaccionar ante un click del mouse, se define el código de `mouseClicked`, los otros cuatro quedan sin especificar. Sin embargo, si se define una clase que implementa a esta interface, es necesario implementar todos los servicios provistos.

Encapsulamiento y GUI

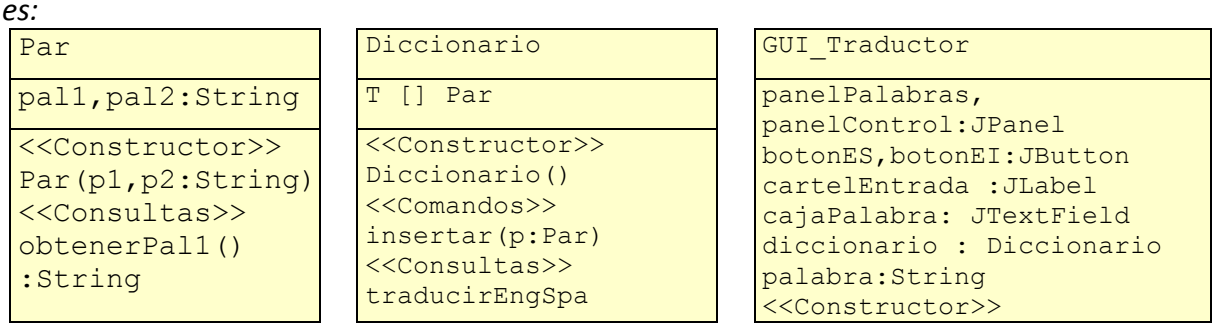
En el diseño de una aplicación, la solución se modula de modo tal que cada clase pueda implementarse sin depender de las demás. En el desarrollo de una aplicación en la cual la interacción con el usuario se realiza a través de una GUI, la clase que implementa la interface gráfica de usuario usa a las clases que modelan el problema, sin conocer detalles de la representación. Análogamente, las clases que modelan el problema se diseñan e implementan si saber si la entrada y salida va a hacerse por consola o mediante una GUI.

El siguiente caso de estudio ilustra cómo definir una GUI que permita traducir palabras de inglés a español y de español a inglés, desconociendo cómo se representa internamente el diccionario. La clase `Traductor` es cliente de los servicios provistos por la clase `Diccionario`. La implementación de estos servicios queda encapsulada en la clase proveedora y es transparente para la clase cliente.

Caso de Estudio: Traductor

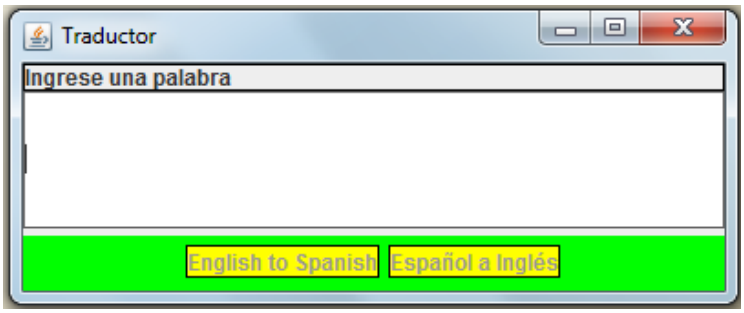
Implemente una clase GUI_Traductor que permita mostrar la traducción de una palabra ingresada en inglés a español o una palabra ingresada en español a inglés.

La clase GUI_Traductor está asociada a Diccionario. La clase Diccionario encapsula a una colección de pares de palabras, la primera palabra del par está en inglés y la segunda es su traducción al español. Claramente un objeto de clase Diccionario puede utilizarse para traducir de inglés a español o de español a inglés. El diagrama de clases para este problema es:



obtenerPal2() :String	(pan:String):String traducirEspIng (pal:String):String	GUI_Traductor(tit:String, dic:Diccionario)

Inicialmente la interfaz corresponde a la siguiente figura:



Cuando el usuario oprime intro, luego de completar la cada de texto, se habilitan los dos botones. Cada botón está asociado a un oyente que envía un mensaje al diccionario para buscar la palabra. Si existe la traducción se muestra en un panel de diálogo. En caso contrario se muestra un mensaje, también en un panel de diálogo.

La solución puede implementarse como sigue:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
class GUITraductor extends JFrame {
//Atributos gráficos
        private JPanel panelPalabras, panelControl;
        private JButton botonES,botonEI;
        private JLabel cartelEntrada;
        private JTextField cajaPalabra;
//Atributos de la aplicación
        private Diccionario diccionario;
        private String palabra;

        public GUITraductor (String tit,Diccionario dicc) {
//Establece la apariencia del frame
         super(tit);
         setSize(400, 200);
         setDefaultCloseOperation(EXIT_ON_CLOSE);
         diccionario = dicc;
//Crea los objetos gráficos
        panelPalabras = new JPanel();
       panelControl = new JPanel();
       botonES = new BotonTraductor();
       botonEI = new BotonTraductor();
        cartelEntrada = new JLabel();
        cajaPalabra = new JTextField ("",15);
        armaGUI();}
private void armaGUI(){
        palabra = " ";
        armarPaneles();
       armarBotones();
//Establece los atributos de las etiquetas
```

```
        cartelEntrada.setText("Ingrese una palabra");
        cartelEntrada.setSize(120,15);
        cartelEntrada.setPreferredSize(new
                        java.awt.Dimension(120,15));
       cartelEntrada.setBorder (new LineBorder(
                             new Color(0,0,0), 1, false));
//Establece los atributos de la caja y la registra al oyente
        cajaPalabra.setSize(120,15);
        cajaPalabra.setPreferredSize(new
java.awt.Dimension(120,15));
       cajaPalabra.setBorder(new LineBorder(new
                              Color(100,100,100), 1, false));
       OyentePalabra oyente = new OyentePalabra();
        cajaPalabra.addActionListener(oyente);
//Inserta los objetos gráficos en los paneles
        panelPalabras.add(cartelEntrada,BorderLayout.NORTH);
        panelPalabras.add(cajaPalabra,BorderLayout.CENTER);
        panelControl.add(botonES);
        panelControl.add(botonEI);
//Establece el organizador del panel de contenido
        getContentPane().setLayout(new BorderLayout());
//Inserta los paneles en el panel de contenido
        getContentPane().add(panelPalabras,BorderLayout.NORTH);
        getContentPane().add(panelControl,BorderLayout.SOUTH); }
private void armarPaneles(){
//Establece los atributos de los paneles
        panelPalabras.setLayout(new BorderLayout());
        panelPalabras.setSize(400,90);
        panelPalabras.setPreferredSize (new
                       java.awt.Dimension(400,90));
        panelControl.setLayout(new FlowLayout());
        panelControl.setSize(400,30);
        panelControl.setPreferredSize (new
                            java.awt.Dimension(400,30));
        panelControl.setBackground(Color.GREEN);}
private void armarBotones(){
//Establece los atributos de los botones y registra sus oyentes
      estadoBotones(false);
       botonES.setText ("English to Spanish");
       OyenteBotonES oyenteES = new OyenteBotonES();
        botonES.addActionListener(oyenteES);
       botonEI.setText ("Español a Inglés");
       OyenteBotonEI oyenteEI = new OyenteBotonEI();
        botonEI.addActionListener(oyenteEI);}
private void estadoBotones(boolean estado){
//Habilita o deshabiliata botones
         botonEI.setEnabled(estado);
         botonES.setEnabled(estado);}
private class OyentePalabra implements ActionListener {
       public void  actionPerformed(ActionEvent event){
            palabra =  (cajaPalabra.getText());
            estadoBotones(true);}
}
private class OyenteBotonES implements ActionListener {
   public void  actionPerformed(ActionEvent event){
```

```
      JOptionPane dialogo = new JOptionPane();
      String traducida = diccionario.traducirEngSpa(palabra);
      if (traducida == null)
         dialogo.showMessageDialog(null,
        "La palabra no figura en el diccionario", "",
         dialogo.INFORMATION_MESSAGE );
      else{
         cajaPalabra.setText("");
         estadoBotones(false);
         dialogo.showMessageDialog(null,
         "La traduccion de "+palabra+ " es " +traducida, "",
         dialogo.INFORMATION_MESSAGE );}}
}
private class OyenteBotonEI implements ActionListener {
    public void  actionPerformed(ActionEvent event){
      JOptionPane dialogo = new JOptionPane();
      String traducida = diccionario.traducirEspIng(palabra);
      if (traducida == null)
         dialogo.showMessageDialog(null,
         "La palabra no figura en el diccionario","",
         dialogo.INFORMATION_MESSAGE );
       else{
         cajaPalabra.setText("");
         estadoBotones(false);
         dialogo.showMessageDialog(null,
         "La traduccion de "+palabra+ " es "+traducida,"",
         dialogo.INFORMATION_MESSAGE );}}
   }
}
class BotonTraductor extends JButton{
public BotonTraductor(){
     setSize(20,5);
     setBackground(Color.YELLOW);
     setBorder(new LineBorder(new Color(0,0,0), 1, false));}
}
```

El objeto ligado a la variable `palabra` se inicializa en el oyente de la caja y se usa para buscar la traducción en las clases de los oyentes.

El problema planteado se modela a partir de los tipos de componentes descriptos antes. La principal diferencia con los ejemplos propuestos es que la implementación de la clase oyente requiere acceder a un diccionario. El diccionario es una estructura de datos que mantiene pares de palabras, la primera palabra del par está en inglés y la segunda en español.

Cuando el usuario realiza una acción sobre `botonES` se envía el mensaje `traducirEngSpa(palabra)` al diccionario. Análogamente, cuando el usuario realiza una acción en `botonEI` se envía el mensaje `traducirEngSpa(palabra)` al diccionario.

La clase `GUITraductor` **usa** a la clase `Diccionario` como una caja negra, conoce únicamente la interface y el contrato. Los datos pueden estar modelados por un arreglo o un objeto de clase `Vector`, las búsquedas pueden implicar un recorrido secuencial o la estructura puede estar ordenada y se aplica entonces búsqueda binaria. Todas esas cuestiones quedan escondidas para la interfaz. De manera análoga, es posible cambiar el diseño y la implementación de la GUI, sin afectar a la clase `Diccionario`.

Otra característica que aparece en esta GUI es que los botones se habilitan y deshabilitan.

Herencia, Polimorfismo y GUI

Una GUI en Java se construye a partir de objetos gráficos de algunas de las clases gráficas provistas por los paquetes AWT y Swing o de clases derivadas. Conceptualmente la implementación de GUI está fuertemente ligada a los conceptos de herencia, polimorfismo y vinculación dinámica.

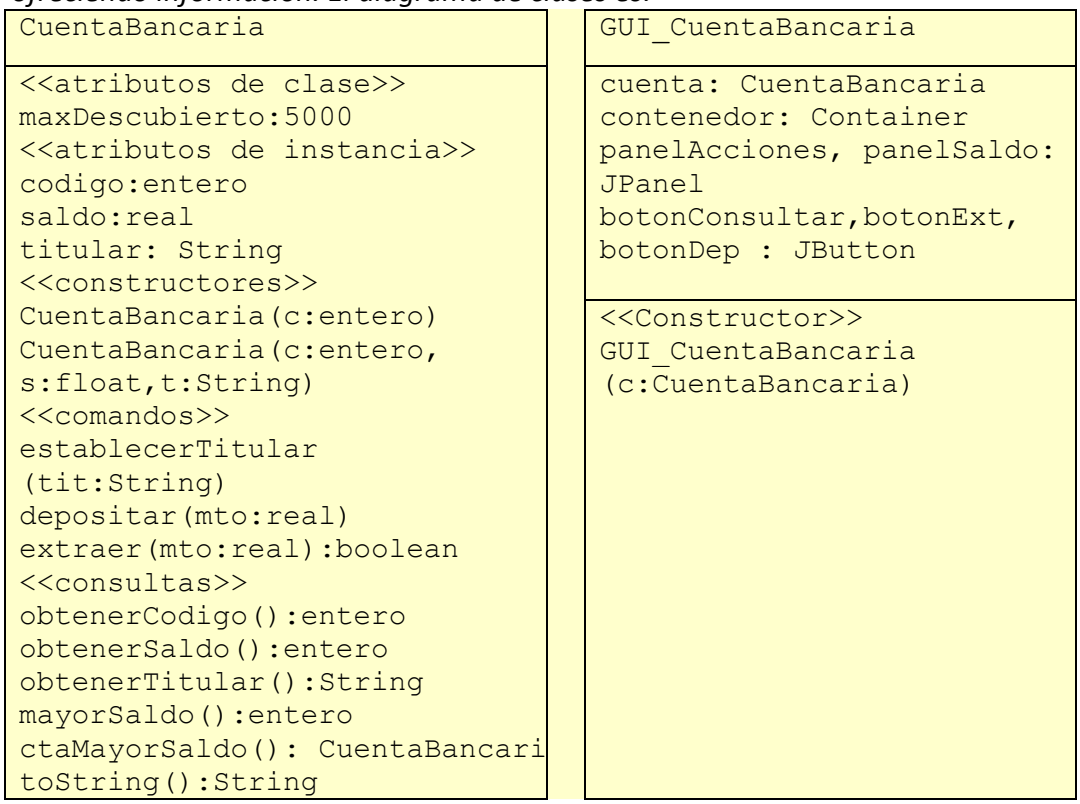
En los ejemplos anteriores se han construido interfaces gráficas creando componentes de alguna clase derivada de la clase `JComponent` provista por el paquete Swing. Esta clase deriva a su vez de `Component` provista por AWT y deriva directamente de `Object`. Cuando una clase extiende a otra **hereda** todos sus métodos de modo que es posible reusar los servicios provistos por las clases más generales.

La clase `Container` de AWT brinda el método `add` heredado por todos sus descendientes. El parámetro formal del método `add` es de clase `Component`.

El siguiente caso de estudio nos permite ilustrar la vinculación entre los conceptos de herencia y polimorfismo con la construcción de interfaces gráficas de usuarios.

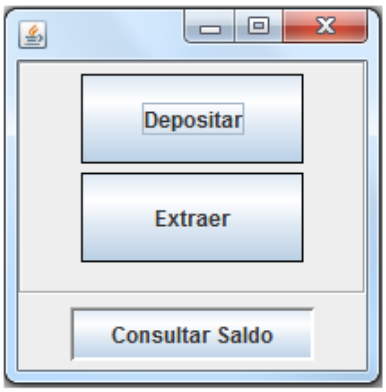
Caso de Estudio: Cuenta Bancaria

Implemente una GUI que brinde botones para efectuar un depósito o extracción en una cuenta bancaria o para consultar el saldo. Los dos primeros botones se insertan en un panel y el tercero en otro. Si el usuario elige depositar o extraer la interfaz muestra un campo de texto para tipear el monto. La acción de extraer requiere verificar que el monto sea menor al saldo de la cuenta. Si no es posible concretar la extracción, aparece un mensaje indicándolo. Tanto el depósito, como la extracción realizada y la consulta de saldo, concluyen con un panel de diálogo ofreciendo información. El diagrama de clases es:

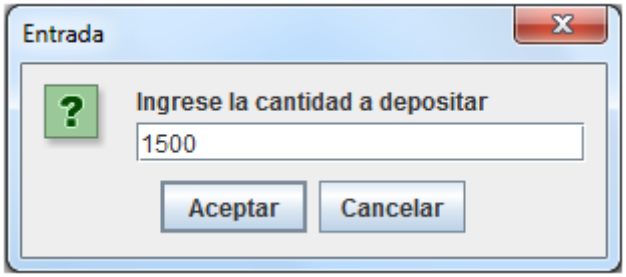


Requiere código > 0 y saldo>=0	

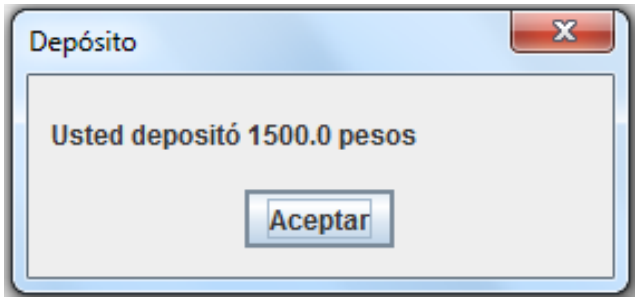
Inicialmente la interfaz corresponde a la siguiente figura:



Si el usuario oprime Depositar aparece el campo de texto:



Y una vez que el usuario oprime Intro,



El código que modela la solución es:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import javax.swing.*;
class GUI_CuentaBancaria extends JFrame {
//Objetos del problema y objetos gráficos
    private CuentaBancaria cuenta;
    private Container contenedor;
    private JPanel panelAcciones, panelSaldo;
    private JButton botonConsultar,botonExt, botonDep;
    public GUI_CuentaBancaria(CuentaBancaria cta) {
//Establece los atributos del frame
     super("Cuenta Bancaria "+cta.obtenerCodigo());
```

```
     setSize(210, 210);
     setDefaultCloseOperation(EXIT_ON_CLOSE);
// Inicializa la cuenta bancaria
     cuenta =cta;
//Obtiene el panel de contenido y crea los paneles
     contenedor = getContentPane();
     panelAcciones = new JPanel();
     panelSaldo = new JPanel();
//Crear los botones
      botonDep = new JButton();
     botonExt = new JButton();
     botonConsultar = new JButton();
     armarBotones();
     armarPaneles();
}
private void armarBotones(){
//Establece atributos de los botones y los vincula a oyentes
     botonDep.setText("Depositar");
     botonDep.setPreferredSize(new Dimension(124, 50));
     botonDep.setSize(150, 50);
      botonDep.setBorder
        (BorderFactory.createCompoundBorder(
         new LineBorder(new java.awt.Color(0, 0, 0), 1, false),
                       null));
     OyenteDepositar oDepositar = new OyenteDepositar();
     botonDep.addActionListener(oDepositar);

     botonExt.setText("Extraer");
     botonExt.setPreferredSize(new Dimension(124, 50));
     botonExt.setSize(150, 50);
      botonExt.setBorder
         (BorderFactory.createCompoundBorder(
          new LineBorder(new java.awt.Color(0, 0, 0), 1, false),
                       null));
     OyenteExtraer oExtraer = new OyenteExtraer();
     botonExt.addActionListener(oExtraer);
     botonConsultar.setText("Consultar Saldo");
     botonConsultar.setPreferredSize(new Dimension(136, 30));
     botonConsultar.setSize(150, 30);
     botonConsultar.setBorder  (BorderFactory.createBevelBorder
        (BevelBorder.LOWERED));
     OyenteConsultar oConsultar = new OyenteConsultar();
     botonConsultar.addActionListener(oConsultar);}
private void armarPaneles() {
// Diagramado de los paneles
      contenedor.setLayout(new BorderLayout());
      panelAcciones.setBorder
        (BorderFactory.createEtchedBorder(BevelBorder.LOWERED));
      panelAcciones.setPreferredSize(new Dimension(160, 130));
      panelAcciones.setSize(160, 125);
// Agrega botones a los paneles
     panelAcciones.add(botonDep);
      panelAcciones.add(botonExt);
     panelSaldo.add(botonConsultar);
// Agrega los paneles al contenedor
```

```
       contenedor.add(panelAcciones, BorderLayout.NORTH);
       contenedor.add(panelSaldo, BorderLayout.SOUTH);     }
private class OyenteDepositar implements ActionListener {
 public void actionPerformed(ActionEvent event){
  float dep;
  String deposito;
  JOptionPane dialogo = new JOptionPane();
  deposito=dialogo.showInputDialog("Ingrese la cantidad a depositar"
);
  if ((deposito != null) && (deposito.length() > 0)){
    dep = Float.parseFloat(deposito);
    dialogo.showMessageDialog(null,"Usted depositó " + dep+ "
          pesos","Depósito", JOptionPane.PLAIN_MESSAGE );
     cuenta.depositar(dep);}
  }
}
private class OyenteExtraer implements ActionListener {
       public void actionPerformed(ActionEvent event){
  float ext;
  String extraccion;
  JOptionPane dialogo = new JOptionPane();
  extraccion=dialogo.showInputDialog("Ingrese la cantidad a extraer"
);
  if ((extraccion != null) && (extraccion.length() > 0)){
   ext = Float.parseFloat(extraccion);
   if (cuenta.extraer(ext)){
       dialogo.showMessageDialog( null,
                 "Usted extrajo " + ext+ " pesos"," Extracción",
     JOptionPane.PLAIN_MESSAGE );
      cuenta.extraer(ext);}
    else
      dialogo.showMessageDialog( null,
          "Usted NO puede extraer esa cantidad",  "Advertencia",
           JOptionPane.WARNING_MESSAGE ); }
  }
}
private class OyenteConsultar implements ActionListener {
  public void actionPerformed(ActionEvent event){
     JOptionPane dialogo = new JOptionPane();
     if (cuenta.obtenerSaldo()>=0)
           dialogo.showMessageDialog(null,
      "Usted tiene un saldo de " + cuenta.obtenerSaldo()+
      " pesos","SALDO",dialogo.INFORMATION_MESSAGE );
     else
       dialogo.showMessageDialog(null,
          "Usted está en descubierto en " +
          (-1)*cuenta.obtenerSaldo() + " pesos",
           "SALDO", JOptionPane.ERROR_MESSAGE );}
   }
}
```

En las instrucciones:

```
     panelAcciones.add(botonDep);
     contenedor.add(panelAcciones);
```

El método `add` es **polimórfico**, el parámetro real puede ser cualquier clase derivada de `Component`. En la primera instrucción un objeto de clase `JPanel` recibe el mensaje `add` con un parámetro de clase `JButton`. En la segunda un objeto de clase `Container` recibe el mensaje `add` con un parámetro de clase `JPanel`.

En la instrucción:

```
botonDep.addActionListener(oDepositar);
```

El parámetro real `oDepositar` es de clase `OyenteDepositar` que deriva directamente de `Object` e implementa a la interface `ActionListener` provista por Java. En la implementación de `addActionListener` claramente el parámetro formal no puede ser de la misma clase que el parámetro real, que se define en cada aplicación. El método `addActionListener` es **polimórfico**, recibe parámetros de distintas clases, todas derivadas de una misma clase base.

```
botonDep.addActionListener(oDepositar);
botonExt.addActionListener(oExtraer);
botonConsultar.addActionListener(oConsultar);
```

La implementación de oyentes como clases internas solo resulta adecuada para definir GUI simples como las de los ejemplos y casos de estudio propuestos. Aun en estos casos, es conveniente estructurar adecuadamente las soluciones dividiendo el problema en subproblemas. En este ejemplo se modula la solución definiendo métodos internos `armarBotones()` y `armarPaneles()`.

La clase `GUI_CuentaBancaria` usa los servicios provistos por `CuentaBancaria` sin conocer la representación de los datos ni la implementación de las operaciones. En tanto se mantenga el contrato, cualquiera de las dos clases puede modificarse sin afectar a la otra.

Correctitud y GUI

La verificación de una aplicación no garantiza que es correcta pero permite asegurar que funciona de acuerdo a los requerimientos para un conjunto de casos de prueba. Cuando la interacción con el usuario se realiza a través de una GUI el flujo de la ejecución queda determinado por eventos que se generan en respuesta a las acciones del usuario. En este contexto es muy difícil o incluso imposible, anticipar todos los flujos de ejecución posibles. En la verificación se analiza el comportamiento de cada componente considerando diferentes flujos de ejecución.

Caso de Estudio: Extracciones en un cajero

Implemente una GUI que permita realizar extracciones de sumas fijas en un cajero. La GUI incluye tres paneles. En el primero aparece un combo box y un botón, en el segundo una etiqueta y en el tercero un arreglo de 5 botones rotulados con números 100, 200, 500, 750 y 1000. El tercer panel inicialmente no está visible.

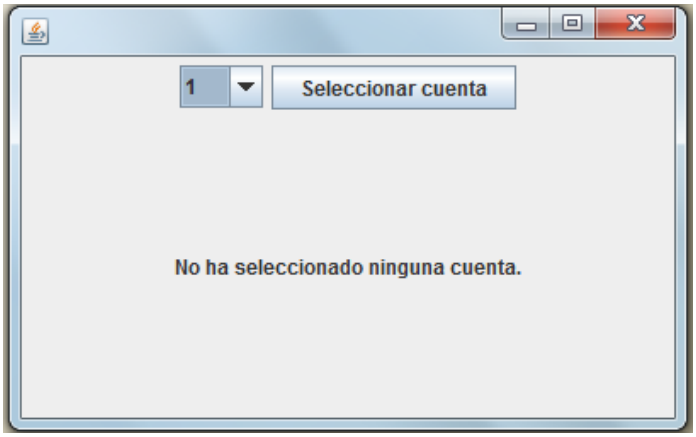
El combo box se inicializa con los códigos de las cuentas bancarias almacenados en una estructura de datos asociada. Cuando el usuario selecciona el código de una cuenta bancaria y oprime el botón seleccionar, se recupera la cuenta bancaria de la cartera y se establece el saldo y el titular como rótulo en la etiqueta. Se pone visible el panel con el arreglo con 5 botones. El usuario selecciona el botón que indica el monto de la extracción. Si el monto es menor que el saldo aparece un panel de opciones mostrando el saldo de la cuenta y se solicita que se confirme o cancele la operación. Si el usuario confirma se modifica el saldo de la cuenta

y se vuelve al estado inicial. Si no es posible realizar la extracción aparece un diálogo adecuado. En este caso la clase GUI_Cajero está asociada a la clase CarteraClientes que a su vez usa los servicios de CuentaBancaria.

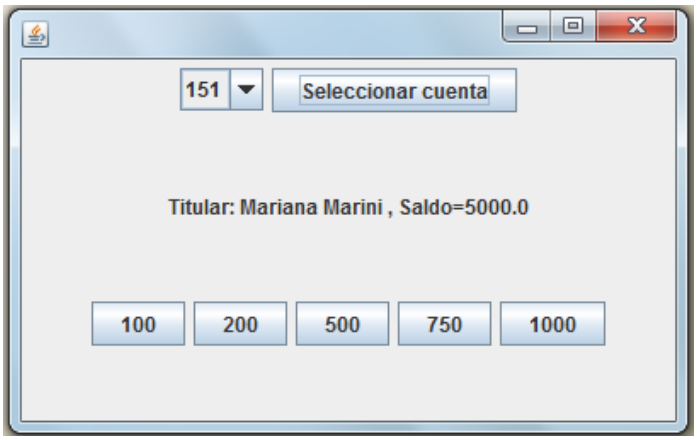
Para verificar la aplicación se propone la siguiente una clase tester:

```
class testCajero{
 public static void main(String[] args) {
   CarteraClientes cc= new CarteraClientes(3);
   CuentaBancaria c1=new CuentaBancaria(1, "Juan Paredes");
   c1.establecerSaldo(10000);
   CuentaBancaria c2=new CuentaBancaria(2, "Juan Carlos Petruza");
   c2.establecerSaldo(8000);
   CuentaBancaria c3=new CuentaBancaria(151, "Mariana Marini");
   cc.insertar(c1);
   cc.insertar(c2);
   cc.insertar(c3);
   c3.establecerSaldo(5000);
   GUI_Cajero cajero = new GUI_Cajero(cc);
   cajero.setVisible(true);}
}
```

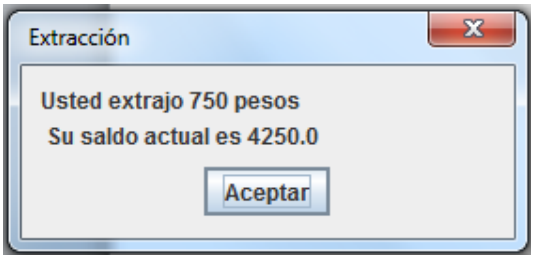
La clase tester establece las cuentas bancarias de la cartera, sin embargo la verficación depende de los eventos generados por el usuario sobre la GUI, que inicialmente aparecerá como muestra la figura:



Si el usuario selecciona la cuenta 151 de la caja de opciones y oprime el botón:



Si el usuario oprime el botón rotulado con 750, aparece un panel de diálogo como el que sigue:



Y la GUI vuelve a solicitar que se seleccione una cuenta. El código de `GUI_Cajero` es:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
class GUI_Cajero extends JFrame {
//Objetos del problema y objetos gráficos
    private CarteraClientes cuentas;
    private CuentaBancaria cuentaActual;
    private Container contenedor;
    private JPanel panelSeleccion, panelInfo, panelCantidad;
    private JButton botonSeleccionar;
    private JButton[] botonera;
    private JLabel labelInfo;
    private JComboBox comboCuentas;
public GUI_Cajero(CarteraClientes ctas) {
// Inicializa la colección de cuentas
        cuentas =ctas;
//Obtiene el panel de contenido y crea los paneles
        contenedor = getContentPane();
        panelSeleccion = new JPanel();
        panelInfo = new JPanel();
        panelCantidad = new JPanel();
//Crea la etiqueta
        labelInfo = new JLabel("No ha seleccionado ninguna
cuenta.");
        armarComboCuentas();
        armaBotones();
//Establece los atributos del frame
        setSize(400, 250);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        armarGUI();
}
private void armarGUI() {
// Establece el diagramado del panel de contenido
        contenedor.setLayout(new BoxLayout(contenedor,
                             BoxLayout.Y_AXIS));
//Inserta los objetos gráficos a los paneles
        panelSeleccion.add(comboCuentas);
        panelSeleccion.add(botonSeleccionar);
        panelInfo.add(labelInfo);
        panelCantidad.setVisible(false);
//Agrega los paneles al contenedor
        contenedor.add(panelSeleccion);
        contenedor.add(panelInfo);
        contenedor.add(panelCantidad);}
private void armarComboCuentas(){
```

```
        String[] lista= new String[cuentas.cantElementos()];
        int n; CuentaBancaria m;
        for (int i=0; i< cuentas.cantElementos(); i++){
            m = cuentas.obtenerCuenta(i);
            n = m.obtenerCodigo();
            lista[i] = ""+n; }
        comboCuentas = new JComboBox(lista);}
private void armaBotones(){
//Establece atributos del boton y lo vincula a su oyente
        botonSeleccionar = new JButton();
        botonSeleccionar.setText("Seleccionar cuenta");
        botonSeleccionar.addActionListener(new OyenteSeleccionar());
//Crea la botonera y vincula todos los botones al oyente
        int cantBotones=5;
        botonera= new JButton[cantBotones];
        for (int i=0; i<botonera.length;i++){
            botonera[i]= new JButton();
            botonera[i].addActionListener(new OyenteExtraccion());}
        botonera[0].setText("100");
        botonera[1].setText("200");
        botonera[2].setText("500");
        botonera[3].setText("750");
        botonera[4].setText("1000");
        for (int i=0; i<5;i++){
            panelCantidad.add(botonera[i]);}}
    private class OyenteSeleccionar implements ActionListener {
        public void actionPerformed(ActionEvent event){
            int nroCuenta= Integer.parseInt
                (comboCuentas.getSelectedItem().toString());
            cuentaActual=cuentas.recuperarCuenta(nroCuenta);
            if (cuentaActual!=null){
                String mensaje= "Titular: " +
                cuentaActual.obtenerTitular() + " , Saldo="+
                cuentaActual.obtenerSaldo();
                labelInfo.setText(mensaje);
                panelCantidad.setVisible(true);}
        }
    }
    private class OyenteExtraccion implements ActionListener {
        public void actionPerformed(ActionEvent event){
            JButton boton= (JButton) event.getSource();
            int extraccion= Integer.parseInt(boton.getText());
            JOptionPane dialogo = new JOptionPane();
            if (cuentaActual.obtenerSaldo()>=extraccion){
                cuentaActual.extraer(extraccion);
                dialogo.showMessageDialog( null,
                    "Usted extrajo " + extraccion+
                " pesos \n Su saldo actual es "
             + cuentaActual.obtenerSaldo()," Extracción",
               JOptionPane.PLAIN_MESSAGE );
                labelInfo.setText("Seleccione una cuenta ");
                cuentaActual=null;
                panelCantidad.setVisible(false);    }
            else
                dialogo.showMessageDialog( null,
```

```
                    "Usted NO puede extraer esa cantidad",
                "Advertencia", JOptionPane.WARNING_MESSAGE ); }
    }
}
```

El oyente de `botonSeleccionar` recupera la cuenta bancaria con código `nroCuenta` de la cartera de cuentas bancarias y la asigna a `cuentaActual`.

```
cuentaActual=cuentas.recuperarCuenta(nroCuenta);
```

Un mismo objeto va a ser referenciado desde la colección y desde el atributo de instancia `cuentaActual` de `GUI_Cajero`. De modo que cuando el oyente de uno de los botones de la botonera envía el mensaje:

```
cuentaActual.extraer(extraccion);
```

Se modifica el saldo de la cuenta en la cartera de cuentas bancarias.

La clase `GUI_Cajero` usa los servicios de `CarteraClientes` y `CuentaBancaria` sin conocer la representación de los datos ni la implementación de las operaciones.

Productividad y GUI

Una de las razones que hicieron de Java un lenguaje muy utilizado en la industria de software es porque favorece la **reusabilidad** a través del uso de **paquetes** y la **extensibilidad** a través de **herencia**. Cada paquete que se importa se usa como una caja negra, conocimiento únicamente la interface de las clases provistas. Cada clase provista por el lenguaje puede ser utilizada para crear objetos o para definir clases más especializadas, con atributos y servicios específicos de la aplicación

El lenguaje brinda muchos recursos prefabricados que contribuyen a mejorar la productividad, entre ellos una colección de clases gráficas organizadas en forma jerárquica y distribuidas en paquetes. El programador importa solo los paquetes que su aplicación requiere. Cada paquete ofrece una colección de clases que pueden ser usadas para:

- Crear objetos gráficos asociados a las componentes de la interfaz
- Definir clases más específicas a partir de las cuales se crearán componentes.

AWT y **Swing** brindan clases que pueden especializarse para crear botones, cajas de texto, menúes, etc. Una de las ventajas de Swing sobre AWT es que permite desarrollar aplicaciones con una apariencia similar a la de la plataforma subyacente con muy poco esfuerzo. Swing no reemplaza a AWT sino que la usa y agrega nuevas clases.

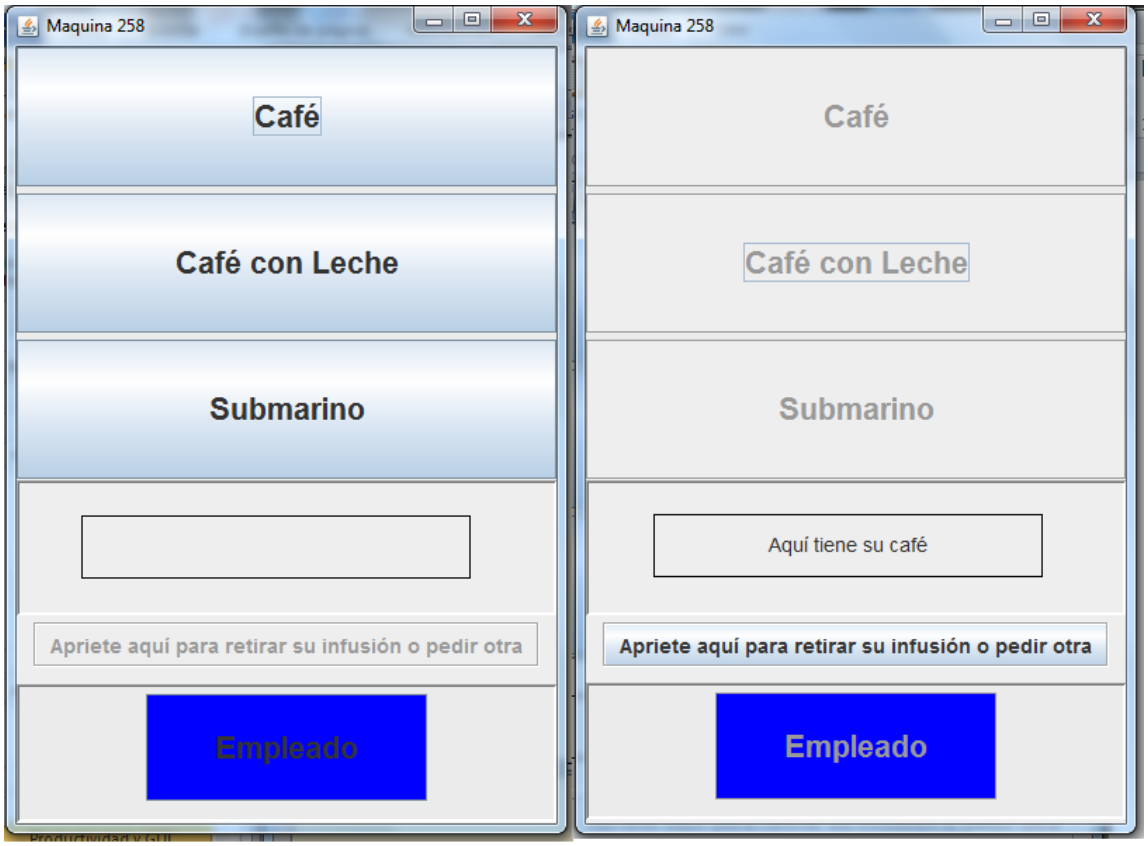
Las clases de los diferentes paquetes están vinculadas a través de relaciones de **asociación** y **herencia**. Un frame por ejemplo tiene una dimensión y tres botones, entre otras componentes. Las clases `Dimension` y `JButton` están asociadas a `JFrame`. La clase `JButton` a su vez hereda de `JComponent`.

En los ejemplos y casos de estudio propuestos se definen clases que extienden a `JFrame` para implementar GUI y se usadan algunas de las clases derivadas de `JComponent` para crear los objetos gráficos asociados a las componentes de estas GUI. En el siguiente caso de estudio se crean botones de una clase definida específicamente para la aplicación. La clase `BotonME` hereda de `JButton`.

Una instancia de `BotonME` puede recibir cualquiera de los mensajes provistos por `JButton`. La definición de esta clase permite crear botones con un diseño específico para esta aplicación.

Caso de Estudio Máquina Expendedora

Implemente una GUI para una máquina expendedora modelo M111 que brinde botones para preparar Café, Café con Leche y Submarino. Los tres botones se incluyen en un panel. Inmediatamente después de que se preparó una infusión, se habilita un botón para retirarla, los demás botones quedan deshabilitados. La información se muestra a través de una etiqueta en un panel. En el último panel el botón Empleado permite recargar la máquina y mostrar un panel de diálogo indicando las cantidades cargadas. La siguiente figura muestra la GUI inicial y luego de oprimir el botón Café:



Los paquetes requeridos para la implementación son:

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.BoxLayout;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Color;
import javax.swing.BorderFactory;
import javax.swing.border.BevelBorder;
import javax.swing.border.LineBorder;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.SwingConstants;
```

```
import javax.swing.WindowConstants;
```

Los atributos de instancia de la clase GUI_M111 son:

```
//Atributo de la Aplicacion
    private M111  me;
//Atributos Gráficos
    private Container pc;
    private JPanel panelBotones, panelES, panelRetirar,
panelEmpleado;
    private JLabel cartelBebida;
    private BotonME botonCafe, botonCafeConLeche,botonSubmarino;
    private JButton botonRetirarInfusion;
    private JButton botonEmpleado;
```

La clase `BotonMe` extiende a `JButton` estableciendo la misma apariencia para todos los botones de la GUI, excepto el rótulo del botón que varía en cada uno se especifica a través de un parámetro:

```
import java.awt.Dimension;
import java.awt.Font;
import javax.swing.*;
class BotonME extends JButton {
    public BotonME(String rotulo){
        setPreferredSize(new Dimension(306, 55));
        setFont(new Font("Arial",1,22));
        setText(rotulo);}
}
```

El constructor de la clase `GUI_M111` puede invocar a métodos internos que resuelvan cada uno de los subproblemas que plantea la implementación:

```
public GUI_M111(M111 m)  {
      super("Maquina "+m.obtenerNroSerie());
      pc = getContentPane();

      me = m;
      armarPaneles();
       armarBotones();
       armarEtiquetas();
       insertarComponentes();}
```

El código de cada método interno es:

```
private void armarPaneles() {
//Captura el panel de contenido y establece su layout
  BoxLayout layoutPc =new BoxLayout(pc,BoxLayout.Y_AXIS);
  pc.setLayout(layoutPc);
//Crea los paneles y establece su layout y apariencia
  panelBotones = new JPanel();
  GridLayout panelBotonesLayout = new GridLayout(3, 1);
  panelBotonesLayout.setHgap(5);
  panelBotonesLayout.setVgap(5);
  panelBotones.setLayout(panelBotonesLayout);
  panelBotones.setPreferredSize(new Dimension(392, 320));
  panelBotones.setSize(369, 250);
  panelBotones.setBackground(new Color(235,235,235));
  panelES = new JPanel();
```

```
  panelES.setPreferredSize(new Dimension(392, 101));
  panelES.setSize(369, 51);
  panelES.setBorder(BorderFactory.createBevelBorder
                    (BevelBorder.LOWERED));
  panelRetirar = new JPanel();
  panelRetirar.setPreferredSize(new Dimension(392, 50));
  panelEmpleado = new JPanel();
  panelEmpleado.setPreferredSize(new Dimension(392, 101));
  panelEmpleado.setSize(369, 80);
  panelEmpleado.setBackground(new Color(235,235,235));
  panelEmpleado.setBorder(BorderFactory.createBevelBorder
                          (BevelBorder.LOWERED));
}
```

Cada objeto de clase `BotonMe` es instancia de la clase `JButton` y por lo tanto puede recibir cualquier mensaje que corresponda a un método provisto por esta clase y sus clases ancestro. En el método interno armarBotones el tamaño de `botonEmpleado` de clase `BotonMe` se establece usando los métodos provistos por las clases ancestro.

```
private void armarBotones() {
//Crea el boton de cafe, crea el oyente y lo registra
      botonCafe = new BotonME("Café");
      OyenteCafe oCafe = new OyenteCafe();
      botonCafe.addActionListener(oCafe);
//Crea el boton de cafe con Leche, crea el oyente y lo registra
      botonCafeConLeche = new BotonME("Café con Leche");
      OyenteCafeConLeche oCafeConLeche = new OyenteCafeConLeche();
      botonCafeConLeche.addActionListener(oCafeConLeche);
//Crea el boton de submarino,  crea el oyente y lo registra
      botonSubmarino = new BotonME("Submarino");
      OyenteSubmarino oSubmarino = new OyenteSubmarino();
      botonSubmarino.addActionListener(oSubmarino);
//Crea el boton del Empleado,  crea el oyente y lo registra
      botonEmpleado = new BotonME("Empleado");
      botonEmpleado.setBackground(Color.BLUE);
     botonEmpleado.setPreferredSize(new Dimension(200, 76));
      botonEmpleado.setSize(180, 36);
      OyenteEmpleado oEmpleado = new OyenteEmpleado();
      botonEmpleado.addActionListener(oEmpleado);
//Crea el boton del retirar,  crea el oyente y lo registra
      botonRetirarInfusion = new JButton();
      botonRetirarInfusion.setText("Retire su infusión");
      botonRetirarInfusion.setEnabled(false);
      botonRetirarInfusion.setFont(new Font("SansSerif",1,14));
      botonRetirarInfusion.setBorder(BorderFactory.createEtchedBorde
r
                              (BevelBorder.LOWERED));
      botonRetirarInfusion.setPreferredSize(new Dimension(360, 32));
      OyenteRetirar oRetirar = new OyenteRetirar();
      botonRetirarInfusion.addActionListener(oRetirar);
}
```

Cada etiqueta tiene una apariencia estándar pero también puede configurarse de acuerdo a las pautas de diseño que se adopten.

```
private void armarEtiquetas() {
//Crea las etiquetas y establece sus atributos
```

```
      cartelBebida = new JLabel();
      cartelBebida.setLayout(new FlowLayout());
      cartelBebida.setBorder(new LineBorder
                             (new Color(0,0,0), 1, false));
      cartelBebida.setPreferredSize(new Dimension(277, 45));
      cartelBebida.setHorizontalAlignment(SwingConstants.CENTER);
      cartelBebida.setHorizontalTextPosition(SwingConstants.LEFT);
      cartelBebida.setFont(new Font("Arial",0,14));
}
private void insertarComponentes() {
//Inserta las componentes en los paneles
    panelBotones.add(botonCafe);
    panelBotones.add(botonCafeConLeche);
    panelBotones.add(botonSubmarino);
      pc.add(panelBotones);
      panelES.add(cartelBebida);
      pc.add(panelES);
      panelRetirar.add(botonRetirarInfusion);
      pc.add(panelRetirar);
      panelEmpleado.add(botonEmpleado);
      pc.add(panelEmpleado);
}
```

Los botones se habilitan y deshabilitan según se invoquen los siguientes métodos internos:

```
private void deshabilitarBotones(){
    botonCafe.setEnabled(false);
    botonCafeConLeche.setEnabled(false);
    botonSubmarino.setEnabled(false);
    botonRetirarInfusion.setEnabled(true);
    botonEmpleado.setEnabled(false);
}
private void habilitarBotones(){
    botonCafe.setEnabled(true);
    botonCafeConLeche.setEnabled(true);
    botonSubmarino.setEnabled(true);
    botonRetirarInfusion.setEnabled(false);
    botonEmpleado.setEnabled(true);
}
```

El código de las clases de los oyentes es:

```
class OyenteCafe implements ActionListener{
   public void actionPerformed(ActionEvent evt) {
//prepara el cafe si la cantidad de ingredientes lo permite
         int cantVasos = me.vasosCafe();
         if (cantVasos>=1){
           me.cafe();
            cartelBebida.setText("Aquí tiene su café");
            }
         else
           cartelBebida.setText("No puede preparar café");
        deshabilitarBotones();
        }
}
class OyenteCafeConLeche implements ActionListener {
     public void actionPerformed(ActionEvent evt) {
```

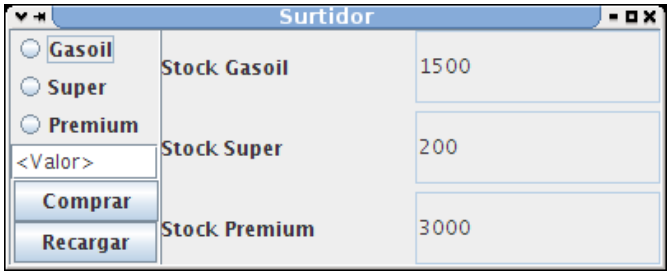
```
//prepara el cafe con leche si la cantidad de ingredientes lo
permite
          int cantVasos = me.vasosCafeConLeche();
          if (cantVasos>=1){
            me.cafeConLeche();
             cartelBebida.setText("Aquí tiene su café con leche");
            }
          else
            cartelBebida.setText("No puede preparar café con leche");
        deshabilitarBotones();
     }
}
class OyenteSubmarino implements ActionListener {
      public void actionPerformed(ActionEvent evt) {
          //Solo prepara el submarino si la cantidad de ingredientes
lo permite
          int cantVasos = me.vasosSubmarino();
          if (cantVasos>=1){
            me.submarino();
             cartelBebida.setText("Aquí tiene su submarino");
            }
          else
            cartelBebida.setText("No puede preparar submarino");
        deshabilitarBotones();}
    }
    class OyenteRetirar implements ActionListener {
       public void actionPerformed(ActionEvent evt) {
          cartelBebida.setText("");;
          habilitarBotones();}
}
class OyenteEmpleado implements ActionListener{
   public void actionPerformed(ActionEvent evt) {

      JOptionPane dialogo = new JOptionPane();
      int c = me.obtenerMaxCafe()-
me.cargarCafe(me.obtenerMaxCafe());
      int a = me.obtenerMaxCacao()-
              me.cargarCacao(me.obtenerMaxCacao());
      int l = me.obtenerMaxLeche()-
              me.cargarLeche(me.obtenerMaxLeche());
      dialogo.showMessageDialog(null,"Se cargo "+c+" grs cafe "
                  +a+" grs cacao "+l+" grs leche ",
                  "Maquina
cargada",JOptionPane.INFORMATION_MESSAGE);
          }
      }
}
```

La clase `GUI_M111` está asociada a la clase `M111` que a su vez hereda de `MaquinaExpendedora`. Las clases oyente usan los servicios provistos para los objetos de clase `M111` sin saber si estos servicios están provistos por esta clase o sus ancestros. La implementación de la GUI puede modificarse sin afectar a sus clases proveedoras y viceversa, en tanto no cambie el contrato establecido.

Problemas Propuestos

Caso de Estudio: Surtidor Combustible

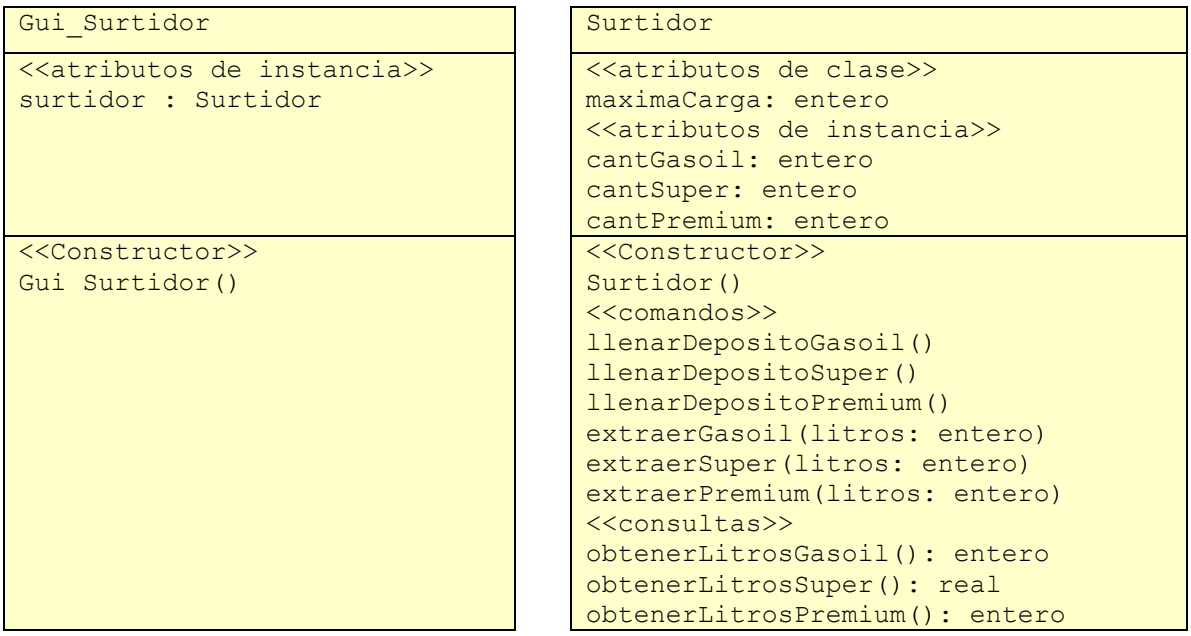
Implemente la interfaz gráfica de un surtidor de combustible como muestra la figura:



Inicialmente solo están habilitados los botones Comprar y Recargar.

- El panel derecho muestra siempre el estado actual del surtidor. Los tipos de combustible se muestran a través de etiquetas y los litros se muestran usando cuadros de texto no editables.
- Si el usuario elige Comprar se activa la selección de tipo de combustible y se desactivan Comprar y Recargar.
- Al presionar el botón Recargar se recargan todos los depósitos de combustible al máximo posible.
- Con los `JRadioButton` se elige con qué tipo de combustible se desea cargar.
- En el cuadro de texto <valor> se ingresa cuántos litros de desea comprar. Luego de ingresarse el valor se modifica el estado interno del surtidor, se habilitan los botones Comprar y Recargar y se deshabilita el resto.

El siguiente diagrama modela las clases cliente y proveedor.



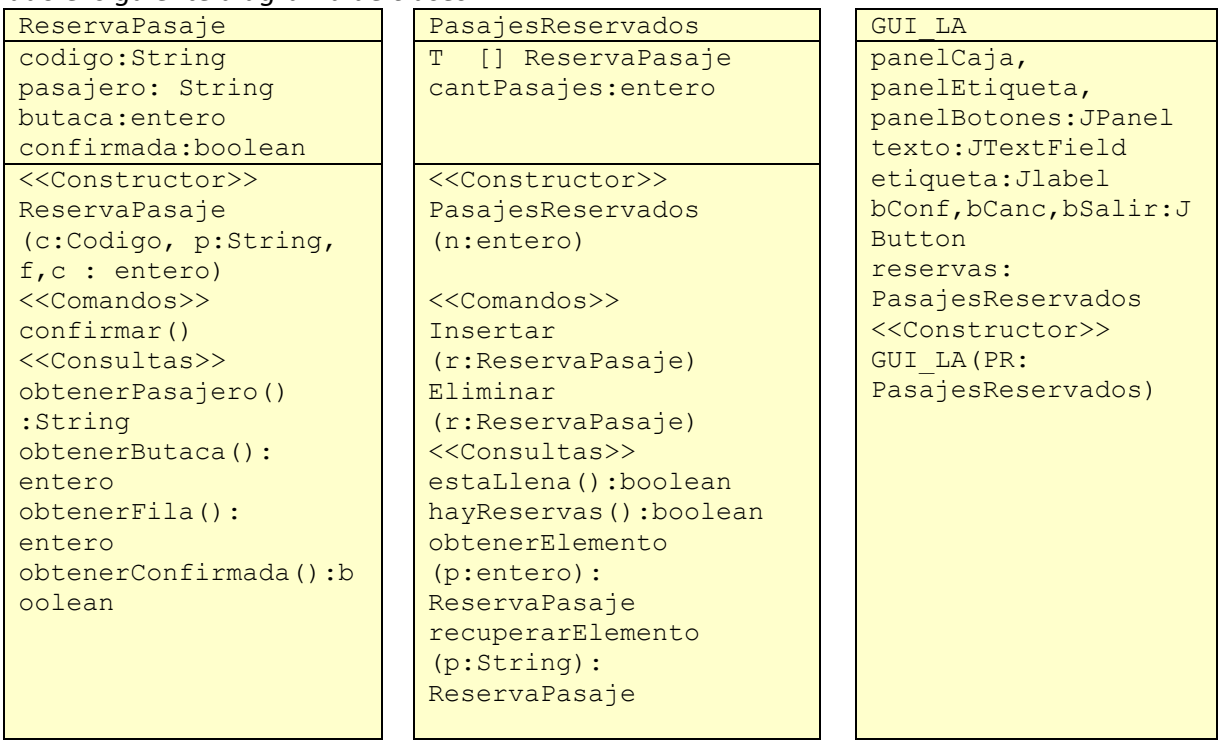
Caso de Estudio: Estación de Servicio

Una estación de servicio dispone de n surtidores representados en una clase Playa que encapsula un arreglo con n elementos de clase Surtidor.

Implemente una interface GUI_Playa con dos paneles, el panel superior muestra un objeto de clase JComboBox con los números de surtidor (1 a 6). El segundo panel tiene el mismo diagramado que en el ejercicio anterior. Inicialmente las componentes del segundo panel no están visibles, recién cuando el usuario selecciona el surtidor, se recupera el surtidor de la playa, se hace visible el segundo panel con los botones Comprar y Recargar activos. Cuando se completa la operación, se vuelve a habilitar la selección de surtidor y los componentes gráficos del segundo panel dejan de estar visibles.

Caso de Estudio: Confirmación Reserva

Dado el siguiente diagrama de clases



Implemente las dos clases y una GUI que incluya tres paneles. En el primero aparece una caja de texto, en el segundo una etiqueta y en el tercero tres botones (Confirmar, Anular y Salir) inicialmente deshabilitados.

Cuando el usuario completa el cuadro de texto con el código de una reserva y oprime enter, el oyente de la caja busca una reserva con ese código y muestra el nombre del pasajero, la fila y la butaca en la etiqueta del segundo panel. Si la reserva no está confirmada habilita los botones del tercer panel. Si la reserva está confirmada o no existe se muestran mensajes adecuados.

Si el usuario oprime el botón Confirmar, se envía el mensaje `confirmar()` a la reserva. Si oprime Anular, se elimina la reserva. Si oprime Salir, se vuelve al estado inicial.