CC3002 - Metodologías de diseño y programación

Juan-Pablo Silva and Ignacio Slater

Departamento de Ciencias de la Computación, Universidad de Chile

March 19, 2020

Contents

Introducción

Part I

Herramientas necesarias y recomendadas

Chapter 1

Git

1.1 Material adicional

Los siguientes enlaces contienen explicaciones alternativas y/o más detalladas de las herramientas disponibles para utilizar *Git*:

- **Documentación oficial de *Git***.
- **Comandos básicos de *Git***.
- **Introducción a *GitHub***: tutorial básico de cómo crear y manejar un repositorio en *GitHub*.
- **Integración de *Git* y *GitHub***: guía de los comandos básicos de *Git* y cómo se relacionan con *GitHub*.
- ***GitHub* flow:** la modalidad (flujo) de trabajo recomendada por *GitHub* para trabajar en equipos. Les recomendamos encarecidamente que utilicen esta modalidad al trabajar individualmente y, en especial, en las tareas de este ramo.
- ***Markdown*:** es la sintaxis utilizada para crear documentación en *GitHub*. Parecida a *HTML* pero más concisa (en realidad es un *superset* de *HTML*).
- ***Issues*:** es una forma de llevar una lista de *tareas* o *problemas* y *metas* en un proyecto. Suelen utilizarse mucho cuando un usuario encuentra un *bug* o solicita la implementación de un *feature* nuevo en algún proyecto.
- ***Fork*:** un *fork* es como clonar un repositorio de otra persona para trabajar paralelamente en el desarrollo de alguna funcionalidad sin cambiar el repo original. Esto es común de hacer cuando se quiere ayudar en el desarrollo de proyectos *open-source*.
- ***GitHub wikis*:** herramienta avanzada para documentar proyectos. La misma que se usa en la wiki del curso.

- **Canal de *YouTube* de *GitHub***.
- **Usar *IntelliJ* para manejar *Git***.
- **Usar *VSCode* para menejar *Git***.
- ***GitKraken*:** interfaz gráfica para manejar repositorios de *Git*. Pueden obtener una licencia gratis postulando a los beneficios de *GitHub Education*.
- ***Mercurial*:** alternativa a *Git* que también es ampliamente usada. *SourceForge* es uno de los ejemplos más importantes de *host* de repositorios de *Mercurial* (el equivalente a *GitHub*).

Part II

Objetos en Java

Chapter 2

Programación orientada a objetos

Hasta el momento, gran parte de lo que ustedes conocen es cómo escribir algoritmos en los que realizan acciones siguiendo una lógica. La programación orientada a objetos (OOP) es un **paradigma** de computación que se organiza en base a **objetos** en vez de acciones y **datos** en lugar de lógica.

Esto requiere un gran cambio de enfoque respecto a la programación imperativa tradicional que están acostumbrados a usar puesto que el enfoque estará en cuáles son los objetos que vamos a manipular en vez de la lógica para manipularlos.

2.1 Objetos

En el contexto de programación, un objeto es un elemento que tiene un comportamiento y un estado definido, comúnmente llamados métodos y campos (este último también aparece en la literatura como propiedades o variables de instancia).

El principal objetivo de utilizar objetos es poder crear estructuras para almacenar información.

Existen muchos beneficios de utilizar *OOP*, pero algunas de las propiedades más importantes son:

- **Transparencia:** La información almacenada dentro del objeto no puede ser "vista" desde afuera de éste.
- **Encapsulación:** Cada objeto maneja sus propios datos y funcionalidades.
- **Composición:** Todos los objetos pueden contener a otros (similar al concepto de composición de funciones en matemática: $f \circ g$).
- **Separación de responsabilidades:** Al utilizarse correctamente, *OOP* permite estructurar un programa en secciones, cada una respondiendo a una responsabilidad

específica, lo que añade modularidad al código.

- **Polimorfismo:** Es la capacidad de un objeto de tipo A de verse y poder utilizarse como uno de tipo B. Esto se verá en más detalle cuando se aplique *OOP* en *Java*
- **Delegación:** Cada objeto ejecuta solo las acciones que le corresponden. Si algo no le corresponde, entonces le manda un mensaje a otro (idealmente a quien si le corresponde) que lo haga. Básicamente es un "si no es mi trabajo que lo haga otro".

2.1.1 Interacciones entre objetos

Por las propiedades de transparencia y encapsulación, un objeto no puede acceder directamente a los componentes de otro, pero entonces: ¿Cómo obtengo la información almacenada dentro de un objeto?

Para interactuar con los objetos existe el concepto de *mensaje*, este concepto hace referencia a que en vez de acceder directamente a los componentes internos de un objeto, "se le pide" al objeto que realice una acción (esto se conoce como *message passing*). Luego, cada objeto decide lo que debe hacer en base al mensaje que recibe. La manera en que un objeto decide qué acción tomar con cada mensaje se conoce como *method-lookup* y se explicará más adelante.

2.2 Clases

Para introducir el concepto de clases empecemos con un ejemplo: si preparo dos tortas siguiendo exactamente la misma receta, con los mismos ingredientes y la misma preparación, entonces surge la duda, ¿son estas dos tortas la misma? La respuesta lógica es que no, son tortas individuales distintas entre ellas a pesar de que su preparación haya sido la misma.

Haciendo un símil con los objetos, podríamos decir que los ingredientes de la torta son sus propiedades y la manera de prepararla son sus métodos, entonces tendríamos dos objetos con las mismas propiedades y métodos pero distintos entre sí, aquí es cuando entran las clases. Una clase es una manera de agrupar objetos, lo que informalmente podría llamarse el *tipo del objeto*. Más formalmente, una clase es una entidad en la programación orientada a objetos a través de la cual se definen las características de todos los objetos pertenecientes al grupo definido por la clase, por esta razón un objeto particular suele entenderse como una *instancia de una clase*.

Volviendo al ejemplo de las tortas, la clase podría verse como la receta utilizada para preparar los pasteles.

La introducción del concepto de clases permite agregar otras características fundamentales de *OOP*.

2.2.1 Herencia

En programación la herencia se entiende como la *especialización* de una clase, esto se ilustra en la figura 2.1[1]

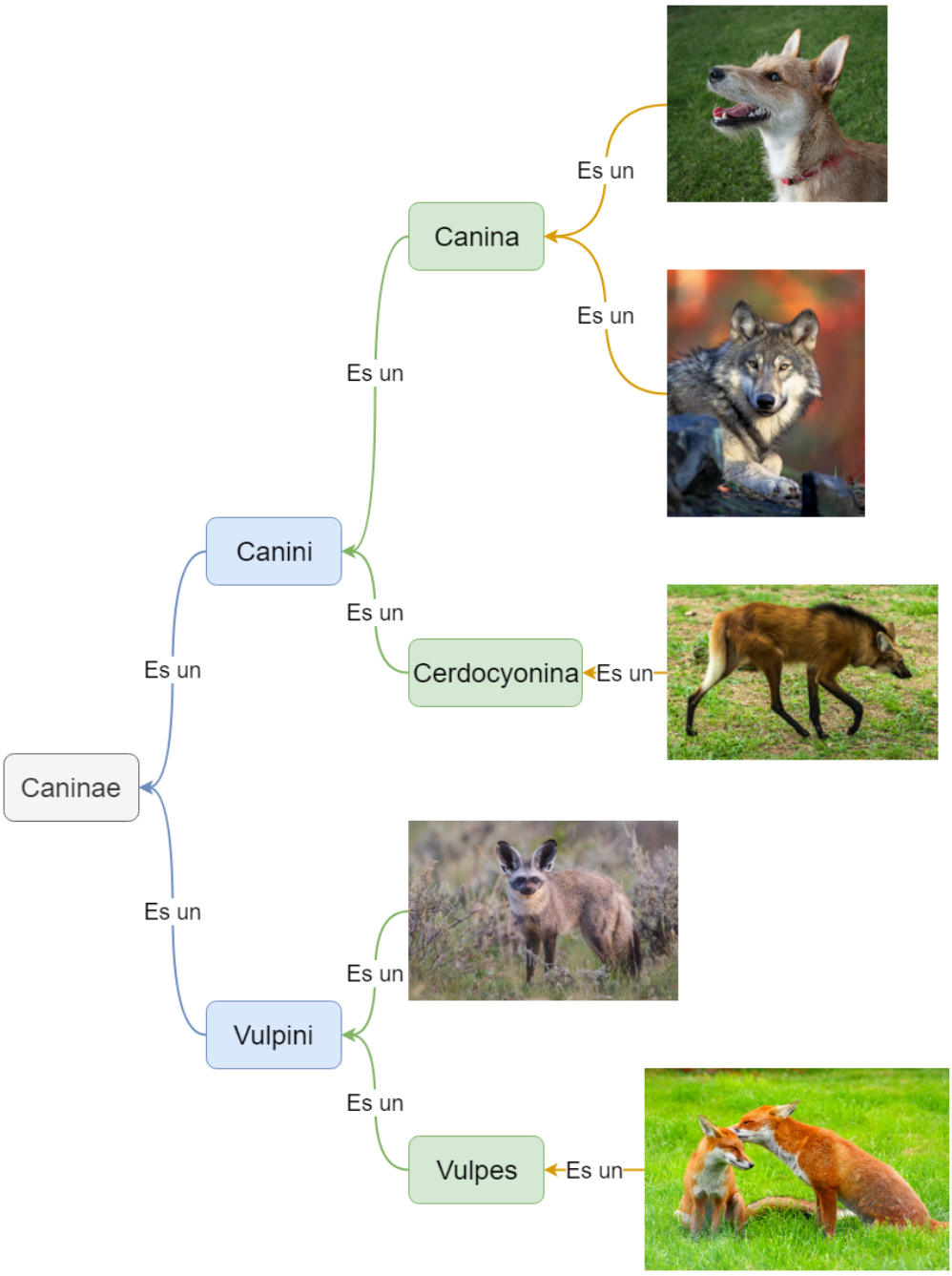

Figure 2.1: Ejemplo de herencia.

[1]Es importante resaltar que los animales no son objetos

En la figura se tiene que cada especie y subespecie es una clase, además se puede apreciar que esto permite darle una organización jerárquica a nuestras clases. Cuando hablamos de herencia en *OOP*, se suele llamar a la clase que está heredando de otra como *clase hija* o *subclase* y a la otra como *clase padre* o *superclase*.

Esta propiedad es importante no sólo por la capacidad de definir subtipos, sino que porque todas las clases hijas **heredan las funcionalidades** de la clase padre. Como esto se cumple para todas las clases de la jerarquía, se tiene que la herencia es una relación transitiva, de modo que $(\forall f,\ f \in \mathcal{C}_0 \mid \mathcal{C}_1 \subset \mathcal{C}_0 \wedge \mathcal{C}_2 \subset \mathcal{C}_1 \implies f \in \mathcal{C}_2)$

Como veremos más adelante, uno de los principales beneficios al momento de utilizar herencia dentro del contexto de programación es evitar la duplicación de código, pero es importante notar que esto es una consecuencia y no el objetivo de utilizar herencia. En un programa bien construido la herencia **debe tener coherencia lógica**, i.e. si bien tanto un avión como un pato pueden volar no tiene sentido crear una superclase para ambos porque son conceptualmente demasiado distintos entre sí.

2.2.2 Principios SOLID

Los **principios SOLID** son convenciones que se tienen en cuenta al momento de utilizar orientación a objetos y que permiten mantener una estructura robusta. Veremos más adelante que existen casos en los que se deben romper algunos de estos principios al momento de diseñar un programa para asegurar la mantenibilidad y extensibilidad del código, pero dentro de lo posible siempre se debe intentar respetar estos principios.

Single-responsibility principle

Una clase debe tener una y solamente una razón para cambiar, por lo que toda clase debe tener una sola responsabilidad.

Para entender esto, considere el siguiente problema: tenemos figuras geométricas y queremos calcular sus áreas.

Una posible solución sería crear una clase que represente una figura geométrica y que de acuerdo al tipo de figura que nos interese, entonces podríamos tener un método:
`calculateArea(String)` que se utilice como `calculateArea("Rectangle")`. Esto funcionaría, pero no respeta el principio, ya que tenemos una sola clase que se encarga de calcular el área de todas las figuras.

Una solución que sí respeta el principio es la de crear clases distintas para cada tipo de figura y que cada una de estas sepa cómo calcular su propia área (en esto se puede aprovechar la herencia).

Open-Closed principle

Los objetos o entidades deben estar abiertos para extenderse, pero cerrados para modificarse.

Este principio hace referencia a que debe ser fácil agregar funcionalidades nuevas o específicas a un programa sin necesidad de cambiar las funcionalidades y propiedades que ya tiene. Uno de los aprendizajes importantes del curso es cómo lograr cumplir con este principio, los capítulos referentes a patrones y metodologías de diseño muestran técnicas y patrones relevantes en este aspecto.

El mismo ejemplo utilizado para el principio anterior aplica aquí. Si tenemos una sola clase que calcula las áreas de todo tipo de figuras, agregar un nuevo tipo de figura implicaría modificar el método `calculateArea(String)`, mientras que si se tiene cada figura como una clase individual, agregar una nueva figura sería simplemente agregar una nueva clase.

Liskov's substitution principle

Sea $q(x)$ una propiedad demostrable para objetos x de tipo T. Entonces $q(y)$ debe ser demostrable para objetos y de tipo S donde S es un subtipo de T.[2]

En terminos más simples, esto quiere decir que las subclases siempre deben ser reemplazables por su clase padre.

Consideren el siguiente problema, queremos crear un modelo para representar aves, crearemos dos en particular: *Palomas* y *Colibríes*. Podemos agrupar estas aves en una clase *Ave* y definir que todas las aves pueden volar. Hasta aquí todo bien.

¿Pero qué pasa si ahora quiero agregar *Pingüinos*? Los pingüinos no pueden volar, pero definimos que todas las aves pueden volar. Nuestra definición inicial rompería entonces el principio de *Liskov*.

Una solución a esto sería crear una nueva clase que represente a las aves voladoras y que sea un subtipo de Ave, de esta forma, un pingüino sería un ave, mientras que una paloma sería un ave voladora.

Interface segregation principle

Un cliente nunca debiera estar forzado a implementar una interfaz que no ocupe o depender de métodos que no ocupe.

Una interfaz es una "promesa" que hace el programador con un cliente (quien use el código) en la que define cuáles son las acciones que puede realizar cualquier clase que implemente dicha interfaz. Este concepto se entenderá mejor cuando veamos aplicaciones de esto.

Tomemos como ejemplo el mismo de las aves del principio anterior. La solución original que se planteó de darle a todas las aves la habilidad de volar no rompia solamente el principio de Liskov sino que este también. Si hubieramos simplemente definido el pingüino como una subclase de ave, habríamos estado forzados a que el pingüino tuviera la capacidad de volar (aún cuando podríamos haber definido el método de tal forma que no hiciera nada).

[2] https://en.wikipedia.org/wiki/Barbara_Liskov

La solución es la misma de antes.

Dependency inversion principle

Las entidades deben depender de abstracciones y no de implementaciones. Los modulos de alto nivel no deben depender de modulos de bajo nivel, sino que deben depender de abstracciones.

Este principio suena más complicado de lo que realmente es. Lo que quiere decir es que al tener dependencias entre clases, la clase que tiene dependencia en otra no debe depender de implementaciones particulares de esta clase, sino que de una abstracción de estas implementaciones.

Usando las aves como ejemplo nuevamente, si tuvieramos una clase con un método `alimentar` para alimentarlas, la implementación de esta clase no debe depender de las implementaciones particulares, e.g. definir un método `alimentar(Paloma)` estaría violando este principio.

La manera correcta de implementar esto sin romper el principio sería crear un único método `alimentar(Ave)`

Part III

Patrones y metodologías de diseño