POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

1. Objetivos

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Introducción

- El diseño de software orientado a objetos es **difícil** y diseñar software orientado a objetos **reutilizable es incluso más difícil**
- Se debe:
  - encontrar los **objetos pertinentes**,
  - factorizarlos en clases con la **correcta granularidad**,
  - definir los **interfaces de las clases**,
  - jerarquías de **herencia** y
  - establecer las **relaciones** clave entre ellas.
- El diseño debería ser específico para el problema en cuestión pero también en general para direccionar **futuros problemas y requisitos**.
- También se quiere **evitar rediseños** o al menos minimizarlos.

1. Introducción

- Los diseñadores orientados a objetos **experimentados** te dirán que:
  - Un diseño reusable y flexible es **difícil, sino no imposible,** obtenerlo correctamente a la primera.
  - Antes de que el diseño finalice, generalmente **intentan reusarlo** varias veces modificándolo cada vez.
  - Lo que no hay que hacer es resolver cada **problema desde cero**
  - Más bien, **reúsan soluciones** que les han funcionado en el pasado. Cuando encuentran una buena solución, **la usan una y otra vez.**
- Tal experiencia es parte de lo que les hace **expertos**

2. Problemas de Diseño

- Encontrar objetos apropiados.
- Determinar la granularidad de los objetos.
- Especificar interfaces de objetos.
- Especificar la Implementación de Objetos
- Poner a funcionar los mecanismos de reutilización
- Relacionar Estructuras del Tiempo de Compilación y de Ejecución

2. Problemas de Diseño

- **Encontrar objetos apropiados.** Los patrones de diseño ayudan a identificar abstracciones menos obvias y los objetos que pueden capturarlos.
- **Determinar la granularidad de los objetos.** Los patrones de diseño describen formas específicas de descomponer un objeto en objetos más pequeños.
- **Especificar interfaces de objetos.** Los patrones de diseño ayudan a definir interfaces identificando sus elementos claves y los tipos de datos que se envían a través de una interfaz. Los patrones de diseño también especifican las relaciones entre las interfaces

2. Problemas de Diseño

- **Especificar la Implementación de Objetos**
  - **Herencia de Clases vs Interfaces.** La clase define el estado interno del objeto y la ejecución de sus operaciones. Por el contrario, el tipo de un objeto sólo se refiere a su interfaz – el conjunto de peticiones a las que puede responder. Un objeto puede tener muchos tipos y objetos de diferentes clases pueden tener el mismo tipo.
  - Desde luego, hay una estrecha relación entre clase y tipo. Cuando se dice que un objeto es una instancia de una clase, implica que el objeto soporta el interfaz definida por la clase.
  - La herencia de clases define un implementación de objetos en términos de otra implementación de objetos. En breve, es un mecanismo para compartir código y estructura. En contraste, la herencia de interfaces describe cuándo un objeto puede ser usado en lugar de otro
  - Muchos de los patrones de diseño dependen de esta distinción.

2. Problemas de Diseño

- **Especificar la Implementación de Objetos**
  - **Programación para una interfaz, no para una implementación**. Hay dos beneficios en la manipulación de objetos solamente en términos de la interfaz definida por las clases abstractas:
    - Los clientes no son conscientes de los tipos específicos de objetos que utilizan, siempre que los objetos se adhieren a la interfaz que los clientes esperan.
    - Los clientes no son conscientes de las clases que implementan estos objetos. Los clientes sólo saben de la clase abstracta de la definición de la interfaz.

2. Problemas de Diseño

- **Especificar la Implementación de Objetos**
  - **Herencia frente a Composición**. Las dos técnicas más comunes para la reutilización de funcionalidad en los sistemas orientados a objetos son herencia de clases y composición de objetos.
  - Reutilización por la sub-clasificación se refiere a menudo como la reutilización de caja blanca. El término "caja blanca" se refiere a la visibilidad: con la herencia, la parte interna de las clases padres son a menudo visibles a las subclases de objetos.
  - La composición es una alternativa a la herencia de clases. Aquí, la nueva funcionalidad se obtiene mediante el ensamblaje o composición de objetos para conseguir una funcionalidad más compleja. Este estilo de reutilización se llama reutilización de "caja negra", ya que los detalles internos de los objetos no son visibles.

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Herencia frente a Composición**. Ventajas de la herencia:
    - Es fácil de usar, ya que se apoya directamente en el lenguaje de programación.
    - También hace que sea más fácil de modificar la implementación siendo reutilizada. Cuando una subclase sobrescribe algunas, si no todas las operaciones, puede afectar a las operaciones que hereda, suponiendo que llamen las operaciones sobrescritas.
  - Desventajas de la herencia:
    - No se puede cambiar la implementación de la clase padre en tiempo de ejecución, porque la herencia se define en tiempo de compilación.
    - Se expone una subclase a los detalles de la implementación de su padre, se dice que “la herencia rompe la encapsulación". La implementación de una subclase se torna tan ligada a la implementación de su clase padre que cualquier cambio en la implementación de la padre obligará a la subclase a cambiar.

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Herencia frente a Composición**. Las dependencias de implementación pueden causar problemas al tratar de reutilizar una subclase. Cuando cualquier aspecto de la implementación heredada no sea apropiado para nuevos dominios de problemas, la clase padre deberá ser escrita de nuevo o reemplazada por otra más adecuada. Una solución a esto es heredar sólo de clases abstractas, ya que éstas normalmente tienen poca o ninguna implementación.

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Herencia frente a Composición**. La composición de objetos se define dinámicamente en tiempo de ejecución a través de objetos que tienen referencias a otros objetos. La composición requiere que los objetos tengan en cuenta las interfaces de los otros, lo que a su vez requiere interfaces cuidadosamente diseñadas que no impidan que un objeto sea utilizado por otros.
  - Pero hay una ventaja en esto: puesto que a los objetos se accede sólo a través de sus interfaces no se rompe la encapsulación. Cualquier objeto puede ser reemplazado en tiempo de ejecución por otro siempre que sean del mismo tipo. Además, como la implementación de un objeto se escribirá en términos de interfaces de objetos, las dependencias de implementación son notablemente menores

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Herencia frente a Composición**. Optar por la composición de objetos frente a la herencia de clases ayuda a mantener cada clase encapsulada y centrada en una sola tarea. De esta manera, nuestras clases y jerarquías de clases permanecerán pequeñas y será menos probable que se conviertan en monstruos inmanejables.
  - Por otro lado, un diseño basado en la composición de objetos tendrá más objetos (al tener menos clases) y el comportamiento del sistema dependerá de sus relaciones en vez de estar definido en una sola clase.

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Herencia frente a Composición**. Idealmente, sólo crearíamos nuevos componentes para lograr la reutilización. Deberíamos ser capaces de conseguir toda la funcionalidad que necesitamos simplemente ensamblando componentes existentes a través de la composición de objetos. Sin embargo, rara vez es éste el caso, puesto que el conjunto de componentes disponibles nunca es, en la práctica, lo suficientemente rico. Reutilizar mediante la herencia hace más fácil construir nuevos componentes que puedan ser combinados con los antiguos.
  - Por lo tanto, la herencia y la composición trabajan juntas.

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Delegación**. Es un modo de lograr que la composición sea tan potente para la reutilización como lo es la herencia. Con la delegación, dos son los objetos encargados de tratar una petición: un objeto receptor delega operaciones en su delegado. Esto es parecido a la forma en que las subclases envían peticiones a las clases padre. Pero, con la herencia, una operación heredada siempre se puede referir al propio objeto a través de las variables miembro *this*. Para lograr el mismo efecto con la delegación, el receptor se pasa a sí mismo al delegado, para que la operación delegada pueda referirse a él.
  - La delegación tiene un inconveniente común a otra técnicas que hacen el software más flexible mediante la composición de objetos: el software dinámico y altamente parametrizado es más difícil de entender que el estático. Hay también ineficiencias en tiempo de ejecución.

2. Problemas de Diseño

- **Poner a funcionar los mecanismos de reutilización**
  - **Parametrización**. Los tipos parametrizados nos dan una tercera forma (además de la herencia de clases y la composición de objetos) de combinar comportamiento en sistemas orientados a objetos. Muchos diseños se pueden implementar usando alguna de estas tres técnicas. Para parametrizar una rutina de ordenación según el tipo de operación que usa para comparar elementos, podríamos hacer que la comparación fuese:
    - Una operación implementada por las subclases
    - Responsabilidad de un objeto pasado a la rutina de ordenación
    - Un argumento de una plantilla que especifica el nombre de la función a llamar para comparar elementos
  - Qué enfoque es mejor depende de nuestras restricciones de diseño e implementación

2. Problemas de Diseño

- **Estructuras que relacionan Tiempo de Ejecución y Tiempo de Compilación.**
  - La estructura en tiempo de ejecución de un programa orientado a objetos suele guardar poco parecido con la estructura de su código. La estructura del código se fija en tiempo de compilación y consiste en clases con relaciones de herencia estáticas. La estructura en tiempo de ejecución de un programa consiste en redes cambiantes de objetos que se comunican entre sí. De hecho, ambas estructuras son en gran medida independientes. Tratar de entender una a partir de la otra es como tartar de enteneder el dinamismo de ecosistemas vivos a partir de la taxonomía estática de plantas y animals y viceversa.
  - Consideremos la distinción entre agregación y asociación de objetos y cómo se manifiestan esas diferencias en tiempo de ejecución y en tiempo de compilación.

2. Problemas de Diseño

- **Estructuras que relacionan Tiempo de Ejecución y Tiempo de Compilación.**
  - La asociación implica que un objeto simplemente conoce a otro. A veces, a la asociación también se le denomina relación de "uso". Los objetos así relacionados pueden pedirse operaciones entre sí, pero no son responsables unos de otros. Es una relación más débil que la agregación y representa mucho menor acoplamiento entre objetos.
  - La agregación implica que un objeto posee a otro o que es responsable de él. Normalmente decimos que un objeto tiene a otro o que un objeto es parte de otro. La agregacion implica que el objeto agregado y su propietario tienen la misma vida
  - Es fácil confundir agregación y asociación, ya que muchas veces se implementan de la misma forma. En última instancia, la asociación y la agregación quedan determinadas más por su intención que por mecanismos explícitos del lenguaje.

2. Problemas de Re-Diseño

- Crear un objeto especificando su clase explícitamente
- Dependencias de operaciones concretas
- Dependencias de plataformas hardware o software
- Dependencias de las representaciones o implementaciones de objetos
- Dependencias algorítmicas
- Fuerte acoplamiento
- Añadir funcionalidad mediante herencia
- Incapacidad para modificar las clases convenientemente

3. Problemas de Re-Diseño

- **Crear un objeto especificando su clase explícitamente**.
  - Especificar un nombre de clase al crear un objeto nos liga a una implementación concreta en vez de a una interfaz. Esto puede complicar cambios futuros.
- **Dependencias de operaciones concretas.**
  - Cuando especificamos una determinada operación, estamos ligándonos a una forma de satisfacer una petición. Evitando ligar la operaciones al código, hacemos más fácil cambiar el modo de satisfacer una petición, tanto en tiempo de compilación como en tiempo de ejecución.

3. Problemas de Re-Diseño

- **Dependencias de plataformas hardware o software**.
  - Las interfaces externas de los sistemas operativos y las interfaces de programación de aplicaciones (API) varían para las diferentes plataformas hardware o software. El software que depende de una plataforma concreta será más difícil de portar a otras plataformas. Incluso resulta difícil mantenerlo actualizado en su plataforma nativa. Por tanto, es importante diseñar nuestros sistemas de manera que se limiten sus dependencias de plataformas
- **Dependencias de las representaciones o implementaciones de objetos.**
  - Los clientes de un objeto que saben cómo se representa, se almacena, se localiza o se implementa, quizá deban ser modificados cuando cambie dicho objeto. Ocultar esta información a los clientes previene lo cambios en cascada.

3. Problemas de Re-Diseño

- **Dependencias algorítmicas.**
  - Muchas veces los algoritmos se amplían, optimizan o sustituyen por otros durante el desarrollo y posterior reutilización. Los objetos que dependen de un algoritmo tendrán que cambiar cuando éste cambie. Por tanto, aquellos algoritmos que es probable que cambien deberían estar aislados.
- **Fuerte acoplamiento.**
  - Las clases con fuerte acoplamiento son difíciles de reutilizar por separado, puesto que dependen unas de otras. Esto lleva a sistemas monolíticos, en los que no se puede cambiar o quitar una clase sin entender y cambiar muchas otras. El sistema se convierte así en algo muy denso que resulta difícil de aprender, portar y mantener. El bajo acoplamiento aumenta la probabilidad de que una clase pueda ser reutilizada ella sola y de que un sistema pueda aprenderse, portarse, extenderse …

3. Problemas de Re-Diseño

- **Añadir funcionalidad mediante herencia.**
  - Particularizar un objeto derivando de otra clase no suele ser fácil. Cada nueva clase tiene un coste de implementación (inicialización, finalización, etc.). Definir una subclase también requiere un profundo conocimiento de la clase padre. Por ejemplo, redefinir una operación puede requerir redefinir otra, o tener que llamar a una operación heredada. Además, la herencia puede conducir a una explosión de clases, ya que una simple extensión puede obligar a introducir un montón de clases nuevas.

3. Problemas de Re-Diseño

- **Añadir funcionalidad mediante herencia (cont.).**
  - La composición de objetos en general y la delegación en particular proporcionan alternativas flexibles a la herencia para combinar comportamiento. Se puede añadir nueva funcionalidad a una aplicación componiendo los objetos existentes de otra forma en vez de definir subclases nuevas de otras clases existentes. No obstante, también es cierto que un uso intensivo de la composición de objetos puede hacer que los diseños sean más difíciles de entender. Muchos patrones de diseño producen diseños en los que se puede introducir nueva funcionalidad simplemente definiendo una subclase y componiendo sus instancias con otras existentes.

3. Problemas de Re-Diseño

- **Incapacidad para modificar las clases convenientemente.**
  - A veces hay que modificar una clase que no puede ser modificada convenientemente. Quizás necesitemos el código fuente y no lo tengamos (como puede ser el caso de una biblioteca de clases comercial). O tal vez cualquier cambio requeriría modificar muchas de las subclases existentes. Los patrones de diseño ofrecen formas de modificar clases en tales circunstancias

4. Objetivos

- El propósito de este libro es registrar la experiencia en diseño de software orientado a objetos como patrones de diseño.
  - Cada patrón de diseño nombra, explica y evalúa sistemáticamente un **diseño importante y recurrente** en un sistema orientado a objetos.
  - Nuestro objetivo es capturar la experiencia de diseño de forma que la gente **pueda usarlo efectivamente**.

4. Objetivos

- Los patrones de diseño:
  - Expresan **técnicas probadas** más accesibles a los desarrolladores de nuevos sistemas.
  - Facilitan **reutilizar diseños y arquitecturas exitosos** . El bajo acoplamiento aumenta la probabilidad de que una clase de objeto puede cooperar con otros
  - Ayudan a **elegir entre las alternativas de diseño** que hacen un sistema reutilizable y a evitar alternativas que comprometen la reutilización.
  - Pueden incluso **mejorar la documentación y mantenimiento** de los sistemas existentes mediante el suministro de una especificación explícita de las interacciones de clases y objetos y su intención subyacente.
  - Ayudan a un diseñador de conseguir un **diseño correcto más rápido**.