

Programación a la defensiva

José Galaviz



¿Para qué diseñamos?

Cuenta la leyenda que en la antigüedad, en alguna ocasión el reino de Frigia carecía de rey. El oráculo consultado indicó que debía ser elegido rey el primer hombre que cruzara la puerta este de la ciudad conduciendo un carro tirado por bueyes. El hombre elegido resultó ser un tal Gordias, un labrador de la región cuya única posesión era la carreta y sus bueyes. El hombre decidió, en agradecimiento, consagrar todas sus posesiones a Zeus (bueno, el dios equivalente), por lo que ató su carreta con un indescifrable nudo al templo de Zeus. Según la leyenda quien consiguiera por fin desatar el carro conquistaría el mundo. Esto no era trivial porque el nudo tenía los cabos ocultos y nadie podía siquiera comenzar a desatarlo.

Se cuenta que, cuando Alejandro Magno pasó por la ciudad alrededor del año 333 AC tomó su espada y cortó la cuerda en algún punto, con lo que hubo cabos y el nudo se pudo deshacer. Típica solución de pensamiento lateral.

Nosotros diseñamos software que resuelve problemas, problemas complejos. El objetivo de nuestra labor de diseño es simplificar lo complejo, deshacer el nudo.



Divide et impera

El objetivo del diseño es dividir el problema en unidades más pequeñas, en diferentes abstracciones que simplifican y hacen manejable la complejidad del problema. Esto se logra sólo cuando la separación que se hace es tal que:

- Minimiza las interacciones de los diferentes tipos de cosas (bajo acoplamiento).
- Todo lo que tiene que ver con una sola cosa se mantiene junto (alta cohesión).

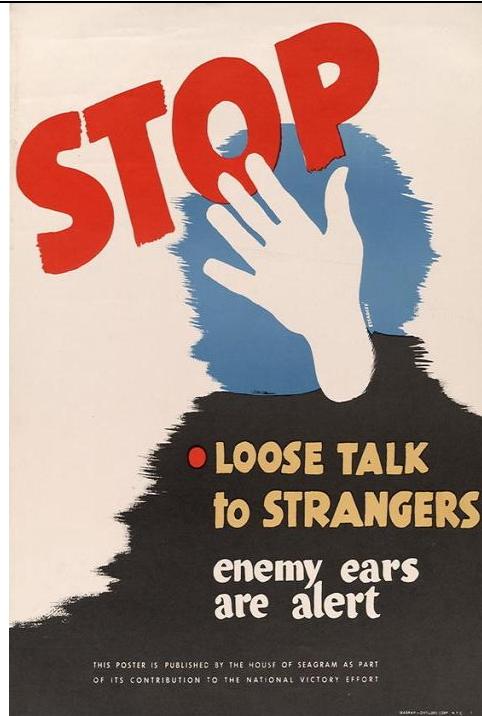
Un buen diseño...



Y es del diseño de lo que depende esencialmente la subsistencia de un sistema. A lo largo de su vida el software debe *sufrir* cambios y adecuaciones, seguramente se le deberán corregir errores. Un mal diseño *sufrirá* (literalmente) las modificaciones y terminará siendo una monstruosidad que poco tendrá que ver con el original. Un buen diseño, en cambio, siempre se salva, conservará su esencia. Para hacerlo bien debemos atenernos a una serie de principios y reglas.

Ley de Demeter

“No hables con extraños”



La ley de Demeter puede formularse diciendo que, dado un método M de un objeto O, M puede invocar:

- Métodos de O.
- Métodos de objetos que hayan sido pasados como parámetros a M.
- Métodos de cualquier objeto creado dentro de M.
- Métodos de objetos que sean atributos de O.
- Métodos de cualquier objeto accesible para O o en la visibilidad de M.

A veces se le llama “one dot law” porque el número de indirecciones en una llamada debería de ser uno. La ley de Demeter tiende a hacer más fácil el mantenimiento de un programa porque las dependencias están “estratificadas”, es decir organizadas por capas.

Don't Repeat Yourself (DRY)



Duplication Is Evil (DIE)

Otro principio que debemos tener en cuenta es el que se identifica con el acrónimo DRY (Don't Repeat Yourself) o bien con DIE (Duplication Is Evil). La intención es evitar la repetición de código previamente elaborado. Pensar en la reutilización de lo ya hecho. Si nos percatamos que debemos escribir lo mismo que ya hemos escrito es porque, seguramente, hay algo que podemos generalizar y abstraer en una clase o módulo y sólo reutilizarlo en vez de volverlo a escribir.

¿Cómo distinguir un mal diseño?

Rigidez



Un pequeño cambio acarrea muchos más

Un diseño puede ser robusto y seguro, pero poco dispuesto a las modificaciones. Si uno adapta una pequeña parte y esta modificación hace necesaria una cascada interminable de otras modificaciones en diversos lugares, se dice que existe *rigidez* en el diseño. La diapositiva ilustra la fábula del roble y el junco. El roble es fuerte y robusto, pero rígido, el viento lo rompe, mientras el junco permanece anclado y su flexibilidad lo salva del viento.

Fragilidad

**Un pequeño
cambio acarrea
múltiples fallas**



Un sistema es frágil cuando al cambiar una pequeña parte de él, se generan una secuencia de fallas aparentemente no relacionadas con el cambio. Un sistema así es prácticamente imposible de mantener. La fragilidad es, con frecuencia, un defecto que se acentúa conforme se acumulan los cambios.

Inmovilidad



Módulos excesivamente ligados al contexto

La inmovilidad es la incapacidad para reutilizar módulos en la construcción de otras aplicaciones. La situación típica es que un programador se percata de que lo que necesita es muy parecido a algo que alguien más hizo antes para otra aplicación. Toma entonces el código hecho, lo revisa y se da cuenta de que, a pesar de ser ciertamente similar la tarea que se hace, hay que modificar sustancialmente el código. Depende de muchas cosas peculiares del sistema en el que está incrustado, cambiarlo puede ser riesgoso para el sistema en el que ya funciona y sería tan laborioso que es mejor rehacerlo todo. Una rémora sólo es capaz de sobrevivir si está junto a un tiburón, sólo allí es útil.



Viscosidad

Es más fácil hacer lo incorrecto que lo correcto

Si el sistema está diseñado de tal forma que, para hacer un cambio como se debe, es decir, sin romper con el diseño general, hay que hacer un esfuerzo sustancialmente mayor que el que hay que hacer para adecuar el sistema de manera, digamos, sucia, con un *hack*, entonces el diseño no está bien. Digamos que para adecuar una clase habría que construir mucha infraestructura en vez de, simplemente hacer una subclase nueva. O que luego de hacer una modificación haya que recompilar muchas partes. En estos casos los programadores tenderán a hacer “cochinadas” que estropean el diseño pero resuelven el problema de manera inmediata: Hacen una clase que viola la encapsulación o hacen cambios que no requieran recompilar mucho.

Premisa

**Tú no puedes escribir
software perfecto, porque
eso no existe**

Pero es difícil no caer nunca en errores. Debemos, de hecho, suponer que el software que escribimos es imperfecto.

**En todo programa no trivial se
esconde, al menos...**

un *bug*

La ley de Murphy para programadores: “Todo programa no trivial tiene, al menos, un error”. Es inevitable.

Programación a la defensiva

- Las cosas que programamos son usadas por otros programadores.
- Y... ve tú a saber cómo quieran usarlas.

Además las cosas que programamos son, muchas veces, usadas por otros programadores. Nunca podemos estar seguros de que las usarán correctamente, de que pasarán los parámetros correctos, de que llamarán al método en el contexto adecuado, de que respetarán la secuencia de llamadas, Etc.

Debemos proteger nuestro software



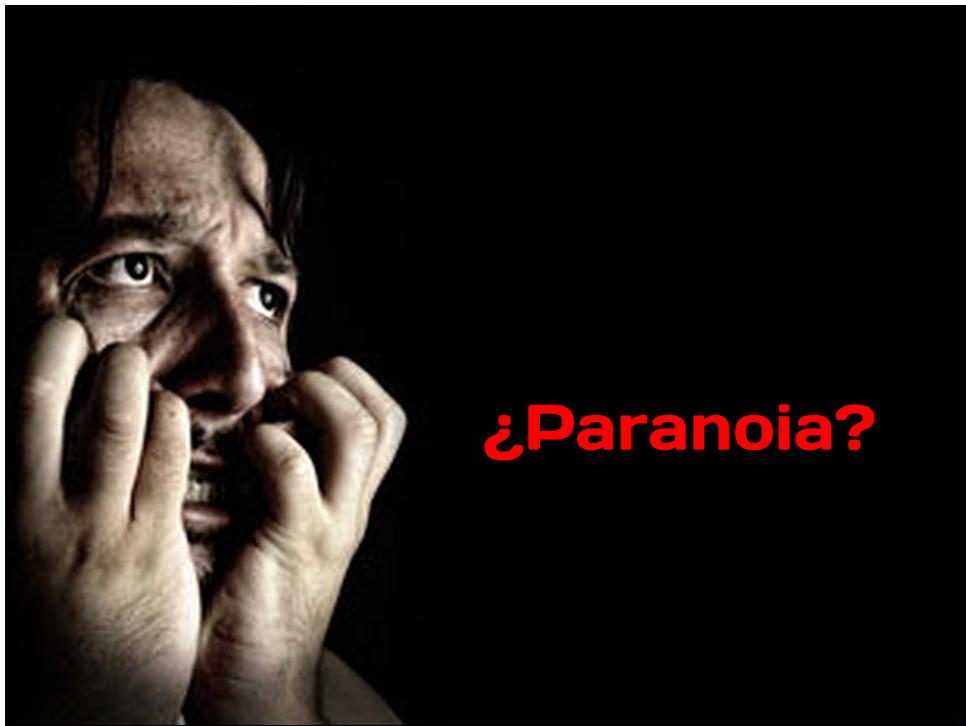
de las amenazas de externas

Así que debemos proteger nuestros módulos, programas, clases, del potencial uso incorrecto proveniente de otros programadores. En la diapositiva, escena de Los Simpson cuando Homero trata de proteger el azúcar.

Y de nosotros mismos



Pero también hay que protegerlo de nuestro propio potencial mal uso.



Suena a una especie de paranoia, a una exageración.



KEEP
CALM
AND
CONTROL
YOU DO

Pero la amenaza es real. No podemos controlar el modo en que otros programadores usarán nuestro software, pero sí podemos controlar la manera en que lo programamos. Debemos prever el mal uso, eso es **programar a la defensiva**.



No podemos predecir cómo será usado nuestro software, pero **podemos hacer predecible la manera en que este se comporta**. No podemos evitar que alguien lo use mal, pero podemos evitar que nuestro código no se percate de ello y se rompa impredeciblemente.

Cuando uno escala en roca siempre puede caer, uno se prepara para que no ocurra, pero puede ocurrir. Uno pone seguros en la pared y usa una cuerda para saber, en lo posible, qué va a ocurrir cuando uno caiga. Para no depender de la buena fortuna, para tener control, aún en medio de lo incontrolable.

Diseño por contrato



Para lograr ese cometido lo importante es definir un contrato estricto entre quien provee un servicio y quien lo utiliza. Quien provee fija sus condiciones, quien lo utiliza está consciente de ellas y decide aceptarlas.

Diseño por contrato *(Design By Contract, DBC)*

- Pre-condiciones.
- Post-condiciones.
- Invariantes de clase.

Se deben especificar los requisitos que deben satisfacer los parámetros (pre-condiciones) las garantías que el servidor provee sobre los resultados y qué condiciones deben cumplirse siempre antes y después de la ejecución de cada servicio.

Por ejemplo, en una clase que implemente una Pila. El método `tope` necesita que la Pila posea al menos un elemento (precondición). Si se cumple esta condición garantiza que regresa una referencia al último elemento en ingresar a la Pila (poscondición) y por su parte la Pila garantiza que siempre, el número de elementos en ella es igual a la diferencia entre el número de ejecuciones de `push` y el de `pop`.

Pre-condiciones

¿Qué debe ser cierto para poder llamar a la rutina?, ¿cuál debe ser el estado del mundo para que pueda trabajar?

Las precondiciones se establecen sobre los parámetros recibidos por los servicios y sobre el estado general del objeto servidor y su contexto.

Post-condiciones

¿Qué garantiza la rutina que va a hacer?,
¿cuál será el estado del mundo cuando termine?

Las postcondiciones se establecen sobre los resultados que el servicio entrega al cliente.

Invariantes de clase

¿Qué condiciones garantiza la clase que serán ciertas desde el punto de vista del usuario de ella? (antes y después de la ejecución de cada método, no durante)

Los invariantes de clase son condiciones que los objetos de la clase deben satisfacer luego de la ejecución de cualquier servicio provisto por la clase.

Verificaciones

Hay dos maneras de verificar condiciones:

- Aseveraciones (*assertions*).
- Excepciones (*exceptions*).

¿Cómo verificamos condiciones? En general hay dos esquemas diseñados para verificar el cumplimiento de algo y tomar las acciones pertinentes si no es así:

- Las aseveraciones o *assertions*.
- Las excepciones.

Aseveraciones (*assertions*)

- ¿Cuando?

Cuando tienes entero control de las variables sobre las que se hace la verificación.

- ¿Para qué?

Para detectar condiciones anómalas que **no deberían de ocurrir** (que deberían ser imposibles). Errores de la lógica del programa.

- Se deshabilitan cuando el programa pasa la etapa de pruebas.

Las aseveraciones se utilizan para verificar que no ocurran cosas que es *imposible* que ocurran, según la lógica del programa. Que el cálculo de un cociente tenga un denominador igual a cero, que en el contexto de un programa para obtener las raíces reales de una ecuación de segundo grado se pase un argumento negativo a una raíz cuadrada. Son cosas que, si ocurren, es porque hay un error en nuestro programa.

Las aseveraciones se usan para verificar las condiciones de variables que estén bajo nuestro control.

Excepciones (*exceptions*)

- ¿Cuando?

Cuando tus rutinas serán usadas por otro y entonces no tienes control sobre lo que puedes recibir.

- ¿Para qué?

Para detectar condiciones de operación anómalas que violan el contrato. Errores de uso que se pueden prever y que **pueden ocurrir**.

- Se dejan allí para siempre.

Las excepciones, en cambio sirven para verificar que no ocurran cosas que sabemos que pueden ocurrir por uso incorrecto de las cosas. Las excepciones son justo eso, condiciones excepcionales que deben ser contempladas porque pueden ocurrir y el programa debe enfrentarlas dignamente. Las excepciones se utilizan cuando las variables sobre las que se verifican las condiciones pueden no estar bajo nuestro control.

Aseveraciones Vs. excepciones

- En una aseveración uno pregunta si ha ocurrido algo que es imposible que ocurra.
- En una excepción uno pregunta si ha ocurrido algo que uno ya sabía que podía ocurrir y no es bueno.
- Aseveraciones: siempre en métodos privados, invariantes de clase, postcondiciones, etapa de pruebas.
- Excepciones: siempre en métodos públicos, en producción.

Las aseveraciones pueden deshabilitarse cuando el programa ha pasado la etapa de pruebas (suponiendo que las pruebas hechas son suficientes). Las excepciones deben dejarse en el código.

Toda rutina debe **ante todo**,
verificar precondiciones

Cada rutina debe, antes que otra cosa, verificar que sus parámetros y el estado del contexto satisfagan sus precondiciones.

Política convenenciera



Exige mucho, trabaja poco

La política de diseño más adecuada es exigir de los parámetros y contexto tanto como sea necesario. Lanzar la excepción a la mínima *provocación*, es decir, ante la mínima violación a las precondiciones. Ofrecer las mínimas garantías posibles. Este esquema, que suena muy mezquino, es para lograr rutinas lo más breves posibles, lo que redunda en que sean más depurables y fáciles de mantener.

- Pide tanto como puedas y verificalo antes.
- Niegate a trabajar si no se cumple.
- Promete tan poco como sea posible.

**Mientras menos pidas y más prometas, más código deberás hacer, probar,
depurar, mantener**

S ingle responsibility.

O pen/closed.

Liskov substitution.

I nterface Segregation.

Dependency Inversion.

El acrónimo SOLID denota una serie de buenas prácticas en el diseño.

Single Responsibility



Cada clase debe ser responsable de una única cosa.

Hay que hacer clases que sean responsables de lo mínimo, de una sola tarea. Eso minimiza el código y por tanto facilita su prueba, depuración y mantenimiento. Hace además la clase más reutilizable, ya que la mezcla de funciones es, típicamente, más rara que una sola función.

Open / Close



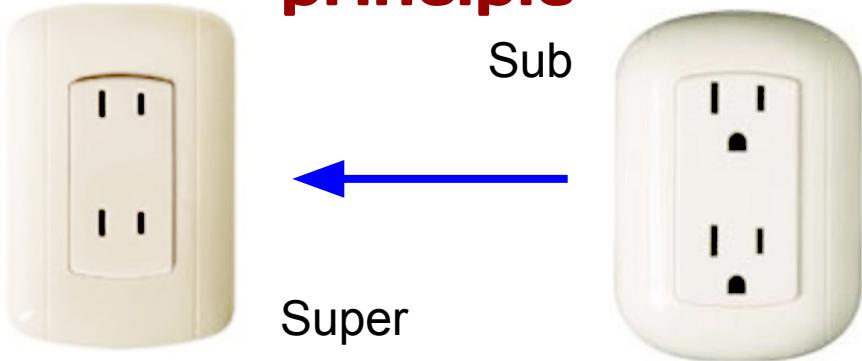
**Misma cara,
diferentes implementaciones**

En un sentido más moderno (polimórfico):

Para cada interfaz puede haber muchas diferentes implementaciones, pero la interfaz misma no debe cambiar.

Cada interfaz es cerrada a modificaciones, pero admite diferentes implementaciones que, al menos, se apeguen a ella.

Liskov substitution principle



Donde quiera que se usaba la superclase, se puede usar la subclase

- Las subclases **deben** aceptar al menos lo mismo que la superclase.
- Las subclases **deben** garantizar al menos lo mismo que la superclase.

Así que en cualquier lugar donde se usa la superclase, se puede usar la subclase.

Las subclases se adhieren al contrato firmado por sus padres

Interface segregation



Interfaz diferente para cada tipo de cliente

Ningún cliente tiene por qué depender de aquellos servicios que no usa.

Es contraproducente poseer una clase maestra que provee de una gran variedad de servicios para diferentes clases de clientes.

Añadir un servicio, significa recompilar la clase y eso significa recompilar a los demás clientes.

Es mejor tener una interfaz diferente para cada tipo de servicio/cliente.

Dependency inversion



No se debe depender de detalles concretos, sino de abstracciones

Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben depender de abstracciones.

Las abstracciones no deben depender de los detalles, los detalles deben depender de abstracciones.

Legacy code



¿Seguro que las necesidades y condiciones están vigentes?

Otra cosa que debemos tomar en cuenta es el uso de código viejo. Suele suceder que lo que necesitamos fue hecho hace tiempo por nosotros mismos o por alguien más. ¿Estamos seguros de que las circunstancias de uso y los requisitos con los que se diseñó aquel software, son aplicables a nuestro contexto actual? No es raro que usemos código que suponía una codificación ASCII o ISO 8859 y ahora necesitamos Unicode o UTF-8 o cosas peores.

Bibliotecas de terceros

¿Seguro que sabes cómo usarla?

No pretender redescubrir el hilo negro cada vez y usar bibliotecas hechas por terceros es una gran idea, en general. Pero cuando hacemos eso ¿podemos asegurar que las cosas que usaremos han sido probadas?, ¿cumplen con nuestros requisitos funcionales?, ¿están bien documentadas?, ¿seguro hacen lo que tú crees?, ¿los parámetros significan lo que estás pensando?, ¿es código seguro?

Código **inseguro**

```
int riesgo(char *input) {  
    char str[1000+1];  
    ...  
    strcpy(str, input); /* copia entrada */  
    ...  
}
```

¿Qué tal si la entrada tiene más de 1000 caracteres?

Debemos, claro está asegurarnos de que nuestro propio código es seguro. No es raro que un programa se rompa y eso pueda ser aprovechado por alguien para obtener privilegios indebidos en un sistema. El código mostrado, por ejemplo, no tiene un comportamiento predecible si la cadena `input` mide más de 1000 caracteres. Es nuestro deber asegurarnos de que esto no ocurra.

Código seguro

```
int sinriesgo(char *input) {  
    char str[1000];  
    ...  
    strncpy(str, input, sizeof(str));  
    /* si strlen(input) == sizeof(str) entonces la  
     copia no termina en NULL */  
    str[sizeof(str) - 1] = '\0';  
}
```

Por ejemplo, usando la versión de copia de cadena que limita el número de caracteres que se copian. Asegurandonos de que la cadena termina, Etc.



¡Oops!

K eep

I t

S imple

S tupid!

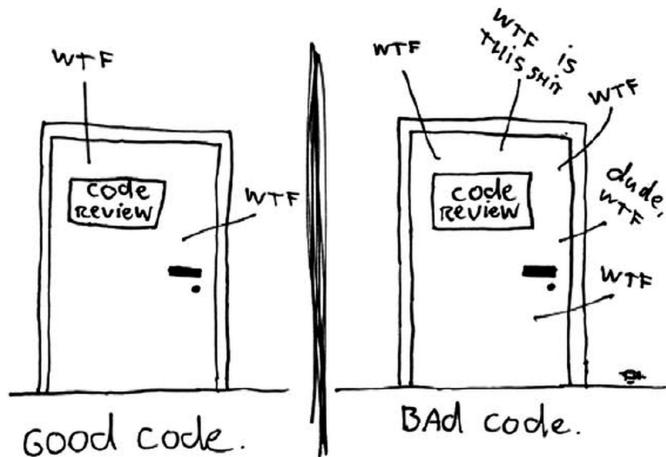


Recordemos que es fundamental mantener el código lo más legible y simple posible. Cuanto mejor hagamos esto nuestro código será más robusto, y útil. Nuestros métodos o funciones deben ser simples:

- Más de cinco parámetros piensalo bien. O los agrupas o rompes la rutina en varias.
- Sólo una actividad, función, resultado.
- No efectos colaterales.
- No uso de cosas que no recibió o que no sean atributos.
- Más de 15 líneas de código, piensalo bien.

No dejes que la paranoia resulte contraproducente complicando el código. “Ni tanto que queme al santo ni tanto que no alumbre”. Como en todo, es cuestión de equilibrio.

The ONLY valid measurement OF Code QUALITY: WTFs/MINUTE



Del libro de Robert C. Martin, *Clean Code*, Prentice Hall, 2009.

FIN

Open / Close

En el sentido original de Bertrand Mayer:

Se deben poder cambiar el comportamiento de una clase extendiendo sus servicios (*open extension*).

Pero no cambiar la manera en que esos servicios son provistos (*closed modification*).

Single Responsibility



Cada clase debe ser responsable de una única cosa.

Hay que hacer clases que sean responsables de lo mínimo, de una sola tarea. Eso minimiza el código y por tanto facilita su prueba, depuración y mantenimiento. Hace además la clase más reutilizable, ya que la mezcla de funciones es, típicamente, más rara que una sola función.