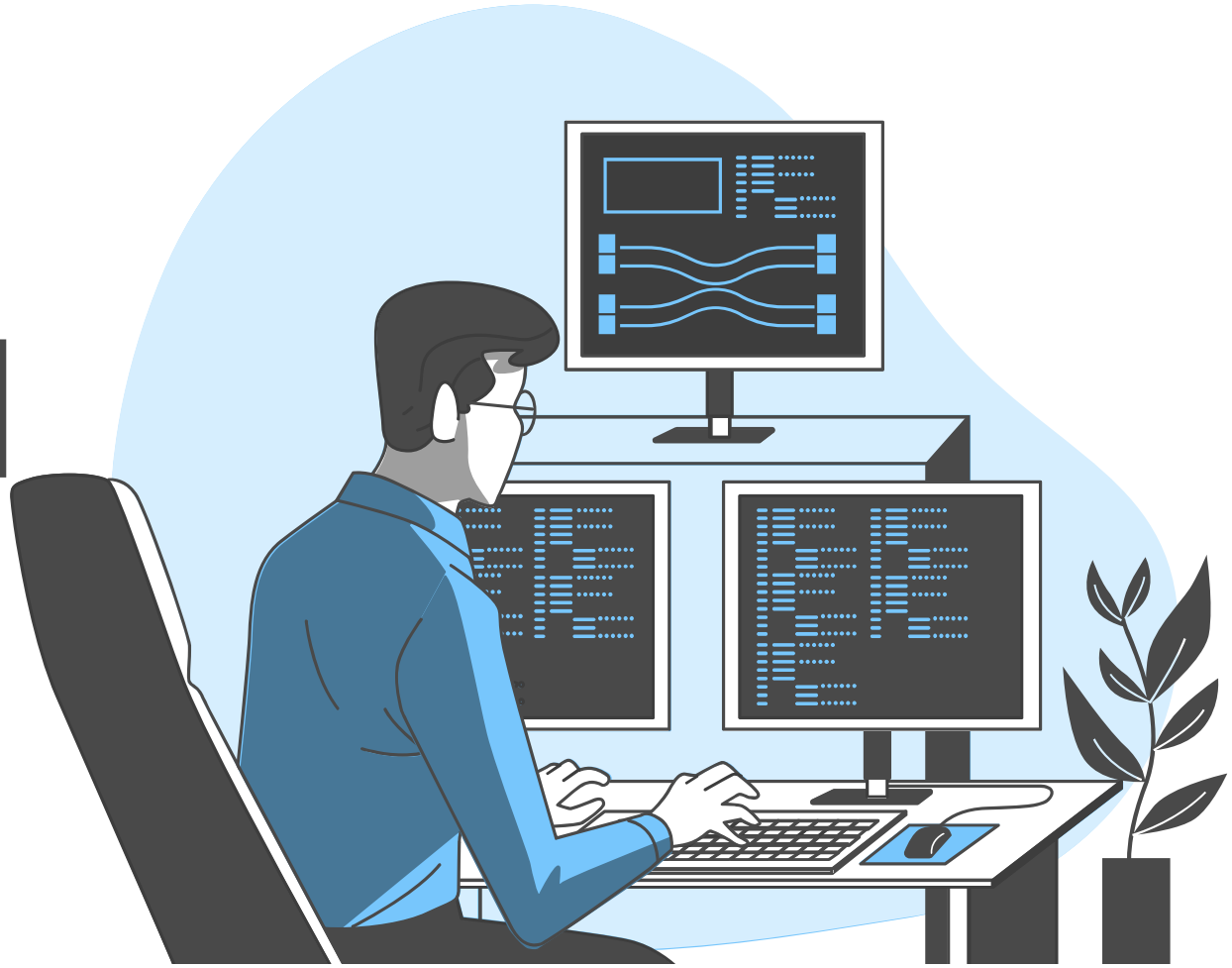
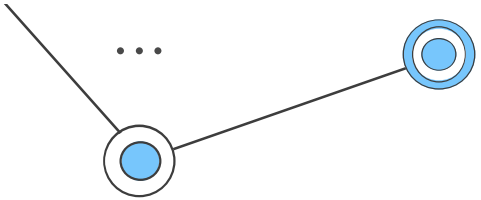
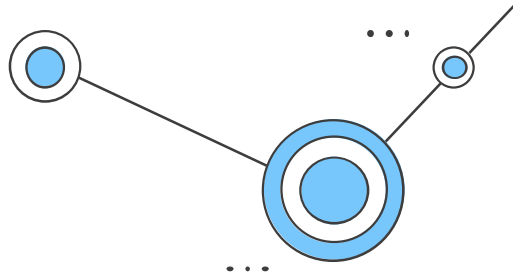
Diseño de software y modularidad

Mario García Prieto – UO279079
Pablo Fernández Díaz – UO271116





Índice

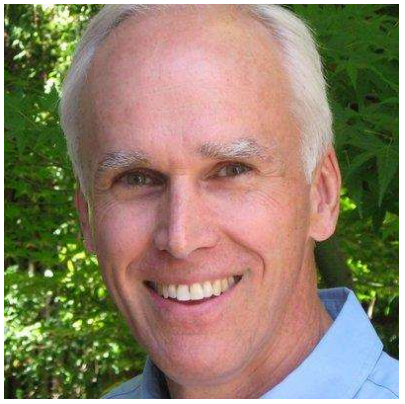


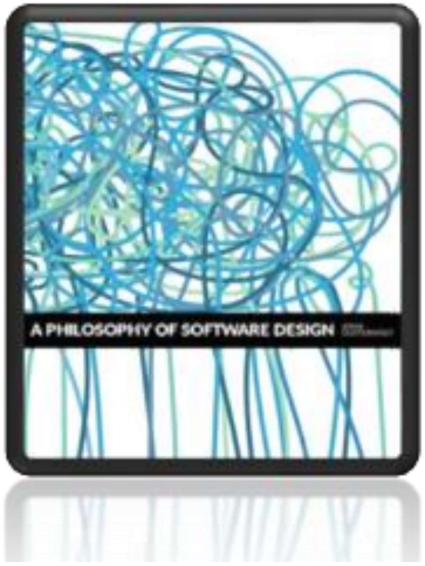
1. Biografía John Ousterhout
2. Metodologías de trabajo
3. Empatía técnica / Código empático
4. Principio del mártir
5. Desarrollo “perezoso”
6. Complejidad incremental
7. Complejidad hacia arriba
8. Complejidad hacia abajo
9. Errores y casos especiales
10. Modularidad
11. Comentarios en el código

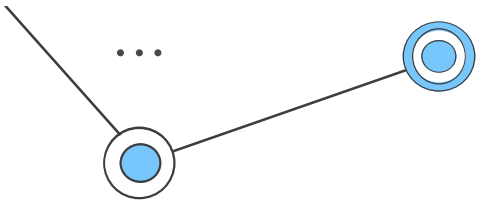
John Ousterhout

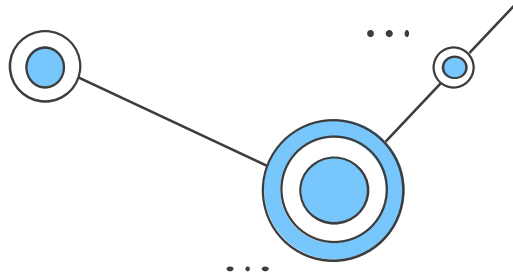
John Ousterhout nació en 1954 en EEUU y es profesor en la Universidad de Stanford.

Es el creador del lenguaje TCL y del conjunto de herramientas TK, además de esto es el autor del libro *"A Philosophy of Software Design", el cual contiene información para administrar la complejidad en los sistemas de software.*









Metodologías de trabajo

Malas prácticas

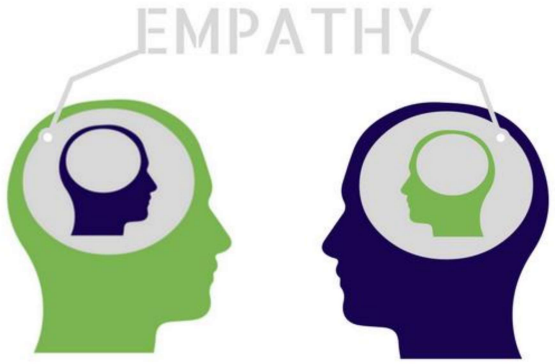
- Programar sin preocuparse por el diseño
- Desarrollo basado en pruebas
- Diseñar primero todo el proyecto y programar después

Buenas prácticas

- Diseño “iterativo”

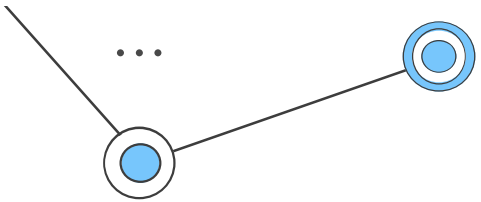
Empatía técnica / Código Empático

La empatía técnica estaría basada en realizar un buen diseño para que en un futuro no te entorpezcas a ti y mismo y a los demás.
En otras palabras, hacer un buen código para que otras personas, incluido tu mismo puedan entenderlo fácilmente

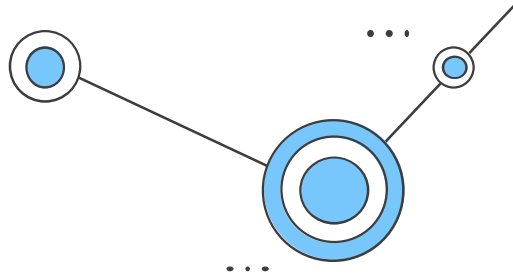


Principio del mártir

Cuando un programador o un grupo de estos hacen el sacrificio de tomar una parte muy compleja del código, ocultando la complejidad para facilitar la vida al resto de desarrolladores.



Desarrollo “perezoso”



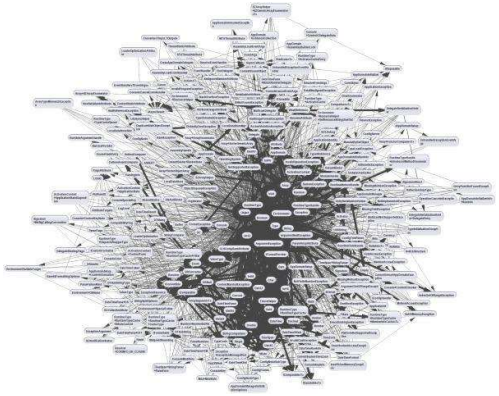
Es cuando se programa sin pensar en el futuro, lo que causará que te entorpezcas a ti mismo y otros.

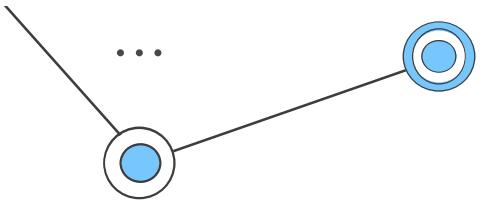
Tomando decisiones basadas en “este problema se resolverá en un futuro”

Complejidad incremental

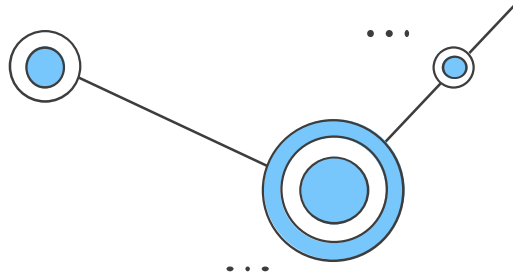
Cuando un sistema se vuelve complejo suele ser debido a pequeños fallos en el diseño que se van cometiendo de forma individual que con el paso del tiempo se van acumulando.

Posteriormente, estos errores son muy complicados de solucionar ya que están divididos en pequeños fallos y no es solamente un fallo.



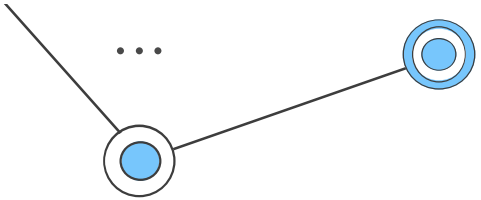


Complejidad hacia arriba

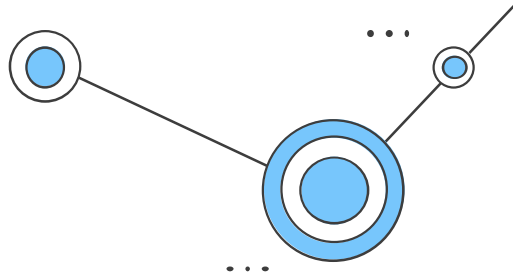


La complejidad hacia arriba consiste en no encapsular la complejidad y se propague hacia “arriba”

Un ejemplo de esta complejidad sería que los usuarios de la aplicación tuvieran que establecer parámetros de lanzamiento.

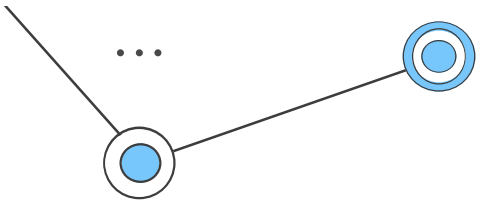


Complejidad hacia abajo

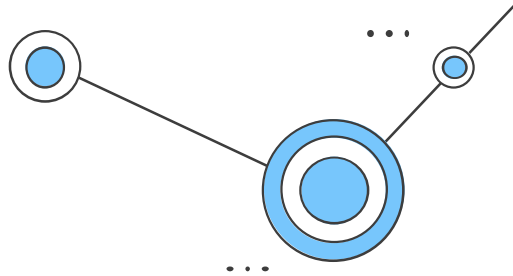


La complejidad hacia abajo consiste en encapsular la complejidad evitando que se propague hacia “arriba”

Un buen ejemplo de complicidad hacia abajo sería el principio del mártir.



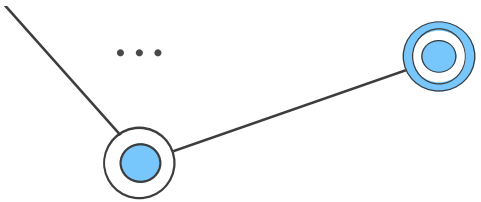
Errores y casos especiales



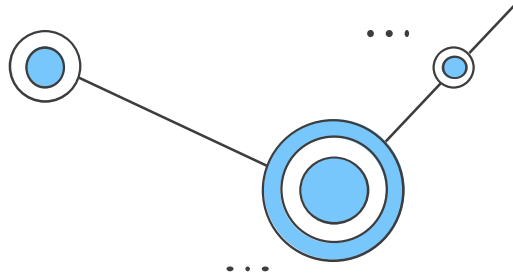
Algunas veces los errores pueden ser “solucionables” y no tienen por qué lanzar una excepción.

Hay que tener mucho cuidado en gestionar los errores y no meterlos en un try/catch e ignorarlos.

A veces es interesante mostrar los errores al usuario.

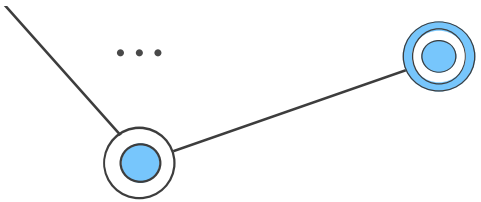


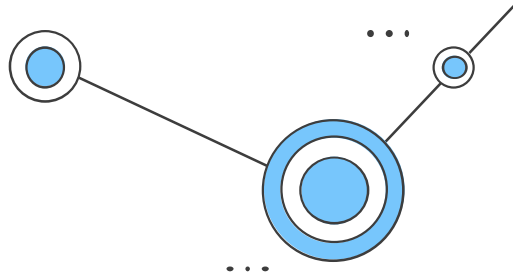
Modularidad



El diseño orientado a módulos consiste en ocultar la complejidad utilizando formas sencillas para algo que en realidad es complicado utilizando una interfaz simple para algo con mucha funcionalidad.

Lo más importante sería mostrar lo importante y ocultar lo que no.





Comentarios en el código

Malas prácticas

- No comentar nada y diseñar el código para que se entienda y no sean necesarios
- Comentar excesivamente

Buenas prácticas

- Comentar lo necesario y fundamental. Un ejemplo sería una función que calcule Fibonacci no haría falta indicarlo en un comentario

Gracias por su atención