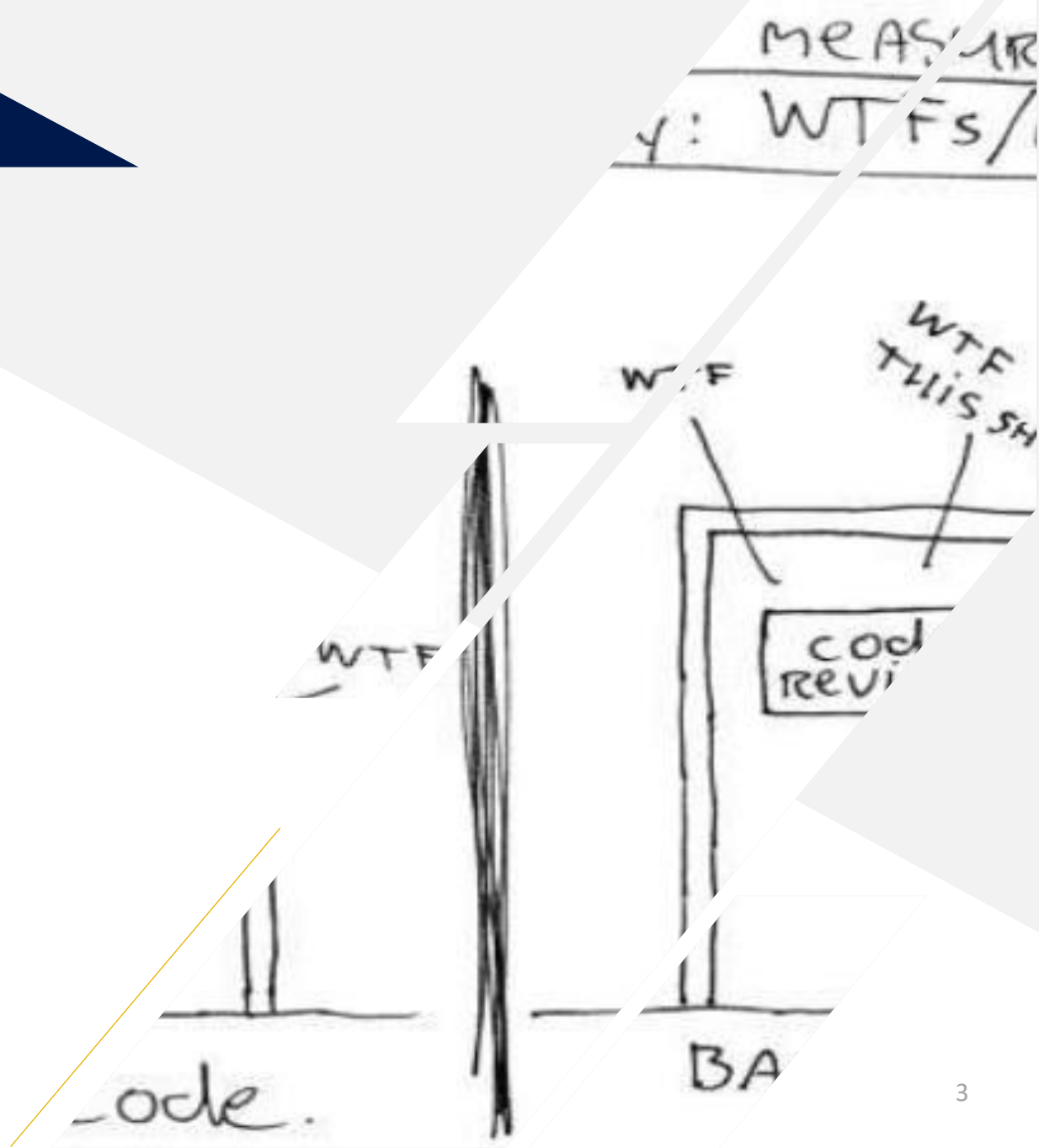
CLEAN CODE

Juan José Quesada S.

Introducción

Introducción

- La maestría es la clave para evitar errores.
- Se consigue de 2 formas: conocimiento y trabajo
- Conocimiento se adquiere de los principios, patrones, practica y heurística. Estos conocimientos se dominan a través de la práctica. Ejemplo de manejar bicicleta por primera vez, el dominio de la teoría no será suficiente



- Hágase el código:
  - La codificación no va a desaparecer ya que representan los detalles de los requisitos
- Código incorrecto
  - Cuando se escribe código incorrecto se hace una bola de nieve al tratar de corregirlo en el camino hasta que se vuelve inmanejable
  - ¿entonces porque lo escribimos?

- Coste total de un desastre
  - Ante un desastre se tiene a aumentar la planilla, aumentando el desastre pues los nuevos no conocen el diseño ni los efectos que provoca un cambio, por tanto aumenta la productividad
- El gran cambio del diseño
  - Si se hace inmanejable el equipo va a presionar para realizar un cambio de diseño y volver a desarrollar

- Actitud
  - Culpamos a los requerimientos cambiantes, a los usuarios, a los directores, sin embargo, cuando hay código incorrecto es culpa de nosotros, somos profesionales, debemos soportar la presión e informar o cuestionar cuando se nos pida desarrollar en poco tiempo o código rápido e incorrecto.
  - Si un paciente le dice al doctor que no se lave las manos para que realice el procedimiento mas rápido, por profesionalismo evidentemente el medico no lo va a hacer
- Enigma
  - Escribir código rápido e incorrecto con tal de salir a tiempo, sabiendo que esto va a ralentizar el trabajo, sin embargo, no se justifica escribir código de esta manera

- ¿El arte del código limpio?
  - No tiene sentido intentar crear código limpio si no sabe cómo hacerlo.
  - Reconocer código limpio y crearlo son cosas distintas. Para crearlo se requiere el uso disciplinado de miles de técnicas mediante un detallado sentido de la corrección.
  - El programador que crea código limpio es un artista

Concepto de código limpio

Bjarne Stroustrup, inventor de C++

- Código debe ser eficaz y elegante. La lógica debe ser directa para evitar errores ocultos, las dependencias deben ser minimas para evitar el mantenimiento, el procesamiento de errores completo y sujeto a una estrategia articulada y el rendimiento debe ser optimo para que los usuarios no tiendan a estropear el código con optimizaciones sin sentido.

Grady Booch, autor deObject Oriented Analysis and Design with Applications

- El código es simple y directo. El código se lee como un texto bien escrito. El código limpio no oculta la intención del diseñador, si no que muestra nítidas las abstracciones y líneas directas de control

Concepto de código limpio

Dave Thomas, fundador de OTI, godfather of the Eclipse strategy

- El código limpio se puede leer y mejorar por parte de un programador que no sea su autor original. Tiene pruebas de unidad y aceptación. Tiene nombres con sentido. Ofrece una y no varias formas de hacer algo. Sus dependencias son mínimas, se definen de forma explícita y ofrece una API clara y mínima. El código debe ser culto en función del lenguaje, ya que no toda la información necesaria se puede expresar de forma clara en el código.

Michael Feathers, autor de Working Effectively with Legacy Code

- Podría enumerar todas las cualidades del código limpio, pero hay una principal que engloba a todas ellas. El código limpio siempre parece que ha sido escrito por alguien a quien le importa. No hay nada evidente que hacer para mejorarlo. El autor del código pensó en todos los aspectos posibles y si intentamos imaginar alguna mejora, volvemos al punto de partida y sólo nos queda disfrutar del código que alguien a quien le importa realmente nos ha proporcionado.

Concepto de código limpio

Ron Jeffries autos de Extreme Programming Installed y Extreme Programming Adventures in C#

- En los últimos años comencé y prácticamente terminé con las mismas reglas de código simple de Beck: ejecuta todas las pruebas. No contiene duplicados, expresa todos los conceptos del diseño del sistema, minimiza el numero de entidades como clases, métodos y similares.

Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader.

- Sabemos que estamos trabajando con código limpio cuando cada rutina que leemos resulta ser lo que esperábamos. Se puede denominar código atractivo cuando el código hace que parezca el lenguaje se a creado para el problema en cuestión.

- Escuelas de pensamiento
  - Se presentan pensamientos en forma absoluta, por la experiencia de los autores.
  - Similares a los artes marciales, no hay un arte mejor que los demás.
  - Ninguna de las escuelas tiene razón absoluta, pero dentro de cada uno actuamos como si las enseñanzas y las técnicas fueran correctas.
- Somos autores
  - Escribimos código, por ende, somos autores.
  - Si somos autores tenemos lectores y ellos juzgaran nuestro trabajo.
  - Si el código es más fácil de leer es más fácil de escribir.

- La regla del boy scout
  - El código se corrompe con el tiempo.
  - Los boy scout tiene una regla sencilla que podemos aplicar al código: dejar el campamento mas limpio de como se encontró.
  - Aplicando esta regla el código se corromperá menos.

Nombres con sentido

Reglas para crear nombre correctos

Usar nombre que revelen intenciones

- Los nombres deben reponder a una serie de cuestiones básicas:
  - Porque existe, que hace y como se usa
- Si necesita comentario no revela su cometido

X	+
int d; // elapsed time in days	int elapsedTimeInDays; int daysSinceCreation;

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
      if (x[0] == 4)
        list1.add(x);
    return list1;
}
```

```
public final static int STATUS_VALUE = 0;
public final static int FLAGGED = 4;

public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
      if (cell[STATUS_VALUE] == FLAGGED)
        flaggedCells.add(cell);
          return flaggedCells;
}
```

Evitar la desinformación

- Evitar dejar pistas falsas que dificulten el significado del código.
- Evitar dejar palabras cuyo significado se aleje del que pretendemos, ejemplo: hp, unix, etc.
- Use list (accountList) solo cuando realmente sea una lista, mejor usar accountGroup.
- Evite usar nombre con variación mínima: XYZControllerForEfficientHanddling / XYZControllerForEfficientStorage
- El uso de ortografía incoherente es desinformación ejemplo: l ó o minúscula confunde con 0 y 1

Realizar distinción con sentido

- Los programadores se crean un problema al crear código únicamente dirigido a un compilador.
- Los nombres de series numéricas (a1, a2, …aN) no desinforman, simplemente no ofrecen información.
- Las palabras adicionales son otra distinicion sin sentido, son redundates, ejemplo ProductInfo, ProductData, Table a nombres de tabla, Variable a nombres de variables.

Usar nombres que se pueden pronunciar

- Cree nombres pronunciables. si no lo puede pronunciar, no podrá explicarlo sin parecer tonto.

X	+

`class DtaRcrd102 {`
`  private Date genymdhms;`
`  private Date modymdhms;`
`  private final String pszqint = "102";`
`};`
	`class Customer {`
	`  private Date generationTimestamp;`
	`  private Date modificationTimestamp;`
	`  private final String recordId = "102";`
	`};`

Usar nombres que se puedan buscar

- Los nombres de una letra y las constantes numéricas no son fáciles de localizar en el texto.
- Preferible nombres de una letra que no se puedan usar como variables dentro de métodos breves.

Evitar codificaciones

- Al codificar información de tipos o ámbitos en un nombre se dificulta la descodificación.
- No parece razonable que todos los nuevos empleados tengan que aprender otro lenguaje de codificación además del código con el que van a trabajar.
- Los nombres codificados resultan impronunciables y suelen escribirse de forma incorrecta.

Prefijos de miembros

- No es necesario añadir m_ como prefijo a los nombres de variables.
- Los usuarios aprenden rápidamente a ignorar el prefijo (o sufijo) y fijarse en la parte con sentido del nombre

X	+
`private String m_dsc;`	`private String descripcion;`

Interfaces e implementaciones

- La I inicial tan habitual en los archivos de legado actuales es, el mejor de los casos, una distracción, y en el peor, un exceso de información.
- Se recomienda codificar la implementación en lugar de la interfaz: ShapeFactoryImp.

Evitar asignaciones mentales

- Los lectores no tiene que traducir mentalmente los nombres en otros que ya conocen.
- Es un problema en los nombres de variable de una letra.
- Un contador de bucles se podría bautizar como i,j,k (pero nunca l) si su ámbito es muy reducido y no hay conflictos con otros nombres.
- Una diferencia entre un programador inteligente y un profesional es que este ultimo sabe que la claridad es lo que importa.

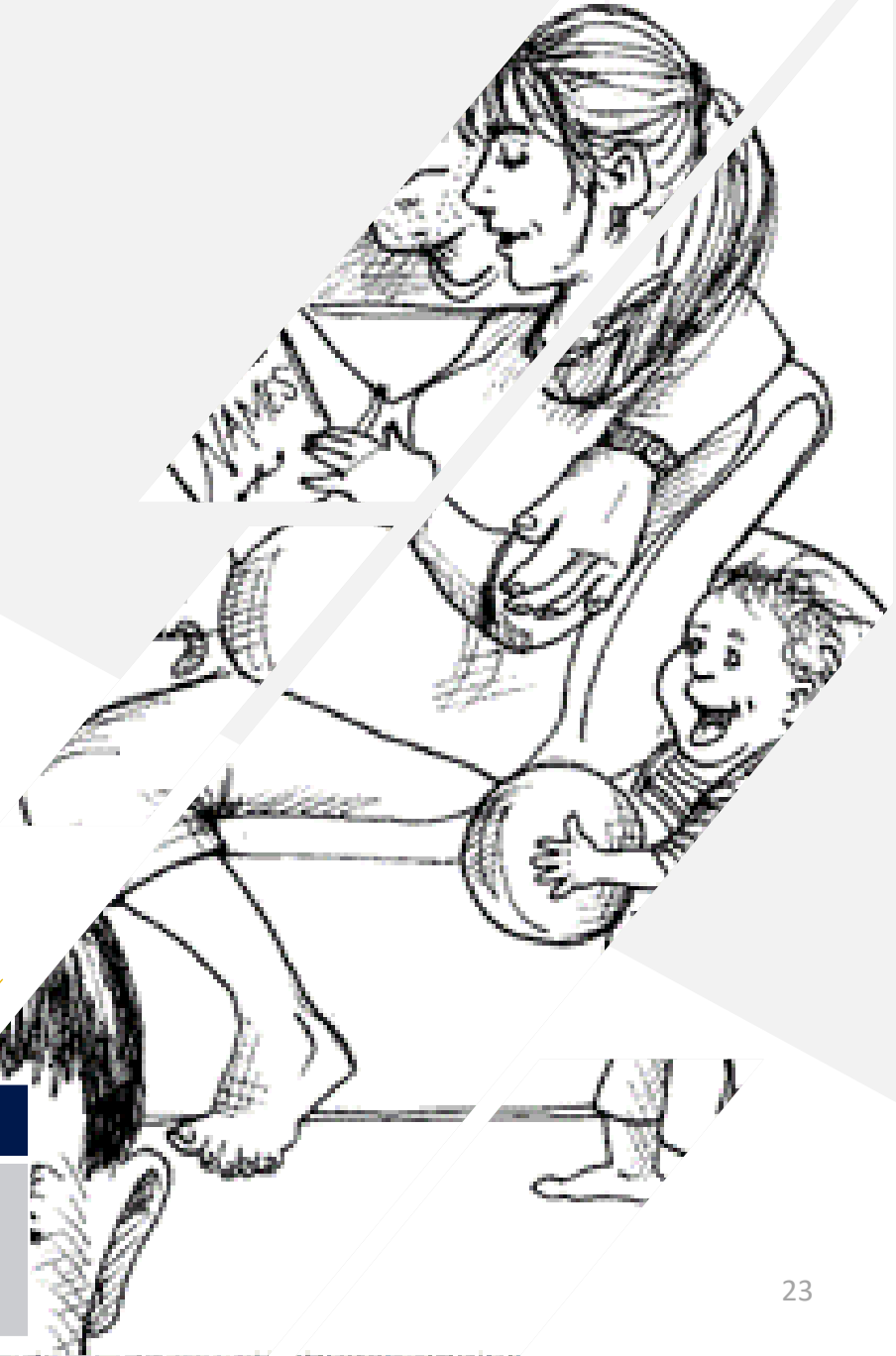
Nombres de clases

- El nombre de clase no debe ser verbo, usar sustantivo (entidad), ej.: Customer, Account, etc.

Nombres de metodos

- Los métodos deben tener nombres de verbo: postPayment. Los accesores o modificadores y predicados deben tener como nombre su valor y usar prefijo: set, get e is.
- Al sobrecargar constructores, use métodos de factoría estáticos con nombres que describan los argumentos:

X	+
`Complex fulcrumPint = new Complex(23.0);`	`Complex fulcrumPint = Complex.FromRealNumber(23.0);`



No se exceda con el atractivo

- Opte por la claridad antes que por el entretenimiento. En el código, el atractivo suele aparecer como formas coloquiales o jergas. Ej.: no user whack() en vez de kill();

Una palabra por concepto

- Elija una palabra por cada concepto abstracto y manténgala. Ej.: resulta confuso usar fetch, retrieve y get como métodos equivalentes en clases distintas.
- Los nombres de funciones deben ser independientes y coherentes para que pueda elegir el método correcto sin necesidad de búsquedas adicionales.
- Un léxico coherente es una gran ventaja para los programadores que tengan que usar su código.

No haga juegos de palabras

- Evite usar la misma palabra con dos fines distintos ej.: método add en diferentes clases que hacen cosas distintas.
- Queremos usar un modelo en el que el autor sea el responsable de transmitir el significado, no un modelo académico que exija investigar el significado mostrado

Usar nombres de dominios de soluciones

- Recuerde que los lectores de su código serán programadores.
- Use términos informáticos, algoritmos, patrones, etc.
- No conviene extraer todos los nombres del dominio del problema ya que no queremos que nuestros colegas tengan que preguntar el significado de cada nombre.

Usar nombres de dominios de problemas

- Cuando no haya termino de programación, use nombre del dominio del problema.

Añadir contexto con sentido

- Algunos nombres tienen significado por si mismos, pero la mayoría no, por ello debe incluirlos en un contexto, en clases, funciones, espacios de nombres con nombres adecuados.
- Cuando todo los demás falle, use prefijos como ultimo recurso.
- Imagine que las variables: nombre, apellido, calle, ciudad y estado, forman una dirección. Si estado se usa de forma asilada en un método ¿sabría que forma parte de una dirección. En este caso se justifica: dirNombre, dirApellido, dirEstado…
- Al menos sabran que forma parte de una estructura mayor

No añadir contextos innecesarios

- No usar prefijo para nombres de todas las clases, a la hora de invocar una clase al digitar los letras tendrá una lista grande de clases.
- Los nombres breves suelen ser mas adecuados que los extensos, siempre que sean claros.
- No añada mas contexto del necesario a un nombre.

Funciones

Son la primera línea organizativa en cualquier programa

Tamaño Reducido

- La primera regla en las funciones es que debe ser de tamaño reducido.
- Deben ser obvias (su nombre describa lo que hace)
- Deben contar una historias y cada una debe llevar a la siguiente en un orden atractivo

Bloques y sangrado

- Implica que los bloque: if, else, while y similares deben tener una línea de longitud, que seguramente, sea la invocación de una función. De esta forma no solo se reduce el tamaño de la función, sino que también se añade un valor documental, por el valor descriptivo de la función.
- También implica que las funciones no tenga estructuras anidadas.
- Por lo tanto el nivel de sangrado de una función no debe ser mayor de uno o dos

Hacer una cosa

- Las funciones solo deben hacer una cosa. Deben hacerlo bien y debe ser lo único que hagan.
- Si solo realiza los pasos situados un nivel por debajo del nombre entonces solo hace una cosa.
- Para saber si una funciona hace mas de una cosa, se extrae otra función de la misma con un nombre que no sea una reducción de su implementación.

Secciones en funciones

- Hay funciones que tiene secciones tales como: declaraciones, inicializaciones y filtros. Síntoma evidente que hacen mas de una cosa
- Las funciones que hacen una sola cosa no se pueden dividir en secciones.

Un nivel de abstracción por función

- Para que las funciones realicen una sola cosa, asegúrese que las instrucciones de la función se encuentran en el mismo nivel de abstracción.
- La mezcla de niveles de abstracción en una función resulta confusa. Los lectores no sabrán si una determinada expresión es un concepto esencial o un detalle.

Leer código de arriba abajo: la regla descendente

- El objetivo es que el código se lea como un texto de arriba abajo.
- Que todas las funciones aparezcan en el siguiente nivel de abstracción para poder leer el programa, descendiendo un nivel de abstracción por vez mientras leemos la lista de funciones.
- La clave es reducir la longitud de las funciones y que solo hagan una cosa.
- Al conseguir esto se mantiene la coherencia de niveles de abstracción.

Instrucciones Switch

¿Observa algún problema en este código?

```
public Money calculatePay(Employee e) throws InvalidEmployeeType {
  switch (e.type) {
    case COMMISSIONED:
      return calculateCommissionedPay(e);
    case HOURLY:
      return calculateHourlyPay(e);
    case SALARIED:
      return calculateSalariedPay(e);
    default:
      throw new InvalidEmployeeType(e.type);
  }
}
```

Instrucciones Switch

- Por su naturaleza las instrucciones switch hacen N cosas. No siempre podemos evitarlas, pero podemos asegurarnos de incluirlas en una clase de nivel inferior y de no repetirlas, para ello recurriremos al polimorfismo.
- Esta funciones tiene varios problemas: es de gran tamaño, si se agrega nuevos tipos aumentara (Incumple el principio open/close) y hace mas de una cosa.
- El peor de los problemas es que hay un numero ilimitado de funciones que tienen la misma estructura. Por ejemplo isPayDay(Employe e ,Date d) o deliverPay(Employe e, Money pay) usarían el mismo switch

Instrucciones Switch

- La solución consiste en ocultar la instrucción switch en una factoría abstracta e impedir que nadie la vea.
- La factoría usa la instrucción switch para crear instancias adecuadas de los derivados de Employee y las distintas funciones, como calculatePay, isPayDay y deliverPay se entregaran de forma polimórfica atreves de la interfaz Employee.

```
public abstract class Employee {
   public abstract    boolean isPayday();
   public abstract    Money calculatePay();
   public abstract    void deliverPay(Money pay);
}
----------------
public interface EmployeeFactory {
   public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
   public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
      switch (r.type) {
         case COMMISSIONED:
           return new CommissionedEmployee(r);
         case HOURLY:
           return new HourlyEmployee(r);
         case SALARIED:
           return new SalariedEmploye(r);
         default:
           throw new InvalidEmployeeType(r.type);
      }
   }
}
```

Usar nombres descriptivos

- Cuanto mas reducida y concreta sea una función, mas sencillo será elegir un nombre descriptivo.
- Un nombre descriptivo extenso es mejor que uno corto breve pero enigmático.
- No tema dedicar tiempo a elegir un buen nombre.
- La elección de los nombres descriptivos clarifica el diseño de los módulos y le permite mejorarlos.
- Sea coherente con los nombre. Use las mismas frases, sustantivos y verbos en los nombres de función que elija para los módulos.

Argumentos de funciones

- Cero es el numero ideal de argumentos. Después uno (monadico) y dos (dialico). Evitar tres (triadico).
- Los argumentos son complejos y que requieren un gran poder conceptual.
- Son mas complicados desde el punto de vista de pruebas. Imagine la dificultad de crear todos los casos de pruebas para garantizar el funcionamiento de las distintas combinaciones de argumentos.
- No esperemos que la información se devuelva a traves de los argumento. Por ello, los argumentos de salida suelen obligarnos a realizar una comprobación doble.

Formas monádicas habituales

- Hay 2 motivos para pasar un solo argumento a una función. Puede que se realice una pregunta sobre el argumento, como en Boolean fileExits(“MyFile”), o que se procese el argumento , lo transforme en otra cosa y lo devuelva.
- Una forma menos habitual pero muy útil para un argumento es un evento. En esta forma, hay argumento de entrada pero no de salida

Argumentos de indicador

- Los argumentos de indicador son horribles. Pasar un valor Booleano a una función es una práctica totalmente desaconsejable.
- Complica inmediatamente la firma del método e indica que la función hace mas de una cosa.
- Se puede dividir la función.

Funciones diádicas

- Este tipo de funciones son mas difíciles de entender que las monádicas. Por ejemplo writeField(name) es mas fácil de entender que writeField(outputStream, name). Aunque en ambos casos el significado es evidente, la primera se capta mejor visualmente.
- Las partes que ignoramos son las que esconden los errores. Pero en ocasiones se necesitan 2 argumento, ejemplo: Point p = new Point(0,0).
- Las combinaciones diádicas no son el mal en persona y tendrá que usarlas. Sin embargo recuerde que tienen un precio y que debe aprovechar los mecanismos disponibles para convertirlas en unitarias. Por ejemplo podría convertir outputStream en una variable miembro de la clase.

Triadas

- Son mucho mas difíciles de entender que las funciones que reciben 2 argumentos. Los problemas a la hora de ordenar, ignorar o detenerse en los argumentos se duplican.
- Por ejemplo assertEquals que acepta 3 argumentos message, expected y actual. ¿Cuánta veces lee el mensaje y piensa que es lo esperado?

Objeto de argumento

- Cuando una función parece necesitar 2 o mas argumentos, es probable que alguno de ellos se incluya en una clase propia. Fíjese la diferencia entre las siguientes declaraciones:
  - Circle makeCircle(double x, doble y, doblue radious)
  - Circle makeCircle(Point center, doblue radious)
- Puede parecer una trampa pero no lo es, es mas entendible.

Listas de argumentos

- En ocasiones tendremos que pasar un numero de variable de argumentos a una función.
  - String.Format(“%s worked”, name, hours)
- Si los argumentos variables se procesan de la misma forma, como en el ejemplo anterior, serán equivalentes a un único argumento de tipo List.
  - Public String format(String format, Object… args)

Verbos y palabras claves

- La selección de nombres correctos para una función mejora la explicación de su cometido así como el orden y el cometido de los argumentos. En formato monádico la función y el argumento deben formar un par de verbo y sustantivo.
- writeField(name) nos dice que name es un campo (field), este es un ejemplo de palabra clave como nombre de función.
- assertEquals se podría haber escrito como assertEqualsActual(expected, actual), lo que mitiga el problema de tener que recordar el orden de los argumentos.

Sin efectos secundarios

- Los efectos secundarios son mentiras. Su función promete hacer una cosa, pero también hace otras ocultas. En ocasiones realiza cambios inesperados en las variables de su propia clase. En ocasiones las convierte en las variables pasadas a la función o elementos globales del sistema.
- El efecto secundario es la invocación de Session.initialize(). La función checkPassword, por su nombre, afirma comprobar la contraseña. El nombre no implica que inicialice la sesión. Por tanto un invocador que se crea lo que dice el nombre de la función se arriesga a borrar los datos de sesión actuales cuando decida comprobar la validez del usuario. Este efecto secundario genera una combinación temporal.

Have No Side Effects

```
public class UserValidator {
  private Cryptographer cryptographer;
  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true; }
    }
    return false;
  }
}
```

Argumentos de salida

- appendFooter(s)
- ¿esta función añade s al final de algo? ¿o añade el final de algo a s? lo sabremos al ver la firma de la función:
- Public void appendFooter(StringBuffer report)
- Esto aclara todo pero para ello hay que comprobar la declaración de la función.
- Todo lo que obligue a comprobar la firma de la función es un esfuerzo doble.
- Por lo general los argumentos de salida deben evitarse. Si su función tiene que cambiar el estado de un elemento, haga que cambie el estado de su objeto contenedor.

Separación de consultas de comando

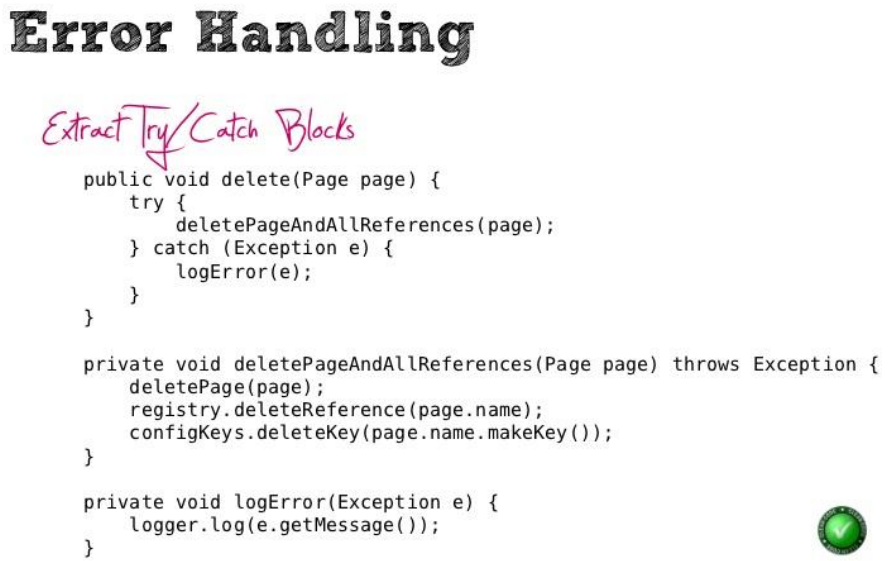
- Las funciones deben hacer algo o responder a algo, pero no ambas cosas, causa confusión. Fíjese en la siguiente función:
  - Public boolean set(String attribute, String value);
- Esta función establece el valor de un atributo y devuelve true en caso de éxito o false si el atributo no existe. Esto provoca la presencia de una extraña instrucción como la siguiente:
  - If (set(“userName”,”unclebob”))…
- Podríamos solucionarlo si cambiamos el nombre de la función set por setAndCheckIfExists, pero no mejoraría la legibilidad de la instrucción if. La verdadera solución es separar el comenado de la consulta para evitar la ambigüedad.
  - If(atribbuteExists (“userName”){
    setAtributte((“userName”,”unclebob”);}

Mejor excepciones que devolver códigos de error

- Devolver códigos de error de funciones de comandos es un sutil incumplimiento de la separación de comandos de consulta. Hace que los comandos usados asciendan a expresiones en los predicados de las instrucciones if
  - If(deletePage(page) == E_OK)
- No padece la confusión entre el verbo y el adjetivo, pero genere estructuras anidadas. Al devolver un código de error se crea un problema: el invocador debe procesar el error de forma inmediata.

Extraer bloques try/catch

- Los bloques try/catch no son atractivos por naturaleza. Confunden la estructura del código y mezclan el procesamiento de errores con el normal. Por ello, conviene extraer el cuerpo de los bloques try y catch en funciones individuales.



El procesamiento de errores es una cosa

- Las funciones solo deben hacer una cosa y el procesamiento de errores es un ejemplo. Por tanto, una función que procese errores no debe hacer nada mas.
- Esto implica que si una función incluye la palabra try, debe ser la primera de la función y que no debe tener nada mas después de los bloques catch / finally.

El imán de las dependencias Error.java

- La devolución de códigos de error suele implicar que existe una clase o enumeración en la que se definen los códigos de error.

```
Public enum Error{
Ok,
Invalid,
No_Such,
Waiting_for_evento;}
```

- Clases como esta son imán para las dependencias; otras muchas clases deben importarlas y usarlas. Por ello, cuando la enumeración Error, es necesario volver a compilar e implementar dichas clases.

No repetirse

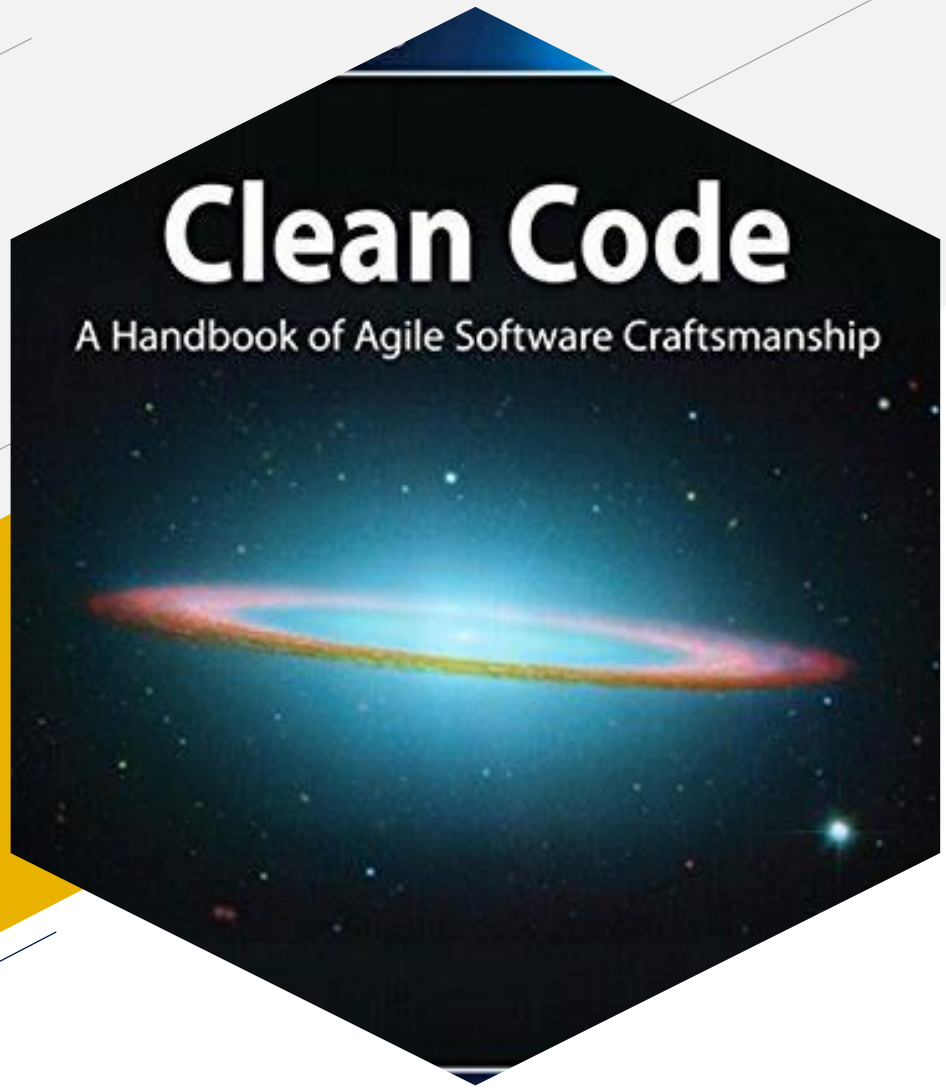
- La duplicación puede ser la raíz de todos los problemas de software. Existen numerosos principios y practicas para controlarla o eliminarla.
- Imagine que todas las formas normales de Codd sirvieron para eliminar la duplicación de datos. Imagine también como la programación orientada a objetos concentra el código en clases base que en otros casos serian redundantes

Como crear este tipo de funciones

- Cuando creo funciones, suelen ser, extensas y complicadas, con abundancia se sangrados y bloques anidados. Con extensas listas de argumentos, nombres arbitrarios y código duplicado, pero también cuento con una seri de pruebas de unidad que abarcan todas y cada una de las líneas de código.
- Por tanto, retoco el código, divido las funciones, cambio los nombres, elimino los duplicados. Reduzco los métodos y los reordeno. En ocasiones, elimino clases enteras, mientras mantengo las pruebas.

Conclusiones

- Los programadores experimentados piensan en los sistemas como en historias que contar, no como en programas que escribir.
- Recurren a las presentaciones del lenguaje elegido para crear un lenguaje expresivo mejor y mas completo que poder usar para contar esa historia.
- Si aplica estas reglas, sus funciones serán breves, con nombres correctos, y bien organizadas, pero no olvide que su verdadero objetivo es contar la historia del sistema y que las funciones que escriba deben encajar en un lenguaje claro y preciso que le sirva para contar la historia.



Bibliografia

Robert C. Martin,

"Clean code: A Handbook of Agile Software Craftsmanship"