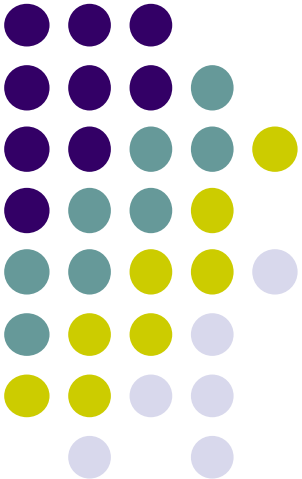
Refactoring

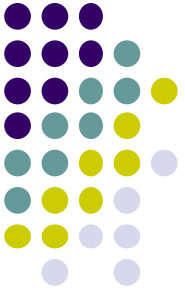
Diseño de Software

Departamento de Sistemas Informáticos y Computación

UPV

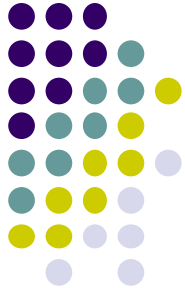


Contenido



- *¿Qué es el Refactoring?*
- *¿De dónde viene el Refactoring?*
- *¿Por qué Refactorizar?*
- *¿Cuándo Refactorizar?*
- *Pruebas. Herramientas de Pruebas y Refactorización*
- *Ejemplos de Refactoring*
- *Refactorización y… Diseño, Rendimiento, Optimización, Reuso y Realidad*
- *El libro de Fowler*
  - ***Bad Smells (Malos Olores)***
  - ***Refactorizaciones***
  - ***Aplicando Refactorings a los Bad Smells***

¿Qué es el Refactoring?



"*Refactoring* es un proceso disciplinado para cambiar un sistema software de manera que no altere el comportamiento externo del código a la vez que mejora su estructura interna." *[Fowler]*

Refactorización ⇦

¿Qué es el Refactoring?

"Una transformación de programas (código fuente - a - código fuente) que preserva el comportamiento"

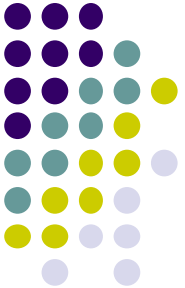
"Un cambio en el sistema que no modifica el comportamiento, pero mejora la calidad no funcional (la simplicidad, flexibilidad y comprensión del software, ...)" [Kent Beck]

¿Qué es el Refactoring?
Definiciones

- (Nombre) – Un cambio hecho en la estructura interna del software para hacer más sencillo la comprensión y la modificación sin cambiar su comportamiento observable.

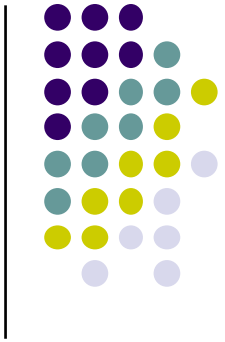
- (Verbo) – estructurar el software aplicando una serie de refactorizaciones sin cambiar su comportamiento observable.

¿De dónde viene el refactoring?



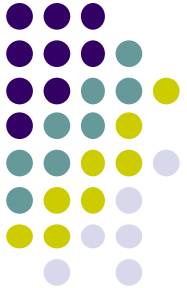
- Ward Cunningham y Kent Beck influenciaron en el campo de Smalltalk.
- Kent Beck – responsable de Extreme Programming.
- Ralph Johnson forma parte del “Gang of Four”
- Bill Opdyke – Tesis Doctoral dirigida por Johnson.
- Martin Fowler (y su libro e interés por el tema).

¿Por qué se debe Refactorizar? Argumento 1



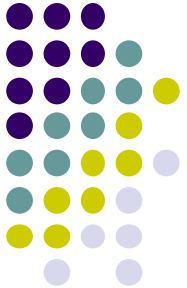
- Refactoring **Mejora el Diseño** del Software.
  – Sin el refactoring, el diseño del programa decaerá. Cuando se cambia el código – los cambios hechos sin una comprensión completa del diseño del código – el código pierde su estructura.

¿Por qué se debe Refactorizar? Argumento 2



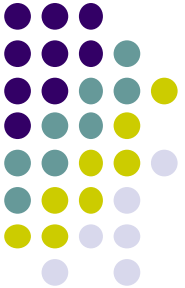
- Refactoring hace que el **software sea fácil de entender** - Existen diversos usuarios de nuestro código. El ordenador, el que escribe el código, y el que actualiza el código. El más importante es el que lo actualiza.

¿Por qué se debe Refactorizar? Argumento 3



- El Refactoring **nos ayuda a encontrar errores (bugs)**. – La refactorización permite entender el código e incluir el entendimiento en el código. En ese proceso se clarifica el código, y esto permite encontrar errores (bugs).

¿Por qué se debe Refactorizar? Argumento 4



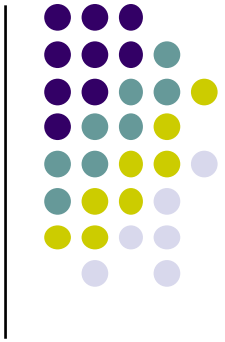
- El Refactoring nos ayuda a **Programar más Rápido**. – Sin un buen diseño, se puede progresar rápidamente al principio, pero pronto un diseño pobre empieza a ralentizar el desarrollo. Puedes gastar tiempo buscando y arreglando errores y entendiendo el sistema en vez de añadir nueva funcionalidad. Las nuevas características necesitan más codificación cuando se parchea sobre parches.

¿Cuándo se debe Refactorizar?

"¿Cuándo debo refactorizar? ¿En cuantas ocasiones? ¿Cuánto tiempo tengo que dedicarle?"

- No es algo a lo que debamos dedicar 2 semanas por cada seis meses …
- … en vez de esto, deberíamos hacerlo mientras desarrollamos!
  - Refactorizar cuando detectemos algún signo/síntoma (un "bad smell") y sepamos qué hacer
  - … cuando tu añades una función
    - Si no es una isla en ella misma (dependencias)
  - … cuando corregimos un error
    - El error es sintomático o es un problema de diseño?
  - … cuando hacemos una revisión de código
    - Una buena excusa para volver a evaluar los diseños y compartir opiniones.

La regla de tres (XP)



- La primera vez que codificas una tarea, ***"just do it".***
- La segunda vez que codificamos la misma idea, *reflexionamos* y volvemos a codificarla otra vez.
- La tercera vez que codificamos la misma idea, es tiempo de *refactorizar!!*

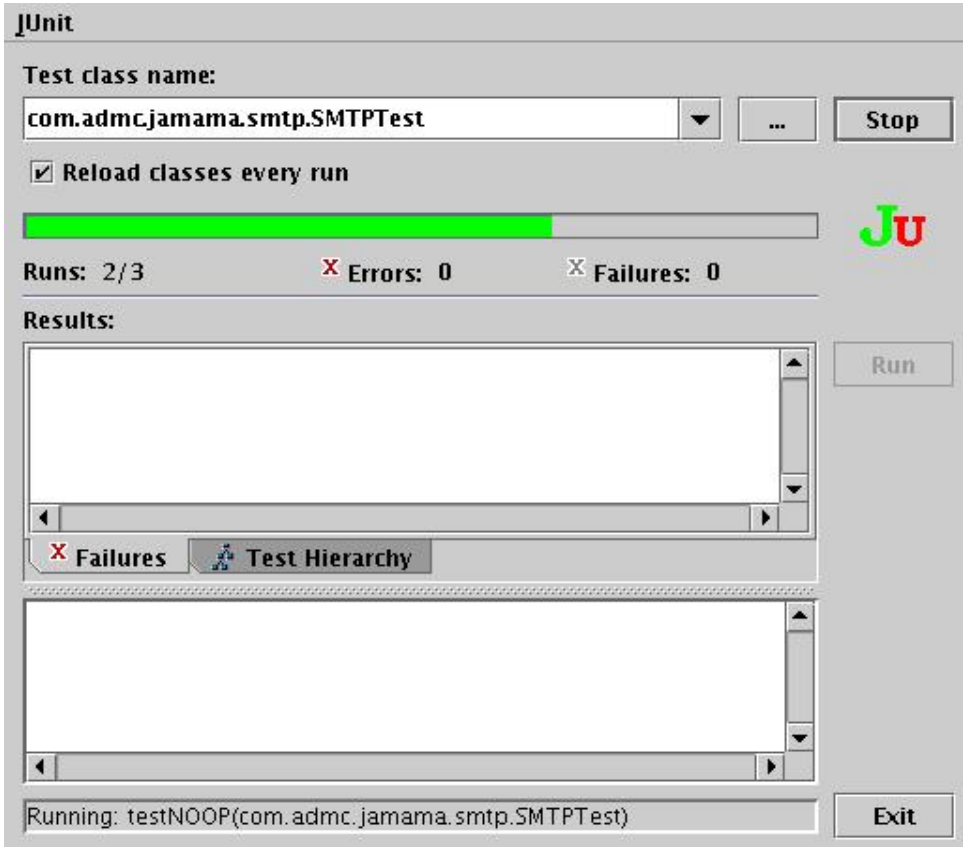
Las *Pruebas* son el primer paso en la Refactorización

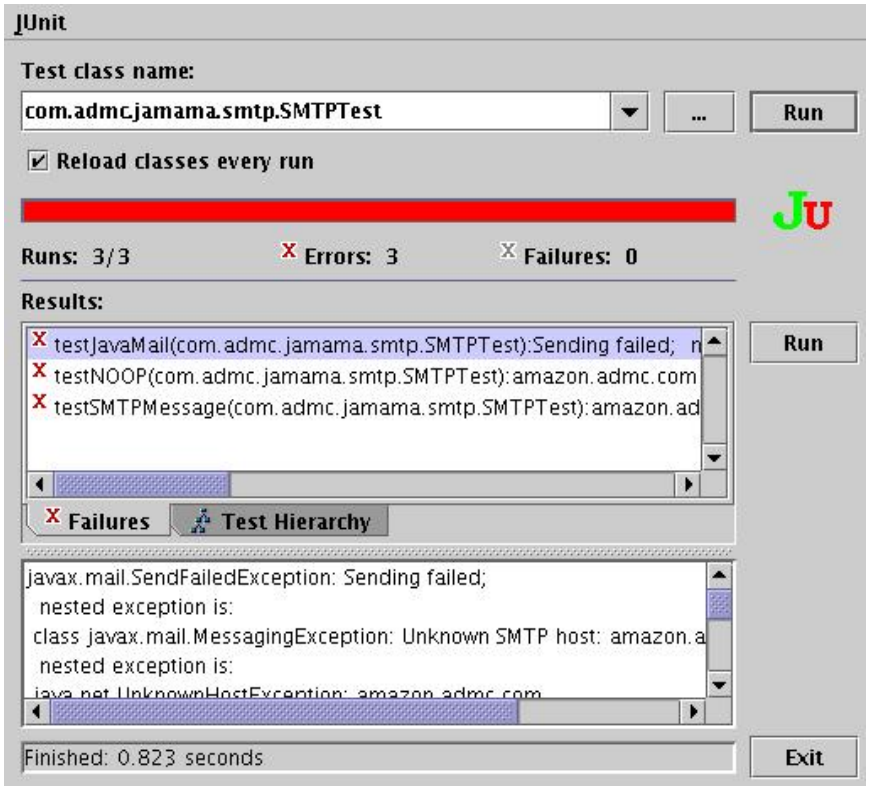
- El primer paso en la refactorización es construir un conjunto sólido de pruebas para una sección del código.
- En el refactoring la funcionalidad no cambia.
- Las Pruebas son la única garantía de que los cambios en el diseño/código no cambian la funcionalidad.
- Herramientas de Prueba.

Herramientas Pruebas

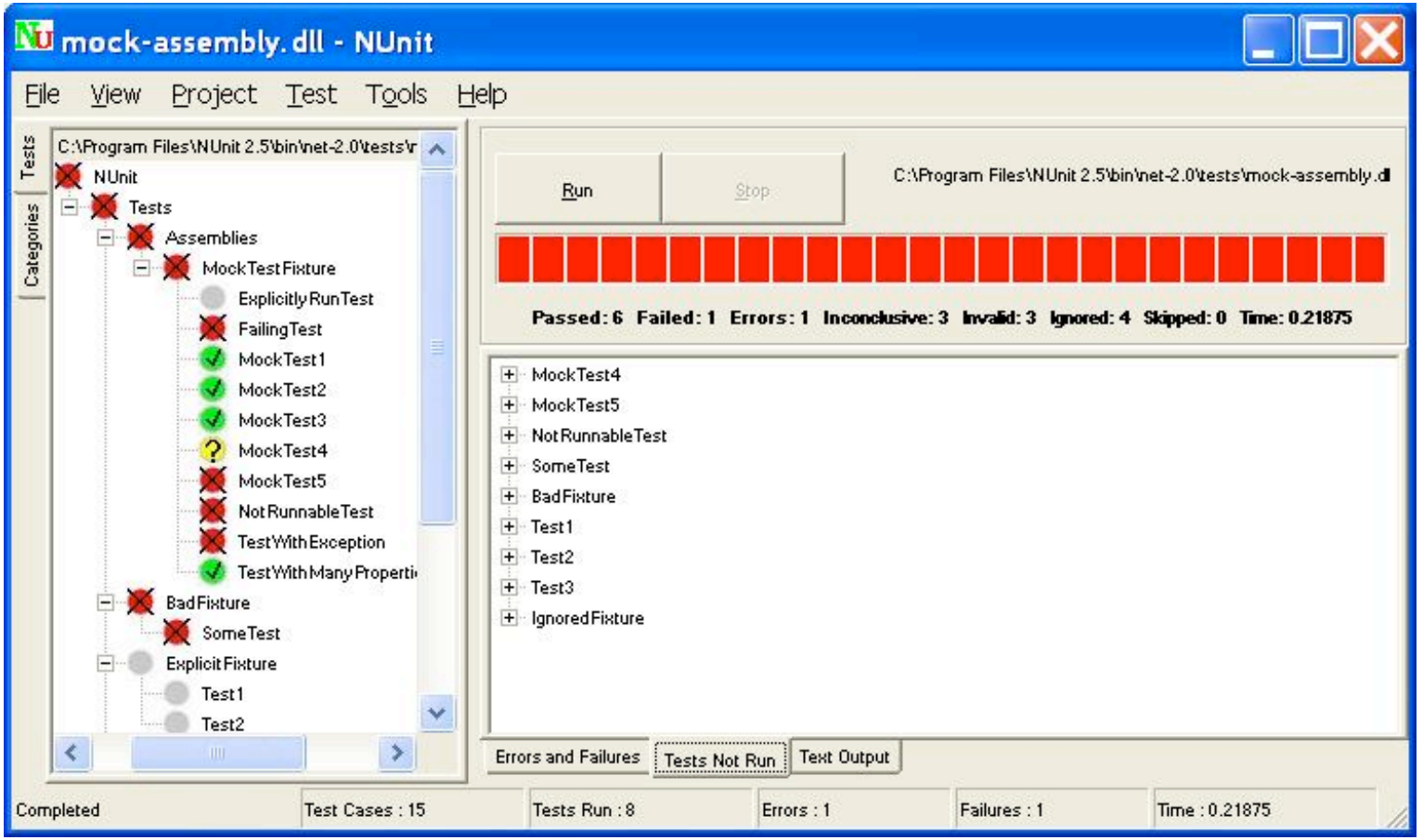
- Actualmente existen diferentes *marcos de pruebas* sencillos que permiten escribir pruebas repetibles.
- Ofrecen software para la ejecución automática de pruebas.
- JUnit, NUnit, csUnit, son algunos de ellos. Se pueden integrar con herramientas de Desarrollo e IDEs como Visual Studio .NET.

JUnit


JUnit
Test class name:
com.admc.jamama.smtp.SMTPTest
...
Stop
Reload classes every run
Runs: 2/3
Errors: 0
Failures: 0
Results:
Run
Failures
Test Hierarchy
Running: testNOOP(com.admc.jamama.smtp.SMTPTest)
Exit


JUnit
Test class name:
com.admc.jamama.smtp.SMTPTest
...
Run
Reload classes every run
Runs: 3/3
Errors: 3
Failures: 0
Results:
testJavaMail(com.admc.jamama.smtp.SMTPTest):Sending failed;
testNOOP(com.admc.jamama.smtp.SMTPTest):amazon.admc.com
testSMTPMessage(com.admc.jamama.smtp.SMTPTest):amazon.ad
Run
Failures
Test Hierarchy
javax.mail.SendFailedException: Sending failed;
nested exception is:
class javax.mail.MessagingException: Unknown SMTP host: amazon.a
nested exception is:
Finished: 0.823 seconds
Exit

nUnit (JUnit para .NET)

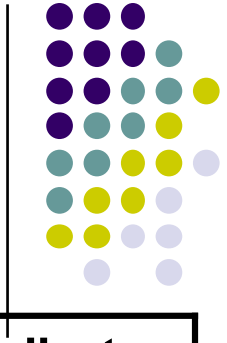


Ejemplos de Refactoring

- **Move Field** – mover las responsabilidades de una clase a otra.
- **Extract Class** – dividir una clase con muchos métodos y gran cantidad de datos.

Ejemplo - Primer Capítulo *Libro Refactoring*

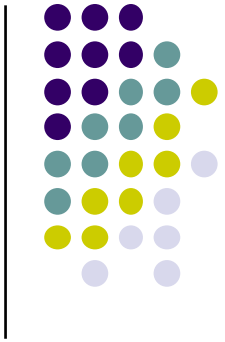
Refactorings Típicos



Refactoring de Clases	Refactorings de Métodos	Refactorings de Atributos
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	push variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Estos refactorings pueden combinarse en restructuraciones más grandes como puede ser la introducción de patrones de diseño.

Refactorización y Diseño



- El Refactoring desempeña un papel especial como complemento del diseño.
- El Refactoring puede ser visto como una alternativa al *upfront design* → ***Extreme Programming***
- Sólo refactorizar no es el modo más eficiente de trabajar. Incluso los programadores que siguen este paradigma hacen algo de diseño.

Refactorización, Rendimiento y Optimización

- En primera instancia el refactoring *puede* hacer el código más lento.
- Se debe gestionar la optimización para mejorar el rendimiento de forma distinta con la intención de corregir este problema.

Refactorización, Rendimiento y Optimización

- La intención del refactoring es conseguir un mejor diseño y código más entendible. A veces, ***no siempre***, esto hará el código más lento.
- La recomendación es construir completamente el sistema y después analizarlo para optimizarlo. Básicamente el 10% del sistema es la parte que ralentiza el sistema y debe corregirse.
- La optimización funciona mejor al final,después de aplicar el refactoring.

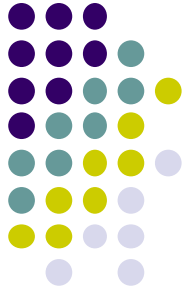
Refactoring, Reuso y Realidad

- Las empresas no quieren pagar por revisar/modificar un programa que funciona. (Paga Ahora o Paga Después)
- El Refactoring ayuda a obtener resultados a corto plazo.
- El Refactoring introduce un coste adicional.

Cosas a Tener Muy en Cuenta

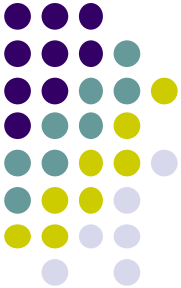
- El Refactoring mejora el diseño del código.
- Promueve una mejor comprensión del código.
- Ayuda a encontrar errores.
- El Refactoring puede generar un incremento en la velocidad de desarrollo.
- Las pruebas unitarias promueven la seguridad y fiabilidad durante el proceso de refactorización.

Refactoring

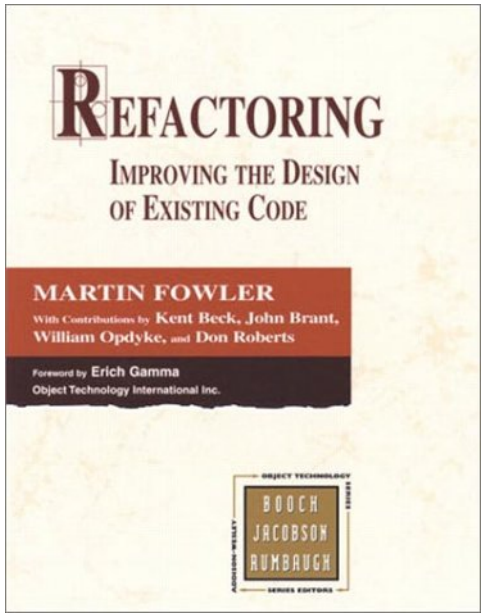


- Ideas Básicas:
  - Empezar con código existente como base y mejorarlo.
  - Cambiar la estructura interna mientras se preserva la semántica global.
- La idea es que se puede mejorar el código de manera significativa. Por ejemplo:
  - Reducir código duplicado
  - Mejorar la cohesión, reducir el acoplamiento
  - Mejorar la parametrización, comprensión, mantenibilidad, flexibilidad, abstracción, eficiencia, *etc* …

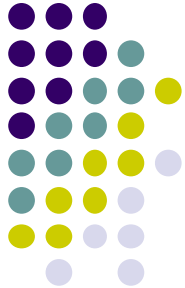
Refactoring



- Referencia Básica:
  - *Refactoring: Improving the Design of Existing Code*, de Martin Fowler (*et al.*),1999, Addison-Wesley
- Fowler, Beck, *et al.* son pesos pesados en el ámbito del Análisis y Diseño OO
  - Patrones de Diseño OO
  - XP (extreme programming)



Refactoring



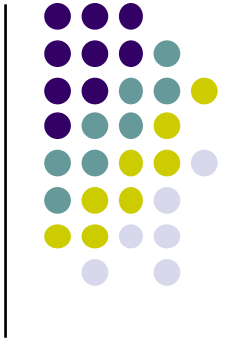
- El libro es muy bueno
  - Estas ideas han ido circulando desde hace mucho tiempo.
    - Los programadores OO experimentados conocerán la mayoría de las ideas introducidas.
  - La mayoría son un catálogo de transformaciones que puedes realizar sobre el código (con apartados de motivación, explicación, variaciones, y ejemplos)

    Por ejemplo: *Extract interface, Move method, Pull up constructor body*
  - Los Refactorings a veces vienen en pareja

    Por ejemplo: *Replace inheritance with delegation* y

    Replace delegation with inheritance

Refactoring

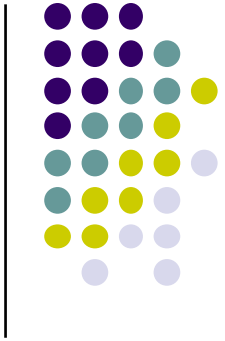


- El libro tiene un catálogo de:
  - 22 “bad smells” (malos olores)

    Cosas que tenemos que buscar (“anti-patterns”)
  - 72 “refactorings”

    Qué hacer cuando las encuentras
  - Como el libro de patrones de diseño GoF, existe solapamiento entre los items del catálogo.
  - Revisaremos algunos de los “bad smells” y explicaremos qué hacer con ellos (“que refactorización aplicar”).

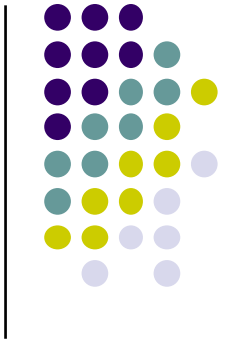
Bad Smells



"If it stinks, change it" Kent Beck

- Duplicated Code
- Long Method
- Large Class (Demasiadas responsabilidades)
- Long Parameter List (Falta un Objeto)
- Divergent Change (La misma clase cambia de diferente manera por diferentes razones)
- Shotgun Surgery (Pequeños cambios distribuidos sobre muchos objetos)

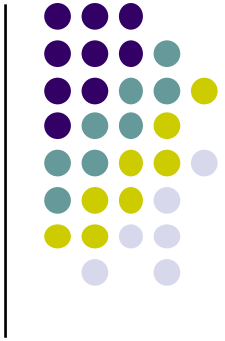
Bad Smells



- Feature Envy (Métodos que necesitan mucha información de otro objeto)
- Data Clumps (Datos que siempre se utilizan juntos (x,y -> punto))
- Primitive Obsession
- Switch Statements (Falta el polimorfismo)
- Parallel Inheritance Hierarchies (Los cambios en una jerarquía requieren los cambios en otra jerarquía)
- Lazy Class (No realiza demasiado)
- Speculative Generality
- Temporary Field (Los atributos sólo se utilizan bajo circunstancias muy específicas)

**** En esta presentación mantendremos los términos en inglés para referenciarlos adecuadamente en el libro de Fowler".***

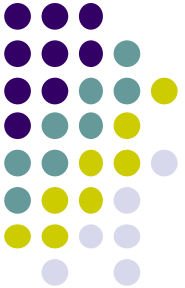
Bad Smells



- Message Chains
- Middle Man
- Inapropriate Intimacy
- Alternative Classe with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments

**** En esta presentación mantendremos los términos en inglés para referenciarlos adecuadamente en el libro de Fowler".***

Categorías de Refactoring

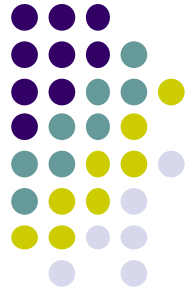


- Composing methods
- Moving features between objects
- Organizing data
- Simplifying conditional expressions
- Making method calls simpler
- Dealing with generalization
- Big refactorings

Refactoring

- Extract Method
- Inline Method
- Inline Temp
- Replace Temp With Query
- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm

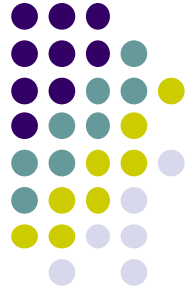
Ejemplo: Extract Method



```
void printOwing (double amount){
      printBanner();
      //print Details
      System.out.println("name: " + _name);
      System.out.println("amount " + amount);
}
```

```
void printOwing (double amount){
      printBanner();
      printDetails(amount);
}
void printDetails(double amount){
      System.out.println("name: " + _name);
      System.out.println("amount " + amount);
}
```

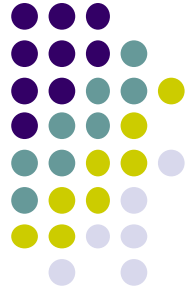
Ejemplo: Inline Method



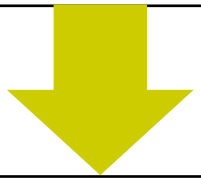
```
int getRating() {
        return (moreThanFiveLateDeliveries())?2:1;
}
boolean moreThanFiveLateDeliveries(){
        return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
        return (_numberOfLateDeliveries > 5)?2:1;
}
```

Ejemplo: Inline Temp

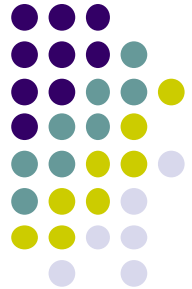


```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```

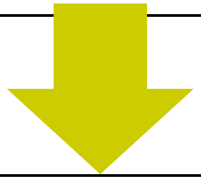


```
return (anOrder.basePrice()> 1000);
```

Ejemplo: Introduce Explaining Variable

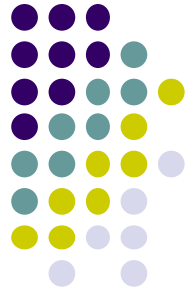


```
if (platform.toUpperCase().indexOf("MAC")>-1) &&
(browser.toUpperCase().indexOf("IE") > -1) &&
wasInitialized() && resize > 0) {
        //do something
}
```

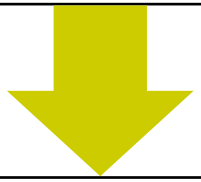


```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC")>-1;
final boolean isIEBrowser= browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized= resize > 0;
if ( isMacOs && isIEBrowser && wasInitialized() && wasResize) {
        //do something
}
```

Ejemplo: Split Temporary Variable

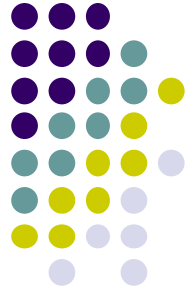


```
double temp = 2 * (_height + _width);
System.out.println(temp);
temp = _height * _width;
System.out.println(temp);
```



```
final double perimeter = 2 * (_height + _width);
System.out.println(perimeter);
final double area = _height * _width;
System.out.println(area);
```

Ejemplo: Remove Assignments to Parameters



```
int discount (int inputVal, int quantity, int
yearToDate){
        if (inputVal > 50) inputVal -=2;
…}
```

```
int discount (int inputVal, int quantity, int
yearToDate){
        int result = inputVal;
        if (inputVal > 50) result -=2;
…}
```

Ejemplo: Replace Temp with Query

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
        return basePrice * 0.95;
else
        return basePrice * 0.98;
```

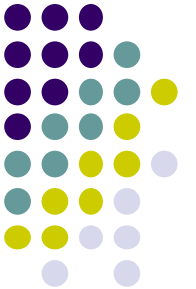
```
if (basePrice() > 1000)
        return basePrice() * 0.95;
else
        return basePrice() * 0.98;

double basePrice() {
        return  _quantity * _itemPrice;
}
```

Ejemplo: Replace Method with Method Object

```
class Order…
        double Price(){
                double primaryBasePrice;
                double secondaryBasePrice;
                double tertiaryBasePrice;
                //long computation
        }
```

```
class PriceCalculator {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        double compute() { … }
}
class Order… {
        double Price(){
                return new PriceCalculator(this).compute();
        }
}
```

Ejemplo: Substitute Algorithm



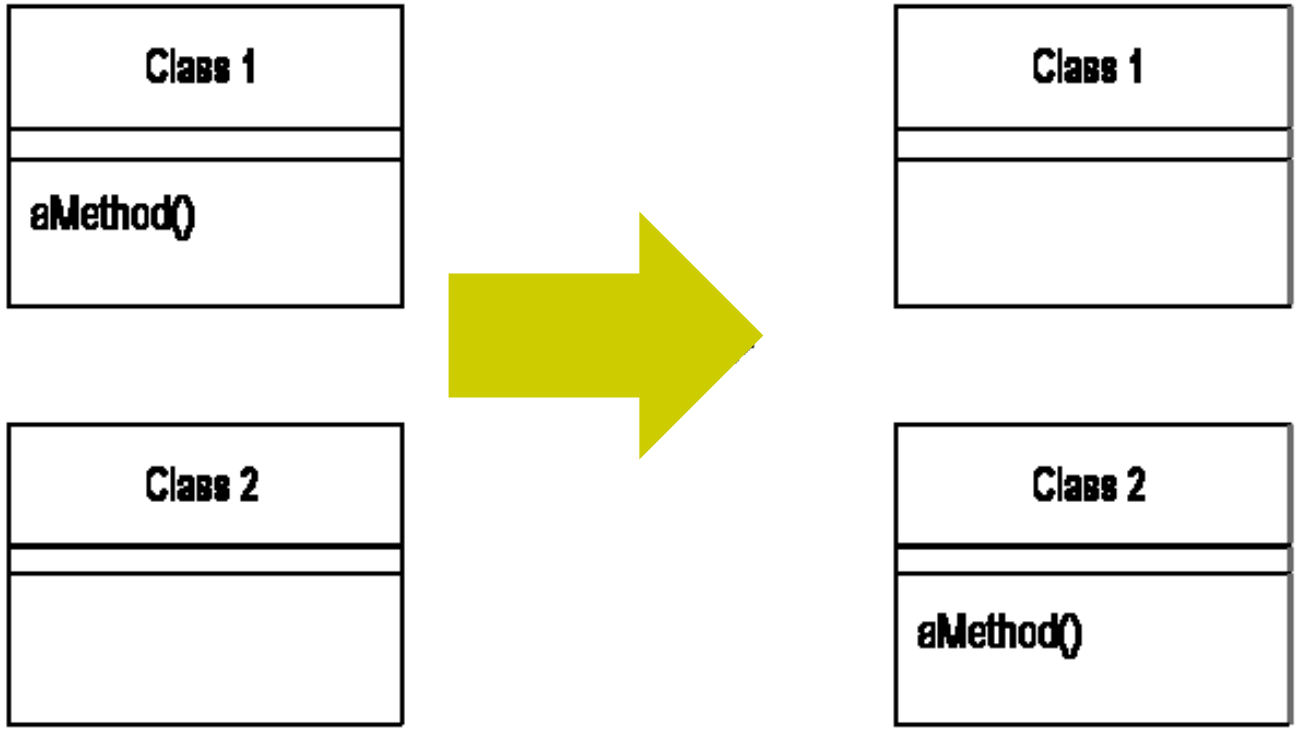
```
String foundPerson (String[] people){
  for (int i=0;i<people.length;i++){
     if (people[i].equals("Don")){return "Don"; }
     if (people[i].equals("John")){return "John"; }
     if(people[i].equals("Kent")){return "Kent"; }
     return "";
   }
}
```

```
String foundPerson (String[] people){
  List candidates = Arrays.asList(new String[]
{"Don","John","Kent"});
     for (int i=0;i<people.length;i++)
       if (candidates.contains(people[i]))
             return people[i];
     return "";
}
```

Refactoring

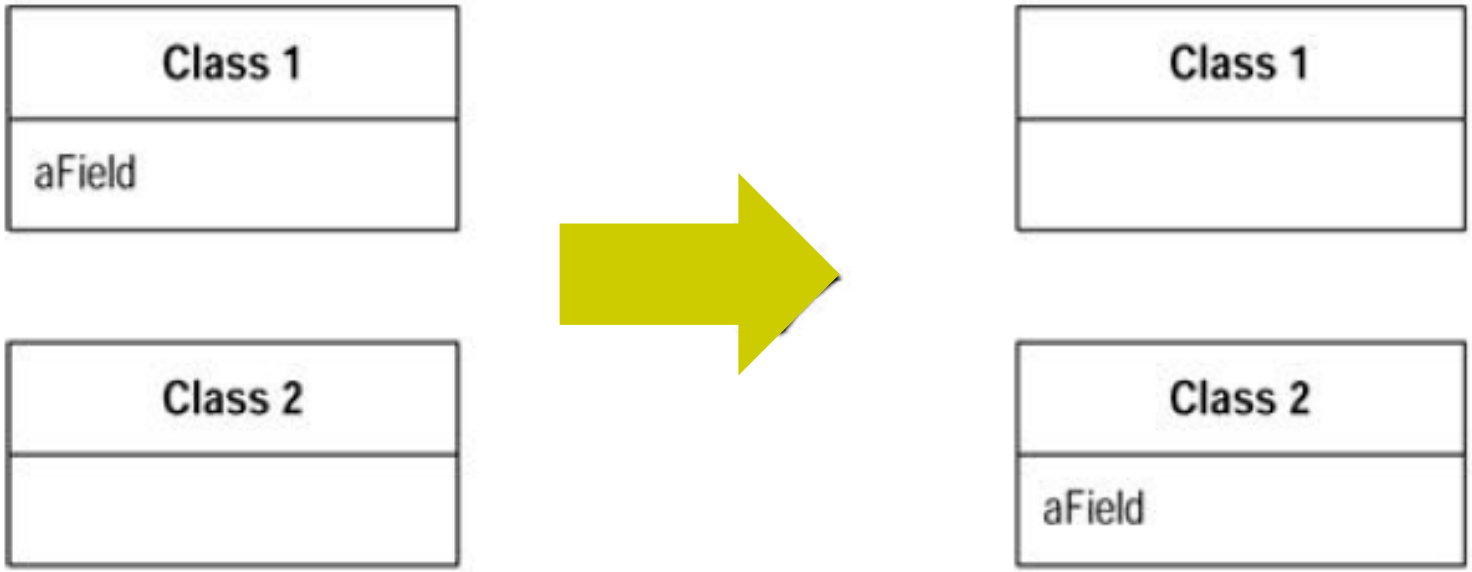
- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension

Ejemplo: Move Method

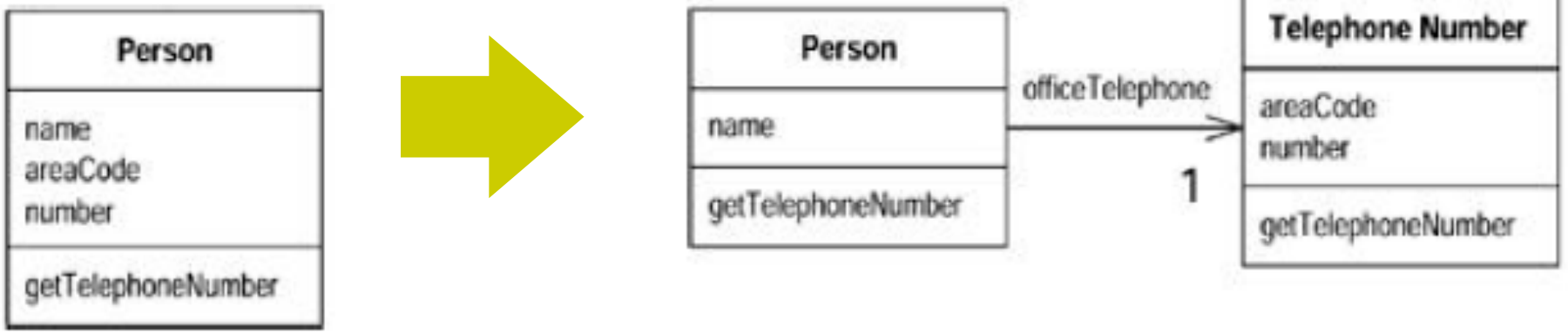


MOVING FEATURES BETWEEN OBJECTS

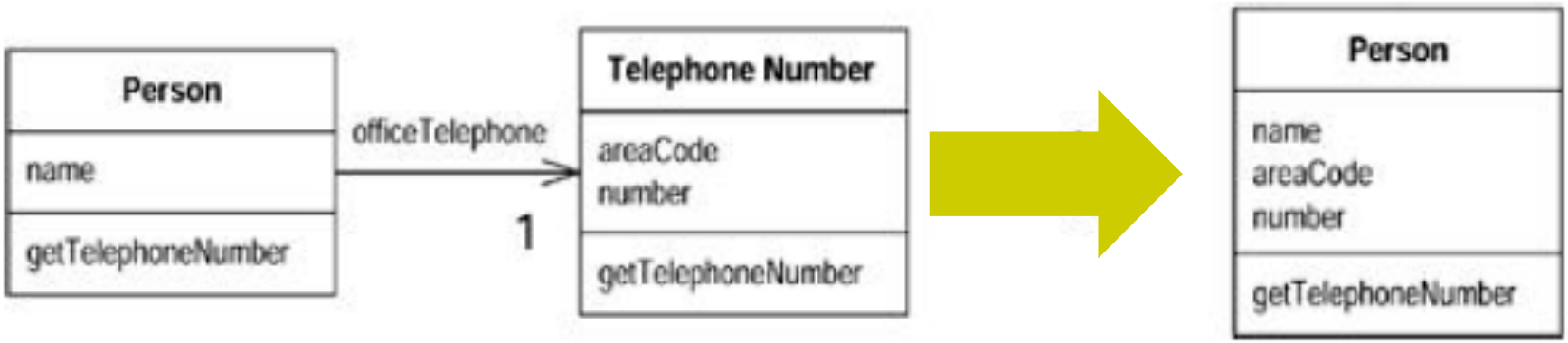
Ejemplo: Move Field



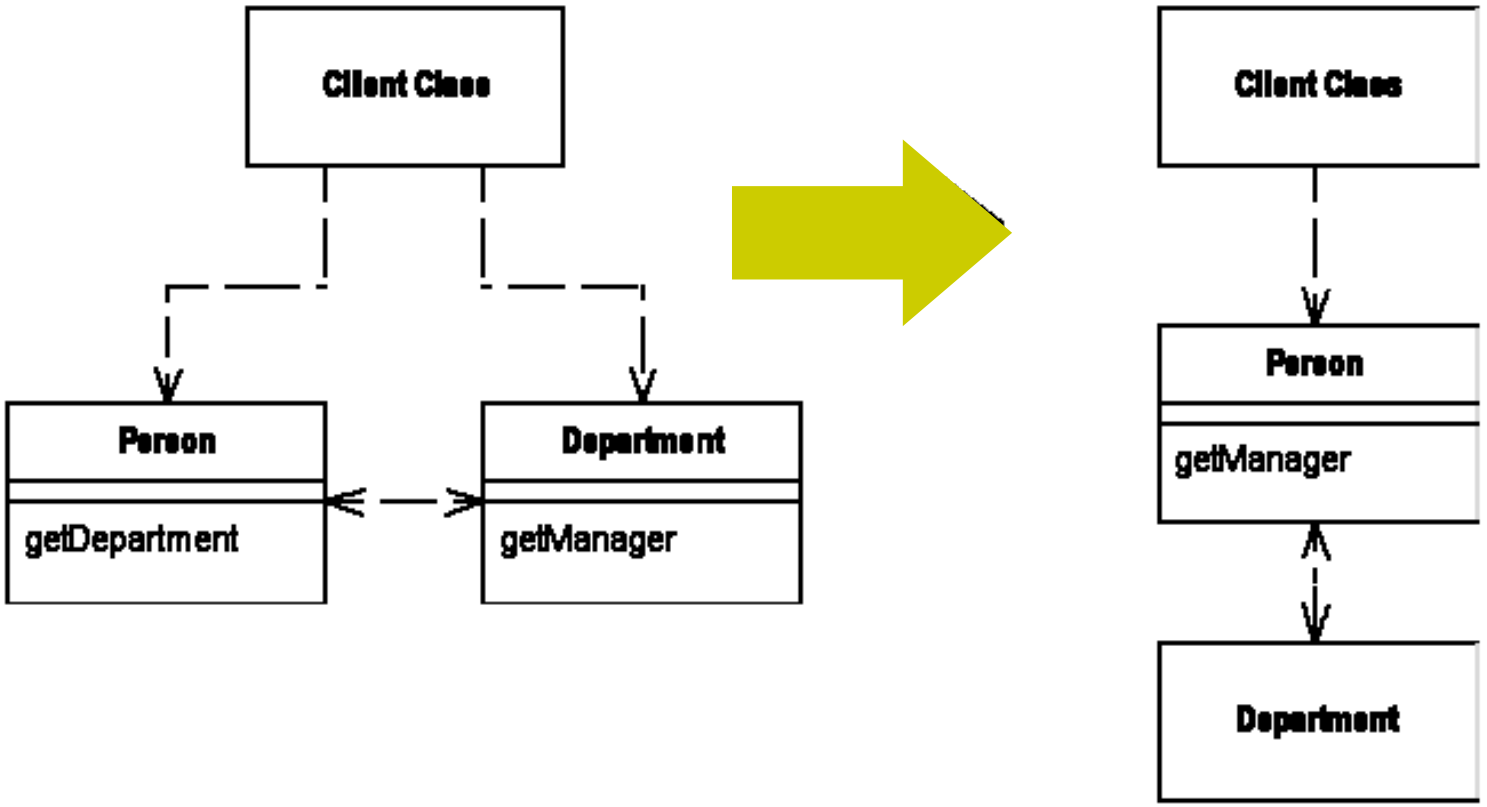
Ejemplo: Extract Class



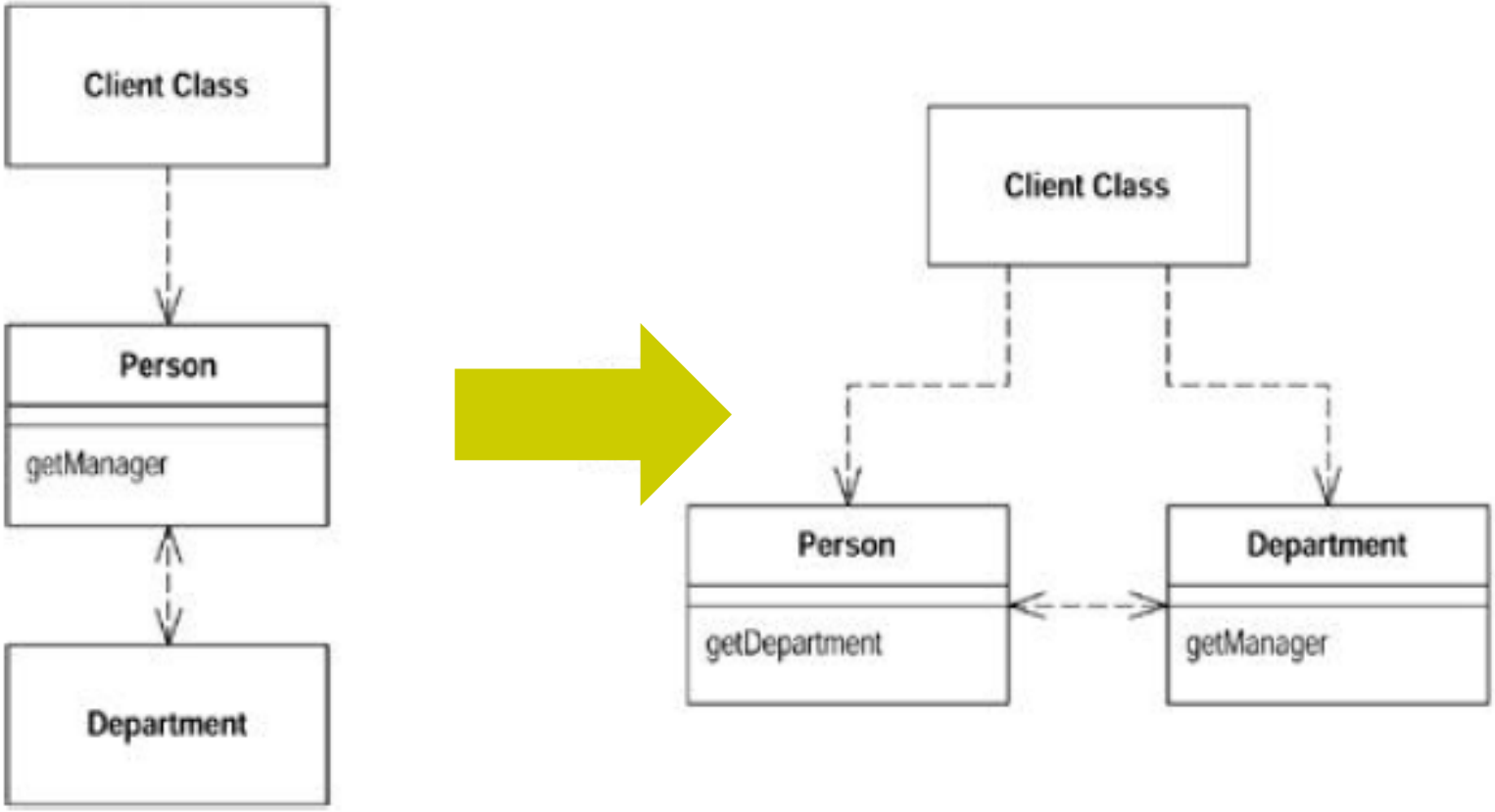
Ejemplo: Inline Class



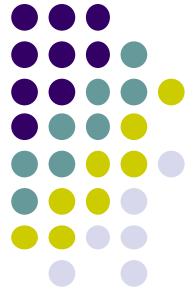
Ejemplo: Hide Delegate



Ejemplo: Remove MiddleMan



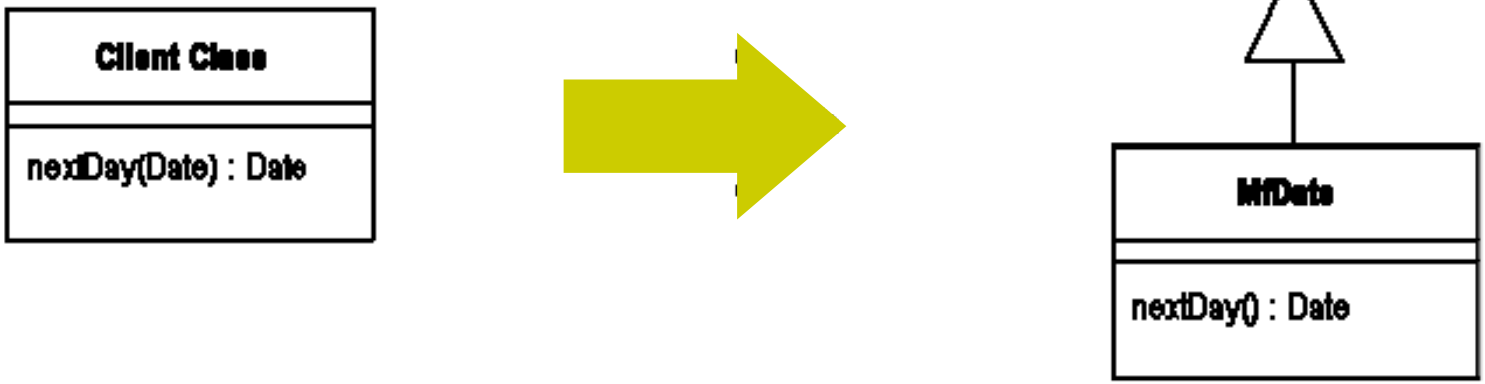
Ejemplo: Introduce Foreing Method

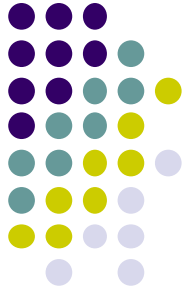


```
Date newStart = new Date (previousEnd.getYear(),
previousEnd.getMonth(), previousEnd.getDate() + 1);
```

```
Date newStart =  nextDay(previousEnd);

private static Date nextDay( Date arg){
        return new Date (arg.getYear(), arg.getMonth(),
arg.getDate() + 1 );
}
```

Ejemplo: Introduce Local Extension

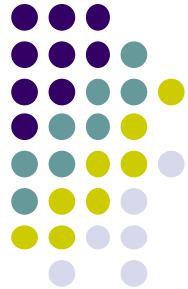




Refactoring

- Self Encapsulate Field
- Replace Data Value with Object
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional

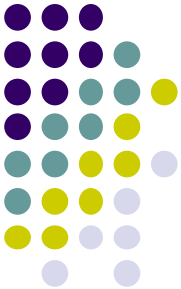
Ejemplo: Self Encapsulate Field

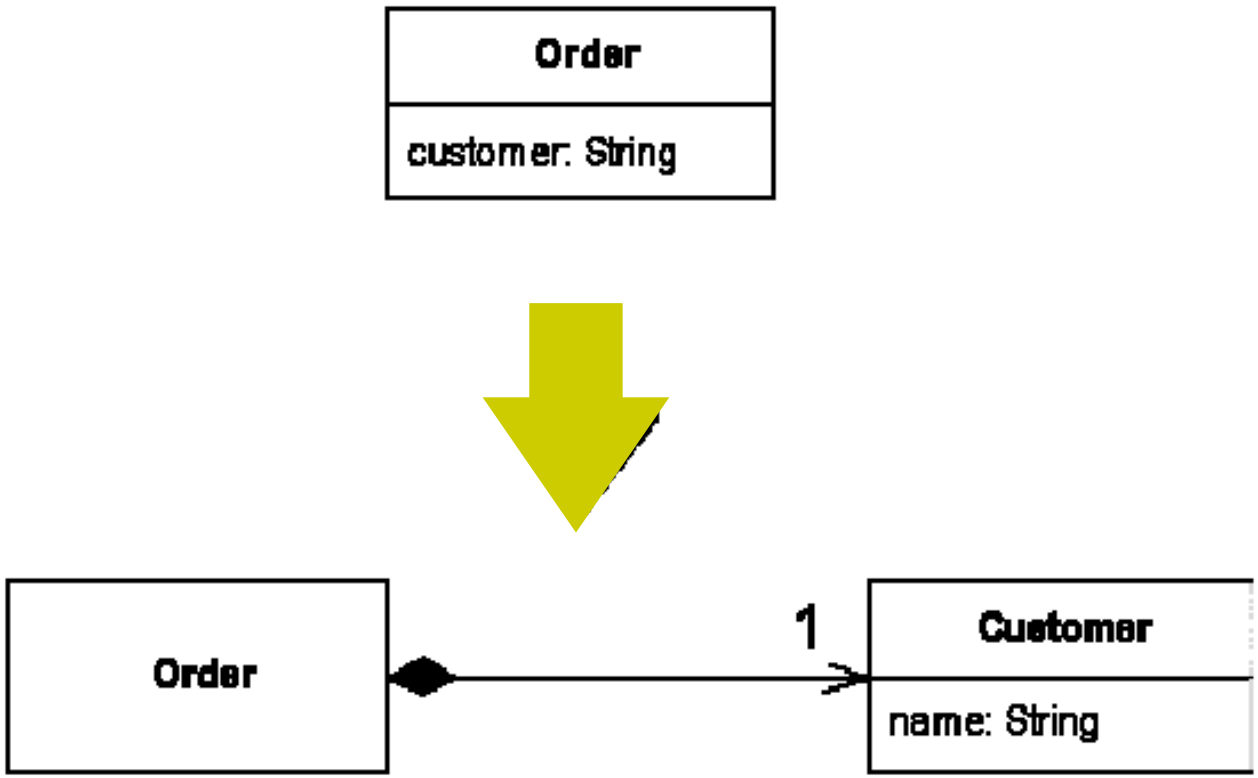


```
private int _low, _high;
boolean includes (int arg) {
        return arg >= _low && arg <= high;
}
```

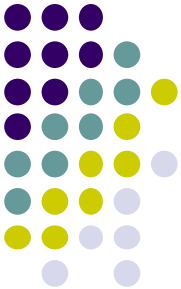
```
private int _low, _high;
boolean includes (int arg) {
        return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

Ejemplo: Replace Data Value with Object





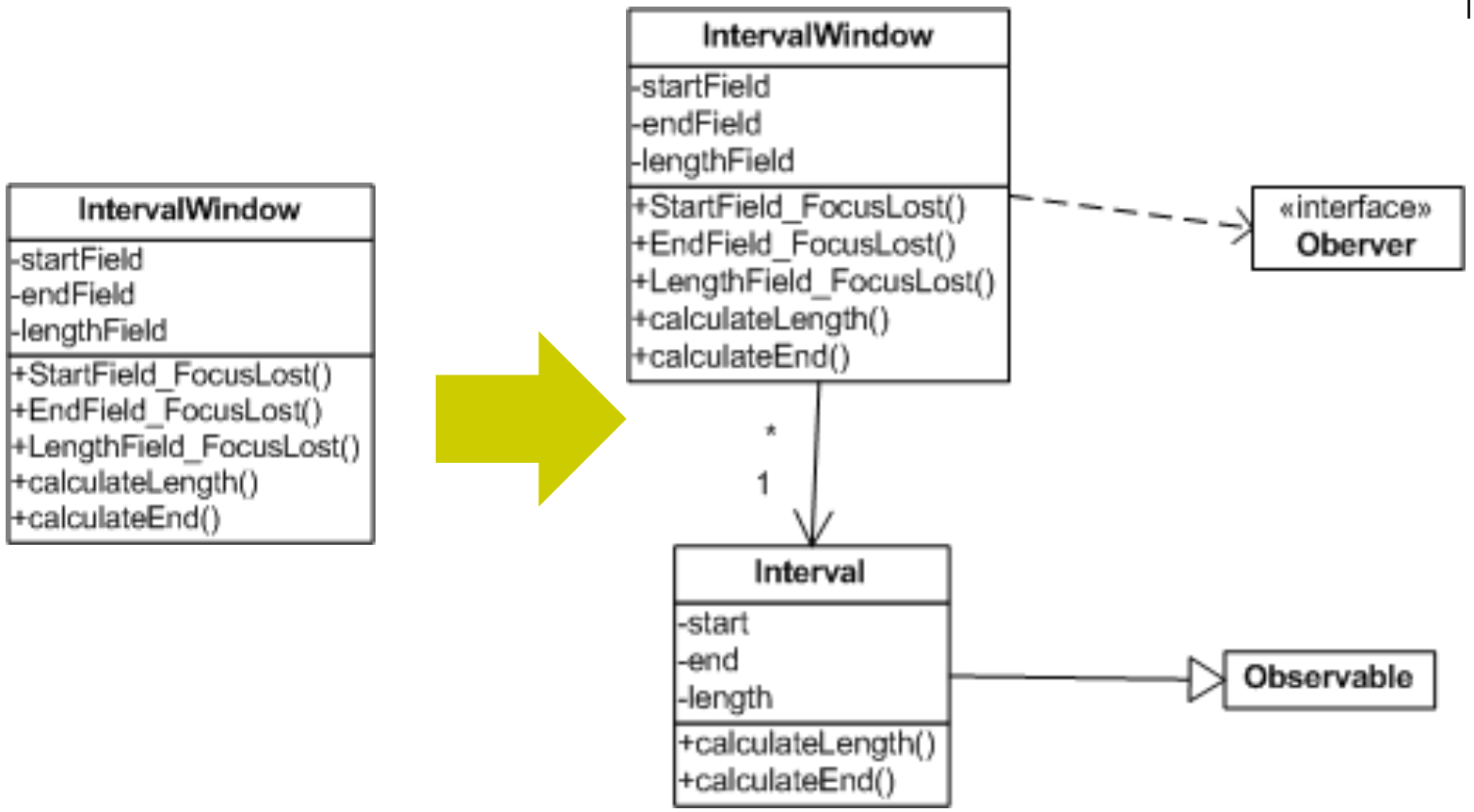
Ejemplo: Replace Array with Object



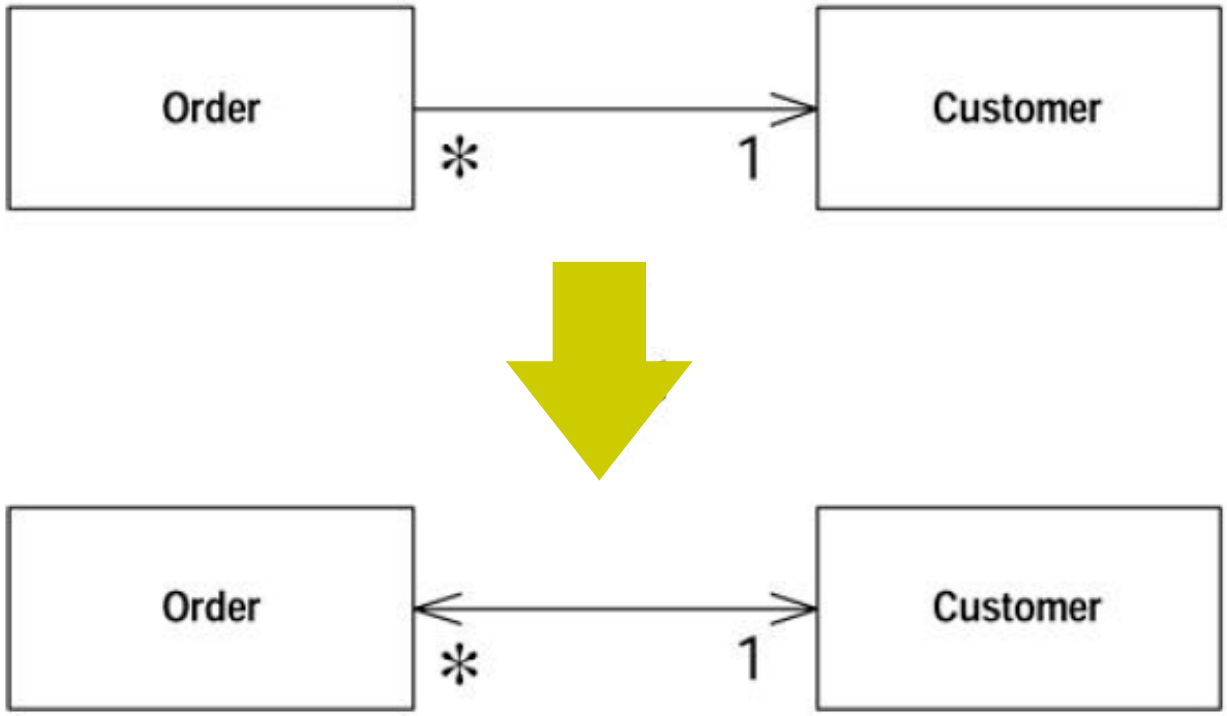
```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

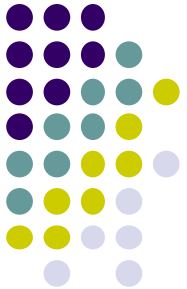
Ejemplo: Duplicate Observed Data

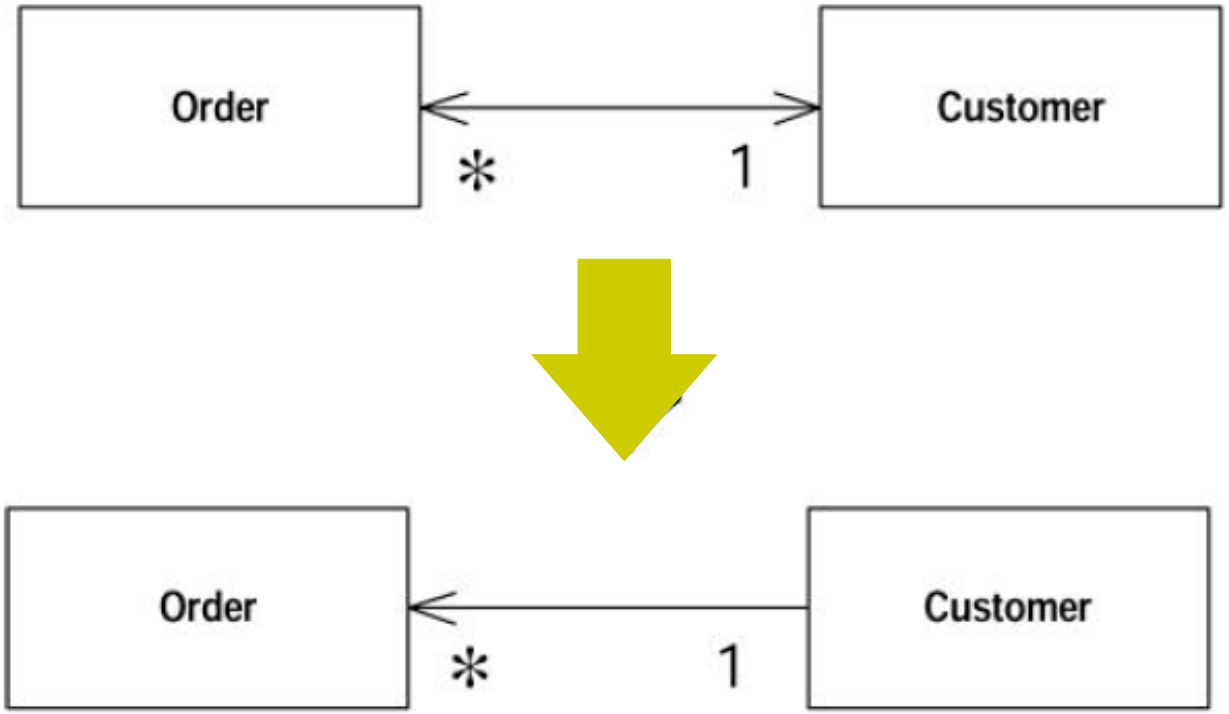


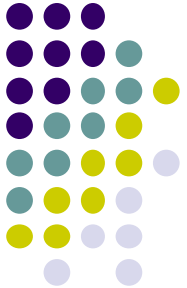
Ejemplo: Change Unidirectional Association to Bidirectional



Ejemplo: Change Bidirectional Association to Unidirectional





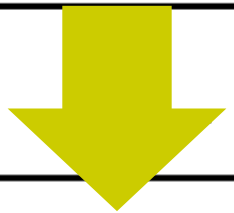


Refactoring

- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Record with Data Class
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

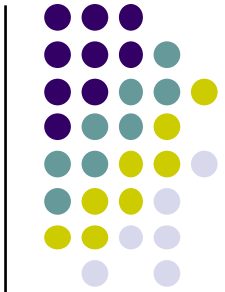
Ejemplo: Replace Magic Number with Symbolic Constant

```
double potentialEnergy(double mass, double height){
   return mass * height * 9.81;
}
```

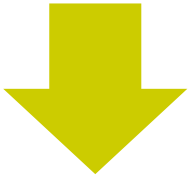


```
double potentialEnergy(double mass, double height){
   return mass * GRAVITATIONAL_CONSTANT * height;
}

static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Ejemplo: Encapsulate Field

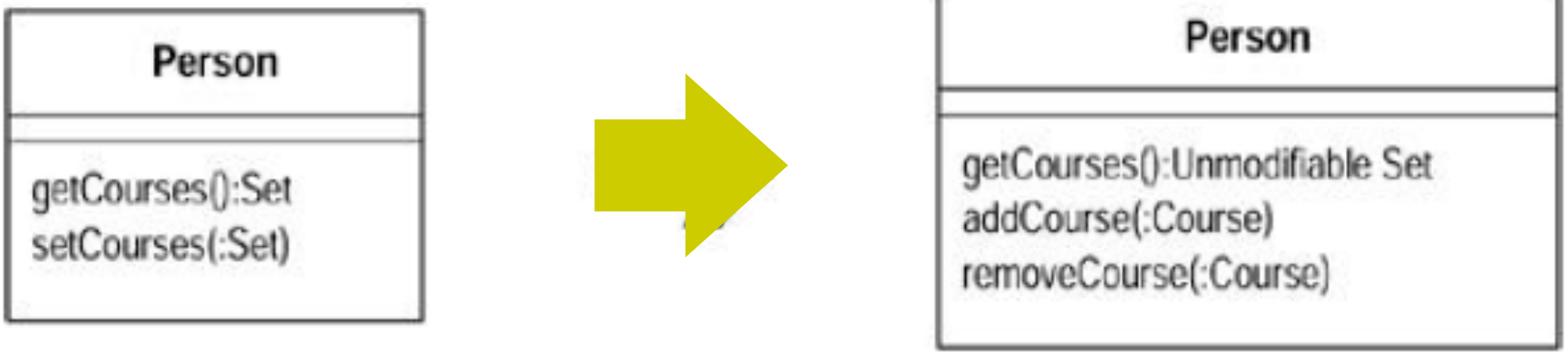


```
public String _name;
```

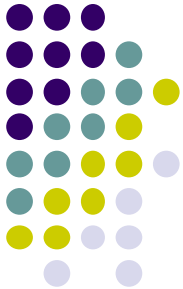


```
private string _name;
public String getName(){ return _name;}
public void setName(String value){_name=value;}
```

Ejemplo: Encapsulate Collection

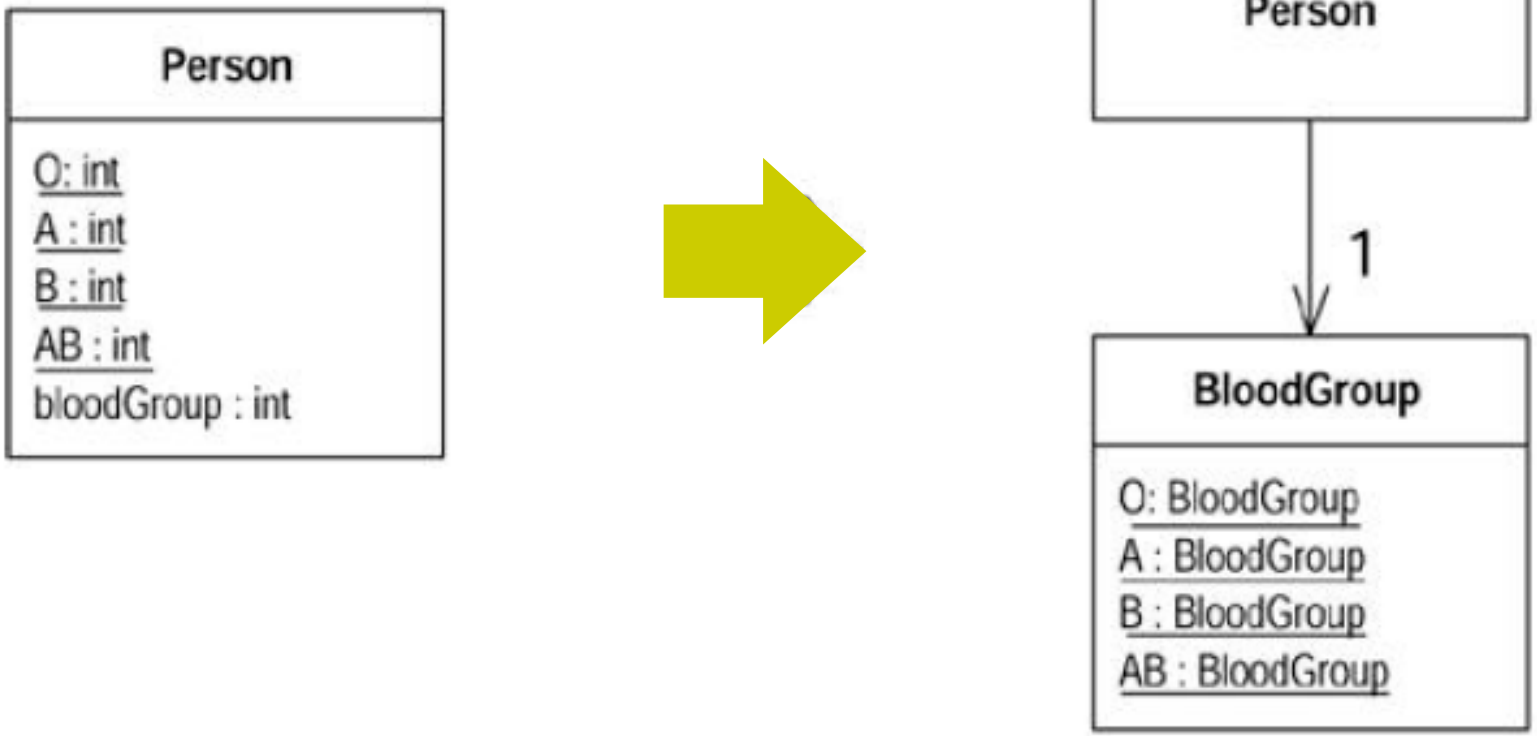


Ejemplo: Replace Record with Data Class

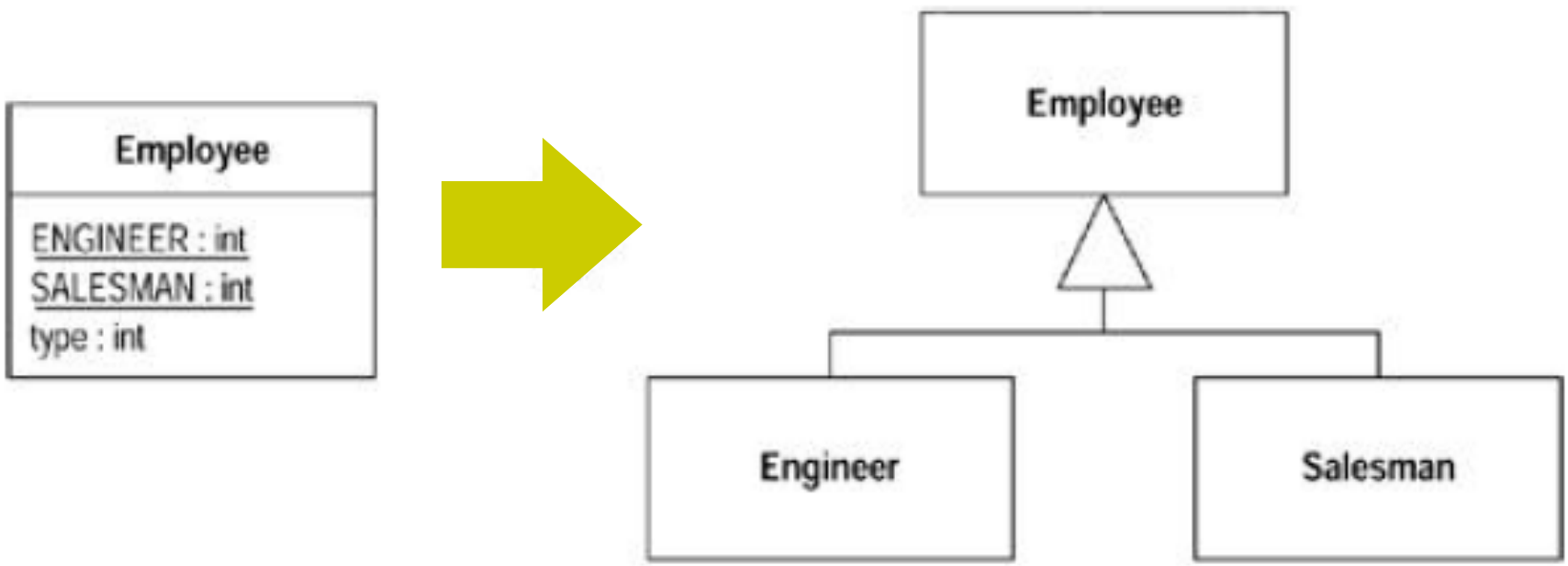


- Create a class to represent the record.
- Give the class a private field with a getting method and a setting method for each data item.

Ejemplo: Replace Type Code with Class

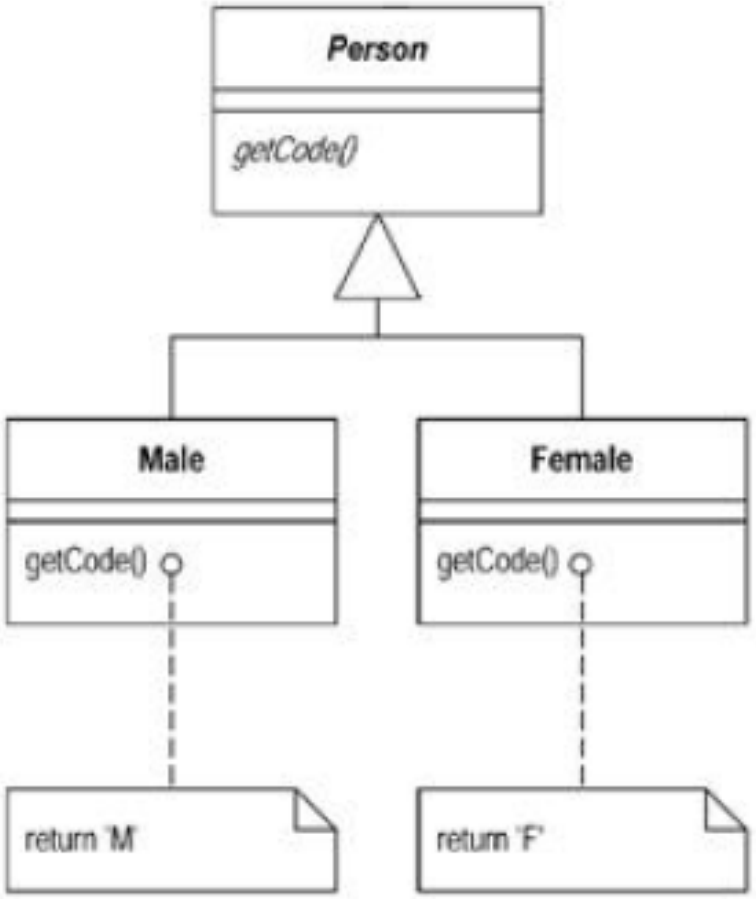


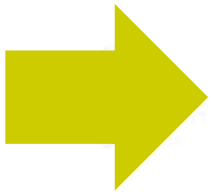
Ejemplo: Replace Type Code with Subclass

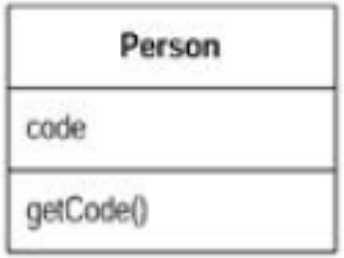


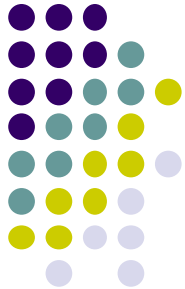
Ejemplo: Replace Type Code with State/Strategy

Ejemplo: Replace Subclass with Fields





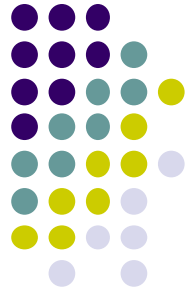




Refactoring

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Remove Control Flag
- Replace Nested Conditional with Guard Clausses
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

Ejemplo: Decompose Conditional



```
if (date.before (SUMMER_START) ||
   date.after(SUMMER_END))
   charge = quantity * _winterRate +
              _winterServiceCharge;
else
   charge = quantity * _summerRate;
```

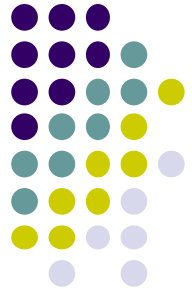
```
if (notSummer(date))
   charge = winterCharge(quantity);
else
   charge = summerCharge(quantity);
```

Ejemplo: Consolidate Conditional Expression

```
double disabilityAmount(){
        if (_seniority<2) return 0;
        if (_monthsDisabled>12) return 0;
        if (_isPartTime) return 0;
        //compute the disability amount
}
```

```
double disabilityAmount(){
        if (isNotEligableForDisability())return 0;
        //compute the disability amount
}
```

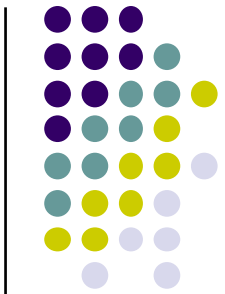
Ejemplo: Consolidate Duplicate Conditional Fragments



```
if (isSpecialDeal()) {
        total = price * 0.98;
        send();
}
else {
        total = price * 0.95;
        send();
}
```

```
if (isSpecialDeal())
        total = price * 0.98;
else
        total = price * 0.95;
send();
```

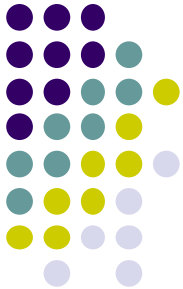
Ejemplo: Remove Control Flag



```
void checkSecurity(String[] people){
          boolean found = false;
          for (int i=0; i< people.length; i++){
                    if (!found) {
                              if (people[i].equals("Don")){
                                        sendAlert();
                                        found = true;
                              }
                              if (people[i].equals("John")){
                                        sendAlert();
                                        found = true;
                              }
                    }
          }
}
```

```
void checkSecurity(String[] people){
          for (int i=0; i< people.length; i++){
                    if (people[i].equals("Don")){
                              sendAlert();
                              break;
                    }
                    if (people[i].equals("John")){
                              sendAlert();
                              break;
                    }
          }
}
```

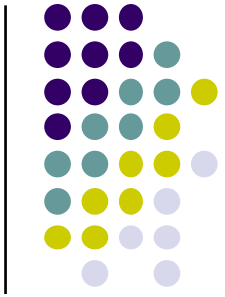
Ejemplo: Replace Nested Conditional with Guard Clausses



```
double getPayAmount(){
          double result;
          if (_isDead) result = deadAmount();
          else {
                    if (_isSeparated) result=separatedAmount();
                    else {
                              if (_isRetired) result = retiredAmount();
                              else result = normalPayAmount();
                    }
          }
          return result;
}
```

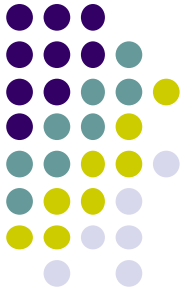
```
double getPayAmount(){
          if (_isDead) return deadAmount();
          if (_isSeparated) return separatedAmount();
          if (_isRetired) return retiredAmount();
          return normalPayAmount();
}
```

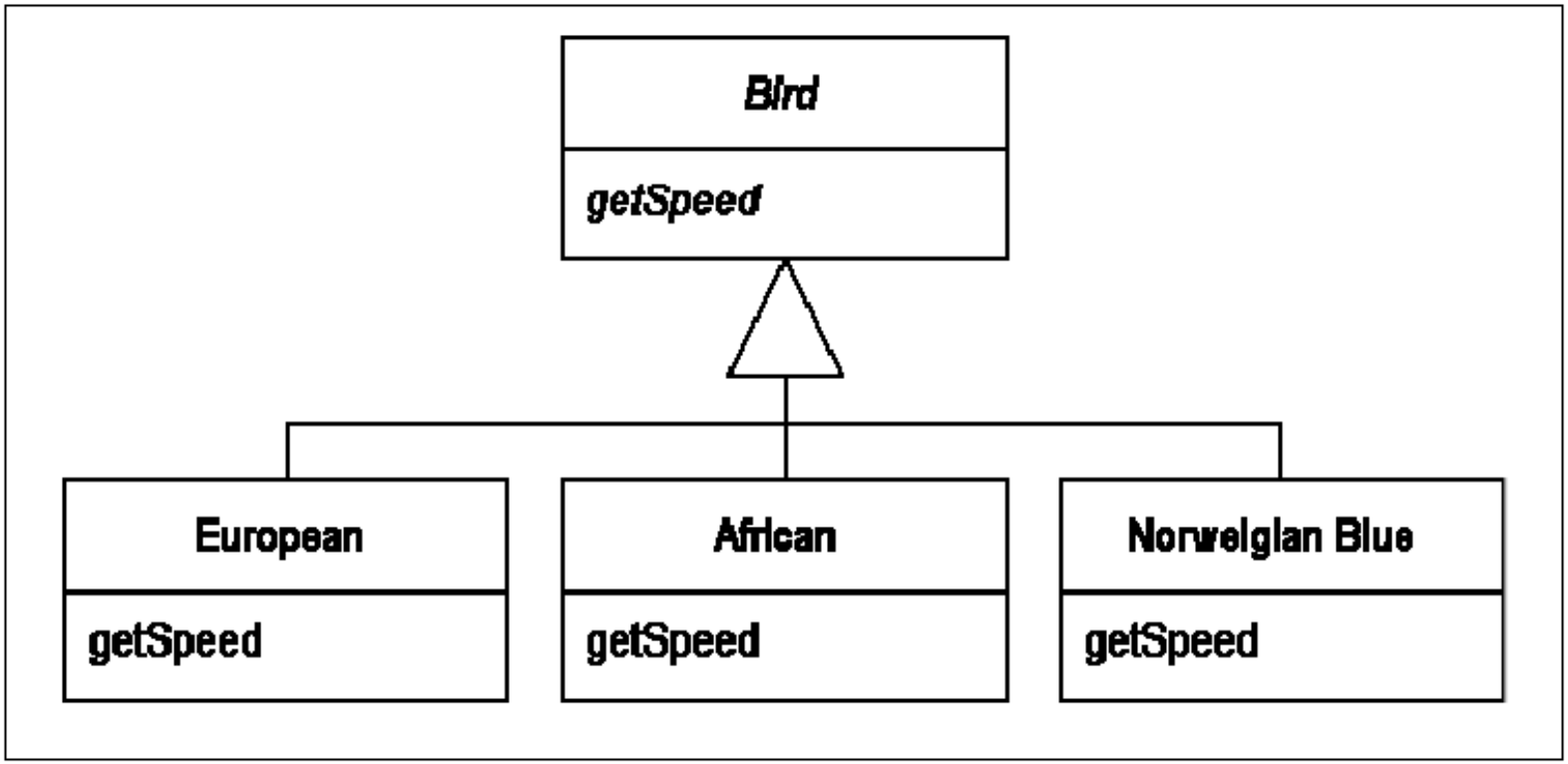
Ejemplo: Replace Conditional with Polymorphism



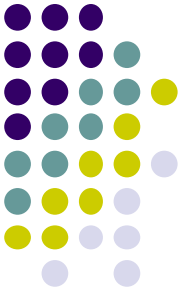
```
double getSpeed() {
   switch (_type) {
       case EUROPEAN:
             return getBaseSpeed();
       case AFRICAN:
             return getBaseSpeed() - getLoadFactor()
                   * _numberOfCoconuts;
       case NORWEIGIAN_BLUE:
             return (_isNailed) ? 0 :
                   getBaseSpeed(_voltage);
   }
   throw new RuntimeException ("Should be
   unreachable");
}
```

Ejemplo: Replace Conditional with Polymorphism

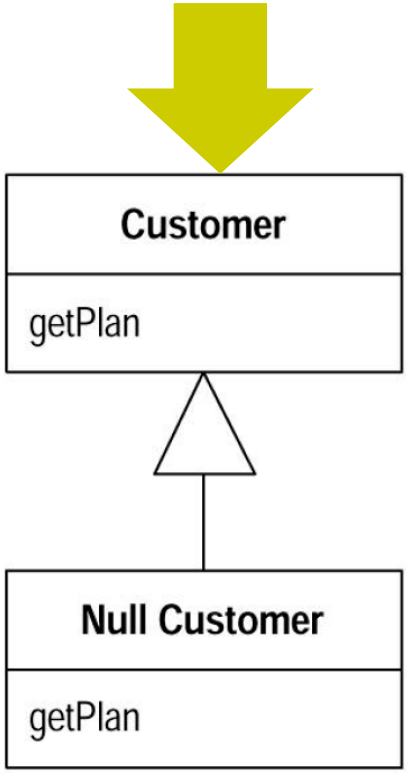




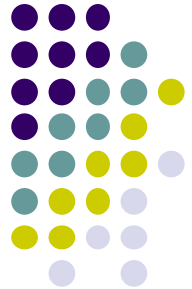
Ejemplo: Introduce Null Object



```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



Ejemplo: Introduce Assertion



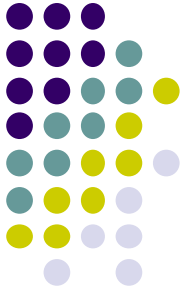
```
double getExpenseLimit(){
          //should have either expense limit or a primary project
          return (_expenseLimit != NULL_EXPENSE)?
                    _expenseLimit:
                    _primaryProject.getMemberExpenseLimit()
}
```

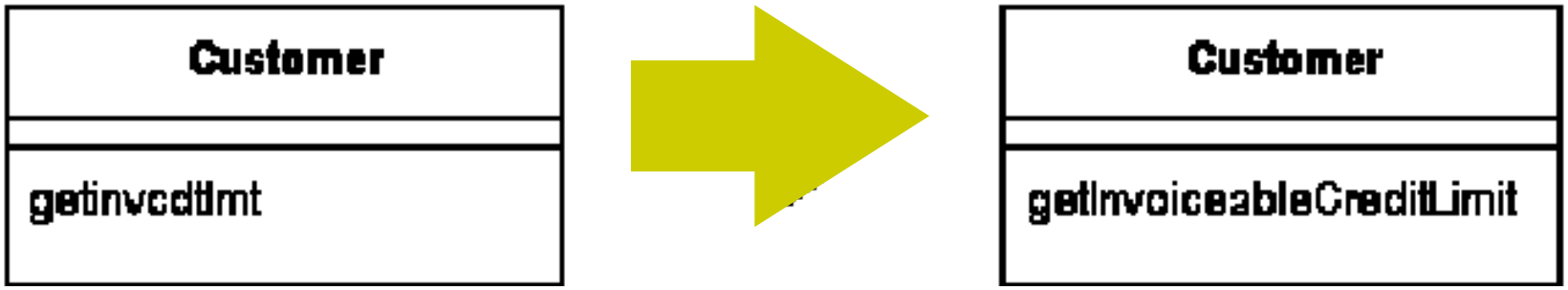
```
double getExpenseLimit(){
          Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
          return (_expenseLimit != NULL_EXPENSE)?
                    _expenseLimit:
                    _primaryProject.getMemberExpenseLimit()
}
```

Refactoring

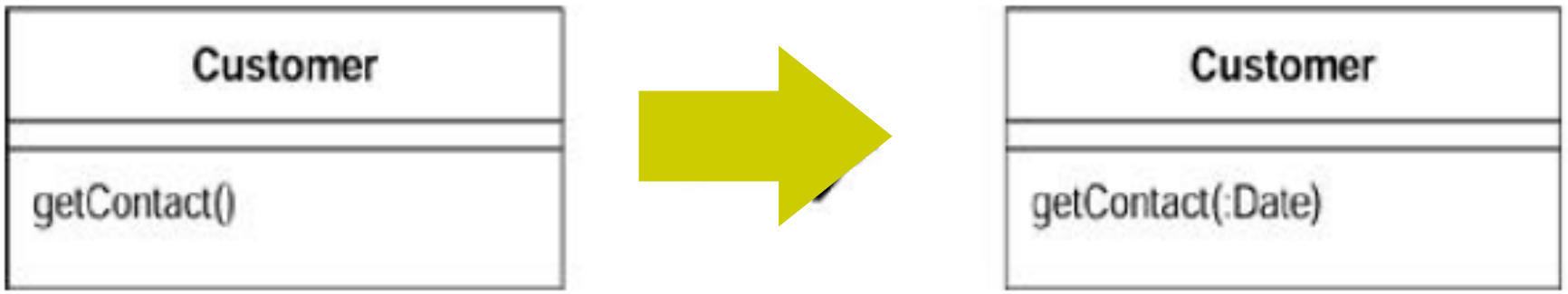
- Rename Method
- Add Parameter
- Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explícit Methods
- Preserve Whole Object
- Replace Parameter with Method

Ejemplo: Rename Method

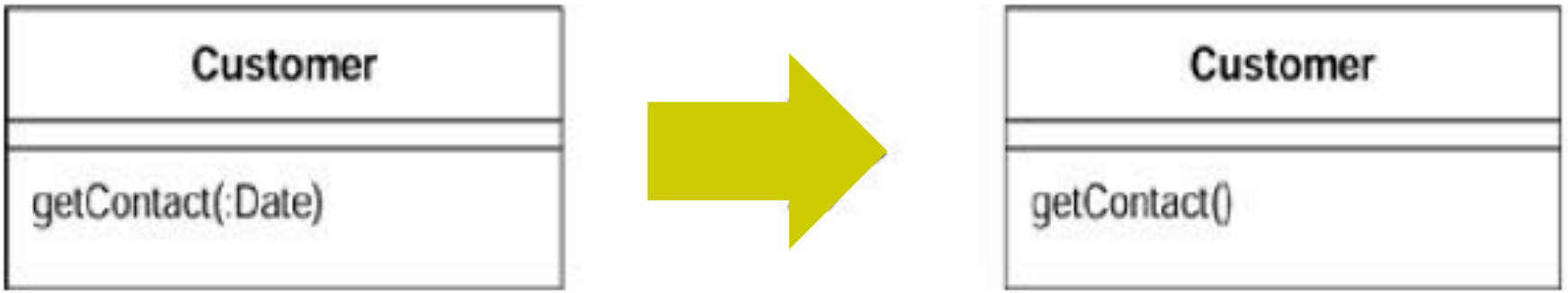




Ejemplo: Add Parameter

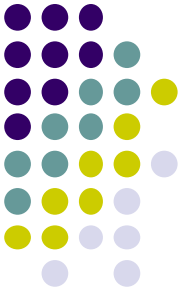


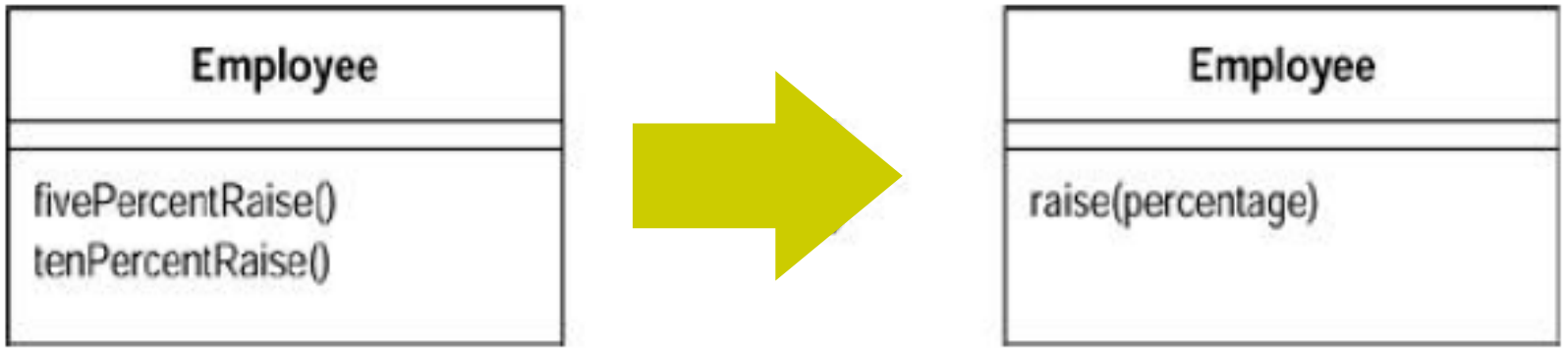
Ejemplo: Remove Parameter



Ejemplo: Separate Query from Modifier

Ejemplo: Parameterize Method



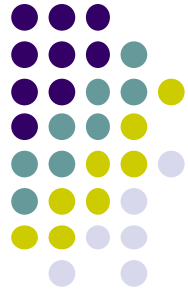


Ejemplo: Replace Parameter with Explicit Methods

```
void setValue(String name, int value){
          if (name.equals("height")) _height=value;
          if (name.equals("width")) _width=value;
          Assert.shouldNeverReachHere();
}
```

```
void setHeigh(int arg){
          _heigh = arg;
}
void setWidth(int arg){
          _width = arg;
}
```

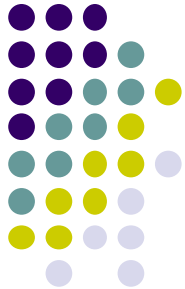
Ejemplo: Preserve Whole Object



```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

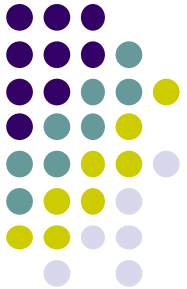
```
withinPlan = plan.withinRange(daysTempRange());
```

Ejemplo: Replace Parameter with Method



```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

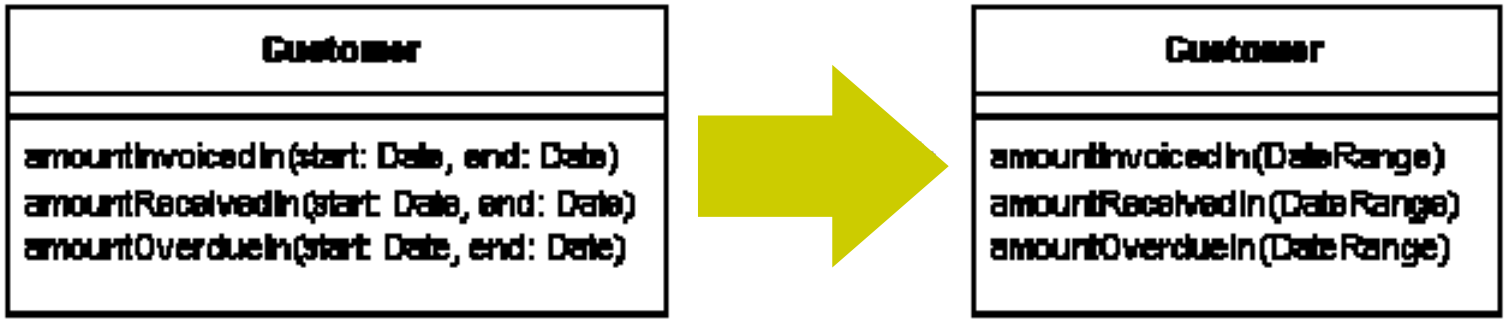
```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice(basePrice);
```



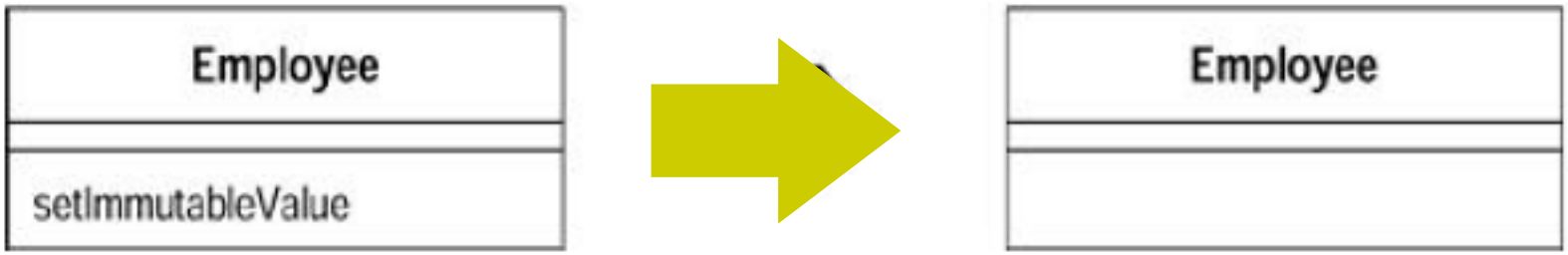
Refactoring

- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

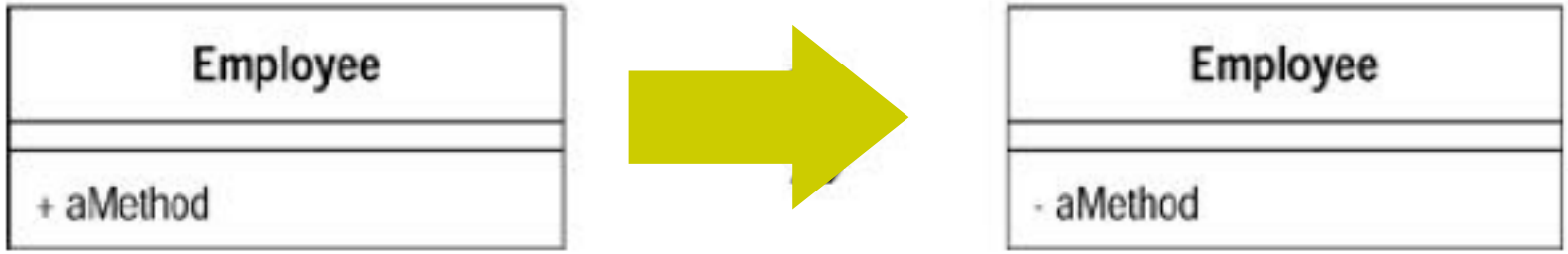
Ejemplo: Introduce Parameter Object



Ejemplo: Remove Setting Method



Ejemplo: Hide Method

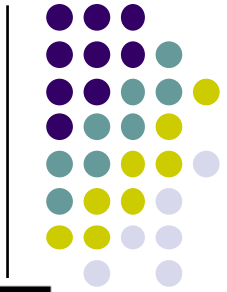


Ejemplo: Encapsulate Downcast

```
Object lastReading(){
          return readings.lastElement();
}
```

```
Reading lastReading(){
          return (Reading)readings.lastElement();
}
```

Ejemplo: Replace Error Code with Exception



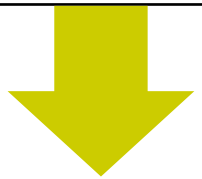
```
int withdraw(int amount) {
   if (amount > _balance)
       return -1;
   else {
       _balance -= amount;
       return 0;
   }
}
```

```
void withdraw(int amount) throws BalanceException {
   if (amount > _balance) throw new
       BalanceException();
   _balance -= amount;
}
```

MAKING METHOD CALLS SIMPLER

Ejemplo: Replace Exception with Test

```
double getValueForPeriod (int periodNumber){
          try {
                    return _values[periodNumber];
          }catch (ArrayIndexOutOfBoundsException e){
                    return 0;
          }
}
```



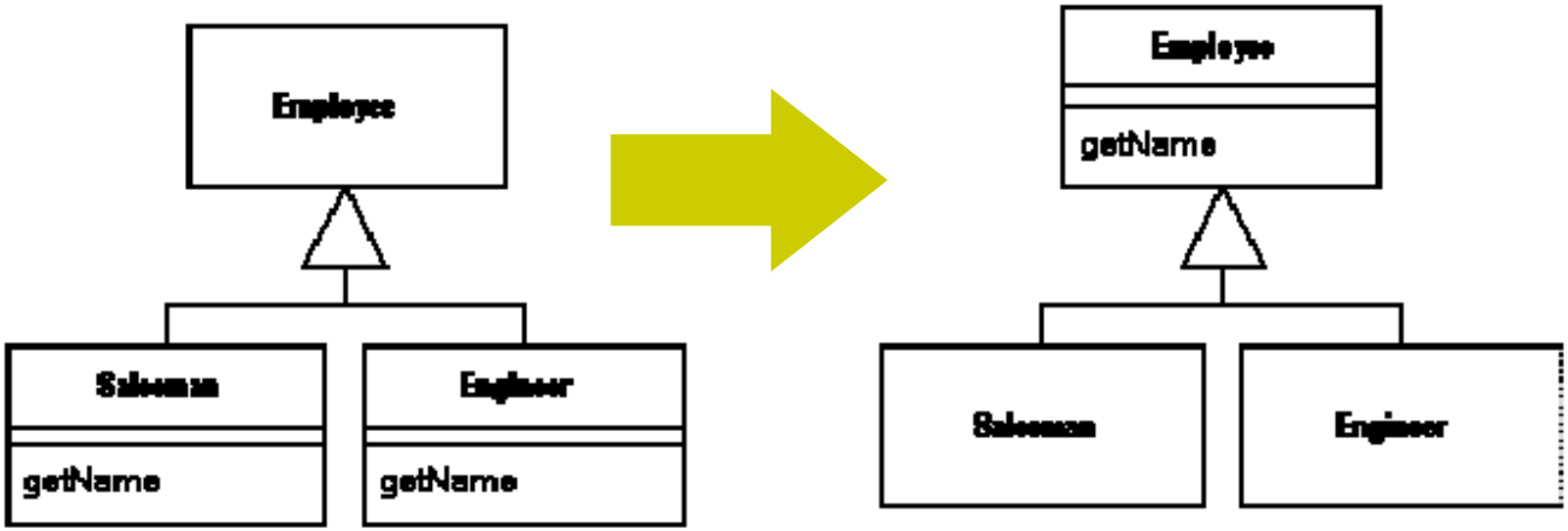
```
double getValueForPeriod (int periodNumber){
          if (periodNumber >= values.length) return 0;
          return _values[periodNumber];
}
```

Refactoring

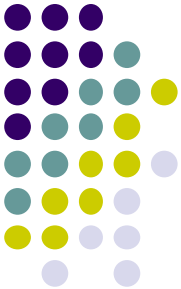
- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

Ejemplo: Pull up Field

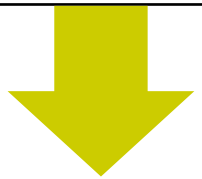
Ejemplo: Pull up Method



Ejemplo: Pull up Constructor Body

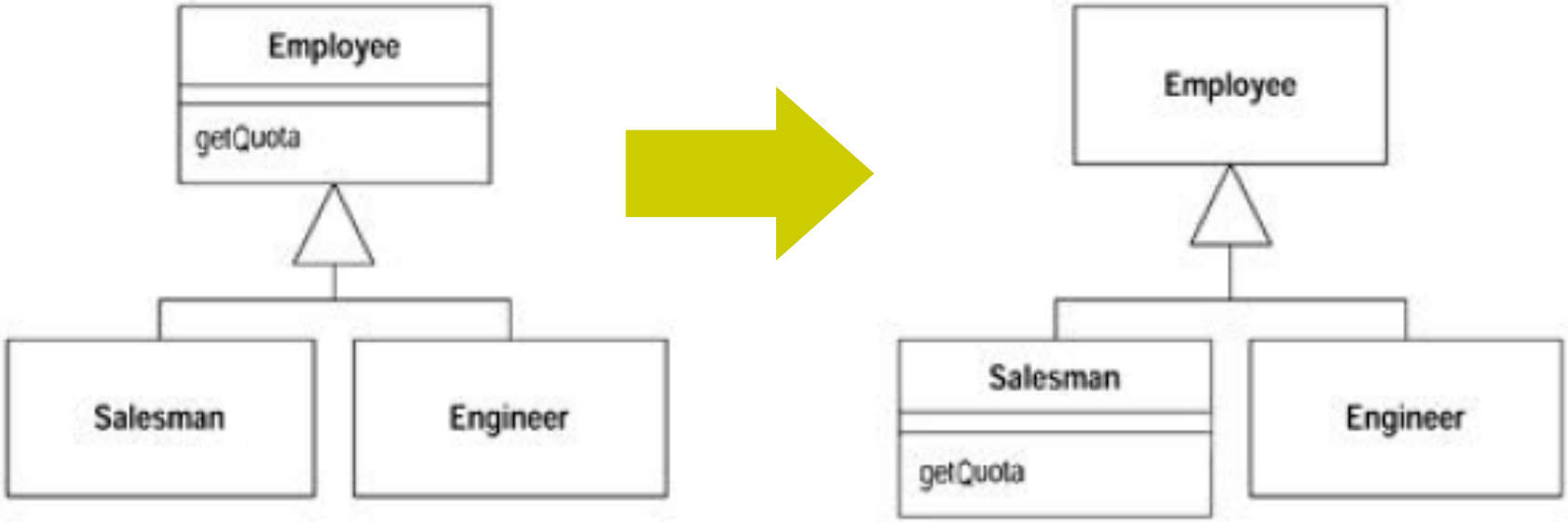


```
class Manager extends Employee...
          public Manager(string name, string id, int grade){
                    _name = name;
                    _id = id;
                    _grade = grade;
          }
... }
```

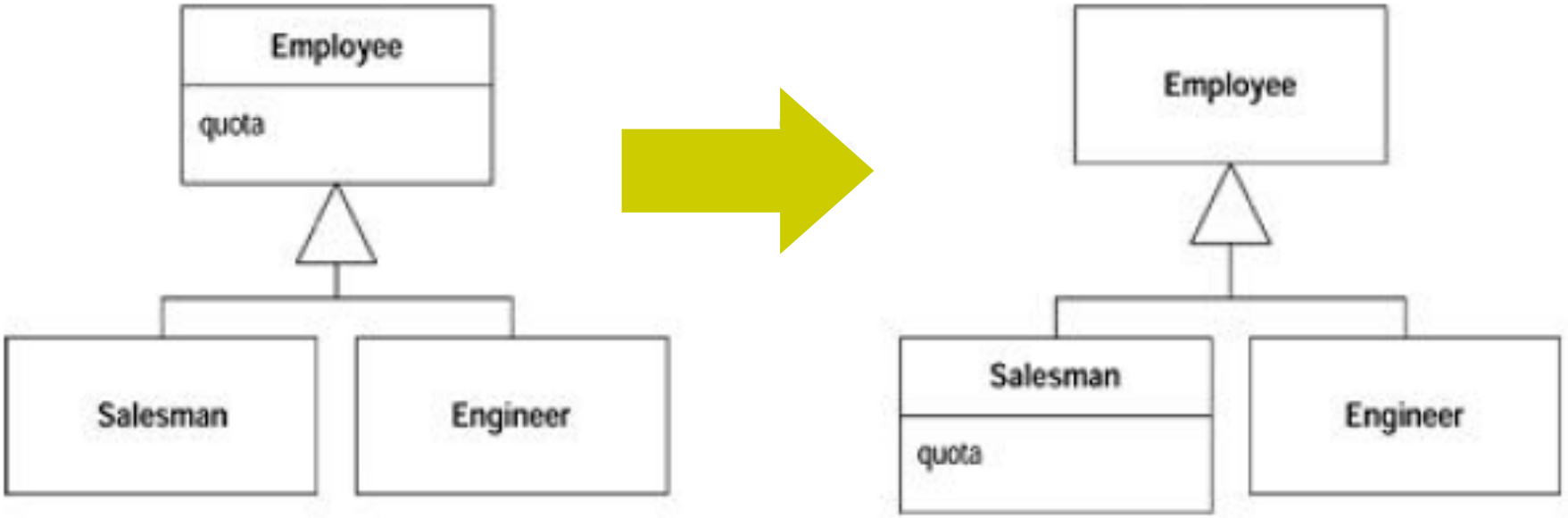


```
class Manager extends Employee...
          public Manager(string name, string id, int grade){
                    super(name, id);
                    _grade = grade;
          }
... }
```

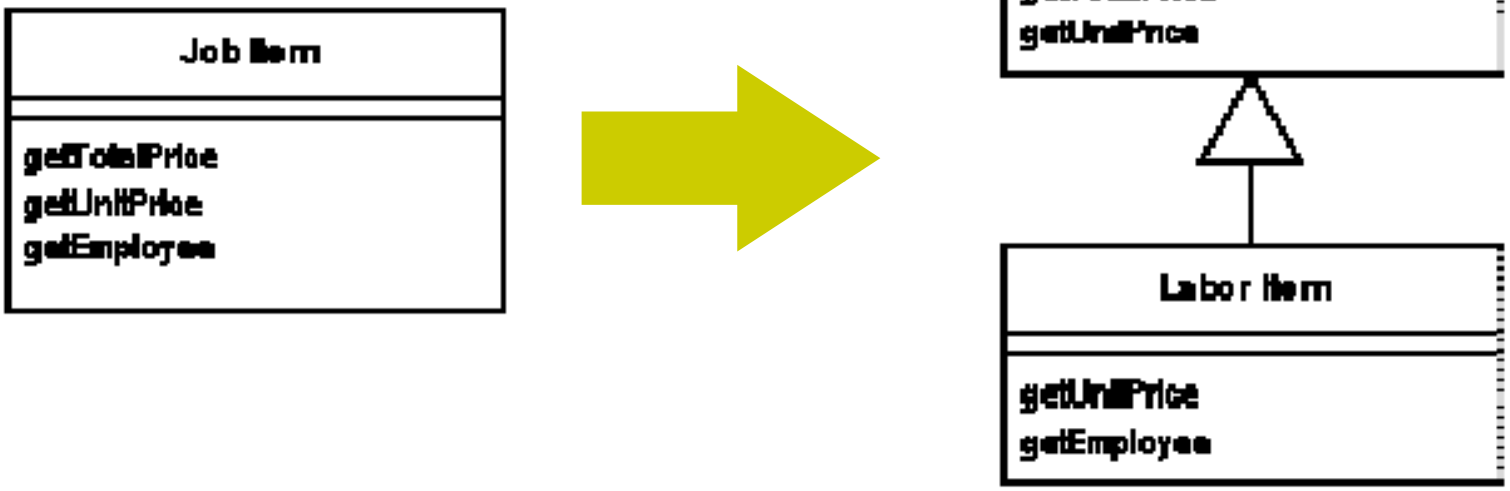
Ejemplo: Push Down Method



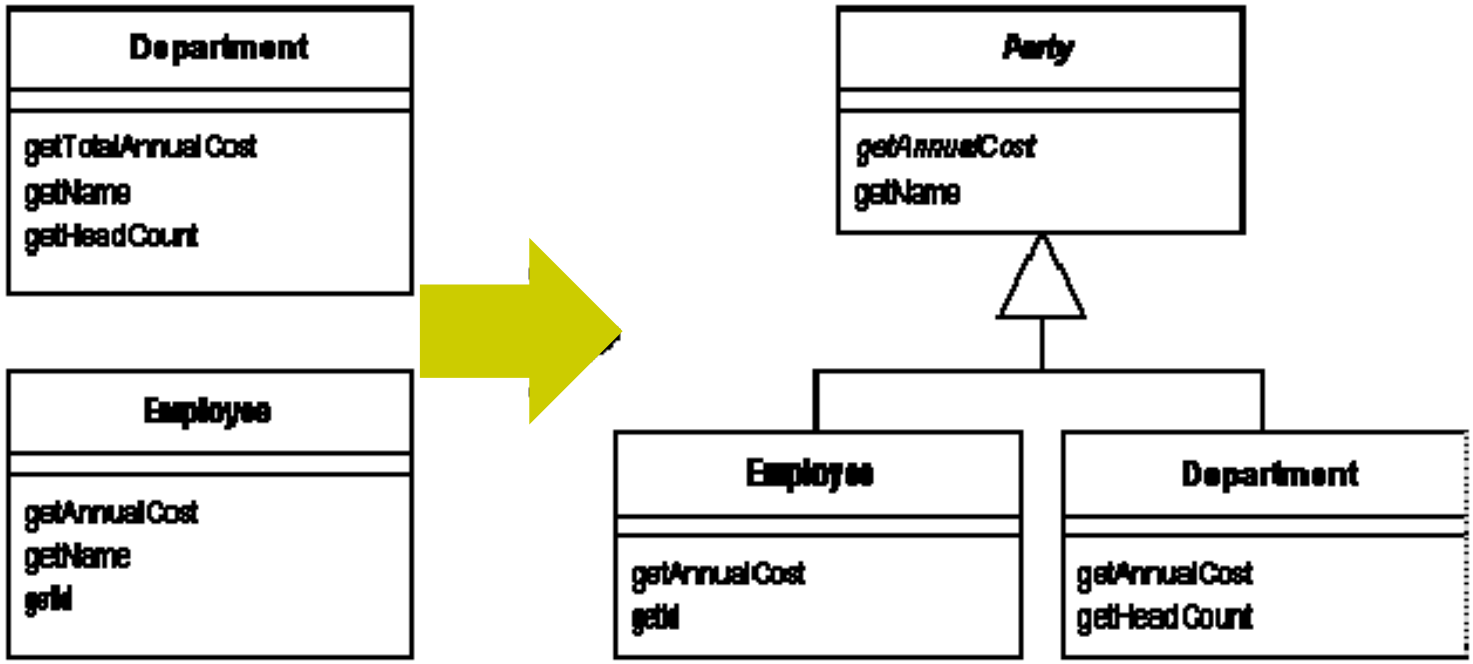
Ejemplo: Push Down Field



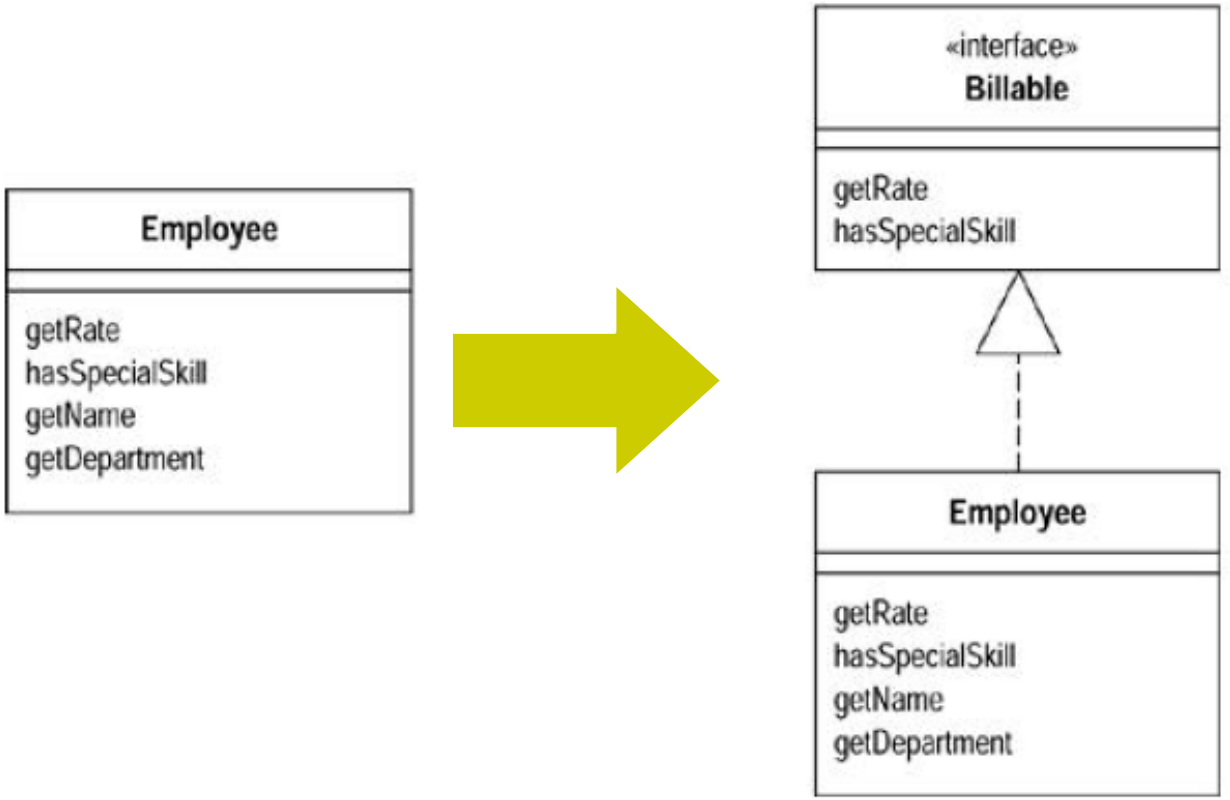
Ejemplo: Extract Subclass



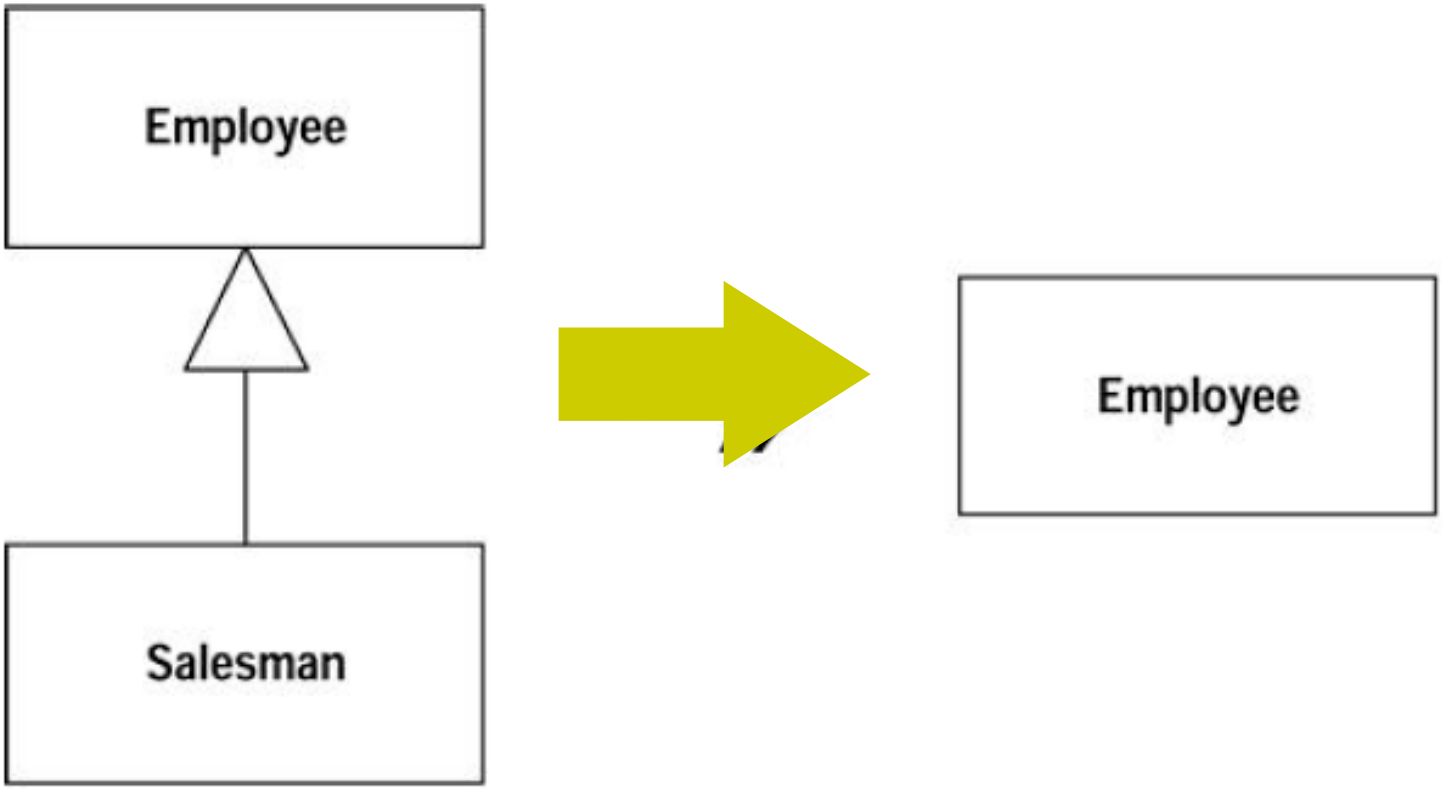
Ejemplo: Extract Superclass



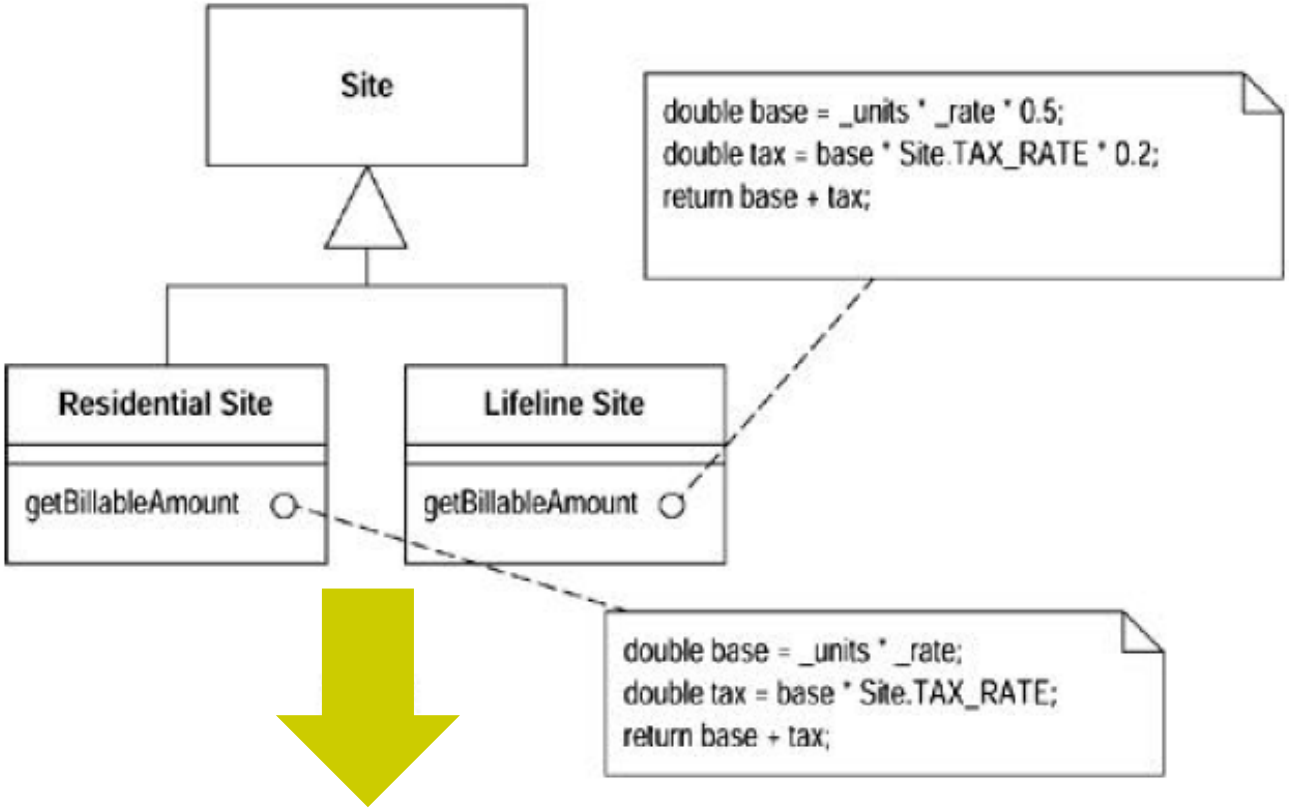
Ejemplo: Extract Interface



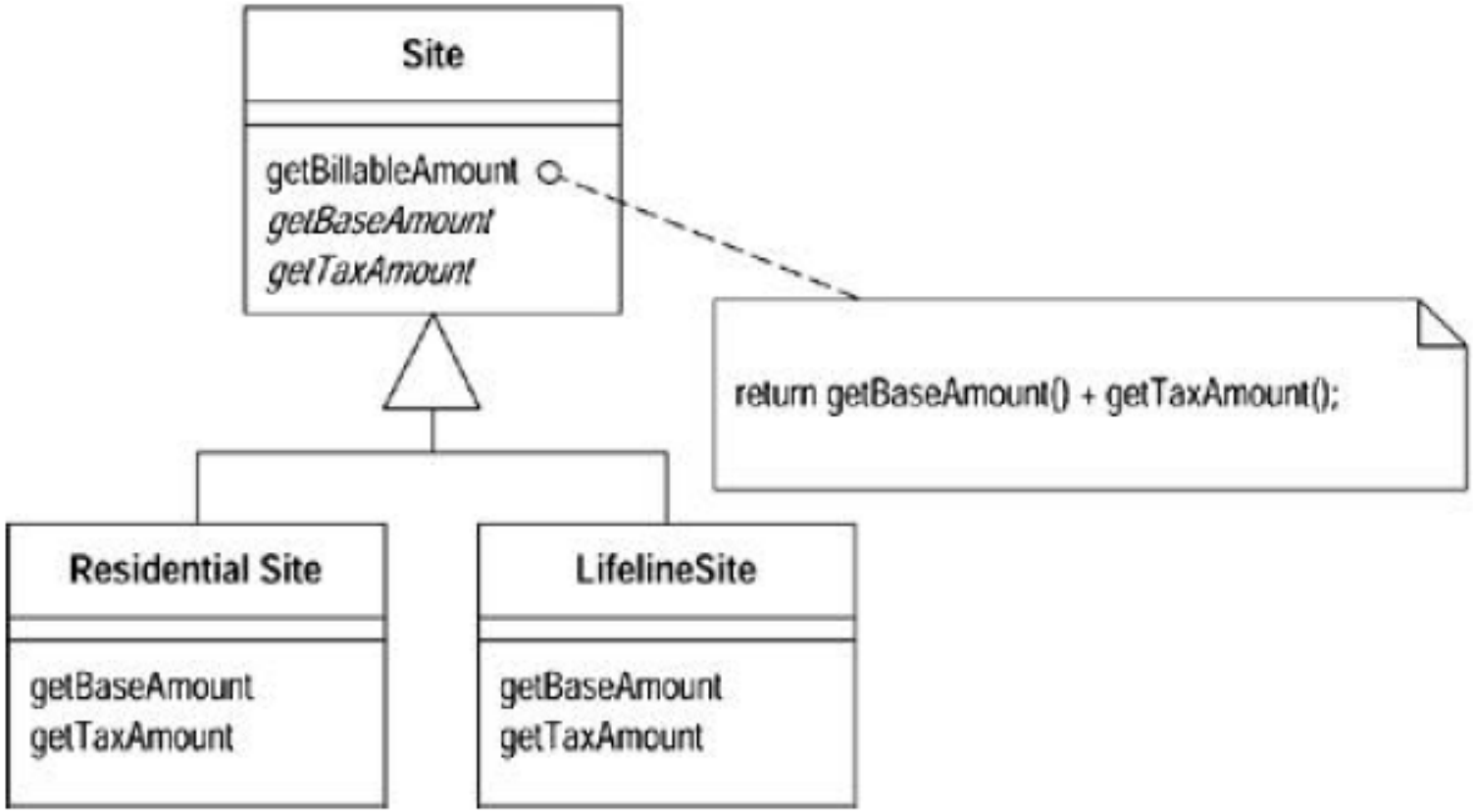
Ejemplo: Collapse Hierarchy



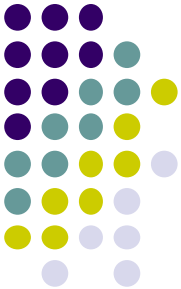
Ejemplo: Form Template Method

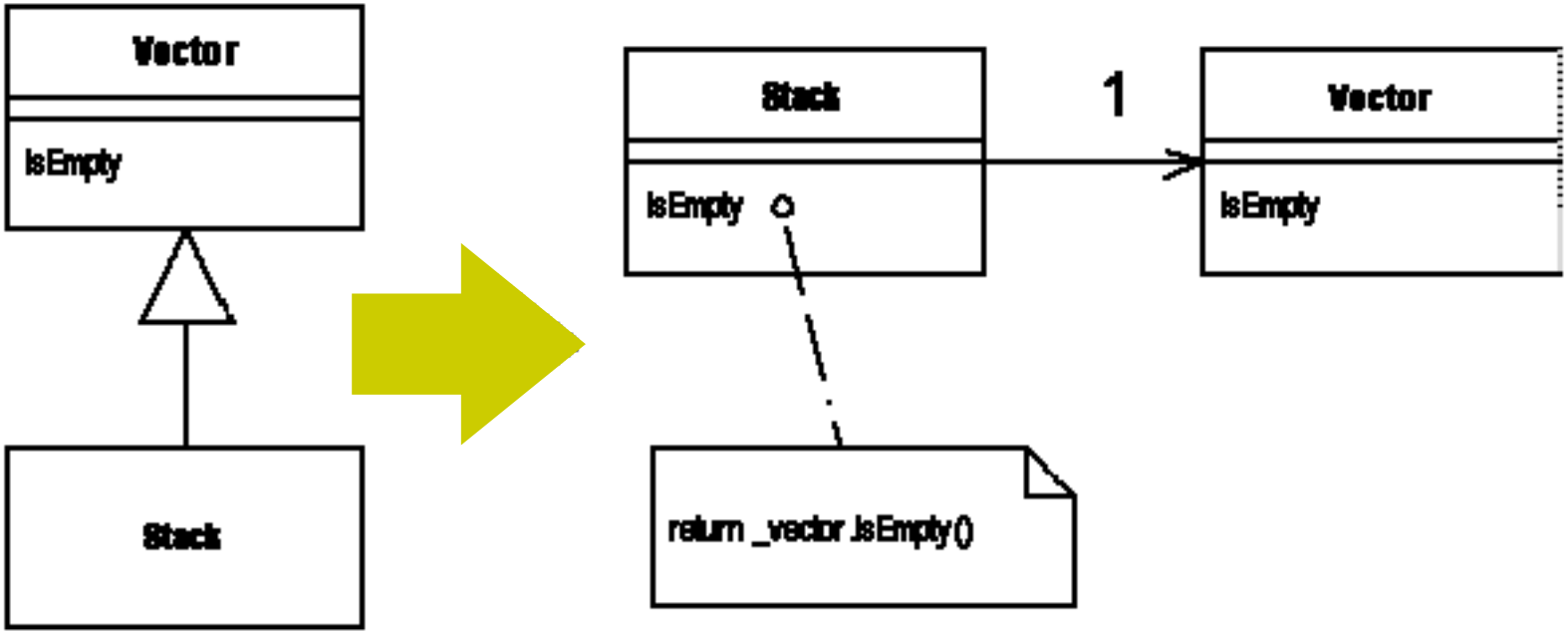


Ejemplo: Form Template Method

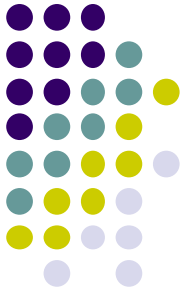


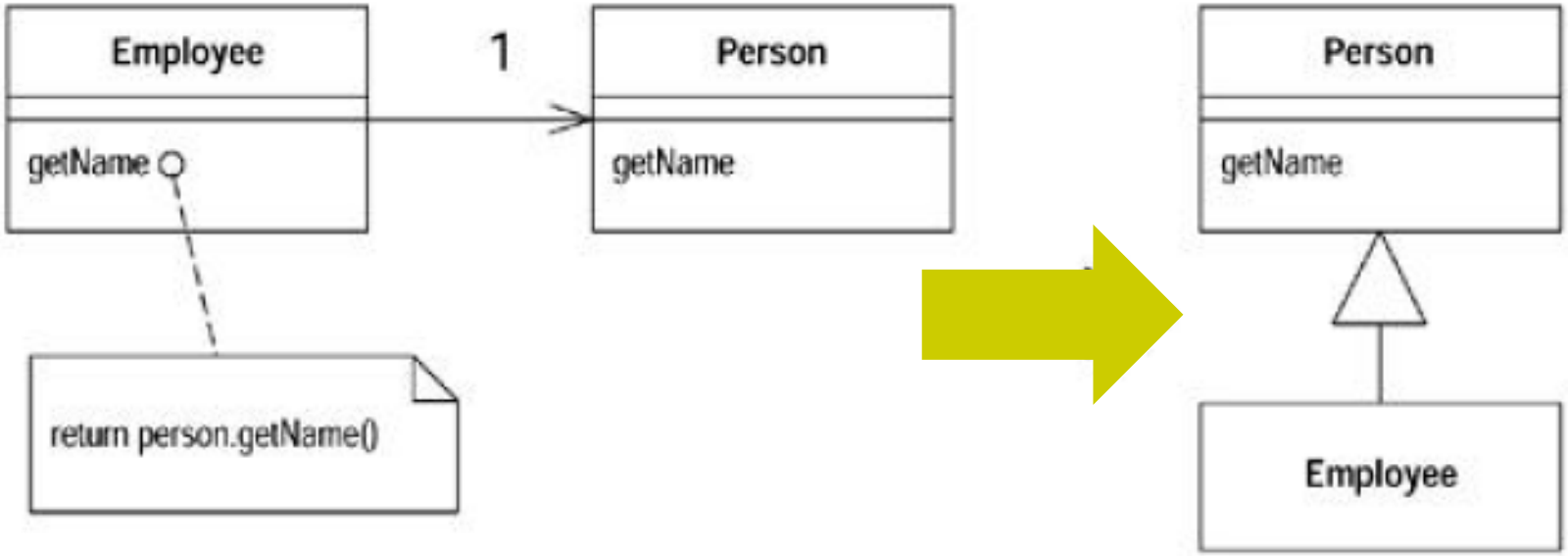
Ejemplo: Replace Inheritance with Delegation

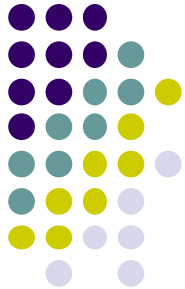




Ejemplo: Replace Delegation with Inheritance



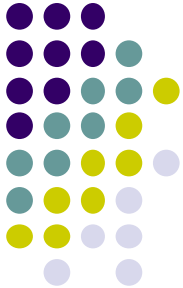


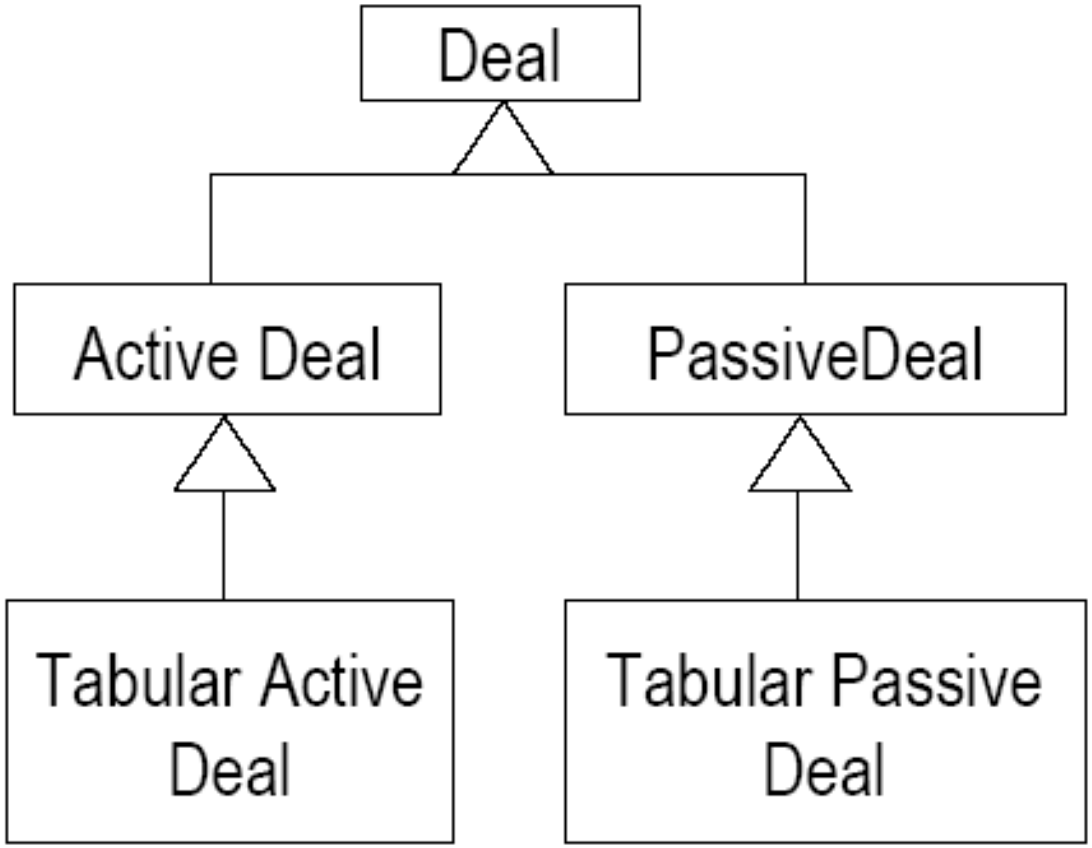


Refactoring

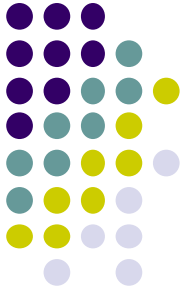
- Tease Apart Inheritance
- Convert Procedural Design to Object
- Separate Domain from Presentation
- Extract Hierarchy

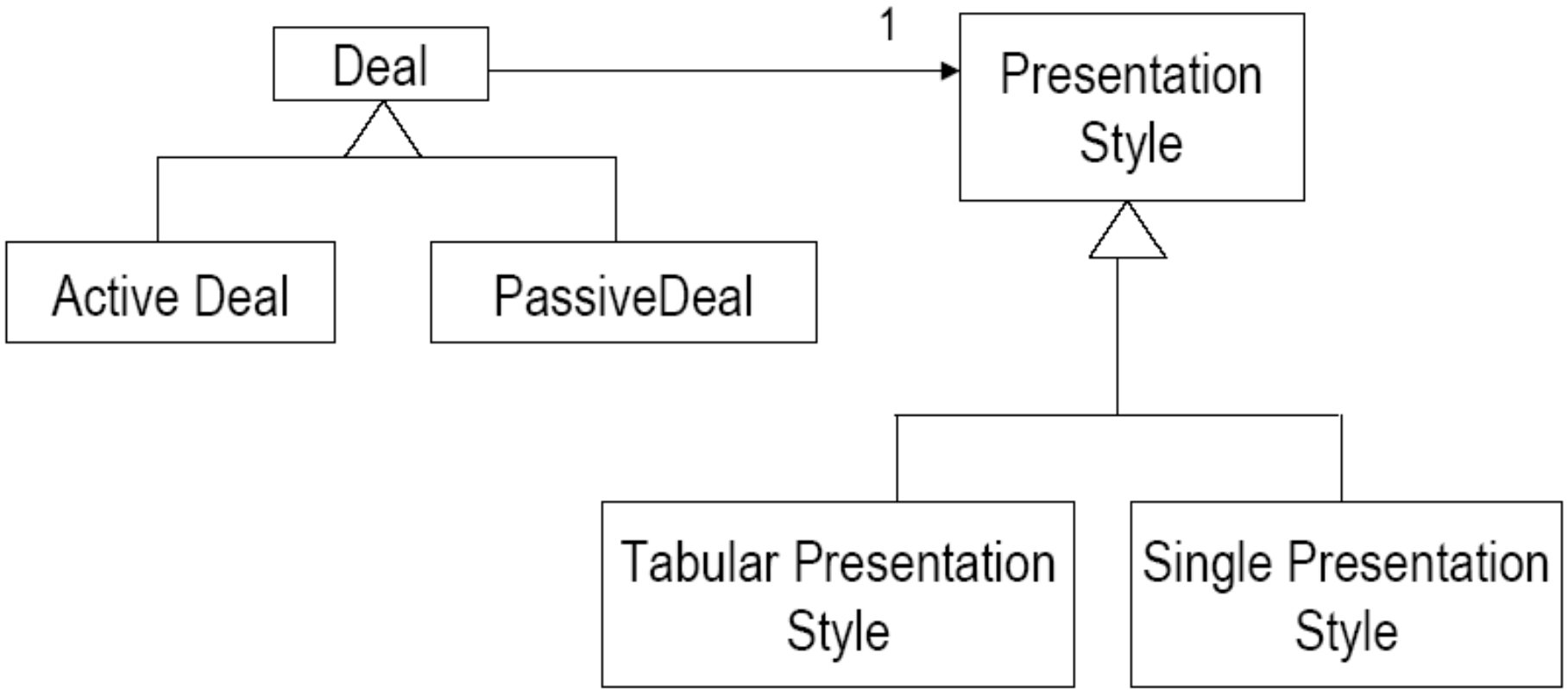
Ejemplo: Tease Apart Inheritance



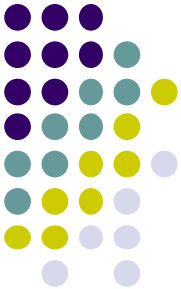


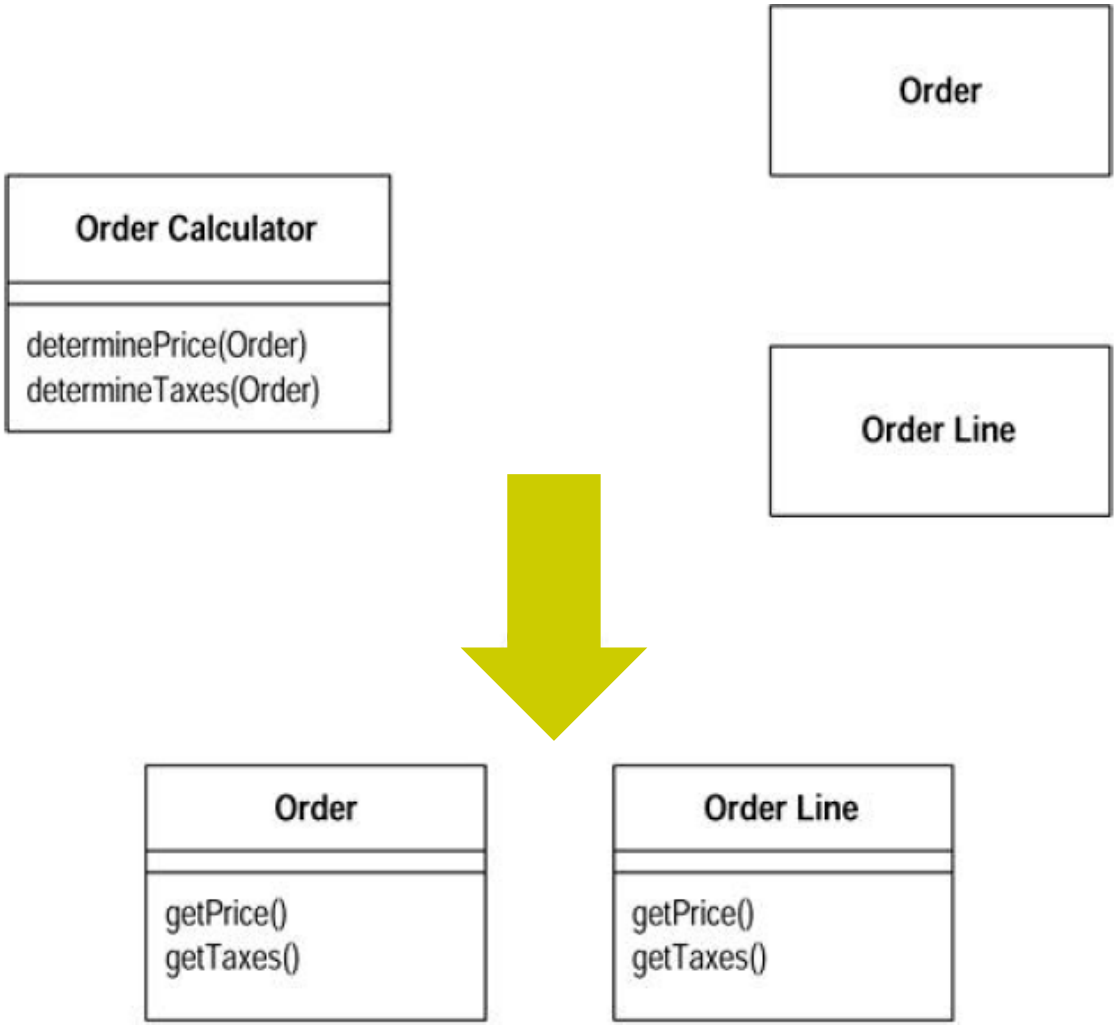
Ejemplo: Tease Apart Inheritance



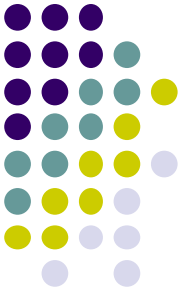


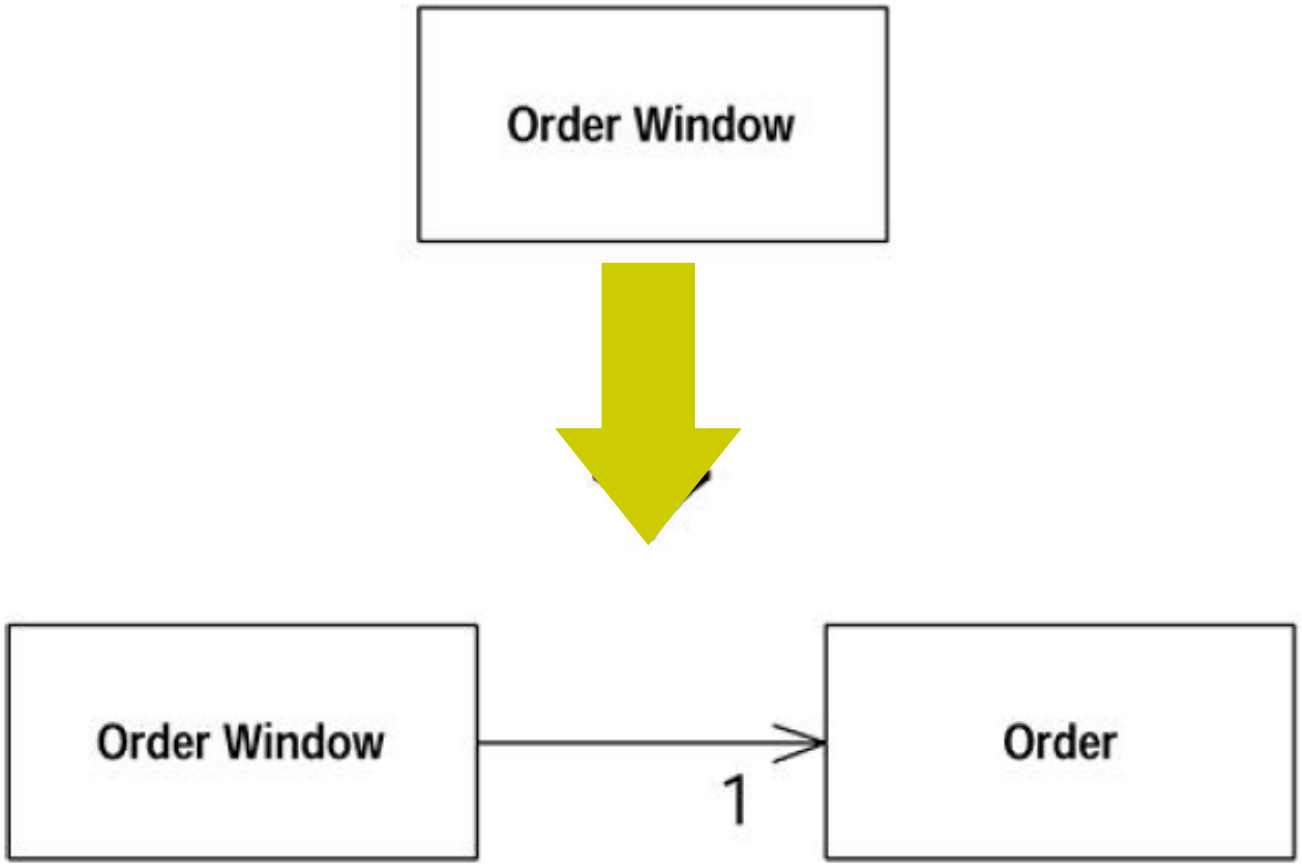
Ejemplo: Convert Procedural Design to Object



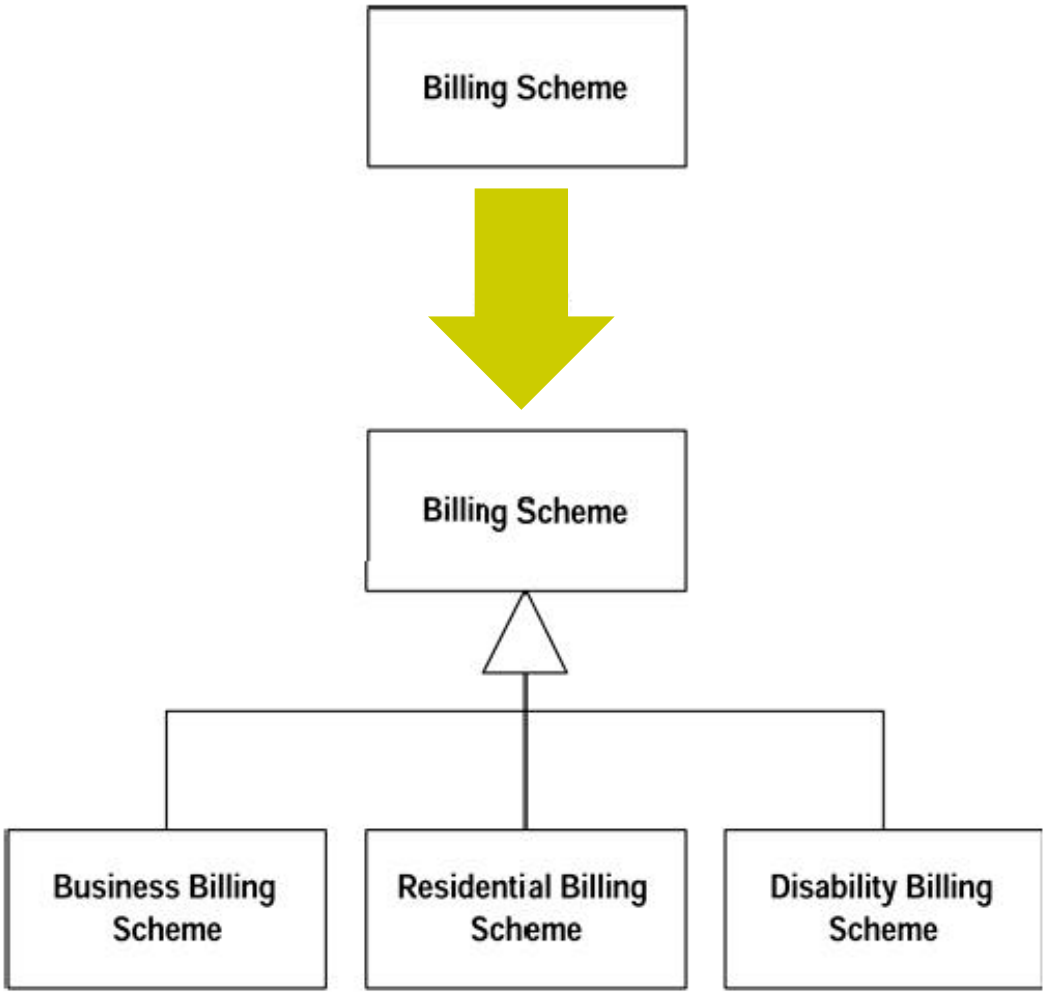


Ejemplo: Separate Domain from Presentation





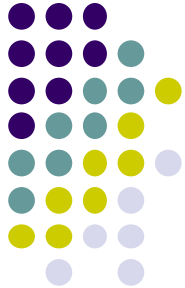
Ejemplo: Extract Hierarchy



Bad smells en el código

- ***Duplicated code***
  - "El *bad smell* número #1"
  - ¿La misma expresión en dos métodos de la misma clase?
    - Crear una rutina/método auxiliar `privada` y parametrizarla
      (Extract method)
  - ¿El mismo código en dos clases relacionadas?
    - Llevar las partes comunes en una clase antecesora común y parametrizar
    - Usar el PD *template method* para dar soporte a la variación en subtareas
      (Form template method)

Bad smells en el código



- ***Duplicated code***
  - ¿El mismo código en dos clases no relacionadas?
    - ¿Deberían estar relacionadas?
      - Introducir un padre abstracto *(Extract class, Pull up method)*
    - ¿El código pertenece realmente a una clase?
      - Convertir la otra clase en un cliente *(Extract method)*
    - ¿Podemos separar/extraer las partes comunes en una subparte/función u objeto función (estrategia)?
      - Convertir un método en un subojeto de ambas clases.
      - El PD *Strategy* nos permite la variación polimórfica de métodos-como-objetos

      (Replace method with method object)

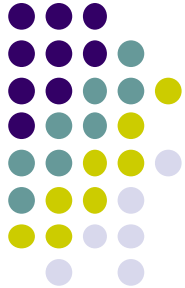
Bad smells en el código

- ***Long method***
  - A veces es un signo de:
    - Intentar hacer demasiadas cosas
    - Abstracciones poco pensadas
  - Lo mejor es pensar cuidadosamente sobre las principales tareas que realiza y cómo se relacionan.
    - Romperlo en pequeños métodos `private` en la clase

      (Extract method)
    - Delegar las subtareas en subobjetos (*por ejemplo el PD template method*)

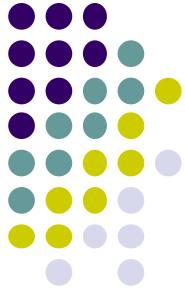
      (Extract class/method, Replace data value with object)

Bad smells en el código



- ***Long method***
  - Heurística de Fowler:
    - *Cuando veas un comentario, crea un método.*
    - A veces, un comentario indica:
      - El siguiente paso importante (en el código)
      - Algo que no es obvio cuyos detalles impiden comprender con claridad la rutina como un todo.
    - En cualquier caso, es bueno dividirlo en partes.

Bad smells en el código

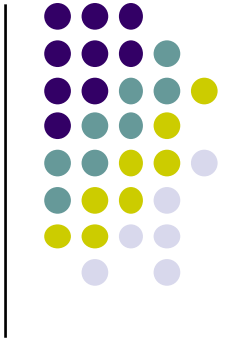


- ***Large class***
  - Demasiadas subpartes y métodos diferentes
  - Un solución en dos pasos:
    - Reunir/Agrupar las piezas pequeñas en subpartes agregadas.
      (Extract class, replace data value with object)
    - Delegar los métodos a las nuevas subpartes.
      (Extract method)
  - Contraejemplo:
    - Las clases de biblioteca a veces tienen interfaces grandes (muchos métodos, muchos parámetros, mucha sobrecarga)
      - Si existen tantos métodos por cuestiones de flexibilidad, entonces dicha clase de biblioteca estará OK.

Bad smells en el código

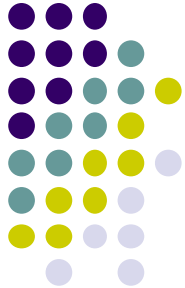
- ***Long parameter list***
  - Las listas de parámetros largas hacen que los métodos sean difíciles de entender por los clientes
  - Esto suele ser un síntoma de:
    - Intentar hacer demasiado
    - … "lejos de casa" (muchos niveles de llamada, camino de acceso largo)
    - … con demasiadas subpartes separadas (no cohesionadas)

Bad smells en el código



- ***Long parameter list***
  - Antiguamente, la programación estructurada enseñó el uso de la parametrización para evitar usar variables globales.
    - Con los módulos y la OO, los objetos poseen mini islas de estado que pueden ser tratadas como globales para los métodos (mientras siguen ocultas al resto del programa).
    - No es necesario pasar a uno de nuestros métodos una subparte de uno mismo como parámetros.

Bad smells en el código



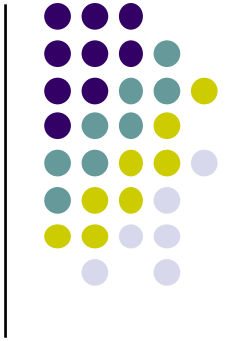
- ***Long parameter list***
  - Solución:
    - ¿Realmente lo necesito?
      - Obtenerlo a través de un objeto

      (Replace parameter with method)
    - ¿Intentando hacer demasiado?
      - Romperlo/Dividirlo en subtareas

      (Extract method)
    - ¿… con demasiadas subpartes separadas (no cohesionadas)?
      - Reunir/Agrupar los parámetros en subpartes agregadas.

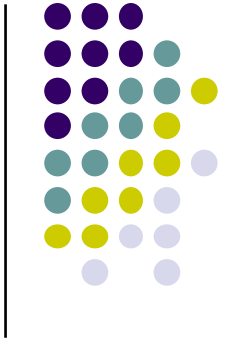
      (Preserve whole object, introduce parameter object)

Bad smells en el código



- ***Divergent change***
  - Ocurre cuando una clase se cambia comúnmente de formas distintas por diferentes razones
  - Lo más seguro, es que esta clase esté intentando hacer demasiado y contenga demasiadas subpartes no relacionadas
  - Con el tiempo, algunas clases desarrollan el "Complejo de Dios"
    - Adquieren detalles/dependencias/propiedades de subpartes que pertecenen a cualquier otro
  - Este es un signo de un cohesión pobre
    - Elementos no relacionados en el mismo contenedor
  - Solución:
    - Dividirla/Romperla, reconsiderar relaciones y responsabilidades *(Extract class)*

Bad smells en el código



- ***Shotgun surgery***
  - … lo opuesto al cambio divergente
    - Cada vez que quieres realizar un cambio sencillo y a primera vista coherente, tienes que cambiar muchas clases un poco cada una.
  - También es un signo clásico de cohesión pobre
    - Los elementos relacionados no están en el mismo contenedor!
  - Solución:
    - Intentar algún agrupamiento, en una clase nueva o existente.

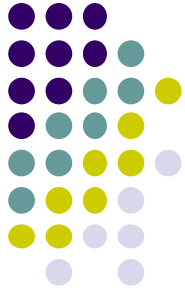
    (Move method/field)

Bad smells en el código

- ***Feature envy***
  - Un método parece estar más interesado en otra clase que en aquella en la que está definido.
    - *Por ejemplo, un método* `A::m()` llama a muchos métodos get/set de la clase `B`
  - Solución:
    - Mover `m()` (o parte de él) a `B`!

    (Move method/field, extract method)
  - Excepciones:
    - Los PD *Visitor/iterator/strategy* donde el propósito es desacoplar los datos del algoritmo
    - Feature envy es un problema más complejo cuando los dos métodos `A` y `B` poseen datos interesantes

Bad smells en el código



- ***Data clumps***
  - Vemos un conjunto de variables que parece que siempre van juntas
    - Pasadas como parámetros, cambiadas y/o accedidas al mismo tiempo
  - Normalmente, esto significa que existe un subobjeto coherente esperando ser reconocido y encapsulado
  - Solución:

(extract class, introduce parameter object)

Bad smells en el código

- ***Primitive obsession***
  - Todas las subpartes de un objeto son instancias de tipos primitivos

    (`int`, `string`, `bool`, `double`, *etc.*)

    Por ejemplo, fechas, monedas, DNI, tel.#, ISBN, etc…
  - A veces, estos objetos pequeños poseen restricciones no triviales que deben ser modeladas

    Por ejemplo, un número fijo de dígitos o caracteres, valores especiales, etc…
  - Solución:
    - Crear algunas clases "pequeñas" que pueden validar y hacer cumplir las restricciones.
      - Esto hace nuestro sistema más robusto y fuertemente tipado.

    (Replace data value with object, extract class, introduce parameter object)

Bad smells en el código

- ***Switch statements***
  - Ejemplo de Fowler:

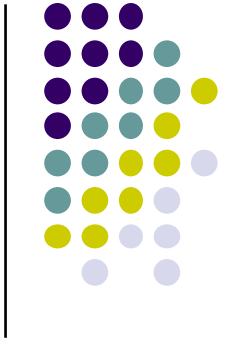
```
Double getSpeed () {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() -
                getLoadFactor() * _numCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0
                : getBaseSpeed(_voltage);
    }
}
```

Bad smells en el código

- ***Switch statements***
  - Falla comprensión del polimorfismo y falta de encapsulación.
  - Solución:
    - Rediseñarlo como un método polimórfico

    (Replace conditional with polymorphism, replace type code with subclasses)

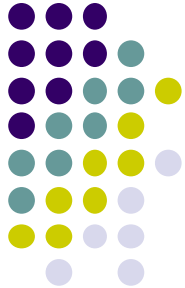
Bad smells en el código



- ***Lazy class***
  - Clases que no hacen mucho diferente de otras clases.
  - ¿existen diversas clases hermanas que no exhiben diferencias de comportamiento polimórfico?
    - Solución:
      - consideraremos unificarlas en un clase padre y añadiremos algunos parámetros

      *(Collapse **hierarchy**, inline class)*
  - A veces, las ***lazy classes*** se generan en diseños ambiciosos o en refactorizaciones que destripan a la clase de comportamiento interesante

Bad smells en el código



- ***Speculative generality***
  - *"Lo necesitaremos algún día …"*
    - Vale, pero, ¿realmente lo necesitamos después de todo?
    - Clases y características extra que añaden complejidad.
  - Tener en mente que el refactoring es un proceso continuo.
    - Si realmente lo necesitamos más tarde, lo podemos añadir después.

    (Collapse hierarchy, inline class, remove parameter)

Bad smells en el código

- ***Message chains***
  - Un cliente llama a un objeto que llama a un subobjeto, que llama a un subobjeto, …
    - Este envio de peticiones multinivel puede dar como resultado que se pasen al cliente sub-sub-sub-objetos.
  - Parece que el cliente sepa la estructura del objeto, incluso si va a pasar a través de intermediarios apropiados.
  - Probablemente es necesario repensar la abstracción…
    - ¿Por qué esta profunda anidación de llamadas y objetos?

    (Hide delegate)

Bad smells en el código

- ***Middle man***
  - La encapsulación viene ligada muchas veces a la delegación
  - La delegación puede ir demadiado lejos (nos damos cuenta de que muchos métodos de clase sólo giran alrededor y piden servicios de subojetos delegados)
  - Probablemente se tenga una abstracción esté poco pensada.
  - Un objeto debe ser algo más que el conjunto de sus partes en términos de su comportamiento!

    (Remove middle man, replace delegation with inheritance)

Bad smells en el código

- ***Inappropriate intimacy***
  - Compartir secretos entre clases, especialmente fuera de las fronteras de la herencia

    Por ejemplo: variables `public`, definiciones indiscriminadas de métodos get/set, clases amigas en C++, datos `protected`
  - Nos lleva a un alto acoplamiento de los datos, a un conocimiento íntimo de las estructuras internas y decisiones de implementación.
    - Hace a los clientes brittle, difíciles de evolucionar, fáciles de romper.
  - Solución:
    - *Uso apropriado* de los métodos get/set
    - Repensar la abstracción.
    - Mezclar clases si se descubre una "relación verdadera"

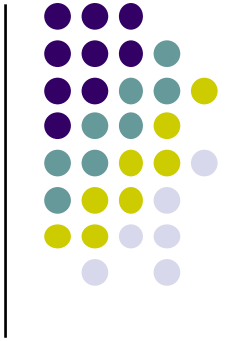
    (Move/extract method/field, change bidirectional association to unidirectional, hide delegate)

Bad smells en el código

- ***Alternative classes with different interfaces***
  - Clases/métodos que parecen implementar la misma o similar abstracción no están relacionadas.
  - Solución:
    - Juntar las clases cercanas.
      - Encontrar un interfaz común.
      - Encontrar un subparte común y eliminarla

    (Extract [super]class, move method/field, rename method)

Bad smells en el código



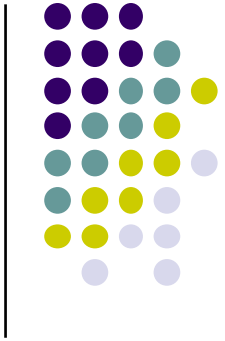
- ***Data class***
  - La clase está constituida sólo por campos de datos (sencillos) y métodos de acceso simples.
  - A veces las clases de datos son razonables si se utilizan con sentido.
  - *"La clases de datos son como niños. Están bien como punto de partida, pero para participar como un objeto adulto, necesitan tomar alguna responsabilidad"*
  - Solución:
    - Mirar en los clientes los patrones de uso que éstos siguen
    - Intentar abstraer algunas características de uso comunes en métodos de la clase y apartar alguna funcionalidad

    (Extract/move method)

Bad smells en el código

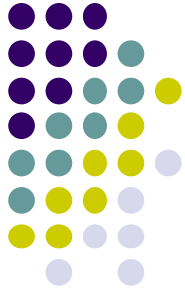
- ***Refused bequest***
  - Las sublcases heredan métodos/atributos pero parece que no usan algunos de ellos.
    - En este sentido, puede ser un buen signo:
      - El padre gestiona las partes comunes y el hijo gestiona las diferencias.
  - ¿La subclase hereda la funcionalidad pero no desea soportar el interfaz del padre?
    - Es mejor utilizar delegación
      (Replace inheritance with delegation)

Bad smells en el código



- ***Refused bequest***
  - Otra perspectiva:
    - Los padres tienen características que son utilizadas por algunos de sus hijos.
  - La solución tipica es crear algunas clases más intermedias como clases abstractas en la jerarquía.
    - Trasladar los métodos peculiares en las clases de los últimos niveles de la jerarquía.
      (Push down field/method)

Bad smells en el código

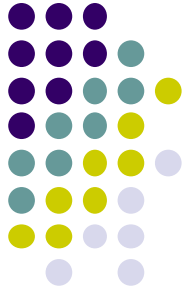


- ***Comments***
  - La filosofía de XP no recomienda los comentarios del código:
    - En vez de ello, hacer los métodos más cortos y usar identificadores largos
    - Fowler no es tan crítico con esto.
  - Fowler dice que los comentarios largos son, muchas veces, signo de código opaco y/o complicado.
    - No está tan en contra de los comentarios tanto como lo está a favor de prácticas de códificación autodocumentadas.
    - En vez de explicar código opáco, reestructuralo!!!!

    (Extract method/class, [many others applicable] …)
  - Los comentarios se usan adecuadamente para documentar el conocimiento, los razonamientos y/o las decisiones tomadas

    Por ejemplo, explicar porqué hemos elegido una opción frente a otra

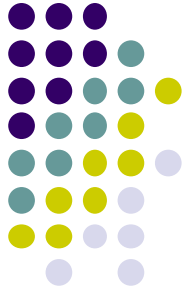
Referencias a Refactoring



Podemos conocer más sobre refactoring y code smells:

- El libro de *Refactoring* [Fowler] http://www.refactoring.com
- Discusiones sobre *code smells* http://c2.com/cgi/wiki?CodeSmell

Herramientas de Refactoring



- Smalltalk
  - Smalltalk Refactoring Browser
- Java
  - IntelliJ Idea
  - Eclipse
  - JFactor
  - XRefactory
  - Together-J
  - JBuilder
  - RefactorIt
  - JRefactory
  - Transmogrify
  - JafaRefactor
  - CodeGuide
- .NET
  - C# Refactory
  - C# Refactoring Tool