Ingeniería de Software I
Mantención y Evolución

Jocelyn Simmonds

Departamento de Ciencias de la Computación

Proceso de cambio de software

El programador debe primero buscar los módulos que deben ser cambiados para implementar un cierto cambio (Concept Location).

Los cambios pueden propagarse, así que hay que hacer un Impact Analysis después de cada cambio.

Concept Location

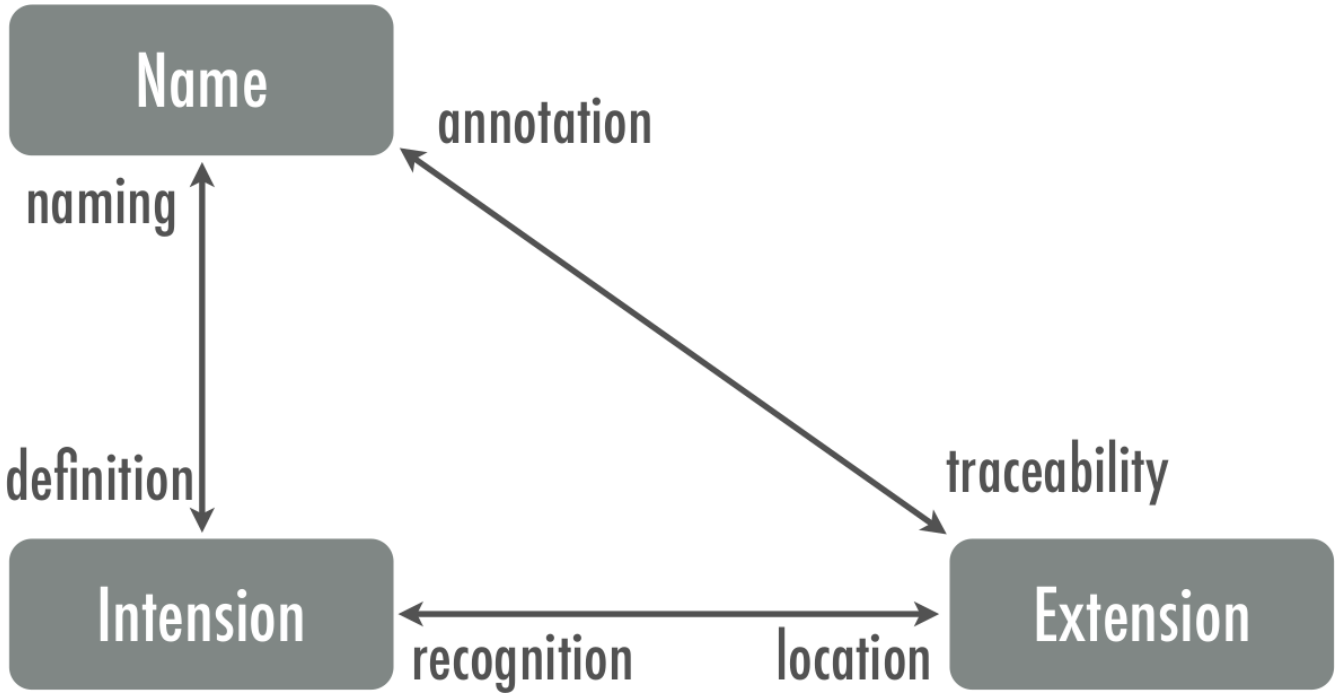
- Los conceptos son el puente nuestra implementación y la visión del cliente, usuarios, etc.
- Cada vez que queremos hacer un cambio, debemos identificar los conceptos que están involucrados en este . . .
  - Ejemplo: "Corrijan el error que ocurre cuando trato de pegar texto en la interfaz gráfica"
  - El programador debe identificar las lineas de código donde esta implementado el concepto "pegar"

Comprensión del código

Concept Location es difícil:

- Nuestros sistemas son cada vez más grandes y complejos . . . asi que es dificil comprenderlos por completo
- En general, buscamos entender lo mínimo necesario para poder realizar un cierto cambio
- La idea es entender como ciertos conceptos están implementados, para poder cambiarlos adecuadamente

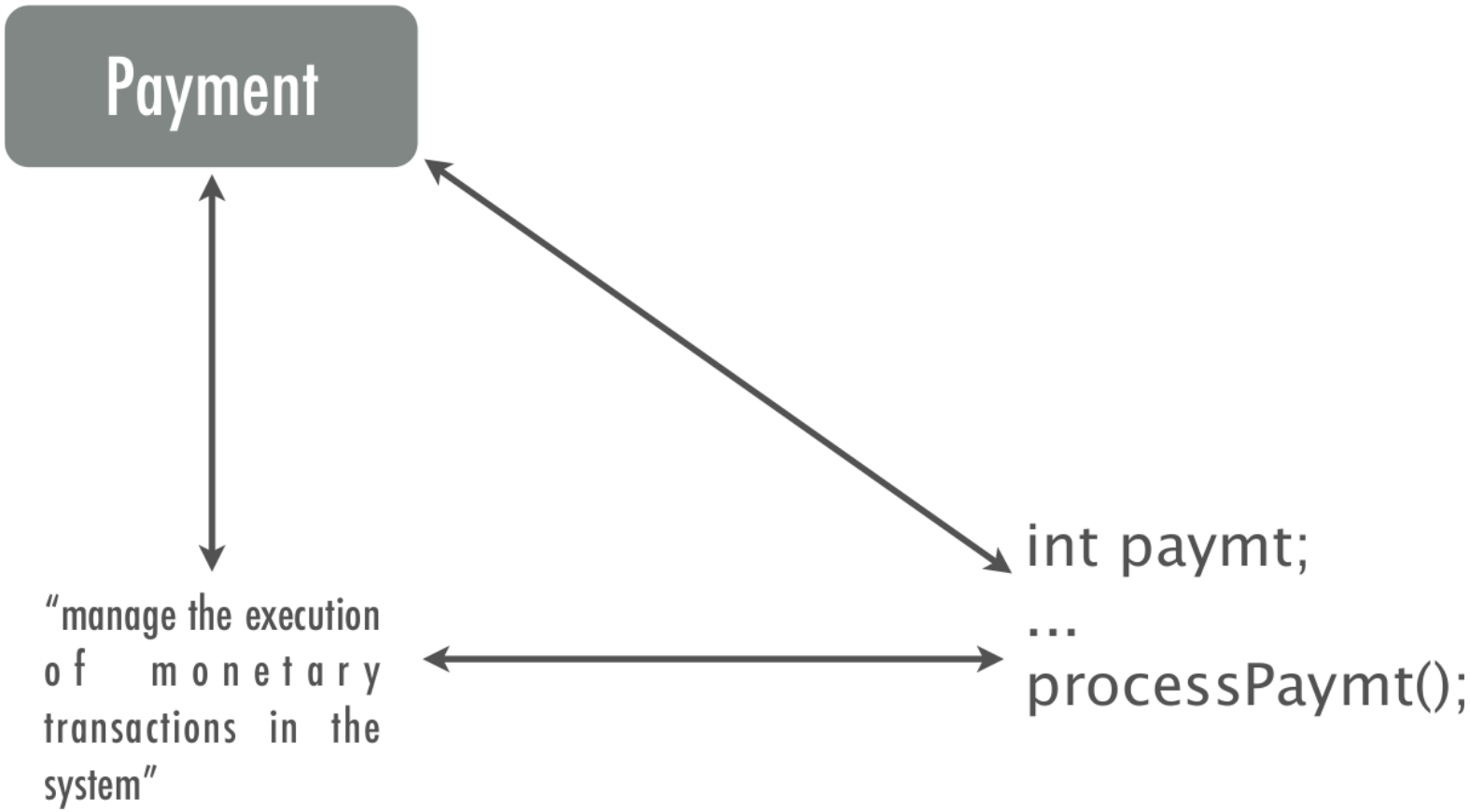
Las distintas facetas de un concepto



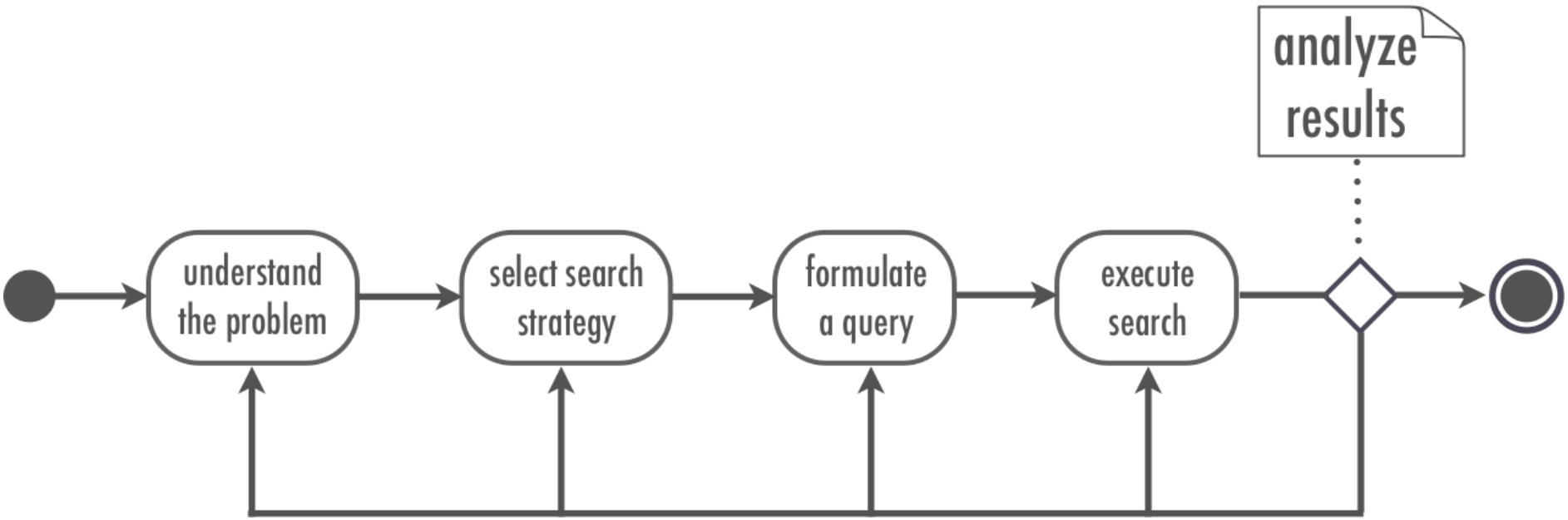
Intension: the internal content of a concept.

Extension: the range of a term or concept as measured by the objects that it denotes or contains, as opposed to its internal content.

Ejemplo


Payment
"manage the execution of monetary transactions in the system"
int paymt;
...
processPaymt();

Proceso de Concept Location



La búsqueda parte de Intension hacia Extension ($\rightarrow$): documentación, requerimientos, diseño, código, trazas de ejecución

Conceptos en el código

Podemos encontrar los conceptos:

- en forma explícita: aparecen directamente en el código
- en forma implícita: el código hace alusiones al concepto, pero no aparece propiamente como tal en el código

Ejemplo:

- explícito: un contador del numero de paginas en un editor de texto
- implícito: la autorización para ver el contenido de un archivo

. . . además

Un solo cambio puede involucrar muchos conceptos . . .

- Hay que analizar cada cambio que nos piden
- Extrayendo los conceptos presentes en el cambio
- Estos conceptos nos permitirán comunicarnos mejor con los programadores
- . . . y podemos usarlos para filtrar el código
- Para llegar a una lista de conceptos que posiblemente serán afectados por el cambio

Ejemplo

"Implementar el pago con tarjeta de crédito"

Conceptos:

- "Implementar": comunicación con el programador
- "tarjeta de crédito": concepto que sera implementado
- "pago": concepto más importante del cambio, encapsula lo principal de la funcionalidad a implementar

Estrategias para Concept Location

Supuesto: concept extensions están identificados por nombre

$$\text{intension} \rightarrow \text{name} \rightarrow \text{extension}$$

Podemos hacer un grep para buscar el concepto por nombre.

Estrategias para Concept Location

Supuesto: concept extensions están identificados por nombre

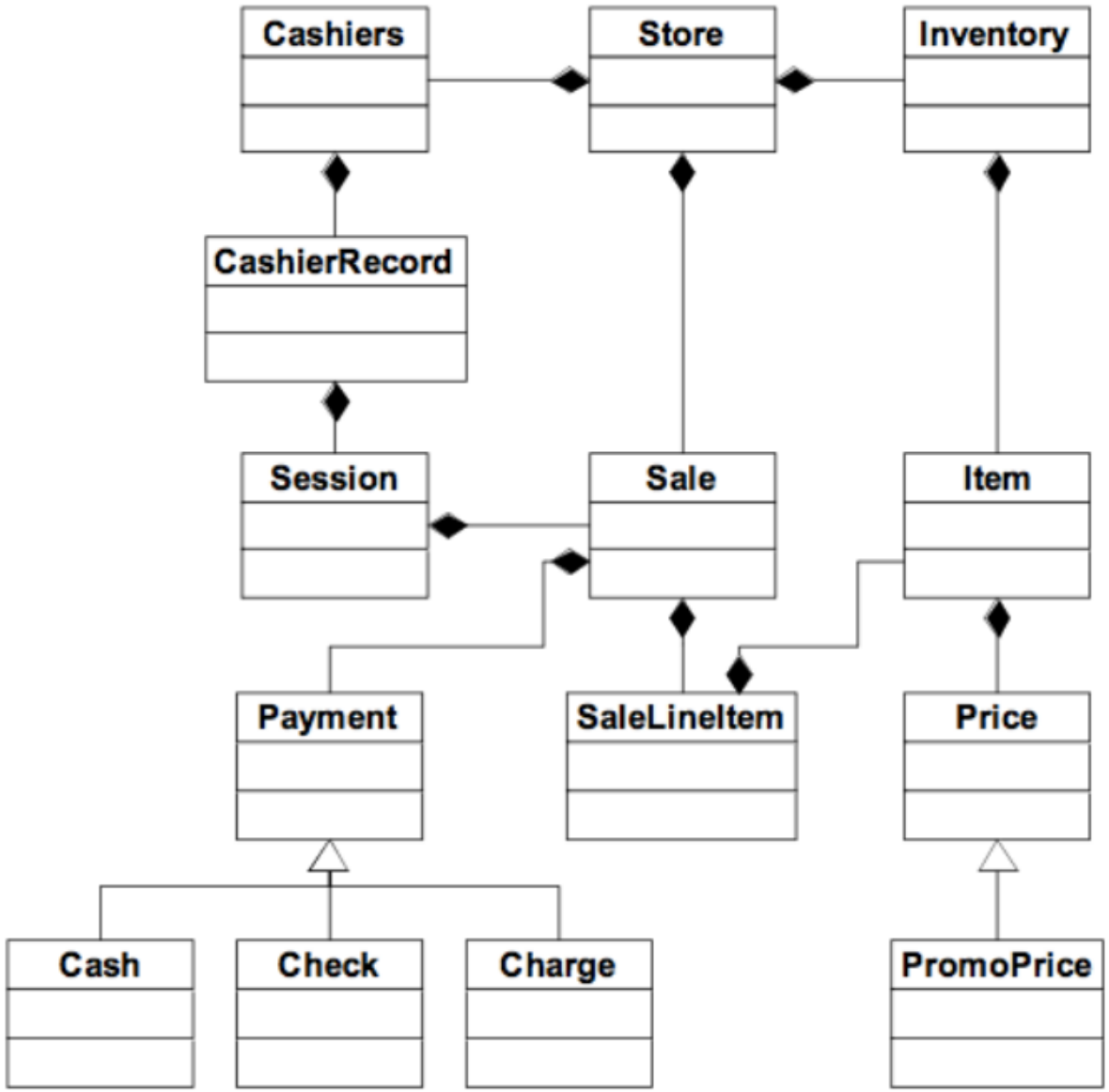
$$\text{intension} \rightarrow \text{name} \rightarrow \text{extension}$$

Podemos hacer un grep para buscar el concepto por nombre.

Dificultades con esta estrategia:

- Homónimos, sinónimos, etc. "Pago" vs. Gasto, Reembolso, Pg, Payment, ...
- Si no hay aciertos: buscar con otro patrón
- Muchos aciertos equivocados: posiblemente en el caso de homónimos
- Demasiados aciertos: refinar la búsqueda
- No funciona muy bien con conceptos implícitos

Estrategia 2: revisar las dependencias entre clases



Proceso de cambio de software

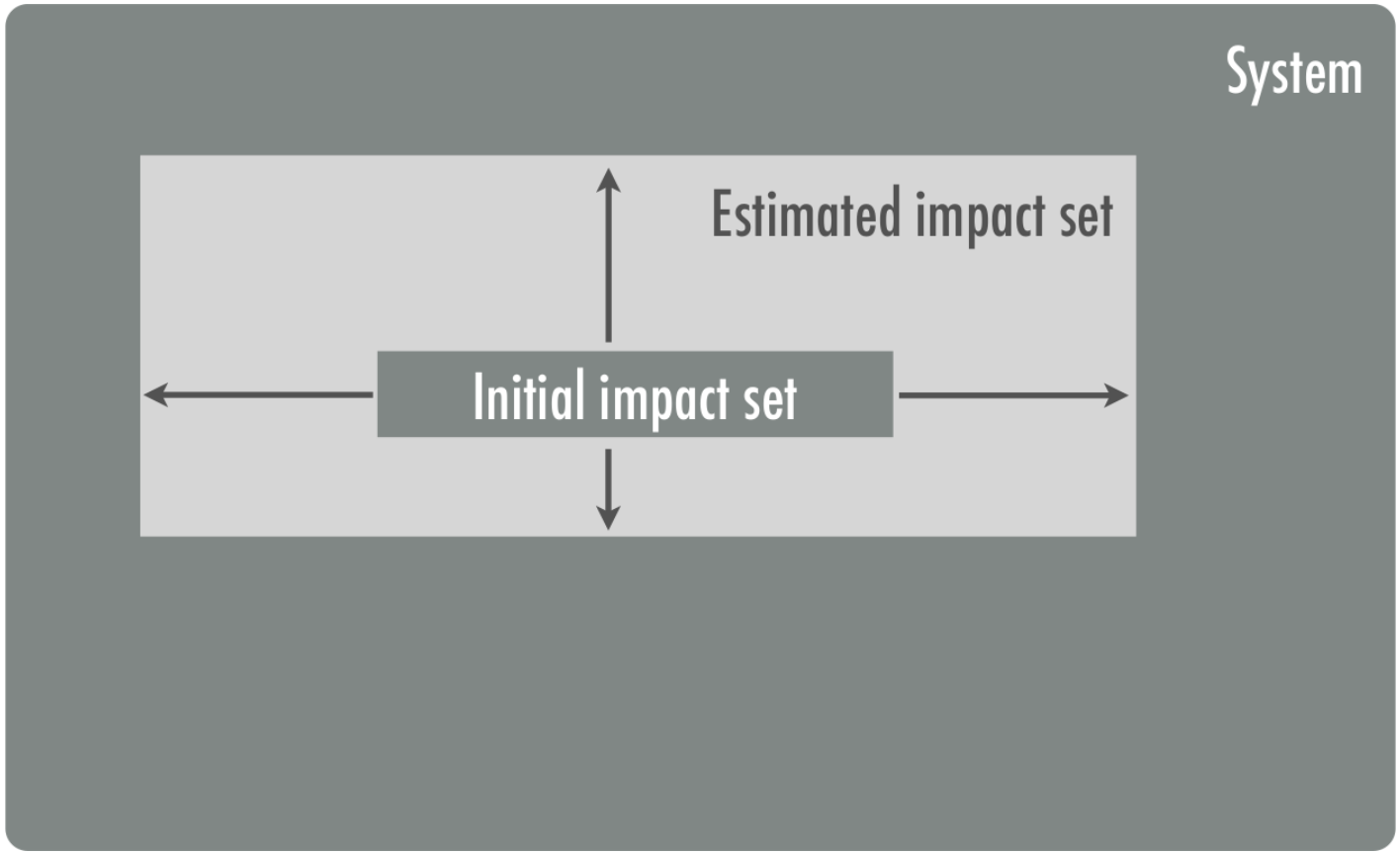
Ok, usando Concept Location encontramos el modulo que debemos cambiar . . . ahora que?

Debemos ver que otros módulos pueden cambiar (Impact Analysis).

Impact Analysis

- En esta fase, debemos elegir la estrategia que usaremos para implementar el cambio, y ver que impacto tendrá
- Las clases, métodos, etc. identificados en Concept Location pasan a formar el "Initial impact set"
- Tenemos que analizar las dependencias entre clases para generar el "Estimated impact set"
  - o sea, las clases que posiblemente serán afectadas por el cambio que implementaremos en el Initial impact set

Estimated Impact Set



Actual Impact Set

Ojo: es usual que el estimated y actual impact set sean distintos, los programadores sub-estiman los cambios a hacer

Interacción entre clases

Vamos a decir que dos clases interactúan si tienen algo en común

- una depende de la otra, o sea, hay algún "contrato" de por medio
- las dos clases trabajan en forma coordinada
  - comparten algo, como código, datos, etc.
- las interacciones se propagan en ambas direcciones

Interacción entre clases

Ojo con confundir interacción y dependencia:

```
class A {
   int getUserColor();
}

class B {
   void paintScreen(int color);
}

class C {
   A a;
   B b;

   void askThenPaint() {
      b.paintScreen(a.getUserColor());
   }
}
```

Clase C usa clases A y B:

- estas clases declaran variables globales
- y asumen ciertas cosas, como que "azul" $= 1$

Interacción entre clases

Ojo con confundir interacción y dependencia:

```
class A {
   int getUserColor();
}

class B {
   void paintScreen(int color);
}

class C {
   A a;
   B b;

   void askThenPaint() {
      b.paintScreen(a.getUserColor());
   }
}
```

Clase C usa clases A y B:

- estas clases declaran variables globales
- y asumen ciertas cosas, como que "azul" $= 1$

- C depende de A y B
- C interactúa con A y B
- A y B también interactúan

Proceso de cambio de software

Ya hicimos Concept Location y Impact Analysis . . .

Actualization: ahora el programador implementa el cambio, y lo propaga a las clases identificadas en el Impact Analysis.

A veces basta con hacer un cambio pequeño . . .

Implementación original:

```
class Address {
        public move(...);
        protected String name;
        protected String streetAddress;
        protected String city;
        protected char state[2], zip[5];
}
```

A veces basta con hacer un cambio pequeño ...

Implementación original:

```
class Address {
        public move(...);
        protected String name;
        protected String streetAddress;
        protected String city;
        protected char state[2], zip[5];
}
```

Versión nueva:

```
class Address {
        public move(...);                    // 2) hay que propagar el cambio ...
        protected String name;
        protected String streetAddress;
        protected String city;
        protected char state[2], zip[9]; // 1) cambio el largo del zip code
}
```

Pequeños cambios

Esta estrategia de cambio es adecuada para:

- pequeños bug fixes
- cuando queremos agregar funcionalidad localizada en forma incremental

Pequeños cambios

Esta estrategia de cambio es adecuada para:

- pequeños bug fixes
- cuando queremos agregar funcionalidad localizada en forma incremental

Un cambio más grande puede requerir implementar un modulo nuevo . . .

¿y como integramos este modulo al sistema?

Polimorfismo

Si es un cambio que hemos anticipado en el diseño, podemos usar polimorfismo para implementarlo

- polimorfismo nos permite agregar comportamiento sin cambiar la interfaz de los objectos existentes
- en este caso, la propagación de los cambios será mínima

Polimorfismo

Ejemplo: estamos implementando SimFarm, un simulador de granjas

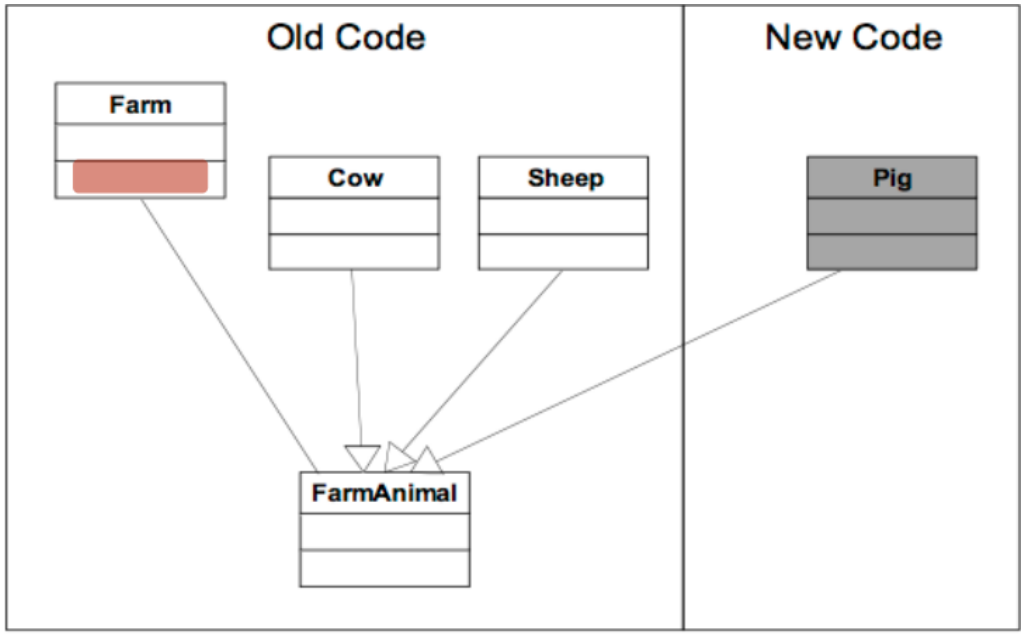
```
class FarmAnimal {
        public abstract void makeSound();
}

class Cow extends FarmAnimal {
        public void makeSound(){
                System.out.println("Moo-oo-oo");
        }
}

class Sheep extends FarmAnimal {
        public void makeSound(){
                System.out.println("Ba-a-a");
        }
}
```

Queremos agregar chanchos a nuestro simulador . . .

Polimorfismo

Podemos extender el sistema agregando una clase nueva . . . y debemos propagar los cambios a las clases que usan `FarmAnimals`

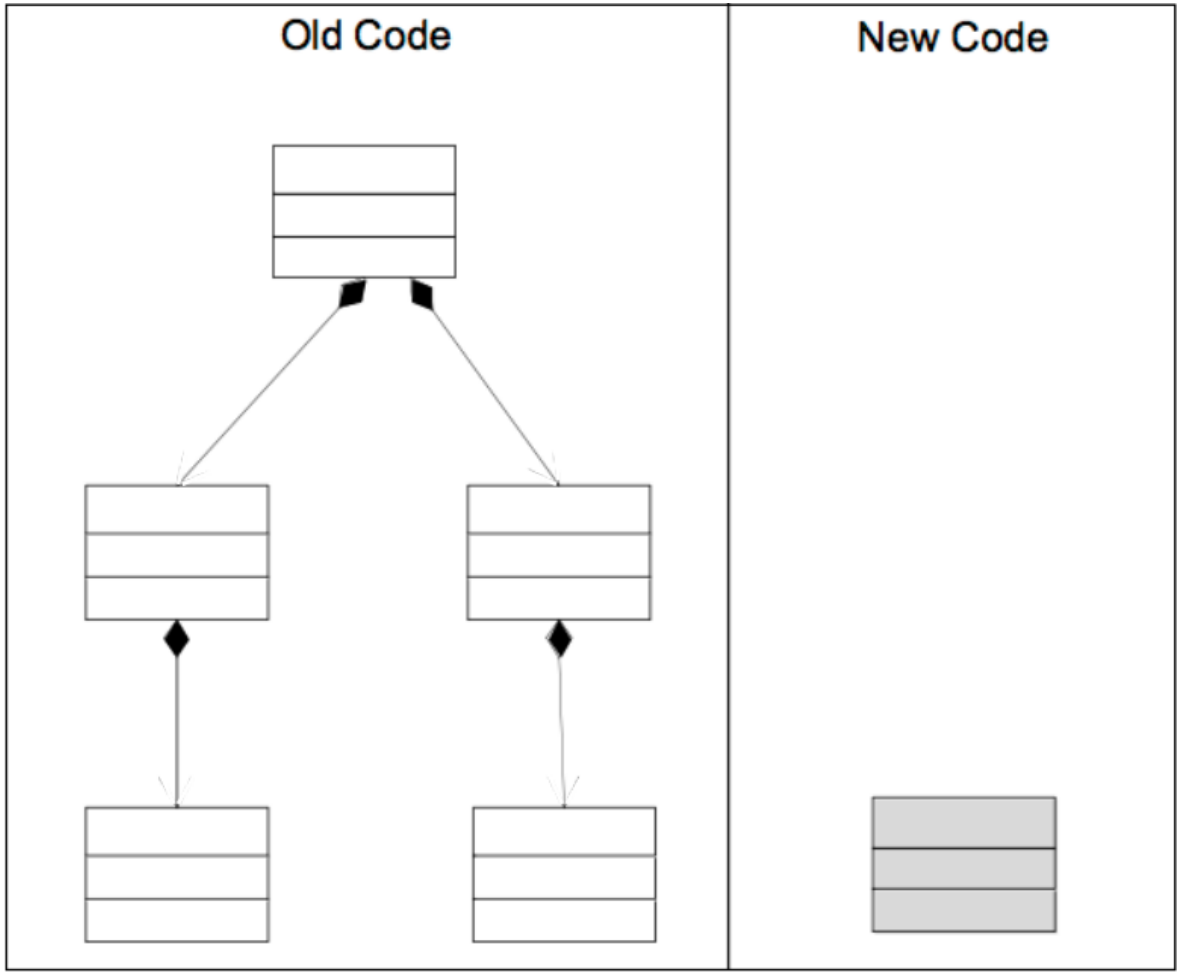
```
class Pig extends FarmAnimal {
    public void makeSound() {
        System.out.println("Oink");
    }
}
```



Para que esto resulte, debemos tener una idea de la familia de conceptos que pudiera extenderse . . . y como se podría extender (queremos evitar cambiar firmas de métodos)

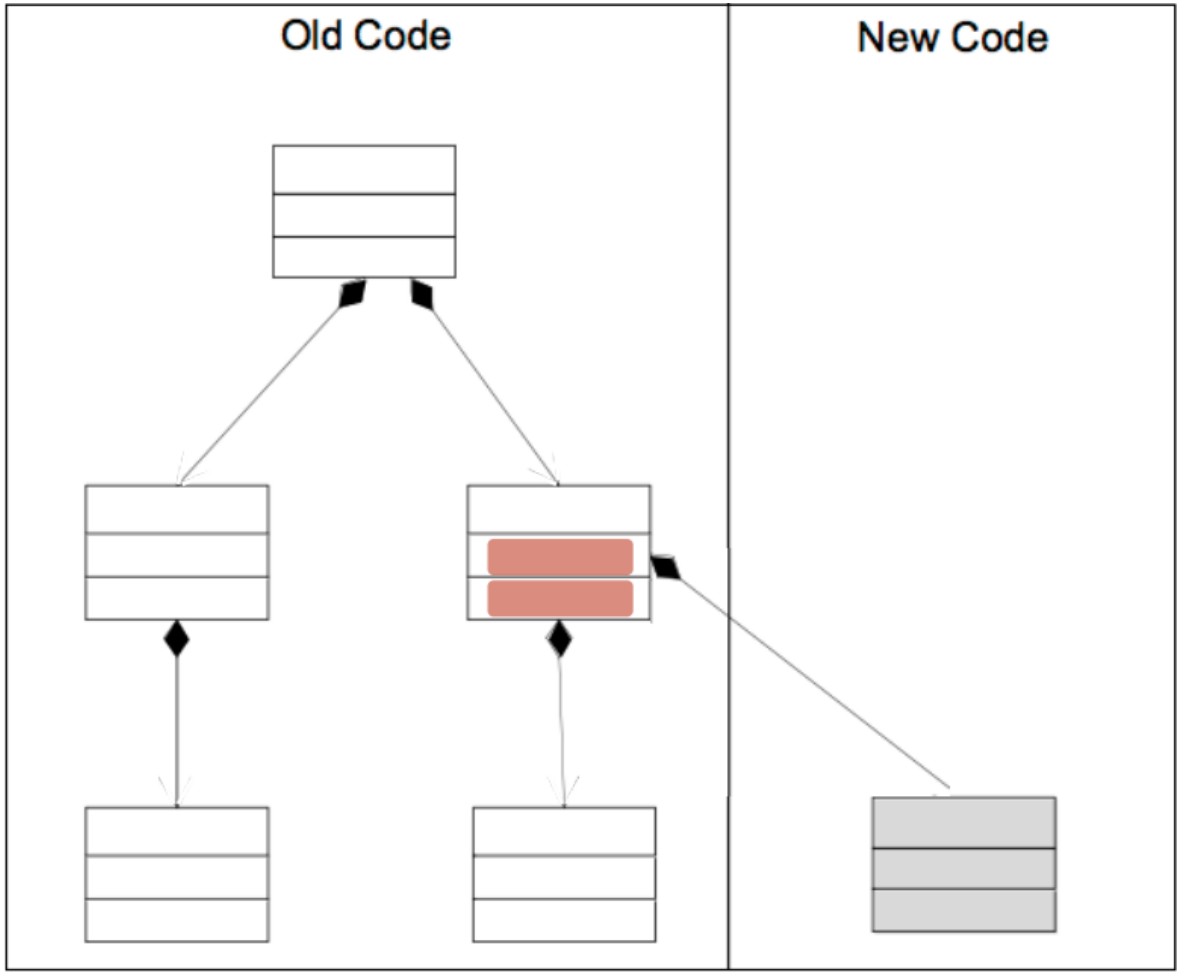
¿Y funcionalidad completamente nueva?

Tenemos que crear un modulo nuevo:



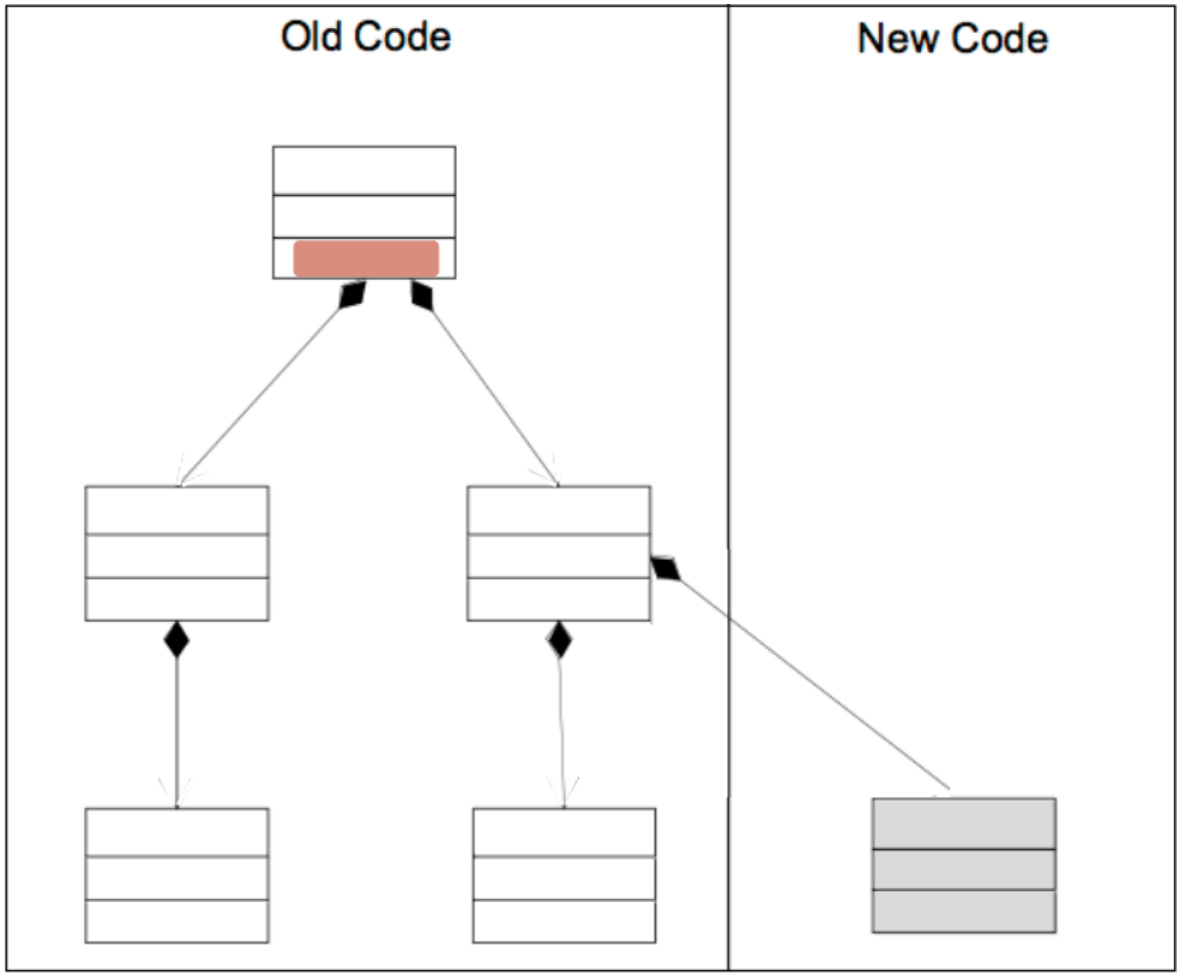
Conectando la nueva funcionalidad

Concept Location nos dirá donde debemos linkear la funcionalidad nueva:



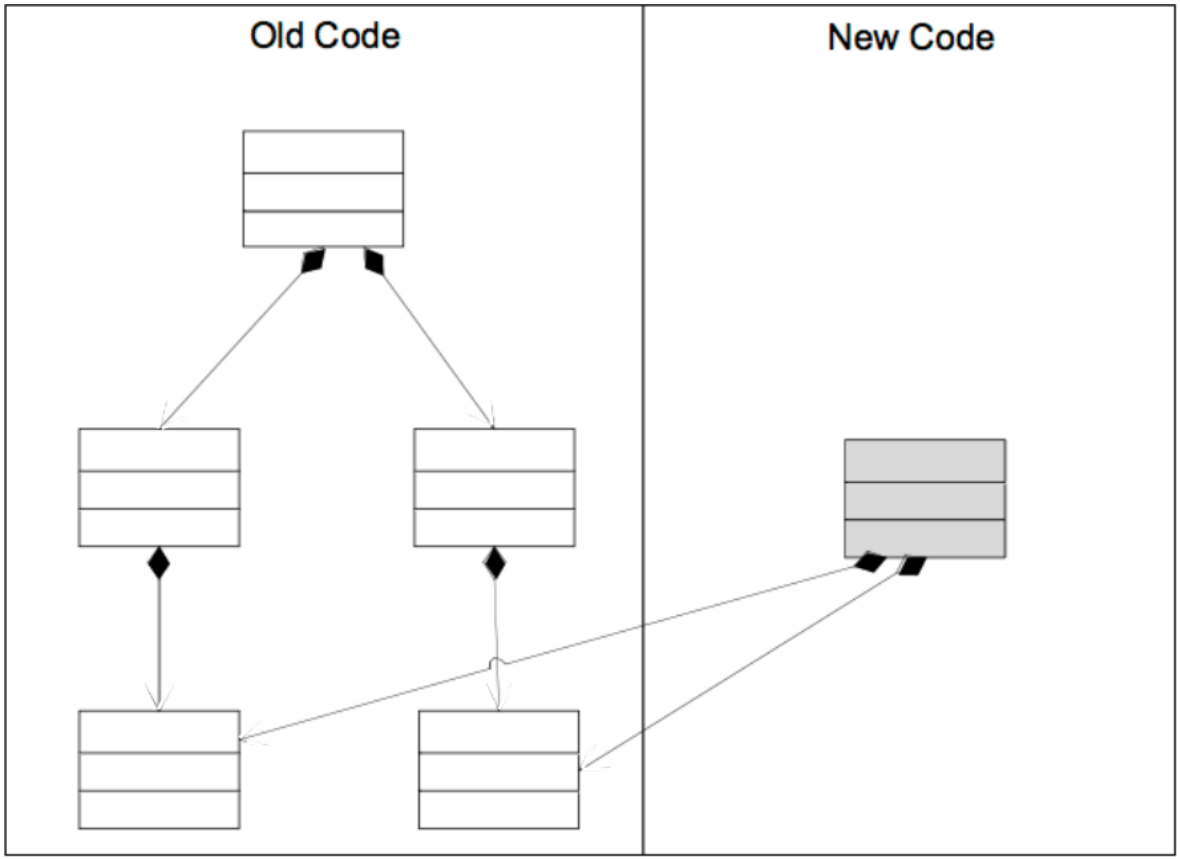
Nueva funcionalidad

Y debemos propagar los cambios (impacto puede ser grande):



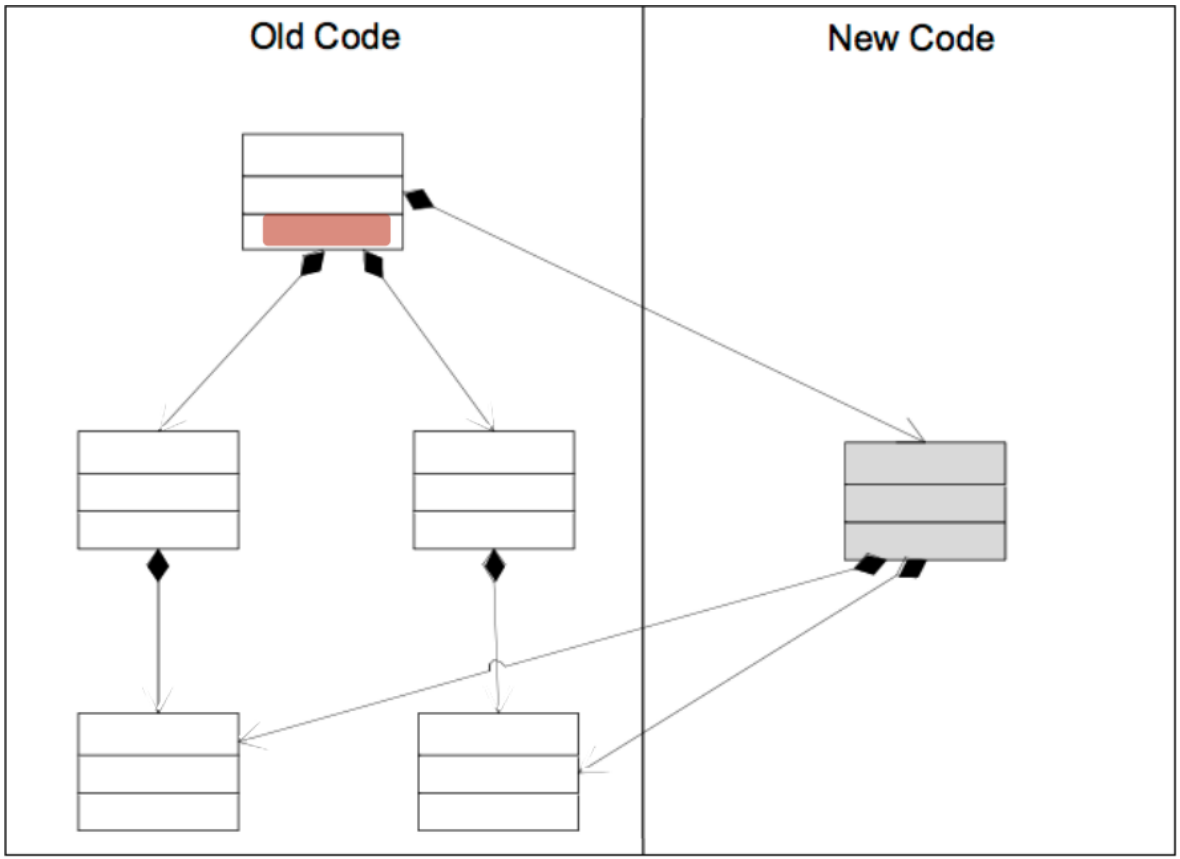
Nuevas interacciones

A veces podemos componer funcionalidad existente para crear nueva funcionalidad. Aquí desarrollamos un nuevo modulo (client) reusando funcionalidad.



Nuevas interacciones

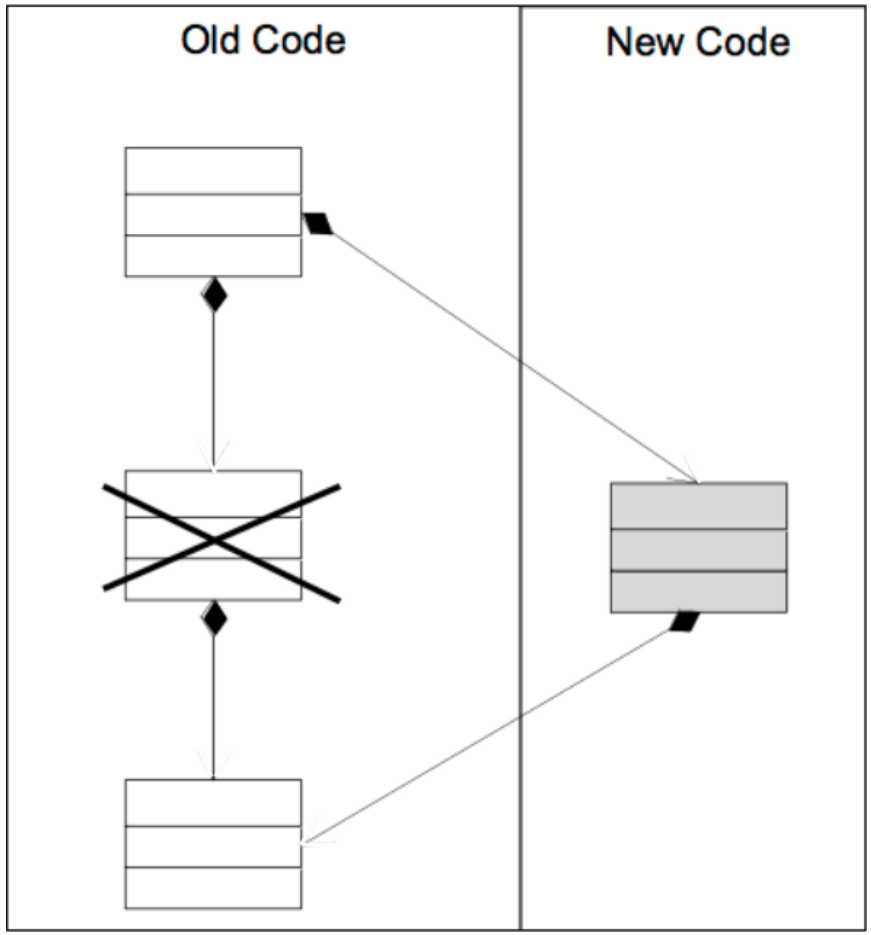
Concept Location nos dirá donde debemos linkear este modulo:



El impacto también puede ser grande, pero al menos reusamos algo de nuestro código.

Cambiar un modulo por otro

A veces debemos reemplazar un modulo existente con uno más nuevo:



Las referencias al modulo viejo ahora deben apuntar al modulo nuevo.

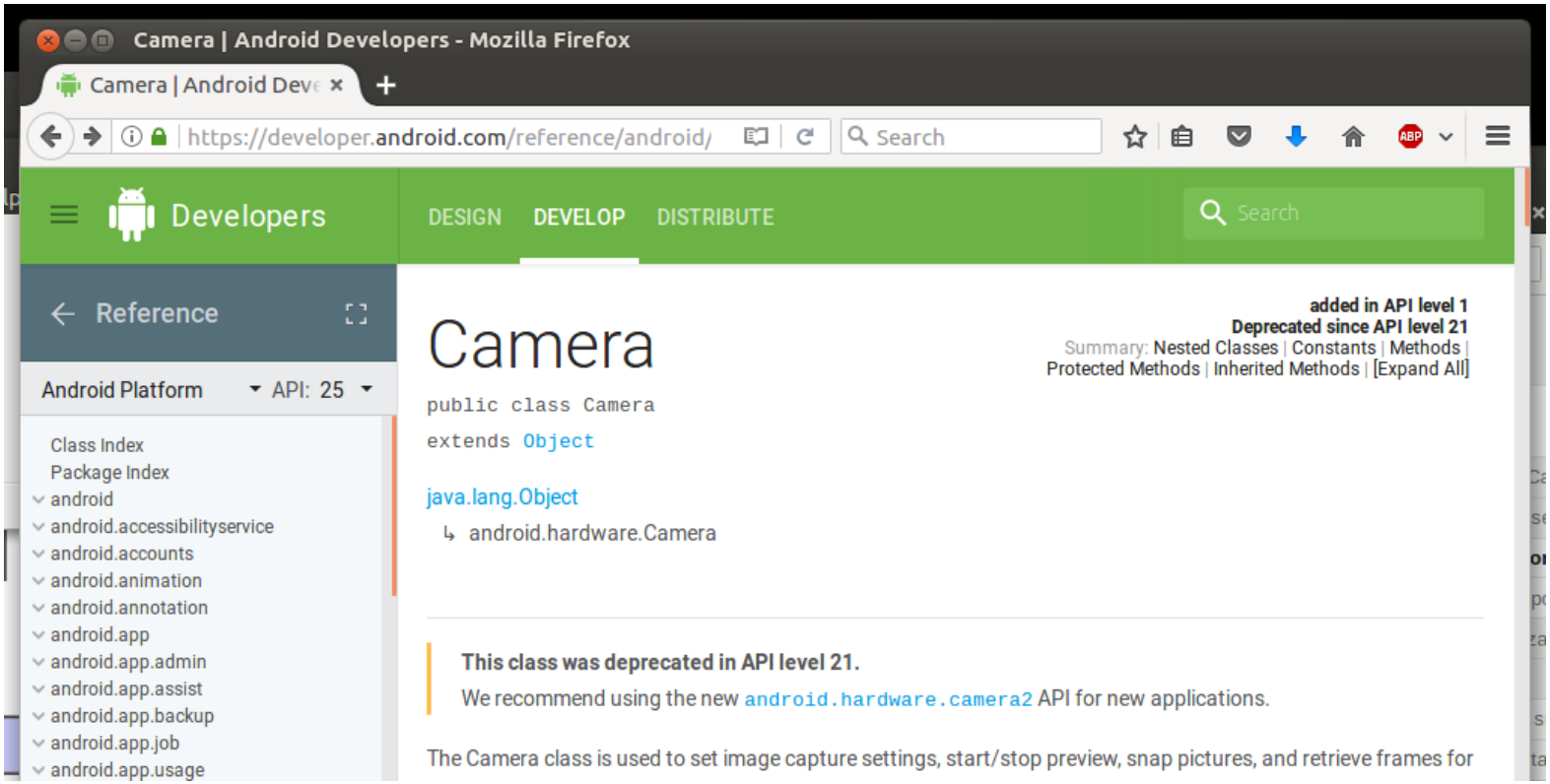
Sin importar la estrategia de cambio que usemos . . .

Una vez que hacemos el cambio, debemos propagarlo al resto del sistema:

- Por ejemplo, cualquier cambio a la firma de un método quebrara las interacciones que dependen de este
- Debemos ser sistemáticos, procesando los cambios uno a uno
- Ahora tenemos el "Changed Set", el conjunto de las clases afectadas por un cambio
  - hay un cierto traslapo con el "Estimated Impact Set"
  - pero en general, los programadores sub-estiman el "Changed Set"
  - esto se debe a la intangibilidad del software

Deprecation en APIs

Si la funcionalidad vieja y nueva puede coexistir, una alternativa viable es Deprecation:



Así, los clientes pueden migrar en forma gradual a la nueva funcionalidad. Esto se usa mucho en APIs, donde la estabilidad es importante.

Costo/beneficio de los cambios

Ejemplo: el sistema muestra la temperatura en Fahrenheit, ahora queremos mostrarla en Celsius

Posibles estrategias de cambio:

1. Hacer los cambios en Core: agregar una función de conversión
2. Hacer los cambios en UI: realizar la conversión antes de mostrar el dato

Costo/beneficio de los cambios

Ejemplo: el sistema muestra la temperatura en Fahrenheit, ahora queremos mostrarla en Celsius

Posibles estrategias de cambio:

1. Hacer los cambios en Core: agregar una función de conversión
2. Hacer los cambios en UI: realizar la conversión antes de mostrar el dato

La segunda estrategia es más facil de implementar, pero puede dificultar mantenciones futuras, porque ahora hay logica de negocios en la UI.

¿Cuándo deberian aplicar las distintas estrategias de cambio?

Estrategia	Grado de propagacion	Caracteristicas
cambio pequeño	pequeño a grande	cambios locales
polimorfismo	pequeño	anticipar en el diseño
modulo nuevo	medio a grande	agrega funcionalidad nueva
cliente nuevo	medio a grande	reusa/compone funcionalidad existente
reemplazar modulo	medio a grande	funcionalidad existente es obsoleta
deprecation	ninguno	funcionalidad existente y nueva puede coexistir

¿La mejor estrategia?

¡Anticipar los cambios!

No siempre es posible, pero si tienen modulos y clases cohesivas y con bajo acoplamiento, los cambios tendran un menor impacto.

Regla de 3: no queremos anticiparnos demasiado y quedar con una solucion sobre-ingenieriada