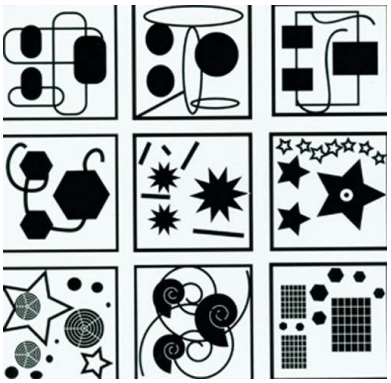
Objetos 2
Nuevos patrones



Alejandra Garrido:
garrido@lifia.info.unlp.edu.ar

Repaso – Clasificación de patrones

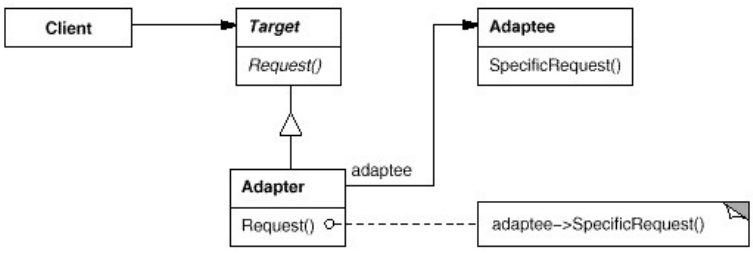
- Creacionales: tienen que ver con el proceso de instanciación
- Estructurales: tienen que ver con la composición entre clases y objetos para formar estructuras más grandes, de las relaciones entre objetos.
- De comportamiento: tienen que ver con los algoritmos y la asignación de responsabilidades entre objetos, y los patrones de comunicación entre objetos.

Patrones ya vistos

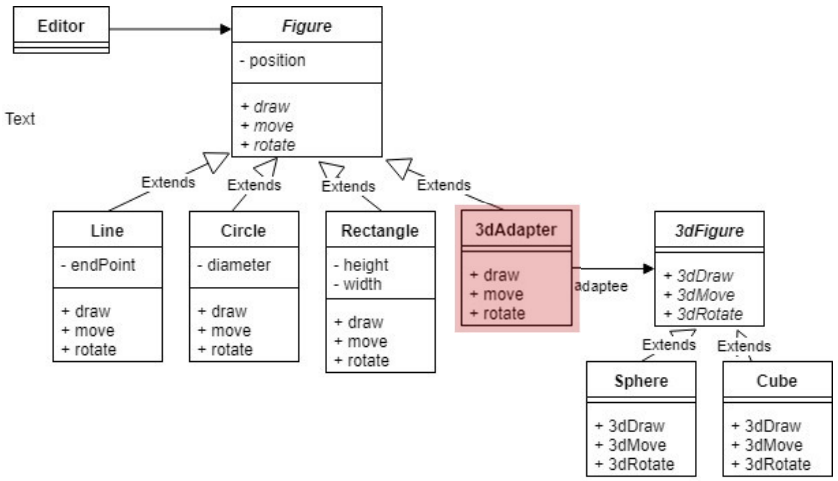
- Adapter (estructural)
- Composite (estructural)
- Template Method (comportamiento)

Hoy veremos 2 nuevos patrones estructurales

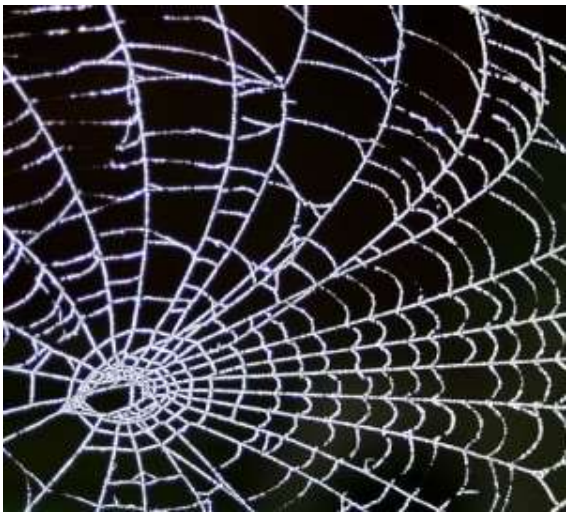
Diagrama de clases de Adapter



Ejemplo de Adapter de figuras



1er nuevo patrón



Ejercicio 1: Edición de fotos en Instagram

- A cada foto podemos aplicarle distintos efectos:
  - Filtros
  - Ajuste de perspectiva
  - Tilt shift
  - Brillo, contraste, temperatura, saturación, etc. Etc.

- Cada una de estas características puede agregarse o quitarse.

Ejercicio 1: Soluciones posibles?

1. Crear subclases de Foto para los distintos efectos

2. Crear una jerarquía separada de efectos, hacer que Foto conozca a todos sus efectos, y agregar métodos en Foto para aplicar los distintos efectos

Ejercicio 1 – Fuerzas del problema

- Queremos agregar responsabilidades a algunos objetos individualmente y no a toda una clase
- Estas responsabilidades pueden agregarse o quitarse dinámicamente
- Si usamos herencia (solución 1) para agregar responsabilidades solo en una subclase de objetos la solución es inflexible, porque se decide estáticamente y no podríamos quitarlas
- Si usamos composición (solución 2) queda un protocolo y una responsabilidad muy grande para la clase original

Ejercicio 2: Streams

- Cuando se necesita procesar una entrada o escribir a una salida, los streams resultan la mejor manera
- En Smalltalk, en Java, en .Net existe el concepto de streams y podemos encontrar una jerarquía de Stream importante, con un protocolo que permite:
  - abrir un stream (colección de acceso secuencial) para lectura o escritura
    Ej: (File named: 'myfile.pdf') readStream
  - leer / escribir el elemento de la posición actual
    #next / #nextPut: / #nextPutAll: / #peek
  - posicionar el stream
    #position / #position: / #upTo:

Ejercicio 2: Streams

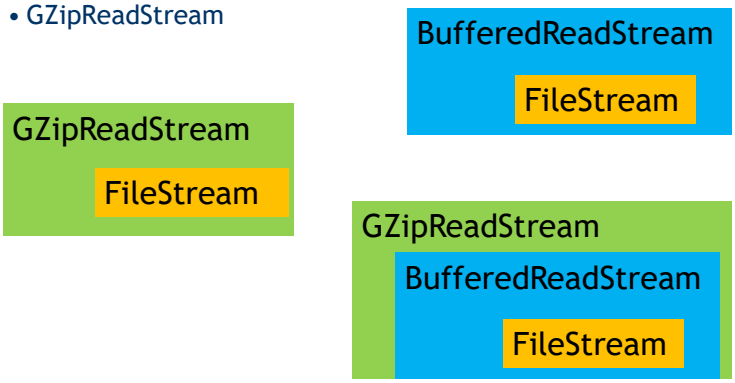
- Los streams se usan también para leer o escribir a archivos del file system.
- Los archivos pueden ser binarios o de caracteres (con distinto encoding)
  - BinaryFileStream
  - ZnCharacterReadStream - ZnCharacterWriteStream
- Deberían poder accederse de una manera eficiente usando buffering
  - ZnBufferedReadStream – ZnBufferedWriteStream
  - ... binarios o de caracteres
- Deberíamos poder comprimirlo/descomprimirlo
  - GZipReadStream – GZipWriteStream
  - ... binarios o de caracteres, buffered o no

Ejercicio 2 – Fuerzas del problema = Ejercicio 1

- Queremos agregar responsabilidades a algunos objetos individualmente
- Estas responsabilidades pueden agregarse o quitarse dinámicamente
- Si usamos herencia (solución 1) para agregar responsabilidades solo en una subclase de objetos la solución es inflexible, porque se decide estáticamente y no podríamos quitarlas
- Si usamos composición (solución 2) queda un protocolo y una responsabilidad muy grande para la clase original

Ejercicio 2: Streams

- Otra manera.
- Supongamos que tenemos 3 posibilidades:
  - FileStream
  - BufferedReadStream
  - GZipReadStream



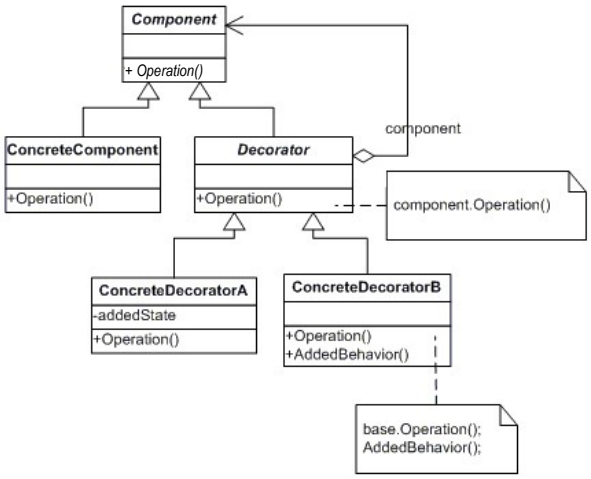
Patrón Decorator

- **Objetivo:** Agregar comportamiento a un objeto dinámicamente y en forma transparente.

- **Problema:** Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían "mutar de clase". El problema que tiene la herencia es que se decide estáticamente.

Patrón Decorator

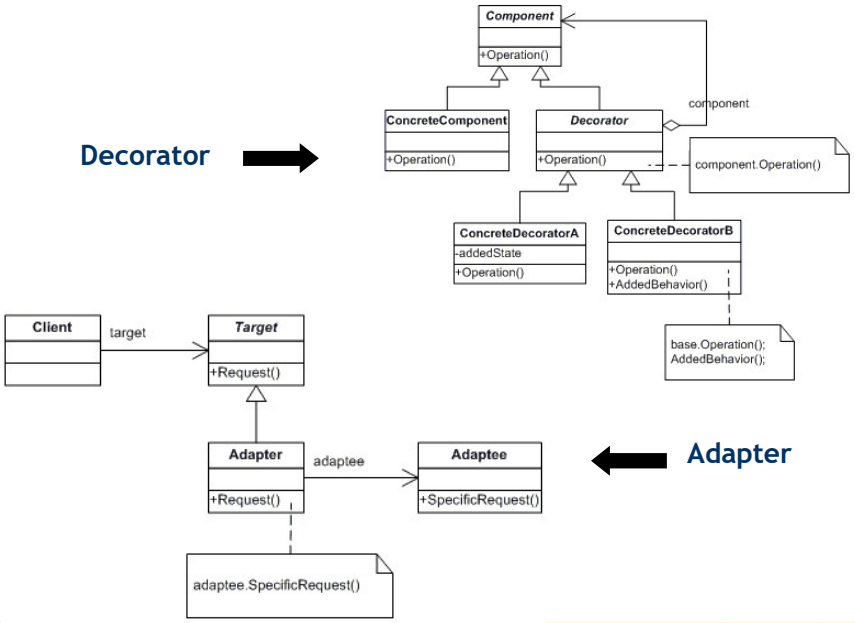
- **Solución:** Definir un decorador (o "wrapper") que agregue el comportamiento cuando sea necesario



Patrón Decorator

- **Consecuencias:**
  - \+ Permite mayor flexibilidad que la herencia.
  - \+ Permite agregar funcionalidad incrementalmente.
  - – Mayor cantidad de objetos, complejo para depurar
- **Implementación:**
  - Misma interface entre componente y decorador
  - No hay necesidad de la clase Decorator abstracta
  - Cambiar el "skin" vs cambiar sus "guts"

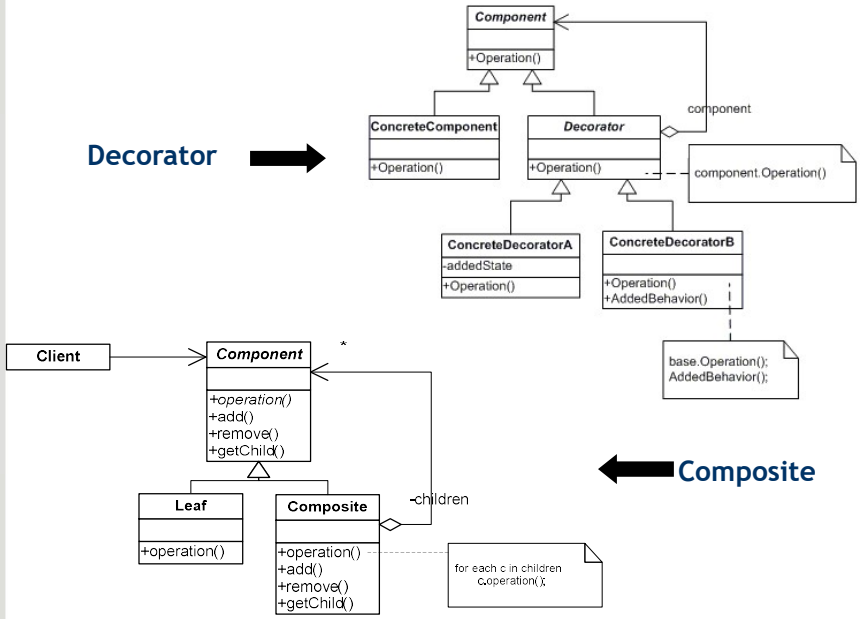
Decorator vs. Adapter



Decorator vs. Adapter (wrappers)

- Ambos patrones “decoran” el objeto para cambiarlo
- Decorator *preserva* la interface del objeto para el cliente.
- Adapter *convierte* la interface del objeto para el cliente.
- Decorators pueden y suelen anidarse.
- Adapters no se anidan.

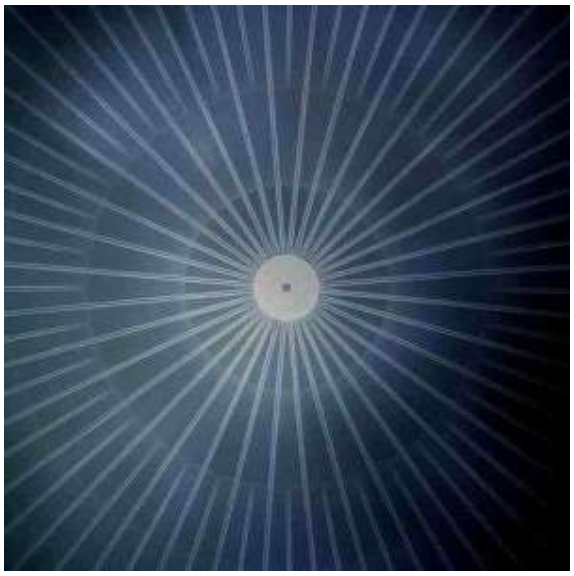
Decorator vs. Composite

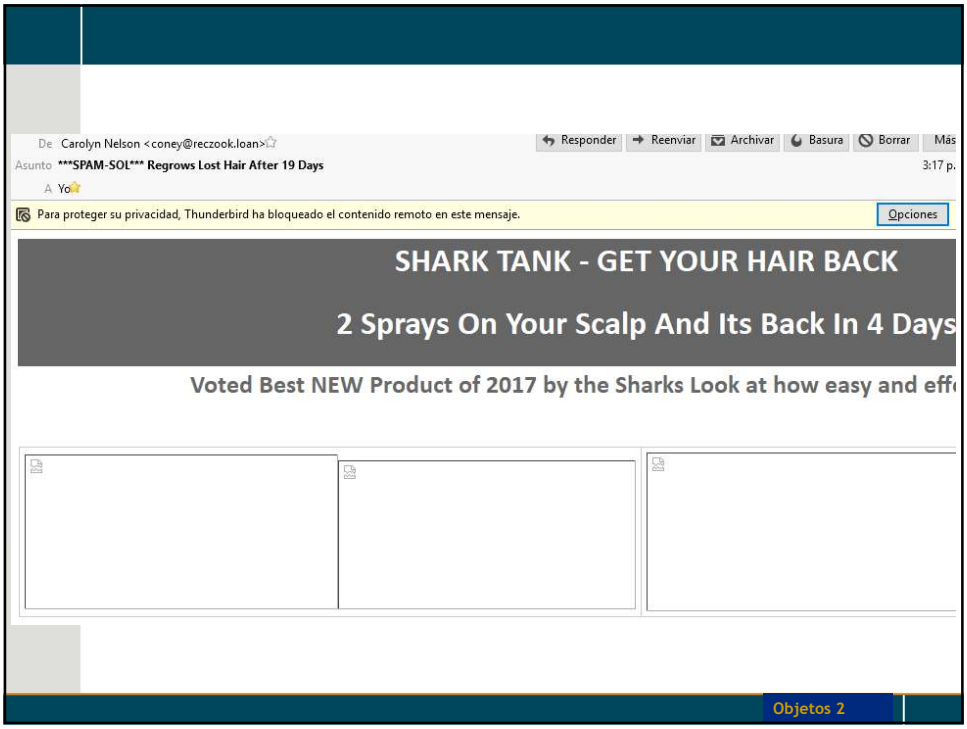


Decorator vs. Strategy

- En qué se parecen?
- En qué se diferencian?

2do nuevo patrón





Carga bajo demanda

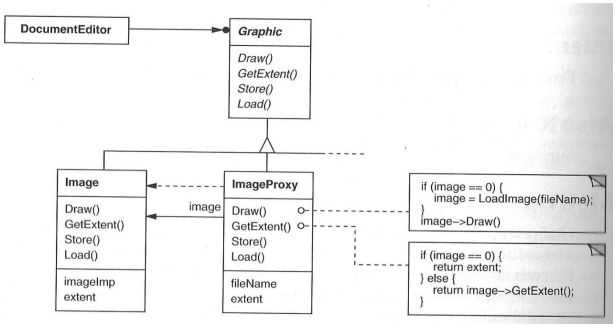
- En muchos casos un email puede tener muchas imágenes, siendo estas pesadas y lentas de cargar
- No queremos que la apertura de un email sea lenta.
- En algunos casos las imágenes ni siquiera serán vistas.

Carga bajo demanda

- Queremos evitar el costo de leer la imagen hasta tanto sea necesario mostrarla
- Igualmente necesitamos un « representante » de la imagen, de manera de darle al cliente un objeto que se vea y actúe como el cliente espera
- Luego, la idea es crear una imagen “falsa”, un impostor que
  - Debe responder a los mensajes de la imagen verdadera.
  - Cuando sea necesario mostrarla en pantalla, debe ir a buscar la imagen original a disco, leerla y mostrarla.

Solución

- Cargar las imágenes bajo demanda, utilizando un objeto *proxy*. El proxy se comporta como una imagen normal y es el responsable de cargar la imagen bajo demanda

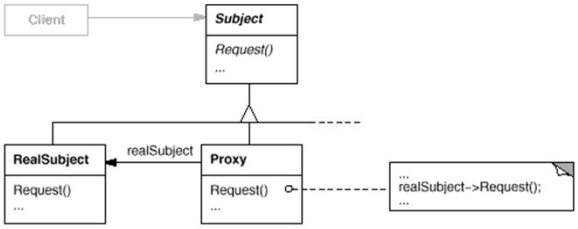


Patrón Proxy

- **Propósito:** proporcionar un intermediario de un objeto para controlar su acceso.
- **Aplicabilidad**: cuando se necesita una referencia a un objeto más flexible o sofisticada

Patrón *Proxy*. Solución

- Colocar un objeto intermedio que respete el protocolo del objeto que está reemplazando.
- Algunos mensajes se delegarán en el objeto original. En otros casos puede que el proxy colabore con el objeto original o que reemplace su comportamiento.



Patrón *Proxy*

- **Ejemplos de uso:**
  - Demorar la construcción de un objeto hasta que sea realmente necesario (virtual proxy).
  - Restringir el acceso a un objeto por seguridad (protection proxy).
  - Implementación de objetos distribuídos (remote proxy).
- **Implementación:**
  - Redefinir todos los mensajes del objeto real ??
  - Proxy no siempre necesita conocer la clase del objeto real

Proxy de protección

```
BankAccount
    v.i. balance

>>balance
    ^balance

>>deposit: anAmount
    balance := balance + anAmount

>>withdraw: anAmount
    balance := balance - anAmount
```

```
BankAccountProxy
    v.i. realAccount

>>balance
    self checkAccess ifTrue: [
        ^realAccount balance]

>>deposit: anAmount
    self checkAccess ifTrue: [
      ^realAccount deposit: anAmount]

>>withdraw: anAmount
    self checkAccess ifTrue: [
      ^realAccount withdraw:
                        anAmount]

>>checkAccess ...
```

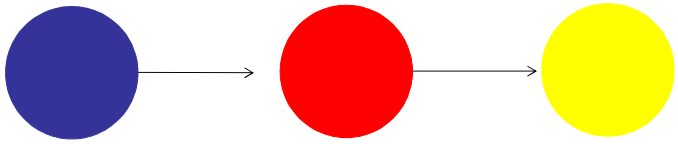

Proxy de acceso remoto

- Para acceder a objetos que se encuentran en otro espacio de memoria, en una arquitectura distribuida
- El proxy empaqueta el request, lo envía a través de la red al objeto real, espera la respuesta, desempaqueta la respuesta y retorna el resultado
- En este contexto el proxy suele utilizarse con otro objeto que se encarga de encontrar la ubicación del objeto real. Este objeto se denomina Broker, del patrón de su mismo nombre

Adapter, Decorator, Proxy

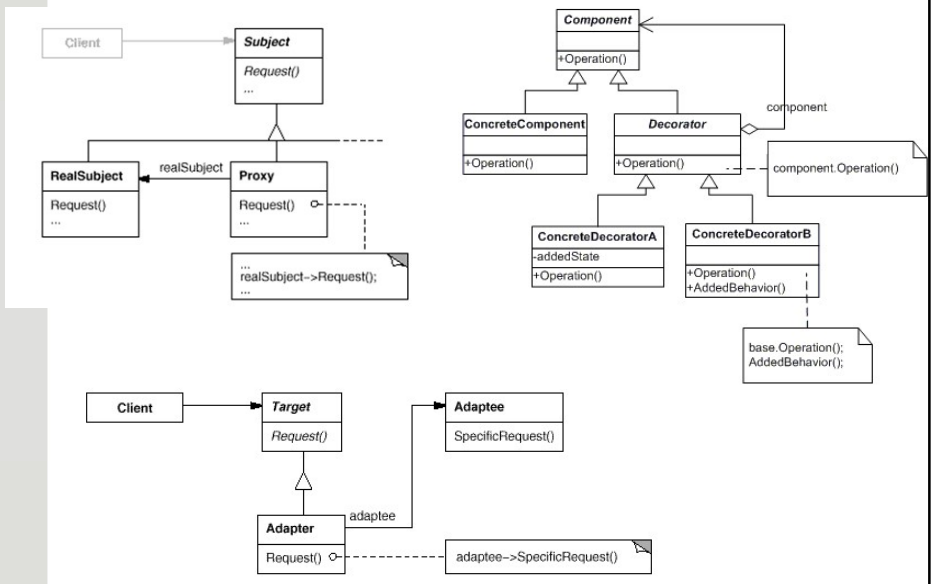
- Todos patrones estructurales
- Todos con diagramas de objetos similares
- Distinto propósito

- A todos se los llama también “wrappers”



Cliente Wrapper Objeto

Proxy vs. Decorator vs. Adapter



- Pattern wrap up

¿Qué cosa es importante del diagrama de clases?

- Clases que componen el patrón
- Jerarquías
- Clases abstractas
- Métodos abstractos
- Relaciones de conocimiento / composición

Clases abstractas y métodos abstractos

- Las clases abstractas son clases a partir de las cuales no pueden crearse instancias.
- ¿Para qué sirven?
  - Para factorizar un comportamiento común
  - Para establecer un protocolo común: métodos abstractos
- **Métodos abstractos**: establecen el protocolo de una jerarquía de clases: aseguran que todo objeto instancia de una subclase puede responder a ese mensaje.
- *En el diagrama de clases se identifican por escribirse en letra itálica.*
- *¿Cómo se identifican en Smalltalk?*
- **Importancia de las clases abstractas.** Por ej. abstract strategy

Implementación de Proxy usando reflexión

- Method look-up
- #doesNotUnderstand: aMessage

- Cuando a un objeto se le envía un mensaje que no implementa, la máquina virtual le envía el mensaje #doesNotUnderstand: al objeto con una "reificación" del mensaje como argumento.
- El mensaje (instancia de Message) contiene al selector y un Array de los argumentos.
- El programador puede examinar el contexto en el que ocurrió el error, cambiarlo y continuar la ejecución
- ➔ En vez de reimplementar en Proxy todos los mensajes, solo se define #doesNotUnderstand:
- ➔ Cada envío de mensaje a instancias de Proxy termina en #doesNotUnderstand:, donde el objeto puede manipular el mensaje para por ejemplo, enviárselo al objeto real

#doesNotUnderstand:

```
ImageProxy>>doesNotUnderstand: aMessage
  | image |
  image := CachedImage on:
          (ImageReader fromFile: fileName) image.

  image perform: aMessage selector
          withArguments: aMessage arguments
```

```
BankAccountProxy>>doesNotUnderstand: aMessage
...
```


Entendiendo qué es reflexión

- Un programa reflexivo es aquel que puede razonar sobre si mismo, es decir que puede ***observarse*** y ***cambiarse*** dinámicamente.
- Es aquel que puede observar su propia ejecución (***introspección***) e incluso cambiar la manera en que se ejecuta ***(intercesión)***.
- Requiere poder expresar y manipular el estado de la ejecución como datos: ***reification***
  - *Reify: to regard (something abstract) as a material or concrete thing*

Videos

- Visita de Ralph Johnson a la Facultad de Informática en 2016, hablando sobre el GoF, luego de 22 años:
- https://www.youtube.com/watch?time_continue=43&v=FOV85WxN31U