Más alla de la orientación al objeto

Reflexión, metaprogramación y programacíon por aspectos

Eric Tanter

etanter@dcc.uchile.cl

DCC–University of Chile (Chile), OBASCO–Ecole des Mines de Nantes (France)

Contenido

- Introducción
  - programación, paradigmas, lenguajes
  - principios fundamentales
- Orientación al objeto
  - fundamentos
  - promesas
  - fallencias
- Necesidades y alternativas

Contenido (2)

- Reflexión y metaprogramación
  - conceptos
  - herramientas y ejemplos en Java
- AOP
  - conceptos
  - varias propuestas
- Conclusiones

Introducción: Programación, paradigmas y lenguajes

Introducción

- Paradigmas de programación
  - funcional
  - lógico
  - procedural
  - objeto/prototipo/agente
- Formas/métodos para resolver un problema
- Dividir para reinar
- Decomposición – recomposición

Dos principios fundamentales en programación

separation of concerns e *information hiding*

Separation of Concerns (SOC)

- *concern* = preocupación, asunto
  - *“separación de preocupaciones”*
- no mezclar todo
- programa estructurado
- cada parte con su responsabilidad
- *reusabilidad, mantención*

Information Hiding

- esconder la información no necesaria
- separación interfaz / implementación
- noción de *modulos*
- baja dependencia entre modulos (*low coupling*)

Nada nuevo!

- 1968, Dijkstra
  - *"la forma de estructurar un programa es tan importante como hacerlo producir la respuesta correcta."*
  - el primero a enseñar el principio de SOC
  - mostró que pedazos de programas pueden ser desarrolladas independientemente
- 1972, Parnas
  - introducción del concepto de *information hiding*
  - decomposición de un sistema en partes que esconden detalles de implementación detras de interfases

Orientación al Objeto: promesas y fallencias

Orientación al Objeto (OO)

- en linea directa con las ideas de Dijkstra y Parnas
- empezó fines de 70s/80s con Simula, Smalltalk, CLOS
- lenguajes ofrecen mecanismos fundamentales
  - objeto = datos + métodos
  - encapsulación
  - herencia (subtipos)
  - agregación, delegación
  - polimorfismo
  - instanciación (clases) / clonación (prototipos)

Los mécanismos y sus promesas

- encapsulación
  - separar implementación / interfaz
  - misma interfaz: intercambiable
  - *evolución, mantención, reusabilidad*
- herencia
  - una decomposición "dominante" (relación *es-un*)
  - una subclase "hereda" de la interfaz y del código de su superclase
  - *reusabilidad, extensibilidad (frameworks)*

Porque no resulta en la realidad, entonces???

Ejemplos de **`org.apache.tomcat`**

(AspectJ.org)

Modularización OK

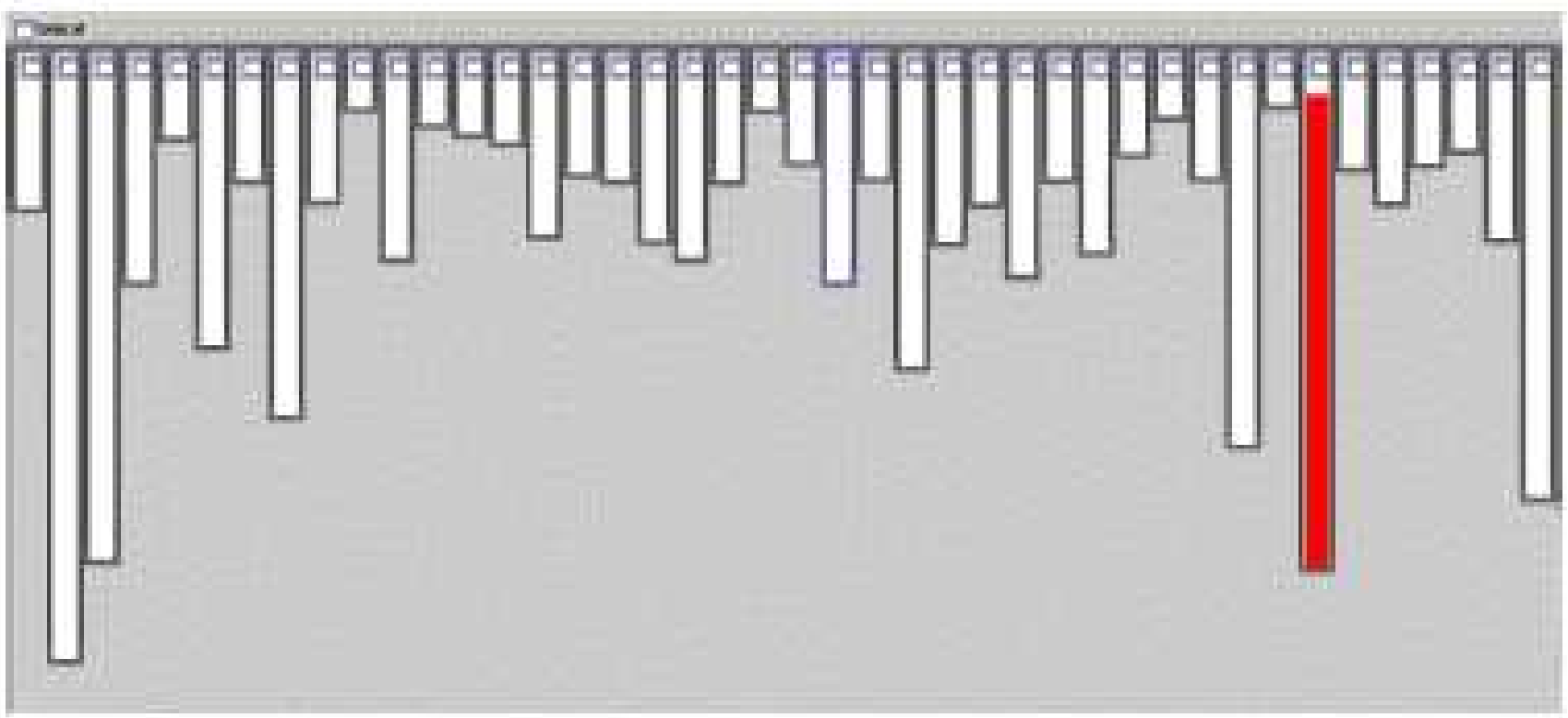


Figure 1: XML parsing

Modularización OK

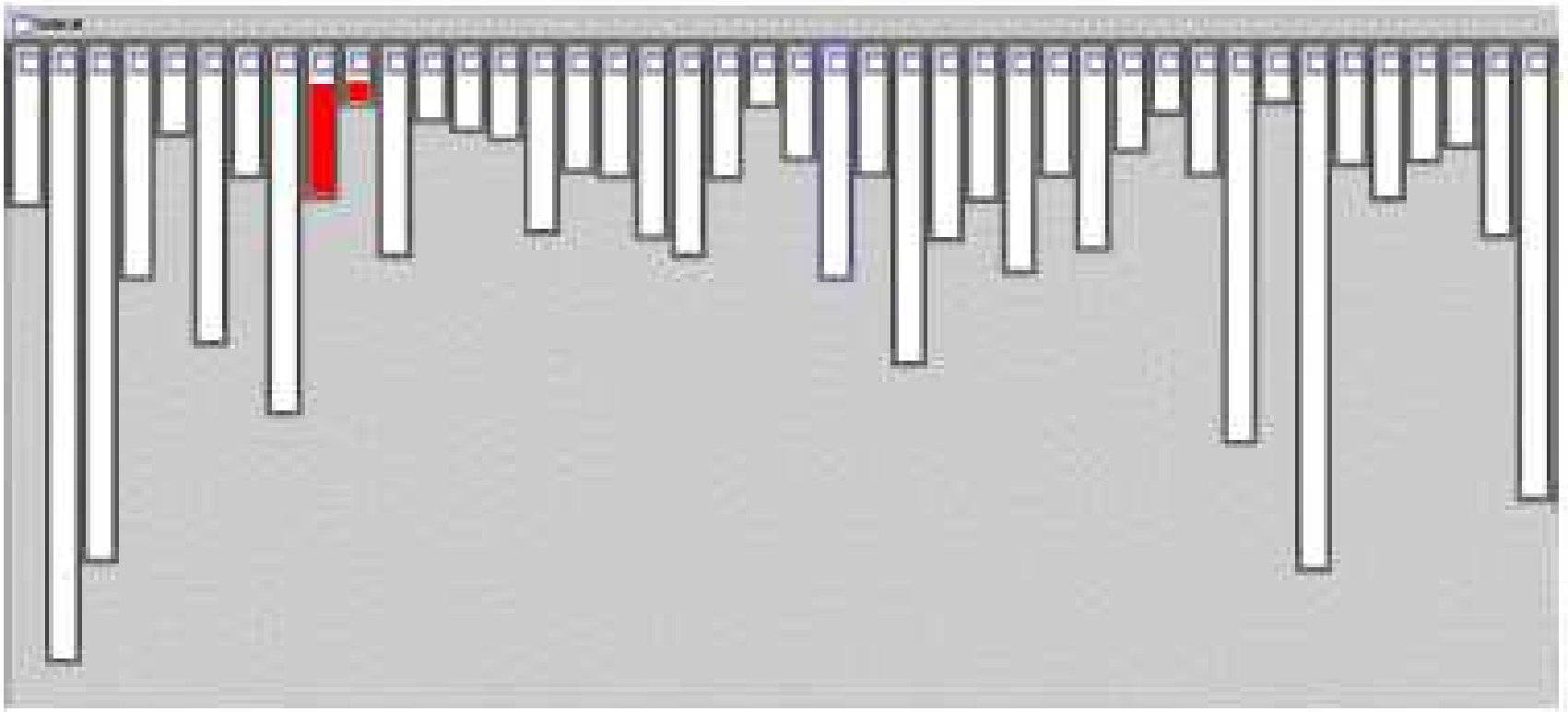


Figure 2: URL pattern matching

Modularización mala

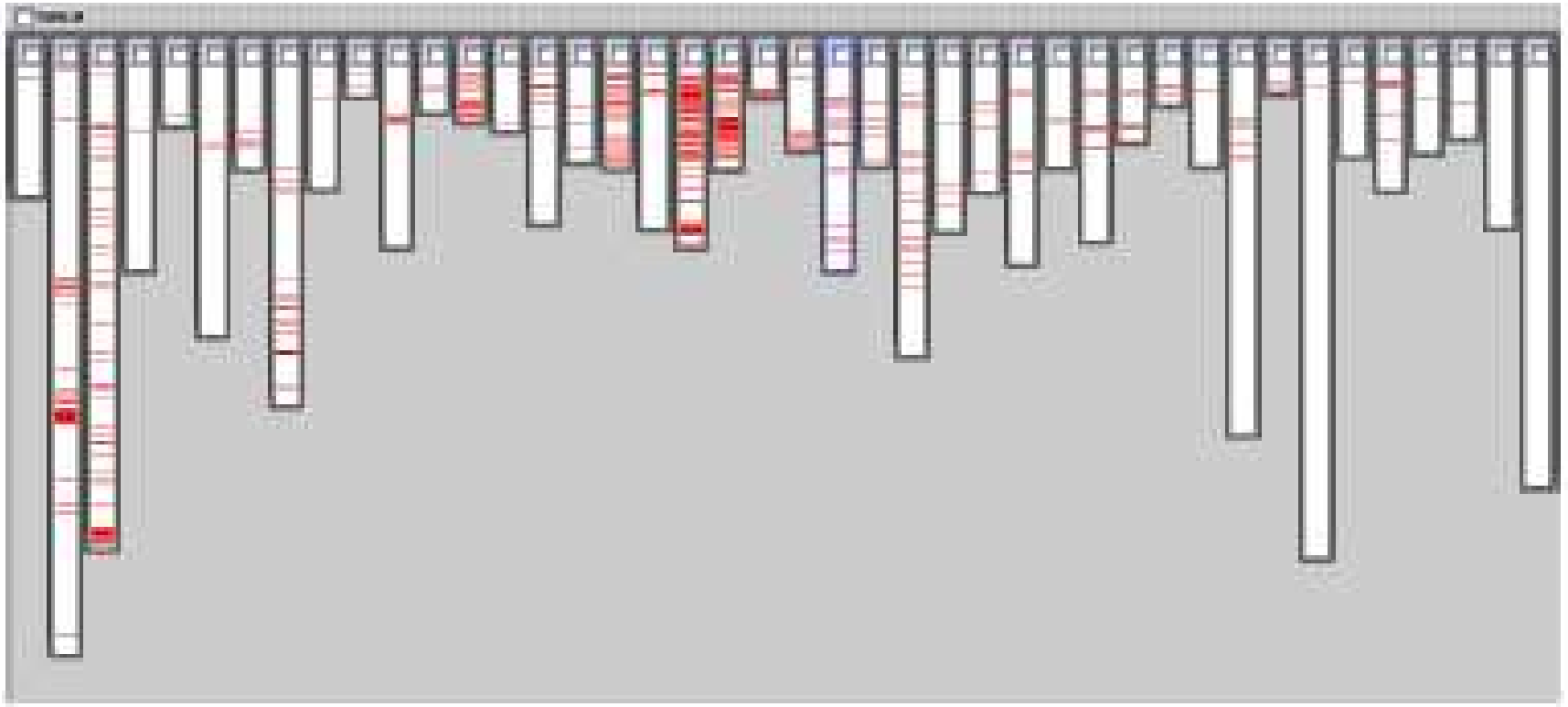


Figure 3: Logging

ECOLE DES MINES DE NANTES

Modularización mala

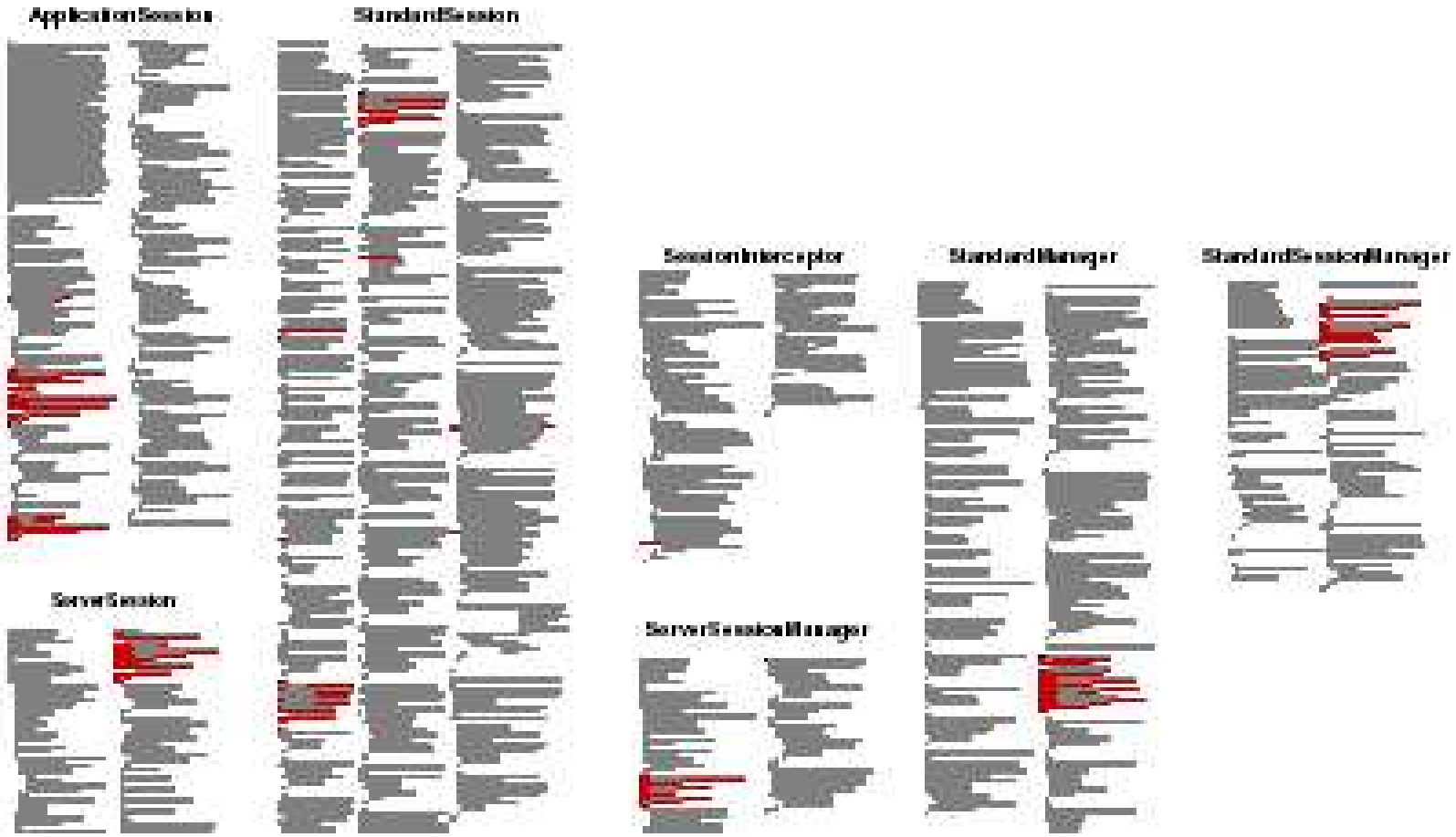


Figure 4: Session expiration

ECOLE DES MINES DE NANTES

Analisis de los problemas de OO

"Los Tres"

- problema de la decomposición dominante
- preocupaciones no-funcionales
- preocupaciones transversales (*crosscutting concerns*)

El problema de la decomposición dominante

- como modelizar objetos hibridos?
  - decomposición de animales en jerarquía
  - como lo hacemos con los mamiferos submarinos, animales anfibios, etc.?
- herencia multiple
  - e.g., C++
  - complejo (conflictos de métodos/variables heredados)
- herencia simple de clase + multiple de interfaces
  - e.g., Java
  - semantica clara pero reuso limitado
- podemos llegar a decomposiciones poco naturales
- perspectivas de evolución limitadas

Preocupaciones no-funcionales (1)

- preocupación *no-funcional* = que *no* tiene que ver con la funcionalidad principal de programa (*business logic*)
- en el mundo real, hay muchas preocupaciones no-funcionales
  - tratamiento de excepciones (*error handling*)
  - seguridad (criptografía, autentificación, autorización)
  - persistencia (base de datos, archivos, etc.)
  - distribución
  - concurrencia y sincronización
  - extensión temporal de funcionalidad
  - …
- al final el código de cada clase es una *mezcla* de código funcional y código no-funcional

Ilustración

```
public void withdraw(double amount)
    throws NotAllowedException {
        if(!authorized(Thread.currentThread()))
            throw new NotAllowedException();
        try {
            checkAmount(amount);
            doWithdraw(amount);
            notifyListeners();
        }
        catch(LimitAmountExceeded e) {
            doWithdraw(e.getMaxAmount());
        }
```

Preocupaciones no-funcionales (2)

- programación más dificil
  - legibilidad del código
  - detección y resolución de errores
  - mantención
- reusabilidad destruida
  - mismo código funcional con otras estrategias no-funcionales?
  - misma estrategia no-funcional con otro código funcional?
  - lo único que queda es copiar/pegar/ajustar. . .

ECOLE DES MINES DE NANTES

Preocupaciones transversales (1)

- tipo de preocupaciones no-funcionales aún más complejo
- son transversales (*crosscut*) a la estructura impuesta por la decomposición principal
- no caben en ningun modulo, son *trascendentes*
- propios a interacciones/relaciones entre modulos
- e.g., estrategia de seguridad sobre varios objetos
  - *se puede llamar A.m() solo si el llamado a B.n() fue exitoso*

Preocupaciones transversales (2)

- código redundante
  - código similar en varios lugares
- dificil de razonar
  - no tiene estructura explicita
  - dificil tener una visión global
- dificil de mantener
  - buscar todo el código involucrado
  - cambiarlo de forma consistente
  - tener cuidado de no romperlo por accidente

Necesidades y alternativas

Necesidades

- ofrecer soporte para:
  - modularizar los preocupaciones no-funcionales de forma adecuada
  - "conectar" tales modulos con el programa funcional
- asegurar que el programa funcional sea *independiente* de los varios modulos no-funcionales (y vice-versa?)
- dar gran poder de expresividad para modulos no-funcionales, ya que pueden necesitar *razonar sobre* el programa funcional y su ejecución

Alternativas

- generación/transformación de programas
- **reflexión y metaprogramación** (90%)
  - separación de preocupaciones
  - otros problemas (programación dinámica, sistemas adaptables, . . .)
- separación multi-dimensional de preocupaciones (Hyper/J)
- **programación por aspectos** (10%)
  - solo separación de preocupaciones

Reflexión y metaprogramación

Contenido

- Origenes
- Sistemas reflexivos: construcción y caracterización
- Aplicaciones de la reflexión
  - programación dinámica
  - adaptación de programas
  - separación de preocupaciones y adaptabilidad
- Cumplimientos y limitaciones

Perspectiva histórica

Perspectiva histórica: datos y programas

- la distinción datos/programas aperece en los 1830s: Charles Babbage y su *Difference Engine No.2*
  - el *granero* contiene los datos
  - el *molino* trata estos datos
- en 1958, von Neumann describe la arquitectura del computador numerico
  - introduce la idea de almacenar el programa en la memoria, esta misma que sirve para guardar los datos. . .
  - es atraido por las perspectivas ofrecidas por el hecho de manipular un programa como datos,
  - un programa mismo podra modificar las propias ordenes que lo hacen ejecutar!

Perspectiva histórica: reflexión

- Filosofía (Brian Cantwell Smith)
  - estudios de la sicología y experencias humanas
    - fundamentos de la conciencia
    - variedades de referencia a si mismo (*self-reference*)
    - variedades de descripciones de si mismo (*self-description*)
  - Primera definición de un sistema reflexivo
- Experimentos en computación (1969/1986)
  - implementación de lenguajes reflexivos
    - 3LISP (Smith/DesRivieres), 3KRS (Van Marck), ObjVLisp (Cointe/Briot),. . .
    - Smalltalk (Ingalls/Deutsch), Scheme (Steel), Loops (Bobrow), . . .
  - Nueva definición de un sistema reflexivo por Pattie Maes y Luc Steels

Defi niciones

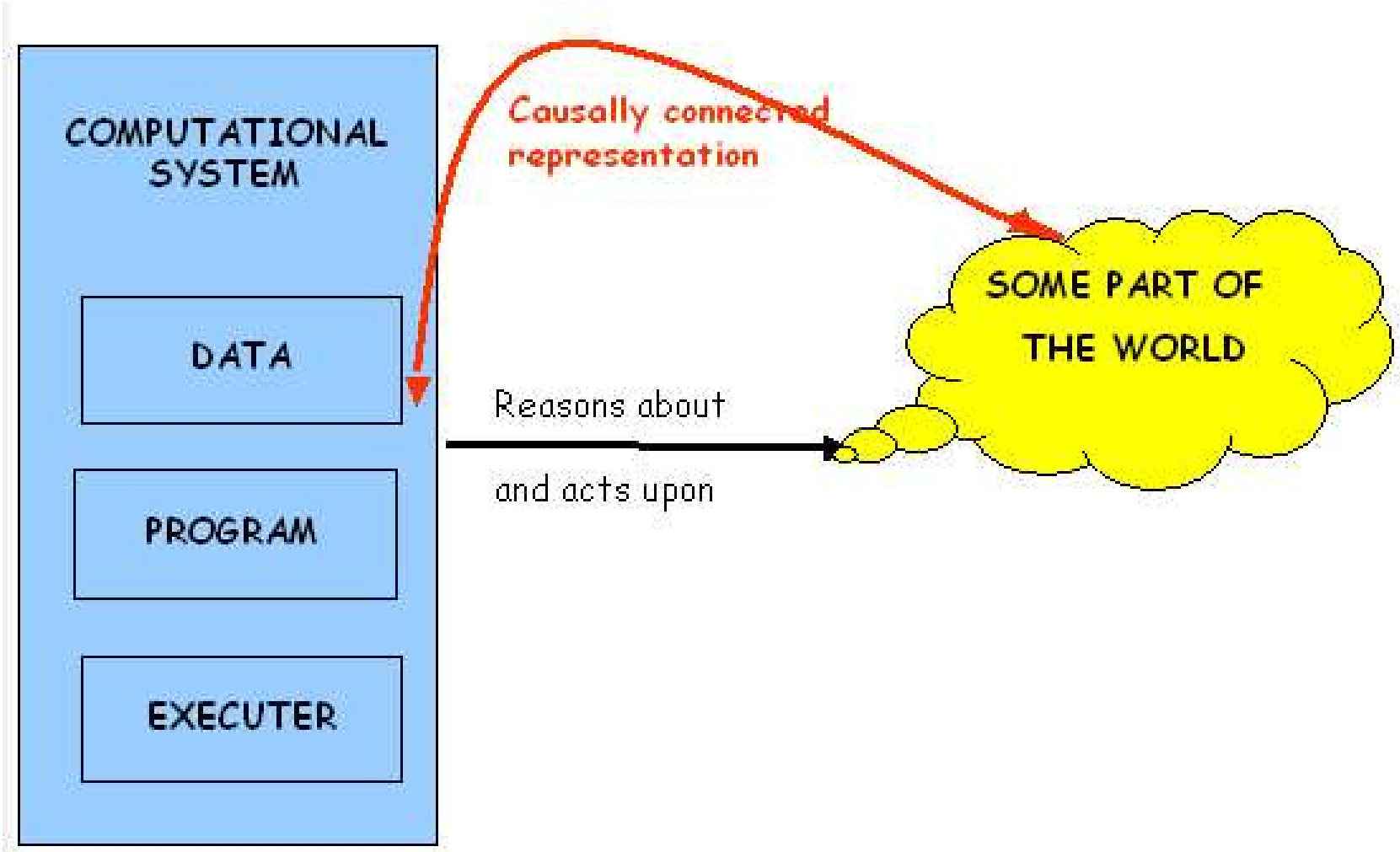
La defi nición de Smith

"La habilidad integral de un proceso de representar, operar y tratarse a si mismo de la misma manera en que representa, opera y trata su tema principal."

Defi niciones de Pattie Maes (PhD, VUB, 1987)

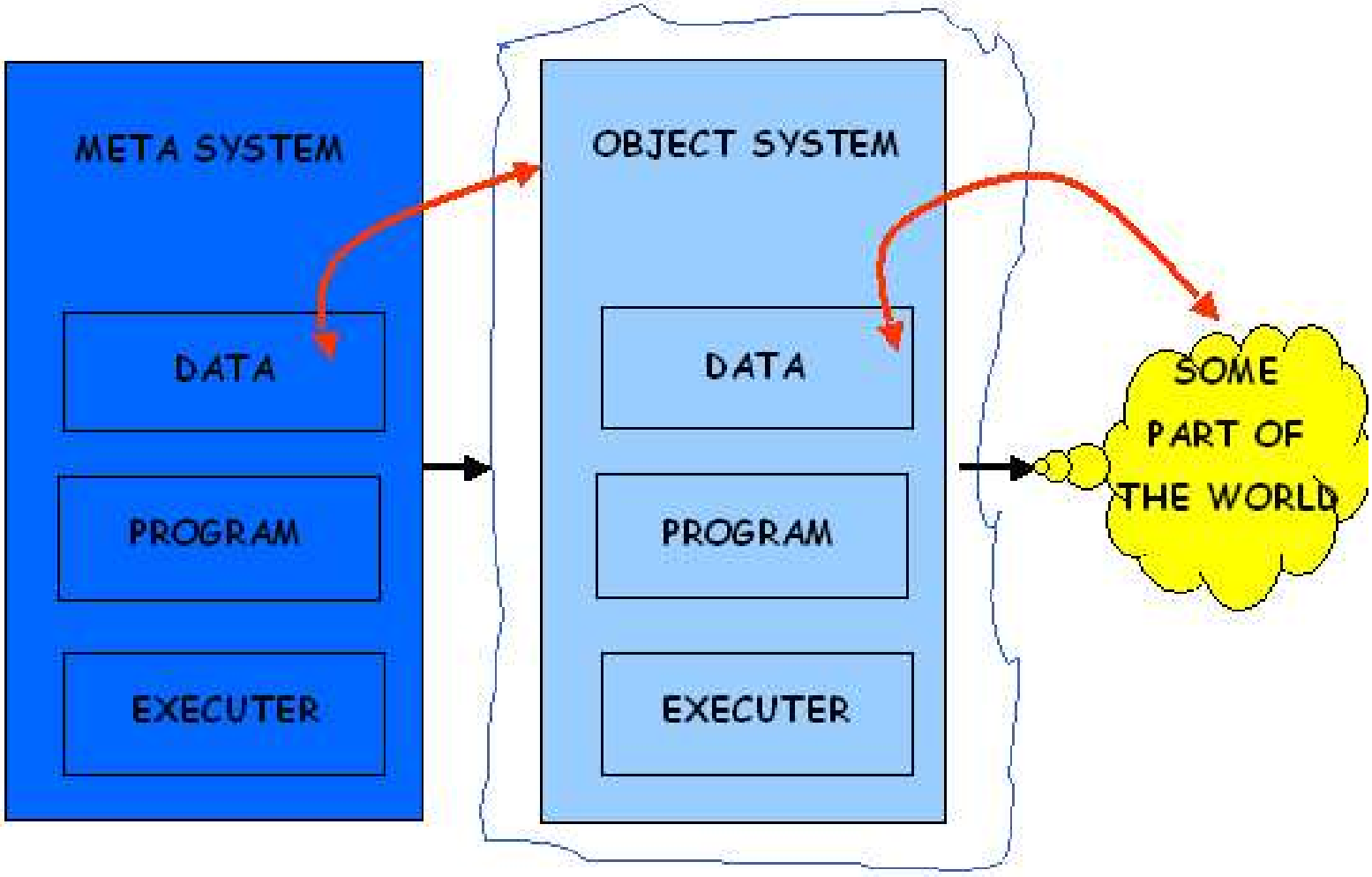
- un Sistema Computacional (SC) es algo que **razona y actua sobre** una parte del mundo, llamada el dominio del sistema.
- un SC es **causalmente conectado** a su dominio si el sistema y su dominio estan vinculado de modo que un cambio en uno provoca un efecto sobre el otro.
- un Sistema Meta es un SC que tiene como dominio **otro SC**, llamado su sistema-objeto.
- Reflexión es el proceso de razonar y/o actuar sobre **si mismo**.
- un Sistema Reflexivo es un meta-sistema causalmente conectado a si mismo (su sistema-objeto es si mismo).

Un sistema computacional



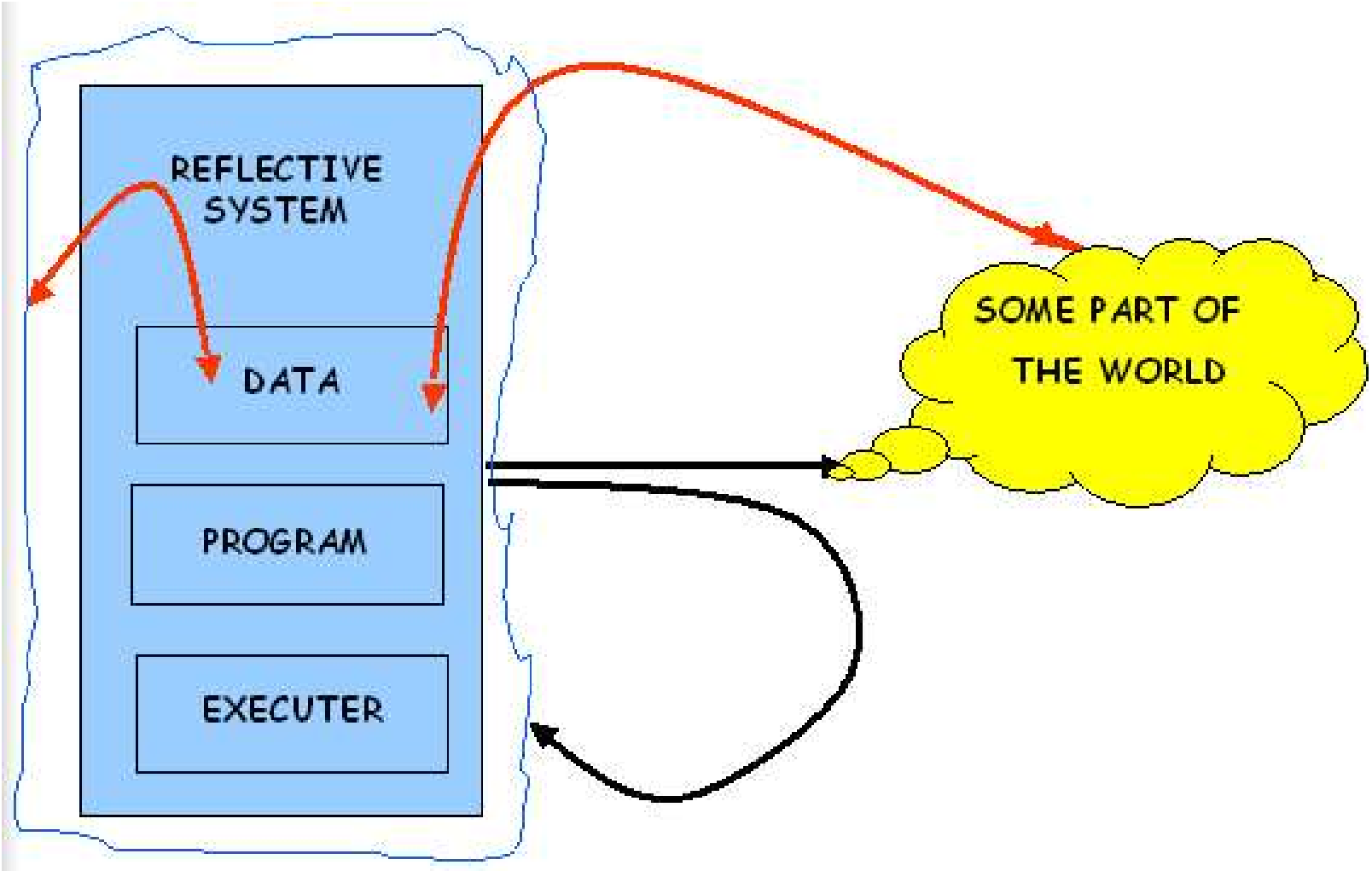
ECOLE DES MINES DE NANTES

Un sistema meta



ECOLE DES MINES DE NANTES

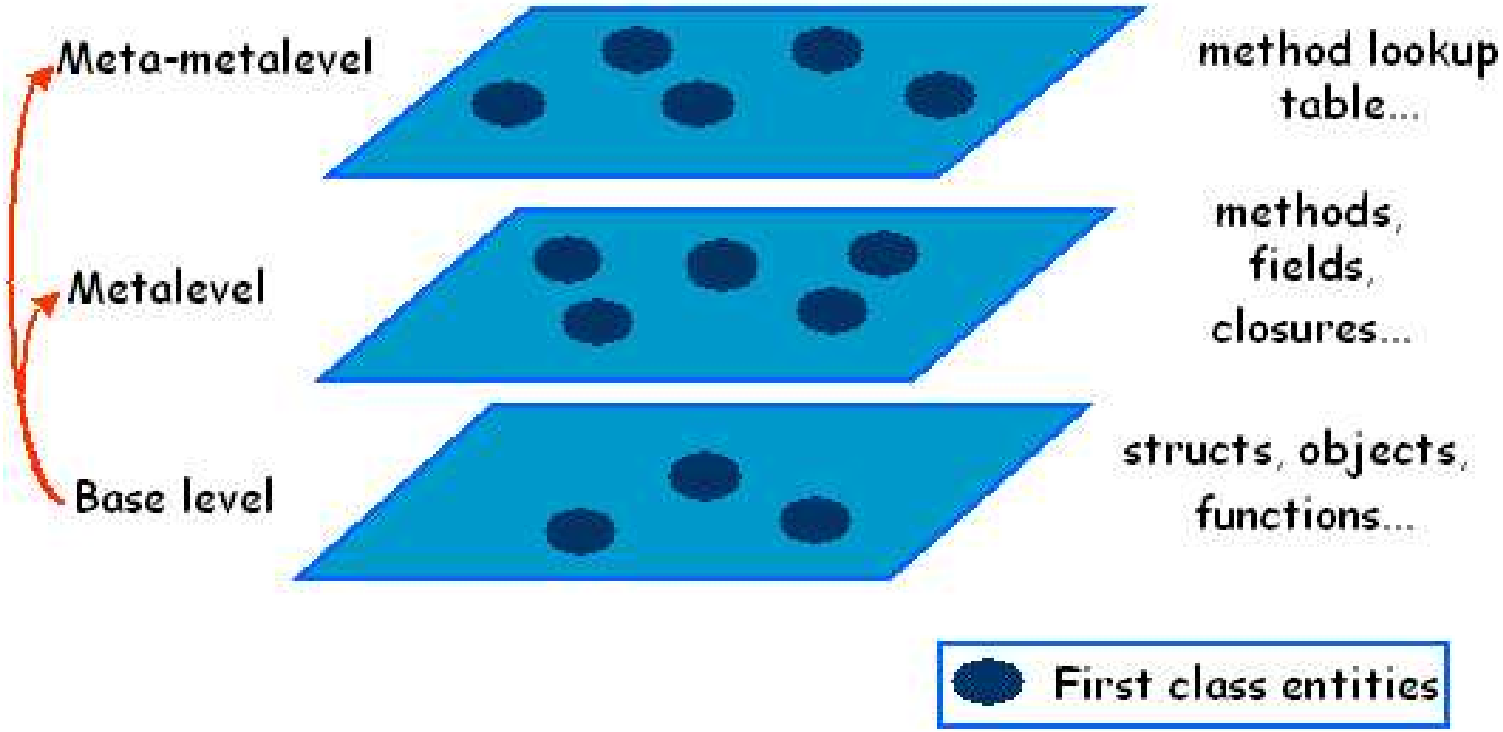
Un sistema reflexivo



ECOLE DES MINES DE NANTES

Niveles meta y base

Un **metanivel** *reifica* lo que es *implicito* (mecanismos, estructuras) al respectivo **nivel de base**



Construir un sistema reflexivo

Ferber, OOPSLA'89

- cuales entidades tienen que ser reificadas (i.e., transformadas en algo que sea manipulable al metanivel)?
- como implementar el lazo causal?
- cuando el control pasa al metanivel?
- como tratar con el problema de regresión infinita?

Cuales entidades reifi car?

- lenguajes funcionales:
  - lambda/clausuras, ambientes, continuaciones,. . .
- lenguajes OO:
  - métodos, clases, mensajes, campos,. . .
- lenguajes OO concurrentes:
  - threads, schedulers, monitores,. . .

Caracterizar un sistema reflexivo

Caracteristicas de un sistema reflexivo

- tipos de reflexión
- tiempo en que ocurre la reflexión
- nivel de asbtracción y granularidad de las meta-entidades

Tipos de reflexión

- estructural / comportamiental
  - **estructural**: razonar/actuar sobre la estructura del programa
    e.g., agregar campo/método, razonar sobre la jerarquía, . . .
  - **comportamiental**: razonar/actuar sobre el comportamiento del programa
    e.g., modificar la semantica de un método, de un mecanismo, . . .
- introspección / intercesión
  - **introspección**: consultar (*lectura*)
    e.g., ver lista de los métodos implementados, obtener nombre de una clase,. . .
  - **intercesión**: modificar (*escritura*)
    e.g., agregar un método, cambiar el nombre de una clase,. . .

Tiempo en que ocurre la reflexión (1)

- compilación
  - provee una representación OO *estática* del programa
  - funciona como un preprocesador
  - necesita codigo fuente
  - e.g., OpenJava (Chiba)
- cargamiento (Java load time)
  - provee una representación OO *estática* del programa
  - transformador de código octal (*bytecode*)
  - aplicable en sistemas dinámicos
  - e.g., Javassist (Chiba), Jinline (Tanter-Ségura)
- soló puede razonar sobre lo estático (clases), pero no tiene sobrecosto al ejecutar el programa

Tiempo en que ocurre la reflexión (2)

- ejecución
  - representación OO *dinámica* del código
  - basado en *hooks* (indirecciones) introducidas
    - por transformación código o modifi cación VM
    - construyen reifi caciones con datos dinámicos
  - extremadamente dinámico
  - e.g., Reflex (Tanter), Iguana/J (Growing-Cahill), Java Reflection API (introspección solamente)
- más poderoso (puede razonar sobre instancias), pero tiene un costo importante

Nivel y granularidad de las meta-entidades

- nivel de abstracción (afecta la usabilidad y el poder):
  - nivel del lenguaje fuente (e.g. Java):
    `Class`, **`Field`**,...
  - nivel del lenguaje intermedio (e.g. Java bytecode):
    `Invokespecial`, **`Pop`**, **`Dup`**,...
- granularidad (afecta el poder):
  - "coarse-grained": estructura de clases y miembros (e.g., Javassist)
  - "fine-grained": acceso al código de los métodos (e.g., Jinline)

Aplicaciones

Aplicaciones

- programación dinámica con el *Java Reflection API*
- reflexión estructural a *load time* con Javassist
- reflexión comportamiental con Reflex

Programación dinámica con el API estandar de Java

Java Reflection API

- “el API de reflexión de Java provee un API pequeño, type-safe, y seguro, que soporta **introspección** sobre clases y objetos en la máquina virtual actual. Si permitido por la estrategía de seguridad, el API puede ser usado para:

- construir nuevas instancias de clases y nuevos arreglos
- acceder y modificar los campos de objetos y clases
- acceder y modificar elementos de un arreglo
- invocar métodos sobre objetos y clases”

Aplicaciones principales

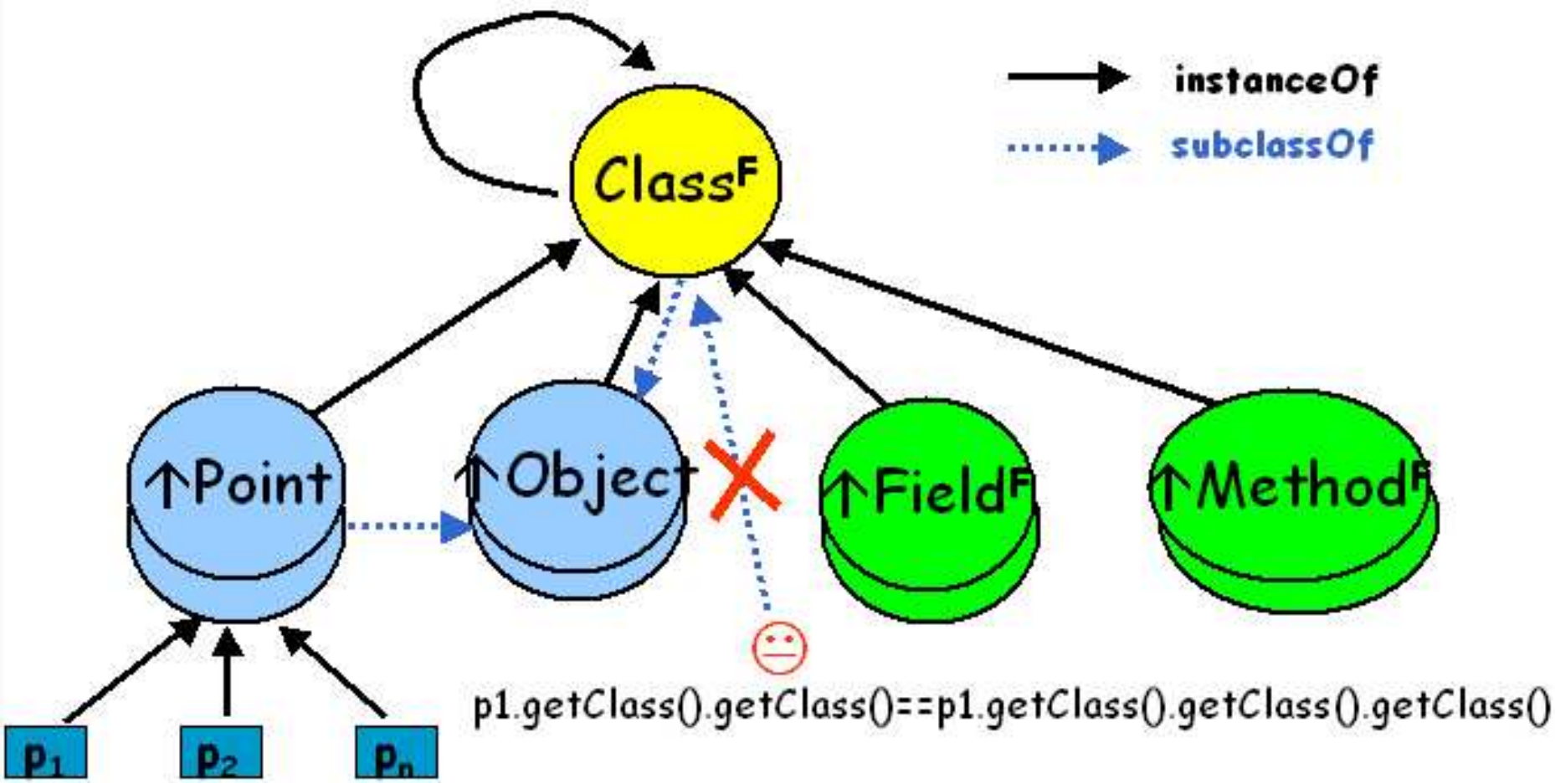
- documentación automatica
- IDEs: browsers, inspectors, debuggers,. . .
- serialización/deserialización
  - representación binaria de un objeto
  - re-creación de un objeto a partir de los bytes
  - usado para persistencia y RMI (invocación remoto)
- componentes
  - descubrimiento dinámico de las propiedades de un componente
  - JavaBeans, Enterprise JavaBeans

Clases y Interfases

- `java.lang.Object`
- `java.lang.Class`
- `java.lang.reflect.Member` (I)
  - `java.lang.reflect.Field`
  - `java.lang.reflect.Method`
  - `java.lang.reflect.Constructor`
- `java.lang.reflect.AccessibleObject`

ECOLE DES MINES DE NANTES

El modelo de clase de Java



ECOLE DES MINES DE NANTES

java.lang.Class

- introspección
  - `String getName()`
  - `Class getSuperclass()`
  - `int getModifiers()`
  - `Method getDeclaredMethod(String, Class[])`
  - `Field getDeclaredField(String)`
- invocación
  - `Object newInstance()`
- obtener una reificación de clase
  - `Class.forName("A")`
  - `aPoint.getClass()`
  - `int.class`

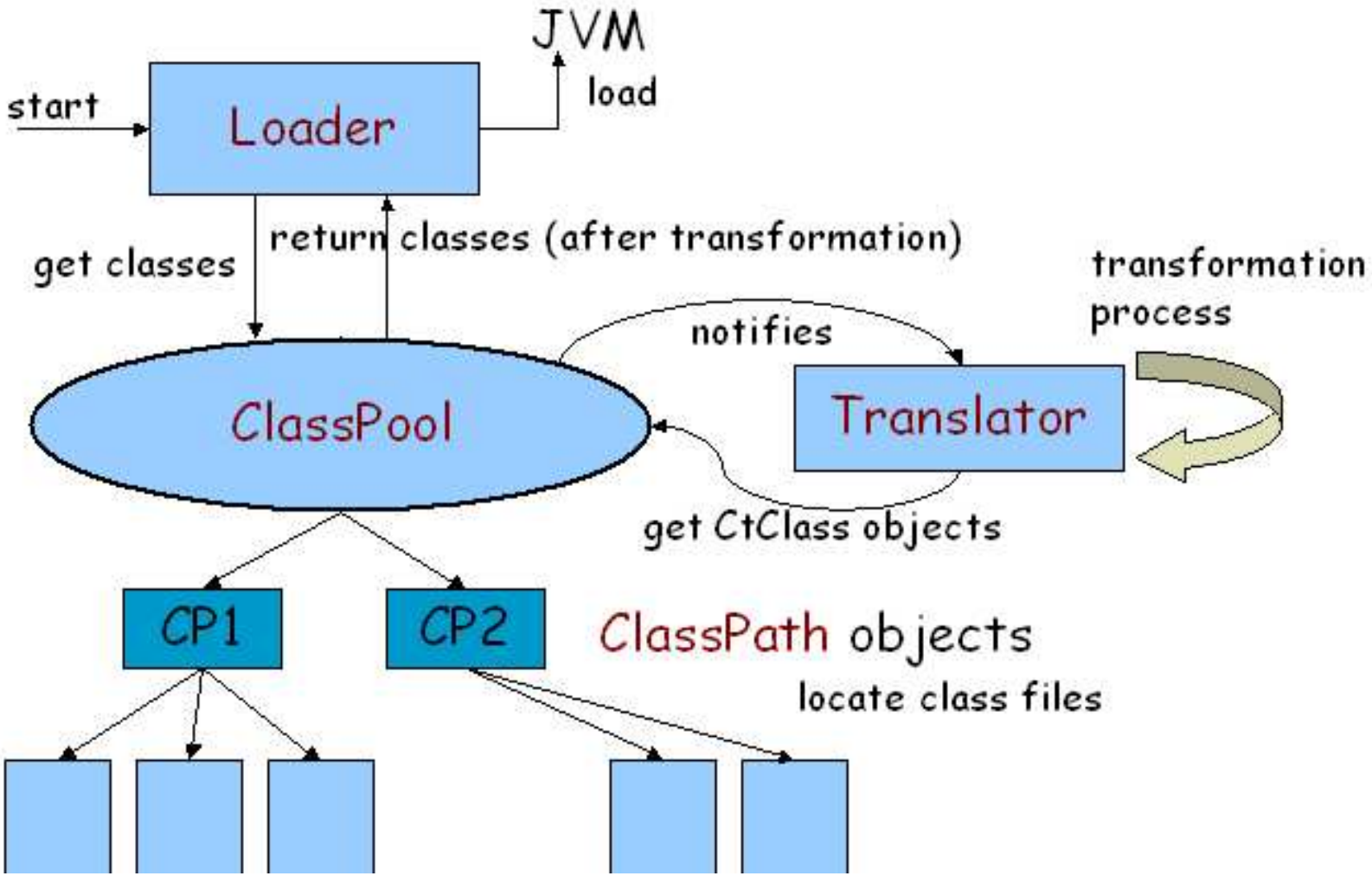
java.lang.reflect.Method

- introspección
  - `String getName()`
  - `Class getDeclaringClass()`
  - `int getModifiers()`
  - `Class getReturnType()`
  - `Class[] getParameterTypes()`
  - `Class[] getExceptionTypes()`
- invocación
  - `Object invoke(Object, Object[])`

Ejemplo de programación dinámica (1)

- “instanciador visual”: el usuario selecciona el nombre de una clase, esta instanciada, y el método **`displayXXX()`** es invocado en este objeto.
- como lo haría de manera clásica?
  - ```
    if(nombreSeleccionado.equals("A"))
        objeto = new A();
    else if(nombreSeleccionado.equals("B"))
        objeto = new B(); ...
    ```
  - ```
    if(objeto instanceof A)
        ((A) objeto).displayA();
    else if(objeto instanceof B)
        ((B) objeto).displayB(); ...
    ```
- y que pasa si el usuario puede agregar nuevas clases?
- mantención, extensibilidad?

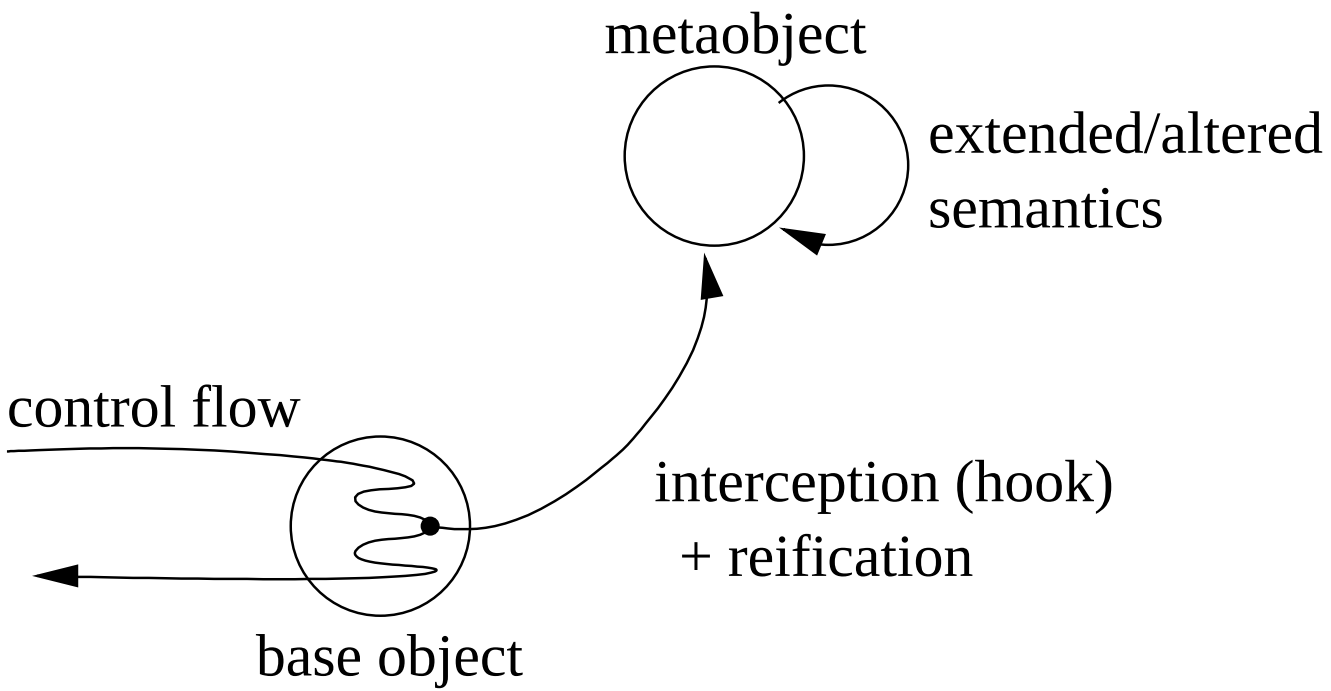
Ejemplo de programación dinámica (2)

- usando el API de reflexión Java:
  - ```
    Class clase = Class.forName(nombreSeleccionado);
    Object objeto = clase.newInstance();
    ```
  - ```
    String nombreMetodo = "display" + nombreSeleccionado;
    Method metodo = clase.getDeclaredMethod(nombreMetodo,
    null);
    ```
  - ```
    metodo.invoke(objeto, null);
    ```
- cual prefiere escribir? mantener? proveer?
    ```

Reflexión estructural con Javassist
aplicación a BCA

Sobre el `ClassLoader` de Java

- definido por la clase *abstracta* **`ClassLoader`** y sus subclases (e.g., **`NetworkClassLoader`**)
- un *class loader* es responsible por cargar classes
  - dado el nombre de una clase, tiene que intentar localizar o generar los datos (arreglo de bytes) que constituyen la definición de la clase
- los *class loaders* definen *namespaces*
  - <clase A cargada por CL1> = <clase A cargada por CL2> ssi CL1 = CL2
- aplicaciones definen subclases de **`ClassLoader`** para:
  - extender la manera de cargar clases (e.g., usando la red)
  - obtener distinctos *namespaces* protegidos
  - *para transformar bytecode!*

Javassist

- una herramienta para *reflexión estructural en tiempo de carga*
- una libreria para editar bytecode Java que permite:
  - modificar una clase cuando se carga
  - crear una nueva clase durante la ejecución
- implementado como un API de alto nivel
  - no necesita conocer bytecode
  - 100% Java, portable (JVM >= 1.2)
- herramienta estable (v2.2), muy usada, viene con código fuente
  - Shigeru Chiba, prof. Tokyo Institute of Technology, Japón
  - **`www.is.titech.ac.jp/~chiba/javassist`**

Arquitectura general de Javassist



API principal de Javassist

- abstracción nivel código fuente
- nuevas reificaciones of Java clases:
  - **`CtClass`**, **`CtMethod`**, **`CtField`**. . .
- clases para describir el cargamiento y las modificaciones
  - **`Loader`**, **`ClassPool`**, **`ClassPath`**
  - **`Translator`**
- **`ClassPool`** sirve para obtener reificaciones de clases (**`CtClass`**), y crear nuevas clases
- **`Translator`** es una interfaz implementada por los programas que quieren modificar clases

CtClass

- provee el protocolo para manipular las clases antes del cargamiento efectivo de ellas
- introspección
  - igual que **`java.lang.reflect`**
  - **`isModified`**, **`isFrozen`**
- **intercesión**
  - **`setName`**, **`setSuperclass`**, **`setModifiers`**,...
  - **`addInterface`**, **`addField`**, **`addMethod`**, ...
- también ofrece un puente a API bajo nivel (bytecode)

Ejemplo: BCA (1)

- BCA = Binary Component Adaptation (adaptación de un componente binario)
- modificar clases para que calzen dentro de un *framework*
  - tenemos una clase **`Calendar`** que implementa una interfaz **`Writable`** (framework externo)

```
class Calendar implements Writable {
   public void write(PrintStream s) {...}
}
```

  - en una nueva versión, **`Writable`** ha sido reemplazada por **`Printable`** – necesitamos adaptar **`Calendar`**

```
class Calendar implements Printable {
   public void write(PrintStream s) {...}
   public void print() { this.write(System.out); }
}
```

BCA (2)

```
class Exemplar implements Printable {
    public void write(PrintStream s) { /* dummy */}
    public void print() { write(System.out); }
}
```

```
class Adaptor implements Translator {

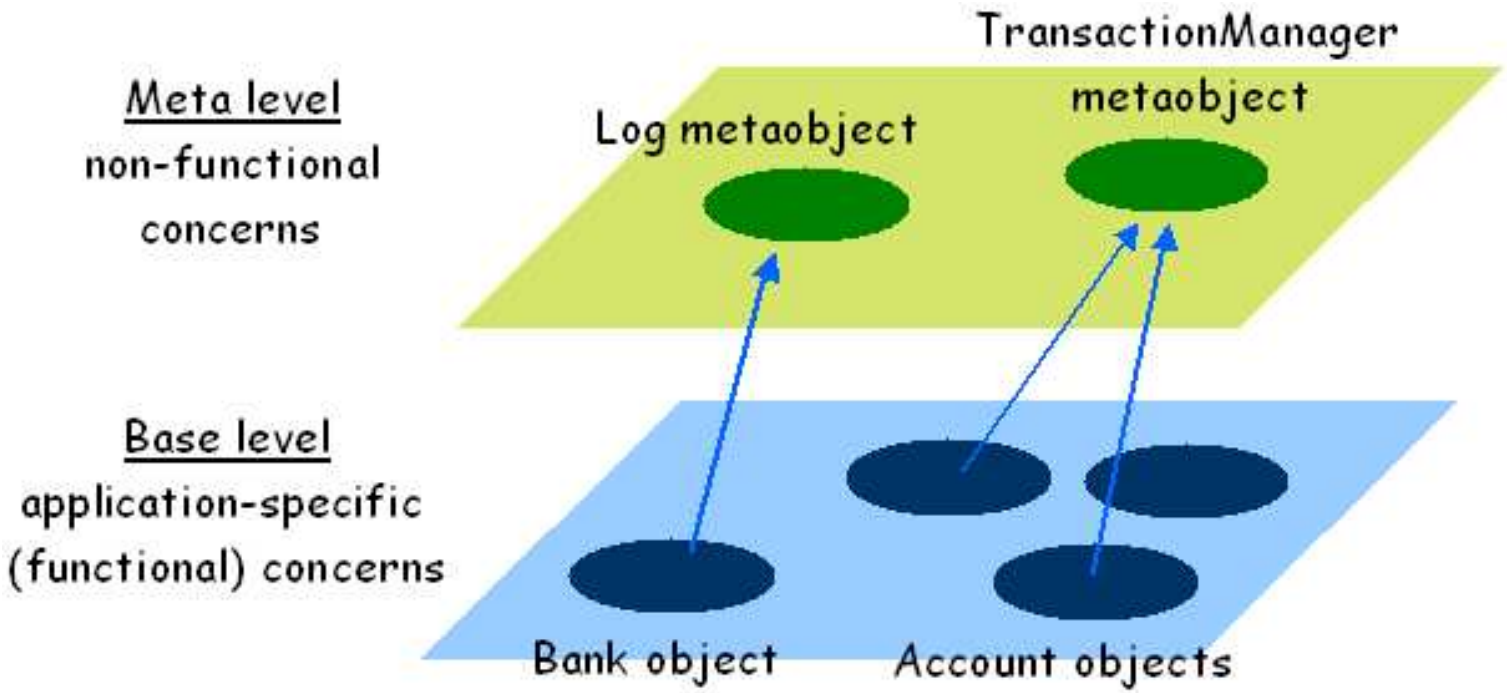
  CtMethod printM;
  CtClass printable;

  public void start(ClassPool pool){
     this.printable = pool.get("Printable");
     CtClass exemplar = pool.get("Exemplar");
     this.printM = exemplar.getDeclaredMethod("print", [ ]);
  }

  // to be continued...
```

ECOLE DES MINES DE NANTES

BCA (3)

```
// class Adaptor continued...

public void onWrite(String classname, ClassPool pool){
   CtClass c = pool.get(classname);
   CtClass[ ] interfaces = c.getInterfaces();
   for(int i=0; i<interfaces.length; i++){
    if(interfaces[i].getName().equals("Writable")) {
      interfaces[i] = printable;
      c.setInterfaces(interfaces);
      c.addMethod(CtNewMethod.copy(printM, c, null));
      return;
} } }
```

ECOLE DES MINES DE NANTES

Finalmente:

```
public class RunAdaptation {
    public static void main(String[ ] args) {
        Adaptor adapt = new Adaptor();
        ClassPool pool = ClassPool.getDefault(adapt);
        Loader cl = new Loader(pool);
        cl.run("MyApplication", args);
}
```

ECOLE DES MINES DE NANTES

Reflexión comportamiental con Reflex
aplicación a SOC

Reflexión comportamiental con metaobjetos

- se asocian **metaobjetos** a los objetos de la aplicación (objetos de base)
- los metaobjetos *extenden* o *modifican* la semantica de un(os) mecanismo(s)
  - invocación de metodo, envio de mensaje, acceso a campo, creación, serialización,. . .
- el vinculo entre un objeto de base y su(s) metaobjeto(s) se llama *vinculo de conexión causal*, o *metavinculo*

Ilustración



Ejemplo

- uso de un metaobjeto para tracear invocación de métodos:
  - ```
    Object handleInvoke(Method m, Object r, Object[] args){
        log(currentTime + ":  " + m.getName() + "on " + r);
        return m.invoke(r, args);
    }
    ```
- los argumentos del método del metaobjeto son **reificaciones** de lo que ocurre al nivel de base
- código genérico, independiente del objeto de base
- código de base inconsciente del código meta
- encapsula una preocupación no-funcional!
  - código de base reusable
  - código meta reusable


    ```

SOC con reflexión comportamiental



ECOLE DES MINES DE NANTES

Reflex (1)

- un sistema **abierto** para reflexión comportamiental en Java
- motivación: *justo lo que uno necesita de reflexión, no más (ya que es caro)*
- 100% Java, portable, inserta los *hooks* por transformación de bytecode Java
- integra con Javassist y Jinline
- **`www.dcc.uchile.cl/~etanter/Reflex`**

Reflex (2)

- soporta todos los mecanismos disponibles al nivel de bytecode + especializaciones
- soporta multiples mecanismos, multiples metavinculos
- mecanismo de especialización del protocolo de los metaobjetos (MOP)
  - basado en MOP genérico, synchronous
  - se puede definir otros protocolos facilmente
- selección de los objetos reflexivos y mecanismos reificados
  - selección muy fina (e.g., solo tal instancia y solo para método **`m`** y **`n`**)
  - estaticamente: archivo de configuración
  - dinamicamente: servicios de creación (e.g., **`Reflex.createObject()`**)

Ejemplo

- definición de un metaobjeto para tracear invocación de métodos:

```
public class TraceMetaobject implements Reflex.MOP {
  public Object handleInvoke(Method m, Object r, Object[] args){
    log(currentTime + ":  " + m.getName() + "on " + r);
    return m.invoke(r, args);
} }
```

- obtener un objeto reflexivo

```
MOP trace = new TraceMetaobject();
Vector v = (Vector) Reflex.createObject("java.util.Vector", trace);
```

- transformar clase automaticamente a load time

```
<objectHook
   name="invoke"
   mechanism="reflex.msgreceive.MsgReceive"
   metaobject="jcc.examples.TraceMetaobject" />
```

```
<classFilter
   class="jcc.examples.FilterVector"
   hook="invoke" />
```

ECOLE DES MINES DE NANTES

Aplicaciones

- aplicación no-trivial a SOC:
  - *Managing references upon object migration: applying SOC*, Proceedings SCCC 2001
- extensión a adaptación dinámica:
  - *Towards transparent adaptation of migration policies*, Proceedings ECOOP Workshop Mobile Object Systems 2002
- otras aplicaciones de MOP a sistemas distribuidos:
  - ProActive, Denis Caromel (INRIA Sophia/France)
    charla SCCC 2002: viernes 8 Nov. – 12:20/13:10

Conclusiones sobre reflexión y metaprogramación

Cumplimientos

- marco conceptual poderoso, con varias areas de aplicaciones
- para SOC
  - soporte para modularizar preocupaciones no-funcionales:
    $\Rightarrow$ *metaobjetos, metaprogramas*
  - soporte para conectar estos modulos:
    $\Rightarrow$ *hooks + metavinculo*
  - independencia programa funcional/preocupaciones no-funcionales:
    $\Rightarrow$ *programa funcional inconciente del metaprograma, metaprograma bastante genérico*
  - poder de expresividad para modulos no-funcionales:
    $\Rightarrow$ *reificaciones + posibilidad de reflexión*

Limitaciones

- reflexión comportamiental es cara
  - reflexión parcial (Reflex)
  - evaluación parcial (N. Jones, DIKU) de la reflexión (J. Noyé, EMN/INRIA)
- complejidad
  - arquitecturas meta
  - poderoso y, entonces, peligroso
  - extremadamente genérico (*demasiado?*)
  - basado en abstracciones del lenguaje de implementación
- semantica
  - no hay ninguna teoría completa de la reflexión
  - metodología para definir metaobjetos transversales (*crosscutting metaobjects*)
  - reglas/esquemas de composición de metaobjetos

Programación por Aspectos (AOP)

Contenido

- Aparición y conceptos
- Lenguajes de aspectos
- Varias propuestas

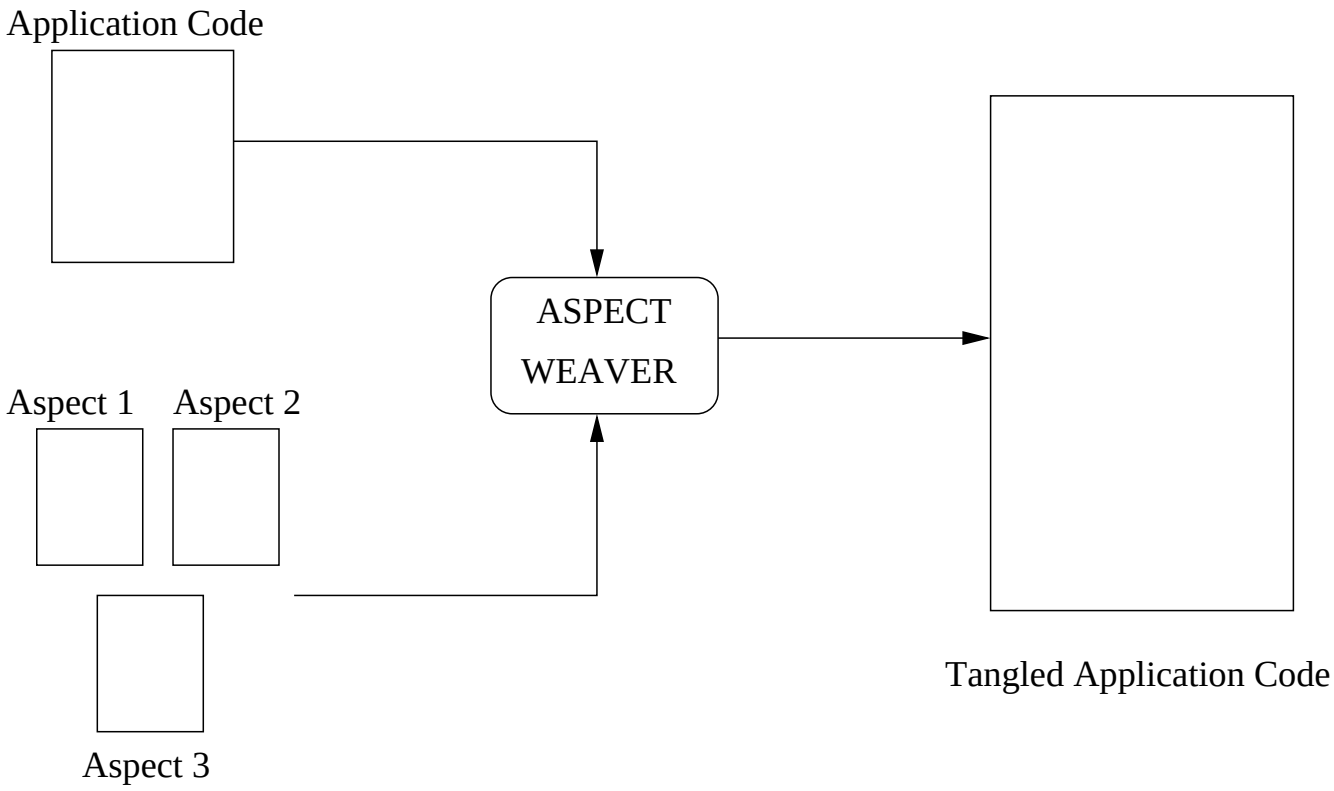
Perspectiva histórica

- Gregor Kiczales, U. British Columbia
- comunidad de la reflexión (*The Art of the MOP*, CRC Press, 1991)
- reflexión es demasiado poderoso, muchas aplicaciones
- *crear un subconjunto estructurado, dedicado a SOC*
  - esconder el *meta*
  - dar la ilusión de siempre quedar en el nivel de base
  - poner esto al alcance de todo programador
  - soporte conceptual mas adecuado para preocupaciones transversales

Kiczales et al., *Aspect-Oriented Programming*, ECOOP 1997

Conceptos (1)

- separar los preocupaciones (*aspectos*) en tiempo de programación
- un *tejedor de aspectos* produce el programa final a partir del código funcional de la aplicación y de la definición de los aspectos



Conceptos (2)

- unos conceptos se hacen explicitos:
  - lenguaje de *corte*: determinar donde intervienen aspectos
  - *puntos de corte* (pointcut/join points)
  - lenguaje de *acción* o de *aspectos*: especificar el comportamiento a realizar en los puntos de corte (programar los aspectos).
- parallelo con reflexión:
  - lenguaje de corte: inserción de hooks
  - lenguaje de acción: lenguaje reflexivo para programar los metaobjetos

Lenguajes de aspectos

- la idea primordial de AOP era ofrecer lenguajes de aspectos *especificos* (DSLs): ASLs
  - e.g., lenguaje para expresar el aspecto de synchronización, el de traceo, etc.
  - más cercano del programador, más directo y facil
- pero con ASLs, es imposible resolver los problemas de interacción y composición de distintos aspectos
  - e.g., aspecto de criptage + aspecto de traceo
  - cual resultado espera? (3 alternativas)
- consecuencia:
  - mucho más trabajos se focalizan en lenguaje de aspectos genérico
  - e.g., evolución de AspectJ

Distintas propuestas (1)

- Kiczales: AspectJ
  - objetivo: sacar AOP del mundo de la investigación
  - consecuencia:
    - herramienta facil de uso, mucho soporte y documentación
    - unos problemas dejados de lado
  - 1 concepto sobre Java
    - *dynamic join points*: puntos de ejecución de programas Java
  - 4 agregados
    - *point cuts*: agrupa join points y valores
    - *advice*: acción adicional que ejecutar en los joint points de un pointcut
    - *inter-class declarations*: referirse a joint points de varias clases
    - *aspect*: unidad modular de compartamiento transversal

Ejemplo AspectJ (1) [aspectj.org]

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update(this);
  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

Ejemplo AspectJ (2) [aspectj.org]

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

```
aspect DisplayUpdating {

  pointcut move():
    call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int));

  after() returning: move() {
    Display.update();
  }
}
```

Distintas propuestas (2)

- Douence/Fradet/Sudhölt (EMN/INRIA): EAOP
  - formalización y realización de herramienta para detección y resolución de problemas de interacción/composición
  - soporte para lógica temporal, dependencias, etc.
- Brichau/De Volder/Mens (VUB/BC): AOLMP
  - usa lenguaje lógico (Tyruba/SOUL) para razonar sobre programas OO
  - declaración de hechos lógicos, inferencias,. . .
  - un marco común para desarrollar varios ASLs
  - maneja composición/interacción con reglas lógicas

Conclusión

- nuevo paradigma emergente y lleno de promesas
- siguen muchos problemas que resolver
- dos tendencias principales:
  - apuro de sacar AOP al mundo industrial (AspectJ)
  - analizar bien todos los problemas y esperar que madure el tema
- referencias:
  - **`aspectj.org`**
  - **`aosd.net`**
  - conferencias: AOSD, GPCE, ECOOP, OOPSLA

Conclusiones

Conclusión

- limitaciones de la orientación al objeto
  - modularizar correctamente preocupaciones no-funcionales, transversales
- alternativas
  - reflexión y metaprogramación
  - programación por aspectos
- problemas abiertos
  - composición
  - interacción
  - metodología