Annotazioni

Partiamo da un caso piuttosto comune nell'ambito di applicazioni desktop: il programma usa immagini ed etichette per i pulsanti, le voci di menù e via discorrendo. E' lecito pensare che, quantomeno nell'ambito di una stessa organizzazione, esista un approccio standard al caricamento di tali risorse. Uno fra i tanti potrebbe essere questo. Ogni programma dispone di una directory "resources" dentro la quale sono immagazzinate le immagini e le etichette usate dal programma. Ogni programma dispone di un singleton deputato al caricamento di tali risorse ad uso delle altre parti del sistema. Volendo restare sul generico, limitiamoci a dire che in ogni sistema esiste un singleton a cui è attribuita la capacità di caricare e rendere disponibili icone e testi. Il nostro obiettivo è spiegare le vele di Java 5, probabilmente oltre i limiti del lecito, per poter scrivere:

```
public class CustomToolBar extends JToolBar {
	private @Resource Icon iconSave, iconLoad;
	private @Resource String tipSave, tipLoad;

	public @InitResources CustomToolBar() {
		JButton b = new JButton(iconSave);
		b.setToolTipText(tipSave);
		add(b);
		b = new JButton(iconLoad);
		b.setToolTipText(tipLoad);
		add(b);
	}
}
```

In altre parole vogliamo esprimere il fatto che iconSave, iconLoad, tipSave e tipLoad siano risorse, nel senso convenzionale di valori esistenti e definiti al di fuori del codice sorgente. E' quello che può fare ogni meta-linguaggio, Java 5 incluso: non interessa il come, basta il significato.

Elementi

Ecco quello che ci serve: una dichiarazione di Risorsa, una di Inizializzatore di Risorse ed una di Caricatore di Risorse. Siamo al livello di convenzioni: una risorsa è, l'abbiamo visto, un qualche valore definito esternamente al codice di cui possiamo conoscere la specie ed un simbolo di identificazione. Qui abbiamo l'annotazione:

```
@Rentention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Resource {}
```

Nome e specie sono derivabili dal campo annotato. Ad esempio, in:

```
private @Resource Icon iconSave;
```

la specie è "javax.swing.Icon" e l'identificatore "iconSave". Entrambi sono determinabili durante l'esecuzione attraverso l'introspezione.

Definiamo, sempre in via convenzionale, un Inizializzatore di Risorse come un costruttore colorato dal compito di attribuire un qualche valore ai campi risorsa:

```
@Rentention(RetentionPolicy.SOURCE)
@Target(ElementType.CONSTRUCTOR)
public @interface InitResources {}
```

Ultimo arriva l'oggetto fisicamente deputato al recupero dei valori da assegnare alle risorse. Ha lo scopo di rendere più flessibile il meccanismo di definizione di quei valori.

```
@Rention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface ResourceLoader {}
```

Per semplicità stabiliamo, ancora in via convenzionale, che chi sia marcato da ResourceLoader deva avere la forma qui esemplificata:

```
@ResourceLoader
```

```
class NomeTipo {
	public Object loadResource(Class<?> type, String name)...

	public static NomeTipo getInstance()...
}
```

Esistono una varietà di modi per evitare questa precondizione ma assumerla renderà la nostra esposizione meno bizzarra. Ricapitoliamo. Ogni volta che, nel codice, vogliamo creare un campo che sia una "risorsa", marcheremo quel campo con l'annotazione @Resource. Ai costruttori delle istanze che possiedano quei campi e ne facciano uso sarà apposto il marchio @InitResources. Uno degli oggetti presenti nel sistema sarà un "caricatore di risorse" e a tale scopo dovrà avere il marchio @ResourceLoader, un metodo loadResource ed un metodo di classe getInstance che restituisca un'istanza di sè stesso.

Impacchettano le annotazioni in una libreria e scrivendo un appunto di dieci righe sul funzionamento del tutto abbiamo per le mani una semplificazione di una certa rilevanza per le nostre applicazione.

Purchè funzioni.

Se Java 5 fosse Zeus, APT sarebbe il suo fulmine. Pomposamente, APT è il preprocessore programmabile del compilatore javac. APT è distribuito insieme al JDK 5 di Sun. Si invoca da linea di comando, come javac:

apt -d . *.java

E' un preprocessore nel senso che analizza ed eventualmente manipola il codice sorgente prima che questi sia dato in pasto al compilatore. E' programmabile nel senso che può ricevere, durante la sua invocazione, delle istruzioni circa il tipo di manipolazione da operare. Tali istruzioni sono in realtà interi programmi Java il che lascia intendere quali possano essere le implicazioni. In questo testo noi ci occupiamo di una manipolazione particolarmente invasiva avente come bersaglio i costruttori annotati con @InitResources e come parametro il tipo annotato con @ResourceLoader. In pratica lasciamo ad APT il compito di iniettare nel codice sorgente delle istruzioni standard altrimenti ripetitive. Ogni costruttore che rechi l'annotazione @InitResources dovrà invocare un metodo "automatedInitResource". Tale metodo sarà iniettato nel sorgente della classe che possieda il costruttore @InitResources. In pseudo codice, il metodo sarà:

```
private void automatedInitResources() {
	dato il ResourceLoader R
	per ogni campo C marcato @Resource di questa istanza
		assegna al campo C il valore ottento da R.loadResource(C.getType(), C.getName())
```

Riassumendo, APT dovrà cercare la classe che rechi il marchio @ResourceLoader: il suo nome pienamente qualificato servirà per costruire il metodo automatedInitResources. Poi dovrà cercare i costruttori @InitResources: in quel costruttore sarà inserita l'invocazione "automatedInitResources();" e in quella classe il metodo corrispondente. Usando la riflessione è facile costruire un metodo che "riempia" tutti i campi "@Resource". Dato XXX, nome pienamente qualificato del tipo recante l'annotazione @ResourceLoader, il metodo sarà:

```
/* APT INJECTION START */
private void automatedInitResources() {
	XXX loader = XXX.getInstance();
	for(java.lang.reflect.Field f : getClass().getDeclaredFields()) {
		if(f.isAnnotationPresent(it.resourceloader.Resource.class)) {
			try {
				f.set(this, loader.loadResource(f.getType(), f.getName()));
			} catch(Exception ex) {
				ex.printStackTrace();
			}
		}
	}
}
/* APT INJECTION END */
```

Un programma per APT.

APT assume i programmi attravero il pattern "factory". Gli strumenti necessari a creare un processore ed un suo produttore di trovano nel pacchetto "tools.jar" che trovate nella directory "lib" del JDK. L'uso di un produttore personalizzato richiede che il nome della classe produttrice sia passato durante l'invocazione di APT attraverso il

parametro -factory:

```
apt -factory mypackage.MyAnnotationProcessorFactory ...seguono i parametri di javac
```

APT invoca il metodo "getProcessorFor" del produttore per l'insieme di dichiarazioni di tipo annotazione che riscontra nei sorgenti da compilare se tali annotazioni siano "supportate" dal produttore. Il produttore dichiara il supporto a certe annotazioni attraverso un metodo "supportedOptions": tale metodo restituisce un insieme di stringhe ognuna delle quali corrisponde al nome pienamente qualificato di un tipo annotazione. L'insieme di stringhe costituisce il gruppo di annotazioni supportate dal processore. Secondo quanto esposto, il nostro processore esamina le annotazioni "InitResources" e "ResourceLoader":

```
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.type.*;
import com.sun.mirror.util.*;
import java.util.*;

public class RLPFactory implements AnnotationProcessorFactory {

	private static final Collection<String> supportedAnnotations
		= Collections.<String>unmodifiableCollection(Arrays.asList(
			"InitResources",
			"ResourceLoader"));

	private static final Collection<String> supportedOptions
		= Collections.<String>emptySet();

	public Collection<String> supportedAnnotationTypes() { return supportedAnnotations; }
	public Collection<String> supportedOptions() { return supportedOptions; }

	public AnnotationProcessor getProcessorFor(
		Set<AnnotationTypeDeclaration> d,
		AnnotationProcessorEnvironment e)
	{
		return new ResourceLoaderProcessor(e);
	}
}
```

Il succo della questione sta nel metodo getProcessorFor. Più o meno dice "dato l'insieme di annotazioni D e l'ambiente APT E, restituiscimi un analizzatore". Il nostro codice crea e restituisce un'istanza di ResourceLoaderProcessor.

ResourceLoaderProcessor.

Un AnnotationProcessor possiede un laconico metodo process(). A conti fatti, è un alter ego di Runnable: qualcuno che fa qualcosa di indefinito. Stupisce, in verità, quante convenzioni possano darsi in un insieme di librerie. Il nostro processore riceve in costruzione in riferimento di tipo AnnotationProcessorEnvironment. Questo parametro è una rappresentazione del preprocessore APT. Tra le molte informazioni ricavabili, AnnotationProcessorEnvironment può dire quali siano le dichiarazioni di tipo soggette ad analisi: per farla breve, si tratta dell'appiglio per capire quali sorgenti siano sottoposti alle attenzioni di APT.

```
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.type.*;
import com.sun.mirror.util.*;
import java.util.*;

public class ResourceLoaderProcessor implements AnnotationProcessor {
	private final AnnotationProcessorEnvironment env;

	public ResourceLoaderProcessor(AnnotationProcessorEnvironment e) {
		env = e;
	}

	public void process() {
		ArrayList<TypeDeclaration> tds = new ArrayList<TypeDeclaration>();
		/* Trova l'elemento annotato con ResourceLoader */
		String resourceLoaderTypeName = null;
		typeloop:
```

```
                        for(TypeDeclaration t : env.getSpecifiedTypeDeclarations()) {
                                Collection<AnnotationMirror> annotations = t.getAnnotationMirrors();
                                for(AnnotationMirror m : annotations) {
                                        AnnotationTypeDeclaration d = m.getAnnotationType().getDeclaration();
                                        if(d.getQualifiedName().equals("ResourceLoader")) {
                                                resourceLoaderTypeName = t.getQualifiedName();
                                                continue typeloop;
                                        }
                                }
                                tds.add(t);
                        }
                for(TypeDeclaration t : tds) {
                        t.accept(DeclarationVisitors.getDeclarationScanner(
                                new InitResourceOwnerVisitor(resourceLoaderTypeName),
                                DeclarationVisitors.NO_OP));
                }
        }
}
```

Il nostro processore ha come primo obiettivo la scoperta del tipo annotato @ResourceLoader. La composizione del metodo automatedInitResources() richiede, infatti, che sia noto il nome pienamente qualificato della classe che possieda i metodi getInstance e loadResource. Essendo ignoto l'ordine di scansione dei sorgenti, accumuliamo le dichiarazioni di tipo (TypeDeclaration) mentre cerchiamo di capire chi sia il @ResourceLoader. Al termine della prima scansione, procederemo quindi a mettere fisicamente le mani sul codice.

The Visitors.

Dopo aver imposto ordine al caos, il metodo process() di ResourceLoaderProcessor passa ad esaminare il codice delle dichiarazioni di tipo (tipo classe o interfaccia o annotazione primari). Per farlo candida un "visitatore di dichiarazioni". Un DeclarationVisitor è una sorta di lettore del il codice sorgente capace di informare il programmatore sul significato degli elementi che incontra durante la sua scansione. La candidatura è realizzata attraverso il metodo accept(DeclarationVisitor) del tipo Declaration (di cui TypeDeclaration è nipote). Il metodo DeclarationVisitors.getDeclarationScanner consente di collegare alla lettura del codice sorgente due visitatori, uno che opera prima del completamento della lettura l'altro a lettura completata. Il nostro codice affida le classi da precompilare ad un visitatore personalizzato, InitResourceOwnerVisitor, il cui scopo è quello di manipolare le classi in cui almeno un costruttore sia marcato @InitResources. Da notare solo il fatto che il nostro InitResourceOwnerVisitor riceva in costruzione il nome pienamente qualificato del tipo annotato con @ResourceLoader.

```
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.type.*;
import com.sun.mirror.util.*;
import java.util.*;
import java.io.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.nio.*;

public class InitResourceOwnerVisitor extends SimpleDeclarationVisitor {
        private String resourceLoaderClassName;

        public InitResourceOwnerVisitor(String loaderClassName) {
                resourceLoaderClassName = loaderClassName;
        }

        public void visitConstructorDeclaration(ConstructorDeclaration d) {
                Collection<AnnotationMirror> annMirr = d.getAnnotationMirrors();
                boolean isInitResource = false;
                for(AnnotationMirror mirr : annMirr) {
                        AnnotationTypeDeclaration dec = mirr.getAnnotationType().getDeclaration();
                        if(dec.getQualifiedName().equals("InitResources")) {
                                isInitResource = true;
                                break;
                        }
                }
                if(isInitResource) {
                        SourcePosition pos = d.getPosition();
                        StringBuilder source = new StringBuilder();
                        loadFully(pos.file(), source);
```

```
                        //elimina l'annotazione @InitResource
                        //Immette nel costruttore l'invocazione automatedInitResources();
                        //Immette nella classe il metodo private void automatedInitResources();
                        //nel caso in cui non sia già presente
                        replace(pos.file(), source);
                }
        }

        private void loadFully(File f, StringBuilder dest) {
                //carica il file f nello StringBuilder dest
        }

        private void replace(File f, StringBuilder source) {
                //riversa nel file f il contenuto di source
        }
}
```

Un DeclarationVisitor è fatto di tanti metodi che ricevono notifiche di lettura di certi elementi individuati nel codice sorgente. Qui sopra, il metodo visitConstructorDeclaration, sovrascrittura di quanto ereditato da SimpleDeclarationVisitor, è invocato quando lo scanner rilevi la sintassi di un costruttore Java. Nel codice verifichiamo preliminarmente che sia presente un'annotazione @InitResources. Ricordiamo, infatti, che noi abbiamo costruito un processore per annotazioni @ResourceLoader e @InitResources a cui APT affida, quando rilevi tali annotazioni, tutto i sorgenti esaminati. Interessante è la dotazione di un metodo getPosition affidata agli oggetti Declaration. La posizione è in primo luogo il file contenente la dichiarazione, accompagnato dal numero di linea e di colonna, il tutto incapsulato in un oggetto SourcePosition. Da notare come le informazioni di posizione siano disponibili solo per quelle dichiarazioni desunte da codice sorgente: il framework stabilisce infatti la rilevabilità di dichiarazioni estratte da codice compilato, per le quali non sarà offerta alcuna posizione. InitResourceOwnerVisitor termina le indicazioni di massima sulla costruzione di un programma per l'analisi e la manipolazione del codice sorgente.

Dal preprocessore al compilatore.

Supponendo di aver impacchettato le classi RLPFactory, ResourceLoaderProcessor e InitResourceOwnerVisitor in un archivio jar (in ipotesi RLP.jar) possiamo fare un passo indietro e tornare al codice con cui abbiamo iniziato il discorso.

```
public class CustomToolBar extends JToolBar {
        private @Resource Icon iconSave, iconLoad;
        private @Resource String tipSave, tipLoad;

        public @InitResources CustomToolBar() {
                JButton b = new JButton(iconSave);
                b.setToolTipText(tipSave);
                add(b);
                b = new JButton(iconLoad);
                b.setToolTipText(tipLoad);
                add(b);
        }
}
```

Abbiamo una classe che fa uso di annotazioni di cui conosciamo il significato convenzionale e per le quali abbiamo creato un programma di precompilazione. Non resta che compilare usando APT:

apt -cp .;RLP.jar -factory RLPFactory -d . CustomToolBar.java

Il parametro -factory istruisce APT sul produttore di processori per annotazioni da usare per la compilazione dei sorgenti. Indicativamente, il prodotto della precompilazione muterà il file CustomToolBar.java in:

```
public class CustomToolBar extends JToolBar {
        private @Resource Icon iconSave, iconLoad;
        private @Resource String tipSave, tipLoad;

        public CustomToolBar() {
                /* APT INJECTION */
                automatedInitResources();
                /* APT INJECTION */

                JButton b = new JButton(iconSave);
                b.setToolTipText(tipSave);
```

```
                add(b);
                b = new JButton(iconLoad);
                b.setToolTipText(tipLoad);
                add(b);
        }

        /* APT INJECTION START */
        private void automatedInitResources() {
                TipoResourceLoaderXXX instance = TipoResourceLoaderXXX.getInstance();
                for(java.lang.reflection.Field f : getClass().getDeclaredFields()) {
                        if(f.isAnnotationPresent(Resource.class)) {
                                try {
                                        f.set(this, instance.loadResource(f.getType(), f.getName());
                                } catch(Exception ex) {
                                        ex.printStackTrace();
                                }
                        }
                }
        }
        /* APT INJECTION END */
}
```

Il sorgente sarà poi passato a javac per la compilazione. Vale la pena di sottolineare che il processore qui scritto modificherà definitivamente il codice sorgente per cui è consigliabile una prolungata fase di sperimentazione prima di pasticciare qualcosa di delicato.

Due parole sul perchè.

Accademicamente, perchè si può. In prospettiva pratica chi scrive è piacevolemente sorpreso dalle possibilità offerte da un preprocessore integrato nella piattaforma con le caratteristiche di analisi offerte da apt. Si è visto un caso certo rozzo ma che ha un suo valore in termini di sollievo dalla necessità di scrivere istruzioni ripetitive.

Il Re è morto, evviva il Re.

La presenza delle annotazioni ed la distribuzione di un preprocessore programmabile sarebbero sufficiente a far ritenere sepolto il linguaggio di programmazione Java, sostituito da J@v@. Manipolare i sorgenti, specialmente con un sistema piuttosto efficace quale APT, significa poter scrivere:

```
@Bean
class Pippo
        @BeanField int x, y, z;

@MainOwner
public class Main {
        void main() {
                Pippo p = new Pippo();
                p.setX(100);
                p.setZ(p.getX() + p.getY());
        }
}
```

e bizzarrie ancora maggiori. Non credo che sia un male in sè. Credo anche che esista un solo limite auspicabile: che i sorgenti recanti annotazioni "generatrici" non circolino al di fuori dell'organizzazione che li abbia prodotti neppure se accompagnati dal processore di annotazioni dedicato. E' importante, dal punto di vista della conoscibilità del linguaggio di programmazione Java™, che i programmatori non siano costretti a saltare da un insieme di annotazioni ad un altro o, per dirla con alcuni, da un dialetto Java ad un altro, finendo col non avere cognizione di alcuna lingua.