



Meta-programmare

Tecniche per creare applicazioni flessibili e configurabili

Nando Dessena, Ethea (www.ethea.it)

Delphi Day 2006, Piacenza 09/06/2006

1 Introduzione

Questa sessione **non tratta di tecnologia**. Le tecnologie si susseguono e passano di moda, e destano (almeno nel sottoscritto) un interesse solo relativo. Le **metodologie** e le **tecniche**, viceversa, se valide sono sempre attuali e apportano molto più valore al bagaglio di uno sviluppatore. Le tecniche che vado a illustrare consentono, insieme a molte altre, di progettare software che possa accogliere cambiamenti ed evoluzioni con pochi traumi. Credo fermamente che questa debba essere la caratteristica fondamentale del software di oggi, perciò elencherò alcune tecniche per creare applicazioni estremamente configurabili, cioè che possano essere ampiamente personalizzate tramite **meta-dati** esterni, senza necessità di modifica o ricompilazione. Elencherò anche i vantaggi che tali tecniche permettono di conseguire, mostrando esempi concreti.

2 Cos'è per me la meta-programmazione

Tra i requisiti impliciti delle applicazioni moderne troviamo senza dubbio la flessibilità e la capacità di adattarsi a contesti che cambiano con rapidità sempre crescente. Questa capacità di adattamento può essere conseguita con varie tecniche, che hanno tutte in comune un aspetto: rendere il codice che si scrive più generico e dare la possibilità di configurarlo, o parametrizzarlo dall'esterno; in altre parole, **portare i dettagli fuori dal codice**. A questo scopo, si fa in modo che il codice sia in grado di interpretare meta-dati che ne condizionano il funzionamento.

Ho visto usare il termine "metaprogramming" per la prima volta nell'ottimo libro "The Pragmatic Programmer" di Andrew Hunt e David Thomas; non so se ne siano loro gli inventori, ma in ogni caso la parola rende bene il concetto che intendo esprimere in questa sede.

Chiamo meta-programmazione, o programmazione per meta-dati, o anche programmazione dichiarativa, un particolare modo di progettare il software. In ogni progetto si traccia una immaginaria linea tra ciò che deve essere svolto dal codice e ciò che invece cambia a seconda dei dati. La mia tendenza è quella di **spostare questa linea** in modo che più cose, in un programma, dipendano dai dati, e meno dal codice. In altre parole, trasformo una parte del codice, specie dettagli che **sospetto possano variare facilmente**, in dati.

Trasformare il codice in dati ne consente, a run time, la **memorizzazione** (su file o database), la **modifica** (sia interattiva, sia automatizzata), il **trasferimento** (tra applicazioni o computer).

Questo purché i dati suddetti siano memorizzati in formati comodi da interpretare, modificare e trasferire (e qui la semplicità e la potenza dei formati **testo**, incluse le derivazioni a marcatori come **XML**, si fanno sentire).

3 Tecniche per meta-programmare

Esistono molte tecniche per "esternalizzare" (perdonate il termine) parti di un'applicazione secondo la filosofia appena esposta. Ma quali parti di un'applicazione è opportuno rendere pilotabili e sostituibili dall'esterno? La risposta a questa domanda varia in funzione dei singoli casi, e più avanti mostrerò esempi concreti in modo da dare un'idea più precisa. In linea di massima, però, si può dire che **non ci sono limiti**. L'unica difficoltà sta nel **rendersene conto**.



Sarà senz'altro capitato a molti di voi di usare le tabelle di lookup per rendere flessibili elenchi di scelte (ad esempio una modalità di pagamento, o un altro tipo di caratterizzazione). Si tratta di un modo comodo per consentire di estendere l'elenco delle modalità di pagamento in seguito senza agire sull'applicazione. Sono anche certo che un programma che richiedesse, per supportare l'aggiunta di una modalità di pagamento, di essere modificato (magari in più punti) e ricompilato verrebbe guardato dai più con sospetto e, al limite, ribrezzo. Dunque, perché ciò che vale per la modalità di pagamento non può valere anche per **algoritmi** di calcolo, **regole** da applicare in una form di data-entry, **etichette** dei campi, **librerie** di accesso ai dati e tipo di database da usare, eccetera?

Tra le tecniche normalmente usate per i nostri obiettivi ci sono:

- ◆ **Caricamento dinamico di librerie/package:** suddividere l'applicazione in parti il più possibile auto-contenute e sostituibili. Strumento potente, anche se non si tratta di vera meta-programmazione in quanto la personalizzazione ha comunque luogo tramite scrittura e compilazione di codice.
- ◆ **Scripting:** demandare alcune responsabilità a codice interpretato caricato a run-time.
 - ◆ Estensione: creazione automatizzata di script da eseguire.
- ◆ Elaborazione di **template** e sostituzione di **macro**.
- ◆ **File e database di configurazione:** spostare tutto ciò che si ritiene passibile di personalizzazione in file esterni da leggere durante l'esecuzione del programma.
- ◆ **Cataloghi:** mantenere centralizzate ed esterne al programma informazioni come la struttura di un database, maschere di data-entry, griglie, campi e relative etichette, strutture dati per import/export, eccetera. Si tratta di una variante e di un'estensione del punto precedente.

3.1 Tecniche universali

Apro una parentesi. Le tecniche di massima appena elencate, perché siano efficaci, vanno accompagnate da regole universali di buona condotta che chiunque sviluppi software dovrebbe tenere sempre presenti. Ad esempio:

- ◆ Conciliare il **principio Open/Closed** (i moduli devono essere *chiusi* alla modifica ma *aperti* all'estensione) con il **principio di Occam** (non introdurre complessità non necessaria al momento).
 - ◆ "Solo ciò che non c'è non si può rompere" (Henry Ford).
- ◆ **Contare "0, 1, ∞"**. Ciò che supporta due elementi può (e deve) supportarne facilmente "infiniti". Potreste pensare che vi servano solo un log a video e un log su file, ma se domani vi servirà anche un log su socket TCP, aggiungerlo sarà semplice solo se avrete progettato fin dall'inizio per l'estensibilità.
 - ◆ "È bene farla semplice quanto possibile, ma non di più" (Albert Einstein).
- ◆ **Programmare per le interfacce e non per implementazioni** (Erich Gamma et al, a.k.a. GoF).

4 Vantaggi e costi

Quali sono dunque i vantaggi che conseguiamo dall'applicazione di tecniche dinamiche e meta-dati?

- ◆ **Estrema flessibilità e configurabilità:** possibilità di influenzare anche in maniera significativa e in modi non previsti il comportamento di un'applicazione dall'esterno. Questo può anche tornare utile per interventi urgenti da applicare "al volo" su sistemi in produzione.
- ◆ **Ridotta possibilità di introdurre bug:** il codice di base, una volta scritto e verificato, si modifica meno spesso.



- ◆ **Maggior riuso:** il codice è più generico e quindi più facile da riusare in contesti diversi da quelli per cui era stato scritto.

Ci sono degli svantaggi, o dei costi? Essenzialmente due, che possono però essere entrambi tenuti sotto controllo:

- ◆ **Tempi di sviluppo iniziale più lunghi:** scrivere e verificare un motore generico per l'importazione di file di testo è più oneroso che scrivere un motore che importa un solo tipo di file di testo di formato ben definito. Anche **molto** più oneroso, a seconda dei casi.
- ◆ **La trappola dell'eccessiva genericità:** occorre evitare di farsi prendere la mano e non cercare di scrivere codice che sia preparato a gestire qualunque remota eventualità. Un buon livello di genericità e parametrizzazione ripaga sempre gli sforzi che è costato; un livello eccessivo difficilmente lo fa, e anzi rischia di trasformarsi in un labirinto in fase di manutenzione.

5 Nella pratica: esempi di meta-programmazione

La sessione prosegue con una fase dimostrativa delle tecniche elencate sopra, prendendo come esempio il framework applicativo EDW che Ethea sta sviluppando in questo periodo. Tutti gli esempi mostrati fanno parte della demo di EDW, che può essere richiesta ad Ethea in caso si voglia approfondirne la conoscenza. Esempi:

- ◆ Uso di meta-dati per la costruzione di form di data-browsing, ricerca e data-entry. In EDW il **MetaSchema** contiene le informazioni sullo schema del database, mentre il **GUICatalog** contiene informazioni sulla struttura dei menu, delle maschere di ricerca e data-entry, sui criteri di ricerca disponibili, sui campi da mostrare, eccetera. Tutti questi dati sono memorizzati in **file XML esterni** di facile manipolazione.
- ◆ Creazione di **codice che genera codice:** i file di cui sopra possono essere creati manualmente con un editor XML, oppure tramite codice Delphi/InstantObjects (e in futuro probabilmente anche per mezzo di un IDE che servirà a lavorare con tutti i meta-dati di EDW).
- ◆ Elaborazione di **template** e sostituzione di **macro**. I **report** in EDW sono spesso basati su template esterni, mentre le macro sono supportate praticamente ovunque siano trattate delle stringhe: programmi, file di configurazione, file XML, eccetera. Il meccanismo di sostituzione macro, come si può immaginare, è un **meccanismo aperto**, cioè può essere esteso con nuove macro, anche dinamicamente.
- ◆ Uso di **scripting** per rendere flessibili e personalizzabili parti di un'applicazione. EDW fa abbondante uso di scripting in diverse aree:
 - ◆ **Programmi:** i programmi, che rappresentano il modo di usare le funzionalità ETL di EDW, sono script che usano classi interne a EDW o anche personalizzate.
 - ◆ **GUI trigger:** si tratta dell'equivalente lato client dei trigger disponibili nei database SQL. I GUI trigger sono innescati automaticamente al verificarsi di determinati eventi durante il dat-entry in EDW. Il set di tipi di trigger disponibili è aperto; un tipo particolare di trigger incapsula uno script e consente quindi massima libertà di estensione.
 - ◆ **File di configurazione:** i file di configurazione di EDW sono script anch'essi, che vengono eseguiti al primo accesso. Tali script sono costituiti principalmente da istruzioni che impostano valori in un oggetto **Configuration** a cui ricevono in input un riferimento. Questo consente di:
 - ◆ Mettere della **logica** (ad esempio controllo di flusso) nei file di configurazione.
 - ◆ Sfruttare il meccanismo degli **"include"** per aumentare la coerenza dei file di configurazione (ad esempio avere un file separato per ogni oggetti da configurare) e allo stesso tempo ridurre le ridondanze.



- ◆ **Data pipe:** si tratta di elementi che concorrono alle attività di trasformazione dati del modulo ETL di EDW; ciascun data pipe ha in input un **dato** (un'entità che ha un nome, un tipo e un valore) e lo può trasformare in un dato di altro tipo e/o valore. Come nel caso dei GUI trigger, esiste un gran numero di data pipe predefiniti, ma è possibile **aggiungerne** di personalizzati. Uno dei tipi di data pipe che si possono aggiungere è appunto uno script, che può quindi applicare trasformazioni impossibili da realizzare con un approccio basato su regole o tabelle.

5.1 Appendice: come progettare meccanismi estensibili

La possibilità di estendere le funzionalità predefinite, di cui ho appena parlato a proposito di GUI trigger, data pipe e macro (e che comunque si applica anche in molte altre circostanze in EDW), è un esempio di rispetto del principio Open/Closed, nonché di applicazione delle tecniche di contare "0, 1, ∞" e programmare rispetto ad interfacce e non implementazioni. Per creare infrastrutture estensibili come queste la ricetta è:

- ◆ Definire una **classe base** (di solito astratta) o un'**interfaccia** che l'applicazione usa.
- ◆ Definire un **registro**: una classe che mantenga un elenco di oggetti o classi registrate; la scelta tra registrare classi o oggetti già istanziati dipende dai casi. Di norma il registro è un **singleton** (cioè ne esiste solo un'istanza, accessibile globalmente, per tutto il ciclo di vita del programma). Il registro consente di registrare e de-registrare nuove classi, e di accedere all'elenco di classi registrate.
- ◆ Definire una **factory**: una classe in grado di istanziare oggetti registrati sulla base di nomi o altri mezzi. Anche la factory è molto spesso un singleton. La sua interfaccia contiene di norma un solo metodo, che serve a **istanziare gli oggetti** (o ottenere riferimenti agli oggetti o alle classi registrate). Factory e registro, per semplicità, possono essere inglobati nello stesso oggetto. Personalmente, io preferisco tenerli separati per maggiore pulizia (anche se ben difficilmente si userà una delle due classi senza l'altra).
- ◆ Definire, se ce ne sono, le **classi predefinite** o altre **classi base**.
- ◆ L'estensione avviene aggiungendo e registrando nuove classi. La registrazione di una nuova classe, normalmente, avviene chiamando il registro nelle sezioni **initialization** (registrazione) e **finalization** (de-registrazione) della unit che definisce la classe. Questo approccio funziona bene anche in caso di caricamento dinamico (per le unit che fanno parte di package).

6 Conclusione

Si potrebbe discutere di tecniche come quelle sopra per ore, e si tratterebbe di una discussione interessante, ma purtroppo il tempo a disposizione è quello che è. Spero con questa presentazione di aver suscitato curiosità e aver portato qualcuno in prossimità della strada del software estensibile e di qualità. Sarebbe un lavoro utile e una bella soddisfazione.