I *design pattern* nella progettazione di software per il supporto alla statistica ufficiale

di Francesco Altarocca[*]

1. Introduzione

Nelle attività necessarie alla produzione del dato statistico sono spesso impiegate tecniche e metodologie introdotte in ambiti diversi da quello statistico. Le elaborazioni, le tecniche di acquisizione di dati, le moderne metodologie di correzione dei dati, i meccanismi per l'aggregazione dei dati, ecc., utilizzano modelli, tecnologie e idee sempre più sofisticati.

Si riescono ad ottenere nuove "informazioni" aumentando il grado di complicazione delle strutture che "modellano" o rappresentano la realtà. Questo processo di miglioramento, in continua evoluzione, rende possibile l'uso di strutture con un grado di astrazione maggiore. Le nuove strutture "complesse", grazie alle entità più generali, sono in grado di "catturare" ulteriori aspetti collegati ad un determinato fenomeno che altrimenti, nei modelli più semplici, sarebbero trascurati.

Nasce, pertanto, l'esigenza di razionalizzare e di "semplificare" il più possibile strutture e programmi sempre più complicati.

Negli ultimi anni, grazie proprio al maggior grado di astrazione cui è capace, si è diffuso un paradigma di programmazione e di progettazione denominato *Object Oriented*. Grazie ad esso, possono essere trascurati alcuni aspetti legati alle macchine fisiche ed è possibile pensare ai programmi come strutture aderenti alla realtà d'interesse, piuttosto che alla rappresentazione e ai meccanismi propri delle macchine fisiche. In tal modo è possibile indirizzare maggiore attenzione sul

---

[*] Collaboratore tecnico enti di ricerca, Istat (DPTS - DCSS SIP/A), fraltaro@istat.it.

problema, piuttosto che su aspetti implementativi, agevolando in particolare la realizzazione dei progetti più complessi.

Più recentemente, soprattutto nelle istituzioni che si occupano di ricerca, sta facendo le sue prime apparizioni un altro paradigma, l'*Aspect Oriented*, che affianca, completa e migliora il modello *Object Oriented*. Questo nuovo paradigma consente di inserire funzionalità trasversali in un programma, come ad esempio il log degli eventi, utilizzando meccanismi appositi e producendo un impatto sul codice estremamente limitato e circoscritto. Ne consegue una maggiore pulizia dei sorgenti, una razionalità di livello superiore e una maggiore rapidità di intervento per la modifica delle soluzioni già disponibili.

Date queste premesse, risulta utile prestare particolare attenzione alle fasi di progettazione del sistema e di "disegno" delle strutture dati necessarie a modellare un problema.

La progettazione è un'attività estremamente complessa e critica per la buona riuscita di un progetto. Una buona progettazione è un requisito indispensabile per garantire ad una soluzione una sufficiente longevità. Nello svolgimento di questa attività sono necessari anche elementi, quali una buona capacità di astrazione e creatività che, tuttavia, da soli non sono sufficienti a garantire la bontà di una soluzione, soprattutto quando il problema da "modellare" è complesso. Oltre a ciò, servono ulteriori ingredienti acquisibili unicamente attraverso l'esperienza e un costante impegno nello studio e nella produzione di modelli per la progettazione. Inoltre, scelte meno restrittive possono in prima istanza risultare peggiori rispetto a quelle che, ad esempio, ottengono risultati più efficienti utilizzando ulteriori ipotesi vere in un particolare istante del ciclo di vita di un software.

Risulta utile, pertanto, cercare di far tesoro delle altrui esperienze e di utilizzare tecniche e gli "schemi" migliori perchè risultati più adatti ai cambiamenti che inevitabilmente si verificano durante il ciclo di vita di un software.

Studiare le esperienze e utilizzare i design pattern non può colmare la modesta esperienza di un progettista, ma risulta un aiuto rilevante per evitare di commettere

gravi errori e per avere un supporto, una sorta di prontuario, per la risoluzione dei problemi di progettazione che ricorrono frequentemente.

2. Design pattern

I design pattern, sono stati introdotti, per quel che concerne la progettazione di sistemi informatici, con la pubblicazione nel 1995 del testo: *"Design Patterns: Elements of Reusable Object-Oriented Software"* (Design Patterns: elementi per il riuso di software ad oggetti).

I quattro ricercatori (Gamma, Helm, Johnson e Vlissides), pionieri di questa metodologia e autori del testo appena citato, descrivono soluzioni semplici, generiche ed eleganti per problemi specifici nel contesto della progettazione ad oggetti. L'uso di questa metodologia permette di raggiungere diversi obiettivi, tutti ugualmente interessanti, tra i quali l'impiego di soluzioni e modelli consolidati, il riuso del codice e la scrittura di codice più generale, elegante e riusabile.

> *"Progettare software ad oggetti è difficile; riuscire a renderlo riusabile è ancora più difficile. Occorre trovare gli oggetti giusti, fattorizzarli in classi con giusta granularità, definire interfacce e gerarchie d'ereditarietà e stabilire fra queste le relazioni fondamentali. La progettazione dovrebbe essere specifica per il problema che si sta affrontando, ma anche sufficientemente generica per problemi e requisiti futuri"* (Gamma *et al.*, 2002).

La seguente definizione riassume in maniera adeguata l'essenza di un design pattern:

> *"Un design pattern attribuisce un nome ad un problema di progettazione, astrae e identifica gli aspetti principali di una struttura progettuale utile per la creazione di un progetto ad oggetti riusabile"* (Gamma *et al.*, 2002).

A supporto della validità dell'idea dei design pattern, che non trova impiego unicamente nella progettazione di sistemi software, viene riportata un'affermazione dell'architetto Christopher Alexander:

> *"Ogni pattern descrive un problema che si ripete più e più volte [...], descrive poi il nucleo della soluzione del problema, in modo tale che si possa usare la soluzione un milione di volte, senza mai applicarla nella stessa maniera".*(Alexander, 1977).

Nei capitoli successivi (dal 3 al 6) viene data una breve descrizione del contesto in cui si colloca ogni pattern presentato. Nel primo paragrafo vengono descritti, invece, lo scopo e la formulazione originale del pattern (Gamma *et al.*, 2002), mentre nei paragrafi successivi sono illustrate alcune applicazioni dei design pattern nel contesto del software per il data entry on line.

2.1. Un esempio di modellazione utilizzando i design pattern

Per riuscire a cogliere concretamente l'utilità di design pattern, verranno introdotte alcune strutture dati e classi utilizzati in una applicazione in via di sviluppo per la generazione di questionari on line (Altarocca, Vaccari 2005). L'esperienza trae origine dall'esigenza di realizzare uno strumento per la raccolta dei dati direttamente dal Web. Data la complessità e la genericità della soluzione è stato speso molto tempo nelle fasi di raccolta dei requisiti e di progettazione del software. In seguito verranno semplificati alcuni aspetti per permettere al lettore di focalizzare l'attenzione sugli elementi utili per le successive trattazioni.

L'applicazione parte da una specifica di un questionario in XML e costruisce "al volo", attraverso alcuni meccanismi che saranno spiegati successivamente, un questionario on line, cioè crea dinamicamente la pagina HTML da presentare all'utente. E' necessario pertanto entrare nel merito della struttura delle classi necessarie all'applicazione.

Un questionario è rappresentato da una oggetto della classe *Questionnaire* (in seguito verranno date brevi descrizioni degli elementi[1] che compongono un questionario).

- **Questionnaire**: cattura gli aspetti relativi all'intero questionario;
- **Page**: i questionari sono composti da pagine (tipicamente divise in base alla tipologia delle informazioni richieste) che seguono un determinato percorso logico;
- **Records**: è responsabile delle operazioni di estrazione dei dati da una base di dati e del loro aggiornamento/inserimento;
- **QElement (Questionnaire Element)**: è la superclasse per tutti gli oggetti contenuti in una pagina. Le sottoclassi sono divise in base alle caratteristiche degli elementi (Figura 3.2):
    - **FieldEl**: questa classe, insieme con le sue derivate (*TxtFieldEl*, *RadioFieldEl*), rappresenta gli elementi che contengono un dato, il quale può essere mostrato all'utente e successivamente modificato. In un'ottica più vicina al dominio del problema, un *FieldEl* rappresenta un campo generico, inteso in senso statistico, del questionario;
    - **SectionEl**: costituisce una sezione di un questionario e può, di conseguenza, contenere altri elementi logicamente correlati;
    - **DynamicEl**: è un elemento proprio dei questionari elettronici. Il compito è quello di catturare gli aspetti dinamici della

---

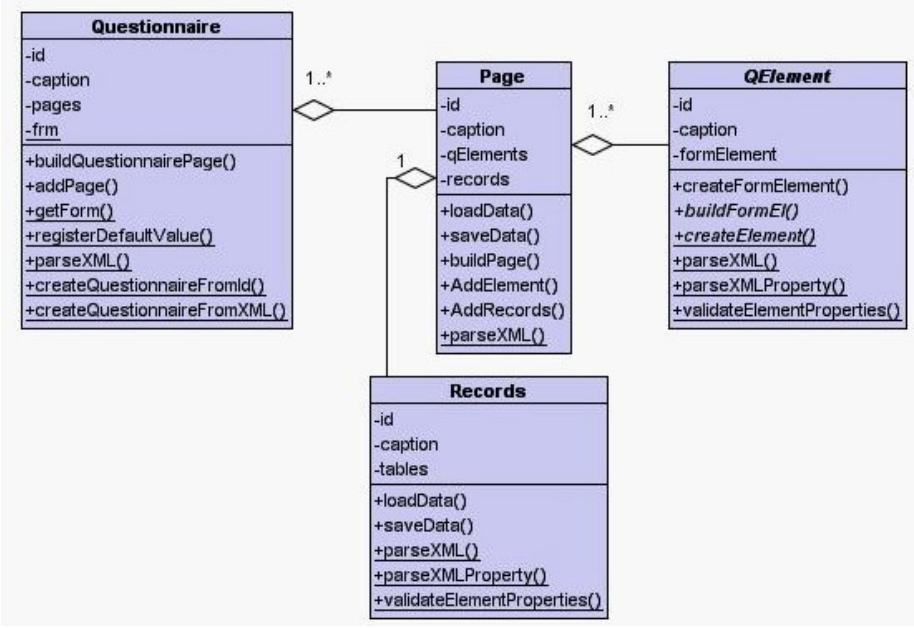
[1] Gli elementi (classi) che vengono presentati possono essere implementati per mezzo di un qualsiasi linguaggio di programmazione ad oggetti, come ad esempio, Java, PHP 5, C++, ecc.. Alcuni linguaggi di programmazione possono avere strumenti diversi come, ad esempio, l'ereditarietà multipla del C++, ma nella pratica non ci sono particolari differenze e gli esempi possono comunque essere modificati per soddisfare le peculiarità di ogni linguaggio di programmazione ad oggetti.

visualizzazione delle informazioni come,ad esempio, un elemento che visualizza la somma di una colonna;

- **StaticEl**: è l'elemento che rappresenta tutti gli oggetti di un questionario che non hanno rilevanza ai fini della raccolta del microdato, ma sono solo di supporto e statici (note, informazioni supplementari, collegamenti ipertestuali, ecc.);
- **DecoratedEl**: tale elemento, anch'esso proprio dei questionari elettronici, ha lo scopo di aggiungere funzionalità non previste nella fase di progettazione mediante l'impiego del design pattern *Decorator*. Ciò consentirà di aggiungere funzionalità o responsabilità speciali, non solo di carattere grafico, come l'aggiunta di codice Javascript, un colore o un font particolari, ad un qualsiasi elemento, inserendolo semplicemente in una o più buste di sottoclassi di *DecoratedEl*.

Nella Figura 2 sono visibili le classi *Questionnaire*, *Page*, *QElements*, *Records* e le loro relazioni. Si rimanda ai paragrafi successivi per una descrizione di alcuni metodi e delle proprietà di tali classi.

Figura 2 - Diagramma UML delle classi Questionnaire, Page, QElement e Records



3. Composite e la struttura degli elementi del questionario QElement

Il primo design pattern illustrato è il *Composite* che è stato utilizzato per la rappresentazione degli elementi contenuti in un questionario.

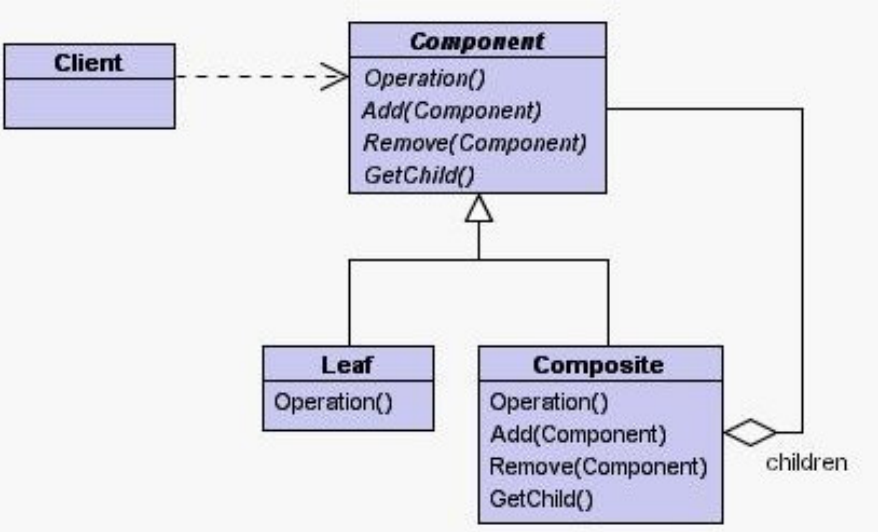
Molte applicazioni permettono di costruire elementi complessi aggregando più "componenti" elementari. Un generico programma potrebbe, quindi, modellare la struttura necessaria a rappresentare la realtà di interesse, introducendo classi primitive e classi aggiuntive per raggruppare elementi primitivi. Questo tipo di rappresentazione pone, però, un problema per chi deve utilizzare gli elementi primitivi e quelli composti. In particolare, è necessario trattare in modo differente le due tipologie di elementi introducendo nell'applicazione un certo grado di complessità. *Composite* permette ai client di trattare gli elementi in maniera omogenea e di comporre i diversi elementi ricorsivamente.

3.1. Composite

Lo scopo del design pattern *Composite* è quello di:

"Comporre oggetti in strutture ad albero per rappresentare gerarchie parte-tutto e consentire ai client di trattare oggetti singoli e composizioni in modo uniforme" (Gamma *et al.*, 2002, p. 163).

Figura 3.1 - Diagramma UML delle classi di Composite



La classe astratta *Component* (Figura 3.1) è l'elemento fondamentale della struttura e rappresenta sia un oggetto elementare sia loro collezioni.

Component dichiara operazioni per oggetti elementari (*Leaf*) e per le collezioni (*Composite*). Naturalmente le operazioni per le collezioni (*Add*, *Remove*, …) non saranno implementate per gli oggetti semplici.

I vantaggi derivanti dall'adozione di *Composite* sono:

- gli oggetti predefiniti possono essere messi insieme per formare oggetti più complessi. Questi ultimi possono, a loro volta, far parte di un altro oggetto complesso in modo ricorsivo;
- la scrittura dei client che utilizzano gli elementi è più semplice perchè non c'è bisogno di discriminare in base al tipo di oggetto;
- l'aggiunta di nuovi tipi di oggetti è semplificata. Le nuove classi potranno essere utilizzate senza apportare modifiche al client.

D'altra parte la genericità introdotta con *Composite* porta con sè anche alcuni svantaggi. A titolo di esempio, si pensi alla classe *Page* (della Figura 2) come ad un oggetto *SectionEl* (fare riferimento al paragrafo successivo e alla Figura 3.2) con alcune caratteristiche aggiuntive, essendo essenzialmente un contenitore di *QElement*. Questo modo di modellare il problema determina due problemi:

1. un oggetto *Page* non dove contenere un altro oggetto *Page*;
2. un oggetto di tipo *Questionnaire* può contenere esclusivamente oggetti di tipo *Page*.

Tali vincoli sono risolti, infatti, ponendo *Page* al di fuori della gerarchia *QElement* (Figura 2), perchè non possono essere gestiti dal sistema dei tipi statico[2] e dovrebbero, quindi, essere gestiti da codice ad hoc, diminuendo l'armonia della soluzione prospettata.

3.2. La gerarchia QElement

La gerarchia degli elementi del questionario abbraccia proprio la filosofia di *Composite*. Si evidenzia subito un'analogia tra la struttura del diagramma UML relativo a *Composite* (Figura 3.1) e quella della gerarchia *QElement* (Figura 3.2). La Tabella 3.1 elenca le corrispondenze dei due diagrammi.

Tabella 3.1 - Corrispondenza delle gerarchie Composite e QElement

Design pattern **Composite**	Struttura **QElement**
Client	Page
Component	QElement
Leaf	TxtFieldEl, RadioFieldEl, ...
Composite	SectionEl

Nel Listato 3 è riportato il codice della funzione *buildFormEl()* della classe *SectionEl*, necessario alla generazione degli elementi del questionario. È importante

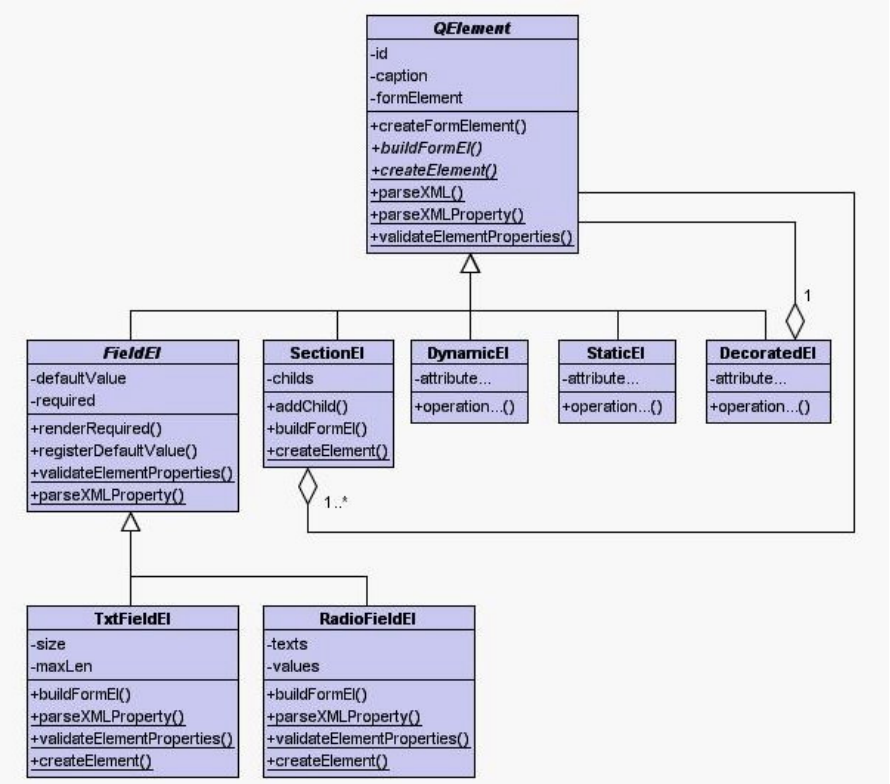
[2] Si dice che un linguaggio di programmazione ha un *sistema dei tipi* se esistono delle regole che assegnano un tipo agli elementi del linguaggio stesso. Un sistema dei tipi è detto *statico* se l'assegnazione dei tipi agli elementi del linguaggio avviene a tempo di compilazione.

notare le classi che hanno bisogno di dialogare con quelle della gerarchia *Qelement* non tengono conto del tipo di elemento con cui interagiscono. Infatti, nella riga 7, quando l'istanza dell'elemento *SectionEl* tenta di creare gli oggetti necessari alla visualizzazione della pagina, richiama su ogni componente in esso contenuto il metodo *buildFormEl()* indipendentemente dal fatto che sia o meno un elemento primitivo o composto.

Listato 3 - Metodo buildFormEl() della classe SectionEl

```
1  function buildFormEl(){
2      [...]
3      $aux[] = $this;
4
5      //crea i figli
6      foreach($this->childs as $Qel){
7          $aux[] = $Qel->buildFormEl();
8      }
9     return $aux;
10 }
```

Figura 3.2 - Diagramma UML delle classi della gerarchia QElement



4. Template method e il parsing dell'XML

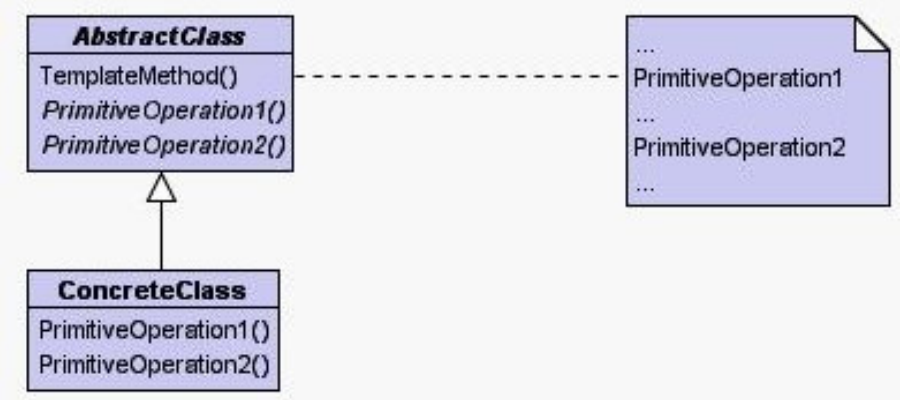
Capita di frequente di avere un algoritmo generale che esegue una serie di operazioni su determinati oggetti di una gerarchia di classi e che cambia alcuni dettagli a seconda dell'"oggetto" trattato in un particolare momento. Un buon suggerimento è quella di scrivere lo "scheletro" dell'algoritmo nella superclasse e le differenze, determinate dalla tipologia di elemento, nelle sottoclassi derivate. Nel caso in cui debbano essere aggiunti nuovi elementi alla gerarchia è sufficiente specificare, qualora fosse necessario, solo alcuni dettagli indispensabili alle nuove sottoclassi. Inoltre, un intervento sul codice dell'algoritmo generale si ripercuote su tutte le classi (a causa dell'ereditarietà), evitando il rischio di produrre trattamenti differenti per i diversi tipi di elementi.

4.1. Template method

L'obiettivo di questo design pattern è descritto come:

> *"Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi dell'algoritmo alle sottoclassi.* Template method *lascia che le sottoclassi ridefiniscano alcuni passi dell'algoritmo senza dover implementare di nuovo la struttura dell'algoritmo stesso"* (Gamma *et al.*, 2002, p. 327).

Figura 4.1 - Diagramma UML delle classi di Template method



La classe astratta *AbstractClass* definisce:

- un metodo concreto *TemplateMethod()* che cattura gli aspetti invarianti di un algoritmo;
- dei metodi astratti *PrimitiveOperation#()* che andranno implementati e personalizzati nelle sottoclassi concrete.

È importante notare come nel metodo *TemplateMethod()* (vedere commento della Figura 4.1) viene catturata l'"essenza" dell'algoritmo, specializzato, in seguito, dalle

implementazioni delle sottoclassi che vengono richiamate dal meccanismo di binding dinamico dei metodi[3].

Le classi derivate da *AbstractClass* ridefiniscono esclusivamente i metodi *PrimitiveOperation#()* per adattare alcune parti dell'algoritmo al caso specifico. Ne deriva che l'utilizzo di questo pattern permette di:

- implementare lo "schema" di un algoritmo un'unica volta e lasciare alle sottoclassi i dettagli dei casi particolari. La modifica di piccoli dettagli dell'algoritmo generale avrà, pertanto, un impatto limitato al solo metodo *TemplateMethod()* della classe astratta;
- portare a fattor comune alcuni comportamenti delle sottoclassi in modo da evitare la duplicazione di codice.

Una importante conseguenza dell'adozione di questo pattern è che possono essere definiti dei metodi *hook* (Gamma *et al.*, 2002, p. 328), che forniscono un comportamento standard per le *PrimitiveOperation#()* che le sottoclassi, se necessario, possono specializzare.

4.2. Il metodo parseXML()

I documenti che rappresentano i questionari e gli elementi in essi contenuti sono rappresentati mediante codice in formato XML. Esiste una stretta correlazione tra gli elementi definiti dalle classi e quelli definiti nel documento XML. È necessario,

---

[3] Nel *binding dinamico dei metodi* l'associazione (binding) dell'oggetto al metodo di eseguire avviene a tempo di esecuzione (dinamico). Questo meccanismo dei linguaggi di programmazione ad oggetti permette ad un oggetto di eseguire il metodo della sua classe anche se è stato dichiarato come appartenente ad una superclasse. In Java il binding dei metodi è dinamico, mentre quello per le variabili è statico.

però, trasformare questo formato in una struttura adeguata alla generazione e all'elaborazione delle informazioni per mezzo di un linguaggio di programmazione. Il meccanismo di trasformazione, da una struttura XML a oggetti di classi costruite per questa applicazione, prende spunto dal design pattern *Template method*.

La responsabilità di operare la trasformazione da documento XML a struttura di oggetti è incapsulata nel metodo *parseXML()* di ogni classe (*Questionnaire*, *Page*, *TxtFieldEl*, ecc.). Ogni oggetto gestisce la porzione XML di propria competenza, delegando alle classi degli oggetti di cui è composto, il compito di creare la sottostruttura. Questa metodo deve essere necessariamente statico; infatti non esiste ancora un'istanza di un particolare oggetto perchè lo si sta creando. Riassumendo, il metodo *parseXML()* ha le seguenti responsabilità:

1. "gestire" la porzione di documento XML di propria competenza;
2. creare un oggetto della propria classe. Quanto detto vale in generale per tutte le classi; la classe *QElement*, invece, definisce un metodo statico *parseXML()* che svolge il ruolo di *TemplateMethod().*

Esso prende come parametro una rappresentazione ad albero[4] del codice XML di un particolare elemento del questionario.

parseXML() si occupa di:

1. chiamare i metodi *parseXMLProperty()* su ogni figlio del nodo dell'albero che è stato passato. Simili metodi si occupano sostanzialmente di raccogliere le proprietà di un elemento in un array chiamato *$properties*;
2. chiamare i metodi *validateElementProperties().* I compiti di questi metodi sono: verificare che tutte le proprietà necessarie siano nell'array *$properties* e valorizzare le proprietà opzionali ai valori di default;

---

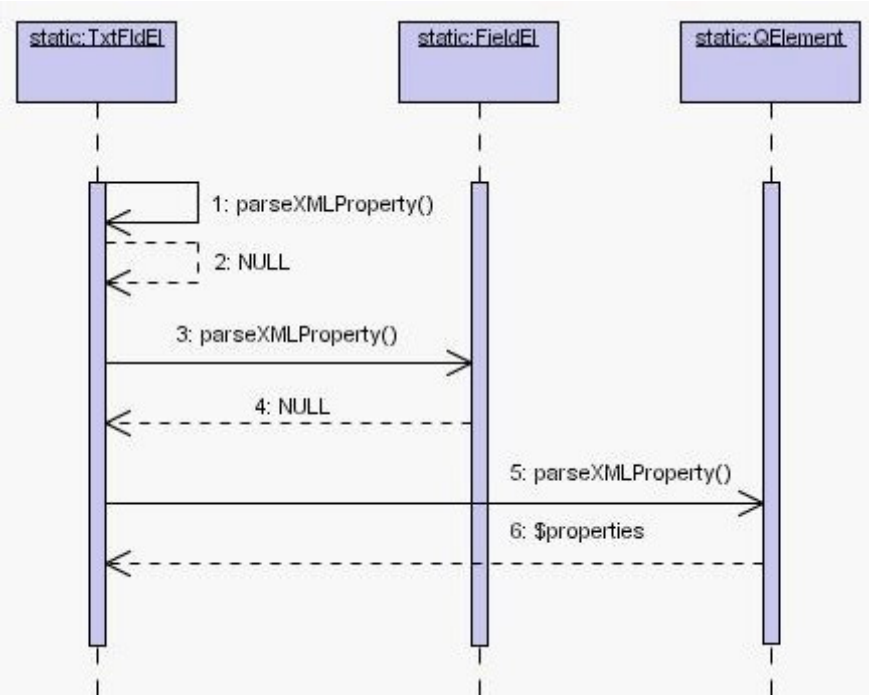
[4] Un qualsiasi documento XML può essere trasformato, in maniera biunivoca, in una struttura ad albero contenente le stesse informazioni. La libreria *XML_Tree* del progetto *PEAR* disponibile per il linguaggio di programmazione *PHP* fornisce una simile funzione.

3. chiamare il metodo *createElement()* e ritornare l'elemento appena creato.

I metodi descritti sono tutti statici; essi vengono appositamente invocati per creare un oggetto derivato da *QElement*. Una caratteristica rilevante dei primi due è che vengono richiamati dal metodo *TemplateMethod()* in modo gerarchico. Nel caso di *parseXMLProperty()*, si risale la gerarchia delle classi finché non viene trovato il metodo capace di gestire la proprietà (se nessuno di essi riesce a gestirla viene restituito il valore NULL), mentre, per quanto riguarda *validateElementProperties()*, la gerarchia viene percorsa tutta perché ogni superclasse contribuisce a validare alcune proprietà. Si è preferito operare in questo modo, piuttosto che chiamare in ogni metodo lo stesso della classe genitore, per mettere a fattor comune il codice. "Scolpire" nel *TemplateMethod()* questo comportamento evita la duplicazione di codice e la possibilità di tralasciare la chiamata al metodo del genitore.

Supponiamo di avere, nella specifica XML, un elemento *TxtFieldEl* e di voler elaborare la proprietà *id*. È interessante vedere (Figura 4.2) il meccanismo di funzionamento del metodo statico *parseXML()* (ereditato da *QElement*) della classe *TxtFieldEl*. Questo metodo tenta di elaborare la proprietà, attraverso *parseXMLProperty(),* partendo dal metodo dalla classe stessa (*TxtFieldEl*). Se questa non è in grado di gestirla, allora sarà lo stesso metodo della classe di livello superiore (*FieldEl*) a tentare di elaborarla, in maniera ricorsiva. Questo meccanismo viene iterato fin quando non viene trovato, nella catena gerarchica, un metodo capace di gestire la richiesta; se nessuno riesce a farlo, arrivati alla radice viene generata un'eccezione. Nel passo 2 del metodo *parseXML()*, necessario per la validazione delle proprietà dell'oggetto, la gerarchia viene visitata per intero perchè ogni classe valida i propri attributi.

Figura 4.2 – Diagramma UML di sequenza dell'esecuzione di una porzione del metodo parseXML()



5. Factory Method e la generazione delle pagine

Nell'attività di scrittura di codice e classi generiche sorge di frequente l'esigenza di dover istanziare oggetti, appartenenti ad una gerarchia di classi ereditate da una superclasse astratta, di cui non si conosce la classe di appartenenza. Questa complicazione deriva dalla necessità di astrarre il più possibile dalla particolare implementazione e per consentire, eventualmente, di estendere una classe astratta per soddisfare i propri bisogni senza la necessità di dover modificare il codice che ne fa uso. Se il meccanismo di creazione dell'oggetto è spostato da un "punto" centrale alle classi concrete, il problema viene risolto.

Procedendo in questa maniera si riescono ad ottenere soluzioni eleganti e generiche. Infatti, non c'è la necessità di sapere quale elemento verrà creato, ma solo che c'è bisogno di crearne uno.

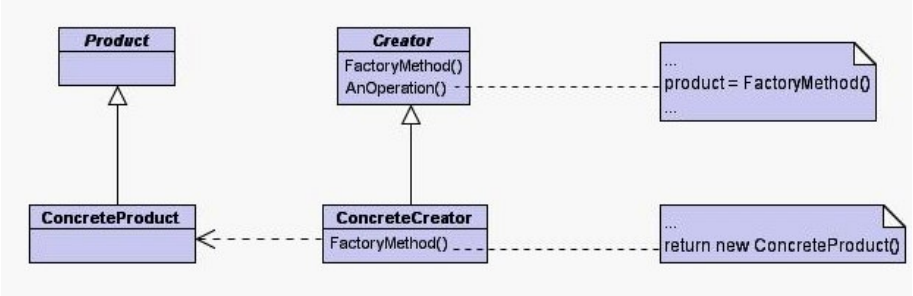
5.1. Factory Method

Lo scopo di questo design pattern, secondo quanto riportato in (Gamma *et al.*, 2002, p. 107), è il seguente:

> *"Definisce un'interfaccia per la creazione di un oggetto, lasciando alle sottoclassi la decisione sulla classe che deve essere istanziata. Il pattern* Factory Method *consente di delegare l'istanziazione di una classe alle sottoclassi".*

Il diagramma UML di *Factory Method* è illustrato nella Figura 5.

Figura 5 - Diagramma UML delle classi di Factory Method



L'istanziazione di un oggetto specifico è un compito delle sottoclassi concrete di *Creator* e sarà svolto proprio dal metodo *FactoryMethod().* L'utilizzo di questo design pattern porta numerosi benefici soprattutto in ambiti più astratti, come ad esempio nei framework, perchè elimina la necessità di inserire riferimenti a classi della particolare applicazione che ne fa uso. Inoltre, i metodi deputati alla creazione delle istanze necessarie, risultano essere molto compatti e facilitando così l'inserimento di nuove classi nella gerarchia. Nel caso specifico del software di data entry, il numero delle classi che compongono la gerarchia, viene di volta in volta incrementato con classi altamente specializzate.

Il metodo *createFormElement()* (corrispondente al metodo *AnOperatrion()* della Figura 5) è ereditato dalle sottoclassi di *QElement*. La classe *Page* chiama il metodo *buildFormElement()* di tutti gli elementi che contiene (Listato 5, righe da 7 a 9). Il metodo *buildFormElement()* si occupa di "disegnare" l'oggetto sul quale è stato invocato, utilizzando le classi messe a disposizione dalla libreria *HTML_QuickForm* di *PEAR*. Questo metodo appartiene alla classe concreta che è in grado di stabilire quale elemento deve essere creato. Nel frammento di codice sottostante è riportato, come esempio, la parte del metodo *buildFormEl()* della classe *TxtFieldEl* responsabile della creazione dell'oggetto concreto.

```
$tmp =& HTML_QuickForm::createElement('text', $this->getId(),
$this->getCaption(true));
```

Il metodo costruisce un oggetto della libreria *HTML_QuickForm*, nel caso specifico un oggetto di tipo *'text'* (che la libreria a sua volta disegnerà, in HTML, come un campo *input* di tipo *text)*. È interessante notare come il design pattern *Factory Method* sia spesso utilizzato insieme con il pattern *Template Method* presentato in precedenza.

In questo caso il metodo *createFormElement()* assume il ruolo di *TemplateMethod()*, mentre il metodo *buildFormEl()* quello di *FactoryMethod()*.

5.2. La creazione degli elementi della form

Attraverso i metodi analizzati in precedenza è stata creata una struttura composta da un oggetto di tipo *Questionnaire*, un oggetto di tipo *Page* e una serie di oggetti di sottoclassi di *QElement*. Per visualizzare il questionario è sufficiente invocare il metodo *buildQuestionnairePage()* sull'oggetto *Questionnaire*. Anche in questo caso

il compito di eseguire una specifica attività viene distribuito fra le classi che compongono la struttura del questionario.

Il metodo *buildQuestionnairePage()* crea una nuova pagina, impostare alcune caratteristiche del questionario, richiama il metodo *renderPage()*, sull'oggetto *Page*, responsabile della creazione degli elementi contenuti nella pagina del questionario, recupera i dati dalle fonti esterne, ecc.

Il metodo render*Page()* (Listato 5) oltre ad inserisce il titolo della pagina, richiama il metodo *createFormElement()* su tutti i figli. Quest'ultimo metodo è il *FactoryMethod()* dell'omonimo design pattern.

Listato 5 – Metodo renderPage() della classe Page

```
1  function renderPage(){
2      if ($this->caption != ""){
3          $frm = Questionnaire::getForm();
4          $frm->addElement('html', "<H3>" .
$this->getCaption(true) . "</H3>");
5      }
6
7      foreach($this->qElements as $Qel){
8          $Qel->buildFormElement();
9      }
10 }
```

6. Il pattern Singleton, la generazione e la registrazione degli identificativi

Il Singleton è uno dei più semplici, e di conseguenza, uno dei più usati, design pattern creazionali[5]. È difficile trovare un sistema o un software che non includa, in qualche modo, il concetto espresso da questo design pattern.

L'idea è quella di garantire l'esistenza di una unica istanza di un particolare oggetto e contestualmente un unico punto di accesso a questa istanza in modo da controllarne l'accesso. Tutto ciò è reso possibile grazie ai meccanismi messi a disposizione dai linguaggi di programmazione ad oggetti.

Un impiego particolarmente importante di questo design pattern è quello che permette di evitare l'uso di variabili globali. Il ricorso a variabili globali, infatti, è considerato un metodo poco elegante di condivisione delle informazioni fra le varie parti di un software.

6.1. Singleton

In (Gamma *et al.*, 2002, p. 127) lo scopo del pattern viene descritto nel seguente modo:

> *"Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale a tale istanza"*.

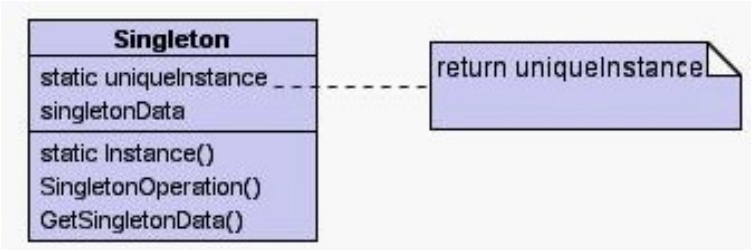
In molti sistemi è presente l'esigenza di avere una sola istanza di un particolare oggetto, accessibile da altri oggetti che ne dovranno farne uso. La creazione dell'unica istanza da parte delle classi che la utilizzano non risulta essere una buona strategia. Sarà, quindi, la classe stessa o una particolare classe "delegata" a preoccuparsi di creare al più un oggetto e di restituirlo quando richiesto. Questa classe è l'unica in grado di controllare l'esistenza della istanza e, qualora non sia

---

[5] I pattern creazionali, in sintesi, forniscono un modo per astrarre il processo di creazione di un oggetto. Tale astrazione è propedeutica al disaccoppiamento dell'oggetto che crea una nuova istanza e l'istanza stessa.

presente, di crearla. Nella Figura 6 è rappresentato il diagramma delle classi di questo design pattern. Nella nota a destra è visibile un frammento di codice del metodo statico *Instance()* il quale, prima di ritornare al richiedente l'istanza, deve verificarne l'esistenza.

Figura 6 - Diagramma UML delle classi di Singleton



I principali vantaggi derivanti dall'adozione di *Singleton* sono:

- accesso controllato. La classe *Singleton* mantiene il controllo su come e quando i client possono accedere all'unica istanza;
- evitare l'uso di variabili globali. L'uso di *Singleton* permette di far riferimento a variabili, che, altrimenti, dovrebbero essere globali, incapsulando, all'interno della variabile statica della classe, la singola istanza.

Nella sua formulazione originale, il design pattern, prevede che l'istanza restituita sia della classe stessa. Infatti, oltre alla variabile statica che contiene l'istanza ed al metodo necessario per recuperare l'oggetto, viene dichiarato il costruttore come *protected* per assicurare l'unicità.

Nel progetto, *Singleton* è stato usato in diversi situazioni e modi. Il primo utilizzo è quello necessario a garantire l'esistenza di un solo oggetto *HTML_QuickForm* e per recuperarlo in modo agevole da diverse classi. Non è stato possibile impiegare la formulazione classica perchè il costruttore di *HTML_QuickForm* è dichiarato pubblico. Sono stati inseriti una variabile statica *$frm* e un metodo statico *getForm()*. Quando un oggetto ha la necessità di accedere alla form, per esempio per aggiungere un elemento alla pagina, è sufficiente chiamare il metodo statico

Questionnaire::getForm(). Si evita, così, di passare alle funzioni o agli oggetti un riferimento esplicito alla form.

6.2. Il generatore di identificativi: la classe IdGen

Un altro impiego di *Singleton* è quello della generazione degli *id* univoci. Quando, nella specifica XML del questionario, non viene specificato l'identificativo di un elemento, allora il software ne crea uno univoco. L'identificativo generato è composto da un prefisso contenente il nome della classe specificata seguito da un contatore.

Facendo riferimento al Listato 6.1 è interessante notare l'implementazione del pattern in questa classe. Viene controllata l'esistenza di un determinato prefisso nell'array associativo *$prefixes*. Se il prefisso (o nome della classe) esiste, allora viene incrementato il contatore; viceversa, viene creata una nuova chiave nell'array *$prefixes*.

Listato 6.1 - La classe IdGen

```
1  class IdGen{
2      static private $prefixes = array();
3      static function getNewId($prefix = ""){
4          [...]
5          if (array_key_exists($prefix, IdGen::$prefixes)){
6              IdGen::$prefixes[$prefix]++;
7          } else{
8              IdGen::$prefixes[$prefix] = 1;
9          }
10         [...]
11         return $prefix . $under .
IdGen::$prefixes[$prefix];
12     }
```

```
13 }
```

Questo comportamento è assimilabile al design pattern *Singleton* perchè per ogni prefisso:

- viene dato un unico punto di accesso alla generazione degli identificativi;
- viene creato un nuovo "oggetto" (chiave nell'array associativo) in grado di mantenere memoria dell'ultimo identificativo utilizzato.

6.3. La registrazione degli id

Per evitare che, nel momento della creazione, due elementi abbiano lo stesso identificativo, essi registrano il proprio *id* chiamando il metodo statico *RegisterId::validateId()*. Il codice della classe *RegisterId* è riportato nel Listato 6.2. La classe *IdGen* genera un nuovo *id* se tale proprietà viene omessa nella definizione dell'elemento. Il ciclo (da riga 11 a 17) e la verifica della disponibilità dell'*id* (riga 13) sono necessari per evitare situazioni in cui la creazione automatica di un identificativo proponesse un nome già utilizzato in modo esplicito da un altro elemento. Se, viceversa, nella definizione del questionario l'*id* di un elemento è specificato, basta controllare che questo non sia stato usato in precedenza (riga 20).
Un'ulteriore complicazione potrebbe sorgere nel caso in cui un *id* generato automaticamente risulti identico ad uno definito successivamente. Tale situazione dovrebbe essere gestita raccogliendo tutti gli *id* espliciti nel codice XML e in un momento successivo generare tutti gli *id* omessi. Questa strategia, però, oltre ad essere difficilmente applicabile per la modalità con la quale vengono generati gli oggetti del questionario, risulta eccessiva in relazione alle finalità della presente trattazione.

Listato 6.2 – La classe RegisterId

```
1  class RegisterId{
2      static private $registeredIds = array();
3      const maxLoops = 10;
4
5      static function validateId($id, $class = "Object"){
6          $i = 1;
7          $validId = false;
8          $tmp = $id;
9
10         if (is_null($id)){//genero l'identificativo
11             while (!(($i >= RegisterId::maxLoops) or
$validId)){
12                 $tmp = IdGen::getNewId($class);
13                 if (!array_key_exists($tmp,
RegisterId::$registeredIds)){
14                     $validId = true;
15                 }
16                 $i++;
17             }
18         }
19
20         if (array_key_exists($tmp,
RegisterId::$registeredIds)){
21             throw new duplicatedIdException($tmp);
22         }else{
23             RegisterId::$registeredIds[$tmp] = 1;
24             return $tmp;
25         }
26     }
27 }
```

7. FlyWeight e *Decorator*

In seguito saranno brevemente introdotti altri due design pattern utilizzati per il progetto del software di data entry. Il primo permette di "snellire" una struttura contenente numerosi elementi simili per risparmiare risorse, il secondo consente di aggiungere funzionalità non ancora previste agli elementi inserendoli in una "busta".

7.1. Il pattern FlyWeight e le espressioni "costante"

> *"Utilizzare la condivisione per supportare in modo efficiente un gran numero di oggetti a granularità fine"* (Gamma *et al.*, 2002, p. 195).

Nella attività di astrazione di un problema o nella rappresentazione di un sistema si utilizza una struttura composta da oggetti "collegati" attraverso diversi meccanismi. La granularità degli oggetti che compongono questa struttura alle volte è molto fine e richiede un numero elevato di oggetti che in gran parte risultano "identici".

Un utilizzo condiviso degli oggetti con le stesse caratteristiche (chiamati *flyweight*) porta, in linea del tutto generale, al risparmio di spazio e, conseguentemente, alla riduzione del tempo di elaborazione. Questo design pattern può essere utilizzato quando:

- il numero degli oggetti gestiti da un'applicazione è molto alto;
- i costi per la memorizzazione degli oggetti sono elevati;
- molti oggetti (o alcune loro caratteristiche) possono essere sostituiti da un numero ridotto di oggetti condivisi.

Molte delle proprietà degli elementi dei questionari, vengono valorizzate per default con espressioni costanti come: *true*, *false*, *NULL*, ecc.. Si è pensato, quindi, di condividere tali oggetti per snellire, ottimizzare e accelerare l'attività di generazione delle pagine. I linguaggi di scripting, infatti, compiono numerose operazioni prima

di generare l'output, proprio perchè il codice viene interpretato[6] e non compilato staticamente.

Nella classe *PhpExp*, che nel progetto è deputata alla rappresentare delle espressioni, sono stati inseriti quattro metodi statici (*instanceNull(), instanceTrue(), instanceFalse() e instanceVoidString()*) e quattro variabili statiche (*FWNullExp, FWTrueExp, FWFalseExp e FWVoidExp*). Il loro compito, come evidenzia l'esempio del Listato 7, è quello di restituire un'istanza della costante, se disponibile, altrimenti di crearla. È interessante osservare come, anche se nel caso specifico non ha particolare rilevanza, l'uso congiunto dei design pattern *FlyWeight* e *Singleton* ottimizzi ulteriormente le risorse creando gli oggetti necessari se non sono ancora disponibili.

Listato 7 - Il metodo instanceTrue()

```
1  static function instanceTrue(){
2      if (is_null(ConstExp::$FWTrueExp)){
3          ConstExp::$FWTrueExp = new PhpExp(true);
4      }
5      //restituisco il FlyWeight
6      return ConstExp::$FWTrueExp;
7  }
```

7.2. Decorator

> *"Aggiungere dinamicamente responsabilità ad un oggetto. I decoratori forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità."* (Gamma *et al.*, 2002, p. 175)

L'uso di questo design pattern è indicato quando:

---

[6] Alcuni linguaggi di scripting prevedono l'uso di tecniche di caching che permettono di saltare qualche passaggio nell'elaborazione degli script.

- si vogliono aggiungere responsabilità ad un particolare elemento;
- si vogliono togliere alcune responsabilità ad un elemento perché non sono proprie dell'elemento o perché utili anche per altre gerarchie di classi;
- non è possibile estendere una classe o non si vogliono aggiungere determinate responsabilità a tutta la classe.

L'idea di *Decorator* è di racchiudere l'oggetto che si vuole "decorare" in un altro che si fa carico di aggiungere la nuova funzionalità. Un approccio di questo tipo permette di generare catene di oggetti che aggiungono funzionalità. Ad esempio, ad un oggetto grafico potrebbero essere "agganciati" un bordo, uno sfondo, ecc. (altro caso d'impiego di questo design pattern si trova nei flussi (Stream) delle API di Java). Pertanto, ad un qualsiasi flusso possono essere aggiunti buffer, codifiche, ecc..

Nel caso del software di data entry, *Decorator* è impiegato per aggiungere funzionalità di varia natura come, ad esempio, codice Javascript oppure caratteristiche di tipo grafico. Dalla Figura 3.2 è possibile capire come un qualsiasi elemento della gerarchia *QElement* possa essere inserito in una o più "buste" *DecoratedEl.* Nell'elaborazione di un elemento "decorato" si tolgono e si elaborano le buste (come avviene negli strati dello stack TCP/IP) fino a raggiungere un elemento della gerarchia *QElement* non appartenente al ramo dei *DecoratedEl*. Il sottoalbero che ha come radice *DecoratedEl* può essere esteso a piacimento per introdurre, in linea del tutto generale, nuove funzionalità non previste nella fase di progettazione.

8. Considerazioni sui design pattern

Questa breve panoramica dei design pattern impiegati nella progettazione del software di data entry ha la finalità di introdurre l'argomento e di cercare di trasmettere le potenzialità di questo strumento nell'attività di "disegno" di un

software. L'applicazione di queste idee conferisce alle soluzione molte caratteristiche interessanti come la solidità, la genericità, la riusabilità e la flessibilità. Molte volte gli sviluppatori e i progettisti sono inconsapevoli del fatto che stiano utilizzando i design pattern. Esistono diversi altri design pattern che vanno citati perché estremamente importanti: Proxy, Adapter, Iterator, Observer, Strategy e Visitor. Quando si elaborano soluzioni particolarmente generiche ed eleganti per un problema di progettazione, esse entrano a far parte dell'insieme dei design pattern.

Un elemento interessante che scaturisce dall'osservazione delle implementazioni qui proposte è la constatazione di come, in alcuni casi, i design pattern si compenetrino l'uno con l'altro. Una classe o un metodo può svolgere contemporaneamente ruoli diversi in design pattern distinti. Ad esempio, il metodo *buildFormEl()* svolge il ruolo di *FactoryMethod()* per un pattern (Figura 5) e il ruolo di PrimitiveOperation#() per il *Template method* (Figura 4.1).

Il metodo *createFormElement(),* invece, ricopre la funzione di *TemplateMethod()* nel design pattern *Template method* e di *AnOperation()* in *Factory Method*.

Alcune volte i pattern vengono utilizzati insieme ad altri. Ad esempio *Composite* e *Template method*, *Fly Weight* e *Singleton*.

I design pattern risultano estremamente efficaci quando si vogliono realizzare framework o soluzioni generiche a problemi particolari. Le classi *IdGen* e *RegisterId*, per esempio, possono essere impiegate in qualsiasi progetto che presenti la necessità di generare automaticamente identificativi univoci e di registrarli. Benefici più modesti, ma non per questo meno importanti, si ottengono applicando i design pattern ai casi concreti personalizzandoli e, quindi, rendendoli difficilmente riutilizzabili.

La letteratura relativa a questa metodologia, non offre solo soluzioni o consigli per risolvere problemi di progettazione. È indubbio, infatti, che le esperienze di altri autori che si sono occupati di questo argomento rappresentano una risorsa preziosa. Il loro lavoro permette di affrontare determinate questioni in maniera diretta

piuttosto che procedendo attraverso una serie di insuccessi, propedeutici allo sviluppo della capacità di inquadrare il problema dalla prospettiva adeguata.

Riferimenti bibliografici

Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., Angel S., *"A Pattern Language"*, Oxford University Press, New York, 1977.

Gamma E., Helm R., Johnson R., Vlissides J., *"Design Patterns: Elementi per il riuso di software a oggetti, prima versione italiana)*, Pearson Educational Italia, Milano, 2002.

Altarocca F., Vaccari C., "Fare Open Source all'Istat: il generatore di data entry per indagini statistiche", *Proceedings of the 1st International Conference on Open Source Systems, July 11-15, 2005 Genova, Italy* 2005.

Abstract

The use of sophisticated techniques in the statistics process production, such as on line data catching, data validation techniques, business intelligence systems integration, involves, among the others, computer science experts. It can happen that tools or frameworks can be easily integrated with our systems, while sometimes is necessary to rewrite some code's parts. As normal software aren't able to satisfy research's needs, raises the requirement to modify systems or to realize a brand new ones. You can often find the similar problems in designing software.

Design patterns name design problems, separate and identify main structures aspects useful to create a reusable object oriented solution. Design patterns allow already used structures to solve problems frequently faced in designing tasks.

We used a questionnaire generating system to show how that technique can be used. This paper deals with it.

Sommario

L'utilizzo di tecniche sempre più sofisticate, nel contesto del processo di produzione del dato statistico, come l'acquisizione di dati on line, le tecniche di correzione dei dati, l'integrazione di sistemi di business intelligence o di data mining, vede coinvolte più soggetti tra i quali importante è l'apporto degli esperti in informatica.

In alcuni casi l'integrazione di strumenti specifici o l'impiego di framework si realizza attraverso una semplice configurazione, mentre in altri è necessario intervenire sul codice per estendere le funzionalità di una soluzione. Inoltre, non sempre sono disponibili sistemi o software capaci di soddisfare le esigenze di un ricercatore per studiare un fenomeno. Si pone, pertanto, il problema di realizzare o modificare un sistema esistente. Nell'attività di progettazione di software emergono spesso problemi simili. I design pattern attribuiscono un nome ad un problema di progettazione, astraggono ed identificano gli aspetti principali di una struttura progettuale utile per la creazione di un progetto ad oggetti riusabile.

Come l'Open Source consente il riuso di soluzioni già codificate, i design pattern permettono il riutilizzo delle idee o delle strutture astratte realizzate per risolvere problemi che di frequente si presentano mentre si progetta un software.

In questo lavoro verranno introdotti alcuni design pattern e il loro impiego nell'ambito della progettazione ed implementazione di un sistema per la generazione automatica di questionari on line.