Francesco Napoleoni
matr. 66791/35

MacroPHPagus

Un framework per lo sviluppo di applicazioni orientate agli oggetti in PHP 5

Relatore
Giuseppe Di Battista

Tutor
Massimo Celidonio

Università degli studi ROMA TRE
Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
A. A. 2005/2006

Indice

Convenzioni usate nel testo

Le seguenti convenzioni sono utilizzate per l'evidenziazione di termini specifici:

nome file o directory: `/usr/local/bin`

nome marchio: *Microsoft Access*

nome variabile o macro: `$variabile_php`, `FLAGS`

oggetto software: Apparato

pattern: *Observer*

tipo di dato: *float*

URI: `http://www.sito.net/index.html`

I termini e le locuzioni stranieri non ricorrenti, ove presenti, sono indicati *in corsivo*.

Le porzioni di programmi e script in qualsiasi linguaggio sono riportate `in carattere a spaziatura fissa`.

Si ricorda che tutti i nomi di prodotti e tecnologie software e hardware citati in questo testo sono marchi registrati, di proprietà dei rispettivi creatori.

Ringraziamenti

Sebbene lo sviluppo effettivo di **MacroPHPagus** sia avvenuto in completa solitudine, desidero comunque ringraziare le seguenti persone per avermi supportato moralmente, oppure messo in condizioni di superare alcuni ostacoli tecnici e burocratici nel mio (lungo) cammino verso il termine dell'esperienza universitaria, o semplicemente per la loro disponibilità: *in primis* i miei genitori, Giuseppe di Battista, Massimo Celidonio, Lorenzo Pulcini, Consuelo Tuveri, il gruppo Comunicazioni Radio della Fondazione Ugo Bordoni, Maurizio Patrignani, Diego Basso, Nicola Ghezzani, Leonardo Costantini.

Capitolo 1

Introduzione

1.1 Panoramica sul progetto

MacroPHPagus è un generatore di codice PHP versione 5 basato su modelli macro: tali modelli sono strutturati in modo da costituire una forma embrionale di *metalinguaggio* per la rappresentazione di classi ed interfacce secondo il paradigma di programmazione orientata agli oggetti.

Nato come un semplice strumento di sviluppo nell'ambito della realizzazione di un software per conto della *Fondazione Ugo Bordoni*, **MacroPHPagus** si sta evolvendo come un *framework* a sé stante, riutilizzabile anche per altre applicazioni.

La generazione di codice non è comunque l'unico scopo di questo software. Infatti, durante l'evoluzione di questa idea, è apparso chiaro che svariate operazioni che normalmente lo sviluppatore di software effettua spesso a mano o aiutandosi con complessi e talvolta costosi strumenti, potevano essere automatizzate, facilitando così la gestione di applicazioni anche grandi.

Il ragionamento ha infine condotto all'individuazione dei seguenti obiettivi fondamentali:

- maggiore leggibilità e manutenibilità del codice PHP, attraverso l'uso di modelli preformattati;

- documentazione del codice;
- rispetto dei principali standard che si stanno affermando nella comunità PHP;
- applicazione di alcuni ben noti *pattern* di progettazione;
- controllo delle versioni;
- superamento di alcune incongruenze (ancora) insite nel linguaggio PHP;
- generazione automatica di *unit test*;
- gestione degli aspetti relativi alla persistenza dei dati.

Alcuni di questi requisiti sono in genere soddisfatti dai cosiddetti *IDE*, che però, a modesto parere dell'Autore, nel mondo PHP sono pochi, giovani ed afflitti da una serie di difetti che possono renderne al limite penoso il loro utilizzo.

MacroPHPagus nasce in questo contesto: l'idea fondamentale è quella di raccogliere in un unico strumento varie funzioni di assistenza allo sviluppo di applicazioni, fermo rimanendo che tale strumento rimanga facilmente accessibile e configurabile, e allo stesso tempo sia il più possibile automatico e comodo per lo sviluppatore.

Allo stato attuale **MacroPHPagus** è ancora in una fase preliminare di sviluppo, e consiste sostanzialmente di una serie di script nati come semplici *hack*, ma è già sufficientemente flessibile ed estendibile per essere utilizzato in progetti di piccola e media dimensione.

MacroPHPagus è stato realizzato interamente con software libero, ed è orientato alla creazione delle "fondamenta" di applicazioni multistrato[1], cercando —con attenzione quasi ossessiva— di attenersi il più possibile agli standard e di minimizzare le assunzioni sulla piattaforma di destinazione.

[1]Per la precisione, **MacroPHPagus** assiste lo sviluppatore principalmente nella realizzazione del dominio e della persistenza dei dati.

Lo sviluppo è avvenuto su una macchina portatile con sistema operativo Linux[2]. I programmi utilizzati sono stati numerosi, ma quasi tutti disponibili direttamente con la distribuzione usata, eccetto *ArgoUML*, *PHPUnit2* e *Zend Framework*, di cui sono creati pacchetti RPM *ad hoc*.

1.2 Requisiti di sistema

MacroPHPagus dovrebbe funzionare quasi completamente *out of the box* su qualsiasi sistema compatibile con Unix che supporti gli strumenti di sviluppo GNU e abbia una shell compatibile con *Bash*.

In particolare, essendo costituito alla base da un *Makefile*, richiede `make` 3.80; per la creazione dei file PHP sono necessari `m4` versione 1.4.3 o successiva e `sed` 4.1.4; la generazione automatica degli *unit test* richiede la presenza di `phpunit` 2.2.1[3] e `grep`.

[2] La distribuzione scelta è stata *Mandriva Linux* 2007 per architetture x86_64.

[3] Questo requisito è deprecato in favore della creazione di modelli appositi gestibili direttamente da **MacroPHPagus**, poiché PHPUnit contiene ancora un certo numero di bug che ne compromettono l'affidabilità in sistemi completamente automatizzati.

Capitolo 2

Come nasce e perché

2.1 Il contesto: sviluppo di un software per la *Fondazione Ugo Bordoni*

L'idea di **MacroPHPagus** nasce nel corso di un progetto di un sistema informatico per la Fondazione Ugo Bordoni, un'importante istituzione collegata con il Ministero delle Comunicazioni, che per esso effettua servizi di consulenza tecnica ed altro in campi di crescente interesse come reti *wireless*, televisione digitale, etc..

Il progetto (chiamato informalmente **Spotlight**) consiste nella realizzazione di una base di dati per il censimento degli *hot spot* *Wi-Fi* ad accesso pubblico sul territorio italiano. Tale base di dati è accessibile e gestibile tramite un'interfaccia web per le operazioni di inserimento/modifica dei dati e per la loro consultazione.

Si tratta dunque di un'applicazione web che presenta una serie di sfide per lo sviluppatore:

- *deve offrire un grado elevato di sicurezza*, poiché manipola dati potenzialmente sensibili, e necessita dunque di limitare l'accesso ad essi a vari livelli;
- deve consentire l'inserimento anche di grandi quantità di dati in

maniera più possibilmente rapida, in vari formati, minimizzando l'intervento umano;

- i dati inseriti devono poter essere controllati dalla FUB prima dell'inserimento nella base di dati;

- deve rendere possibile la consultazione di un sottoinsieme di tali dati in maniera anonima via web, tramite mappe geografiche del territorio italiano.

Oltre a questi requisiti astratti, esistono vincoli sul software da utilizzare: in particolare, l'applicazione deve essere scritta in PHP, ed il DBMS a disposizione è *MySQL*; si è scelto inoltre di basarsi il più possibile su software *open source* e su tecnologie indipendenti dalla piattaforma, in modo da minimizzare i costi iniziali e facilitare l'uso di **SPOTLIGHT** agli utenti finali.

È in questo contesto che è nata l'esigenza dapprima di verificare l'esistenza e lo stato delle tecnologie basate su PHP in Internet e, successivamente, di sviluppare in proprio alcuni strumenti di sviluppo, tra cui **MACROPHPAGUS**.

2.2 Uno sguardo alla situazione attuale in rete

2.2.1 Il linguaggio PHP

PHP è notissimo ormai da anni e largamente impiegato nella realizzazione di applicazioni principalmente orientate al web, anche di una certa complessità. Il perché di questa diffusione è evidente: si tratta di uno strumento molto ricco di funzionalità, facile da apprendere anche per sviluppatori nuovi allo scripting lato server, integrabile con altri software di solida reputazione, quali *Apache HTTP server* e *MySQL*[1]. *Last but not lea-*

[1]Non è inutile notare che negli ultimi anni si sia coniata la sigla *LAMP* in riferimento alle applicazioni web basate (anche) su PHP ed i software citati, operanti su sistemi Linux.

st è disponibile gratuitamente per lo scaricamento ed è compatibile con i sistemi operativi più diffusi.

Bisogna considerare però che lo sviluppo di PHP è soggetto ad una certa entropia, un po' per la sua stessa natura di linguaggio di scripting per la generazione di pagine web dinamiche non pensato inizialmente per reggere grosse applicazioni, un po' anche per gli innumerevoli e talvolta contraddittori contributi da parte della comunità PHP nel mondo, che raccoglie culture e visioni alquanto eterogenee. Ciò ha creato problemi di scalabilità e sicurezza; inoltre la parallela crescita di tecnologie concorrenti (quelle legate a Java *in primis*, ASP, etc...) e il notevole successo dei linguaggi orientati agli oggetti e delle tecniche ad essi correlate, ha costretto gli sviluppatori di PHP ad adeguare frettolosamente il linguaggio, con esiti non sempre felicissimi, almeno nelle fasi preliminari.

Di fatto solo con la versione 5 si inizia a vedere un supporto al modello OO che possa reggere il confronto con linguaggi più longevi, una maggiore attenzione ai concetti di incapsulamento e modularità, l'implementazione di alcuni pattern (es. *Iterator*), la creazione (tuttora in corso) di API orientate agli oggetti e di una "Standard Library" paragonabile a quelle ben note di C++ e Java, l'aggiunta di funzionalità di introspezione (riflessività) degli oggetti software, strumenti più sofisticati per la persistenza dei dati, ed altro.

In altre parole, solo ora si sta cercando di mettere ordine nel caos spingendo decisamente verso convenzioni e *best practices* ormai largamente accettate in altri contesti; è parere dell'Autore che ci sia ancora strada da fare per PHP, nonostante ciò che pensino alcuni entusiasti (interessanti le argomentazioni in [11]).

2.2.2 Gli strumenti di sviluppo

In rete sono presenti numerosi progetti di IDE per PHP, sia liberi che commerciali, disponibili per i più comuni sistemi operativi. Tra questi spiccano nomi "eccellenti" quali *Zend Studio*, scritto dagli stessi autori del

linguaggio e probabilmente il più completo sul mercato, *Dreamweaver*, storico ambiente di *authoring* di siti web, ora acquisito da Adobe Systems, Inc.. Sul versante *open source* va citato sicuramente *PHPEclipse*, che sembra piuttosto promettente in termini di funzionalità e il cui sviluppo procede a grandi passi; un'ulteriore nota positiva è che, essendo scritto in Java, può essere usato su qualsiasi sistema operativo per cui sia stata scritta una JVM.

In realtà, né per **SPOTLIGHT**, né per **MACROPHPAGUS** è stato utilizzato alcuno di questi pur validi strumenti. Le ragioni sono essenzialmente le seguenti:

1. economiche: si è voluto evitare l'acquisto di prodotti di cui non fosse possibile avere un'esperienza d'uso in anteprima, tanto più che in rete sono reperibili gratuitamente prodotti pregevoli con caratteristiche simili;

2. difficoltà legate all'installazione: spesso, purtroppo, l'installazione di grossi pacchetti software in ambiente Linux comporta una serie di problemi tecnici che possono renderla al limite sfiancante per l'utente[2];

3. l'apprendimento dell'uso di queste applicazioni può creare frustrazione nello sviluppatore che proviene per esempio da Java, ove ci sono specifiche piuttosto chiare largamente accettate per ogni aspetto del ciclo di vita di un'applicazione: in PHP infatti, come già rilevato nella sezione 2.2.1, tali specifiche mancano, sono incomplete o mutevoli, oppure sono soggette a "scuole di pensiero".

Queste considerazioni hanno portato alla decisione di utilizzare un editor semplice da installare ed utilizzare sotto Linux, che supportasse

[2]Di fatto, negli ambienti *Unix-like*, l'installazione di software assume talvolta le sfumature di una vera e propria arte, riservata a pochi personaggi che si specializzano nella conoscenza delle particolarità della procedura di compilazione/installazione di questo o di quel programma.

l'evidenziazione della sintassi non solo di PHP, integrato nell'ambiente grafico KDE, ed estendibile tramite *plug–in*: la scelta è ricaduta quindi su *Quanta Plus* che, pur non essendo paragonabile agli IDE citati sopra, possiede tutte le caratteristiche richieste.

2.2.3 API per PHP

Notoriamente una delle fortune di PHP è dovuta alla presenza di numerose funzioni che permettono di effettuare le più svariate operazioni, come manipolare stringhe di caratteri, accedere a file e database, e quant'altro, utilizzando relativamente poche righe di codice.

La rapidità e la comodità di questo approccio in piccole applicazioni sono notevoli, ma non altrettanto si può dire nel caso di sistemi di dimensione da media a grande: infatti, poiché è possibile utilizzare tutte le funzioni ovunque, c'è il rischio potenziale di aggirare il criterio di separazione delle responsabilità tra componenti diversi dell'applicazione con conseguenti problemi aggiuntivi in fase di progettazione e di programmazione, notando *en passant* che si possono introdurre elementi di programmazione procedurale in contesti tipicamente ad oggetti. Altro problema è che un certo numero di funzioni non è *thread–safe*, e questo può avere ripercussioni nell'uso concorrente di risorse condivise e non solo.

Fortunatamente con la versione 5 del linguaggio e, meglio ancora, con la recente introduzione dello *Zend Framework*, il team di sviluppo di PHP sembra aver recepito le problematiche discusse brevemente qui, e non solo: ora finalmente esiste un supporto del paradigma di programmazione OO molto più rigoroso rispetto al passato, e sono state create nuove API che sfruttano tale supporto, **stabilendo di fatto nuovi standard** che andranno a soppiantare la pletora di soluzioni —più o meno buone— già esistenti per la risoluzione di problemi comuni.

In particolare diviene una pietra miliare proprio lo Zend Framework, che fornisce ai programmatori tutta una serie di API OO legate alla gestione della persistenza, delle sessioni HTTP, ACL, internazionalizzazione, e

molto altro, che sono preziose nello sviluppo di grandi sistemi, come nel caso di **Spotlight**. Operando in tal modo, i creatori del linguaggio si riappropriano della loro funzione di guida per gli sviluppatori, collaborando con la comunità per la creazione di soluzioni largamente condivise e di qualità.

2.2.4 Dei *framework* e dei *template*

Negli ultimi anni il crescente interesse verso i cosiddetti *pattern*, cioè dei modelli di risoluzione di problemi ricorrenti a livello di progettazione di un'applicazione, un'architettura e quant'altro, ha portato alla creazione di nuovi software che assistono lo sviluppatore nella loro implementazione. Allo stesso modo i vari linguaggi di programmazione sono soggetti ad alcune regole, in genere dettate dai loro creatori, in maniera da stabilire uno standard di formattazione del codice che ne migliori la gestione.

Nella comunità PHP c'è un notevole fermento, sia nella creazione di standard di codifica, sia nell'implementazione di pattern e *best practices*, con l'obiettivo —ad ora solo parzialmente raggiunto— di mettere PHP in diretta competizione con linguaggi come Java, Perl o Python nella realizzazione di applicazioni con alto grado di sicurezza e scalabilità.

Si assiste così alla crescita di iniziative come *PEAR*, un archivio di API per i più svariati utilizzi che, pur ospitando progetti piuttosto eterogenei quanto a qualità e frequenza degli aggiornamenti, ha il pregio di stabilire o comunque ribadire alcune convenzioni riguardanti la codifica e la gestione delle nuove librerie che si vogliono rendere disponibili al programmatore. Avendo guadagnato un notevole prestigio nel corso degli anni, è il punto di riferimento per l'installazione di importanti strumenti come *PHPUnit*[3] e *PHPDocumentor*[4].

[3] Si tratta di un framework per lo *unit testing* in PHP, nato come un *port* di *JUnit*.

[4] Altro famoso progetto che di fatto rappresenta lo standard per la documentazione del codice, anche qui sulla falsariga di *Javadoc*, di cui si propone come un'evoluzione orientata al PHP.

Sul fronte delle applicazioni per il web, sono attivi alcuni progetti importanti, tra cui vale la pena di ricordare *CakePHP* e *Symfony*[5], che implementano il pattern MVC ed offrono una serie di facilitazioni per la gestione della persistenza e dell'interfaccia utente.

Anche per quanto riguarda la persistenza dei dati, ci sono molte scelte: da *PEAR::MDB* e *Propel* all'estensione PDO creata ed integrata nel linguaggio dagli stessi autori di PHP; ognuno di questi offre una serie di API che astraggono in maniera sempre più raffinata dal metodo di memorizzazione effettivo dei dati.

Una importante tecnica utilizzata nei vari framework e non solo è quella del *templating*, ossia la realizzazione di **modelli** di file contenenti una serie di costrutti comuni riutilizzabili e personalizzabili a seconda delle esigenze: ad esempio, nella realizzazione di un sito web, si crea un file HTML contenente un'intestazione ed un menu comuni a tutte le pagine del sito, ma privo dei contenuti specifici di ogni pagina; l'autore è libero quindi di personalizzare tale file *in maniera statica*, creandone tante copie quante sono le pagine del sito, e riempiendole manualmente di contenuti, oppure —più comunemente— *in maniera dinamica*, avvalendosi di costrutti in qualche linguaggio che si occupano di generare al volo pagine web ad ogni richiesta HTTP.

Talvolta si arriva a definire veri e propri metalinguaggi, caratterizzati da un'estrema semplicità d'uso, che consentono anche ai "non addetti ai lavori" di pubblicare sul web le loro informazioni in maniera razionale e con un aspetto accattivante: è il caso dei software alla base di *forum*, *blog*, *wiki*, che ormai spopolano in Internet, e che spesso sfruttano PHP come motore interno.

Il concetto stesso di template (sempre di pagine web) è alla base del linguaggio PHP, il cui codice è spesso immerso in quello HTML, e da cui è separato da uno speciale *tag*; lo stesso principio è applicato ad esempio

[5]Da notare che entrambi i prodotti sono ispirati a *Ruby on Rails*, un maturo framework per il linguaggio Ruby che rappresenta una pietra miliare nel suo settore.

anche alle tecnologie ASP e JSP.

I template interessano anche ai programmatori che devono sviluppare porzioni ripetitive di codice contenenti alcune differenze al loro interno, e sono preoccupati di limitare errori e di scrivere il meno possibile: esiste perfino un famoso pattern *GoF* chiamato ***Template** method*, che risponde al problema di scrivere in classi diverse metodi che abbiano in comune numerose parti di codice. Una sua descrizione è presente in [5].

2.3 Come e dove si colloca MacroPHPagus?

Il panorama presentato nella sezione 2.2 è notevolmente vario e soprattutto mutevole nel tempo, per cui risulta difficile da una parte scegliere tra ciò che è già esistente, e dall'altra creare qualcosa che non sia una mera "reinvenzione della ruota", tenendo pure in considerazione le possibilità limitate di un singolo sviluppatore indipendente, rispetto ad un'immensa comunità preesistente da anni di tecnologie, personaggi e gruppi di lavoro, ognuno con la propria esperienza, le proprie convenzioni, le proprie idiosincrasie.

Dunque lo sforzo del singolo deve essere orientato a trovare un equilibrio tra l'adozione di tecniche e prodotti più affermati, e l'innovazione attraverso la creazione di strumenti nuovi a partire da idee o punti di vista più o meno originali; ovviamente tutto ciò non può prescindere da vincoli temporali e di contesto, così come dalle proprie abilità creative.

MacroPHPagus è nato per rispondere all'esigenza di sviluppare un'applicazione ad oggetti non banale, racchiudendo il codice base di classi ed interfacce in modelli preformattati, e al contempo di applicare alcuni pattern al codice finale. Nel corso del tempo sono inoltre sorte alcune problematiche cui non si è trovata risposta adeguata effettuando ricerche in Internet, come ad esempio una gestione delle collezioni di oggetti che prescindesse dalla manipolazione diretta degli `array` di PHP; la soluzione è stata quella di sviluppare delle soluzioni *ad hoc* da integrare nel progetto.

Dunque quello che inizialmente era semplicemente il *makefile* di un'applicazione PHP, ha acquisito nel tempo una sua dignità, incorporando funzionalità valide non solo in quel particolare contesto, ma riutilizzabili altrove.

A quel punto si sono posti due problemi fondamentali:

- separare questo nuovo strumento dal progetto principale;
- stabilire delle specifiche che ne delimitassero il campo di azione.

Al momento della stesura di questo documento la situazione è la seguente: **nessuno dei due punti è ancora risolto**, poiché in effetti **MacroPHPagus** è ancora in una fase embrionale, e si potrebbero immaginare diversi scenari di evoluzione interessanti, che saranno brevemente discussi nel capitolo 6.

Comunque, al di là di tali scenari, l'obiettivo fondamentale rimane quello di ottenere uno strumento tecnico per gli sviluppatori PHP che sia *ben concepito*, ancor prima che ricco di funzionalità, e che sopperisca a quelle che l'Autore ritiene delle lacune, a vari livelli, tanto del linguaggio stesso, quanto delle tecnologie e degli standard ad esso correlati.

Capitolo 3

Caratteristiche ed uso

3.1 Visione e installazione

Come già accennato precedentemente, **MacroPHPagus** consiste attualmente di:

- un modello di *makefile* configurabile, contenente alcuni *target* predefiniti per la produzione di file PHP;
- una struttura di macro M4 che svolgono la trasformazione effettiva in codice PHP di file opportunamente scritti;
- una semplice libreria di API OO per PHP 5.

È importante notare che non sono mai richiesti privilegi amministrativi sulla macchina di lavoro, ed inoltre che qualsiasi file prodotto da **MacroPHPagus** è indipendente da esso, ossia

> *un'applicazione costruita con* **MacroPHPagus** *non ha bisogno di esso per funzionare.*

Eventualmente, una volta generati i file desiderati è possibile migrare in piena sicurezza verso altri software di sviluppo, con il vantaggio di avere codice leggibile, documentato, conforme agli ultimi standard, in cui

ulteriori interventi si limitano sostanzialmente alla modifica del corpo dei metodi.

Discorso a parte per le API: esse possono essere installate sia in una directory di sistema (es. `/usr/share/mst_php5_api/`), oppure nella stessa directory contenente i file dell'applicazione, specificando (per ora) manualmente il percorso nella direttiva `include_path` di `php.ini`.

Attualmente queste API consistono in:

- un'implementazione parziale di un framework per la gestione delle collezioni di oggetti analogo al *Java Collections Framework*, compatibile con gli strumenti offerti dalla SPL, introdotta nella versione 5 di PHP;
- un *parser* di file di configurazione testuali con struttura ad albero;
- un framework minimale per la gestione della persistenza dei dati, basato su Zend Framework.

3.2 Principi e raccomandazioni d'uso

L'architettura attuale di **MacroPHPagus** va notevolmente incontro agli sviluppatori *GNU–style*, dato che si basa proprio su strumenti GNU ed è accessibile tramite la shell di sistema; i file sorgente sono testuali, perciò basta un semplice *editor* di testo.

La procedura prevista è la seguente:

1. all'inizio dello sviluppo di un'applicazione PHP 5 ad oggetti, si predispone la directory che la conterrà, copiandovi i file di **MacroPHPagus**;
2. si creano i file macro contenenti la descrizione di classi ed interfacce, nonché di alcuni pattern;
3. tramite il comando `make` si generano i file PHP corrispondenti;

4. opzionalmente è possibile generare un file di configurazione per la persistenza dei dati, in cui sono elencati gli oggetti persistenti e le informazioni sul metodo per la memorizzazione;

5. se si dispone dello schema di un database MySQL, il comando `make` è in grado di inserirlo nel DBMS, previa creazione manuale del DB stesso;

6. i file ottenuti possono essere "installati" in una directory scelta dall'utente;

7. sempre tramite `make` si possono generare in blocco gli *unit test* per tutte le classi abilitate tramite un *flag* nel file sorgente.

I file sorgente devono essere salvati tutti nella directory principale dell'applicazione affinché possano essere "(ri-)compilati".

I file sorgente **devono** rispettare alcuni vincoli sui nomi utilizzati al fine di evitare di essere ignorati in fase di processamento. Le regole sui nomi sono illustrate in dettaglio nella sezione 4.2.1.

Una volta eseguito il comando `make`, i file ottenuti sono tutti nella stessa directory, salvo gli eventuali scheletri degli unit test, che vengono scritti in una sottodirectory specifica. Per maggiori informazioni si vedano le sezioni 4.2.2 e 4.1.4.

È estremamente importante (per ora) che ogni file sorgente su cui si inizia a lavorare sia creato a partire dai modelli `*_tpl.m4`, attenendosi scrupolosamente alla sintassi proposta. I modelli esistenti sono illustrati nell'appendice B.

Si consiglia, nella fase di stesura del corpo di un metodo, di rispettare le regole sullo stile di codifica di PHP, in particolare riguardo alle tabulazioni, che consistono come impostazione predefinita in 4 spazi (*NON* usare il carattere di tabulazione). La lunghezza della tabulazione è modificabile agendo sulla variabile `TABSIZE` della configurazione del makefile. Eventuali errori di sintassi nel corpo dei metodi saranno ignorati da **MacroPHPagus**.

3.3 Esempio d'uso

Si supponga di aver scritto tre file sorgente, chiamati:

`ConfigParser_cls.php.m4`

`IMapper_int.php.m4`

`IOException_cls.php.m4`

Tali file contengono evidentemente una classe non astratta, un'interfaccia e una classe eccezione. Si noti la chiarezza nei nomi dei file (e delle rispettive classi ed interfacce), che scritti in questa maniera risultano autodescrittivi.

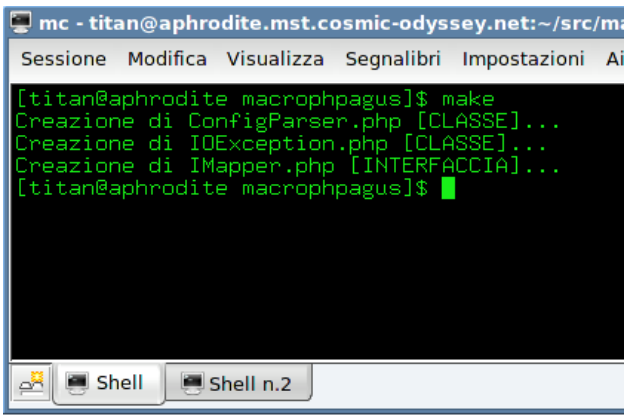

Figura 3.1: *Output* del comando `make` in caso di successo.

Eseguendo il comando `make` senza argomenti si ottiene (salvo errori) la creazione dei seguenti file:

`ConfigParser.php`

`IMapper.php`

`IOException.php`

Come si vede in figura 3.2, **MacroPHPagus** crea un'intestazione — sottoforma di commento PHP— che contiene alcune informazioni usate

internamente, quindi genera le inclusioni di altri file, seguite dal DocBlock con la documentazione della classe ed i relativi tag di PHPDocumentor, ed infine il codice della classe stessa, aggiungendo ove possibile ulteriori commenti per migliorare la leggibilità del codice.

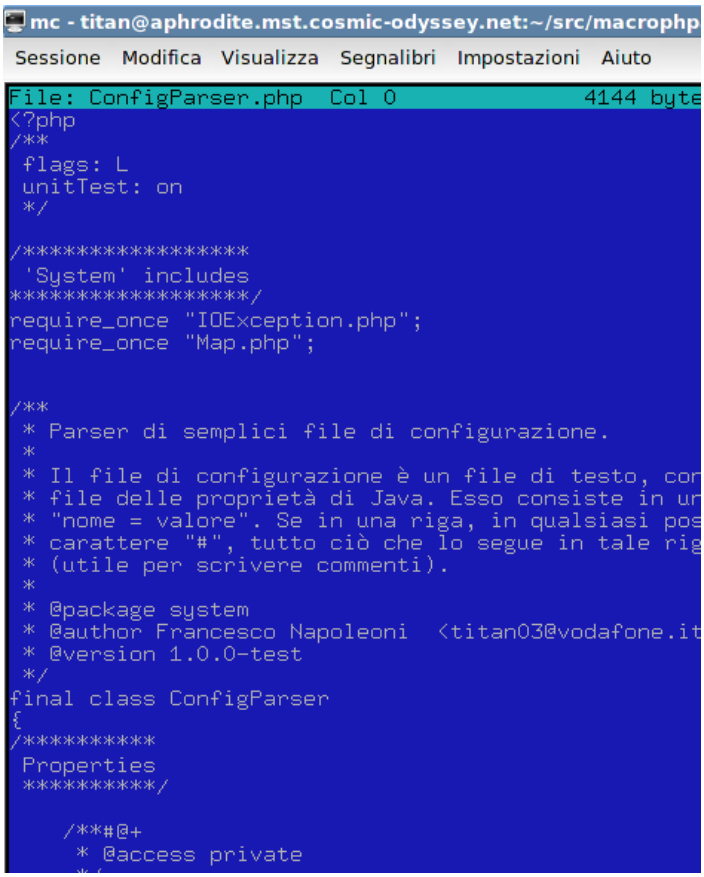

Figura 3.2: Esempio di classe generata con **MacroPHPagus**.

Non appena generato il codice, si può creare automaticamente lo scheletro degli unit test per le classi che sono state abilitate con l'apposita macro, agendo come in figura 3.3.

Si noti nella figura che è stato aggiunto e processato un nuovo file chiamato `Map_cls.php.m4`, che costituisce un requisito per la classe **ConfigParser**, nonché una dipendenza del target `unit_tests` di *make*.

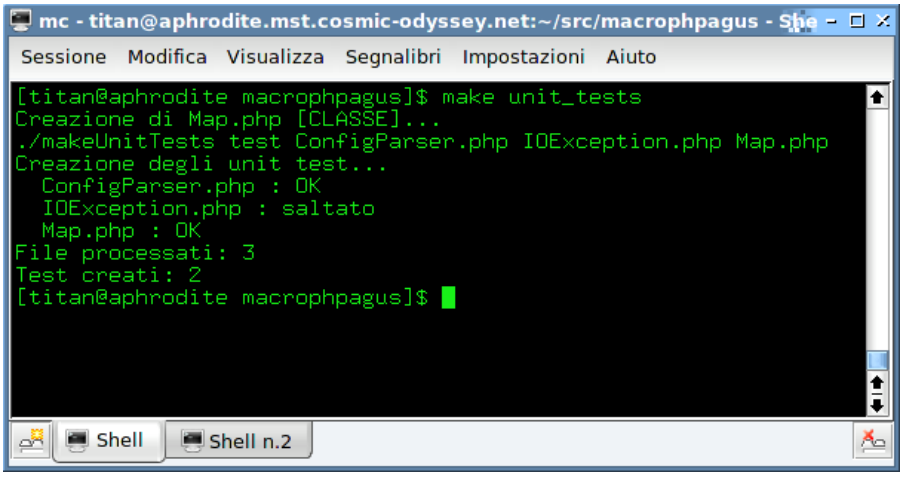

Figura 3.3: Risultato della creazione degli unit test.

La classe **IOException** viene scartata perché rappresenta un'eccezione software e quindi non può essere processata allo stesso modo delle altre classi.

Al termine dell'esecuzione del target sono visualizzate le statistiche sui file processati.

Capitolo 4

Architettura

4.1 Panoramica

La struttura complessiva di **MacroPHPagus** riflette fortemente le influenze sull'Autore dell'esperienza con Linux ed il software libero in generale, con gli strumenti di sviluppo storici di Unix, e con Java.

Le prossime sezioni illustreranno i dettagli tecnici di questo software, mettendo in evidenza anche le idee mutuate dalle esperienze citate.

4.1.1 Compatibilità con gli standard

Durante le prime fasi di lavoro su **Spotlight** (il progetto brevemente illustrato nella sez. 2.1), è apparso subito importante adottare dei modelli per gestire nella maniera più razionale ed omogenea grandi quantità di classi ed interfacce secondo il principio *DRY*[1], nonché per risparmiare sul tedioso compito di scrittura del codice[2]; inoltre, allo scopo di assicurarne

[1]Gli esperti nei più svariati ambiti scientifici amano oltremodo creare fantasiosi acronimi. In questo caso abbiamo un possibile candidato ad undicesimo Comandamento: ***D**on't **R**epeat **Y**ourself*.

[2]Mai come nel progetto in questione tali esigenze sono state stringenti: infatti le responsabilità di progettista, analista, programmatore, tester nonché redattore della documentazione gravavano inevitabilmente sullo stesso essere umano...

la leggibilità e la compatibilità con alcuni degli strumenti più diffusi, si è reso necessario studiare le specifiche sulla qualità del codice PHP dettate dagli stessi creatori del linguaggio, e ribadite nelle raccomandazioni agli sviluppatori PEAR (si veda a riguardo [8], cap. 4.).

Man mano che si è proceduto nello sviluppo e con la scoperta di nuovi strumenti, sono stati assimilati i seguenti standard:

- standard di codifica per PHP PEAR;
- formato della documentazione di PHPDocumentor 1.3;
- nomi dei target di `make` più comuni.

Nel modello di classe adottato si privilegia il principio di inizializzazione procrastinata (*lazy*) delle proprietà.

Altro aspetto importante è l'adozione delle ultime tecnologie realizzate per PHP, in particolare riguardanti il nuovo supporto OO di PHP 5, la recente SPL e Zend Framework. Come già abbondantemente discusso sotto vari aspetti nel capitolo 2, appare che i risultati più importanti da parte della comunità PHP in direzione dello sviluppo OO siano avvenuti solo in tempi recenti; proprio per questo motivo si è preferito orientarsi verso implementazioni di tecnologie magari giovani ed incomplete, ma più rispettose dei principi teorici, e —dettaglio non irrilevante— attivamente supportate dallo stesso team di sviluppo di PHP.

Il risultato è che i file PHP generati da **MacroPHPagus** sono sintatticamente corretti, altamente autodocumentanti e compatibili con le ultime versioni di qualsiasi altro strumento di sviluppo che rispetti le più recenti specifiche del linguaggio.

Da notare che le API scritte a completamento di **MacroPHPagus** sono state realizzate con il suo generatore di codice, e che a loro volta recepiscono interfacce e convenzioni già consolidate, adattandole alle nuove funzionalità di PHP 5 (ad es. l'implementazioni degli iteratori e delle eccezioni).

Tutti questi accorgimenti fanno sì che **MacroPHPagus**, se da una parte è un software ai suoi primi passi, decisamente meno raffinato degli IDE già esistenti, sia nondimeno ben concepito e proteso verso il futuro.

4.1.2 Separazione tra struttura degli oggetti software e formato del codice PHP

L'idea di poter astrarre —almeno parzialmente— rispetto alla sintassi dello specifico linguaggio di programmazione è affascinante poiché consente allo sviluppatore di concentrarsi maggiormente sugli aspetti concettuali dell'applicazione che sta realizzando, specialmente se non dispone di un team di programmatori specializzati; un aspetto correlato è quello di un'implementazione sintetica delle classi in corso di realizzazione tramite un metalinguaggio semplice, che possa rapidamente e soprattutto *automaticamente* essere tradotto in qualsiasi momento nel linguaggio finale desiderato.

MacroPHPagus per ora non pretende di soddisfare a pieno tali desideri; ciò nondimeno, grazie ai file di macro che accetta in ingresso, offre un certo numero di facilitazioni in tal senso: già ora il programmatore non ha che da scrivere il codice dei metodi, la documentazione (nel formato di PHPDocumentor, con qualche semplificazione) e ben poco altro, dal momento che può avvalersi di flag e macro appositamente realizzati per "confezionare" il risultato secondo le proprie esigenze.

L'obiettivo è quello di sintetizzare il più possibile il "codice" dei file sorgente, cosicché lo sviluppatore possa avere un colpo d'occhio sulla struttura della classe che sta realizzando, senza perdersi nei meandri della sintassi PHP più di quanto non sia strettamente necessario.

4.1.3 Applicazione di pattern e costrutti comuni

Un altra caratteristica desiderabile per uno sviluppatore è avere —laddove possibile— un'implementazione di alcuni pattern ben noti, oppu-

re degli scheletri di costrutti del linguaggio come cicli, strutture condizionali, e quant'altro richieda la scrittura di codice ripetitivo e/o prolisso.

A questo scopo sono definite macro che permettono di effettuare alcune utili operazioni:

- inserire il codice appropriato per rendere una classe *singleton*;
- inserire e formattare appropriatamente la documentazione delle classi come *DocBlock* ed i relativi tag di PHPDocumentor;
- una macro speciale, chiamata `FLAGS`, permette di specificare sia l'estensibilità della classe, sia eventuali pattern applicati, o se si tratta di un'eccezione;
- includere il codice di altri file PHP da posizioni predefinite;
- abilitare la creazione automatica di *unit test* per la specifica classe (posto che non si tratti di un'eccezione o di una classe astratta);
- specificare se la classe è persistente, ed eventualmente il tipo di metodo di serializzazione[3].

4.1.4 Strumenti di programmazione PHP 5 a corredo

Come già accennato precedentemente, è in corso di realizzazione una serie di librerie OO che vanno a colmare alcune lacune di analoghi strumenti già esistenti in rete. Naturalmente non vi è alcuna pretesa di completezza né di ottimalità delle soluzioni presentate, ma si è ritenuto comunque importante avere a disposizione strumenti ad alto livello durante le fasi di sviluppo.

Attualmente le API disponibili sono contenute nel pacchetto **SYSTEM** e consistono in:

[3]Questa funzionalità è solo parzialmente implementata, dato che dipende da un'API per la gestione della persistenza ancora in fase di progettazione, che sarà discussa più avanti in questo capitolo.

- un'implementazione delle mappe associative fortemente ispirata a **HashMap** di Java;

- una gerarchia di eccezioni comuni nelle applicazioni, integrabile con quella della SPL;

- un parser di file di configurazione testuali.

È anche allo studio un pacchetto apposito per la gestione della persistenza basato su Zend Framework, con l'obiettivo di astrarre il più possibile dai metodi di serializzazione dei dati, seguendo le preziose indicazioni presenti in [2] e in [5], cap. 34.

Vale la pena di soffermarsi sulla necessità di questi strumenti.

Collezioni di oggetti

Storicamente PHP ha basato la gestione delle collezioni di dati prevalentemente sul tipo predefinito `array`, che può essere utilizzato sia per vettori e matrici che per semplici mappe associative. Solo ultimamente, con l'introduzione della SPL, e specificamente degli **ArrayObject**[4] e degli iteratori, si può disporre di strumenti più sofisticati, e tuttavia ancora insufficienti: ad esempio non sono ancora gestite strutture dati come liste ed alberi con un'interfaccia comune, ad oggetti; le chiavi degli array possono essere solo numeri o stringhe, mentre sarebbe utile poter avere chiavi di tipo `object`.

Per rispondere a queste esigenze, **MacroPHPagus** propone di implementare in PHP una versione "riveduta e corretta" del Java Collection Framework, che è parte delle API di Java fin quasi dalla sua nascita, non trascurando però le novità introdotte con la SPL.

L'idea è di simulare le varie strutture dati agendo opportunamente sugli array di PHP, ed estendendo le funzionalità della SPL.

[4] **ArrayObject** è un *wrapper* del tipo `array` per l'uso in applicazioni ad oggetti.

Per ora sono implementate solo le mappe associative, che sono di uso comune nelle applicazioni ad oggetti come **SPOTLIGHT** per la rappresentazione dei multioggetti di UML. Le caratteristiche fondamentali di tali mappe sono:

- approccio ad oggetti;
- supporto delle chiavi di tipo `mixed` (oggetti compresi), implementato tramite la funzione predefinita `serialize()` di PHP;
- importazione di array già esistenti.

Eccezioni

In PHP 5 è stata introdotta una gerarchia di generiche eccezioni, orientate essenzialmente al *debugging* delle applicazioni. A fianco di esse **MACROPHPAGUS** offre ulteriori estensioni, utilizzabili in contesti più generali, non legati esclusivamente ad errori di programmazione.

Ad ora è implementata la sola classe **IOEXCEPTION**, che rappresenta l'eccezione omonima sul versante Java, ed è estendibile allo stesso modo; per il futuro sono previste ulteriori gerarchie, tutte utilizzate dalle varie classi delle API di **MACROPHPAGUS** e disponibili al programmatore.

File di configurazione

La classe **CONFIGPARSER** legge file di configurazione scritti nel seguente formato:

```
proprietà1=valore1
proprietà2.sottoproprietà1 = valore lungo e separato da spazi

# i commenti sono ignorati

# la seguente riga è valida
altronome=
```

...

Due sono le caratteristiche fondamentali di questa implementazione:

- permette una semplice rappresentazione ad albero di gerarchie di proprietà, usando la notazione puntata;
- consente l'uso di lettere accentate, simboli ed alfabeti diversi da quello latino sia per i nomi delle proprietà che per i loro valori, grazie al supporto alla codifica UTF-8.

Tramite il comando `make` è possibile generare automaticamente file di configurazione leggibili da **ConfigParser**, contenenti le informazioni necessarie alla configurazione della persistenza, compresa la mappatura tra oggetti del dominio e le tabelle del DB.

L'attuale implementazione di **ConfigParser** agisce direttamente su un file testuale e utilizza un array come rappresentazione interna della configurazione.

Persistenza degli oggetti

Al momento della stesura di questo documento è in corso di progettazione un sistema di gestione della persistenza dei dati basato sul pacchetto **Zend_Db** di Zend Framework, con la finalità di rendere più possibile trasparenti le operazioni di (s)materializzazione degli oggetti del dominio. Questo obiettivo può essere raggiunto organizzando opportunamente gli oggetti del dominio in modo che **MacroPHPagus** sia in grado di generare tanto lo schema del DB, quanto il file di configurazione contenente le mappature O/R.

Devono inoltre essere rese possibili operazioni su DB o su altri sistemi di memorizzazione anche ad oggetti non strettamente appartenenti al dominio dell'applicazione, e deve essere supportato l'accesso a basi di dati tramite ODBC[5].

[5]ODBC non è ancora supportato da **Zend_Db**, e quindi sarà necessario scrivere un adattatore apposito.

Il sistema si baserà sul seguente principio di funzionamento: gli oggetti del dominio estendono una classe astratta chiamata **PersistentObject**, e ognuno di essi avrà un suo specifico OID, indipendente dall'applicazione e generato automaticamente secondo la tecnica *High/Low*, discussa in [3]; saranno consentiti inserimenti e recuperi di collezioni di oggetti del dominio.

4.2 M4: il motore interno

4.2.1 Organizzazione dei file macro

I file di macro M4 sono utilizzati da **MacroPHPagus** per generare il codice PHP di classi ed interfacce: si tratta di file testuali con codifica dei caratteri UTF-8, nominati secondo il seguente formato:

- *NomeClasse*`_cls.php.m4`, con *NomeClasse* il nome dell'oggetto che si vuole rappresentare;
- *NomeInterfaccia*`_int.php.m4`, con *NomeInterfaccia* il nome dell'interfaccia[6].

È possibile scrivere da zero tali file, ma è preferibile usare i modelli preformattati inclusi nella distribuzione, e cioè:

`class_tpl.m4` : elenca e documenta le macro necessarie per creare un file contenente una generica classe, eccezioni escluse;

`interface_tpl.m4` : elenca e documenta le macro necessarie per creare un file contenente un'interfaccia;

[6]Si consiglia di utilizzare per i nomi delle classi ed interfacce —e quindi anche per i relativi nomi file— la cosiddetta notazione "a gobbe di cammello": in altre parole i nomi devono essere tutti minuscoli salvo che per le iniziali di tutte le parole che li compongono, e non possono contenere spazi, trattini o sottolineature.

`exception_tpl.m4` : elenca e documenta le macro necessarie per creare un file contenente una classe speciale che gestisce un'eccezione;

tali modelli vanno modificati aggiungendo il proprio codice, e devono essere rinominati come discusso sopra.

Il nome dato al file sarà utilizzato da M4 come nome del relativo oggetto. Ad esempio, se si vuole creare una classe chiamata **HotSpot**, il nome del file di macro deve essere `HotSpot_cls.php.m4`, in modo che l'output di M4 sia un file chiamato `HotSpot.php` contenente il seguente frammento di codice:

```
...
class HotSpot
{
...
```

La sintassi dei file di macro è documentata nei modelli predefiniti, illustrati nell'appendice B.

I file sorgente devono essere processati da M4, previa inclusione di questi file:

- `macros.m4`, che contiene l'inizializzazione di M4 e la definizione alcune macro di uso generale;
- `php5_templates.m4`, contenente le definizioni di frammenti di codice PHP opzionali;
- `class_generator.m4` oppure `interface_generator.m4`, che contengono il codice base di classi ed interfacce, effettuano alcuni controlli di coerenza, ed applicano la formattazione e le macro definite nei file sorgente.

A proposito dei controlli di coerenza, è opportuno spendere due parole sul loro funzionamento: fondamentalmente essi assolvono al compito di evitare errori logici grossolani nella scrittura dei sorgenti, come il tentativo

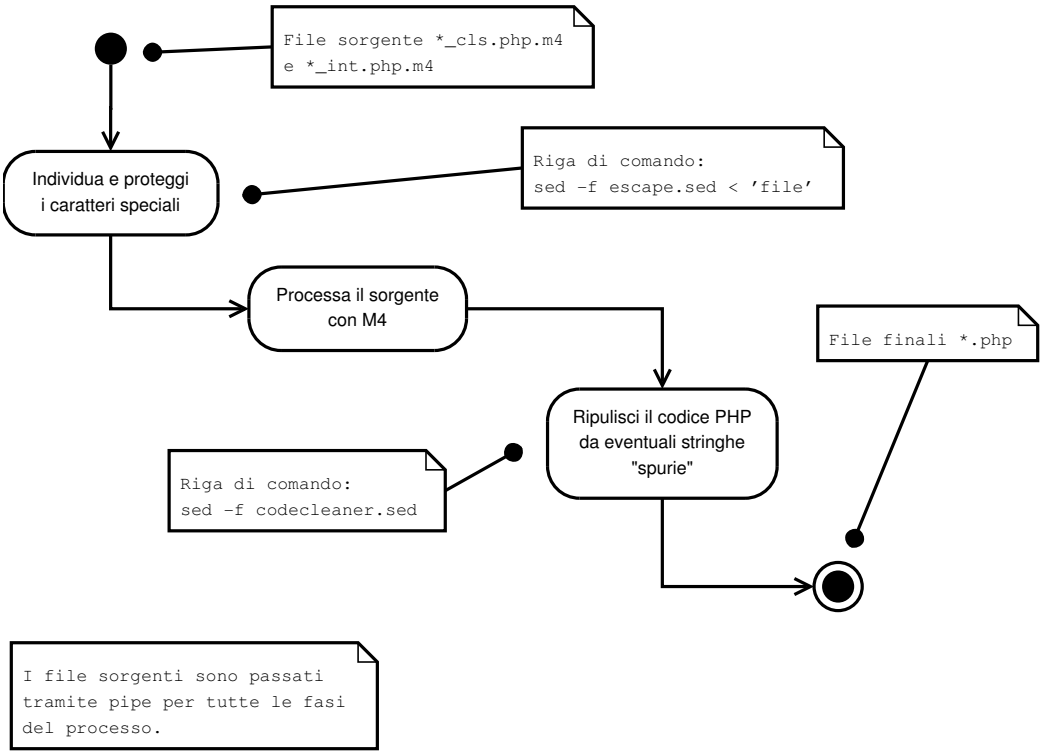

Figura 4.1: Diagramma delle attività del processore di macro.

di dichiarare come foglia una classe astratta, oppure abilitare la generazione di unit test per particolari tipi di classi, quali le eccezioni. Va detto che questi controlli sono per ora alquanto ingenui e poco flessibili, ma sono previste migliorie nell'immediato futuro.

Ulteriore funzione implementata è la dichiarazione di una classe come persistente, che comporta l'estensione automatica di **PersistentObject** e l'inserimento delle informazioni relative al metodo di memorizzazione come commento all'interno del codice PHP; tali informazioni saranno poi utilizzate da **MacroPHPagus** per generare le mappature opportune.

4.2.2 Script esterni

MacroPHPagus ricorre all'uso di script esterni quando deve effettuare operazioni complesse. Attualmente due sono i casi in cui ciò avviene; essi

sono descritti di seguito.

`makeConfig` – Configurazione della persistenza

Si tratta di uno script *Bash* deputato alla creazione di un file di configurazione nel formato richiesto per **ConfigParser** (→ sez. 4.1.4, pag. 26), usato per specificare le mappature O/R degli oggetti del dominio e le informazioni sui supporti di memorizzazione usati, in particolare quelle di connessione ad un DBMS.

La sua sintassi di invocazione è la seguente:

```
makeConfig nome_file_da_generare [nome_DBMS DB_HOST DB_NOME
    DB_NOMEUTENTE DB_PASSWORD]
```

i cui parametri sono autoesplicativi.

È importante notare che **questo script non deve essere mai eseguito manualmente**, poiché esiste un apposito *target* di `make` che inserisce automaticamente i dati richiesti.

Stesso discorso vale per il file di configurazione generato: anche questo *non va modificato a mano*, in quanto `makeConfig` inserisce in tale file le informazioni trovate nella configurazione di **MacroPHPagus**[7] ed inferisce automaticamente le mappature leggendo l'apposito *flag* `P` nei file contenenti le classi PHP[8]. Eventuali modifiche manuali sono sovrascritte dalle successive esecuzioni di `make`.

`makeUnitTests` – Generazione automatica di unit test

Anche in questo caso si ha uno script Bash eseguito automaticamente da `make`. Esso si occupa di creare lo scheletro dei test dei metodi delle classi che sono state abilitate tramite la macro `ENABLE_TEST`. La sua attuale implementazione richiede che sia installato nel sistema *PHPUnit2*, poiché usa il comando `phpunit` con l'opzione `--skeleton`.

[7] La configurazione del makefile sarà discussa più avanti, nella sezione 4.3.2.

[8] I flag impostabili per le classi sono descritti in dettaglio nell'appendice B.1.

`makeUnitTests` ha la seguente sintassi:

```
makeUnitTests test_dir NomeClasse1.php ...
```

in cui `test_dir` indica il percorso relativo della directory che conterrà gli scheletri dei test generati, mentre i successivi parametri consistono in una lista arbitrariamente lunga di classi PHP da processare.

`makeUnitTests` genera un file nominato `NomeClasseXTest.php` per ogni corrispondente file `NomeClasseX.php`, controllando che:

1. quest'ultimo contenga nell'intestazione la stringa `unitTest: on`, presente in genere alla quarta riga;
2. in caso affermativo avviene il controllo dell'esistenza di un file chiamato `NomeClasseXTest.php`[9] nella directory `test_dir`;
3. in caso negativo il file viene passato a `phpunit` che genera materialmente lo scheletro del test per quella classe.

Lo stato di avanzamento del processo ed il risultato finale vengono scritti sullo *standard output*, mentre tutti i messaggi di `phpunit` vengono annotati nel file di log `phpunit.log`, all'interno della directory `test_dir`.

4.3 Interfaccia utente tramite `make`

Il punto di accesso principale alle funzioni di **MacroPHPAgus** è costituito dal comando `make`. L'idea fondamentale è che l'applicazione desiderata venga costruita in una directory contenente la struttura di file di **MacroPHPAgus**, dove saranno salvati pure i sorgenti ed i file di configurazione: ogni operazione avviene esclusivamente entro quella directory, o al più in una sua sottodirectory, senza richiedere privilegi più elevati di quelli di cui l'utente dispone nel sistema.

[9] Questa operazione serve ad evitare che test già creati precedentemente possano essere accidentalmente sovrascritti.

L'utente crea una copia dei modelli per ogni classe o interfaccia che desidera scrivere, quindi esegue dalla shell il comando

```
$ make [{NomeClasse|NomeInterfaccia}.php ...]
```

ove, come si vede, i target corrispondono ai nomi file che si vogliono generare, e **non** ai file sorgente. L'invocazione di `make` senza parametri provoca il processamento di *tutti* i file classe ed interfaccia.

Il comportamento di `make` può essere modificato agendo sia sui file di configurazione (`config.mk` e `usr_config.mk`), sia aggiungendo target al makefile. *La modifica dei target esistenti è fortemente sconsigliata*, a meno di non conoscerne molto bene i meccanismi interni di funzionamento.

4.3.1 Struttura del makefile

Il makefile risiede nella directory principale dell'applicazione e contiene un certo numero di target che assistono alcune fasi della costruzione. I principali target sono discussi di seguito, eccetto `all`, che evidentemente crea il codice PHP di tutti i sorgenti che trova, `help` e `clean`, il cui significato è ovvio.

`%.php` Vi sono due target con lo stesso nome: uno si occupa di convertire in PHP i singoli sorgenti `*_cls.php.m4`, mentre l'altro converte i singoli file `*_int.php.m4`. Richiede SED e M4 per funzionare, nonché i seguenti file:

- `macros.m4`;
- `php5_templates.m4`;
- `class_generator.m4`;
- `interface_generator.m4`;
- `codecleaner.sed`;

- `escape.sed`;

Dipendenze: sorgenti `*_cls.php.m4`, `*_int.php.m4`, `*_generator.m4`.

`$(CLASSES_PHP)` e `$(INTERFACES_PHP)` Questi due target si occupano di creare in blocco il codice PHP di tutti i file sorgente rispettivamente con estensioni `_cls.php.m4` (classi) e `_int.php.m4` (interfacce).

Dipendenze: `%.php`.

`unit_tests` Crea gli scheletri degli unit test per le classi abilitate nella directory specificata dalla variabile `TESTDIR`, in `config.mk`. Non sovrascrive eventuali test già presenti. La sua esecuzione richiede lo script `makeUnitTests` e PHPUnit.

Dipendenze: `$(CLASSES_PHP)`.

`$(CONF_FILENAME)` Crea il file di configurazione contenente la configurazione dell persistenza, a partire dalla configurazione di **MacroPHPagus** e dai flag impostati nelle classi PHP. Richiede lo script `makeConfig`.

Dipendenze: `$(CLASSES_PHP)`.

4.3.2 Configurazione del makefile

Due sono i file deputati a contenere la configurazione di **MacroPHPagus**, chiamati rispettivamente `config.mk` e `usr_config.mk`.

`config.mk` È il file contenente la configurazione di base, ossia i percorsi dei programmi di sistema, i nomi degli script, ed altre informazioni che concernono il comportamento di **MacroPHPagus**. I valori presenti non dovrebbero in genere essere mai modificati dall'utente, dato che impostazioni errate possono facilmente pregiudicare il funzionamento di tutto il framework.

`usr_config.mk` In questo file l'utente può inserire tutte le informazioni che riguardano l'applicazione che intende sviluppare. In particolare sono attualmente supportate le variabili (ed i loro eventuali valori di default):

- `APPNAME;`
- `CONF_PATH = ./;`
- `CONF_FILENAME = persistence.conf;`
- `DBMS_NAME;`
- `DB_NAME;`
- `DB_USERNAME = $(DB_NAME);`
- `DB_PASSWORD = $(DB_NAME);`
- `DB_HOST = localhost`.

Capitolo 5

Limitazioni e bug

5.1 Considerazioni sullo stato attuale del progetto

MacroPHPagus è un software che sta attualmente muovendo i suoi primi passi: nato da una costola del progetto **Spotlight**, sta ora seguendo una propria linea di sviluppo indipendente allo scopo di renderlo effettivamente riutilizzabile per altre applicazioni, efficiente e possibilmente indipendente dal sistema sottostante.

In effetti, a ben vedere, un progetto come questo —ancorché ambizioso e fondato su intuizioni non banali— deve affrontare un certo numero di difficoltà, quantomeno per "sopravvivere": anzitutto va considerato che esso viene interamente portato avanti da una sola persona, come strumento di supporto nello sviluppo del già citato **Spotlight**; questo ovviamente comporta che non possa ricevere una completa dedizione da parte del suo Autore. Il problema non è da poco, dato che il mondo del software per PHP si evolve in maniera molto veloce, ricevendo supporto da molti soggetti con possibilità tecniche ed economiche ben maggiori di quelle di un singolo sviluppatore indipendente.

È convinzione dell'Autore che **MacroPHPagus** possa avere maggiore successo per le idee che propone o ribadisce, piuttosto che non per la

sua effettiva implementazione; ciò non toglie che il suo sviluppo debba continuare, portando nel prossimo futuro alla sua pubblicazione in Internet con una licenza libera. Questo allo scopo di verificare l'interesse da parte della comunità rispetto al modello di sviluppo proposto, e magari ricevere contributi per la sua evoluzione.

Comunque, indipendentemente da ciò che il futuro riserverà per questo progetto, è opportuno rilevare quali siano ad oggi le limitazioni dell'attuale implementazione di **MacroPHPagus**, discutendone successivamente gli spunti di sviluppo. Nelle prossime sezioni e nel capitolo successivo saranno illustrati in dettaglio questi aspetti.

5.1.1 Dipendenza da Unix/Linux

Come già visto precedentemente, **MacroPHPagus** si basa su di uno strumento di sviluppo storico di Unix/Linux, cioè *Make*. Ciò comporta problemi di accoppiamento con il suo modello di sviluppo : Make è un programma pensato principalmente per la costruzione di programmi in linguaggi di programmazione (compilati) tradizionali, particolarmente C, C++ e Fortran, e necessita di adattamenti per poter lavorare con altri linguaggi e per supportare catene di sviluppo diverse da quella che prevede la compilazione di file sorgente e la produzione di codice eseguibile da installare nelle apposite directory presenti nei *filesystem* di Unix.

Inoltre la presenza di script della shell contenenti invocazioni ad altri programmi, anche essi legati a Unix/Linux, rende difficoltoso l'uso di **MacroPHPagus** su altre piattaforme, come *Windows*, a meno di non ricorrere all'installazione di scomodi e pesanti strati di compatibilità come *Cygnus*.

Meglio sarebbe migrare verso sistemi più strettamente legati a PHP e meno al sistema operativo, come *Phing*[1], che, stando alla presentazione sul suo sito, dovrebbe offrire una serie di caratteristiche in parziale sovrapposizione con **MacroPHPagus**.

[1] *Phing* è uno strumento di sviluppo analogo a Make (ma più vicino per concezione ad *Apache Ant*), specifico per la creazione di applicazioni PHP.

5.1.2 Struttura e prestazioni

Sebbene non siano ancora stati effettuati test approfonditi di **MACRO-PHPAGUS** su grandi quantità di dati, è possibile già da ora affermare che l'attuale implementazione soffre di alcuni problemi strutturali abbastanza seri.

Anzitutto l'uso di M4 come "compilatore" dei file sorgente sembra non essere la scelta migliore, considerando le possibilità di evoluzione del progetto: il suo codice è confusionario e difficilmente mantenibile, specie per grandi programmi, e manca di alcune strutture fondamentali dei linguaggi di programmazione.

Anche i file sorgente risentono dei problemi di M4, ed è per questo che è stato necessario effettuare un pre-processamento per delimitare i caratteri speciali e scrivere in maniera molto accurata le macro per evitare ripercussioni sulla formattazione del risultato. Nonostante ciò l'*output* di M4 deve essere ulteriormente filtrato per emendare il codice PHP di alcune sequenze di caratteri estranee al linguaggio[2].

5.1.3 Costrutti e pattern supportati

Sebbene l'uso di modelli preimpostati consenta allo sviluppatore di scrivere più rapidamente quantomeno la struttura delle proprie classi, si nota che ancora si deve inserire *tutto* il codice PHP dei metodi, compresi i rientri, variabili, costanti ed altri costrutti: ciò significa avere un miscuglio di linguaggi senza regole precise, soggetto ad errori dei più svariati tipi.

Altro problema strettamente collegato è che non si possono generare test delle classi prima dell'implementazione dei metodi e la generazione del codice PHP, così come non è possibile ancora indicare quali operazioni di test vanno effettuate per i singoli metodi.

[2] A questo proposito è tuttavia utile osservare che tali sequenze sono state prudentemente delimitate come commenti, in modo da non creare errori di sintassi PHP.

Non esiste una gestione né il test delle eccezioni che possono essere lanciate di volta in volta dai vari metodi.

Quanto a pattern e frammenti di codice ricorrenti, si può dire che ad oggi l'unica vera implementazione è quella del pattern *Singleton*, mentre è scarsamente supportata l'inclusione automatica di file PHP tramite la funzione `__autoload()`, di cui è necessaria una revisione profonda.

Da notare che per alcuni pattern esistono dei flag riservati, i quali però non producono modifiche al codice finale, ma hanno solo uno scopo informativo.

Mancano inoltre dei modelli di *mapper* degli oggetti persistenti ed il corrispondente codice SQL per la generazione di uno schema di DB.

5.1.4 Dipendenza da `phpunit`

MacroPHPAgus si avvale attualmente dello script `phpunit` per generare lo scheletro dei test di una determinata classe.

Sebbene ciò abbia il pregio di avere a disposizione una comoda funzionalità senza necessità di scrivere codice aggiuntivo, esiste un prezzo da pagare in termini di flessibilità nonché di affidabilità del programma esterno.

Infatti la versione di PHPUnit utilizzata è la 2.2.1, che è attualmente piuttosto datata, e contiene alcuni bug non di poca importanza. Dal punto di vista di **MacroPHPAgus**, la cosa peggiore che può capitare è che il programma termini bruscamente senza dare messaggi di errore e con uno stato di uscita pari a 255, che non risulta documentato.

In tal caso lo sviluppatore può attraversare tempi duri prima di scoprire che forse un errore di sintassi nascosto da qualche parte nel codice PHP della classe mandi in confusione `phpunit`, che non sapendo come comportarsi non trova niente di meglio che "commettere suicidio" senza dire perché. Evidentemente non è questo che ci si aspetta da un framework che voglia offrire funzionalità ad alto livello, facili e rapide da usare.

5.2 Bug noti

5.2.1 Conflitti con M4

Come già osservato nella sezione 5.1.2, l'uso di M4 si rivela alla lunga una scelta infelice. La scarsa chiarezza del suo linguaggio e le sue bizzarrie sull'uso dei delimitatori (apici, virgolette o quant'altro), rendono difficoltoso scrivere e leggere programmi complessi che implementino tutte le funzionalità di **MacroPHPagus**.

Un esempio emblematico dei difetti di M4 è il seguente: il suo linguaggio non prevede la presenza di cicli iterativi come il ciclo `for`, ma ne rende possibile l'implementazione in maniera ricorsiva.

Osservando la macro che lo definisce [3] si può avere un'idea della difficoltà della sintassi di M4:

```
# forloop(var, from, to, stmt)
define([[forloop]],
  [[pushdef([[$1]], [[$2]])_forloop([[$1]], [[$2]], [[$3]],
    [[$4]])popdef([[$1]])]])
define([[_forloop]],
  [[$4[[]]ifelse($1, [[$3]], ,
    [[define([[$1]], incr($1))_forloop([[$1]], [[$2]],
      [[$3]], [[$4]])]])]])
```

Da notare l'uso delle doppie parentesi quadre attorno ai parametri, scelta imposta per evitare confusione con i vari delimitatori usati da PHP.

Come si vede la leggibilità di questo frammento di codice è piuttosto bassa, e la notevole "sensibilità" di M4 alle stringhe non virgolettate ha creato spesso situazioni in cui la macro in questione entrava in un ciclo infinito, bloccando il processamento senza messaggi di errore.

[3] La macro `forloop` è definita nel file `macros.m4`, ed è un riadattamento di quella omonima fornita con la distribuzione di M4.

5.2.2 Assenza di controllo sugli errori

Alla luce di quanto detto finora, diviene evidente come sia facile commettere errori nella scrittura di file sorgente, specialmente in progetti molto grandi, e trovarsi nelle situazioni in cui il processamento va a buon fine, ma produce risultati indesiderati, o che M4 termini con messaggi di errore che dicono poco o punto sulle reali cause.

Ciò avviene perché non è stato definito formalmente un linguaggio né delle vere e proprie regole di codifica, e le macro definite non effettuano in genere controlli sui parametri passati in ingresso, fatta eccezione per alcuni casi, descritti nella sezione 4.2.1.

Allo stesso modo non sono effettuate verifiche di particolari condizioni, come la ridefinizione di macro esistenti, l'inserimento di parole chiave di M4 nel codice PHP, ed altro.

Capitolo 6

Sviluppi futuri di MacroPHPagus

6.1 Obiettivi ed aspettative

Ormai risulta chiaro che **MacroPHPagus** è un progetto piuttosto ambizioso, che aspira ad assistere lo sviluppatore nel numero maggiore possibile di fasi del ciclo di vita di un'applicazione PHP. Proprio per questo motivo ci sono diverse strade aperte, più o meno pervie, per la sua evoluzione.

Una possibilità altamente desiderabile sul lungo termine è che si vada verso la definizione di un vero e proprio *metalinguaggio* capace di descrivere applicazioni ad oggetti e tradurle, almeno in parte, nei linguaggi di programmazione più diffusi, distaccandosi quindi da PHP; in questo modo potrebbe ambire a divenire un *middleware* tra UML ed i linguaggi di programmazione orientati agli oggetti, presumibilmente richiamando l'attenzione di coloro che attualmente sostengono xUML[1].

Un'altra prospettiva —più realistica sul breve termine— è che **MacroPHPagus** rappresenti la base concettuale per la creazione di un IDE basato su macro, corredato da un'API per PHP.

[1] *Executable **UML*** è una controversa metodologia di sviluppo che punta alla realizzazione di applicazioni interamente ed unicamente definite da diagrammi UML ed un qualche OCL, la cui traduzione in linguaggi concreti è completamente affidata ai compilatori. Va detto che sebbene tale prospettiva risulti piuttosto affascinante, suscita tuttavia perplessità tra gli "addetti ai lavori" (si vedano in proposito [13] e [15]).

Per ora è comunque importante delimitare meglio le sue caratteristiche e rendere più sicure le funzionalità che offre, stabilendo anche una scala di priorità sulle migliorie da apportare.

6.2 Una nota su versioni e revisioni

L'intero codice di **MacroPHPagus**ed i sorgenti LaTeX di questo documento sono conservati in un deposito CVS privato, allo scopo di tenere traccia delle modifiche effettuate volta per volta, e per la gestione delle versioni (*milestone* e rilasci) del software.

Le versioni sono numerate con la notazione

```
x.y.z[-{dev|alpha|beta}]
```

in cui

- `x` è il numero di versione principale, incrementato ad ogni revisione generale dell'architettura;
- `y` rappresenta un particolare rilascio della versione `x`, incrementato in caso di aggiunta o modifica di alcune funzionalità o aggiornamento di componenti dipendenti da software esterni, ma senza variazioni strutturali;
- `z` indica le revisioni di mantenimento; rimane bloccato a zero finché nella stringa della versione permane uno dei suffissi tra parentesi quadre, quindi viene incrementato in genere in caso di correzioni di *bug*;
- il suffisso `dev` viene applicato ad una versione dall'inizio della suo sviluppo fino ad una prima implementazione (in genere incompleta) di tutte le caratteristiche pianificate;
- successivamente si entra nella fase `alpha`, in cui si effettuano i primi test e si completano e si raffinano le implementazioni di tutte le componenti;

- quando tali operazioni sono ragionevolmente completate, il software passa alla fase `beta`, in cui le implementazioni vengono "congelate", cioè non subiscono più modifiche sostanziali, ma vengono sottoposte ad ulteriori test, e si procede all'eventuale correzione di bug scoperti;
- non appena si ritiene che l'attuale versione sia sufficientemente stabile, si eliminano tutti i suffissi, indicando che il software è pronto per il rilascio.

La versione di **MacroPHPagus** illustrata in questo documento è la `1.0.0-alpha`.

6.3 Generali

- *in corso* — integrazione di Zend Framework;

6.4 Processore dei file sorgente e linguaggio

- *rinviata (2.0)* — migrazione da Make a Phing;
- *rinviata (2.0)* — conversione dei file

  - `macros.m4`
  - `php5_templates.m4`
  - `*_generator.m4`

  da M4 a PHP;
- *rinviata (2.0)* — conversione degli script Bash

  - `makeConfig`
  - `makeUnitTests`

  in PHP;

- *rinviata (1.1)* — realizzazione della logica di generazione di schemi SQL a partire dagli oggetti del dominio;
- *rinviata (1.1)* — eliminazione della dipendenza da PHPUnit.

6.5 Modelli

- *rinviata (2.0)* — conversione dei file `*_tpl.m4` in PHP;
- *rinviata (1.1)* — creazione modelli di unit test PHP;
- *rinviata (1.1)* — creazione modelli di tabelle SQL;
- *in corso* — creazione modelli di classi *mapper* di oggetti persistenti;

6.6 API

- *in corso* — realizzazione di un framework per la persistenza degli oggetti basato su Zend Framework;
- *in corso* — completamento dell'implementazione delle collezioni di oggetti ed integrazione con la SPL di PHP 5;
- *in corso* — integrazione della gerarchia di eccezioni con quella della SPL;
- *rinviata (1.1)* — sviluppo di un adattatore per l'accesso a database tramite ODBC e Zend Framework.

Capitolo 7

Conclusioni

Il progetto **MacroPHPagus** rappresenta un'interessante sfida che — come si è visto— va oltre gli scopi di questo testo, poiché si ritiene che ci siano ampi margini di miglioramento per uno strumento che parte da un approccio leggermente diverso da quello di altri software di sviluppo trovati in rete, ma che si è già dimostrato estremamente utile.

La sua caratteristica principale è quella di voler creare un ponte tra le fasi di progettazione e di implementazione definendo un linguaggio più semplice e più astratto di quello finale, con molti punti di "aggancio" con UML: in effetti sarebbe interessante poterlo integrare con gli strumenti CASE già esistenti, così da poter facilmente convertire diagrammi UML in una prima implementazione, che può essere già utile per creare in anticipo una pila di test, ed avere un'anteprima degli oggetti software concreti che si stanno realizzando.

Essendo inoltre già predisposto per la gestione della persistenza, e disponendo di librerie ad oggetti ad alto livello di astrazione, **MacroPHPagus** si propone di facilitare la programmazione anticipando e nascondendo il più possibile i dettagli implementativi del linguaggio finale.

Per ora c'è ancora una lunga strada da fare per giungere agli obiettivi proposti, ma è opinione dell'Autore che il punto di partenza sia buono soprattutto a livello concettuale, e che possa fornire spunti anche per il

miglioramento di altri strumenti di sviluppo esistenti, anche oltre la vita dell'implementazione presentata in questo testo.

Appendice A

API di PHP 5

A.1 Pacchetto system

A.1.1 Classe ConfigParser

Parser di semplici file di configurazione.

Il file di configurazione è un file di testo, con un formato simile ai file delle proprietà di Java. Esso consiste in una lista di coppie `nome = valore`. Se in una riga, in qualsiasi posizione, è presente il carattere #, tutto ciò che lo segue in tale riga sarà ignorato (utile per scrivere commenti).

Versione: `1.0.0-dev`;

Posizione: `ConfigParser.php`, riga 27.

Elenco delle variabili

- *array* `$configElements`

Elenco dei metodi

- *COSTRUTTORE* `__construct` ()
- *string* `getElemByName` (*string* `$elemName`)

- *array* `getElements` ()
- *array* `getElemsByGroup` (*string* `$groupName`)
- *void* `parseConfig` (*string* `$fileName`)
- **Map** `toMap` ()

Variabili

array `$configElements`	
Accesso	privato
Posizione	riga 43
Descrizione Array associativo contenente gli elementi del file di configurazione interpretato.	

Metodi

COSTRUTTORE `__construct` ()	
Accesso	pubblico
Posizione	riga 57
Descrizione Costruttore di default.	

string `getElemByName` (*string* `$elemName`)	
Accesso	pubblico
Posizione	riga 121
Descrizione Restituisce il valore corrispondente al nome passato in ingresso. Se l'elemento cercato non è presente, viene restituito il valore nullo.	
Parametri • *string* *`$elemName`* Nome dell'elemento di cui si vuole ottenere il valore corrispondente.	

array `getElements` ()	
Accesso	pubblico
Posizione	riga 108
Descrizione Restituisce un array con la lista degli elementi validi trovati nel file.	

array `getElemsByGroup` (*string* `$groupName`)	
Accesso	pubblico
Posizione	riga 135
Descrizione Restituisce l'insieme degli elementi appartenenti al gruppo specificato. Se il gruppo richiesto non esiste, viene restituito un valore nullo.	
Parametri • *string* `$groupName` Nome del gruppo di elementi che si vuole recuperare.	

void `parseConfig` (*string* `$fileName`)	
Accesso	pubblico
Posizione	riga 75
Descrizione Restituisce un array con la lista degli elementi validi trovati nel file.	
Parametri • *string* `$fileName` Nome del file da di configurazione da interpretare.	

Map toMap ()	
Accesso	pubblico
Posizione	riga 155
Descrizione Esporta il contenuto del file di configurazione in una mappa associativa. Da notare che questo metodo crea solo una mappa lineare, cioè inserisce tutte le chiavi valide del file di configurazione ed associa ad esse i corrispondenti valori, ignorando eventuali notazioni puntate nelle chiavi.	

A.1.2 Classe IOException

Eccezione relativa alle operazioni di I/O.

Versione: `1.0.0`;

Posizione: `IOException.php`, riga 27;

Estende: Exception.

A.1.3 Classe Map

Implementazione PHP 5 di una mappa associativa.

PHP fino alla versione 5 sembra non avere ancora una gestione adeguata delle collezioni di oggetti, affidandosi quasi esclusivamente al tipo interno array.

Tale tipo risulta inadeguato sotto molti aspetti: anzitutto non è "orientato agli oggetti", impone limiti sul tipo usato per le chiavi, e inoltre richiede operazioni relativamente di basso livello per la sua manipolazione.

Questa classe rappresenta un primo tentativo di superamento di tali limitazioni, fungendo da wrapper per il tipo *array*: la sua interfaccia si ispira apertamente a java.util.Map, e permette di creare e manipolare facilmente mappe hash all'interno di applicazioni orientate agli oggetti, trattandole come oggetti software, ignorandone i dettagli implementativi.

Versione: `1.0.0-alpha`;

Posizione: `Map.php`, riga 31;

Implementa: COUNTABLE[1].

ELENCO DELLE VARIABILI

- *boolean* `$isEmpty`
- *array* `$map`

ELENCO DEI METODI

- *COSTRUTTORE* `__construct` ()
- *void* `clear` ()
- *boolean* `containsKey` (*mixed* `$key`)
- *boolean* `containsValue` (*mixed* `$value`)
- *integer* `count` ()
- *mixed* `get` (*mixed* `$key`)
- *boolean* `isEmpty` ()
- *void* `put` (*mixed* `$key`, *mixed* `$value`)
- *void* `putAll` (*array* `$a`)
- *void* `remove` (*mixed* `$key`)
- ***DEPRECATO*** — *integer* `size` ()

[1] Interfaccia appartenente alla SPL, descritta in [12].

Variabili

boolean `$isEmpty`	
Accesso	privato
Posizione	riga 54
Descrizione	
Flag di mappa vuota.	

array `$map`	
Accesso	privato
Posizione	riga 47
Descrizione	
Array associativo usato per la rappresentazione interna della mappa.	

Metodi

COSTRUTTORE `__construct` **()**	
Accesso	pubblico
Posizione	riga 68
Descrizione	
Crea una mappa vuota.	

void `clear` **()**	
Accesso	pubblico
Posizione	riga 85
Descrizione	
Rimuove tutte le associazioni dalla mappa.	

boolean `containsKey` (*mixed* `$key`)	
Accesso	pubblico
Posizione	riga 98
Descrizione Controlla se la mappa contiene un'associazione per la chiave indicata.	
Parametri • *mixed* `$key`	

boolean `containsValue` (*mixed* `$value`)	
Accesso	pubblico
Posizione	riga 108
Descrizione Controlla se la mappa contiene il valore indicato.	
Parametri • *mixed* `$value`	

integer `count` ()	
Accesso	pubblico
Posizione	riga 118
Descrizione Restituisce il numero di associazioni chiave–valore presenti nella mappa.	

mixed `get` (*mixed* `$key`)	
Accesso	pubblico
Posizione	riga 133
Descrizione Restituisce l'elemento corrispondente alla chiave in ingresso. Da notare che il metodo restituisce `null` sia se la chiave indicata non esiste nella mappa, che se la chiave esiste ma è associata al valore nullo. Usare `containsKey()` per distinguere tra i due casi.	
Parametri • *mixed* `$key`	

boolean `isEmpty` ()	
Accesso	pubblico
Posizione	riga 148
Descrizione Controlla se la mappa non contiene associazioni.	

void `put` (*mixed* `$key`, *mixed* `$value`)	
Accesso	pubblico
Posizione	riga 158
Descrizione Associa il valore indicato alla chiave indicata.	
Parametri • *mixed* `$key` • *mixed* `$value`	

void `putAll` (*array* `$a`)	
Accesso	pubblico
Posizione	riga 173
Descrizione Inserisce nella mappa il contenuto dell'array passato come parametro (chiavi e corrispondenti valori). È importante notare che le associazioni già presenti nella mappa non saranno sovrascritte da eventuali elementi dell'array contenenti chiavi esistenti.	
Parametri • *array* `$a` Array da inserire nella mappa.	

void `remove` (*mixed* `$key`)	
Accesso	pubblico
Posizione	riga 185
Descrizione Rimuove l'associazione relativa alla chiave indicata.	
Parametri • *mixed* `$key`	

integer `size ()`	
DEPRECATO	
Usare `count()`, per compatibilità con la SPL.	
Accesso	pubblico
Posizione	riga 118
Descrizione	
Restituisce il numero di associazioni chiave–valore presenti nella mappa.	

Appendice B

Modelli di file di macro

B.1 class_tpl.m4

```
# GNU M4 macro template for PHP 5 class generation.

# Flags are useful for specifying some of the internal
# modifiers or features of the class (extensibility, patterns
# applied, ...); some of them are used to generate the
# appropriate PHP code.
# Currently used flags follow:
# A: Abstract
# C: Controller/Façade {pattern}
# D: Adapter {pattern}
# E: Exception
# F: Factory {pattern}
# L: Leaf class
# P: Persistent object
# S: Singleton {pattern}
define([[FLAGS]], [[ACDEFLPS]])

# defines the mapper type used to persist the current object.
```

```
# Recognized/reserved arguments are listed below:
# FileTxt: text file
# FileXML: XML document (file)
# LDAP: LDAP directory
# ODB: Object Oriented DBMS
# RDB: [object] relational DBMS
define([[TMAPPER]],[[FileTxt|FileXML|LDAP|ODB|RDB]])

# This maps to the '@author' PHPDoc tag.
define([[AUTHOR]],
  [[Name "nickname" Surname  <email@address>]])

# This maps to the class '@package' PHPDoc tag.
define([[PACKAGE]], [[package]])

# This maps to the class '@version' PHPDoc tag.
define([[VERSION]], [[x.y.z]])

# uncomment this to set the '@deprecated' tag for this class.
# define([[DEPRECATED]])

# This maps to the class DocBlock short description.
define([[DESCRIPTION]], [[]])

# This maps to the class DocBlock long description.
define([[COMMENT]], [[dnl
]])

# uncomment this to enable the creation of a unit test for
# this class.
#define([[ENABLE_TEST]])
```

```
# To be defined only and only if this is a child of another
# class.
#define([[_EXTENDS_]], [[SuperClass]])

# Comma-separated list of implemented interfaces.
#define([[_IMPLEMENTS_]], [[Interface1, Interface2, ...]])

# ## DEPRECATED ##
# uncomment the following macro to use the PHP __autoload()
function.
# define([[AUTOLOAD]])

# Comma-separated list of PHP files to be included from
# 'system' directories (e. g. PEAR packages).
# define([[_SYSINCLUDE_]],
#   [[file1.php, dir1/file2.php, ...]])

# Comma-separated list of PHP files to be included from
# 'user' directories, i. e. those files belonging to the
# current application, under
# $_SERVER['DOCUMENT_ROOT']/<app_name>.
# define([[_USRINCLUDE_]],
#   [[file1.php, dir1/file2.php, ...]])

define([[CONSTANTS]], [[dnl
//&]])

define([[PROPERTIES]], [[dnl
//&]])
```

```
define([[CONSTRUCTOR]], [[dnl
//&]])

define([[METHODS]], [[dnl
//&]])
```

B.2 interface_tpl.m4

```
# This maps to the '@author' PHPDoc tag.
define([[AUTHOR]],
  [[Name "nickname" Surname  <email@address>]])

# This maps to the class '@package' PHPDoc tag.
define([[PACKAGE]], [[package]])

# This maps to the class '@version' PHPDoc tag.
define([[VERSION]], [[x.y.z]])

# uncomment this to set the '@deprecated' tag for this class.
# define([[DEPRECATED]])

# This maps to the class DocBlock short description.
define([[DESCRIPTION]], [[]])

# This maps to the class DocBlock long description.
define([[COMMENT]], [[dnl
//&]])

# Comma-separated list of implemented interfaces.
#define([[_IMPLEMENTS_]], [[Interface1, Interface2, ...]])
```

```
define([[METHODS]], [[dnl
//&]])
```

B.3 exception_tpl.m4

```
define([[FLAGS]], [[E]])

# This maps to the '@author' PHPDoc tag.
define([[AUTHOR]],
  [[Name "nickname" Surname  <email@address>]])

# This maps to the class '@package' PHPDoc tag.
define([[PACKAGE]], [[package]])

# This maps to the class '@version' PHPDoc tag.
define([[VERSION]], [[x.y.z]])

# This maps to the class DocBlock short description.
define([[DESCRIPTION]], [[]])

# This maps to the class DocBlock long description.
define([[COMMENT]], [[dnl
//&]])

define([[_EXTENDS_]], [[Exception]])

define([[PROPERTIES]], [[dnl
//&]])

define([[METHODS]], [[dnl
public function __toString() {
```

```
    return __CLASS__
     . ": [{$this->code}]: {$this->message}\n";
}
//&]])
```

Appendice C

Glossario

Sono qui descritti i termini standard usati nel testo.

Framework: nell'ingegneria del software, un *framework* consiste di una struttura di supporto software che definisce linee guida e strumenti per la creazione di applicazioni con particolari caratteristiche, come compatibilità con uno o più standard, implementazione di pattern architetturali, e quant'altro.

***Hot–spot*:** installazione di uno o più apparati per la connessione ad Internet tramite rete locale *wireless*.

IDE: acronimo di ***I**ntegrated **D**evelopment **E**nvironment*, ovvero *Ambiente integrati di sviluppo*. Si tratta di uno strumento che assiste uno o più sviluppatori nella realizzazione di applicazioni software.

LAMP: acronimo usato per le applicazioni web basate su **L**inux, ***A**pache HTTP Server*, ***M**ySQL*, e **P**HP oppure **P**erl oppure **P**ython.

Materializzazione: creazione ed inizializzazione di un oggetto software a partire dai dati letti da un supporto di memorizzazione.

Modello: nel contesto di **MacroPHPagus**, si tratta di un file parzialmente compilato contenente lo scheletro di un file di macro che descrive

un oggetto software in PHP. Lo sviluppatore deve copiare e personalizzare tale file per creare le classi e le interfacce della propria applicazione a oggetti.

PDO: acronimo di ***P**HP **D**ata **O**bjects*.

RPM: acronimo ricorsivo di ***R**PM **P**ackage **M**anager*. È un sistema di gestione di pacchetti software per Linux e Unix: il suo nome si riferisce tanto al formato dei file contenenti tali pacchetti, quanto al programma che si occupa di crearli, installarli, disinstallarli o aggiornarli.

Serializzazione: salvataggio dello stato di un oggetto software su un supporto di memorizzazione.

Template: usato come sinonimo di **Modello** (v.).

***Wi–Fi*:** Nome commerciale delle reti locali senza fili basate sulle specifiche *IEEE* 802.11.

Bibliografia

[1] *Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone*: **Basi di dati**, seconda edizione, McGraw–Hill, 1999.

[2] *Scott W. Ambler*: **The design of a robust persistence layer for relational databases**, 2005, `http://www.ambysoft.com/essays/persistenceLayer.html`.

[3] *Scott W. Ambler*: **Enterprise-ready Object IDs**, 2001, `http://www.ddj.com/dept/architect/184415770`.

[4] *AA. VV.*: **GNU `make`**, 2003, `http://www.delorie.com/gnu/docs/make/make_toc.html`.

[5] *Craig Larman*: **Applying UML and patterns**, seconda edizione, Prentice Hall, 2002.

[6] *AA. VV.*: **MySQL 3.23, 4.0, 4.1 reference manual**, 2006, `http://dev.mysql.com/doc/refman/4.1/`.

[7] *Scott W. Ambler*: **Mapping objects to relational databases: O/R Mapping in detail**, 2005, `http://www.agiledata.org/essays/mappingObjects.html`.

[8] *Daniel Convissor, Martin Jansen, Alexander Merz*: **PEAR manual**, 2007, `http://pear.php.net/manual/`.

[9] *Jack D. Herrington*: **PHP trucchi e segreti**, O'Reilly/Hops - Tecniche nuove, 2006.

[10] *AA. VV.*: **PHP manual**, 2007, `http://www.php.net/manual/`.

[11] *Jack D. Herrington*: **The PHP scalability myth**, 2003, `http://www.onjava.com/pub/a/onjava/2003/10/15 /php_scalability.html`.

[12] *AA. VV.*: **SPL — Standard PHP Library**, 2007, `http://www.php.net/ helly/php/ext/spl/`.

[13] *Scott W. Ambler*: **Be realistic about the UML: it's simply not sufficient**, 2006, `http://www.agilemodeling.com/essays/realisticUML.htm`.

[14] *Scott W. Ambler*: **A UML profile for data modeling**, 2006, `http://www.agiledata.org/essays /umlDataModelingProfile.html`.

[15] *Martin Fowler*: **UML Distilled: Guida rapida al linguaggio di modellazione standard**, terza edizione, Addison–Wesley/Pearson Education Italia, 2006.