

Usi errati dell'ereditarietà in progetti reali* - (Parte prima)

L'articolo presenta alcuni esempi di progettazione ad oggetti nei quali viene fatto un uso errato dell'ereditarietà. Si vuole in questa sede discutere sull'opportunità d'impiegare una particolare forma d'ereditarietà, cercando di proporre di volta in volta eventuali soluzioni alternative.

Dr. Andrea Baruzzo

L'ereditarietà è una delle caratteristiche più importanti della programmazione orientata agli oggetti e contribuisce senza ombra di dubbio a caratterizzare questo paradigma, differenziandolo ad esempio da quello procedurale. Non a caso, l'ereditarietà è anche uno dei primi concetti che vengono introdotti in qualsiasi corso di object-orientation, sia parlando di Java, sia trattando altri linguaggi come il C++ (a cui il presente articolo farà riferimento). Diversi anni fa, quando il C++ iniziava lentamente a diffondersi anche a livello industriale, tale costrutto probabilmente più d'ogni altro venne accostato al "nuovo" modo di *pensare ad oggetti*. Nonostante diversi problemi "d'infanzia", come il supporto talvolta lacunoso di compilatori e *linker*, la mancanza di uno standard e la scarsa esperienza su vasta scala, quella "nuova" prospettiva di costruire artefatti più simili alle entità del mondo reale iniziò ad affermarsi. Tutta l'attenzione rivolta verso il processo di classificazione alla base del quale si fonda poi l'ereditarietà, indusse una parte di programmatori dell'epoca a credere che scrivere codice OO significasse soprattutto progettare gerarchie di classi. Questa convinzione rimane tuttora abbastanza radicata, al punto da ingenerare dei veri e propri abusi. D'altronde molti sviluppatori oggi esiterebbero a battezzare come "object-oriented" un programma che non fa pesante uso di derivazioni e polimorfismo.

In questo articolo cercheremo invece di mostrare che diversi usi ritenuti "comuni" dell'ereditarietà sono invece palesemente errati e andrebbero del tutto evitati. Gli esempi discussi serviranno da traccia per individuare una serie di principi di progettazione sulla base dei quali tenteremo di modificare il design di partenza. Alcune situazioni sono tipiche del C++, ma la gran parte delle raccomandazioni presentate sono applicabili a qualsiasi linguaggio di programmazione orientato agli oggetti.

Codice dal mondo reale

Molte volte il codice che si discute in un articolo tecnico ricade all'interno di una classe di esempi stereotipati, la cui semplicità è essenziale in considerazione dello spazio contenuto messo a disposizione sulle pagine di una rivista. Nel tentativo di riportare poi quel codice alle situazioni reali spesso si rimane inevitabilmente con un senso di disagio, principalmente perché la soluzione analizzata risulta poco scalabile all'aumentare della complessità. Nell'economia di un articolo che tratta di progetti reali, questo mi è sembrato il primo limite sul quale lavorare. Di conseguenza, gli esempi che vedremo sono stati recuperati principalmente dall'ambito produttivo. I casi tratti dalla letteratura sono stati accuratamente selezionati, privilegiando quelli che garantissero un sicuro impatto in vere applicazioni e

non fossero solo una "cornice" ad una qualche tecnica di programmazione.

Uno degli aspetti di distinzione tra un "esempio giocattolo" e l'ambiente di sviluppo reale è sicuramente l'interazione tra più oggetti. Perdere questo elemento significherebbe ridurre drasticamente la portata delle raccomandazioni. Per fornire una minima struttura a tutto il materiale presentato è stato però necessario suddividere il lavoro in due puntate. Questo tipo di sforzo è, comunque, sicuramente più vicino a quello che compiamo ogni giorno nelle nostre attività di analisi e codifica. Riconoscere un particolare "schema" nel codice di un'applicazione reale è maggiormente oneroso, ma non di meno è un contributo importante al fine d'applicare concretamente la metodologia ad oggetti. È convinzione dell'autore, quindi, che per acquisire una buona padronanza nella progettazione sia necessario compiere quest'ulteriore passo, e tentare di rapportare le problematiche così individuate alla complessità del software moderno. Per far questo, è necessario lavorare spesso su più piani: da un lato sfruttare la modellazione del software per *organizzare* il design su larga scala, dall'altro applicare le tecniche e gli idiomi di programmazione che assicurano un'implementazione corretta dell'architettura su scala più piccola.

Il principio di sostituibilità

Prima d'iniziare con gli esempi, introduciamo un principio che ci guiderà lungo tutta la discussione. La forma più comune d'ereditarietà in C++ è quella pubblica. L'idea centrale si fonda sull'equivalenza che esiste tra una sottoclasse ed un sottotipo [Pes97]. È possibile dare una definizione operativa di cosa significhi derivare *pubblicamente*. In particolare, diremo che (una classe derivata) B è un sottotipo di (una classe base) A se e solo se può essere usata al posto di quest'ultima in ogni circostanza in cui ci si aspetti di avere un'istanza di A. Ecco allora che possiamo spiegare in modo estremamente naturale le relazioni d'ereditarietà esistenti, ad esempio, tra le classi "Mammifero" e "Cane". Un cane, infatti, è un mammifero. Questo tipo di relazione non a caso è stata battezzata nella terminologia ad oggetti come <IS-A>. Quando deriviamo pubblicamente, quindi, intendiamo dire che ogni asserzione valida per una classe base è applicabile anche alle classi derivate. In generale, il contrario non è necessariamente vero. Uno studente è una persona, ma non tutte le persone sono studenti. Il principio di sostituibilità di Liskov afferma questo concetto apparentemente semplice, ma che è in realtà più restrittivo di quanto sembri. Le prossime due osservazioni evidenzieranno alcune ragioni.

Può succedere che, creando una nuova classe derivata, introduciamo anche dei *nuovi vincoli*, ossia delle assunzioni non presenti fino a quel momento (e nei confronti delle quali le vecchie classi potrebbero essere del tutto impreparate). Altre volte, invece, può capitare di inserire un concetto all'interno di una gerarchia preesistente, principalmente per sfruttare una qualche forma di riuso, esponendoci di nuovo al rischio che quello stesso concetto possa non offrire le stesse garanzie fornite dagli altri elementi della gerarchia. Entrambe le situazioni appena descritte rappresentano un uso errato dell'ereditarietà pubblica,

* Questo documento è una versione estesa e rivista dell'articolo "Usi errati dell'ereditarietà in progetti reali - parte prima", pubblicato sulla rivista *Computer Programming*, rubrica Progettazione, n°108, Dicembre 2001 - Gruppo Editoriale Infomedia. Ultimo aggiornamento: novembre 2008.

come vedremo nei prossimi paragrafi. Affinché il principio di sostituibilità venga rispettato, dobbiamo garantire tre importanti proprietà:

1. Le precondizioni di ogni metodo ridefinito nelle classi derivate non devono richiedere più delle corrispondenti precondizioni nella versione della classe base. Semmai, potrebbero chiedere di meno, ma mai di più.
2. Le post-condizioni di un metodo nel contesto delle classi derivate devono assicurare almeno tanto quanto le corrispondenti post-condizioni nella versione della classe base.
3. Gli invarianti di una classe derivata non devono essere più restrittivi di quelli della corrispondente classe base, analogamente alle post-condizioni.

Cosa significa tutto ciò dal punto di vista del design di una gerarchia di classi? Vuol semplicemente dire che derivando non dobbiamo chiedere di più e non dobbiamo garantire di meno di ciò che richiediamo / garantiamo nel contesto della classe base. In altre parole, se vogliamo usare correttamente l'ereditarietà pubblica, dobbiamo derivare solo nei casi in cui estendiamo una classe oppure ridefiniamo una sua qualche funzione. Ogni altro uso implica una possibile restrizione della classe base [Pes97] [Sei95]. In questi ultimi casi, usare la classe derivata (più ristretta) al posto della classe base (più generale) è un'ovvia ma non così facilmente riconoscibile violazione del principio di sostituibilità.

Esempi di abuso dell'ereditarietà pubblica

Partiamo con un esempio che riflette un errore comune nella progettazione ad oggetti. Consideriamo un sistema di autenticazione degli accessi per una piccola rete. Ogni utente può accedere da un terminale e, in base ai privilegi che gli sono stati assegnati, è in grado di effettuare specifiche operazioni come leggere o scrivere su un "oggetto". Supponiamo di aver individuato tre tipologie di utenti ben distinte: i cosiddetti *ospiti*, gli *utenti privilegiati* e gli *amministratori* di sistema. Decidiamo allora di creare una classe base *NetworkAuthenticator* che astragga le funzionalità comuni del protocollo di verifica e deriviamo poi da questa tre classi derivate, ognuna delle quali risulta specializzata nell'autenticazione della corrispondente tipologia di utenti: *AdminAuthenticator*, *GuestUserAuthenticator* e

PowerUserAuthenticator (vedi **Figura 1**).

```
class NetworkAuthenticator {
public:
    NetworkAuthenticator(void);
    virtual ~NetworkAuthenticator(void);

    // operations
    bool BasicLogonControl(void);
    bool BasicObjectAccessControl(void);
    bool BasicTaskAccessControl(void);
    ...
};

class GuestUserAuthenticator :
public NetworkAuthenticator {
    GuestUserAuthenticator(void);
    ~GuestUserAuthenticator(void);

    // operations
    bool GuestLogonControl(void);
    bool GuestObjectAccessControl(void);
    bool GuestTaskAccessControl(void);
    ...
};

class PowerUserAuthenticator :
public NetworkAuthenticator {
    PowerUserAuthenticator(void);
    ~PowerUserAuthenticator(void);

    // operations
    bool PowerLogonControl(void);
    bool PowerObjectAccessControl(void);
    bool PowerTaskAccessControl(void);
    ...
};

class AdminAuthenticator :
public NetworkAuthenticator {...};
```

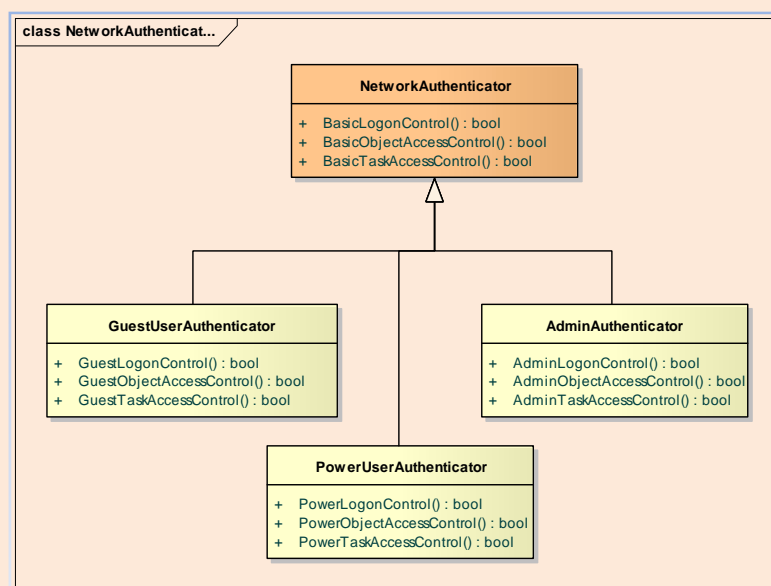


Figura 1 - La gerarchia NetworkAuthenticator

Possiamo pensare che `GuestLogonControl()` richiami a sua volta `BasicLogonControl()` e che, in generale, anche le altre funzioni di controllo sugli accessi richiamino le corrispondenti versioni nella classe base ogniqualvolta sia necessario eseguire un'operazione comune. Il fatto che l'autenticazione venga eseguita autonomamente nelle singole versioni a livello di classe derivata può far spacciare questo design come orientato agli oggetti. Il campanello d'allarme in quest'esempio consiste invece nell'osservare l'uso che abbiamo fatto dell'ereditarietà pubblica, speculando esclusivamente sulla possibilità di riutilizzare il codice fornito nella classe base. È giusto continuare ad utilizzare l'ereditarietà pubblica quando, esattamente per questo scopo (riutilizzo dell'implementazione) esiste in C++ l'ereditarietà privata? Come abbiamo visto, infatti, l'ereditarietà pubblica è una tecnica che ci consente di specializzare un comportamento comune mediante polimorfismo [Cii95]. Ciò che, nella mia esperienza, risulta spesso difficile mantenere separate è l'intenzione di *ridefinire* un comportamento comune dalla comodità di *riutilizzare* una parziale implementazione, e non l'operazione in quanto tale. Nell'esempio precedente, non c'è traccia di una simile intenzione in quanto non c'è ereditarietà d'interfaccia, ma solo d'implementazione: infatti non a caso il programmatore ha rinunciato ad utilizzare delle funzioni virtuali (distruttore a parte), chiaro segno che non si stava ponendo nell'ottica di un cliente esterno che invoca polimorficamente delle operazioni. Il contesto d'uso modellato è piuttosto quello delle interazioni tra le classi derivate e la corrispondente classe base. Parlando sempre d'interfaccia, si può anche obiettare sul fatto che non ci sia omogeneità tra i livelli diversi della gerarchia. Basti pensare alla necessità di manipolare separatamente i metodi `PowerObjectAccessControl()`, e `GuestObjectAccessControl()`, legando il codice chiamante ai dettagli delle singole classi derivate. Questo è un chiaro sintomo che non stiamo affatto pensando in termini di polimorfismo. Il problema non è solo una mera questione d'imparare tanti nomi diversi per una stessa funzionalità concettuale. Ci sono ripercussioni ben più importanti, per esempio sulla manutenzione, poiché simili gerarchie tendono a diventare piuttosto lunghe e decisamente più complesse del necessario.

Ovviamente non basta limitarsi ad aggiungere la parola riservata "virtual" davanti ad ogni metodo per trasformare il design in orientato agli oggetti. Abiliteremmo sì un aspetto meccanico, ma continueremmo a comunicare intenzioni di progetto piuttosto controverse, senza considerare il fatto che penalizzeremmo oltremodo le prestazioni complessive (poiché l'introduzione di metodi virtuali aumenta le dimensioni delle tabelle virtuali (v-table) necessarie ad implementare il polimorfismo). Quello che invece è necessario fare è ripensare in toto all'uso dell'ereditarietà come strumento di modellazione concettuale, riconsiderando attentamente le relazioni tra la classe base e quelle derivate, cercando di attuare il polimorfismo solamente se siamo in grado d'individuare un uso vantaggioso, oppure rinunciando del tutto ad esso, e ripiegando sull'ereditarietà privata, oppure su altre forme di riutilizzo del codice come il contenimento e la delegazione. Nel **Riquadro 1** riassumiamo alcune regole utili ad identificare simili errori. Osserviamo a tal proposito come la classe `NetworkAuthenticator` soffra di tutti i sintomi descritti nel **Riquadro 1**, il che avvalorava in modo sistematico l'ipotesi di un errore di progettazione.

Oggetti e ruoli

Alcune volte succede che non è il costrutto dell'ereditarietà in sé ad essere inadeguato, bensì la struttura gerarchica che si costruisce attorno alle diverse astrazioni in gioco. Ne deriva una

Riquadro 1 – Quattro regole per identificare un probabile uso errato dell'ereditarietà pubblica

Ereditare pubblicamente esclusivamente per riutilizzare il codice della classe base e non per invocare dei comportamenti comuni mediante polimorfismo costituisce una violazione del principio di sostituibilità. In tali casi, abbiamo almeno quattro indizi che possono essere individuati durante una revisione di codice. Se essi concorrono simultaneamente, evidenzieranno un sicuro abuso d'ereditarietà (pubblica). Vediamoli di seguito:

1. *La classe base non ha attributi, né metodi di tipo protected. Appare evidente la mancanza di una qualche "interfaccia per derivazione". Di per sé è già questo un buon motivo contro l'ereditarietà in quanto tale, indipendentemente dal fatto che si tratti di pubblica o privata [Sut00];*
2. *La classe base non fornisce nessun metodo virtuale, a parte eventualmente il distruttore. Si deduce quindi l'intenzione di non usare la classe in modo polimorfo;*
3. *La classe base incapsula delle funzionalità comuni, ma non rivela alcun'autonomia nei confronti di queste. Da ciò si deduce che la classe non fornisce un'astrazione abbastanza forte: in altre parole non evoca una vera relazione <IS-A> con le classi derivate;*
4. *Le classi derivate non sfruttano parti non pubbliche della classe base, al pari di un qualsiasi altro cliente esterno. La conseguenza logica è che potrebbero fare la stessa cosa anche se fossero legate alla classe base mediante una relazione meno forte dell'ereditarietà, come ad esempio il contenimento. Usare una relazione più forte del necessario è un errore di design.*

scarsa modellazione, principalmente perché manchiamo di riconoscere (e quindi rappresentare) uno specifico concetto che è parte essenziale del problema. L'esempio che mostriamo ora difetta di una non accurata descrizione del ruolo di un oggetto all'interno del sistema software.

Consideriamo un semplice framework per il controllo automatico di un processo. In generale, avremo a che fare con funzionalità a diversi livelli d'astrazione, coinvolgendo attività di misurazione, di gestione e d'interfacciamento con dispositivi di varia natura. Concentriamoci per il momento sulle prime due situazioni (riprenderemo l'esempio in seguito per discutere qualche aspetto di sincronizzazione).

Le misurazioni delle quantità controllate dal processo di controllo vengono acquisite da una serie di sensori. In base poi ai valori rilevati, uno o più attuatori vengono azionati. Un attuttore è un sottosistema che può essere usato per influenzare l'ambiente esterno di un sistema fisico. Tale definizione è volutamente generica: i confini che dividono ciò che è un attuttore da ciò che non lo è sono piuttosto confusi. Non è infrequente, tra l'altro, che uno stesso oggetto fisico sia utilizzato prima in un ruolo e poi in un altro, in base alle circostanze.

Tenendo conto di queste considerazioni, decidiamo di aggiungere al nostro framework le classi *Sensor* e *Actuator* (quasi certamente saranno delle classi base astratte, anche se per ora le manteniamo concrete). Se un utilizzatore volesse costruire una classe che funga talvolta da sensore e talvolta da attuttore, potrebbe aggregare i due oggetti appena costruiti in un unico oggetto *ActuSense*. In alternativa, si potrebbe impiegare l'ereditarietà multipla ed ottenere lo stesso effetto. Quest'ultima è la scelta che ci apprestiamo a discutere.

```
class ActuSense : public Actuator,
                 public Sensor
{
public:
    // constructors
    ActuSense(void);

    // operations
    // sensor's functionality
    // Measure abstracts the effect
    // of a measurement operation
    Measure GetMeasure(void);

    // actuator's responsibility
    void ActivateSubSys(int
                       controlSubSysId);

private:
    ...
};
```

ruolo in un dato momento un'istanza *ActuSense* stia svolgendo e, di conseguenza, non ci protegge da eventuali usi impropri.

```
// in some scope...
ActuSense actuSensor;
...
actuSensor.ActivateSubSys(nuclearPlantId);
    // is actuSensor in the "right" state
    // to support this operation?
```

Per raggiungere questo scopo è necessario modellare una classe ruolo, anziché basarci solo sull'interfaccia delle singole classi.

Un ruolo diventa un concetto esplicito del design quando è anche un chiaro concetto del dominio del problema. Possiamo considerare ogni ruolo come una specifica prospettiva assunta da un oggetto analizzato rispetto ad un preciso punto di vista (o prospettiva). Essendo essi stessi delle specializzazioni di concetti più generali, ai ruoli è applicabile l'usuale processo d'astrazione e, quindi, diventa ragionevole rappresentarli mediante delle classi [Kri96].

Dalle interfacce alle classi "ruolo"

Riconsideriamo allora le diverse entità in gioco. Abbiamo dei sensori, degli attuatori e vogliamo modellare la possibilità di far agire un componente in entrambi i ruoli, seppure in momenti diversi. Potremmo pensare d'introdurre nel dominio del nostro framework il concetto di *ComponentRole* che rappresenta l'astrazione "ruolo" dalla quale deriviamo due distinte classi:

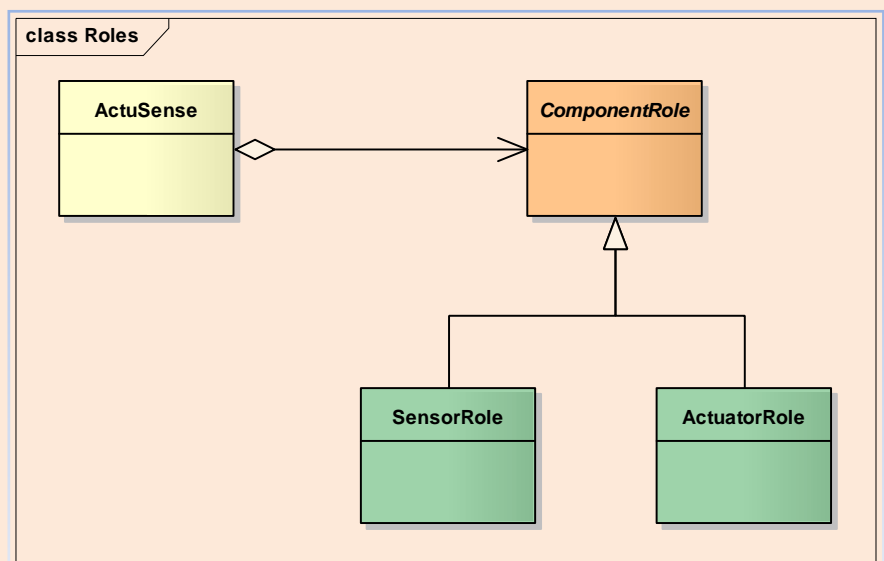


Figura 2 – Modellazione del concetto di ruolo

Uno dei più importanti principi che dobbiamo tenere in considerazione quando progettiamo l'interfaccia di un'astrazione è la separazione delle responsabilità (*separation of concerns*). Le attività di lettura di un valore riportato dal sensore (ad esempio una temperatura) e l'azionamento di un sottosistema preposto ad un qualche controllo (ad esempio un sistema di raffreddamento) sono due servizi la cui semantica è ben diversa, per cui andrebbero separati in interfacce diverse. Nel progetto della classe *ActuSense*, tuttavia, il design non rende esplicito quale

SensorRole e *ActuatorRole*. Otteniamo quindi il diagramma di classi di **Figura 2**. Abbiamo così legato la classe *ActuSense* con una relazione di aggregazione al concetto di ruolo. Sarà sufficiente creare di volta in volta un nuovo ruolo che rimpiazzerà il precedente ed usare l'oggetto *actuSensor* di conseguenza. Non avremo più entrambe le interfacce disponibili allo stesso momento, evitando usi inconsistenti.

Notiamo, inoltre, come questa particolare soluzione permette di risolvere elegantemente anche un altro tipo di problema: il

cambiamento dinamico di ruolo. Senza un'esplicita modellazione di tale concetto, quando un sensore funge da attuatore, dobbiamo creare un nuovo oggetto (di tipo *Actuator*) ed inizializzarlo con i dati della vecchia istanza di *Sensor*. Così facendo, però, violiamo il principio dell'identità di un oggetto [Sei95], poiché si tratta sempre dello stesso oggetto che agisce prima in un modo e poi in un altro. Il modello che abbiamo costruito ora rispetta tale principio e non richiede alcuna duplicazione di informazione. Un'ulteriore considerazione andrebbe fatta per quanto concerne i vincoli di molteplicità che dovremmo specificare per l'aggregazione. In altre parole, dovremmo chiederci se sia possibile, nel dominio considerato, avere un componente che riveste *contemporaneamente* i ruoli di attuatore e sensore (ad esempio, un componente composto). Quest'analisi ci potrebbe indurre non solo a specificare delle molteplicità, ma anche a considerare l'opportunità d'impiegare il pattern Composite [GOF95]. L'idea chiave nell'esempio proposto si basa sull'utilizzo dell'ereditarietà come meccanismo per strutturare il dominio del problema, delegando all'astrazione ruolo il compito di fornire una classificazione flessibile che può variare nel tempo.

Revisioni del codice: interfacce che rivelano abusi d'ereditarietà

Prendiamo spunto dal precedente esempio per esaminare un abuso che viene spesso commesso nel progetto di gerarchie di classi, ma che è anche possibile individuare facilmente durante una revisione. Il problema che qui vogliamo evidenziare è un'altra conseguenza della tendenza ad usare l'ereditarietà principalmente come forma di riutilizzo del codice. Parliamo, infatti, delle cosiddette "interfacce grasse" ("fat" interface), ossia di gerarchie basate su classi base le cui interfacce sono poco coese[†]. Consideriamo la situazione nella quale dobbiamo gestire diversi sensori e vogliamo registrare i segnali in ingresso su ciascuno di essi per un determinato intervallo di tempo. Introduciamo quindi una classe *Timer* che gestirà la sincronizzazione. Di seguito ne proponiamo una versione.

```
class Timer {
public:
    // register service
    void Register (int timeout,
                  int timerId,
                  TimerClient* pClient);
};

// handler of the Timer class for a client
class TimerClient {
public:
    // event notification for every client
    virtual void NotifyTimeout (int tmId)= 0;
};
```

La prima cosa che un cliente di *Timer* deve fare è registrarsi per mezzo della funzione *Register()*, specificando il periodo di registrazione ed un identificatore che lo assocerà univocamente ad una particolare istanza di *Timer*. Creiamo quindi la classe *TimerClient* che incapsula al suo interno il meccanismo di notifica degli eventi per i quali il cliente si è registrato. Il cliente, fornendo un riferimento a se stesso, permetterà alla classe *Timer* di

[†] Una ripetuta violazione del principio di separazione delle responsabilità può essere un chiaro sintomo di un'interfaccia grassa.

Riquadro 2 – Quando usare l'ereditarietà non pubblica

Riepiloghiamo alcune situazioni nelle quali è utile usare il costruito dell'ereditarietà non pubblica come scelta di design (ammesso che sia supportata dal linguaggio di programmazione utilizzato).

1. **Accesso ad un membro protetto di una classe.** Il riferimento qui è rivolto soprattutto alle funzioni (metodi) e non ai dati in quanto si presuppone che una classe sia incapsulata e, quindi, protetta dagli accessi esterni.
2. **Ridefinizione di una funzione virtuale.** Spesso capita di disporre di una classe (possibilmente base e astratta) che mette a disposizione almeno una funzione virtuale. Ereditare ed implementare un override di quel metodo virtuale è spesso l'unica possibilità che abbiamo per adattare alle nostre necessità il comportamento della classe. Volendo ridefinire una singola implementazione, è probabile che l'ereditarietà pubblica risulti inadeguata.
3. **Classi virtuali.** Può accadere di dover condividere una classe virtuale comune, come nel caso dell'ereditarietà multipla (ad esempio, una classe che eredita da un'altra avente le sue stesse basi virtuali).
4. **Costruttori e classi virtuali.** Tenendo presente che la classe maggiormente derivata è responsabile dell'inizializzazione di tutte le classi virtuali di partenza, se dobbiamo usare un nuovo costruttore altro non resta da fare se non ereditare (anche qui l'ereditarietà pubblica appare inadeguata).
5. **Life-time di oggetti e sotto-oggetti.** L'ereditarietà impone un ordine preciso sulla costruzione di un oggetto e dei suoi sotto-oggetti. Questa proprietà può essere sfruttata per garantire che un determinato oggetto venga creato prima dell'uso di un altro e distrutto dopo.
6. **Ottimizzazione della classe base vuota** [Mey97]. Questo è un caso interessante di uso dell'ereditarietà privata in chiave d'ottimizzazione. A volte, infatti, può capitare che una classe implementata in termini di un'altra non contenga attributi e che sia principalmente un wrapper sulle singole funzionalità. Quello che otteniamo è una classe vuota (senza attributi). Usare l'ereditarietà al posto del contenimento può comportare un risparmio di spazio poiché il compilatore è in grado di permettere che un sotto-oggetto base occupi uno spazio nullo. Se avessimo sfruttato la soluzione del contenimento, invece, ciò non sarebbe stato possibile, anche se la questi non contiene alcun dato. Va comunque osservato che attualmente non tutti i compilatori effettuano l'ottimizzazione per le classi vuote. Inoltre il beneficio che se ne può trarre è percepibile solo in presenza di tanti oggetti "vuoti" (nell'ordine delle decine di migliaia) [Sut00].
7. **Polimorfismo controllato.** Questa tecnica rappresenta probabilmente l'uso più frequente dell'ereditarietà protetta, in quanto rende maggiormente visibile nelle classi derivate l'intento di usare il polimorfismo.

notificare gli eventi d'interesse. Il meccanismo di propagazione degli eventi è del tutto analogo a quello descritto dal pattern *Observer*: l'operazione *Register()* attiva il timer e, ogniqualvolta il timeout scatterà, *Timer* notificherà l'evento al suo cliente servendosi del puntatore *pClient* registrato [GOF95].

Il nostro obiettivo iniziale era quello di registrare i segnali in ingresso su un sensore in un determinato periodo di tempo. Avendo nel nostro design già definito una classe *Sensor* che rappresenta in astratto ogni tipo di sensore, sembra naturale ora introdurre nella gerarchia un nuovo concetto che fungerà da "sensore temporizzato" e che chiameremo *TimedSensor*. È logico pensare di derivare pubblicamente quest'ultimo dalla classe base *Sensor* (in fin dei conti, un sensore temporizzato non è ancora un tipo di sensore?). A questo punto, un altro problema sorge: com'è possibile permettere ad un oggetto di tipo *TimedSensor* di poter sfruttare i servizi messi a disposizione da *TimerClient*? In altre parole, come può un oggetto *TimerClient* comunicare con un oggetto *TimedSensor*? Per supportare questo tipo di comunicazione, una possibilità consiste nel derivare *Sensor* direttamente da *TimerClient*. Quello che otteniamo è il diagramma di classi della **Figura 3**.

L'effetto della derivazione pubblica di *Sensor* nei confronti di *TimerClient* consiste nel far sì che l'interfaccia di quest'ultimo diventi parte integrante di quella di *Sensor* e delle sue derivate. Osserviamo che *TimedSensor* è adesso in grado di registrarsi come cliente di *Timer* in qualsiasi momento senza alcuno sforzo aggiuntivo. Tutto perfetto? Non proprio. Sebbene sia allettante sfruttare l'ereditarietà come strumento di riuso del codice, bisogna far fronte ora ad un aumento del livello d'accoppiamento tra le varie classi. Supponiamo che non tutti i tipi di sensori debbano usufruire dei servizi di *Timer*. Per questi rami di

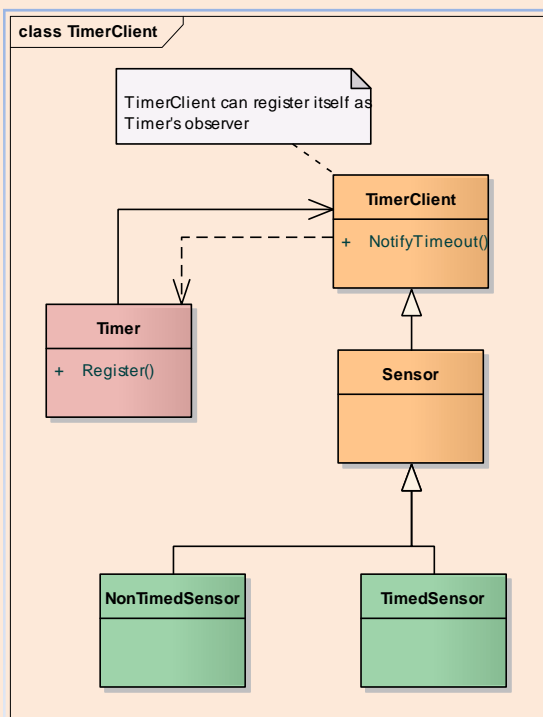


Figura 3 - Un esempio di ereditarietà usata come meccanismo di riuso del codice

derivazione, la porzione d'interfaccia relativa a *TimerClient* risulta del tutto inutile. Le cose sono destinate a peggiorare a fronte di cambiamenti nella classe *TimerClient* che si propagherebbero anche alle classi totalmente scorrelate dalle funzioni di *Timer* (tutto il ramo *NonTimedSensor*). In ultima analisi, è proprio questo tipo di ripercussioni che rendono la manutenzione e l'estensione così critica, soprattutto in progetti complessi. Per evitare tali difficoltà, spesso si finisce per implementare una modifica nel posto sbagliato, finendo col rendere ancor più ingarbugliato il codice. È preferibile quindi individuare simili situazioni prima ancora di realizzarle nei progetti a cui lavoriamo, anziché affidarsi a scorciatoie di dubbia opportunità. Un chiaro sintomo del problema delle "interfacce grasse" connesso all'ereditarietà è facilmente riconoscibile ogniqualvolta ci troviamo ad inserire una nuova funzione in una classe base per beneficiare solo una parte della gerarchia.

Il design pattern *adapter* come tecnica di separazione delle interfacce

Poiché la soluzione precedente inserisce a livello di classe base un'interfaccia che soddisfa solo una parte delle classi derivate, essa costituisce anche una palese violazione del principio di sostituibilità di Liskov. Si tratta dunque di un uso errato dell'ereditarietà pubblica (singola). Il nostro nuovo obiettivo sarà adesso quello di separare l'interfaccia di *TimerClient* da quella dei suoi clienti, preservando il riuso del codice solo per le classi interessate a questo servizio. Rivisitiamo allo scopo un noto design pattern: *Adapter*. *Adapter* è nato per risolvere il problema dell'incompatibilità di interfacce dal punto di vista di un cliente che deve invocare una funzionalità mediante una specifica interfaccia. Capita spesso che una funzionalità resa disponibile da una classe possa essere riusata in un contesto diverso. Tuttavia, la classe al momento della sua progettazione non può essere a conoscenza dei dettagli di tutti i possibili contesti d'uso. Si produce così un'interfaccia esposta al rischio d'essere incompatibile quantomeno con quella dei suoi nuovi utilizzatori. *Adapter* risolve tale problema in due modi:

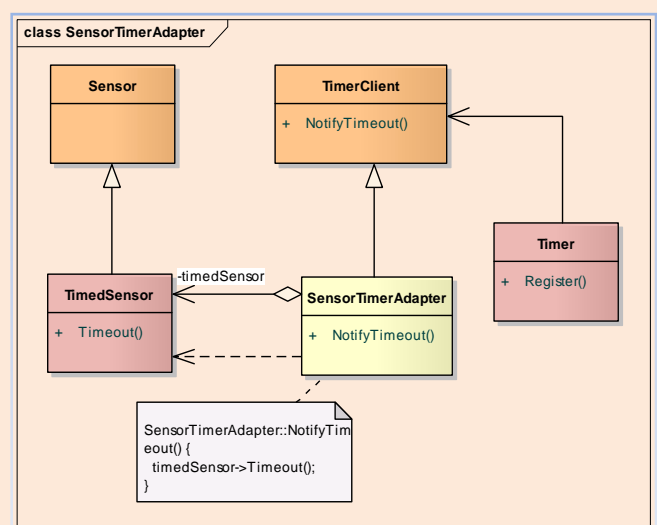


Figura 4 - Il pattern *Adapter* e la separazione delle interfacce

1. Impiegando l'ereditarietà multipla ("Adapter Class Form");
 2. Aggregando la classe che fornisce il servizio direttamente all'interno dei suoi clienti, i quali implementeranno la loro interfaccia in termini della prima ("Adapter Object Form");
- Presentiamo qui la seconda delle due versioni. Chi fosse interessato alla versione basata sull'ereditarietà multipla può consultare il testo "Design Pattern"**[GOF95]**. In questo nuovo design, quando un oggetto *TimedSensor* vuole sfruttare i servizi di temporizzazione forniti da *Timer*, collaborerà direttamente con un oggetto *SensorTimerAdapter*. Quando il timer notificherà l'evento di timeout, spedisce un messaggio a *SensorTimerAdapter* e questi, a sua volta, lo delegherà a *TimedSensor*.

```
// object form of adapter pattern
class TimedSensor : public Sensor {
public:
    virtual void Timeout (int timerId);
};

class SensorTimerAdapter : public TimerClient
{ public:
    SensorTimerAdapter (TimedSensor*
                        pTimedSensor):
        m_pTimedSensor (pTimedSensor) {}

    virtual void NotifyTimeout (int timerId) {
        // delegation to the inner object
        m_pTimedSensor->Timeout (timerId);
    }
    ...
private:
    TimedSensor* m_pTimedSensor;
};
```

Questa soluzione realizza il principio di separazione delle interfacce **[Mar96]** poiché mantiene separati i clienti di *Sensor* da *Timer*, proteggendoli da modifiche estranee al loro contesto. Un altro pregio di questo design è che non siamo obbligati ad avere in *TimedSensor* la stessa interfaccia di *TimerClient*: un aspetto caratterizzante il pattern Adapter (da cui, infatti, deriva il suo nome). Osserviamo com'è proprio la necessità di avere interfacce uguali a farci ragionare in termini "additivi" nei confronti dell'ereditarietà. Ogni qualvolta vogliamo supportare una nuova interfaccia siamo tentati di aggiungere questa alla classe base, sottoponendoci al rischio di incorrere nel problema delle "interfacce grasse". Il pattern in questione rappresenta un interessante tentativo di continuare a supportare le funzionalità aggiuntive, evitando di accoppiare direttamente i clienti ai fornitori. La **Figura 4** mostra il nuovo diagramma UML per l'implementazione basata su Adapter.

Conclusioni

In questa prima parte abbiamo discusso alcuni usi errati dell'ereditarietà pubblica, legati principalmente alla violazione di una "buona" regola di progettazione ad oggetti. In particolare, abbiamo affrontato la separazione delle interfacce, le classi ruolo e il principio di sostituibilità di Liskov che sintetizza il concetto di derivazione pubblica. Nella prossima (ed ultima) parte di questo articolo esamineremo altre forme d'ereditarietà come quelle virtuali, private e protette, cercando di evidenziare delle linee guida per ciascuna di loro. La prospettiva, ancora una volta, sarà quella del linguaggio C++ che permette di supportare tutte queste diverse forme di polimorfismo per sottotipo.

Bibliografia

- [Cli95] Cline, Marshall, et. al. - "C++ FAQs", Addison Wesley, 1995
- [GOF95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M. – "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995
- [Lak96] Lakos, John - "Large Scale C++ Software Design", Addison Wesley, 1996
- [Mar96] Martin, Robert - "The Interface Segregation Principle", C++ Report, June 1996
- [Meye97] Nathan, Meyers - "The Empty Base C++ Optimization", Dr. Dobb's Journal, Aug. 1997
- [Kri96] Kristensen, B. B., et. al. - "Roles: Conceptual Abstraction Theory and Practical Languages Issues", Special Issue on TAPOS in Object-Oriented Systems, 1996
- [Pes97] Pescio, Carlo - "Programmazione ad Oggetti e Programmazione Generica", Computer Programming N°62, Ottobre1997
- [Sei95] Seidewitz, Ed, et. al. - "Reliable Object-Oriented Software", SIGS Books, 1995
- [Sut00] Herb, Sutter - "Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions", Addison Wesley, 2000

Dr. Andrea Baruzzo

abaruzzo@computer.org

È laureato in Scienze dell'Informazione presso l'Università degli Studi di Udine.

Si occupa di ricerca, formazione e consulenza sia in ambito accademico, sia in ambito industriale. Le sue principali aree di interesse sono l'analisi, la progettazione e lo sviluppo di sistemi software ad oggetti (OOA/OOD/OOP), la qualità del software e le tecniche machine learning. È inoltre membro di IEEE Computer Society.