Design Patterns

Andrea Polini

Ingegneria del Software
Corso di Laurea in Informatica

Sommario

Riuso - generalità

Riuso tipico approccio ingegneristico alla costruzione di sistemi.
Sistemi risultano da integrazione di sottosistemi spesso non “banali”.

Motivazioni e benefici legati al riuso:

- ridotti costi di sviluppo,
- sviluppo da parte di specialisti,
- ridotti tempi di rilascio,
- aumentata qualità del software
- ridotto rischio del processo,

Riuso e tipiche problematiche

Problematiche tipiche nel riuso del software:

- Incremento nei costi di gestione del sistema
- Mancanza di strumenti di supporto al riuso
- Sindome del “NIH”
- Gestione di libreria di componenti
- Integrazione ed adattamento di componenti

Riuso - generalità

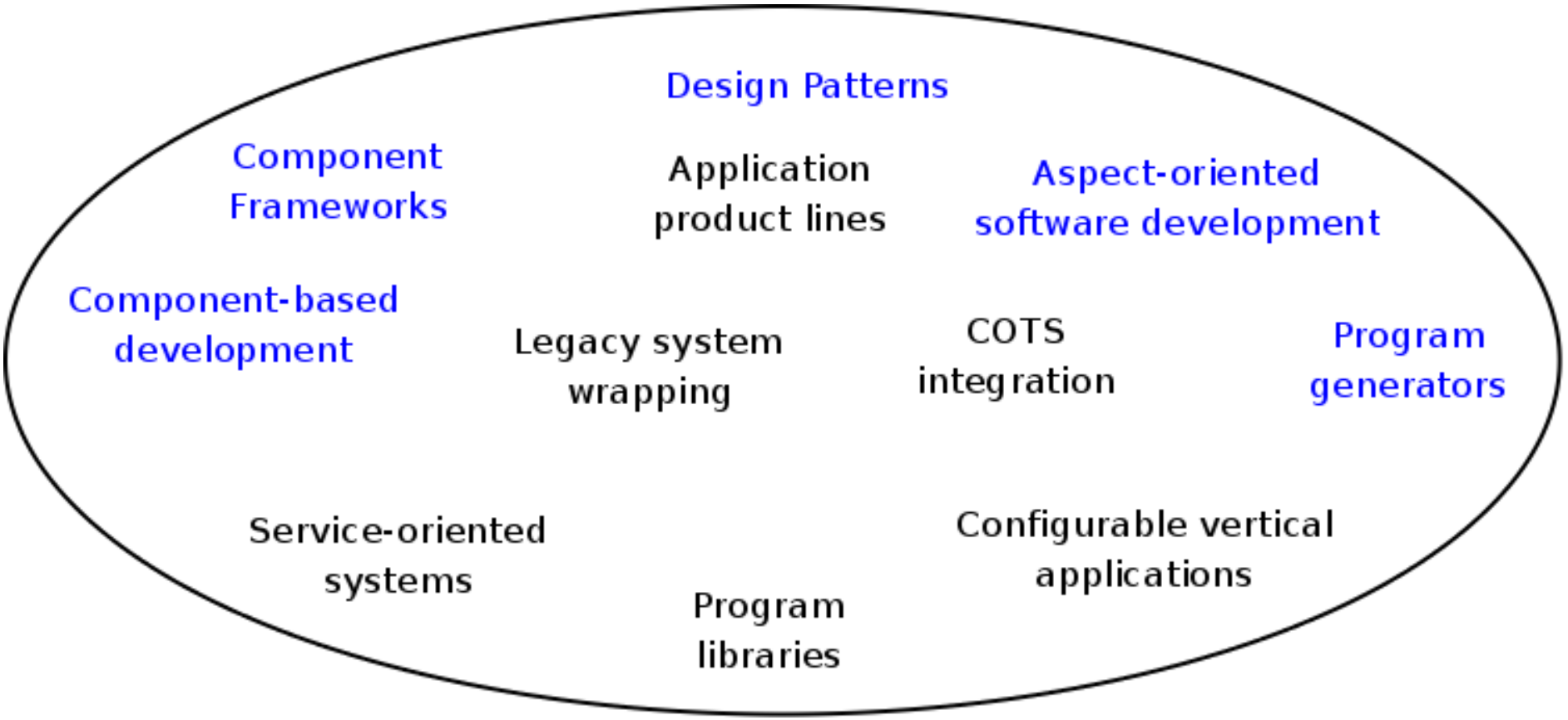
Riuso può interessare manufatti ma anche riuso a livello concettuale

Riuso di manufatti:

- Riuso di intere applicazioni
- Riuso di componenti e sottosistemi
- Riuso di oggetti e librerie

Sommario

Riuso - una panoramica



Quale approccio?

Fattori che possono guidare nella scelta di un approccio piuttosto che un'altro:

- Tempistiche dello sviluppo
- Attesa longevità dell'applicazione
- Conoscenze e capacità del team di sviluppo
- Caratteristiche di criticità del software
- Dominio applicativo
- Piattaforma di esecuzione

Design Patterns

Definizione: un pattern descrive un problema che ricorre spesso e propone una possibile soluzione in termini di organizzazione di classi/oggetti che generalmente si è rilevata efficace a risolvere il problema stesso.

Design Pattern sono caratterizzati da quattro elementi principali:

- Nome - riferimento mnemonico che permette di aumentare il vocabolario dei termini tecnici e ci permette di identificare il problema e la soluzione in una o due parole
- Problema - descrizione del problema e del contesto a cui il pattern intende fornire una soluzione
- Soluzione - descrive gli elementi fondamentali che costituiscono la soluzione e le relazioni che intercorrono tra questi
- Conseguenze - specifica le possibili conseguenze che l'applicazione della soluzione proposta può comportare. Si riferiscono ad esempio a possibili problemi di spazio o efficienza della soluzione, oppure ad applicabilità con specifici linguaggi di programmazione

Design Patterns e documentazione

Una volta ben definito e generalmente accettato da una communità di sviluppatori il pattern diventa anche un'ottimo strumento per documentare il software.

Esistono esempi di applicazioni interamente descritte attraverso l'uso di design pattern (e.g. JUnit)

Testo di Riferimento:
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Pattern: elements of reusable Object Oriented software
Addison Wesley

Materiale di studio a:
`http://it.wikipedia.org/wiki/Design_pattern`

Design Pattern

perché è una soluzione efficace?

Si riferisce ad un riuso di un'attività di progettazione dunque:

- Aspetto Economico: riduce i tempi ed i costi di progetto dei singoli moduli
- Aspetto Tecnico: riduce i rischi di progetto e possibilità di errori nel progetto stesso. Soluzione attuata e riutilizzata più volte e di cui si possono prevedere le caratteristiche a priori (e.g. comportamento non-funzionale)
- Aspetto Antropologico: semplifica la comprensione del progetto da parte di terzi fornendo livello di astrazione più chiaro

Come definire un pattern in modo da facilitarne il riuso

Descrizione dei Design Pattern secondo la *GoF*

La definizione di collezioni di pattern in uno specifico dominio applicativo sono il primo passo per poter stabilire un contesto di riuso
Nel libro della GoF i pattern sono descritti attraverso i seguenti punti:

- Nome: sintetizza l'essenza del pattern. Obiettivo è ampliare il vocabolario di progetto.
- Intento: Descrizione che cerca di rispondere alla domanda "cosa fa il DP?" "A quale problama di progettazione si rivolge?"
- Aka (Also known as): eventuali altri nomi che identificano il pattern
- Motivazioni: Scenario che illustra il problema e come il pattern fornisce una soluzione
- Applicabilità: situazioni di applicabilità e come poter riconoscere tali situazioni

Come definire un pattern in modo da facilitarne il riuso

Descrizione dei Design Pattern secondo la *GoF*

- Struttura: rappresentazione grafica tramite UML delle classi coinvolte e delle loro relazioni
  - Partecipanti: classi che partecipano al pattern, loro relazioni e responsabilità (class diagram, object ed activity diagram)
  - Collaborazioni: Come collaborano le varie classi per raggiungere gli obiettivi (sequence diagram)
- Conseguenze: vantaggi e svantaggi dell'uso del pattern e possibili effetti collaterali nell'uso del pattern
- Implementazione: tecniche che e suggerimenti per l'implementazione con riferimento anche a specifici linguaggi di programmazione
- Codice sorgente di esempio: frammenti di codice che forniscono una guida per l'implementazione
- Usi noti: esempi di uso in sistemi esistenti
- Patterns correlati: differenze e relazioni più importanti con altri pattern. Tipico uso concomitante.

Classificazione dei Pattern

Elenco di pattern può essere corposo (GoF ha definito 23 pattern generali) risulta importante cercare di identificare caratteristiche per la loro classificazione. Ciò semplifica anche lo studio e la comprensione dei pattern stessi. Sono stati idendificati due criteri principali:

- A cosa si riferisce il pattern
  - Oggetti: relazioni fra oggetti che possono modificarsi a tempo di esecuzione
  - Classi: riguardano relazioni tra classi e sottoclassi
- Cosa fa il pattern (scopo)
  - Creazionali: riguarda il processo di creazione di oggetti
  - Strutturali: riguarda la composizione di classi e di oggetti
  - Comportamentali: definisce come classi e oggetti interagiscono e distribuiscono fra loro delle responsabilità

Certamente è possibile classificarli secondo altri criteri

Object vs. Class pattern

Pattern creazionali di classe risolvono il problema della creazione di oggetti attraverso la delega a sottoclassi.

Per contro pattern di oggetti creazionali risolvono il problema attraverso la definizione di specifiche interazioni tra oggetti.

Pattern comportamentali di classe usano ereditarietà per definire algoritmi e flussi di controllo.

Per contro i pattern di oggetti comportamentali definiscono interazioni tra oggetti che permettono di svolgere una determinata attività

GoF classificazione

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	***Adapter (class)***	Interpreter Template Method
	Object	***Abstract Factory*** Builder Prototype ***Singleton***	Adapter (object) Bridge Composite Decorator Façade Flyweight ***Proxy***	Chain of Responsibility Command Iterator Mediator Memento ***Observer*** State ***Strategy*** Visitor

In che modo i DP contribuiscono a semplificare la progettazione

- Identificare gli oggetti necessari: scomporre sistema in oggetti è compito difficile. Pattern aiutano nell'identificazione di oggetti che rappresentano astrazioni non banali a partire dalla definizione del problema che si vuole risolvere.
- Determinare granularità degli oggetti: l'applicazione di un pattern porta con se scelte riguardanti la granularità. Ci sono pattern che permettono di definire oggetti come composizione di oggetti più semplici oppure pattern che permettono di rappresentare sottosistemi completi con semplici oggetti
- Definizione delle interfacce: l'applicazione di un pattern porta con se le scelte riguardanti le interfacce ed i meccanismi di interazione tra gli oggetti
- Definire implementazione: applicare un pattern comporta specifiche scelte implementative. In particolare vi sono pattern che si focalizzano su ereditarietà mentre altri si focalizzano su meccanismi di composizione

Pattern Creazionali

DP creazionali

Astraggono il meccanismo di generazione di istanze di una classe con l'obiettivo di rendere il sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati. La creazione degli oggetti viene tipicamente gestita da apposite "strutture".

Un DP creazionale che opera su classi userà ereditarietà per definire la classe da instanziare. Mentre DP che operano su oggetti useranno meccanismi di delega.

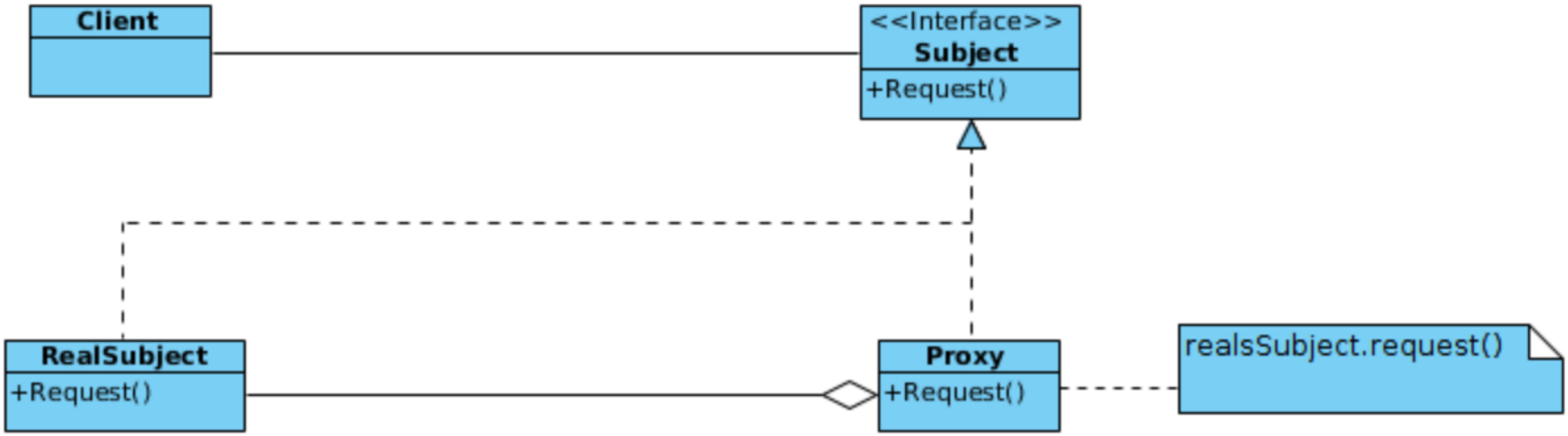
In generale semplificano la composizione di classi e la favoriscono rispetto all'uso dell'ereditarietà.

I pattern creazionali forniscono molta flessibilità nel cosa, il come, il chi ed il quando della creazione di oggetti

Factory Method

- Scopo: fornire un meccanismo per la creazione di oggetti simili (e.g. che implementano la stessa interfaccia) senza specificare quali siano le loro classi concrete
- Motivazione: si ha bisogno di determinare l'esatto tipo dell'oggetto da instanziare solo a run-time
- Sinonimi: Virtual Constructor
- Applicabilità:
  - sistema deve essere indipendente dalle modalità di creazione, composizione e rappresentazione dei suoi prodotti
  - si vuole una libreria (i.e. insieme di classi) che esponga soltanto l'interfaccia e non la sua implementazione

Factory Method - Struttura



Factory Method

- Conseguenze:
  - isola classi concrete
  - consente di cambiare in modo semplice l'implementazione ed in comportamenti esposti da un insieme di prodotti
  - promuove il riuso di prodotti/artefatti/codice
  - aggiunta e supporto di nuovi di prodotti semplice
- Partecipanti:
  - Factory (`Factory`) dichiara ed implementa i meccanismi di creazione per un insieme oggetti simili (e.g. che condividono la stessa interfaccia)
  - AbstractProduct (`ProductBase`) dichiara un'interfaccia (i.e. classe astratta o interface) per una tipologia di oggetti prodotto
  - ConcreteProduct (`ConcreteProductA`, `ConcreteProductB`, `ConcreteProductC`) definisce un oggetto prodotto che dovrà essere creato dalla corrispondente factory implementa l'interfaccia definita da AbstractProduct
  - Client (`Client`) crea istanze di ConcreteProduct attraveso la Factory utilizza soltanto l'interfaccia dichiarate in AbstractProduct

Abstract Factory

Ulteriori informazioni:
`http://it.wikipedia.org/wiki/Abstract_factory`

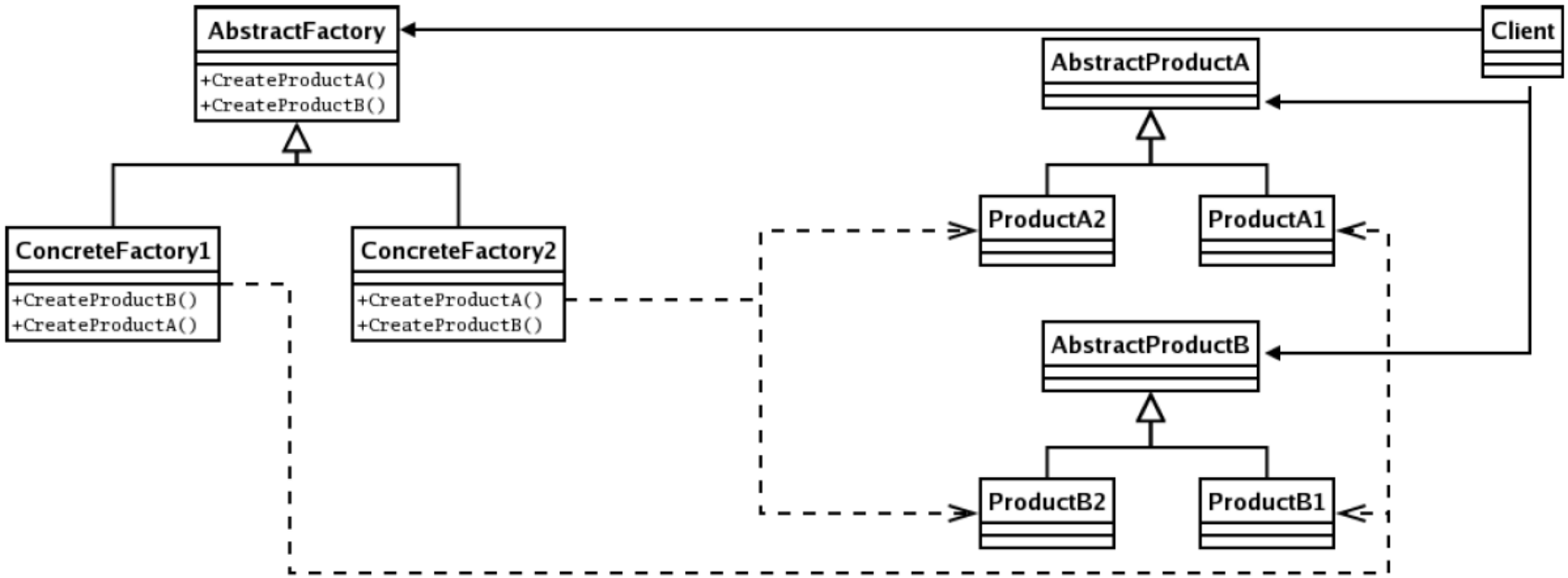
- Scopo: definire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le classi concrete.
- Motivazione: spesso ci si trova di fronte al problema di voler istanziare un oggetto senza specificare precisamente la classe. Comportamento chiaramente non possibile con meccanismi di creazione tipo `new`. Caso tipico delle interfacce grafiche e delle interfacce che mantengono coerenza con il tema in uso.
- Applicabilità: utilizzato quando:
  - sistema indipendente dalle modalità di creazione, composizione;
  - sistema configurabile dipendentemente dalle caratteristiche di una tra piú tipologie di oggetto
  - libreria di classi fornendo solo interfaccia ma non implementazioni specifiche

Abstract Factory

- Conseguenze:
  - isolamento delle classi concrete. Gli oggetti non contengono dettagli che si riferiscono ai meccanismi di creazione degli oggetti. La creazione di un particolare oggetto è visto come un normale servizio fornito da classi preposte allo scopo.
  - Risulta semplice la modifica/aggiunta di classi rappresentanti factory concrete. L'uso di queste risulta estremamente localizzato. Dunque prodotti diversi si ottengono modificando la factory.
  - Il supporto a nuove tipologie di prodotto che richiederebbero di modificare l'abstract factory risulta complessa in quanto richiede di modificare tutte le classi concrete.

Abstract Factory

- Struttura:



Abstract Factory

- Partecipanti:
  - ***AbstractFactory***: Dichiara un'interfaccia per le operazioni di creazione di oggetti astratti
  - ***ConcreteFactory***: implementa le operazioni di creazione di oggetti concreti
  - ***AbstractProduct***: Dichiara un'interfaccia per una tipologia di oggetti
  - ***ConcreteProduct***: definisce un oggetto che dovrà essere creato dalla corrispondente factory concreta
  - ***Client***: utilizza soltanto le interfacce dichiarate sopra
- Collaborazioni:
  - spesso esiste una sola istanza di una ConcreteFactory a run-time per le diverse tipologie
  - La classe AbstractFactory delega la creazione di oggetti alle sue sottoclassi.

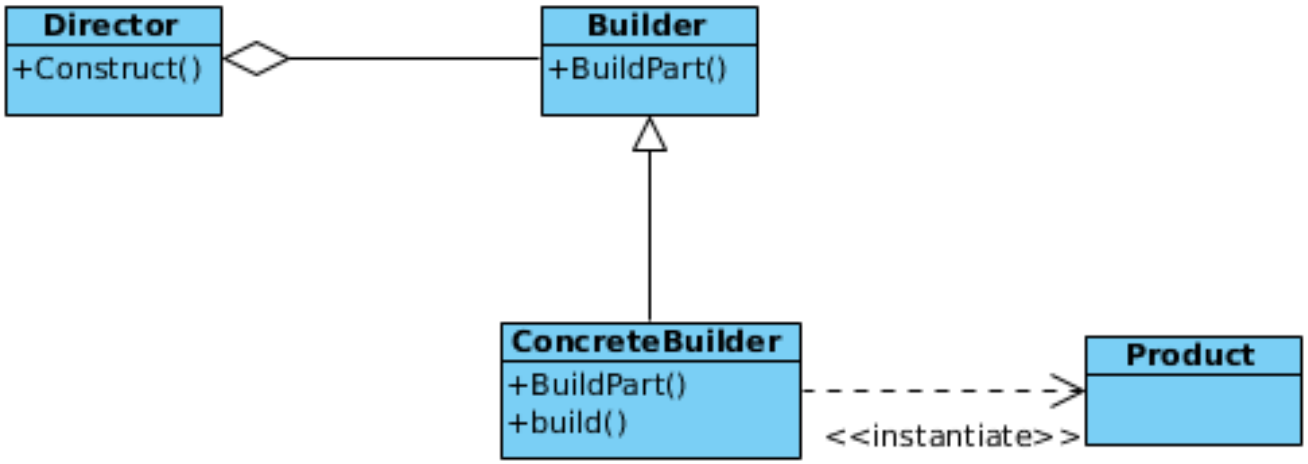
Abstract Factory

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: associato ad altri pattern creazionali in particolare al Singleton per avere classi concrete che non ammettono istanze multiple.

Builder

- Scopo: Separare la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa generare differenti rappresentazioni.
- Motivazione: Si immagini un generatore di profili per un gioco di ruolo. Il modo più semplice di popolare il sistema è permettere al computer di creare i profili. Ma se si volesse selezionare alcune caratteristiche quali professione, genere, colore dei capelli, etc. la generazione diventa un processo passo-passo che termina quando tutte le caratteristiche sono state selezionate. Il Builder pattern permette di creare differenti caratteri di un oggetto evitando il problema del “constructor pollution”.

Builder - Struttura



Builder

- Applicabilità:
  - da usare quando algoritmo di creazione di oggetto complesso deve essere reso indipendente dalle parti
  - processo di creazione diverso dalla rappresentazione
- Partecipanti:
  - Director (`Director`): costruisce l'oggetto usando algoritmi complessi e l'interfaccia di builder.
  - Builder (`Builder`): specifica un'interfaccia (classe astratta) per la costruzione di parti di oggetti complessi
  - ConcreteBuilder (`ThiefBuilder, WarriorBuilder, MageBuilder`): fornisce un interfaccia per la costruzione del prodotto semplificandone gli aspetti. Fornisce un'interfaccia per recuperare oggetto creato.
  - Product (`Hero`): prodotto complesso che deve essere creato

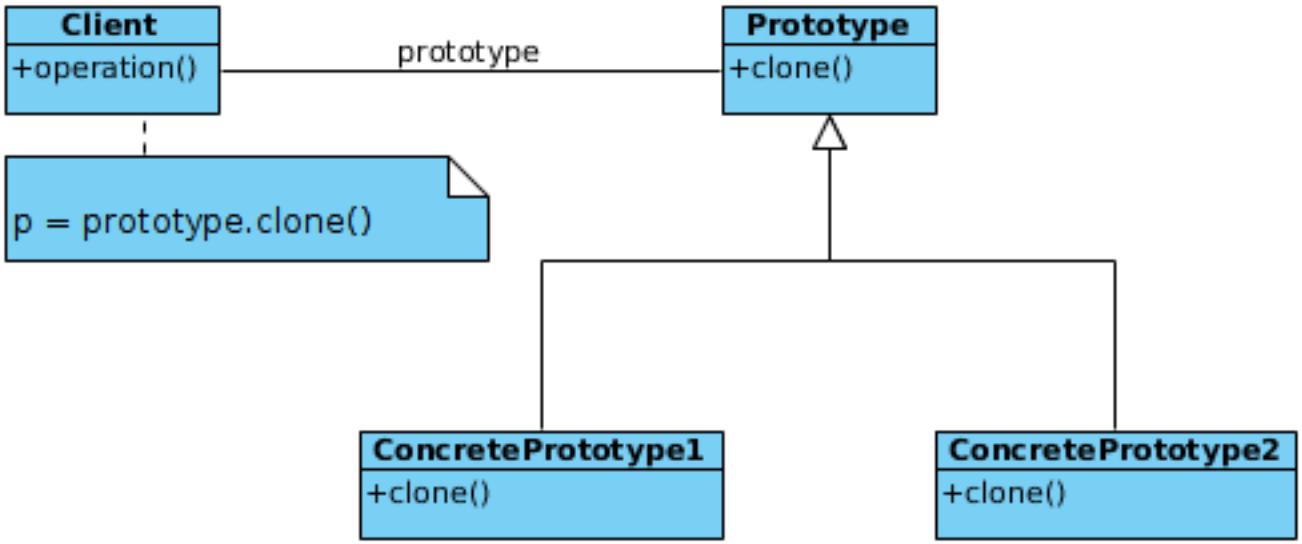
Builder

- Conseguenze:
  - Isola il codice di costruzione da quello di rappresentazione
  - controllo più fine sul processo di costruzione
  - più facile cambiare la rappresentazione interna degli oggetti
- Utilizzi noti:
  - java.lang.StringBuilder
  - java.nio.ByteBuffer as well as similar buffers such as FloatBuffer, IntBuffer and so on.
  - java.lang.StringBuffer
  - All implementations of java.lang.Appendable
  - Apache Camel builders
  - Apache Commons Option.Builder
- Pattern correlati: ....

Prototype

- Scopo: Permette di specificare oggetti a partire da un prototipo che può essere poi dettagliato
- Motivazione: Si vogliono realizzare artefatti complessi a partire da framework generali per quel contesto. Ad esempio costruire editor di modellazione per specifiche notazioni riusando meta-modelli definiti per la notazione.
- Sinonimi:
- Applicabilità: quando il sistema deve essere indipendente da come è prodotto, creato, e rappresentato:
  - quando le classi da istanziare sono create a run-time (dynamic loading)
  - evitare di costruire gerarchie di classi “factory” che rappresentano parallelamente la gerarchia delle classi
  - quando le istanze della classe possono avere poche differenti combinazioni di stato, è più conveniente usare prototipi piuttosto che creare classi manualmente
  - quando processo di creazione è costoso rispetto al cloning.

Prototype - Struttura



Prototype

- Conseguenze: Come per altri pattern nasconde il prodotto concreto dal client riducendo il numero di “nomi” conosciuti allo stesso.
  - Aggiunta rimozione di prodotti a run-time più facile
  - specifica di nuovi oggetti variando soltanto alcuni valori
  - specifica di nuovi oggetti variando la struttura
  - ridotto subclassing
  - configurare applicazione configurando dinamicamente le classi
- Partecipanti:
  - ***Prototype***: dichiara interfaccia per cloning
  - ***ConcretePrototype***: implementa operazione di cloning
  - ***Client***: crea un nuovo oggetto richiedendo cloning

Prototype

- Implementazione:
  - uso di un prototype manager
  - operazione di “clone”
  - inizializzazione dei cloni

  Java include interfaccia generale per cloning “java.lang.Cloneable”. Attenzione a gestione di riferimenti condivisi in oggetti complessi.
- Utilizzi noti: . . .
- Pattern correlati: una “Abstract Factory” potrebbe usare prototipi per per creare oggetti. Il prototipo potrebbe essere utile per pattern quali “Composite” e “Decorator”

Approfondimenti:
`https://refactoring.guru/design-patterns/prototype/java/example`

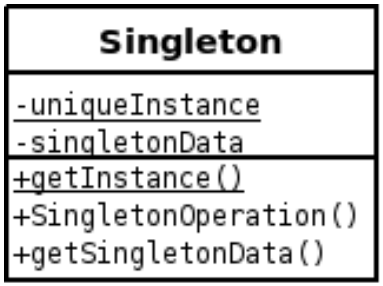
Singleton

Ulteriori informazioni:
http://it.wikipedia.org/wiki/Singleton

- Scopo: Fare in modo che a run-time esista al più una sola istanza di una classe e fornire un punto globale di accesso a tale istanza
- Motivazione: É spesso importante avere una sola istanza di una classe a run-time. Si consideri ad esempio il caso del File System Manager o del Window Manager di un sistema. (Kdm o Gdm per chi usa Linux). Dunque come possiamo fare in modo da garantire che a run-time non sia possibile avere più istanze di una stessa classe e che tale istanze sia facilmente accessibile ai potenziali utilizzatori?
- Applicabilità: il pattern singleton puè essere usato quando:
  - deve esistere una sola istanza di una classe e punto di accesso noto agli utilizzatori
  - quando l'unica istanza deve poter essere estesa attraverso definizione di sottoclassi ed i client non devono essere modificati come conseguenza di ciò

Singleton

- Conseguenze: controllo degli accessi all'istanza, spazio dei nomi ridotto, possibilità di estensione ad aver un numero n di istanze, facile manutenzione
- Struttura:



- Partecipanti:
  - ***Singleton***: definisce un'operazione `getInstance` che permette ai client di accedere all'unica istanza disponibile della classe. La detta operazione deve essere un'operazione di classe ovvero in Java dovrà essere marcata dalla parola chiave `static`

Singleton

- Collaborazioni: I client possono accedere ad un'istanza di singleton soltanto invocando l'operazione `getInstance`
- Conseguenze: Il pattern Singleton offre importanti benefici:
  - Accesso controllato ad un'unica istanza
  - Riduzione dello spazio dei nomi
  - Raffinamento delle operazioni e della rappresentazione interna
  - Gestione di un numero variabile di istanze
- Implementazione: si possono fare alcune considerazioni sui differenti meccanismi messi a disposizione dai differenti linguaggi OO per la definizione di operazioni di classe o per altri meccanismi che possono influenzare l'implementazione con un dato linguaggio.
- Utilizzi noti: ...
- Pattern correlati: altri pattern creazionali potrebbero usare il pattern singleton per esempio per risolvere gli specifici problemi "creazionali" associandoli alla necessità di avere una sola istanza.

Combinazioni e concorrenza

- Come si comportano le classi singleton in presenza di clienti concorrenti?
- Come combinare Abstract Facotry e Singleton?

Pattern Strutturali

Pattern Strutturali

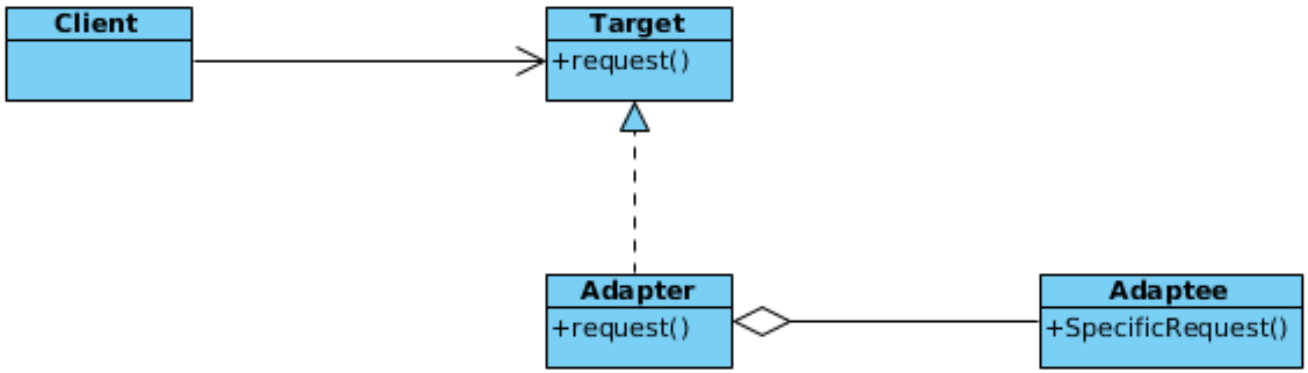
I pattern strutturali descrivono come comporre oggetti o classi per creare e riutilizzare strutture complesse fornendo interfacce più adatte all'uso da parte di oggetti client.

- basati su classi: utilizzano l'ereditarietà per comporre interfacce o implementazioni esempio: ereditarietà multipla combina due o più classi per ottenere una classe con tutte le proprietà delle superclassi
- basati su oggetti: modellano le modalità di composizione di oggetti per realizzare nuove funzionalità hanno maggiore flessibilità poiché è possibile cambiare la composizione durante l'esecuzione

Adapter

- Scopo:
  - gestire interfacce incompatibili
  - fornire un'interfaccia stabile a classi funzionalmente simili o con interfacce diverse
- Motivazione: spesso le classi di un sistema vengono strutturate/progettate cercando di offrire un profilo riusabile. Tuttavia, può capitare che queste classi non possono essere effettivamente riusate; per esempio perchè la loro interfaccia offerta non rispecchia esattamente i requisiti (o la segnatura) richiesti di uno specifico dominio
- Sinonimi: Wrapper
- Applicabilità:
  - si vuole utilizzare una classe esistente ma la sua interfaccia non è compatibile con quella che serve
  - si vuole realizzare una classe riusabile che coopera con altre classi anche se scorrelate o impreviste e con una interfaccia eventualmente incompatibile

Adapter - Struttura



Adapter

- Conseguenze:
  - adatta Adaptee a Target attraverso la classe concreta Adapter; la classe Adapter non è indicata se si volesse gestire l'adattamento di una classe ma anche tutte le sue sotto-classi
  - quanto "lavoro" deve fare la classe Adapter? dipende da quanto sono simili le interfacce Target ed Adaptee.
  - l'uso del pattern non è sempre trasparente ai Client
- Partecipanti:
  - ***Target***: definisce l'interfaccia di dominio con il client
  - ***Client***: coopera con oggetti conformi all'interfaccia Target
  - ***Adaptee***: definisce l'interfaccia esistente da adattare
  - ***Adapter***: realizza il meccanismo di adattamento

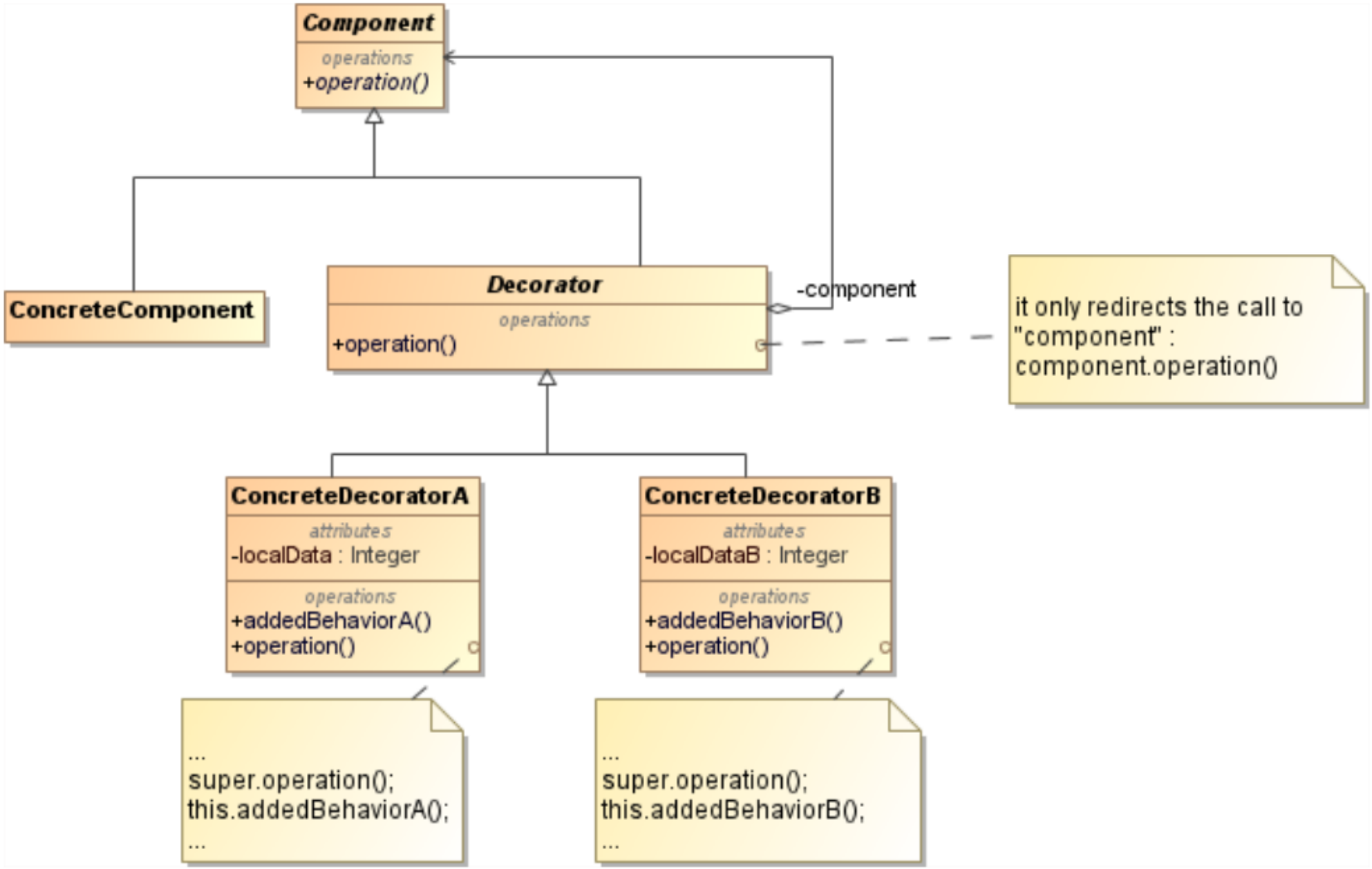
Adapter

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: ....

Decorator

- Scopo: aggiungere dinamicamente responsabilità ad un oggetto. I decoratori forniscono un'alternativa flessibile alla definizione di sottoclassi come strumento per l'estensione delle funzionalità
- Motivazione:
  - GUI toolkit dovrebbe consentire di aggiungere proprietà come bordi, scorrimento, ai singoli elementi grafici il modo per aggiungere responsabilità è tramite ereditarietà
    - estendere una classe e aggiungere il bordo
  - definire un approccio flessibile che consenta di racchiudere il componente da "decorare" in un altro che abbia la sola responsabilità di aggiungere il bordo/scrollbar/etc oggetto contenitore detto decorator decorator ha un'interfaccia conforme all'oggetto decorato in modo tale da essere trasparente ai vari client decorator trasferisce le richieste al componente decorato effettuando azioni aggiuntive (esempio decorando il bordo) prima o dopo il trasferimento della richiesta essendo trasparente ai client è possibile annidare i decorator consentendo l'aggiunta di un numero illimitato di responsabilità agli oggetti decorati
- Sinonimi: Wrapper

Decorator - Struttura



Decorator

- Applicabilità:
  - si vuole poter aggiungere responsabilità a singoli oggetti dinamicamente ed in modo trasparente, senza coinvolgere altri oggetti
  - si vuole poter togliere responsabilità agli oggetti
  - l'estensione diretta attraverso le definizione di sottoclassi non è praticabile
    - esplosione di sottoclassi per supportare ogni possibile combinazione
- Conseguenze:
  - maggiore flessibilità rispetto all'utilizzo dell'ereditarietà (multipla) statica
    - responsabilità possono essere aggiunte e rimosse in esecuzione semplicemente collegando e scollegando i decoratori agli oggetti decorati
    - in caso di ereditarietà è necessario creare una nuova classe per ogni responsabilità (es. BorderedScrollableTextView, BorderedTextView)

    consente di evitare di definire classi troppo complesse nella gerarchia

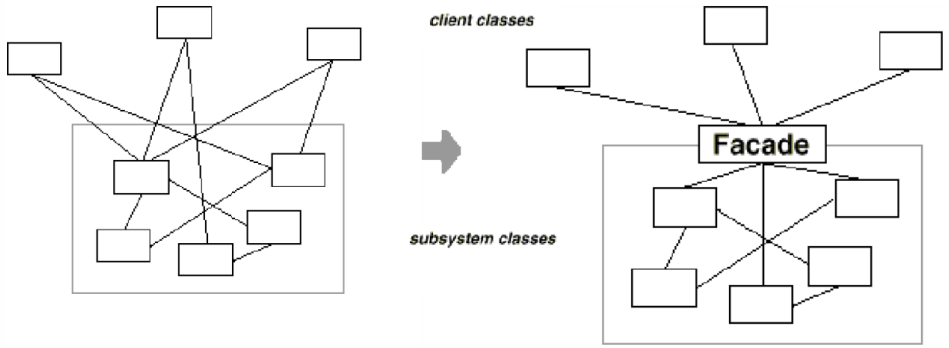
Decorator

- Partecipanti:
  - Component (`VisualComponent`): definisce l'interfaccia comune per gli oggetti ai quali possono essere aggiunte responsabilità dinamicamente
  - ConcreteComponent (`TextView`): definisce un oggetto al quale possono essere aggiunte responsabilità ulteriori
  - Decorator: mantiene un riferimento ad un oggetto Component e definisce un'interfaccia conforme all'interfaccia di Component
  - ConcreteDecorator (`BorderDecorator, ScrollDecorator`): aggiunge responsabilità al componente
- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: ....

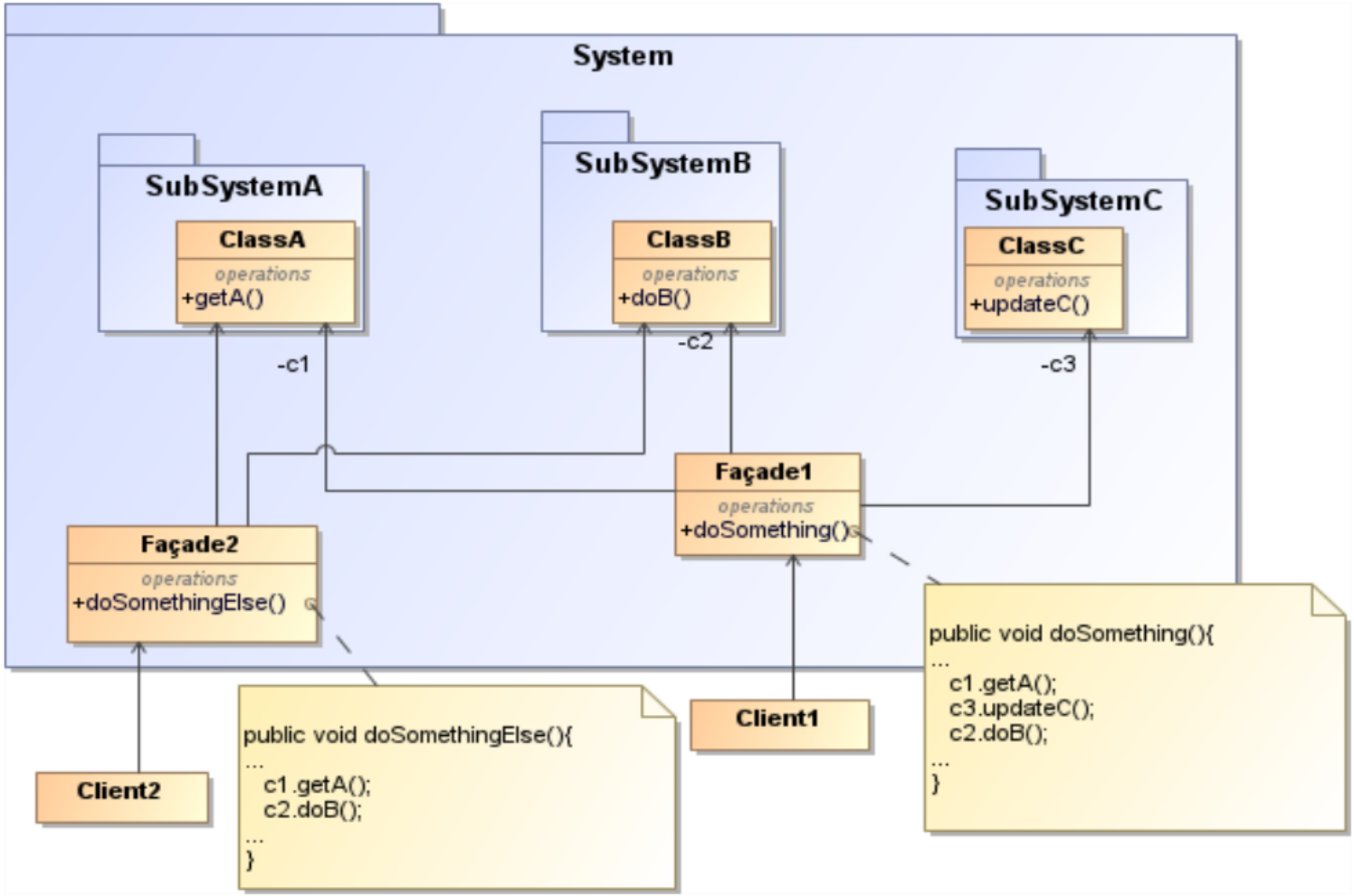
Façade

- Scopo: fornire una interfaccia unificata ad un insieme di interfacce di un sottosistema
- Motivazione:
  - è richiesta un'interfaccia comune ed unificata per un insieme disparato di implementazioni (i.e. classi o interfacce), come se si stesse identificando un sottosistema
  - minimizzare le comunicazioni e le dipendenze tra sottosistemi
  - mascherare l'implementazione di un sottosistema
    - l'implementazione può cambiare nel tempo ma non le funzionalità offerte

idea



Façade - Struttura



Façade

- Applicabilità:
  - progettazione architetturale di un sistema
  - fornire una interfaccia ad un sistema complesso o che evolve nel tempo
  - aumentare il grado di riuso di un sottosistema
  - stratificare un sistema identificando i punti di accesso ad ogni sottosistema
  - ci sono molte dipendenze tra l'implementazione del sottosistema ed i suoi client
  - un client per realizzare una singola operazione logica deve accedere a più classi del sottosistema molto differenti tra loro
- Partecipanti:
  - System: definisce un sistema del quale si vuole nascondere i dettagli implementativi all'esterno
  - SubSystem (`SystemA, SystemB, SystemC`): definisce eventuali sottomoduli del sistema
  - Façade (`Façade1, Façade2`): definisce l'interfaccia comune per l'accesso alle funzionalità del sistema può implementare logiche composizionali per esportare funzionalità di sistema
  - Client (`Client1, Client2`): utilizzatore delle funzionalità del sistema

Façade

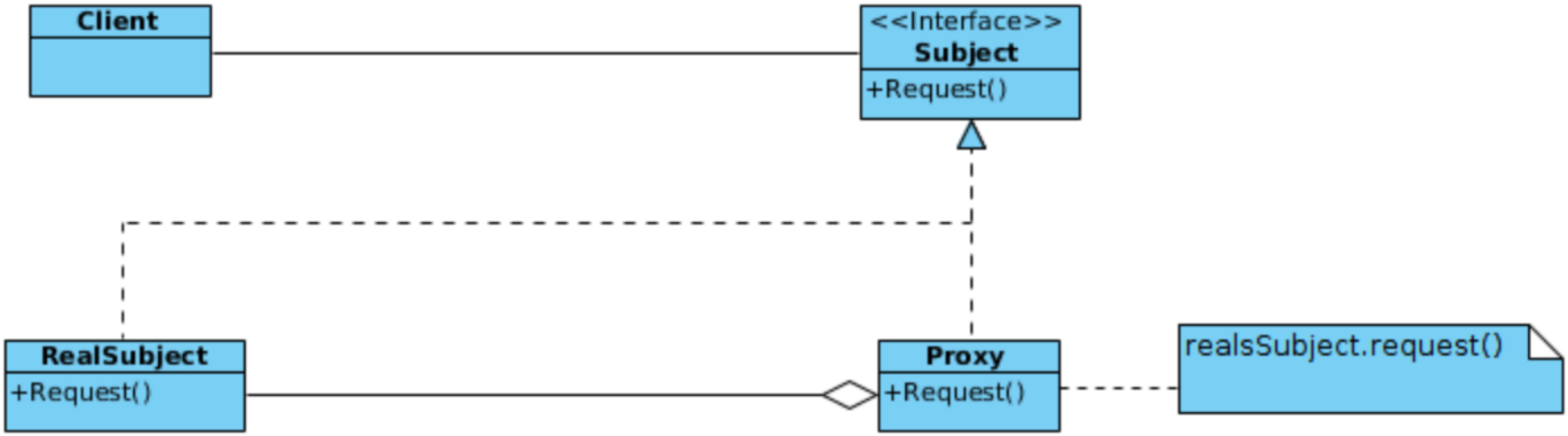
- Conseguenze:
  - riduce il numero di oggetti che un client deve gestire
  - rende il sistema più riusabile
  - riduce il grado di accoppiamento tra un sistema ed i suoi client, mitigando la ripercussione delle modifiche sul sistema anche sui client
  - la strutturazione in livelli consente di progettare sistemi indipendenti, con un basso tasso di dipendenze circolari
- Implementazione: . . .
- Utilizzi noti: . . .
- Pattern correlati: . . . .

Proxy

Ulteriori informazioni:
`http://it.wikipedia.org/wiki/Proxy_pattern`

- Scopo: fornire un surrogato o un segnaposto per un altro oggetto al fine di controllare l'accesso all'oggetto stesso. Allo stesso tempo può essere utile per rimandare la creazione di un oggetto ad un momento successivo al fine di ridurre uso di risorse.
- Motivazione: motivo può essere differire il costo di creazione ed inizializzazione dell'oggetto ad un momento successivo quando questo è effettivamente necessario
- Applicabilità: il pattern proxy puè essere usato secondo diverse tipologie:
  - remote proxy: fornisce rappresentazione locale per un oggetto residente in un diverso spazio di indirizzamento
  - virtual proxy: gestisce su richiesta la creazione di oggetti costosi
  - protection proxy: controlla l'accesso ad un oggetto. Si rivela utile quando possone esserci diversi diritti di accesso allo stesso oggetto.

Proxy - Struttura



Proxy

- Conseguenze:
  - introduce un livello di indirezione nell'accesso ad un oggetto. In generale operazioni molto complesse possono portare a degrado.
  - disaccoppiano cliente e oggetto stesso ottimizzando l'uso delle risorse rimandando solo al momento necessario l'esecuzione di operazioni potenzialmente costose.
- Partecipanti:
  - ***Proxy***: mantiene un riferimento che consente al proxy di accedere all'oggetto rappresentato. Proxy deve implementare la stessa interfaccia del referenziato al fine di nascondere la propria presenza al cliente
  - ***Subject***: definisce l'interfaccia comune per l'oggetto referenziato ed il proxy. Il proxy può sempre essere usato dove è richiesto l'oggetto referenziato.
  - ***RealSubject***: caratterizza l'oggetto referenziato dal proxy

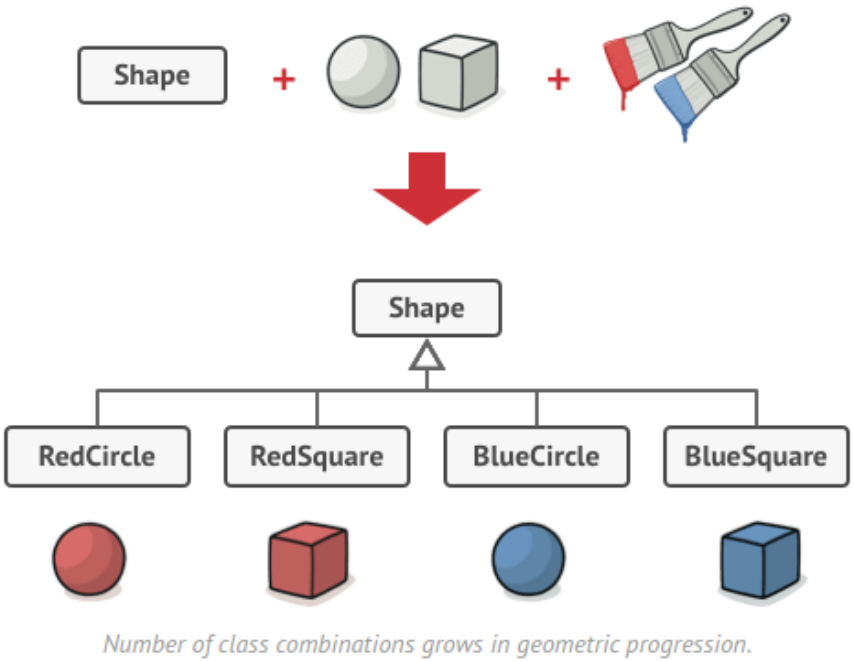
Proxy

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: ....

Caratterizzazioni importanti che comunque nel nostro studio non verranno approfondite.

Bridge

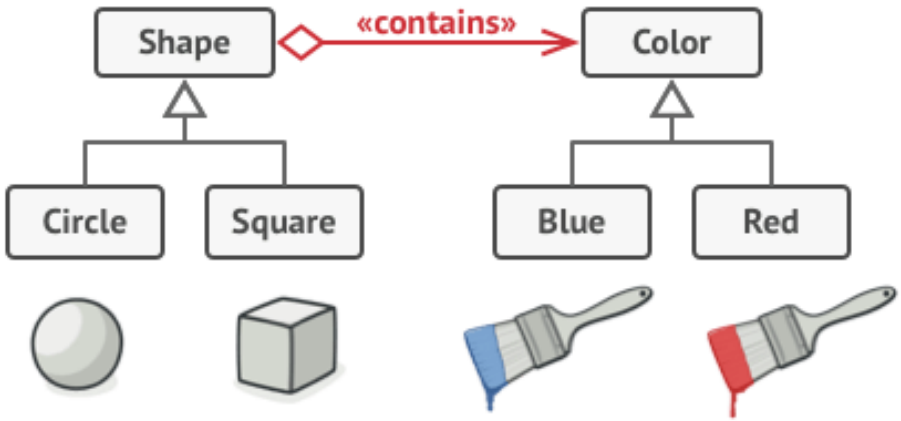
- Scopo: Dividere classi o famiglie di classi correlate in due gerarchie, astrazione e implementazione, che possono così essere sviluppate indipendentemente
- Motivazione:



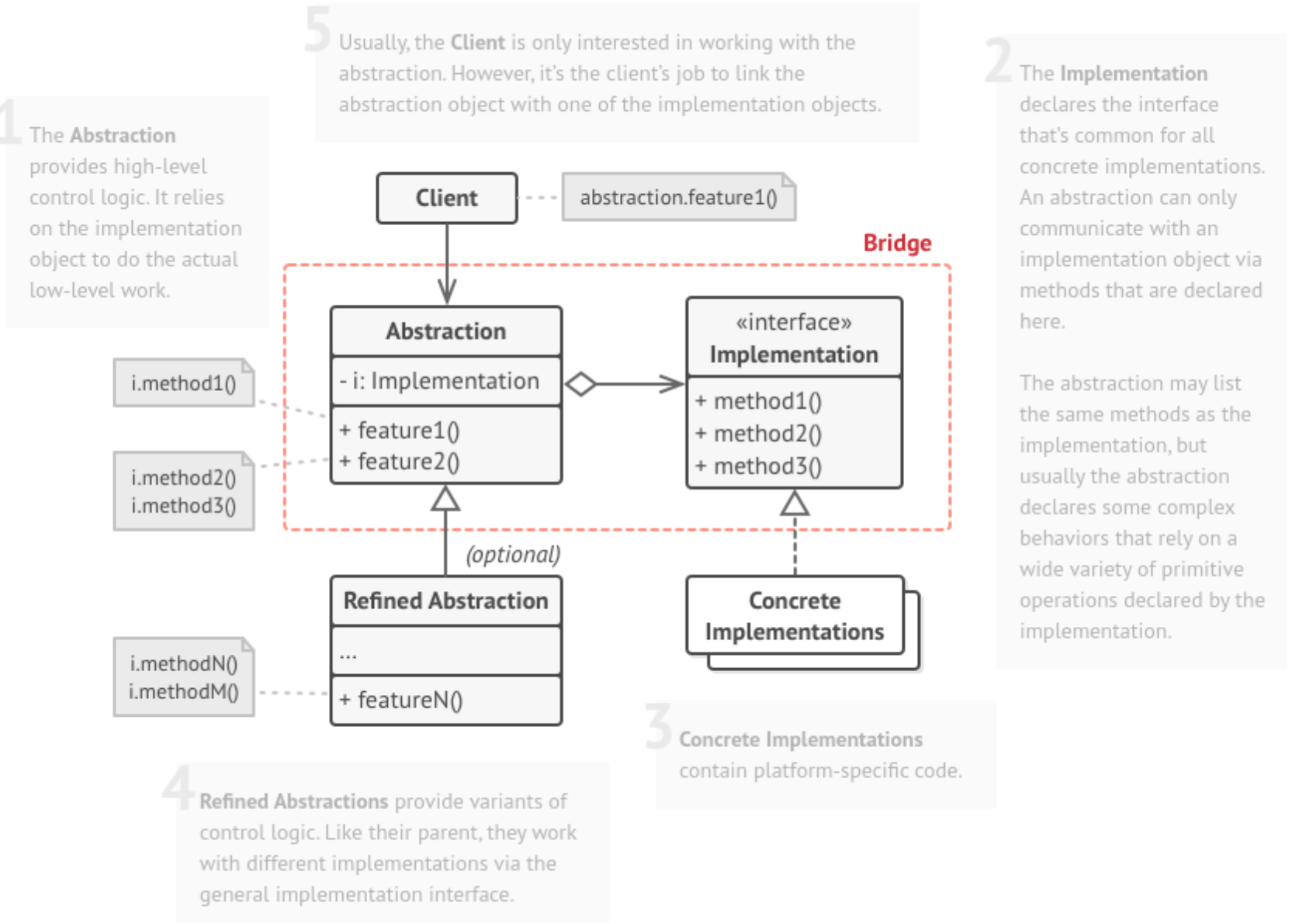
Number of class combinations grows in geometric progression.

- Sinonimi: Handle, Body

Bridge idea



Bridge - Struttura e codice di esempio



Bridge

- Conseguenze:
  - Disaccoppiamento di aspetti differenti di un oggetto
  - Maggiore flessibilità ed Estendiblità
  - maggiore “hiding” di dettagli implementativi

Bridge

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: Adapter, Strategy, State, Abstract Factory

Composite

https://refactoring.guru/design-patterns/composite

Flyweight

https://refactoring.guru/design-patterns/flyweight

Pattern Comportamentali

Pattern Comportamentali

I pattern comportamentali hanno a che fare con gli algoritmi e gli assegnamenti di responsabilità tra gli oggetti. Caratterizzano sistemi di controllo complessi.

Chain of Responsibility

https://refactoring.guru/design-patterns/chain-of-responsibility

Command

https://refactoring.guru/design-patterns/command

Mediator

https://refactoring.guru/design-patterns/mediator

Memento

https://refactoring.guru/design-patterns/memento

Observer

https://refactoring.guru/design-patterns/observer

State

https://refactoring.guru/design-patterns/state

Strategy

https://refactoring.guru/design-patterns/strategy

Visitor

https://refactoring.guru/design-patterns/visitor