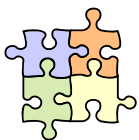


Tattiche architetturali  
(prima parte)

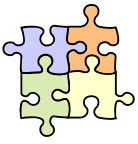
Dispensa ASW 310

ottobre 2011



- Fonti

- [SAP] Chapter 5, Achieving Qualities
- [Bachmann, Bass, Nord, 2007] Modifiability Tactics, Technical report CMU/SEI-2007-TR-002
- [Scott, Kazman, 2009] Realizing and Refining Architectural Tactics: Availability, Technical report CMU/SEI-2009-TR-006
- [Kim, Kim, Lu, Park, 2009] Quality-driven architecture development using architectural tactics, The Journal of Systems and Software, 82, 2009
- [Parnas, 1972] On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, 15, 1972



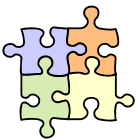
## - Obiettivi e argomenti

### □ Obiettivi

- introdurre le tattiche architettonali come guida per il raggiungimento di attributi di qualità
- illustrare alcune tattiche architettonali

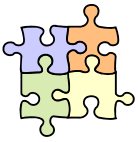
### □ Argomenti

- tattiche architettonali
- tattiche per le prestazioni
- tattiche per la modificabilità
- *tattiche per la disponibilità*
- *tattiche per la sicurezza*
- discussione



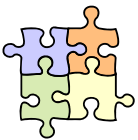
## \* Tattiche architettonali

- La comprensione di attributi e scenari di qualità consente di raccogliere ed organizzare i requisiti di qualità di un sistema informatico
  - tuttavia, non offre nessuna indicazione su come sia possibile raggiungere tali obiettivi di qualità
- I principali approcci architettonali che guidano il raggiungimento degli attributi di qualità
  - *tattiche architettonali* [SAP] – una tattica è una decisione di progetto che influenza il controllo di un attributo di qualità
  - *stili architettonali* [POSA] – uno stile di decomposizione architettonale – che di solito sostiene un certo numero di qualità, mediante l'applicazione di più tattiche



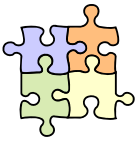
## Tattiche architettoniche

- Il progetto di un sistema consiste di un insieme di decisioni di progetto
  - alcune decisioni di progetto sostengono i requisiti funzionali – altre decisioni controllano gli attributi di qualità
- Una **tattica architettonica** (o, semplicemente, **tattica**) è una decisione di progetto fondamentale che influenza il controllo della risposta di un attributo di qualità [SAP]
  - detto in altro modo, una tattica è una trasformazione architettonica che ha effetto sul comportamento del sistema rispetto ad un particolare attributo di qualità
  - ad esempio, l'introduzione della ridondanza per aumentare la disponibilità del sistema



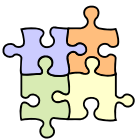
## Le tattiche codificano esperienza

- Le tattiche architettoniche hanno lo scopo di catturare e descrivere quello che gli architetti fanno in pratica
  - come i design pattern e gli stili architettonici, le tattiche descrivono delle decisioni progettuali (relative ad attributi di qualità) che sono state effettivamente applicate con successo in numerose situazioni
- Le tattiche costituiscono un approccio a grana più fine rispetto agli stili architettonici
  - sono più semplici da comprendere ed applicare – poiché influenzano il controllo di un singolo attributo di qualità
  - la loro applicazione ha lo scopo di raffinare il progetto di un'architettura – ad esempio, un progetto inizialmente basato sull'applicazione di stili architettonici



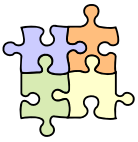
## Tattiche architetturali

- Ciascuna tattica è un'opzione di progetto per l'architetto
  - ad esempio, ci sono diverse tattiche per aumentare la disponibilità di un sistema
  - ciascuna tattica controlla in modo diverso il raggiungimento di un attributo di qualità
  - l'applicazione di una tattica (che è una scelta di progetto) ha impatto su uno o più elementi architetturali, e/o su una o più relazioni tra elementi, presenti in una o più viste architetturali
  - l'applicazione di una tattica può avere effetti collaterali – positivi oppure negativi – sul raggiungimento di altri attributi di qualità
- In pratica, l'architetto deciderà di applicare una collezione di tattiche, per realizzare una **strategia architetturale**
  - l'approccio (solitamente di compromesso) adottato al fine di raggiungere gli obiettivi complessivi di qualità del sistema



## - A che punto siamo?

- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione le tattiche?
  - abbiamo già identificato gli scenari architetturali
  - abbiamo già scelto uno (o più) stili architetturali in grado di sostenere gli scenari architetturali più rilevanti
  - abbiamo già applicato questi stili architetturali nelle varie viste, ed identificato, in ciascuna vista, un insieme di elementi architetturali, ed un insieme di relazioni ed interazioni tra di essi
  - questa decomposizione dovrebbe già sostenere gli scenari architetturali più rilevanti

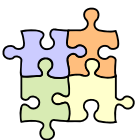


## A che punto siamo?

- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione le tattiche?
  - tuttavia, abbiamo anche capito che l'architettura corrente non è in grado di sostenere tutti gli scenari architetturali
  - tra di questi, abbiamo scelto uno o più scenari prioritari
  - che cosa possiamo fare affinché l'architettura sostenga anche questi scenari?

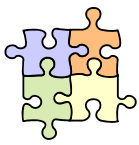
*questo è il punto in cui entrano in gioco le tattiche*

- possiamo selezionare ed applicare una o più tattiche
- ciascuna tattica è una scelta di progetto
  - che modifica uno o più elementi architetturali e/o una o più relazioni tra elementi architetturali – tra gli elementi e le relazioni presenti in una o più viste architetturali
  - al fine di sostenere un attributo di qualità di interesse per gli scenari architetturali presi in considerazione



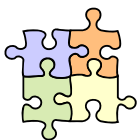
## - Un catalogo di tattiche

- Per comodità di illustrazione, le tattiche architetturali sono raggruppate per tipologia di attributo di qualità che sostengono – ad esempio, “tattiche per le prestazioni” e “tattiche per la disponibilità”
  - è importante capire che le tattiche mostrate, per quanto importanti, costituiscono un elenco incompleto



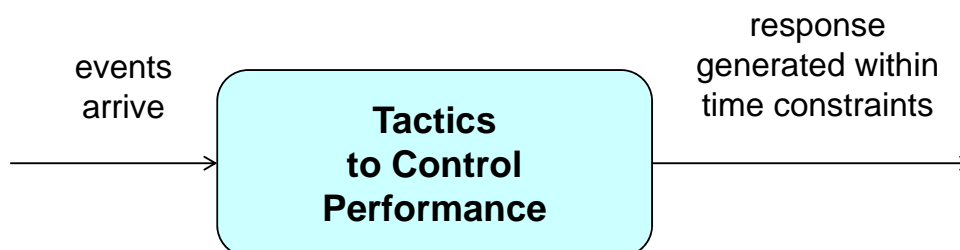
## \* Tattiche per le prestazioni

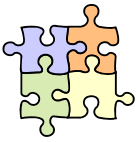
- Le **prestazioni** (performance) hanno a che fare con il **tempo di esecuzione**
  - si verificano degli eventi, ed il sistema deve rispondere a questi eventi
  - gli scenari di qualità relativi alle prestazioni hanno solitamente l'obiettivo di caratterizzare la distribuzione degli arrivi di questi eventi, nonché di imporre vincoli sul tempo entro cui bisogna rispondere ad un evento oppure sul numero di eventi ai quali bisogna rispondere in un'unità di tempo
- In questa trattazione consideriamo **tattiche per le prestazioni** che hanno l'obiettivo di controllare il **tempo di risposta ad un evento**
  - ovvero, il tempo entro cui viene generata la risposta ad un evento



## Tattiche per le prestazioni

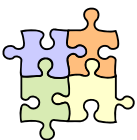
- In questa trattazione consideriamo tattiche per le prestazioni per controllare il tempo di risposta ad un evento





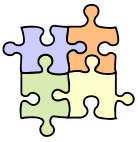
## Ragionamento

- Il tempo di risposta ad un evento è costituito da
  - consumo di risorse – il tempo effettivamente dedicato a generare la risposta all'evento
    - comprende il tempo di CPU
    - ma anche il tempo per l'accesso ai dati e per la comunicazione in rete tra elementi di calcolo distribuiti
  - attesa – il tempo in cui la computazione è bloccata in attesa di risorse
    - legato alla disponibilità di risorse, alla contesa di risorse (in presenza di eventi multipli da gestire), nonché alla dipendenza temporale (e necessità di sincronizzazione) tra computazioni diverse
  
- In corrispondenza, ci sono diverse categorie di tattiche per le prestazioni, rivolte a ridurre i vari contributi al tempo di risposta ad un evento



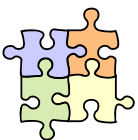
## Categorie di tattiche per le prestazioni

- Tre categorie principali di tattiche per le prestazioni
  - richiesta di risorse – *resource demand*
    - tattiche orientate a ridurre il tempo dedicato alla gestione di un evento, dunque al consumo di risorse individuale – ovvero, relativo ad una singola richiesta
      - ad es., “migliora l’algoritmo”
  - gestione di risorse – *resource management*
    - anche la scelta e la gestione delle risorse può influenzare il tempo di risposta
      - ad es., “usa un processore più efficiente”
  - arbitraggio di risorse – *resource arbitration*
    - quando c'è contesa di risorse, un'opportuna schedulazione nel loro uso ne consente un utilizzo più efficiente
      - ad es., “maggior priorità agli eventi più importanti”



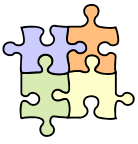
## - Resource demand (1)

- Tattiche volte a ridurre le risorse richieste per elaborare un evento
- *Increase computational efficiency*
  - migliorare l'efficienza temporale dell'algoritmo per l'elaborazione dell'evento diminuisce il tempo di risposta
  - in alcuni casi, si può migliorare l'efficienza temporale scambiando tempo con un'altra risorsa – ad es., memorizzare risultati intermedi per evitare un loro successivo ricalcolo
- *Reduce computational overhead*
  - esempi di overhead – comunicazione interprocesso e di rete, trasformazione del formato dei messaggi, cifratura, ...
  - l'uso di intermediari (spesso importanti per la modificabilità o altre qualità) aumenta le risorse consumate nell'elaborazione di un evento – l'eliminazione di intermediari può ridurre il tempo di risposta – spesso bisogna trovare un compromesso



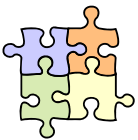
## Resource demand (2)

- Tattiche volte a ridurre il numero di eventi da elaborare
- *Manage event rate*
  - in alcuni casi è possibile (ed accettabile) ridurre la frequenza di campionamento con cui vengono monitorate le variabili ambientali – questo riduce il numero di eventi (richieste) da elaborare
- *Control frequency of sampling*
  - se non è possibile controllare l'arrivo di eventi generati esternamente, in alcuni casi è possibile (ed accettabile) ignorare alcuni eventi dalla coda delle richieste



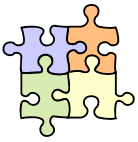
## Resource demand (3)

- Tattiche basate sul controllo delle risorse utilizzate
- *Bound execution time*
  - in alcuni casi è possibile (ed accettabile) porre un limite al tempo di esecuzione che può essere dedicato ad elaborare un evento – accettando un risultato approssimato
  - ad esempio, limitare il numero di iterazioni in un algoritmo iterativo
- *Bound queue sizes*
  - questo controlla il numero massimo di eventi che vengono posti in coda – e dunque le risorse da utilizzare per elaborare questi eventi



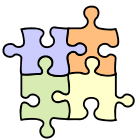
## Osservazioni (1)

- Prima di andare avanti nell'illustrazione di altre tattiche, è possibile fare alcune utili osservazioni
  - applicabilità delle tattiche
    - non sempre è possibile o accettabile applicare tutte le tattiche a disposizione
  - applicazione delle tattiche
    - l'applicazione di una tattica può avere effetto su uno o più elementi architetturali – e/o su una o più relazioni tra elementi architetturali – in una o più viste architetturali
    - ad es., *increase computational efficiency* ha effetto sulla caratterizzazione esterna di un'operazione di un elemento architetturale
    - ad es., *reduce computational overhead* può cambiare la scelta di alcuni elementi architetturali oppure della loro interconnessione



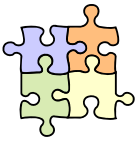
## Osservazioni (2)

- effetto qualitativo delle tattiche
  - l'applicazione di una tattica può portare a controllare nel modo desiderato un obiettivo di qualità – ad es., ridurre il tempo di risposta
    - in questa trattazione ci limitiamo ad illustrare delle intuizioni circa l'effetto qualitativo delle tattiche – ma ricordiamo che le tattiche descrivono decisioni progettuali che sono state effettivamente applicate con successo
- effetto quantitativo delle tattiche
  - in alcuni casi è possibile fare anche ragionamenti *quantitativi* sull'effetto dell'applicazione di una tattica
- detto in altro modo, qui stiamo facendo affermazioni solo sulla *direzione* dell'effetto dell'applicazione di una tattica, ma non sulla sua *quantità* – ma considerazioni quantitative sono spesso possibili ed effettivamente necessarie



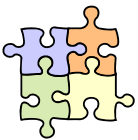
## Osservazioni (3)

- tattiche ed effetti collaterali
  - l'applicazione di una tattica ha di solito effetti benefici su un attributo di qualità – ma può anche avere effetti collaterali (negativi oppure positivi) su altri attributi di qualità
  - la progettazione dell'architettura è spesso basata su compromessi (trade-off) tra decisioni architettrurali
  - questa trattazione descrive solo alcuni degli effetti collaterali delle tattiche mostrate
- quando è necessario applicare una tattica?
  - non è utile applicare una tattica se il corrispondente attributo di qualità è già ben controllato
  - sono utili tecniche per la valutazione delle architetture – per capire in che misura viene controllata ciascun attributo di qualità e se gli obiettivi di qualità complessivi del sistema sono raggiunti



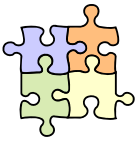
## - Resource management (1)

- Tattiche per la gestione delle risorse
- *Increase available resources*
  - risorse (processori, memorie, reti, ...) aggiuntive o più veloci hanno il potenziale per ridurre il tempo di risposta
  - a fronte di un ovvio aumento del costo del sistema
- *Introduce concurrency*
  - se possibile, si può ridurre il tempo per l'elaborazione di un evento (o di un flusso di eventi) gestendo attività diverse di questa elaborazione in parallelo, in modo concorrente
  - ad esempio, usando thread diversi per svolgere gruppi di attività differenti – oppure un thread diverso per ciascun flusso di eventi differenti
  - per sfruttare al meglio la concorrenza, è importante anche allocare in modo opportuno i thread alle risorse (load balancing)



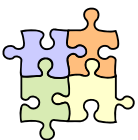
## Resource management (2)

- *Maintain multiple copies of either data or computations*
  - lo scopo della replicazione di risorse è ridurre la contesa – che si verificherebbe utilizzando delle risorse singole
  - ad esempio, la replicazione di un server (una computazione) – insieme ad un meccanismo di load balancing
    - creando un thread per ciascun evento diverso, per elaborare eventi diversi in parallelo
  - il caching è un esempio di tattica basata sulla replicazione dei dati
    - in questo caso, poiché i dati nella cache sono solitamente la copia di dati esistenti anche altrove, il sistema deve assumersi la responsabilità aggiuntiva di mantenere le varie copie di uno stesso dato opportunamente consistenti e sincronizzate

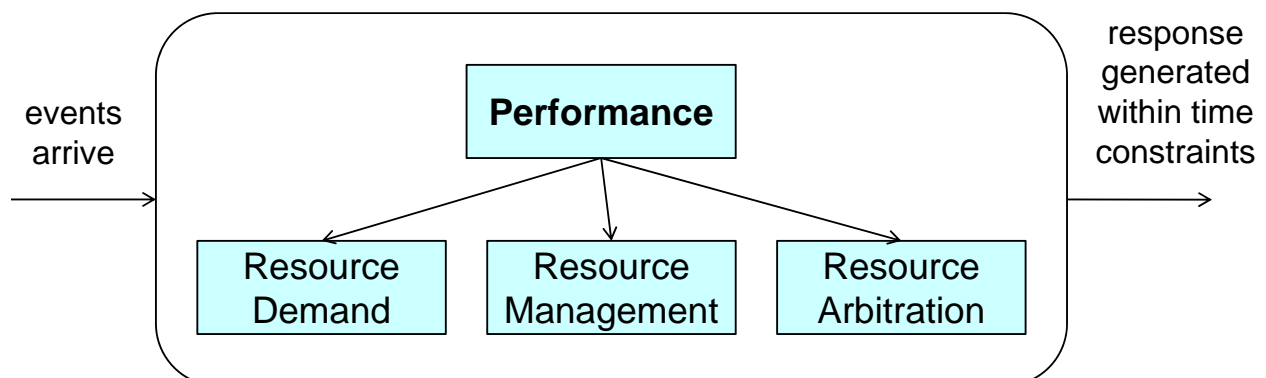


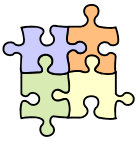
## - Resource arbitration

- L'arbitraggio delle risorse si basa sulla schedulazione dell'uso delle risorse, per consentirne un loro utilizzo più efficiente
  - una politica di scheduling comprende solitamente una politica per l'assegnazione di priorità ed una di dispatching – che può prevedere o meno meccanismi di pre-emption
- Alcune politiche di scheduling comuni
  - *first-in/first-out*
  - *fixed-priority scheduling*
    - la priorità può essere assegnata sulla base dell'importanza semantica, sul tipo di deadline o sulla base del tempo atteso di elaborazione dell'evento
  - *dynamic-priority scheduling*
    - round robin, oppure sulla base della distanza dalla deadline
  - *static scheduling*

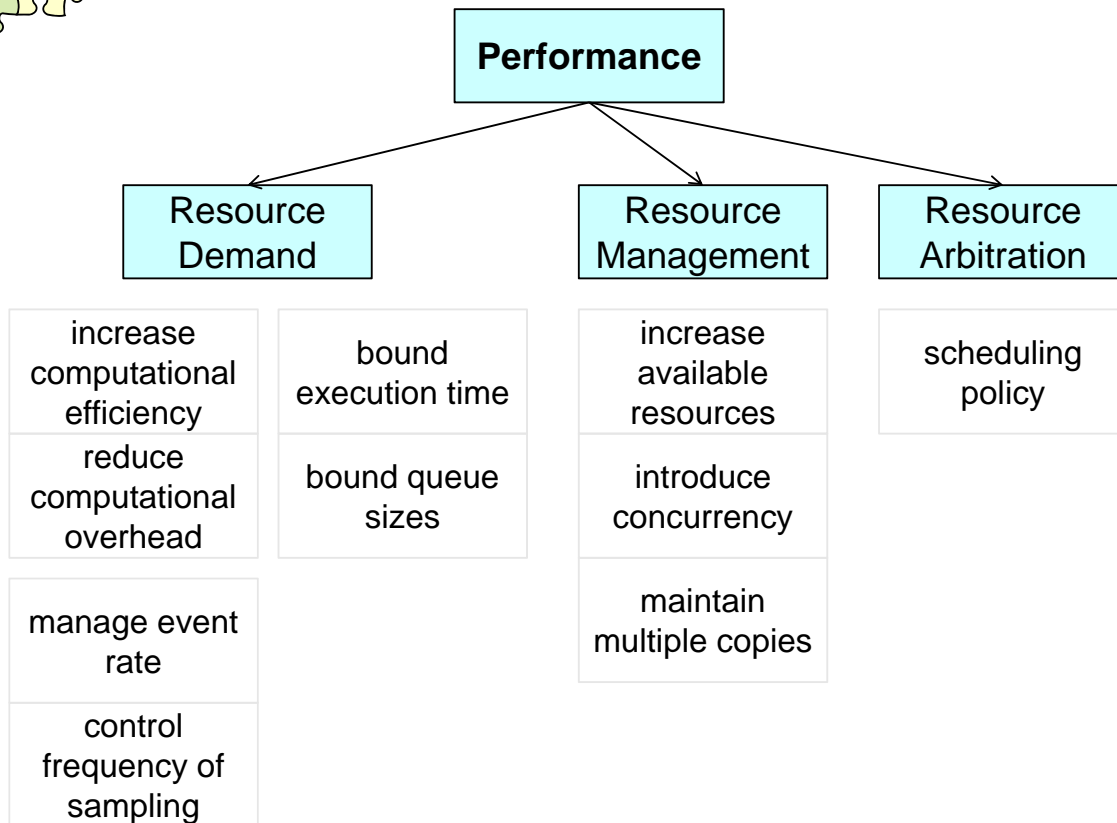


## - Tattiche per le prestazioni - sintesi





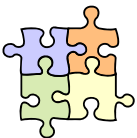
## Tattiche per le prestazioni - sintesi



25

Tattiche architetturali

Luca Cabibbo - ASw



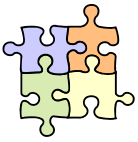
## \* Tattiche per la modificabilità

- La **modificabilità** è un attributo di qualità che riguarda il costo dei cambiamenti e si riferisce alla facilità con cui un sistema software può accomodare cambiamenti
  - ci sono diverse dimensioni da prendere in considerazione
    - che cosa potrà cambiare? ipotizziamo che l'unità di modifica sia una "responsabilità"
      - una **responsabilità** è un'azione, una decisione da prendere o una conoscenza da mantenere da parte di un sistema software o di un suo elemento
    - chi farà il cambiamento? sono possibili diversi casi: sviluppatore, amministratore del sistema oppure utente
    - quando sarà effettuato il cambiamento? casi possibili: in sede di progettazione, di deployment oppure di esecuzione
    - come viene misurato il costo del cambiamento? consideriamo solo il tempo umano

26

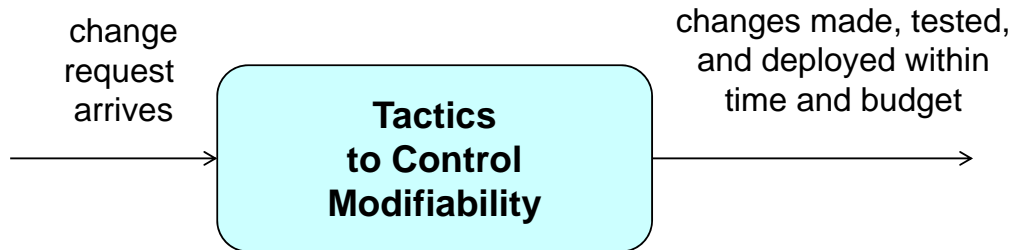
Tattiche architetturali

Luca Cabibbo - ASw

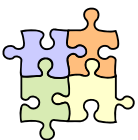


## Tattiche per la modificabilità

- In questa trattazione consideriamo *tattiche per la modificabilità* per controllare il *tempo ed il costo per implementare, testare e deployare un cambiamento*

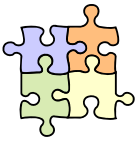


- Osservazione: nella progettazione dell'architettura
  - l'architetto non deve effettuare *ora* il cambiamento
  - piuttosto, deve garantire che certi tipi di cambiamenti potranno, *in futuro*, essere gestiti facilmente



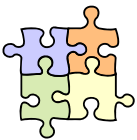
## Ragionamento

- Contributi al costo (medio) per la modifica di una responsabilità  $R$ 
  - costo (medio) della modifica relativa direttamente alla singola responsabilità  $R$ 
    - la modifica avverrà nell'ambito dell'elemento  $E_R$  a cui è assegnata la responsabilità  $R$
  - costo (medio) della modifica di tutte le responsabilità  $R_i$  alle quali la modifica va propagata
    - questo costo va pesato rispetto alla probabilità che una modifica di  $R$  ( $E_R$ ) richieda anche una modifica di  $R_i$  ( $E_{R_i}$ )
    - queste modifiche avverranno nell'ambito di elementi che dipendono (direttamente o indirettamente) da  $E_R$
  - costo del deployment della modifica
- In corrispondenza, diverse categorie di tattiche per la modificabilità, con l'obiettivo di ridurre questi contributi di costo



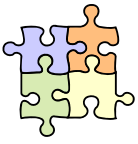
## Modificabilità, accoppiamento e coesione

- La modificabilità di un sistema è correlata a misure/metriche come l'accoppiamento e la coesione degli elementi del sistema
  - la **coesione** è una misura della forza delle relazioni tra le responsabilità di uno specifico modulo
  - l'**accoppiamento** è una misura della forza delle dipendenze tra moduli
    - l'accoppiamento si riduce quando le relazioni tra elementi che non appartengono allo stesso modulo sono ridotte
  - intuitivamente – e comunque solo in prima approssimazione
    - il costo della modifica della responsabilità  $R$  nell'ambito dell'elemento  $E_R$  (a cui è assegnata la responsabilità  $R$ ) è commisurato (in modo inverso) alla coesione di  $E_R$
    - il costo delle modifiche in altri elementi diversi da  $E_R$  è commisurato (in modo diretto) all'accoppiamento degli altri elementi software verso  $E_R$



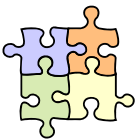
## Parentesi: Alcune forme di coesione

- Alcune forme di coesione
  - coesione per pura coincidenza
  - coesione logica
    - gli elementi raggruppati in un modulo sono logicamente correlati, ma implementati in modo indipendente
  - coesione temporale
    - gli elementi sono usati all'incirca nello stesso tempo
  - coesione di comunicazione
    - gli elementi devono accedere agli stessi dati o dispositivi
  - coesione sequenziale
    - gli elementi sono usati in un ordine particolare
  - coesione funzionale
    - gli elementi contribuiscono a svolgere una singola funzione
  - coesione dei dati
    - una classe implementa un tipo di dato



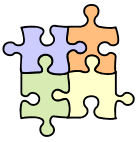
## Parentesi: Alcune forme di accoppiamento

- Alcune forme di accoppiamento
  - accoppiamento di dati interni
    - una classe modifica direttamente le variabili d'istanza di un'altra classe
  - accoppiamento mediante dati globali
    - due o più classi condividono dati globali
  - accoppiamento di controllo
    - una classe esegue operazioni in un ordine fissato, ma l'ordine è controllato altrove
  - accoppiamento di componenti
    - una classe gestisce dati che sono istanze di altre classi
  - accoppiamento mediante interfaccia e parametri
    - una classe richiede l'esecuzione di operazioni ad altre classi
  - accoppiamento per sottoclasse



## Tattiche come trasformazioni

- Nella progettazione, le responsabilità sono assegnate ad elementi software e, in particolare, a moduli (unità di sviluppo e, pertanto, di modifica)
  - un aspetto fondamentale relativo alle tattiche per la modificabilità è la possibilità di riorganizzare responsabilità
    - una responsabilità può essere decomposta in più sotto-responsabilità separate
    - più responsabilità (o sotto-responsabilità) possono essere fuse
    - una responsabilità (o sotto-responsabilità) può essere mossa da un elemento (modulo) ad un altro
  - obiettivo dell'applicazione delle tattiche per la modificabilità è infatti identificare un'assegnazione di responsabilità che minimizzi il costo dei cambiamenti attesi



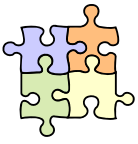
## Categorie di tattiche per la modificabilità

- Categorie principali di tattiche per la modificabilità
  - *reducing the cost of modifying a single responsibility*
    - per ridurre il costo di modificare una singola responsabilità
  - *increase cohesion*
    - per ridurre il costo dei cambiamenti intervenendo sulla coesione del sistema
  - *reducing coupling*
    - per ridurre il costo dei cambiamenti intervenendo sull'accoppiamento del sistema, per prevenire una propagazione a cascata delle modifiche
  - *defer binding time*
    - per controllare il costo ed il tempo richiesto dal deployment della modifica

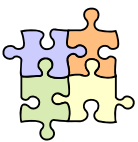
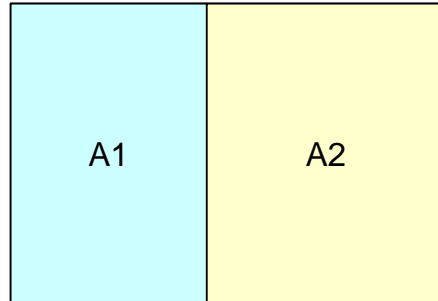


## - Reducing the cost of modifying a single res.

- L'approccio di base per ridurre il costo della modifica di una singola responsabilità è separare le responsabilità in base ai cambiamenti previsti
- *Split a responsibility*
  - sia R la responsabilità che è il target di una particolare modifica M prevista – ovvero, M è uno specifico scenario di modificabilità
  - se la responsabilità R comprende molte capacità/funzionalità, allora il costo della modifica sarà alto
  - se invece la modifica M si ripercuote solo su una porzione di R, allora è possibile ridurre il costo atteso della modifica raffinando la responsabilità in più sotto-responsabilità, e ponendo queste sotto-responsabilità in moduli diversi
  - un criterio per separare responsabilità in modo efficace è che le sotto-responsabilità possano essere modificate separatamente

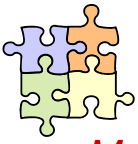


## Split a responsibility



## - Increasing cohesion

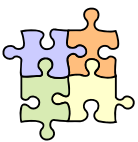
- Le tattiche per aumentare la coesione hanno in genere l'obiettivo di ridurre il numero di moduli sui quali un certo cambiamento si ripercuote direttamente
  - ovvero, moduli le cui responsabilità vanno cambiate per realizzare il cambiamento richiesto
  - sebbene non ci sia necessariamente una relazione precisa tra il numero di moduli su cui si ripercuote un cambiamento ed il costo di implementare il cambiamento, ridurre le modifiche ad un piccolo insieme di moduli ha generalmente l'effetto di ridurre tale costo
  - lo scopo di queste tattiche è assegnare (durante la progettazione) responsabilità a moduli in modo che ciascuno dei cambiamenti previsti abbia una portata limitata



## Increasing cohesion (1)

### □ *Maintain semantic coherence*

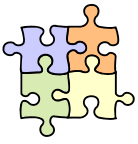
- la **coerenza semantica** si riferisce alle relazioni tra le responsabilità assegnate ad un modulo – anche con riferimento ad un insieme di cambiamenti attesi
  - scopo è garantire che tutte queste responsabilità lavorino insieme – senza far eccessivo affidamento ad altri moduli
- questo è possibile scegliendo ed assegnando responsabilità che hanno coerenza semantica
- le metriche di accoppiamento e coesione sono un tentativo di misurare la coerenza semantica
  - tuttavia, queste metriche non prendono in considerazione il contesto dei cambiamenti previsti
  - viceversa, la coerenza semantica, rispetto alla coesione, è misurata anche rispetto ad un insieme di cambiamenti previsti



## Maintain semantic coherence

### □ *Maintain semantic coherence*

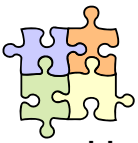
- ad esempio, se A e B sono responsabilità, ed un certo cambiamento atteso M si ripercuote su  $A' \subset A$  e  $B' \subset B$ , allora può essere utile
  - collocare in uno stesso elemento A' e B'
  - collocare A-A' e B-B' in elementi diversi
- attenzione, quello riportato qui sopra è solo un esempio



## Maintain semantic coherence

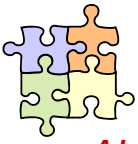
attenzione, è solo un esempio  
dell'applicazione di maintain semantic  
coherence

non sempre maintain semantic  
coherence si applica in questo modo



## Un criterio per la decomposizione in moduli

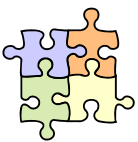
- Un celebre articolo [Parnas, 1972] suggerisce il seguente criterio per la decomposizione in moduli di un sistema
  - we propose that one begins with a list of difficult design decisions or design decisions which are likely to change – each module is then designed to hide such a decision from the others [Parnas, 1972]
  - in questo criterio è possibile trovare due contributi significativi
    - che ciascuna decisione di progetto difficile o soggetta a cambiamento – in particolare, relativa ad un cambiamento previsto – vada assegnata ad un modulo diverso – “maintain semantic coherence”
    - che queste decisioni vadano nascoste ad altri moduli – è un suggerimento della tattica “use encapsulation”



## Increasing cohesion (2)

### ▣ *Abstract common services*

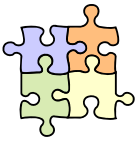
- un modo per sostenere il riuso è realizzare moduli specializzati che forniscono servizi comuni ad altri moduli
- se questi servizi comuni sono ad un livello opportuno di astrazione (ovvero, implementati in una forma generale, più generale che non rispetto ai singoli utilizzi), allora è sostenuta anche la modificabilità
  - infatti, modifiche ai servizi comuni vanno realizzate solo una volta – piuttosto che in tutti i moduli in cui i servizi sono utilizzati



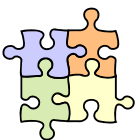
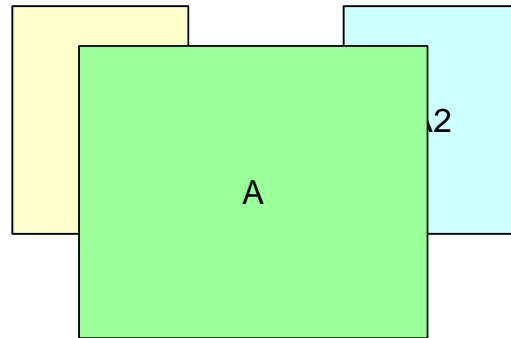
## Abstract common services

### ▣ *Abstract common services*

- alcuni esempi
  - un modulo offre un servizio  $A'$  – ma anche un altro servizio  $A''$ , che è una variante di  $A'$ 
    - allora può essere utile definire un servizio comune  $A$  più generale di  $A'$  e  $A''$  – implementandolo una sola volta, in una forma leggermente più generale
    - ogni modifica del servizio  $A$  dovrà poi essere realizzata (e verificata) una sola volta anziché due volte
  - più in generale, refactoring per generalizzare responsabilità simili
  - anche l'uso di framework e middleware è un'applicazione di questa tattica

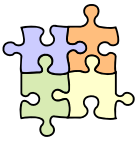


## Abstract common services



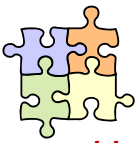
## - Reducing coupling

- Le tattiche per ridurre l'accoppiamento hanno in genere l'obiettivo di ridurre il numero di moduli sui quali un certo cambiamento si ripercuote indirettamente
  - ovvero, moduli le cui responsabilità non vanno cambiate direttamente per realizzare il cambiamento
  - ma si può doverne comunque cambiare l'implementazione per accomodare cambiamenti in altri moduli
- Un *ripple effect* da una modifica è la necessità di effettuare un cambiamento a moduli su cui la modifica non si ripercuote direttamente
  - ad es., A viene modificato (a causa di un cambiamento richiesto M), ma anche B va modificato (non per il cambiamento M, ma a causa delle modifiche in A)
  - questo è in genere motivato da una qualche forma di *accoppiamento/dipendenza* di B da A



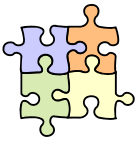
## Forme di accoppiamento/dipendenza

- Alcune forme di accoppiamento/dipendenza tra elementi A e B
  - sintassi dei dati o dei servizi – ad es., B dipende dal prototipo di un'operazione di A, o dal formato dei messaggi scambiati con A
  - semantica dei dati o dei servizi – assunzioni (contratti) su messaggi ed operazioni
  - sequenza dei dati o del controllo – ad es., alcune operazioni vanno invocate in un certo ordine
  - identità dell'interfaccia – ad es., B dipende da (il nome di) un'interfaccia implementata da A
  - locazione – B dipende dalla locazione runtime di A
  - qualità dei servizi/dei dati – B dipende dalla qualità del servizio/dei dati offerti da A
  - esistenza – B dipende dall'esistenza di A

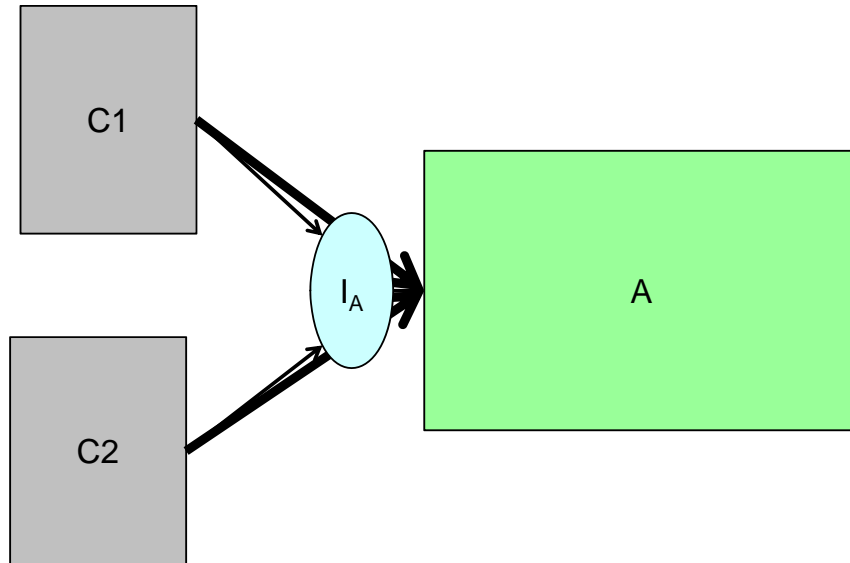


## Reducing coupling (1)

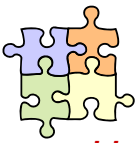
- *Use encapsulation*
  - è comune decomporre le responsabilità di un elemento in parti più piccole, scegliendo alcune parti da rendere pubbliche ed altre da mantenere private – le responsabilità pubbliche sono rese disponibili ad altri moduli mediante apposite interfacce
  - scopo dell'*incapsulamento* di un elemento A è ridurre la probabilità di propagazione dei cambiamenti nell'elemento A verso altri elementi
    - ciò richiede una separazione netta tra interfacce ed implementazione – sulla base di interfacce esplicite e stabili
  - il criterio di decomposizione dell'*incapsulamento* è quello di isolare ciascun cambiamento atteso in un singolo modulo, per prevenire la propagazione dei cambiamenti ad altri moduli [Parnas, 1972]



## Use encapsulation

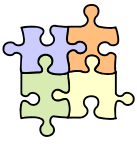


- L'uso dell'interfaccia  $I_A$ 
  - riduce l'accoppiamento tra C1 e C2 ed A
  - protegge C1 e C2 da cambiamenti nell'implementazione di A

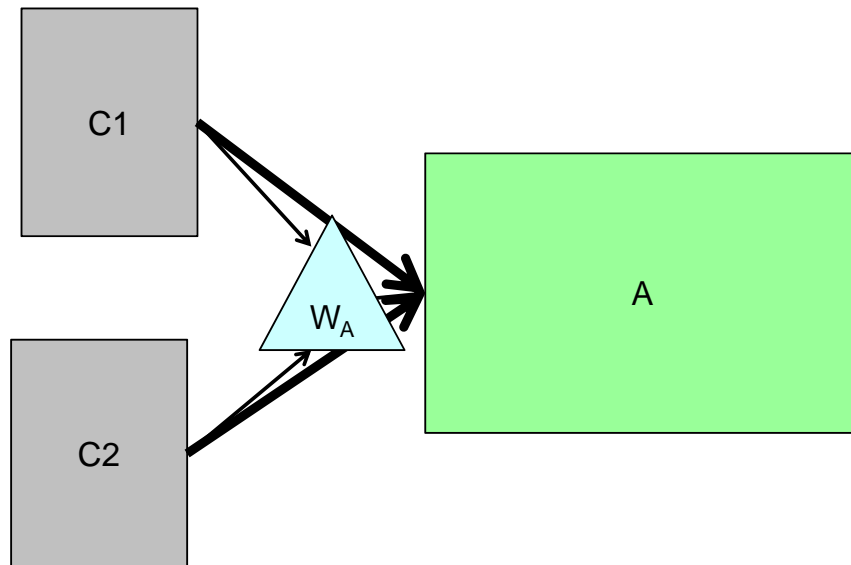


## Reducing coupling (2)

- *Use a wrapper*
  - un *wrapper* (è una forma di incapsulamento) è un'interfaccia per un modulo che trasforma i dati o controlla le informazioni per il modulo
  - in generale, l'incapsulamento ha lo scopo di occultare informazioni – questo può avvenire usando delle trasformazioni come parte della strategia di occultamento
    - un wrapper ha lo scopo di trasformare invocazioni
  - lo scopo dell'uso di un wrapper attorno ad un elemento A è di evitare che certe modifiche debbano essere propagate all'elemento A
    - in particolare, modifiche esterne ad A, ma relative all'uso che altri elementi devono fare di A
    - questo è efficace nella misura in cui il costo di modificare il wrapper è minore del costo di modificare il modulo A



## Use a wrapper

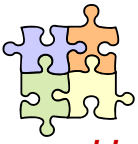


- L'uso del wrapper  $W_A$ 
  - protegge A da cambiamenti che C1 e C2 vogliono fare nell'uso di A



## Reducing coupling (3)

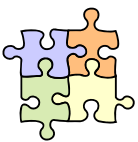
- *Raise the abstraction level*
  - alzare il livello di astrazione per una responsabilità vuol dire rendere la responsabilità più generale – ovvero in grado di eseguire più funzioni – mediante un'opportuna parametrizzazione delle sue attività
    - i parametri possono essere definiti in termini di variabili semplici o di un "linguaggio" complesso – che magari richiede un'opportuna interpretazione
    - i parametri ricevuti vengono dapprima convertiti in una forma interna, poi la responsabilità originale viene eseguita
  - più la responsabilità è generale, più è probabile che i cambiamenti richiesti possano essere gestiti intervenendo sul linguaggio di input (in termini di parametri che si è già in grado di gestire) che non modificando la responsabilità



## Reducing coupling (4)

### □ *Use an intermediary*

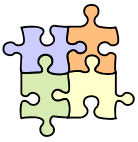
- un intermediario può rompere la dipendenza tra un elemento A ed un elemento B
  - l'intermediario ha lo scopo di gestire le attività associate con la dipendenza
- il tipo di intermediario da utilizzare dipende dal tipo di dipendenza che si vuole rompere
  - ad esempio, se A produce messaggi e B li consuma, l'uso di un canale di comunicazione può rimuovere la conoscenza di A sul fatto che il consumatore dei suoi messaggi sia B
- alcuni tipi di intermediari
  - un design pattern per ridurre la dipendenza dai servizi – ad es., facade, proxy, adapter, bridge, mediator, ...
  - un broker o un servizio di directory
  - una factory, per ridurre la dipendenza dall'esistenza



## Reducing coupling (5)

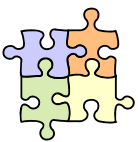
### □ *Restrict communication paths*

- si tratta di un caso speciale dell'uso di un intermediario
- questa tattica consiste nel rimuovere una dipendenza relativa ad una necessità di comunicazione, incanalando questa comunicazione in un intermediario



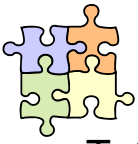
## - Defer binding time

- Il costo di una modifica dipende anche dal tempo in cui la modifica viene effettuata – questo porta ad un’ulteriore categoria di tattiche per la modificabilità – che sono in parte trasversali alle precedenti
  - sono tattiche con lo scopo di controllare il costo ed il tempo richiesto dal deployment della modifica
  - intuizione: finché c’è un’opportuna preparazione, più tardi nella vita di un sistema si verifica una modifica, minore è il suo costo
    - la chiave è l’ “opportuna preparazione”
    - attenzione, senza l’ “opportuna preparazione” è normalmente vero il contrario!
  - in effetti, alcune di queste tattiche (in particolare, per effettuare modifiche a runtime) richiedono che le particolari modifiche siano note in anticipo
    - altre richiedono che siano comunque note le “tipologie” di modifiche attese



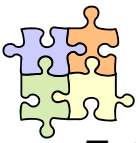
## Defer binding time (1)

- Tattiche per effettuare modifiche al tempo della codifica
  - *Parametrize modules*
    - un modulo è sufficientemente generale per gestire una varietà di attività, che possono essere selezionate usando opportuni parametri
  - *Polymorphism*
    - il polimorfismo è basato sull’identificazione di responsabilità che sono comuni ad un insieme di servizi
    - consente di fare plug-and-play di elementi software che offrono tali servizi – senza ripercussioni sui loro client
  - *Use aspect-oriented programming*
    - l’AOP è basata sull’uso di aspetti, specifici per la gestione di alcune responsabilità, che sono mantenuti separati dal resto del codice



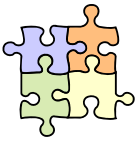
## Defer binding time (2)

- Tattiche per effettuare modifiche a build time
  - *Use component replacement*
    - uso di un componente che implementa responsabilità specifiche per il sistema da costruire
- Tattiche per effettuare modifiche a deployment time
  - *Use configuration-time binding*
    - i servizi da selezionare devono essere stati “astratti” e generalizzati
    - l’infrastruttura usata deve consentire di selezionare i servizi o componenti da utilizzare al momento del deployment
- Tattiche per effettuare modifiche a initialization time
  - *Use resource files*
    - ad esempio, file di configurazione, che consentono di scegliere i parametri di esecuzione per un modulo

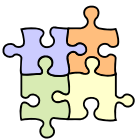
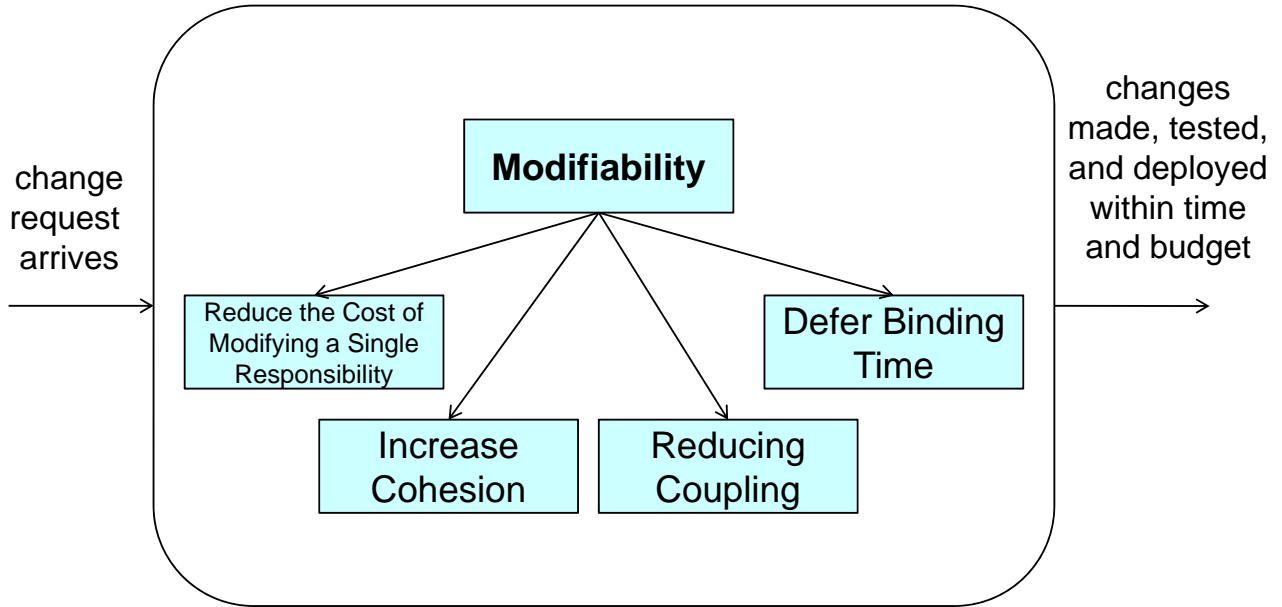


## Defer binding time (3)

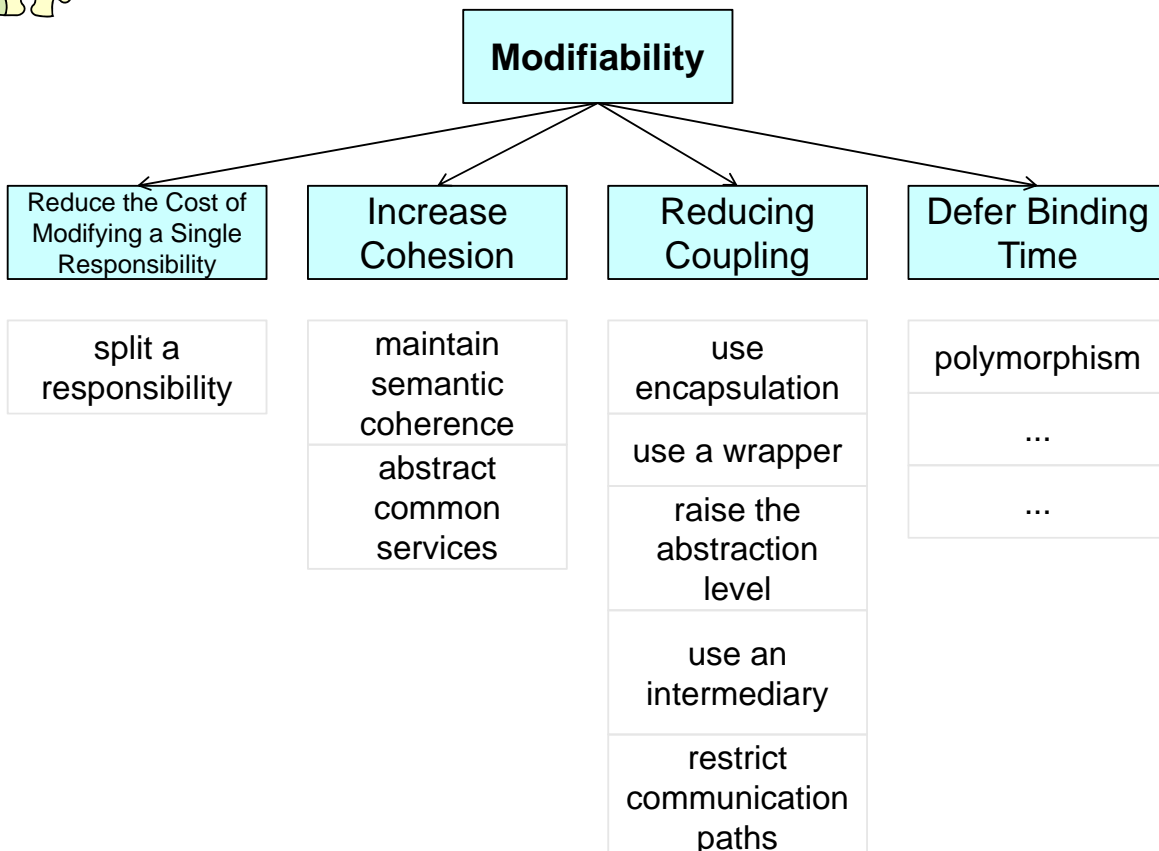
- Tattiche per effettuare modifiche a runtime
  - *Use runtime registration*
    - il servizio di registrazione è in grado di interagire con una varietà di elementi che si registrano
  - *Interpret parameters*
    - in un modulo sufficientemente generale e parametrizzato
  - *Use start-up time binding*
    - vengono usati dei parametri allo start-up
  - *Use runtime binding*
    - vengono usati dei parametri all’avvio di un modulo
  - *Use name servers*
    - il name server memorizza parametri che controllano il comportamento del sistema
  - *Use plug-ins*
    - sulla base di un servizio in grado di selezionare ed utilizzare un opportuno plug-in per il comportamento scelto

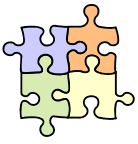


# - Tattiche per la modificabilità - sintesi



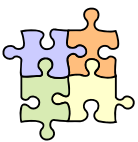
# Tattiche per la modificabilità - sintesi





## \* Discussione

- Sono state presentate alcune tattiche architettoniche – nel contesto del controllo di alcuni attributi di qualità
  - applicare una tattica vuol dire prendere una decisione di progetto per controllare un certo attributo di qualità
    - questo ha impatto sull'architettura – ovvero, sulla scelta degli elementi, delle loro responsabilità, o di come sono messi in relazione – e su come l'architettura supporta le qualità
  - la presentazione è stata qualitativa e informale
    - sono talvolta disponibili modelli che descrivono l'effetto quantitativo dell'applicazione delle tattiche
    - è inoltre possibile modellare queste decisioni di progetto – anche sulla base dell'applicazione di scenari
  - sono stati mostrati alcuni compromessi che occorre valutare nel controllo congiunto di più attributi di qualità
  - esistono altri attributi di qualità ed altre tattiche



## Discussione

- La progettazione di un'architettura richiede l'applicazione di una collezione di tattiche, per realizzare una **strategia architettonica**
  - l'approccio (solitamente di compromesso) adottato al fine di raggiungere gli obiettivi complessivi di qualità del sistema
- Un altro approccio fondamentale per la progettazione dell'architettura è basato sugli stili architettonici
  - uno stile architettonico può essere basato sull'applicazione di un certo numero di tattiche
  - nel seguito del corso questa affermazione sarà esemplificata nel contesto dello studio degli stili architettonici