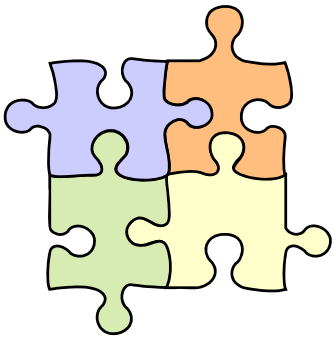
Luca Cabibbo



Architetture Software

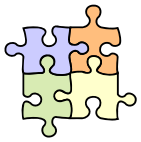
Tattiche architetturali
(prima parte)

Nella progettazione e costruzione
di sistemi complessi
introduci e mantieni opzioni
il più a lungo possibile.
Ne avrai bisogno.

Robert Spinrad

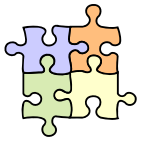
Dispensa ASW 310

ottobre 2014



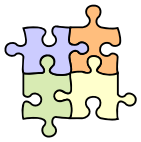
- Fonti

- [SAP] Part 2, Quality Attributes
  - Chapter 4, Understanding quality attributes
  - Chapter 5, Availability
  - Chapter 7, Modifiability
  - Chapter 8, Performance
  - Chapter 9, Security
- [Parnas] On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, 15, 1972
- [Bachmann, Bass, Nord] Modifiability Tactics, Technical report CMU/SEI-2007-TR-002, 2007
- [Scott, Kazman] Realizing and Refining Architectural Tactics: Availability, Technical report CMU/SEI-2009-TR-006, 2009
- [Kim, Kim, Lu, Park] Quality-driven architecture development using architectural tactics, Journal of Systems and Software, 82, 2009

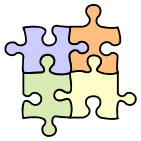


- Obiettivi e argomenti

- Obiettivi
  - introdurre le tattiche architetturali come guida per il raggiungimento di attributi di qualità
  - illustrare alcune tattiche architetturali
- Argomenti
  - tattiche architetturali
  - tattiche per le prestazioni
  - tattiche per la modificabilità
  - *tattiche per la disponibilità*
  - *tattiche per la sicurezza*
  - discussione

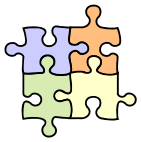


- Wordle



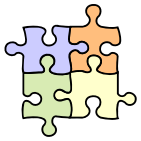
* Tattiche architetturali

- La comprensione di attributi e scenari di qualità consente di raccogliere e 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 architetturali che guidano il raggiungimento degli attributi di qualità
  - *tattiche architetturali* [SAP] – una tattica è una decisione di progetto che influenza il controllo di un attributo di qualità
  - *stili architetturali* [POSA] – uno stile di decomposizione architetturale – che di solito sostiene un certo numero di qualità, mediante l'applicazione di più tattiche



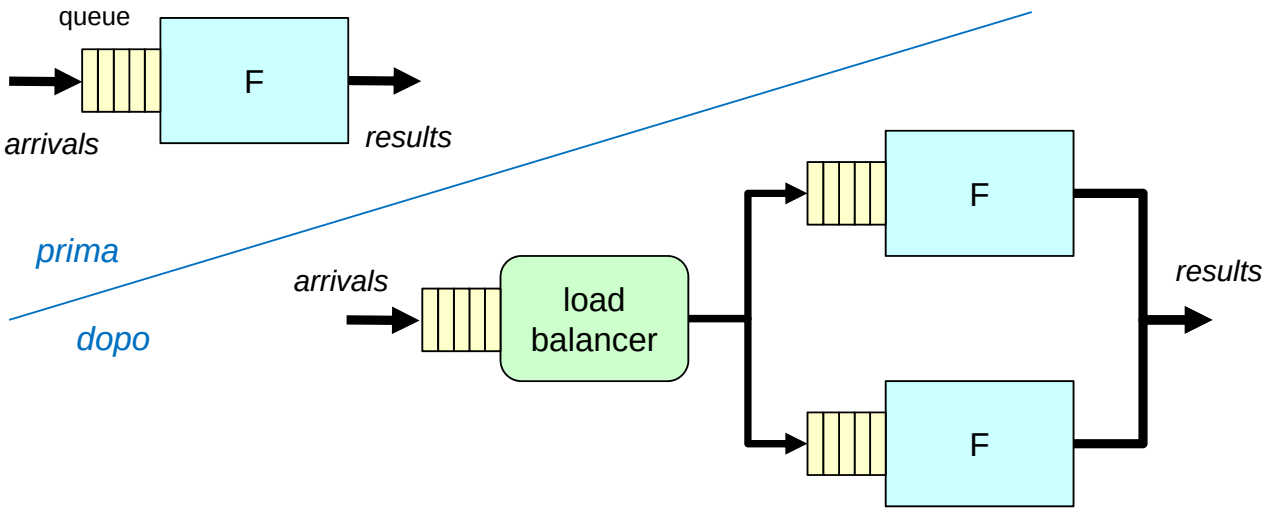
Tattiche architetturali

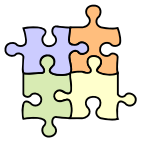
- Il progetto di un sistema consiste di un insieme di decisioni di progetto
  - alcune decisioni di progetto sostengono i requisiti funzionali – altre decisioni controllano gli attributi di qualità
- Una **tattica architetturale** (o, semplicemente, **tattica**) è una decisione di progetto fondamentale che influenza il controllo della risposta di un attributo di qualità [SAP]
  - detto in altro modo, una tattica è una trasformazione architetturale che ha effetto sul comportamento del sistema rispetto a un particolare attributo di qualità
  - ad esempio, migliorare il tempo di risposta di un sistema tramite l'uso di più istanze/copie di un elemento funzionale, oltre a un elemento per effettuare il bilanciamento del carico



Tattiche architetturali

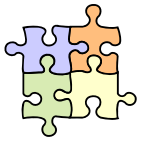
- Una tattica è una *trasformazione* architetturale che ha effetto su come il sistema controlla la risposta di un attributo di qualità
  - ad esempio, migliorare il tempo di risposta di un sistema tramite l'uso di più istanze/copie di un elemento funzionale, oltre a un elemento per effettuare il bilanciamento del carico





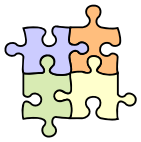
Le tattiche codificano esperienza

- Le tattiche architetturali hanno lo scopo di catturare e descrivere quello che gli architetti hanno fatto (e continuano a fare) in pratica
  - come i design pattern e gli stili architetturali, le tattiche descrivono delle decisioni progettuali (relative ad attributi di qualità) che sono state effettivamente applicate con successo in numerose situazioni – e che possono essere ancora applicate in nuove situazioni
- Le tattiche costituiscono un approccio a grana più fine rispetto agli stili architetturali
  - sono più semplici da comprendere e 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 architetturali



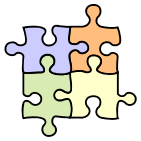
Tattiche architetturali

- 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?

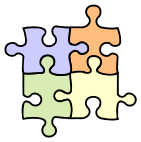
- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione le tattiche?
  - abbiamo già identificato gli scenari architetturali
  - abbiamo già scelto uno o più stili architetturali in grado di sostenere gli scenari architetturali più rilevanti
  - abbiamo poi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



A che punto siamo?

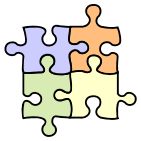
- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione le tattiche?
  - tuttavia, abbiamo anche capito che l'architettura corrente non è in grado di sostenere tutti gli scenari architetturali
  - tra di questi, abbiamo scelto uno o più scenari prioritari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

    questo è il punto in cui entrano in gioco le tattiche
  - possiamo selezionare e 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



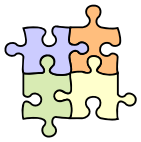
- Un catalogo di tattiche

- 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



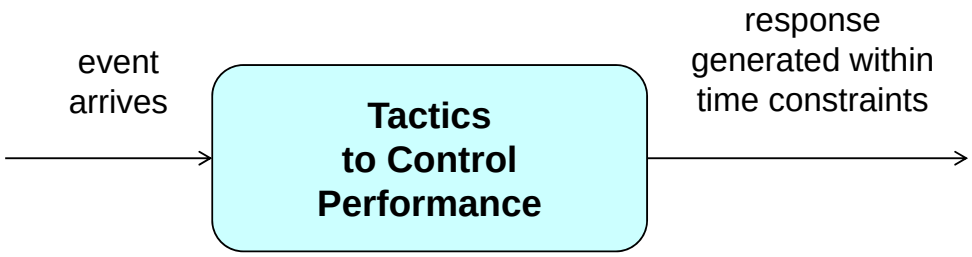
* Tattiche per le prestazioni

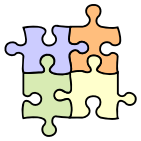
- Le *prestazioni* (performance) hanno a che fare con il *tempo di esecuzione* – ovvero, con la capacità del sistema di far fronte a requisiti temporali
  - si verificano degli eventi – richieste, messaggi, eventi temporali – e il sistema deve rispondere a questi eventi
  - gli scenari di qualità relativi 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 a 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 a un evento*
  - ovvero, il tempo entro cui viene generata la risposta a un evento



Tattiche per le prestazioni

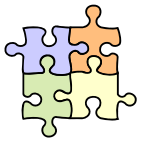
- In questa trattazione consideriamo tattiche per le prestazioni per controllare il tempo di risposta a un evento





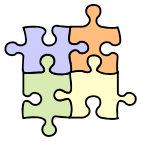
Ragionamento

- Il *tempo di risposta* a un evento è costituito da
  - *tempo di elaborazione* (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
  - *tempo di 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 a un evento



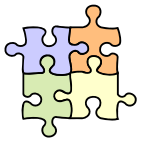
Categorie di tattiche per le prestazioni

- Due categorie principali di tattiche per le prestazioni
  - controllare la richiesta di risorse – *control resource demand*
    - queste tattiche cercano di ridurre il tempo dedicato alla gestione di un evento, dal lato della richiesta di risorse necessarie per elaborare l'evento
      - ad es., "migliora l'algoritmo"
  - gestione di risorse – *manage resources*
    - queste tattiche operano dal lato dell'offerta delle risorse, per gestire le elaborazioni in modo più efficace
      - ad es., "usa un processore più potente"



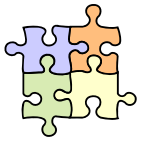
- Control resource demand (1)

- Tattiche volte a ridurre le risorse richieste per elaborare un evento
- *Increase resource efficiency*
  - l'elaborazione di un evento richiede l'esecuzione di un algoritmo – un miglioramento dell'efficienza temporale di questo algoritmo 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 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



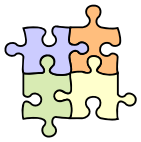
Control resource demand (2)

- Tattiche volte a ridurre le risorse richieste per elaborare un evento
- *Bound execution times*
  - in alcuni casi è possibile (e accettabile) porre un limite al tempo di esecuzione che può essere dedicato ad elaborare un evento – accettando un risultato che ha un'approssimazione prevedibile
  - ad esempio, limitare il numero di iterazioni in un algoritmo iterativo



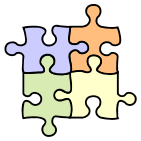
Control resource demand (3)

- Tattiche volte a ridurre il numero di eventi da elaborare
  - contesto: gli eventi da elaborare sono un flusso di misurazioni provenienti da un gruppo di sensori
- *Manage sampling rate*
  - in alcuni casi è possibile (e accettabile) ridurre la frequenza di campionamento con cui vengono monitorate le variabili ambientali – questo riduce il numero di eventi (richieste) da elaborare (ma costo di una qualche perdita di accuratezza)
  - spesso usata insieme alla tattica *Bound execution times*
- *Limit event response*
  - se non è possibile controllare l'arrivo di eventi generati esternamente, in alcuni casi è possibile (e accettabile) ignorare alcuni eventi dalla coda delle richieste – ad esempio, gestendo una coda degli eventi e limitandone la lunghezza



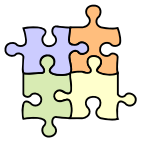
Control resource demand (4)

- Tattiche volte a ridurre il numero di eventi da elaborare
  - contesto: gli eventi da elaborare sono un flusso di misurazioni provenienti da un gruppo di sensori
- *Prioritize events*
  - se non tutti gli eventi sono ugualmente importanti, è possibile assegnare loro delle priorità – per poter poi decidere (se accettabile) di ignorare eventi a bassa priorità in caso di carenza di risorse disponibili nel sistema (che vanno monitorate)



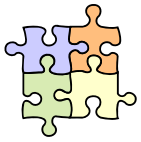
Osservazioni (1)

- 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 resource efficiency* ha effetto sulla caratterizzazione esterna di un'operazione di un elemento architetturale
    - ad es., *reduce overhead* può cambiare la scelta di alcuni elementi architetturali oppure della loro interconnessione



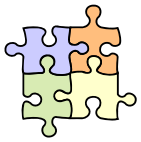
Osservazioni (2)

- 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



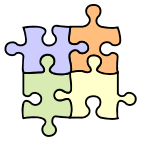
Osservazioni (3)

- 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 architetturali
  - 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



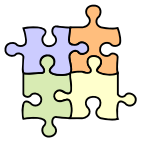
- Manage resources (1)

- Tattiche per la gestione delle risorse
- *Increase resources*
  - risorse (processori, memorie, reti, ...) addizionali o più veloci hanno il potenziale per ridurre il tempo di risposta
  - è la tattica più semplice – a fronte di un ovvio aumento dei costi
- *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 differente
  - per sfruttare al meglio la concorrenza, è importante anche allocare in modo opportuno i thread alle risorse (load balancing)



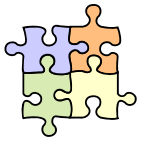
Manage resources (2)

- *Maintain multiple copies of 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 – insieme a un meccanismo di load balancing
    - creando un thread per ciascun evento individuale diverso, per elaborare più eventi diversi in modo concorrente
- *Maintain multiple copies of data*
  - è anche possibile replicare dei dati, e consentirne l'accesso concorrente – ad esempio, mediante meccanismi di caching o di *data replication*
    - poiché esistono più copie degli stessi dati, potrebbe essere necessario assegnare della responsabilità aggiuntive (cioè, non inizialmente presenti) per mantenere opportunamente consistenti e sincronizzate le varie copie di ciascun dato

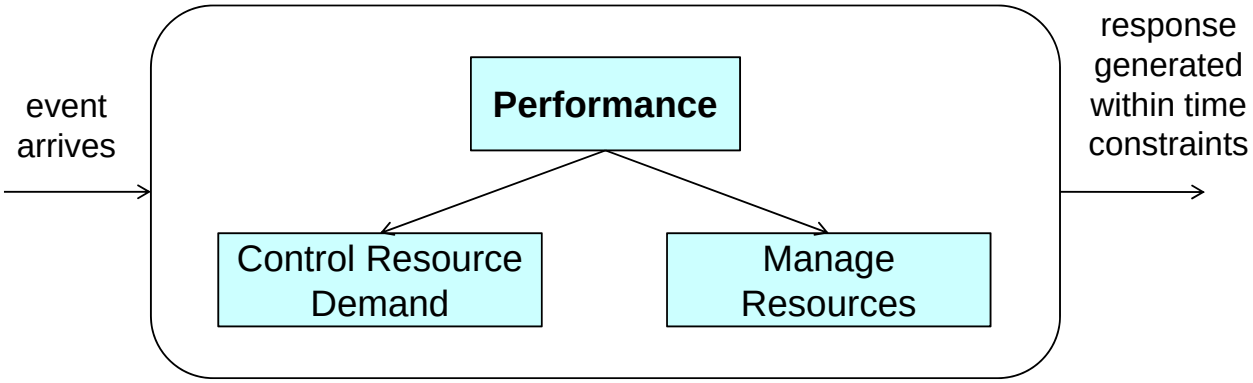


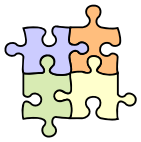
Manage resources (3)

- *Bound queue size*
  - questa tattica controlla il numero massimo di eventi posti in coda – comunemente usata con *Limit event response*
  - bisogna gestire la responsabilità di decidere cosa fare in caso di trabocco delle code
- *Schedule resources*
  - in caso di contesa di risorse, l'arbitraggio e la schedulazione delle risorse può consentirne un loro utilizzo più efficiente
  - è ad esempio possibile schedulare processori, buffer e reti
  - sono possibile diverse strategie – ad esempio, FIFO, fixed priority, dynamic priority, static scheduling – adatte a situazioni diverse

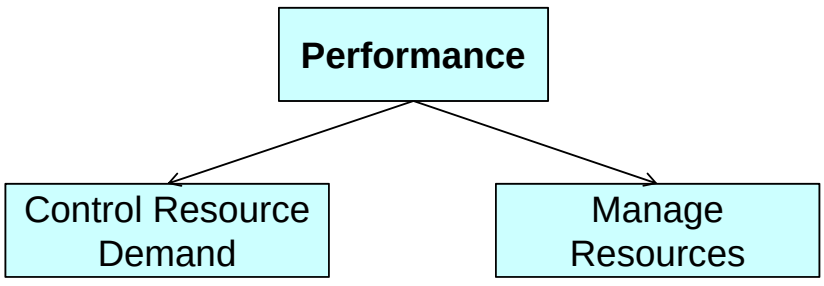


- Tattiche per le prestazioni – sintesi

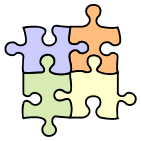




Tattiche per le prestazioni – sintesi

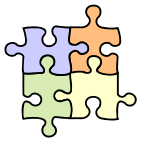


manage sampling rate	reduce overhead	increase resources	bound queue sizes
limit event response	bound execution times	introduce concurrency	schedule resources
prioritize events	increase resource efficiency	maintain multiple copies of computations	
		maintain multiple copies of data	



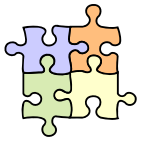
* Tattiche per la modificabilità

- 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
  - la maggior parte dei costi di un tipico sistema software si verificano dopo che il sistema è stato inizialmente rilasciato
  - i cambiamenti del software sono comuni e ubiqui
    - per aggiungere, modificare o rimuovere funzionalità
    - per correggere difetti e per migliorare alcune qualità
    - per migliorare l'esperienza d'uso del sistema
    - per abbracciare nuove tecnologie, nuove piattaforme, nuovi protocolli e standard
    - per far interoperare sistemi diversi – anche se non sono mai stati progettati per farlo
    - …



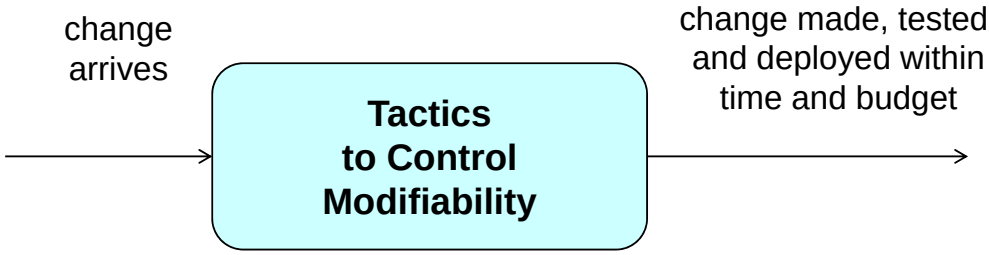
Tattiche per la modificabilità

- Nella pianificazione per i cambiamenti, ci sono diverse dimensioni che vanno prese in considerazione
  - *che cosa potrà cambiare?* qui 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
  - *qual è la probabilità del cambiamento?* non è possibile progettare un sistema per essere facilmente modificabile a fronte di ogni possibile cambiamento
  - *chi effettuerà 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?* qui consideriamo solo il tempo umano

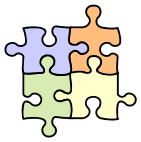


Tattiche per la modificabilità

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

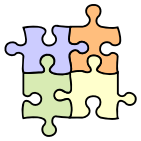


- Osservazione: nella progettazione dell'architettura
  - l'architetto non deve effettuare *ora* il cambiamento
  - piuttosto, deve garantire che *certi tipi* di cambiamenti attesi potranno, *in futuro*, essere gestiti facilmente



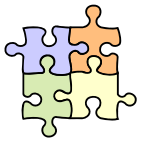
Ragionamento

- 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 Ri alle quali la modifica va propagata
    - queste modifiche avverranno nell'ambito di elementi che dipendono (direttamente o indirettamente) da E_R
    - questo costo va pesato rispetto alla probabilità che una modifica di R (E_R) richieda anche una modifica di Ri (E_{Ri})
  - costo del deployment della modifica
- In corrispondenza, diverse categorie di tattiche per la modificabilità, con l'obiettivo di ridurre questi contributi di costo



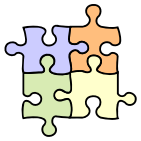
Modificabilità, accoppiamento e coesione

- La modificabilità di un sistema è correlata a misure/metriche come la coesione, l'accoppiamento e la dimensione degli elementi
  - la *coesione* è una misura della forza delle relazioni tra le responsabilità di uno specifico modulo
    - la coesione misura l'"unità dello scopo" del modulo
    - per questo, è anche una misura (inversa) della probabilità che uno scenario di cambiamento che ha impatto su una responsabilità del modulo richieda anche il cambiamento di responsabilità differenti (di altri moduli)



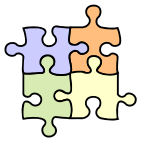
Modificabilità, accoppiamento e coesione

- La modificabilità di un sistema è correlata a misure/metriche come la coesione, l'accoppiamento e la dimensione degli elementi
  - l'*accoppiamento* è una misura della forza delle dipendenze tra moduli
    - per questo, è anche una misura (diretta) della probabilità che un cambiamento che ha impatto su una responsabilità richieda anche il cambiamento di responsabilità differenti
    - l'accoppiamento si riduce quando le relazioni tra elementi che non appartengono allo stesso modulo sono ridotte



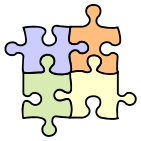
Modificabilità, accoppiamento e coesione

- La modificabilità di un sistema è correlata a misure/metriche come la coesione, l'accoppiamento e la dimensione degli elementi
  - in pratica, anche la *dimensione* di un modulo può avere impatto sul costo per modificare il modulo
    - decomporre un modulo in più parti può ridurre il costo dei cambiamenti – ma solo nella misura in cui la decomposizione riflette il tipo di cambiamenti che si possono verificare – ovvero, nella misura in cui la decomposizione aumenta la coesione e diminuisce (o non aumenta) l'accoppiamento



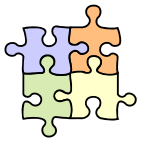
Modificabilità, accoppiamento e coesione

- La modificabilità di un sistema è correlata a misure/metriche come la coesione, l'accoppiamento e la dimensione degli elementi
  - 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



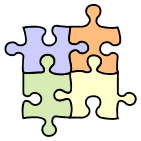
Parentesi: Alcune forme di coesione

- 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



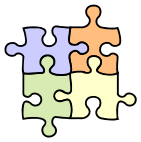
Parentesi: Alcune forme di accoppiamento

- 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



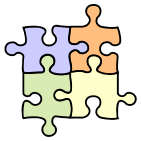
Tattiche come trasformazioni

- 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 – "semplifica, combina ed elimina"
    - 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) a un altro
  - l'obiettivo dell'applicazione delle tattiche per la modificabilità è identificare un'assegnazione di responsabilità che minimizzi il costo dei cambiamenti attesi



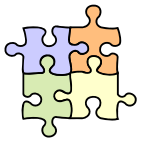
Categorie di tattiche per la modificabilità

- Categorie principali di tattiche per la modificabilità
  - *reduce size of a module*
    - per ridurre il costo di modificare una singola responsabilità
  - *increase cohesion*
    - per ridurre il costo dei cambiamenti intervenendo sulla coesione del sistema
  - *reduce coupling*
    - per ridurre il costo dei cambiamenti intervenendo sull'accoppiamento del sistema, per prevenire una propagazione a cascata delle modifiche
  - *defer binding*
    - per controllare il costo e il tempo richiesto dal deployment della modifica

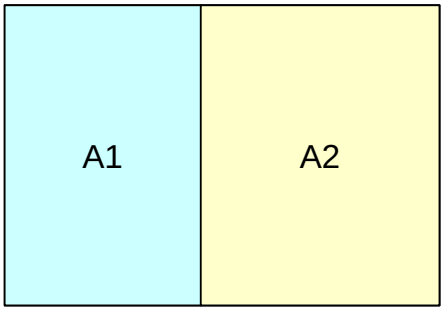


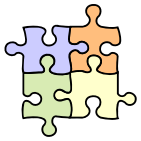
- Reduce the size of a module

- Un approccio di base per ridurre il costo di una modifica (di una singola responsabilità) è separare le responsabilità in base ai cambiamenti previsti
- *Split module* (era *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



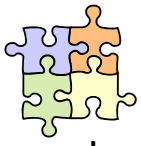
Split module





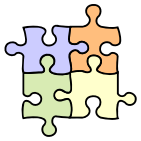
Esempio: Split module

- Esempio
  - il nostro sistema deve interagire con un altro sistema (ad es., esterno) per fruire di un servizio
  - cambiamento atteso: fruire il servizio da un sistema diverso
  - questo cambiamento può essere gestito mediante un Adapter – il cambiamento atteso viene isolato utilizzando un'interfaccia e il polimorfismo
  - nel momento in cui il servizio deve essere fruito da un altro sistema, è possibile definire un nuovo Adapter – scrivendo nuovo codice, ma senza modificare il codice esistente
    - in alcuni casi, questo è il modo preferibile per gestire un cambiamento



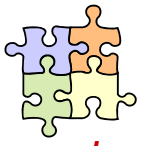
- Increase cohesion

- 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 e il costo di implementare il cambiamento, ridurre le modifiche a 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



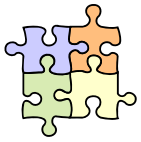
Increase cohesion (1)

- *Increase semantic coherence*
  - la *coerenza semantica* si riferisce alle relazioni tra le responsabilità assegnate a un modulo anche con riferimento a 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 e 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 a un insieme di cambiamenti previsti

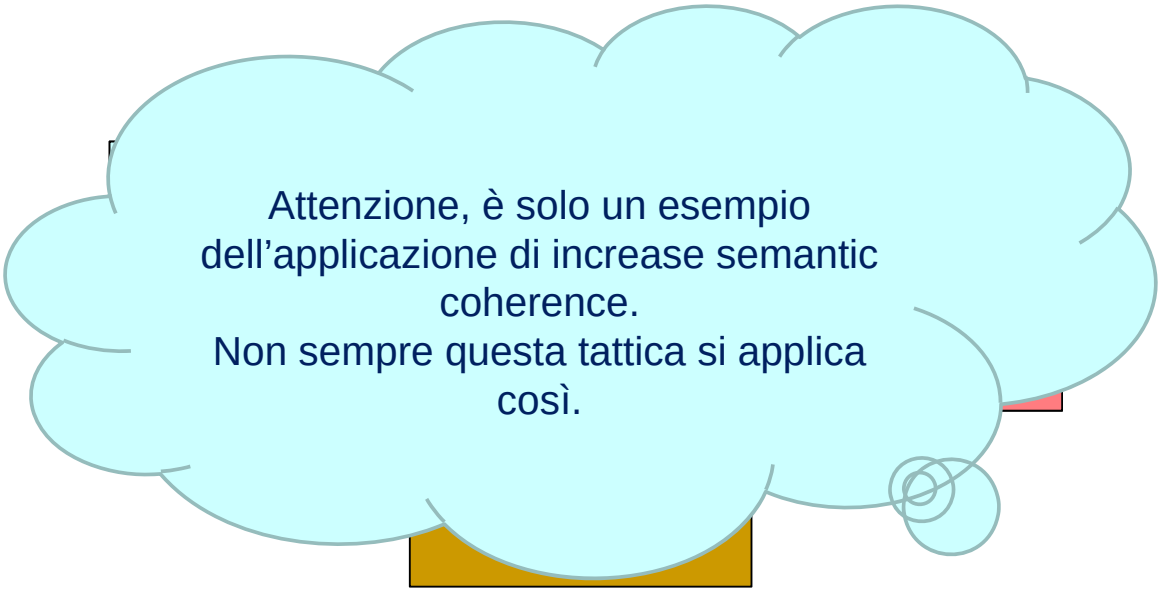


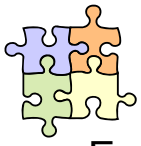
Increase semantic coherence

- *Increase semantic coherence*
  - ad esempio, siano A e B delle responsabilità che sono coese rispetto ad un qualche criterio, ad esempio funzionale
  - sia M un cambiamento atteso che si ripercuote su A'⊂A e B'⊂B
  - in questo caso, 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



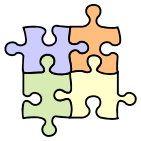
Increase semantic coherence





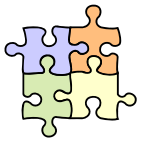
Esempio: Increase semantic coherence

- Esempio
  - protocolli di rete
  - cambiamenti attesi: definire nuovi protocolli applicativi, variare protocolli esistenti, fornire nuove implementazioni dei protocolli con riferimento a nuovi tipi di hardware
  - i protocolli di rete sono implementati distribuendo i diversi tipi di responsabilità in una pila di strati – e raggruppando le responsabilità in modo che abbiano qualche forma di coerenza semantica
    - ad esempio, responsabilità relative all'hardware sono allocate nello strato dei protocolli a livello fisico, e non con lo strato dei protocolli applicativi
  - poiché ciascun cambiamento atteso è relativo a un singolo tipo di responsabilità, potrà essere gestito nell'ambito di un singolo strato



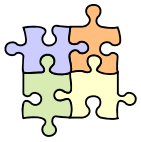
Un criterio per la decomposizione in moduli

- Un celebre articolo di [Parnas] del 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"
  - in questo criterio è possibile trovare due contributi significativi
    - ciascuna decisione di progetto difficile oppure soggetta a cambiamento – in particolare, relativa a un cambiamento previsto – va assegnata a un modulo diverso – "increase semantic coherence"
    - queste decisioni vanno nascoste ad altri moduli – è un suggerimento della tattica "encapsulate"



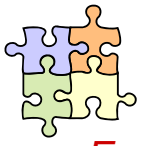
- Reduce coupling

- 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



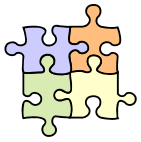
Forme di accoppiamento/dipendenza

- 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 e 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 del servizi/dei dati – B dipende dalla qualità del servizio/dei dati offerti da A
  - esistenza – B dipende dall'esistenza di A

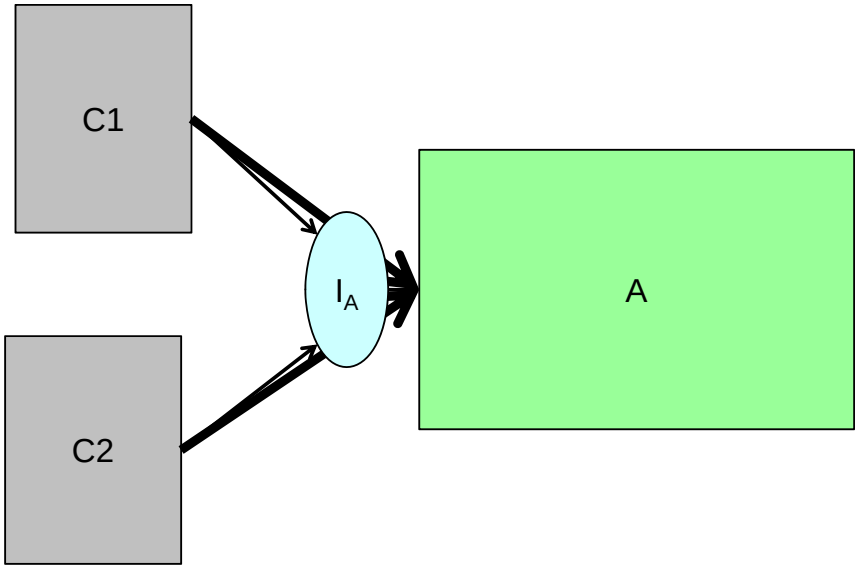


Reduce coupling (1)

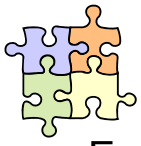
- *Encapsulate*
  - è comune decomporre le responsabilità di un elemento in parti più piccole, scegliendo alcune parti da rendere pubbliche e altre da mantenere private – le responsabilità pubbliche sono rese disponibili ad altri moduli mediante apposite interfacce
  - scopo dell'*incapsulamento* di un elemento A è ridurre la probabilità di propagazione dei cambiamenti nell'elemento A verso altri elementi
    - ciò richiede una separazione netta tra interfacce e 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]



Encapsulate

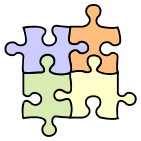


- L'uso dell'interfaccia I_A
  - riduce l'accoppiamento tra C1 e C2 ed A
  - protegge C1 e C2 da cambiamenti nell'implementazione di A



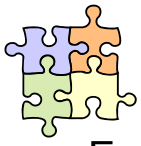
Esempio: Encapsulate

- Esempio
  - nella pila dei protocolli di rete, ogni protocollo è implementato sulla base di servizi offerti dallo strato inferiore
  - questi servizi sono fruiti sulla base di un'interfaccia – e indipendenti dalla specifica implementazione – e questa indipendenza favorisce la modificabilità dei diversi servizi
  - l'accesso a una base di dati avviene sulla base di uno schema logico (tabelle, colonne, chiavi, ecc.) – che incapsula lo schema fisico della base di dati
  - è possibile intervenire sullo schema fisico della base di dati per ottimizzare l'accesso alla base di dati stessa – senza doverne cambiare lo schema logico – né le applicazioni che la accedono



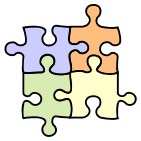
Reduce coupling (2)

- *Use an intermediary*
  - un intermediario può rompere la dipendenza tra un elemento A e 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 esempi 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



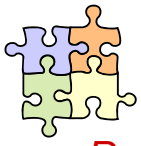
Esempio: Use an intermediary

- Esempio
  - l'accesso a una base di dati da parte di un'applicazione avviene mediante il suo schema esterno (viste sullo schema logico) – lo schema esterno è un intermediario che fornisce una vista dei dati della base di dati specifica per l'applicazione
  - in alcuni casi, il cambiamento di un'applicazione può essere gestito ridefinendo il suo schema esterno – ma senza cambiare lo schema logico della base di dati
  - in casi più complessi, è necessario cambiare anche lo schema logico della base di dati – ma le altre applicazioni possono essere protette da questo cambiamento adeguando i loro schemi esterni



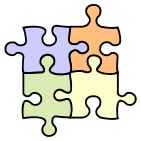
Reduce coupling (3)

- *Restrict dependencies*
  - si tratta di un caso speciale dell'uso di un intermediario
  - questa tattica consiste nel rimuovere una dipendenza relativa a una necessità di comunicazione, incanalando questa comunicazione in un intermediario
  - ad esempio, applicata nell'architettura a strati stretta



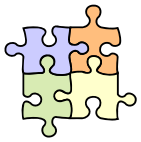
Reduce coupling (4)

- *Refactor*
  - questa tattica generalizza la tecnica del refactoring (sviluppata nel contesto dei metodi agili) ad un contesto architetturale
  - in generale, ha lo scopo di ridurre duplicazioni nel codice (che sono una forma di accoppiamento)
  - ad esempio, quando un cambiamento ha impatto su due moduli – perché hanno delle responsabilità duplicate (almeno in modo parziale)
    - in questo caso, queste responsabilità vanno estratte dai due moduli e messe a "fattor" comune
    - la co-locazione di responsabilità comuni riduce l'accoppiamento – dato che l'accoppiamento misura le dipendenze tra moduli differenti

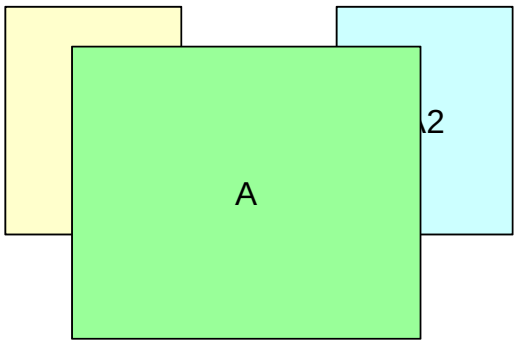


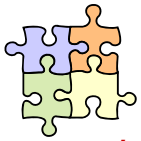
Refactor

- *Refactor*
  - alcuni esempi
    - un modulo offre un servizio A' – un altro modulo offre 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



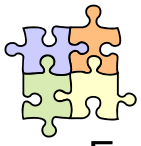
Refactor





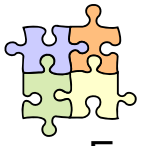
Reduce coupling (5)

- *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 a 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 tutte le versioni specifiche dei servizi
  - un modo comune nel rendere un servizio più astratto è basato su una parametrizzazione delle sue attività
    - i clienti del servizio fanno richieste specificando parametri – spesso definiti in termini di un "linguaggio" opportuno
    - obiettivo è fare in modo che molti cambiamenti possano essere gestiti cambiando semplicemente la scelta dei parametri



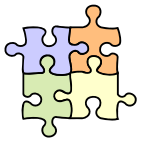
Esempio: Abstract common services

- Esempio
  - una base di dati viene acceduta mediante istruzioni SQL – e non mediante operazioni procedurali di accesso ai dati
  - alcuni componenti offrono un'interfaccia a messaggi/documenti anziché un'interfaccia procedurale – l'accoppiamento con un componente che ha un'interfaccia a messaggi è considerato solitamente inferiore all'accoppiamento con un componente che ha un'interfaccia procedurale



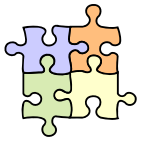
Esempio: Abstract common services

- Esempio
  - un browser web – che supporta plug-in dedicati alla presentazione di contenuti specifici
  - cambiamenti attesi: cambiamenti (miglioramenti) nelle funzionalità base del browser devono poter essere fruite dai diversi plug-in – e non devono richiedere un loro aggiornamento
  - il browser può fornire ai suoi plug-in un insieme di servizi atomici comuni – che i diversi plug-in possono utilizzare per realizzare funzionalità più complesse o specifiche
  - questo avverrà più facilmente se i servizi offerti dal browser sono effettivamente comuni e astratti, ovvero indipendenti da ogni specifico (plug-in) consumatore dei servizi



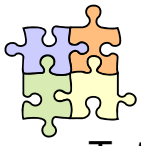
- Defer binding

- Il costo di una modifica dipende anche dal tempo in cui la modifica viene effettuata – questo porta a un'ulteriore categoria di tattiche per la modificabilità – che sono in parte trasversali alle precedenti
  - sono tattiche con lo scopo di controllare il costo e 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



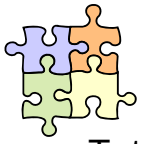
Defer binding (1)

- 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



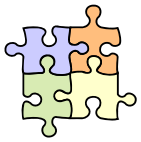
Defer binding (2)

- 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

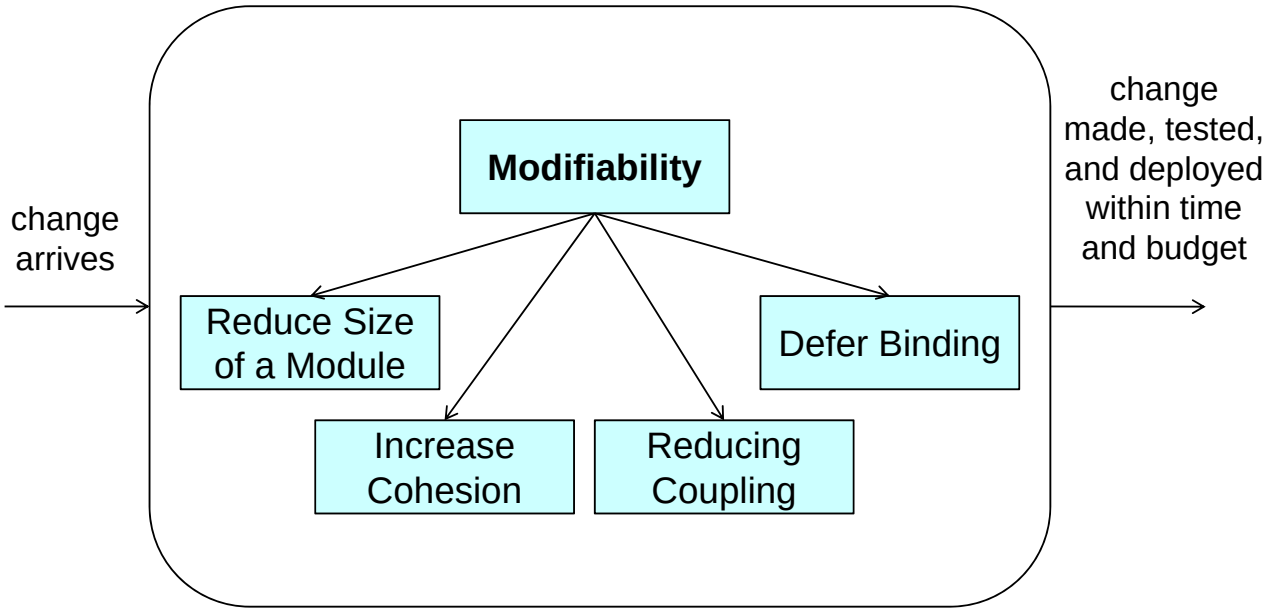


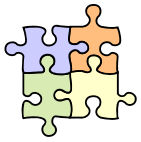
Defer binding (3)

- 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

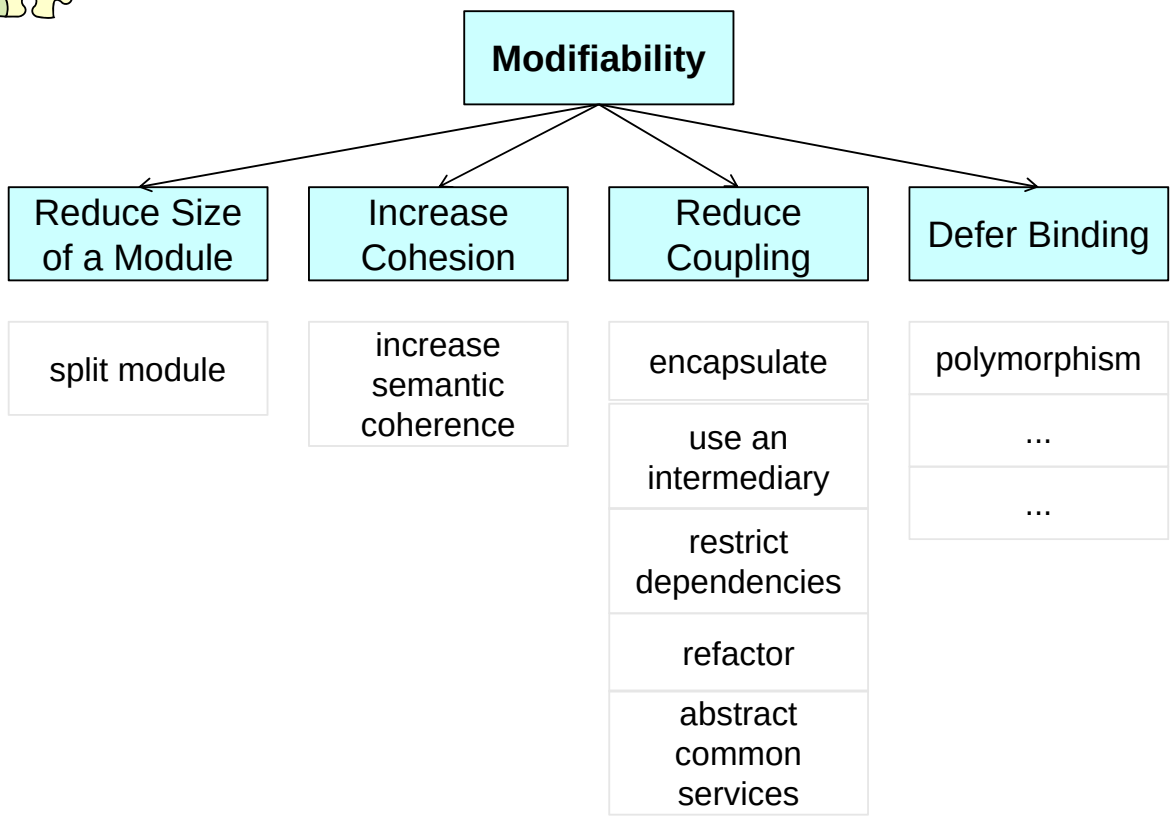


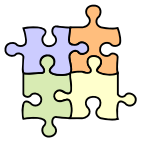
- Tattiche per la modificabilità – sintesi





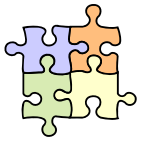
Tattiche per la modificabilità – sintesi





* Discussione

- Sono state presentate alcune tattiche architetturali – 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 sostiene 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 e altre tattiche



Discussione

- La progettazione di un'architettura richiede l'applicazione di 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
- Un altro approccio fondamentale per la progettazione dell'architettura è basato sugli stili architetturali
  - uno stile architetturale 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 architetturali