



Luca Cabibbo  
Architettura  
dei Sistemi  
Software

# Modificabilità

**dispensa asw230**  
ottobre 2023

*Everything changes  
and nothing stands still.*  
*Heraclitus*

1

Modificabilità

Luca Cabibbo ASW



## - Riferimenti

- ❑ Luca Cabibbo. **Architettura del Software: Strutture e Qualità**. Edizioni Efestò, 2021.
  - Capitolo 9, **Modificabilità**
- ❑ Parnas, D.L. **On the Criteria To Be Used in Decomposing Systems into Modules**. Communications of the ACM. 1972.
- ❑ Bachmann, F., Bass, L., and Nord, R. **Modifiability Tactics**. Technical report CMU/SEI-2007-TR-002. 2007.
- ❑ Martin, R.C. and Martin, M. **Agile Principles, Patterns, and Practices in C#**. Prentice Hall, 2007.

2

Modificabilità

Luca Cabibbo ASW



## - Obiettivi e argomenti

### □ Obiettivi

- presentare la qualità della modificabilità
- illustrare alcune attività e tattiche per la progettazione per la modificabilità

### □ Argomenti

- modificabilità
- progettare per la modificabilità
- discussione



## \* Modificabilità

### □ **Modificabilità** (*modifiability*)

- la capacità del sistema di essere flessibile a fronte di cambiamenti inevitabili dopo il suo rilascio iniziale, in modo bilanciato rispetto ai costi di fornire tale flessibilità

### □ La modificabilità

- riguarda i *cambiamenti* e il costo per realizzare tali cambiamenti
- misura la facilità con cui un sistema software può accomodare cambiamenti
- una qualità relativa all'*evoluzione* (*evolution*) del sistema
- la modificabilità è spesso importante
  - un sistema di successo deve essere in grado di soddisfare non solo gli obiettivi di business correnti di un'organizzazione, ma anche quelli **futuri**
  - in alcuni sistemi è richiesta un'evoluzione continua



- I cambiamenti del software sono ubiqui e comuni
  - possono riguardare diversi aspetti
    - ad es., correggere difetti, aggiungere funzionalità, migliorare l'usabilità, utilizzare nuove tecnologie, interagire con altri sistemi esterni, ...
  - possono avvenire in qualunque momento della vita di un sistema software
  - la modificabilità riguarda soprattutto i cambiamenti che avverranno *dopo* che il sistema è stato rilasciato



## Scenari di modificabilità

- Scenari (requisiti architetturealmente significativi) di modificabilità
  - ciascuno è relativo a un (tipo di) **cambiamento atteso**
  - deve essere relativo a un cambiamento rilevante – che va gestito in modo efficace per sostenere il business di un'organizzazione
  - uno scenario deve includere anche una specifica del costo previsto per effettuare la modifica richiesta
- È utile prendere in considerazione diverse domande
  - che cosa potrà cambiare?
  - quanto è importante il cambiamento? quanto è probabile?
  - quando sarà effettuato il cambiamento? chi lo effettuerà?
  - come viene misurato il costo del cambiamento?



## Scenari di modificabilità

- ❑ Che cosa potrà cambiare?
  - ipotizziamo che l'unità di modifica sia una generica “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
- ❑ Quanto è importante il cambiamento? Qual è la probabilità che il cambiamento venga effettivamente richiesto?
  - vanno identificati i cambiamenti più significativi e più probabili
- ❑ Quando sarà effettuato il cambiamento? Chi lo effettuerà?
  - i cambiamenti possono essere effettuati in momenti e modi diversi e da persone differenti – in particolare, dagli *sviluppatori* (Dev), dagli *operatori* (Ops) o dagli *utenti finali* del sistema
  - qui ci concentriamo soprattutto sui cambiamenti che devono essere effettuati dagli sviluppatori

7

Modificabilità

Luca Cabibbo ASW



## Scenari di modificabilità

- ❑ Come viene misurato il costo del cambiamento?
  - il “costo” speso per un cambiamento comprende il *tempo* e il *costo* richiesto per *implementare*, *verificare* e *rilasciare* il cambiamento
  - qui ci concentriamo soprattutto sul costo per implementare un cambiamento (che deve essere effettuato dagli sviluppatori)
  - anche gli aspetti della verifica e del rilascio sono importanti – ma verranno trattati più avanti nel corso

8

Modificabilità

Luca Cabibbo ASW



## - Considerazioni sulla modificabilità

- Contributi principali al **costo (atteso) richiesto per modificare una certa responsabilità R**
  - costo (atteso) della modifica relativa direttamente alla singola responsabilità R
    - nell'elemento  $E_R$  a cui è assegnata la responsabilità R
  - costo (atteso) della modifica di tutte le responsabilità  $R_i$  a cui la modifica va propagata
    - negli elementi che dipendono (direttamente o indirettamente) da  $E_R$
    - questo costo va pesato rispetto alla probabilità che una modifica di R (in  $E_R$ ) richieda anche una modifica di  $R_i$  (in  $E_{R_i}$ )
  - *costo della verifica della modifica*
  - *costo del rilascio della modifica*



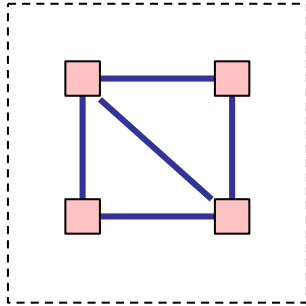
## Modificabilità, accoppiamento e coesione

- 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 – ovvero, dell'“unità di scopo” del modulo
  - l'**accoppiamento** è una misura della forza delle dipendenze tra moduli
  - anche la **dimensione** di un modulo può avere impatto sul costo per modificare il modulo

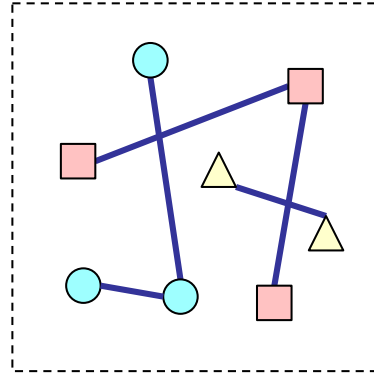


# Coesione

- La **coesione** è una misura della forza delle relazioni tra le responsabilità di uno specifico modulo – ovvero, dell’“unità di scopo” del modulo



coesione più alta



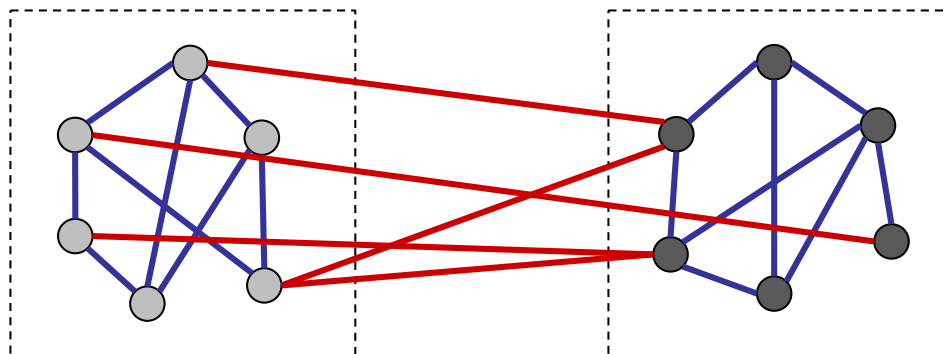
coesione più bassa



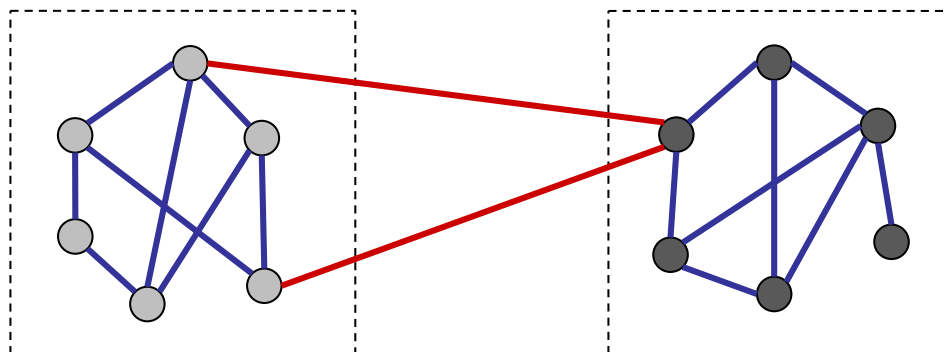
# Accoppiamento

- L’**accoppiamento** è una misura della forza delle dipendenze tra moduli

accoppiamento più alto



accoppiamento più basso





# 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 prima approssimazione
    - il costo della modifica della responsabilità  $R$  nell'ambito dell'elemento  $E_R$  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  $E_R$
    - il costo della modifica di un elemento  $E$  è in genere commisurato (in modo inverso) anche alla dimensione di  $E$
  - dunque, la coesione deve essere alta, l'accoppiamento deve essere basso e i moduli (abbastanza) piccoli

13

Modificabilità

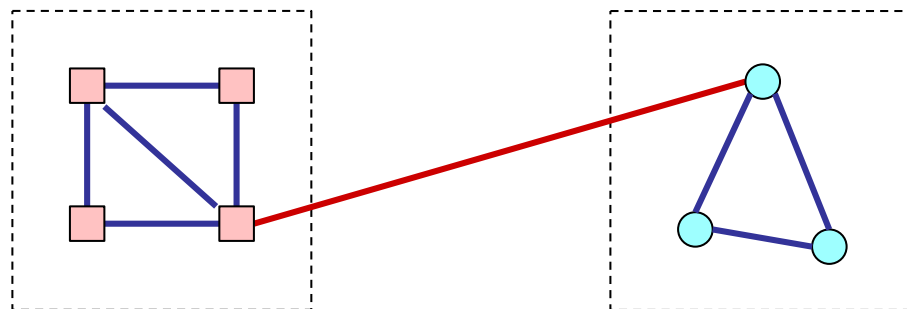
Luca Cabibbo ASW



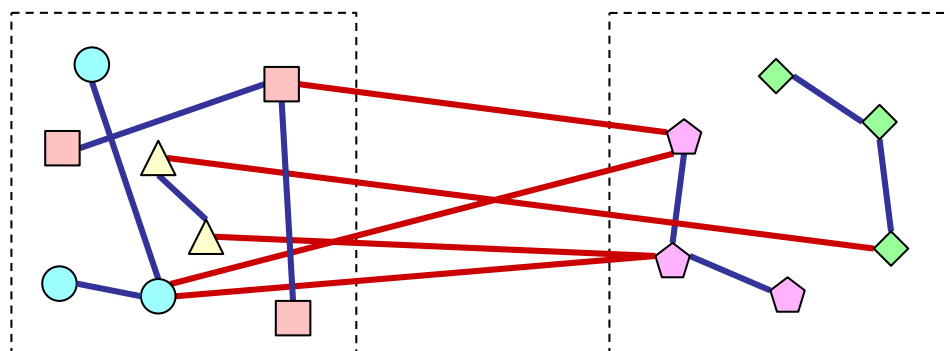
# Modularità

- La **modularità** è la proprietà di un sistema di essere composto da moduli che sono fortemente coesi e debolmente accoppiati

modulare  
(coesione alta e  
accoppiamento  
basso)



non modulare  
(coesione bassa e  
accoppiamento  
alto)



14

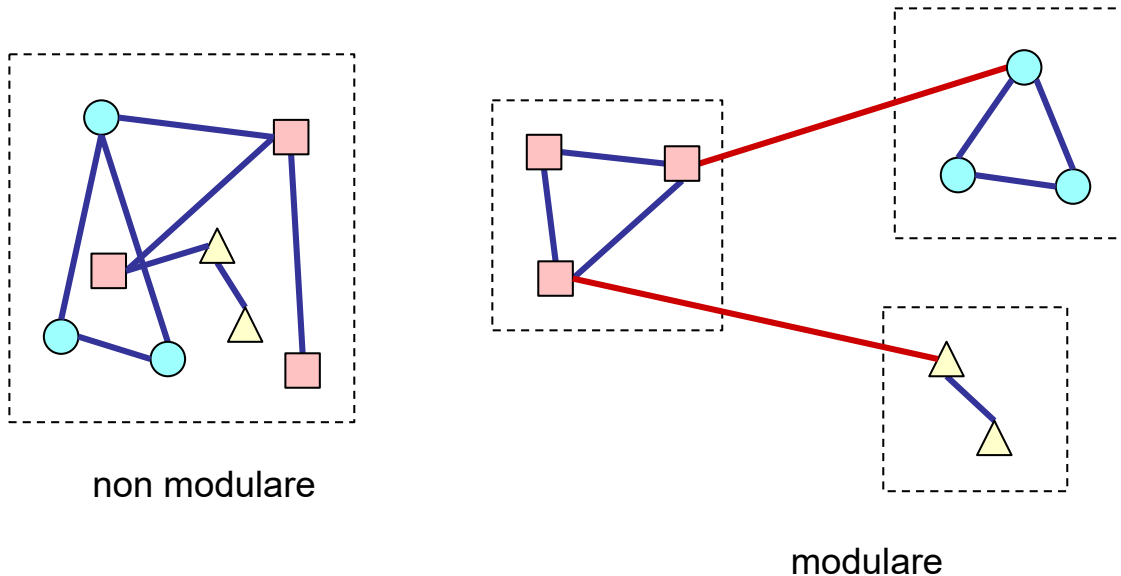
Modificabilità

Luca Cabibbo ASW



## Modularità

- La **modularità** è la proprietà di un sistema di essere composto da moduli che sono fortemente coesi e debolmente accoppiati



15

Modificabilità

Luca Cabibbo ASW



## Forme di coesione

- Esistono tante forme di coesione – eccone alcune, dalle più deboli (meno buone) alle più forti (più buone) (orientativamente)
  - coesione per pura coincidenza (cattiva!)
  - coesione temporale – gli elementi del modulo sono usati all'incirca nello stesso tempo
  - coesione logica – il modulo implementa delle funzionalità logicamente simili, ma implementate in modi indipendenti
  - coesione di comunicazione – gli elementi devono accedere agli stessi dati o dispositivi
  - coesione sequenziale – gli elementi del modulo sono usati in un ordine particolare
  - coesione funzionale – gli elementi contribuiscono a svolgere una singola funzione (forte!)
  - coesione dei dati – un modulo implementa un tipo di dato o una singola entità logica (forte!)

16

Modificabilità

Luca Cabibbo ASW





## Migliorare la coesione

- Come si migliora la coesione di un sistema?
  - cambiando la scelta dei moduli
  - cambiando l'assegnazione di responsabilità ai moduli



## Forme di accoppiamento

- Esistono tante forme di accoppiamento – eccone alcune, dalle più forti (più cattive) alle più deboli (meno cattive) (orientativamente)
  - accoppiamento di dati interni – un modulo accede e modifica direttamente i dati di un altro modulo
  - accoppiamento mediante dati globali – due o più moduli condividono dati globali
  - accoppiamento di controllo – l'ordine in cui vanno eseguite le operazioni definite in un modulo è controllato altrove
  - accoppiamento per estensione – un modulo implementa un'interfaccia specificata da un altro modulo oppure estende le funzionalità di un altro modulo
  - accoppiamento di componenti – un modulo conosce altri moduli o gestisce istanze di altri moduli
  - accoppiamento mediante interfaccia e parametri – un modulo richiede l'esecuzione di operazioni ad altri moduli



## Forme di accoppiamento

- Altre forme di accoppiamento – tra una coppia di elementi A e B
  - esistenza – A dipende dall'esistenza di B
  - spaziale – A dipende dalla locazione di B
  - temporale – A deve interagire con B in modo sincrono
  - nella qualità del servizio o dei dati – A dipende dalla qualità del servizio o dei dati offerti da B
  - di piattaforma – A deve essere realizzato con la stessa piattaforma o tecnologia di B
  - nel rilascio – se il rilascio di una nuova versione di B richiede anche il rilascio contestuale di una nuova versione di A
  - tra team di sviluppo – riguarda il coordinamento richiesto tra il team che sviluppa A e il team che sviluppa B



## Connascenza



- La **connascenza** è un tipo particolare di accoppiamento
  - due moduli sono connascenti se il cambiamento in un modulo richiede anche una modifica nell'altro per mantenere la correttezza complessiva del sistema
  - forme di connascenza statica – dalle più deboli alle più forti
    - connascenza di nome – i moduli devono essere d'accordo sul nome di un'entità
    - connascenza di tipo – i moduli devono essere d'accordo sul tipo di un'entità
    - connascenza di significato o di convenzione – i moduli devono essere d'accordo sul significato di certi valori
    - connascenza di posizione – i moduli devono essere d'accordo sull'ordine dei valori
    - connascenza di algoritmo – i moduli devono essere d'accordo su un algoritmo utilizzato



- La **connascenza** è un tipo particolare di accoppiamento
  - due moduli sono connascenti se il cambiamento in un modulo richiede anche una modifica nell'altro per mantenere la correttezza complessiva del sistema
  - forme di connascenza dinamica – dalle più deboli alle più forti – sono tutte più forti delle forme di connascenza statica
    - connascenza di esecuzione – riguarda l'ordine in cui devono essere eseguite più operazioni
    - connascenza di temporizzazione – riguarda la sincronizzazione tra thread e processi
    - connascenza di valori – riguarda valori che devono cambiare insieme in componenti diversi
    - connascenza di identità – riguarda l'identità di entità che devono essere accedute da più componenti
  - inoltre, si parla anche di connascenza sincrona e asincrona



## Migliorare l'accoppiamento

- Come si migliora l'accoppiamento di un sistema?
  - in genere l'accoppiamento tra moduli non può essere eliminato
  - però è spesso possibile
    - sostituire una forma di accoppiamento (connascenza) peggiore con una forma di accoppiamento (connascenza) migliore
    - spostare l'accoppiamento da un modulo a un altro modulo in cui l'accoppiamento è meno problematico



## Modificabilità, verifica e rilascio

- Altri contributi al costo di una modifica sono il costo della verifica e il costo del rilascio (delivery)
  - il costo della verifica (test) dipende dalla verificabilità del sistema – che esamineremo in un successivo capitolo
  - il costo del rilascio dipende da tanti aspetti – anche questi saranno discussi in capitoli successivi



## \* Progettare per la modificabilità

- Alcune attività nella progettazione per la modificabilità e l'evoluzione di un sistema [SSA]
  - identifica le necessità di evoluzione
    - identifica i cambiamenti attesi più rilevanti per il sistema, e valuta il loro potenziale impatto e la loro probabilità
    - decidi quando dovrà essere gestito ciascun cambiamento, chi dovrà farlo e come
  - valuta la modificabilità corrente del sistema
  - raffina l'architettura
  - la modificabilità ha un costo – pertanto “scegli le tue battaglie”



## - Tattiche per la modificabilità

- [SAP] propone tattiche per la modificabilità per controllare il tempo e il costo per implementare, verificare e rilasciare **un cambiamento atteso**
  - le attività “implementare, verificare e rilasciare” vanno intese come attività che dovranno essere svolte in futuro, dopo il rilascio iniziale del sistema
  - durante la progettazione dell’architettura, invece, un aspetto fondamentale è la possibilità di riorganizzare i moduli, le loro responsabilità e le loro dipendenze
    - “elimina, combina, riorganizza e semplifica”
  - l’obiettivo della progettazione per la modificabilità è identificare (adesso) un insieme di moduli, a cui sono assegnate delle responsabilità, in modo da minimizzare il costo (futuro) 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
  - ***defer binding***
    - per controllare il tempo e il modo in cui effettuare la modifica e il suo rilascio



## - Reduce the size of a module

- Un approccio di base per ridurre il costo delle modifiche è separare le responsabilità in base ai cambiamenti previsti
- **Split module**
  - sia R la responsabilità che è il target di uno specifico scenario di modificabilità – ed M il modulo a cui è assegnata la responsabilità R
  - se il modulo M comprende molte capacità/funzionalità (oltre a R), allora il costo della modifica sarà alto
    - se la modifica ha impatto solo su una porzione di M, allora si può ridurre il costo atteso della modifica decomponendo il modulo M in moduli più piccoli
  - un criterio per separare responsabilità in modo efficace è che i moduli piccoli possano essere modificati indipendentemente

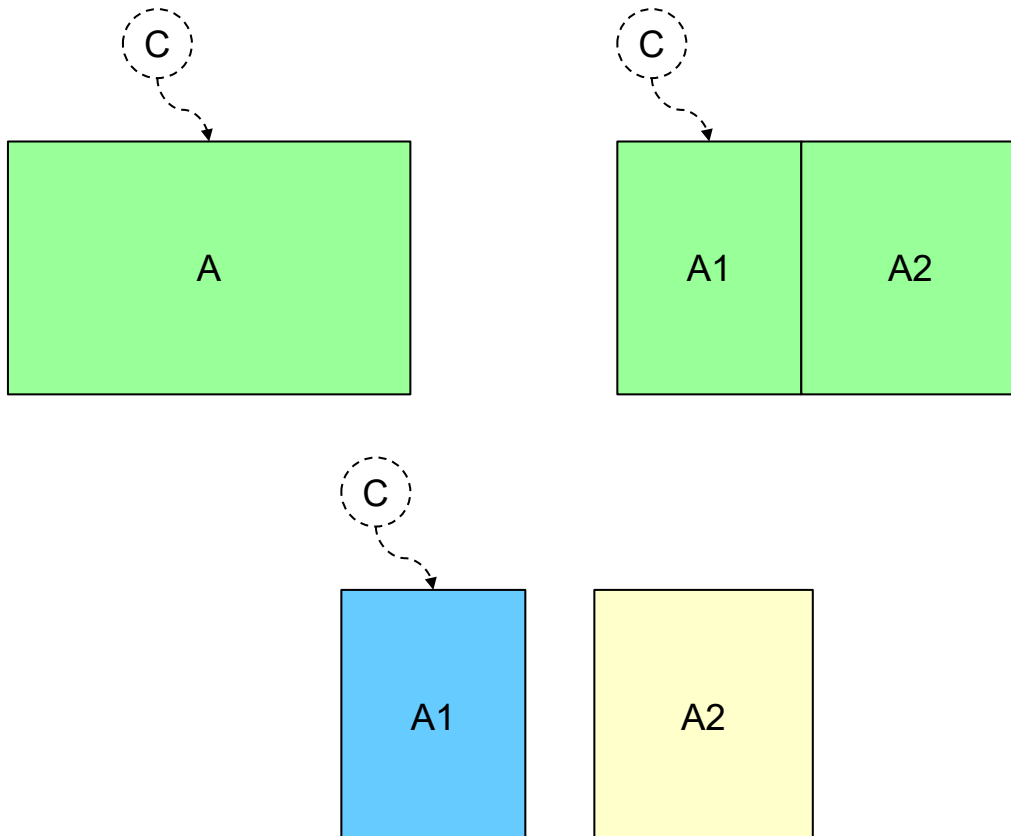
27

Modificabilità

Luca Cabibbo ASW



## Split module



28

Modificabilità

Luca Cabibbo ASW



## Esempio: Split module

### □ Esempio

- il sistema deve interagire con un sistema esterno per fruire di un servizio
- cambiamento atteso: fruire il servizio da un sistema esterno differente
- questo cambiamento può essere isolato utilizzando un'interfaccia e il polimorfismo e gestito mediante un Adapter



## - Increase cohesion

- Le tattiche per aumentare la coesione hanno di solito l'obiettivo di localizzare i cambiamenti
  - ridurre il numero di moduli sui quali un certo cambiamento si ripercuote direttamente può ridurre il costo della modifica



## Increase semantic coherence

- **Increase semantic coherence** (*Redistribute responsibilities*)
  - la coesione e l'accoppiamento sono solo un tentativo di misurare la modificabilità di un sistema
  - la **coerenza semantica** è una misura della forza delle relazioni tra le responsabilità assegnate a un modulo **anche con riferimento alla probabilità che i cambiamenti attesi abbiano impatto sulle responsabilità del modulo**
    - un modulo è semanticamente coerente se le sue responsabilità sono funzionalmente coese e se rientrano tutte nella portata di uno o più cambiamenti attesi
  - le responsabilità vanno inizialmente assegnate sulla base di opportune forme di coesione (e di accoppiamento)
  - poi, per ciascun modulo, vanno identificati i cambiamenti attesi che possono avere impatto su quel modulo – nel caso, alcune responsabilità vanno riassegnate

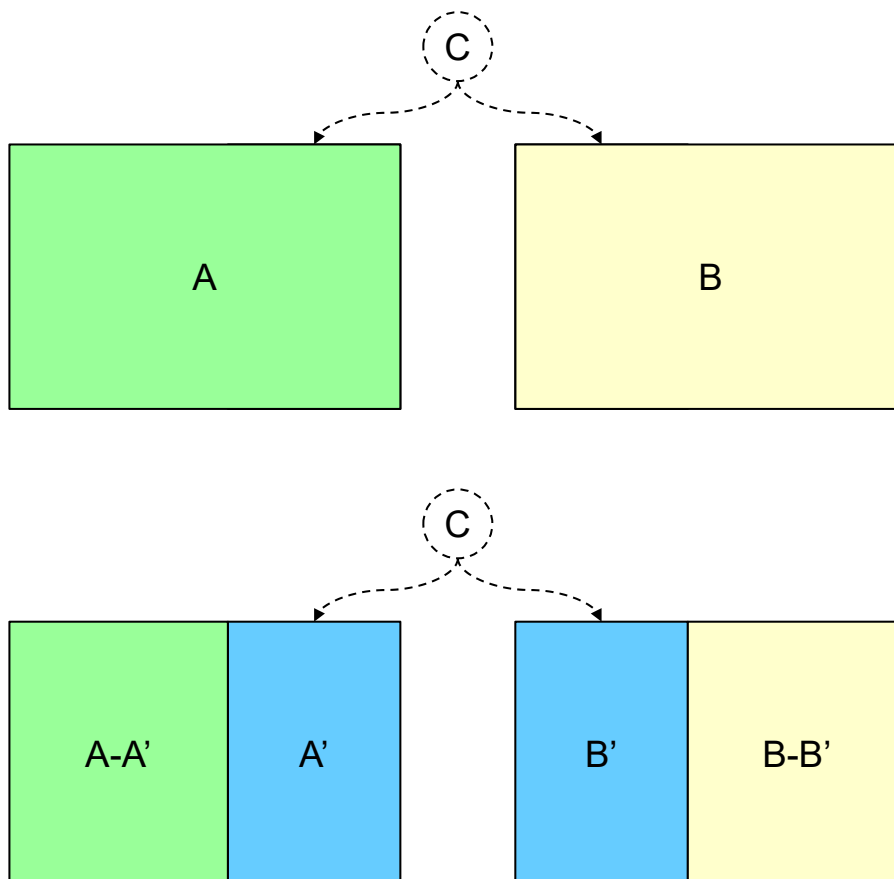
31

Modificabilità

Luca Cabibbo ASW



## Increase semantic coherence



32

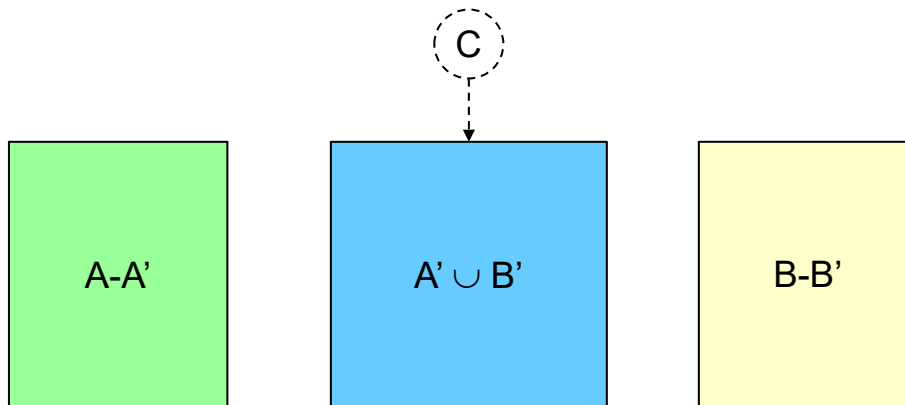
Modificabilità

Luca Cabibbo ASW





## 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 – le responsabilità sono ripartite tra gli strati raggruppandole in modo che abbiano qualche forma di coerenza semantica
- in genere, i cambiamenti attesi sono relativi a un singolo tipo di responsabilità – pertanto, ciascuno di questi cambiamenti 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”
  - due contributi significativi
    - ciascuna decisione di progetto difficile oppure soggetta a cambiamento va assegnata a un modulo diverso – “increase semantic coherence”
    - queste decisioni vanno nascoste ad altri moduli – “encapsulate”



## - Reduce coupling

- Le tattiche per ridurre l'accoppiamento hanno l'obiettivo di evitare i cosiddetti effetti a cascata – per ridurre il numero di moduli sui quali un certo cambiamento atteso si ripercuote *indirettamente*
  - un *effetto a cascata (ripple effect)* da una modifica è la necessità di effettuare un cambiamento a moduli su cui la modifica non si ripercuote direttamente
    - questo è in genere motivato da una qualche forma di *accoppiamento o dipendenza* tra questi moduli



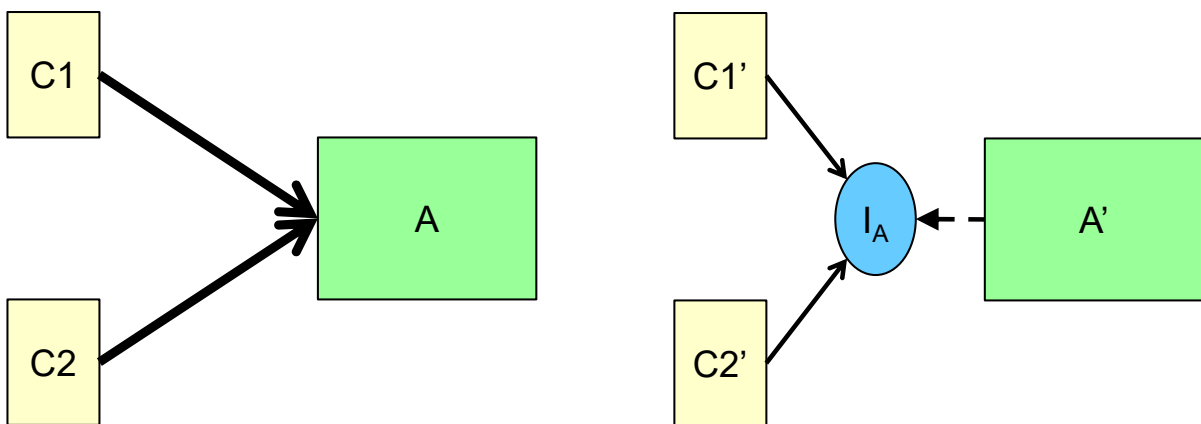
# Encapsulate

## Encapsulate

- l'**incapsulamento** di un elemento software separa in modo esplicito la sua interfaccia – pubblica e stabile – dalla sua implementazione – privata e soggetta ad evoluzioni e variazioni
  - le interazioni tra elementi devono avvenire solo tramite le loro **interfacce pubbliche**
- l'incapsulamento di un elemento A, basato su un'interfaccia **stabile**, ha lo scopo di ridurre la probabilità di propagazione dei cambiamenti nell'elemento A verso altri elementi
- l'incapsulamento ha lo scopo di isolare i cambiamenti attesi in un modulo, nascondendoli ad altri moduli [Parnas]



# Encapsulate





## Esempio: Encapsulate

### □ Esempi

- nella pila dei protocolli di rete, ogni protocollo è implementato sulla base di servizi offerti dallo strato inferiore
- la modificabilità è favorita dal fatto che questi servizi sono fruiti solo sulla base di un'interfaccia
- l'accesso a una base di dati avviene sulla base di uno schema logico – che incapsula lo schema fisico della base di dati
- questo consente, ad es., di modificare lo schema fisico della base di dati – senza dover modificare le applicazioni che la accedono



## Use an intermediary

### □ *Use an intermediary*

- un *intermediario* è un elemento introdotto per rompere la dipendenza (indesiderata) tra un elemento A e un elemento B
  - le responsabilità assegnate all'intermediario riguardano spesso proprio la gestione della dipendenza
- esistono tanti tipi di intermediari – che consentono di rompere dipendenze diverse
  - ad es., molti design pattern o pattern architetturali – come facade, proxy, adapter, bridge, mediator, factory, ... – oppure un broker o un servizio di directory
- intuizione: l'accoppiamento non si può mai eliminare del tutto – però spesso si può spostare dove dà meno fastidio



## Esempio: Use an intermediary

### □ Esempio

- l'accesso a una base di dati da parte di un'applicazione avviene mediante il suo schema esterno – che è un intermediario che fornisce una vista dei dati della base di dati che è specifica per l'applicazione
- molti cambiamenti possono essere gestiti ridefinendo lo schema esterno per un'applicazione – ma senza cambiare lo schema logico condiviso della base di dati
- le applicazioni possono essere protette da cambiamenti che richiedono la modifica dello schema logico adeguando i loro schemi esterni (che è meno costoso che non cambiare le applicazioni)

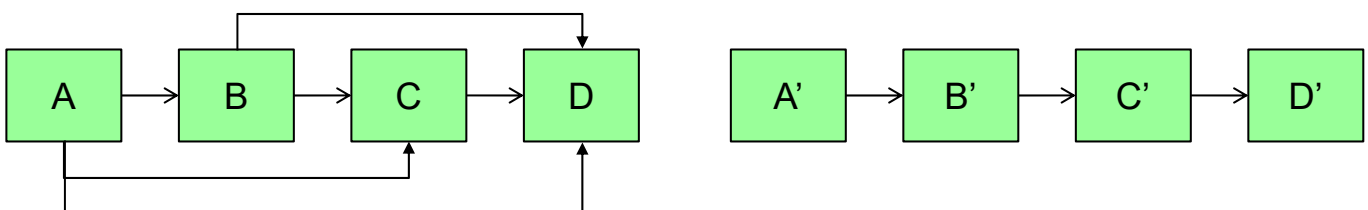


## Restrict dependencies

### □ *Restrict dependencies*

- questa tattica consiste nel rimuovere una dipendenza relativa a una necessità di comunicazione, riducendo l'insieme dei moduli con cui ciascun modulo può comunicare direttamente – questa comunicazione può essere poi incanalata e gestita da un intermediario

### □ Esempio





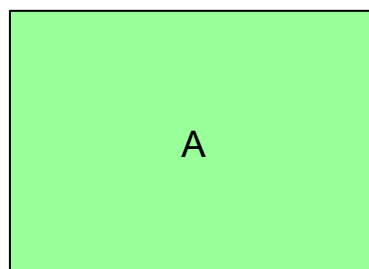
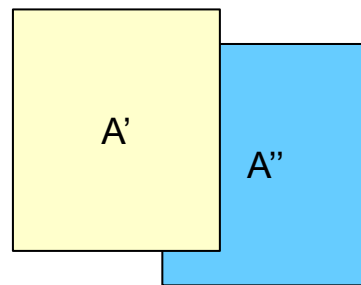
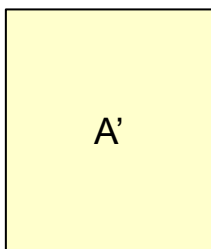
# Refactor

## □ Refactor

- ha l'obiettivo di ridurre duplicazioni nel codice
- il codice di interesse per più moduli viene messo a "fattor" comune in un solo modulo
  - in modo che i cambiamenti attesi riguardanti queste responsabilità possano essere gestiti in quel modulo, una sola volta
- più in generale, il refactoring può essere applicato per generalizzare responsabilità simili



# Refactor





# Abstract common services

## □ *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, se sono implementati in una forma generale), allora è sostenuta anche la modificabilità
- un modo comune nel rendere un servizio più astratto è basato su una parametrizzazione delle sue attività – spesso realizzata mediante l'uso di un “linguaggio”



# Esempio: Abstract common services

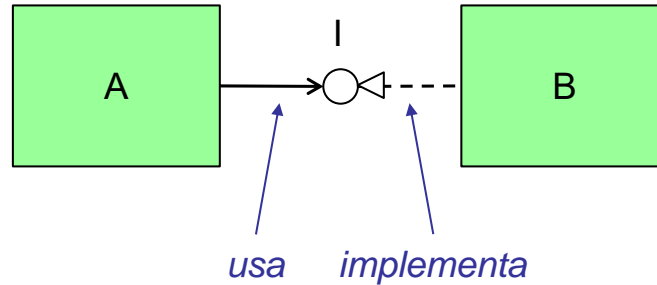
## □ Esempio

- 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 basato su un'interfaccia a messaggi è in genere inferiore all'accoppiamento basato su un'interfaccia procedurale



# - Interfacce e dipendenze

- Si considerino i moduli A e B e un'interfaccia I – con A che usa I (ne invoca le operazioni) e B che implementa I (ne implementa le operazioni)

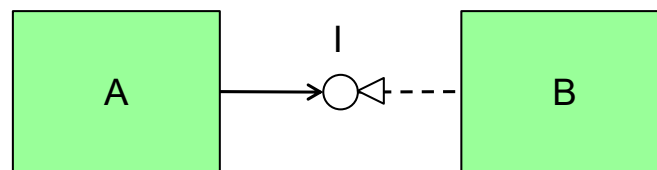


- in che verso è la dipendenza? da A verso B o da B verso A?
- la risposta da questa domanda dipende da quale dei due moduli è “proprietario” dell'interfaccia I

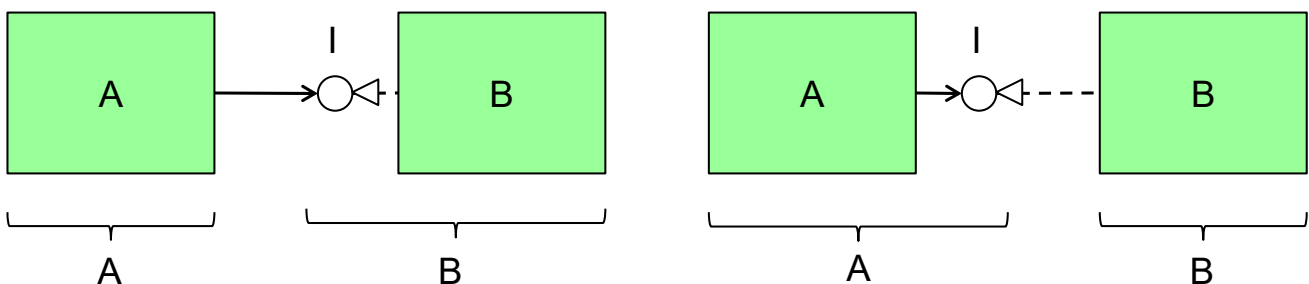


# Interfacce e dipendenze

- In che verso è la dipendenza? da A verso B o da B verso A?



- se B è il proprietario di I, allora la dipendenza è da A verso B – ma se invece A è il proprietario di I, allora la dipendenza è da B verso A







## Interfacce e dipendenze

- Ma che vuol dire essere “proprietario” di un’interfaccia I?
  - i moduli A e B verranno assegnati ai team di sviluppo  $T_A$  e  $T_B$
  - normalmente (però si veda dopo) ogni team dovrebbe poter gestire in modo autonomo i moduli di cui è responsabile – comprese le interfacce di pertinenza del modulo
    - ad es., se B è il proprietario di I, allora il team  $T_B$  dovrebbe essere il responsabile non solo di implementare il modulo B, ma anche di definire e far evolvere l’interfaccia I
  - più in generale, vista l’importanza delle interfacce nell’architettura del software, sono possibili diverse scelte riguardanti le interazioni tra i moduli – ma anche e soprattutto le relazioni tra i rispettivi team di sviluppo



## Interfacce e dipendenze tra team

- Alcune possibili relazioni tra team che riguardano un’interfaccia di interesse per una coppia di team
  - *Partnership* – i due team, pur se autonomi nelle implementazioni, sono interdipendenti negli obiettivi – quindi si incontreranno di frequente per definire insieme le interazioni tra i loro moduli
    - in questo caso, l’interfaccia I è in “comproprietà” tra i due moduli e i due team – entrambi i moduli dipendono dall’interfaccia I
  - *Customer-Supplier* – il team Supplier deve soddisfare i bisogni del team Customer – con il team Customer che esprime le proprie richieste ma, alla fine, è il team Supplier a decidere quali richieste soddisfare e quando
    - in questo caso, l’interfaccia I è di proprietà del team Supplier, ma il Customer può negoziare le caratteristiche di I



## Interfacce e dipendenze tra team

- Alcune possibili relazioni tra team che riguardano un'interfaccia di interesse per una coppia di team
  - **Conformist** – il team X che è il proprietario dell'interfaccia non ha nessun interesse a conoscere e soddisfare i bisogni dell'altro team – si pensi ad un team Y che utilizza una libreria di Google – il team Y che utilizza la libreria si deve conformare alle decisioni del team proprietario X
    - in questo caso, l'interfaccia I è di proprietà esclusiva del team X, e il team Y non può negoziare le caratteristiche di I

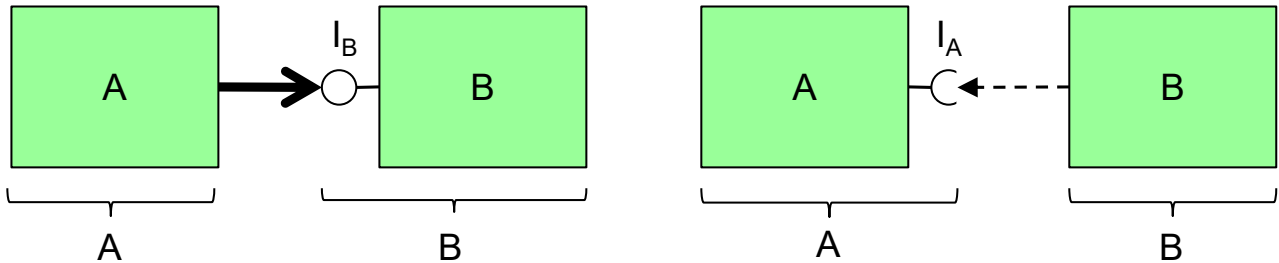


## - Dependency Inversion Principle

- Il seguente principio specializza la tattica dell'incapsulamento
  - **Dependency Inversion Principle (DIP)** [Martin]
    - i moduli di alto livello non dovrebbero dipendere dai moduli di basso livello – piuttosto, entrambi dovrebbero dipendere da opportune astrazioni
    - le astrazioni non dovrebbero dipendere dai dettagli – piuttosto, i dettagli dovrebbero dipendere dalle astrazioni
  - in pratica, il DIP può consentire di trasformare una dipendenza  $A \rightarrow B$  in una dipendenza inversa  $A \leftarrow B$ 
    - soprattutto quando A è di livello più alto di B



## Esempio: Dependency Inversion Principle



53

Modificabilità

Luca Cabibbo ASW



## Esempio: DIP

### □ Esempio

- il modulo A definisce la logica di business di un servizio, il modulo B si occupa di gestire l'accesso ai dati persistenti per quel servizio – chi deve dipendere da chi?
  - il modulo A è di livello più alto di B, dunque è B che deve dipendere da A
  - pertanto, A definirà un'interfaccia che specifica il modo in cui A vuole accedere ai dati persistenti del servizio – questa interfaccia sarà basata sulle effettive esigenze di A – e B implementerà questa interfaccia
  - l'alternativa (peggiore) è che B definisca una generica interfaccia per accedere ai dati persistenti del servizio, e che A dipenda da questa generica interfaccia – che però potrebbe non rispondere alle specifiche esigenze di A, rendendo più complicata l'implementazione di A

54

Modificabilità

Luca Cabibbo ASW



## - Defer binding

- ❑ Il costo di una modifica dipende anche dal momento (nel ciclo di vita dello sviluppo) in cui è possibile effettuare quella modifica
  - intuizione: finché c'è un'opportuna preparazione, più tardi nel ciclo di vita di un sistema si verifica una modifica, minore è il suo costo
- ❑ Le tattiche nella categoria *Defer binding* (rimanda/posticipa il collegamento/l'attivazione di una modifica)
  - consentono di controllare il tempo e il modo (e il costo) per effettuare una modifica e il suo rilascio – per alcune modifiche che devono essere note (più o meno bene) in anticipo



## Defer binding

- ❑ Tattiche per effettuare modifiche al tempo della codifica
  - parametrizzare i moduli
  - usare il polimorfismo
  - usare la programmazione orientata agli aspetti (AOP)
- ❑ Tattiche per effettuare modifiche al momento della compilazione
  - sostituzione di componenti
- ❑ Tattiche per effettuare modifiche al momento dell'installazione (deployment)
  - collegamento al momento della configurazione



## Defer binding

- Tattiche per effettuare modifiche al momento dell'avvio (initialization time)
  - file di risorse – ad es., un file di configurazione
  - collegamento al momento dello start-up – ad es., con parametri specificati al momento dell'avvio
  
- Tattiche per effettuare modifiche a runtime
  - registrazione a runtime – di parametri e servizi, in un registry
  - lookup dinamico – di parametri e servizi, da un registry
  - interpretazione di parametri – in un modulo sufficientemente generale e parametrizzato
  - uso di plug-in
  - metadati e riflessione

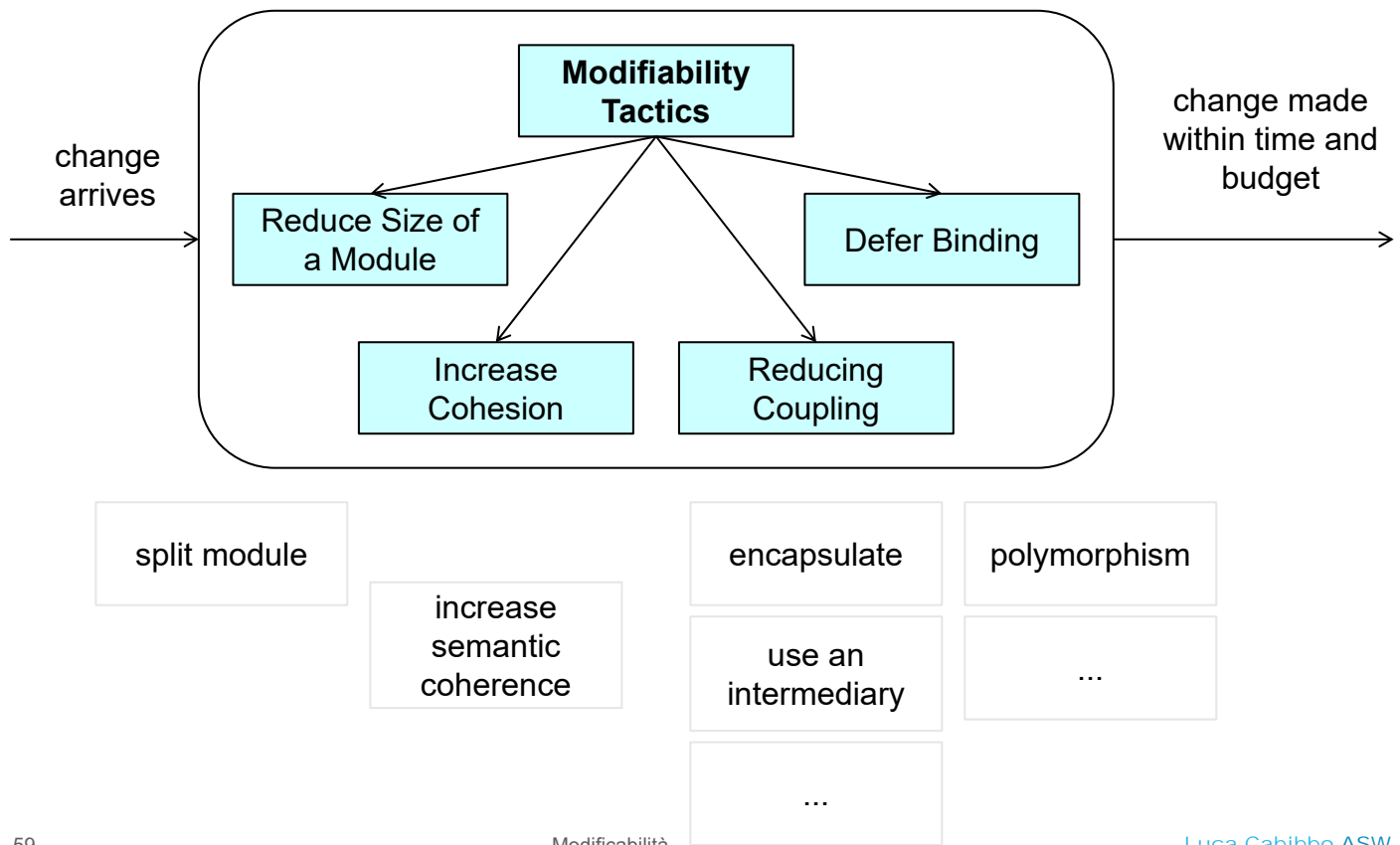


## Configuration management

- Un aspetto importante che emerge da quest'ultima categoria di tattiche è la gestione dei *dati di configurazione delle applicazioni*
  - ogni servizio o componente software richiede in genere dei dati di configurazione – ad es., le credenziali di accesso ai servizi infrastrutturali
    - i dati di configurazione devono essere spesso organizzati in diversi profili di esecuzione
    - alcuni dati di configurazione sono sensibili
  - queste configurazioni vanno gestite esternamente ai servizi (*externalized configuration*) – in modo che possano variare indipendentemente dai servizi



# Tattiche per la modificabilità



59

Modificabilità

Luca Cabibbo ASW



## - Discussione

- Le tattiche per la modificabilità che sono state presentate
  - esemplificano l'idea di "tattica come trasformazione"
    - l'applicazione di una tattica può cambiare l'architettura – con l'obiettivo di raggiungere un miglior controllo di un attributo di qualità
  - sono applicate in molti pattern architetturali – ad es., Layers, Pipes and Filters, Microkernel, Reflection, MVC, Broker, ...

60

Modificabilità

Luca Cabibbo ASW



## - Altre opzioni di progettazione per la modificabilità



- La prospettiva dell'*evoluzione* di [SSA] propone alcuni suggerimenti, tattiche e opzioni di progettazione per la modificabilità
  - contieni (localizza) il cambiamento – *increase semantic coherence, split module, encapsulate, use an intermediary, ...*
  - crea interfacce estensibili – *abstract common services*
  - applica tecniche di progettazione che facilitano il cambiamento – *use an intermediary* – ad es., collega elementi diversi mediante degli opportuni design pattern
  - applica pattern architetturali basati su meta-modelli – ad es., Reflection
  - costruisci punti di variazione nel software e punti di estensioni standard
  - cambia in modo affidabile – riguarda la verifica e il rilascio



## \* Discussione

- La modificabilità riguarda la flessibilità con cui è possibile gestire cambiamenti in un sistema software dopo che questo è stato rilasciato
  - si tratta di una qualità importante in molti sistemi – poiché un sistema di successo deve essere in grado di soddisfare gli obiettivi di business di un'organizzazione – non solo quelli correnti, ma anche quelli futuri
  - un sistema software flessibile può infatti abilitare (anziché ostacolare) l'organizzazione che lo utilizza nell'evoluzione del proprio business