OO testing
(maggio '16)

Enrico Vicario
Laboratorio Scienza e Tecnologie del Software
Dipartimento Sistemi e Informatica
Università di Firenze

enrico.vicario@unifi.it
www.dsi.unifi.it/~vicario

Il caso speciale di OO Testing

- Teoria ed esperienza meno consolidata rispetto a Structured Programming (SP)
  - OOP più recente
  - Meno rilevante nei contesti safety critical dove il testing è prescritto
    - Transizione da c a c++ in atto (verso quale esito?)
- Il ruolo speciale del testing in eXtreme Programming
  - Principio del Test First
  - Strumenti di supporto Junit, CppUnit
  - Documentazione in sostituzione di diagrammi e testo
  - Supporto al regression testing,
    abilitare pratiche di refactoring, proprietà condivisa, integrazione frequente, …
  - Minore enfasi e risultati consolidati sulla metodologia di test selection
- OOP aggiunge a SP vari fattori di complessità
  - 1/3) Complessità del flusso di controllo
  - 2/3) Stato, visibilità e concorrenza sugli oggetti
  - 3/3) Polimorfismo e separazione fra le viste di classi e oggetti
  - (language hazards: a kind of anti-idioms)

In questa complessità cosa dobbiamo testare?

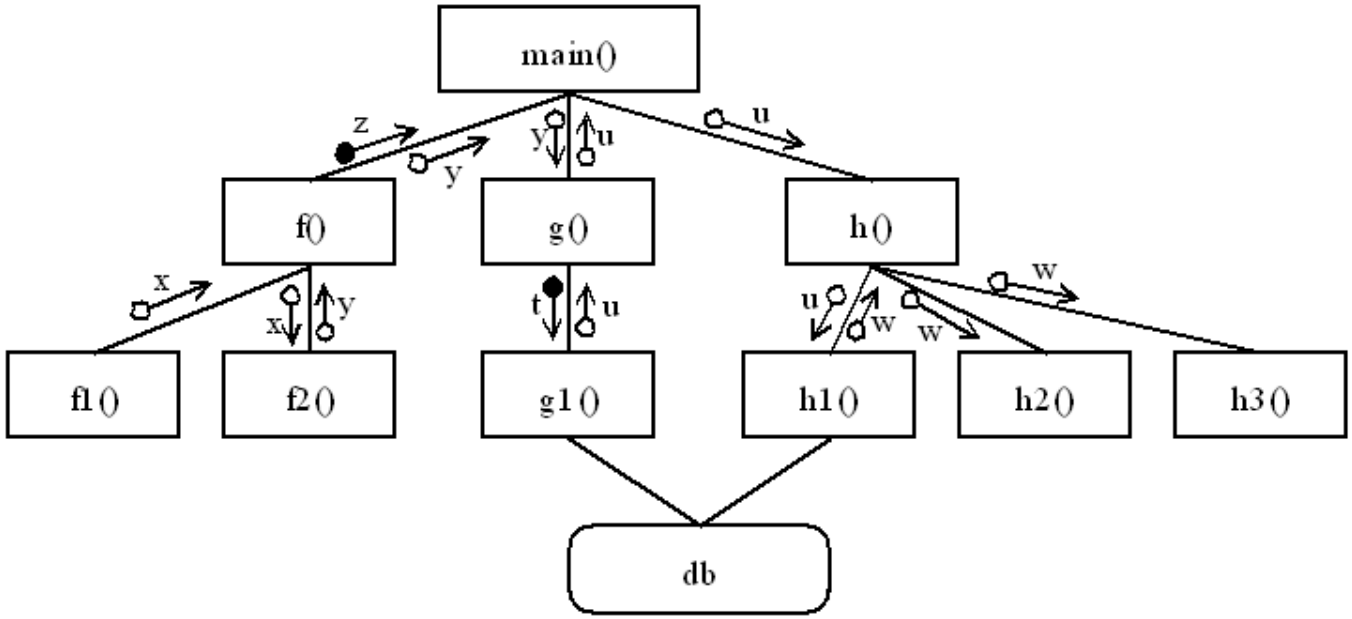
- Applicare principi generali della metodologia del testing adattando metodi pensati per Structured Programming
  - Alcune ricette: Call graph, Class dependency graph

- Diversi livelli di unità e integrazione e diversa combinazione di approccio funzionale e strutturale
  - Unit testing sui singoli metodi e Unit testing su una singola classe
    - Svolto dal programmatore e basato su criteri strutturali
  - Integration testing su microarchitettura di classi
    - E' cosa diversa lavorare sul class diagram che specifica il progetto oppure sul class diagram estratto dal codice
    - Diverso impatto sulla pratica di sviluppo: disponibilità di documentazione, testing manuale o computer aided
  - System testing
    - In prospettiva funzionale
    - Basato su use case diags
    - Magari accompagnato da coverage analysis

1/3) Complessità del flusso di controllo

Flusso di controllo in Structured Programming

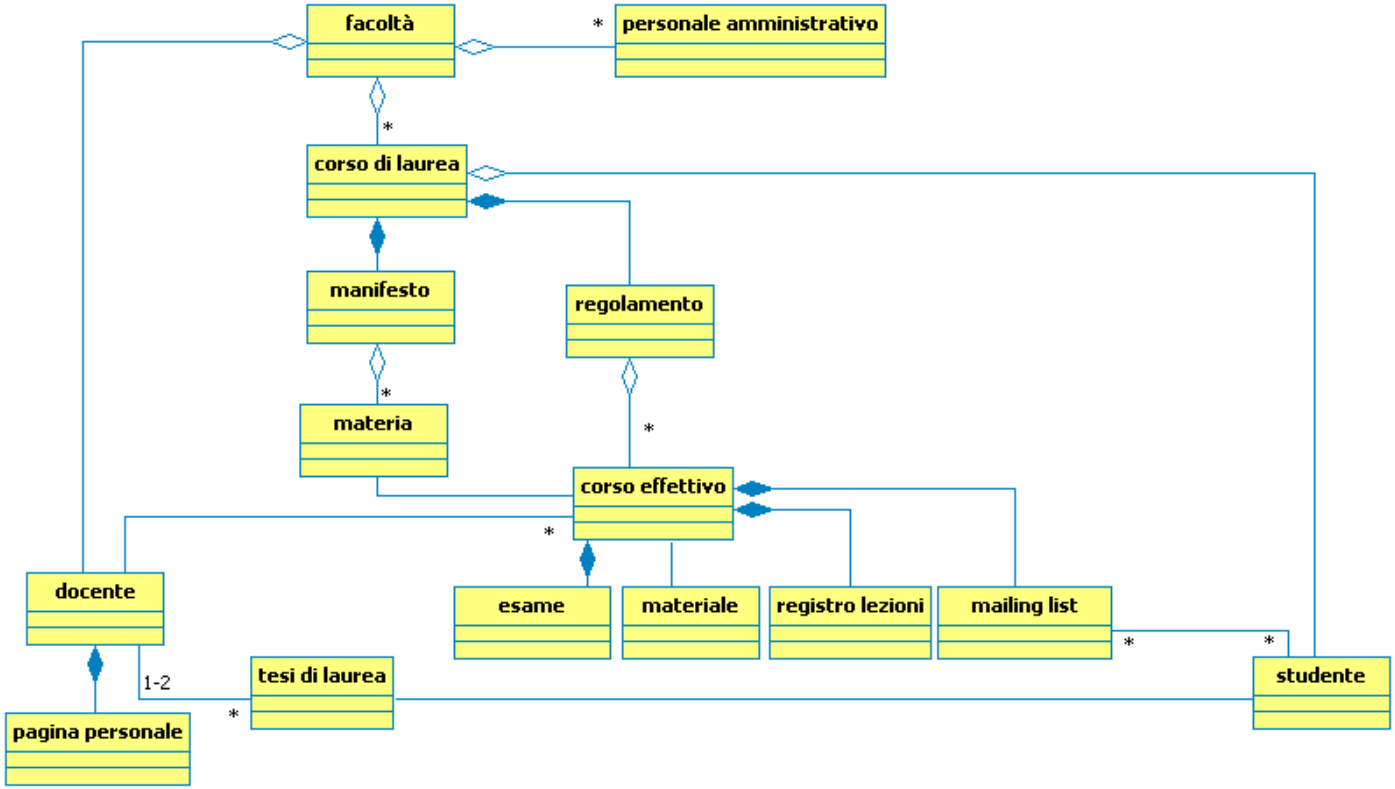
- In SP il codice è organizzato in una gerarchia di funzioni
- La gerarchia realizza una funzione (“trasformazione”),
o un insieme di funzioni smistate da uno o più “centri di transazione”
  - Spesso si realizza una confluenza sui livelli bassi
(riuso sul livello di libreria)
  - La carta strutturata fornisce una chiara visione di come gira il controllo
- Esiste un modello di riferimento su come fare girare il controllo



Flusso di controllo in Object Oriented Programming

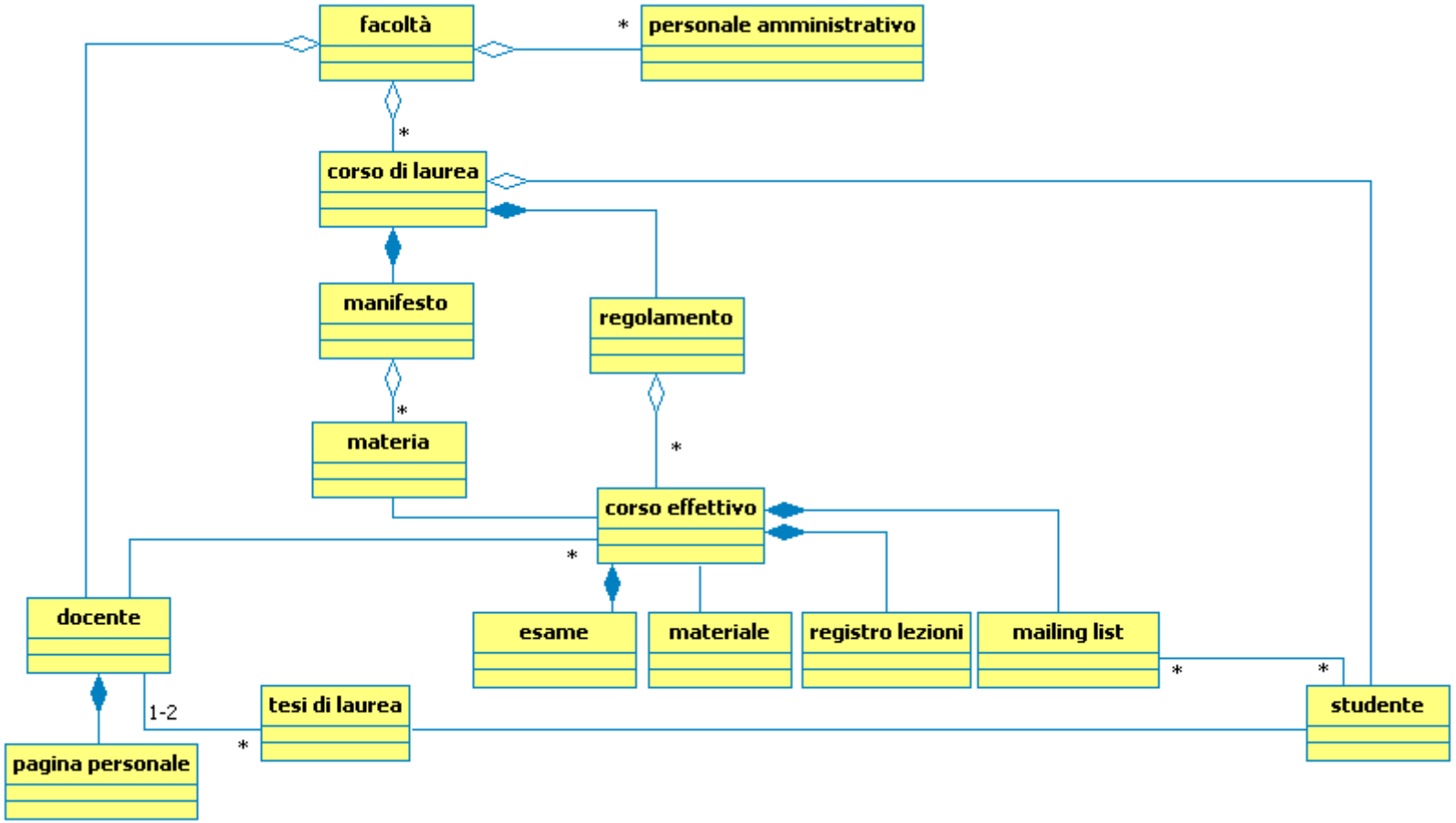
In OOP la topologia su cui avviene la computazione è un grafo di oggetti, … e quello che vede il programmatore è un grafo di classi

- Il grafo svolgere più computazioni corrispondenti a più casi d'uso
- Ciascuno corrispondente a un diverso sequence diagram
  - Eg: aggiungiEsame, trovaDocente, elencaMaterie, …
  - Non strettamente specificato dalla vista statica del class diagram
  - (Iconix, Analisi di robustezza)



Flusso di controllo in OOP

- Il controllo non gira secondo un modello predefinito
  - Ciascun oggetto trasferisce il controllo all'oggetto che detiene la responsabilità delegata e non a un subordinato
  - Tra i due spesso esiste una relazione di uso dinamica e non una relazione strutturale



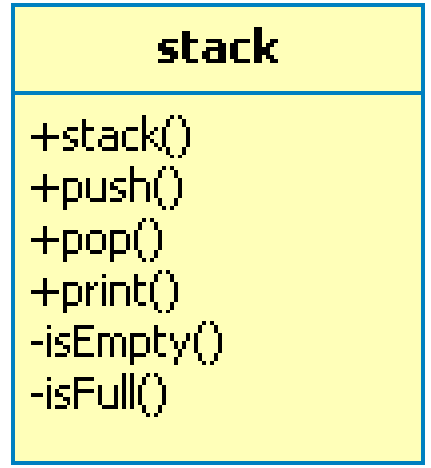
Da SWE: Flusso di controllo in Object Oriented Programming

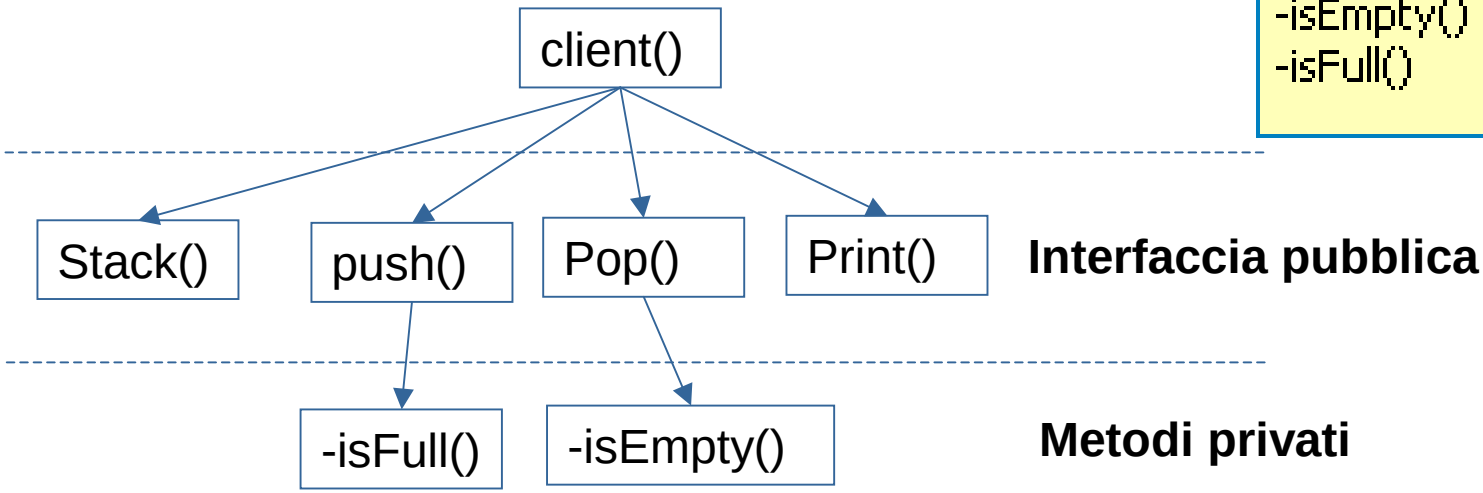
- E’ dimostrato che OOP favorisce l’evolvibilità, il riuso, la produttività
  - Una architettura è capace di assorbire una varietà di responsabilità coesive sui dati che tratta, e non una singola funzione
  - E’ sempre possibile aggiungere un responsabilità ad una classe, o aggiungere una classe
  - E’ sempre possibile attraversare la rete degli oggetti in qualche modo diverso per rispondere a un nuovo caso di uso
  - Meccanismi di interazione complessi: call-back (in ultimo è una ricorsione), forwarding, inversione di responsabilità, …
- Molto più flessibile rispetto a SP
  - dove l’intera gerarchia è finalizzata a svolgere una funzione con responsabilità coesive in senso funzionale
- … ma tutto questo evidentemente complica il testing
  - tanto più con il prevalere di programmazione basata sulla composizione prima che sull'ereditarietà

Una ricetta a partire dal control flow testing

Call graph

- I nodi sono metodi e gli archi sono dipendenze di uso
- Distingue metodi pubblici, privati e protetti
  - Sui metodi pubblici può avvenire un uso concorrente da parte di più clients esterni
- Un esempio di intra-class testing:
  stack su lista sequenziale



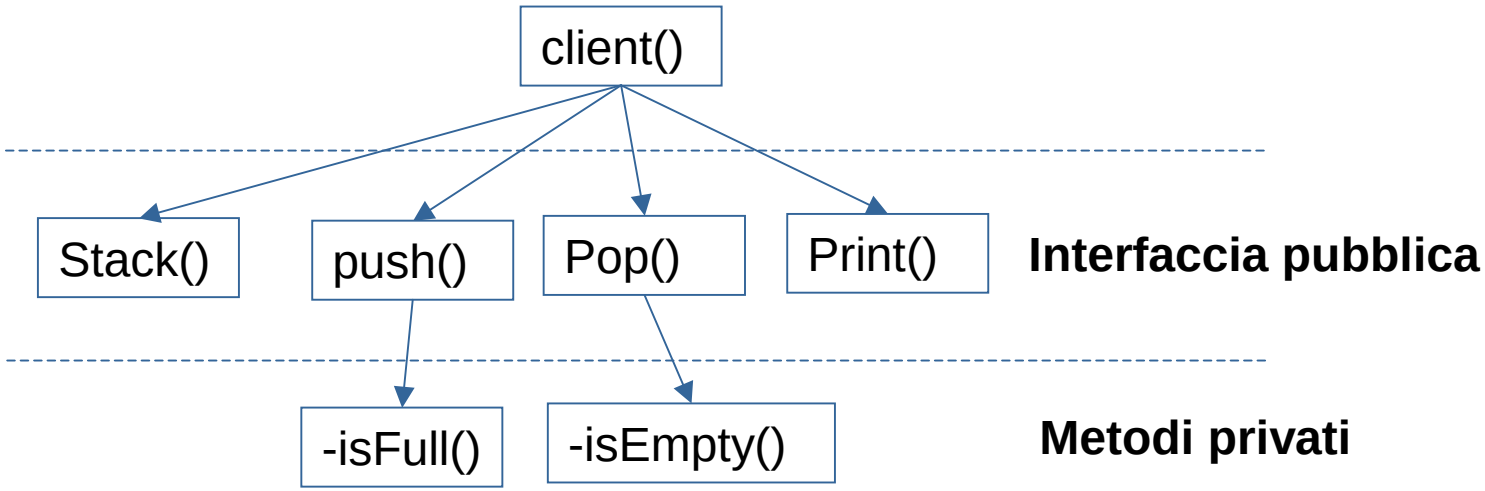


- Nella terminilogia del testing classico sarebbe "inter-procedural"
  ma qui prevale "intra-class"
  - Variabili condivise, minore complessità di metodi e signatures, …

Copertura sul call graph

Criteri di copertura

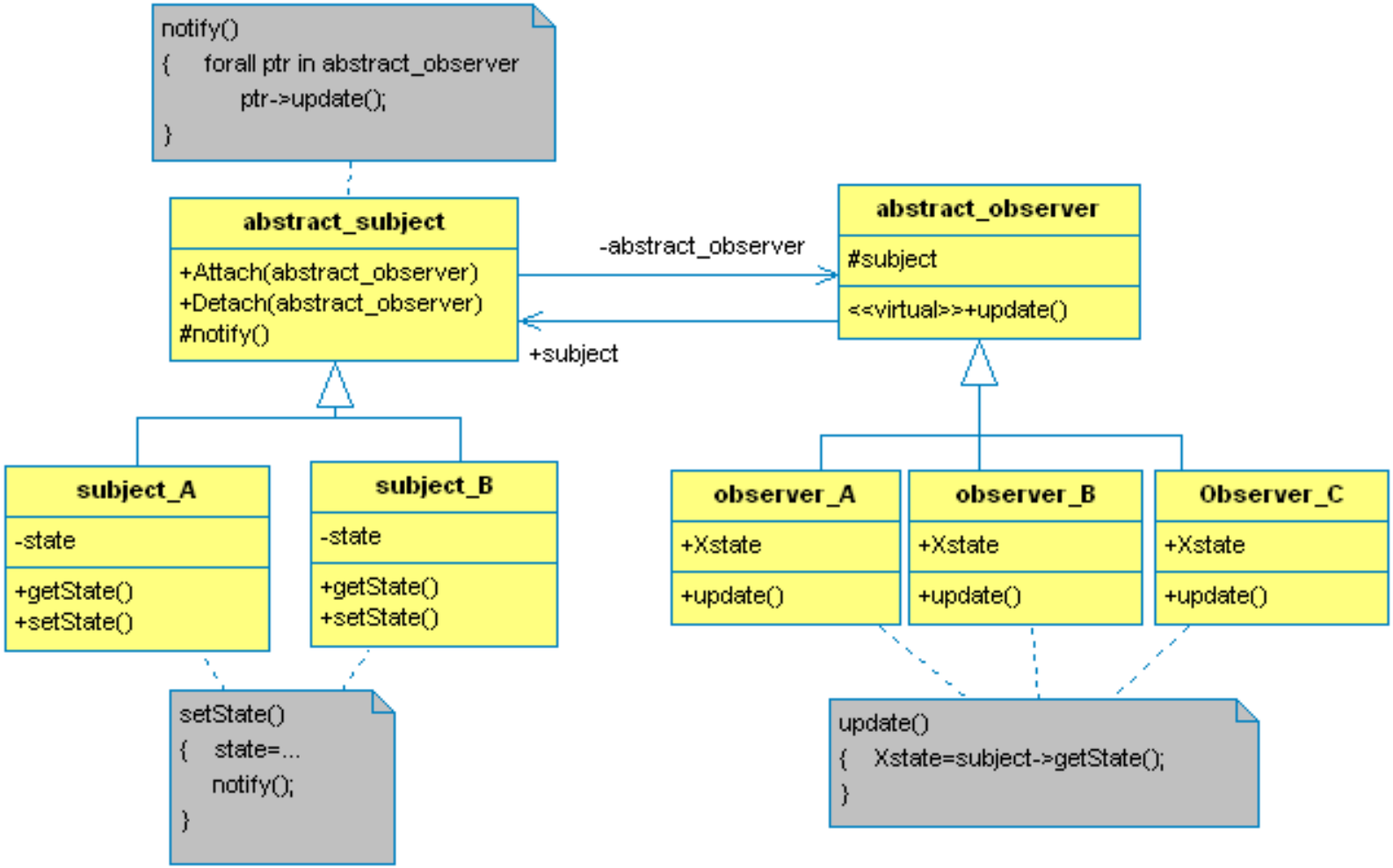
- All nodes: invocare almeno una volta ogni metodo (function coverage)
  - Solo i metodi pubblici, o i protetti, nel package, o anche i privati
  - stack()->push()->isFull()->pop()->isEmpty()->print()
- All Edges: un metodo può invocare diversi metodi (branch)
  oppure uno stesso metodo può essere invocato da più metodi (confluence)



- Così facendo non si esercita il possibile impatto di
  diversi ordini di interleaving nelle attivazioni

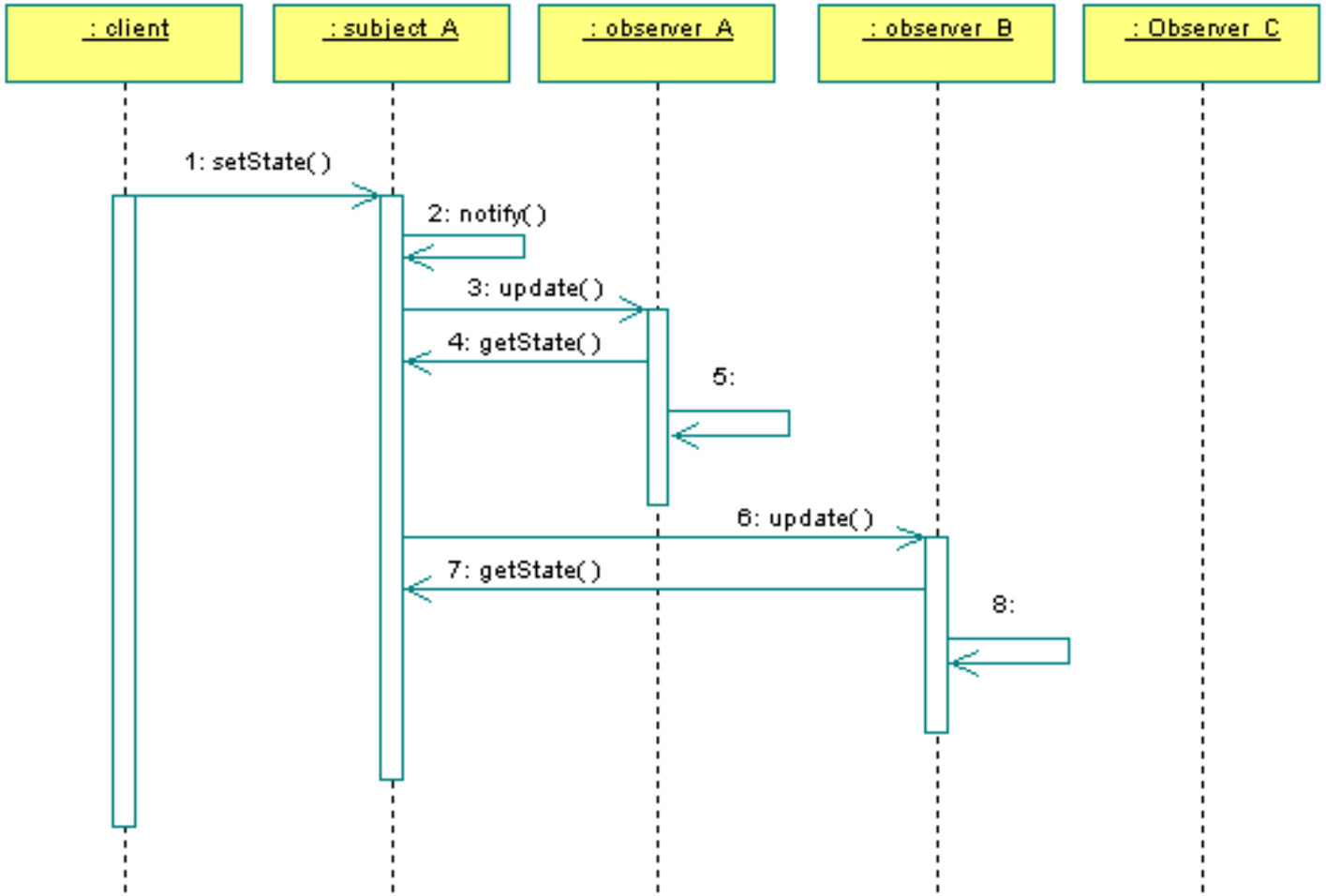
Da SWE, esercizio: come testare il pattern Observer ?

- definisce una dipendenza uno-a-molti per cui quando un oggetto modifica il suo stato tutti i suoi dipendenti ne ricevono notifica



Da SWE: Observer

- Installazione dinamica con responsabilità a carico dell'observer
  - Observer si registra su subject con attach/detach
  - Subject notifica agli osservatori registrati con notify
  - In ricezione del notify, observer invoca get_State



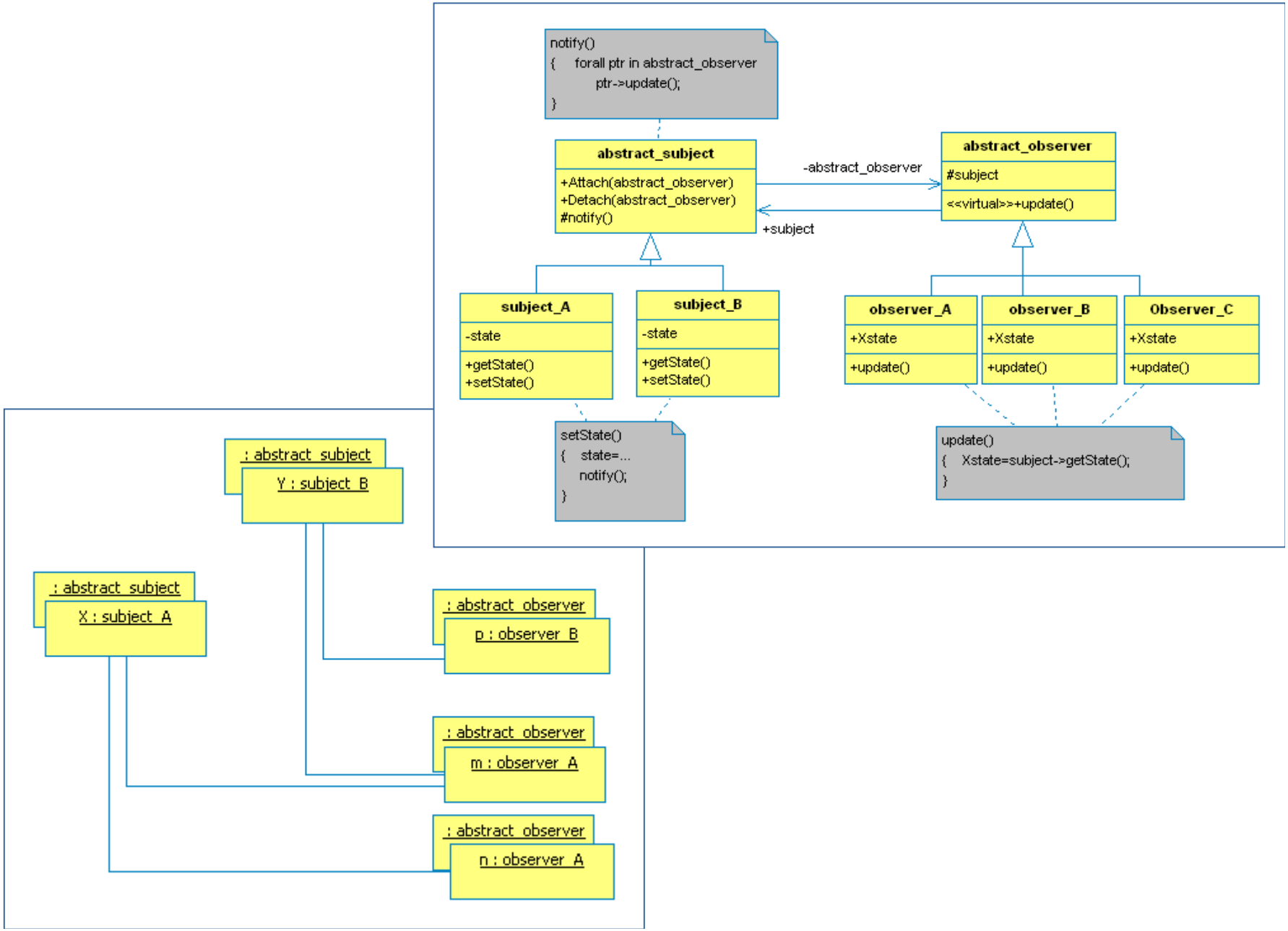
- Publish & subscribe

2/3) Stato, visibilità e concorrenza sugli oggetti

Stato, visibilità e concorrenza sugli oggetti

- In SP le variabili locali terminano il tempo di vita con la funzione
  - Limitato uso della direttiva static
  - … e delle variabili globali (che purtroppo sono spesso usate col c)
- In OOP un oggetto incapsula attributi che mantengono il valore anche quando il controllo non risiede in uno dei metodi dell'oggetto (stato)
  - Gli attributi hanno visibilità globale entro la classe
- Lo stesso oggetto può essere usato da oggetti diversi e anche di tipo diverso (concorrenza)
  - Esempio: uno Stack usato da più clients
  - Esempio: il Subject astratto in uno schema Observer
  - Gli oggetti hanno sostanzialmente visibilità globale
    - È sufficiente conoscerne l'indirizzo
    - In sostanza si realizza un uso pervasivo di quello che in c sarebbe un puntatore a funzione

Esempio di concorrenza: il caso dell'observer



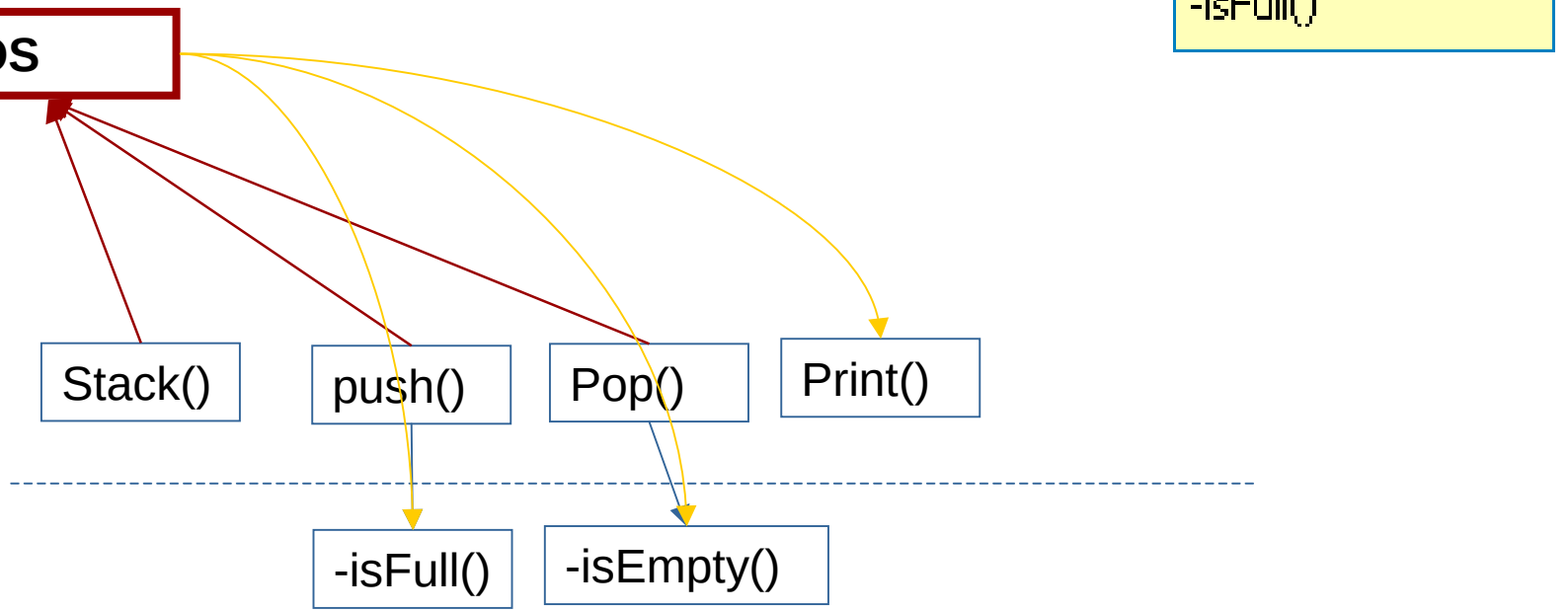
Una ricetta dal dataflow testing centrato su attributi

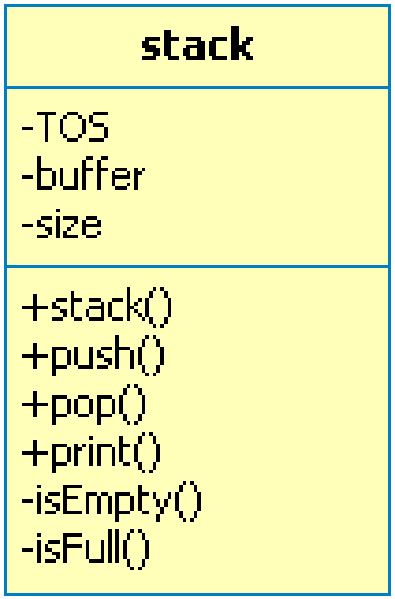
- Approccio: esercitare def/use paths rispetto agli attributi di un oggetto
  - Call graph annotato con relazioni def-use sugli attributi che formano lo stato dell'oggetto
  - Magari limitato ad attributi più rilevanti rispetto all'interazione
- applicabile nello unit testing di una classe
  - Risponde al problema di condivisione degli attributi
  - Aumenta l'accoppiamento tra i metodi rispetto a SP, ma l'accoppiamento è localizzato
- applicabile nello integration testing di microarchitetture
  - Risponde al problema del mantenimento dello stato e della concorrenza nell'uso di uno stesso oggetto

Unit testing di una classe : Il caso di uno stack

- Attributi condivisi: buffer, TOS, size

- Analisi dataflow sul TOS
  - Def: stack(), push(), pop()
  - Uses: print(), -isEmpty(), -isFull()
  - Oss: non distinguiamo p-uses e c-uses
  - OSS: il def in push() avviene dopo l'uso in isFull()

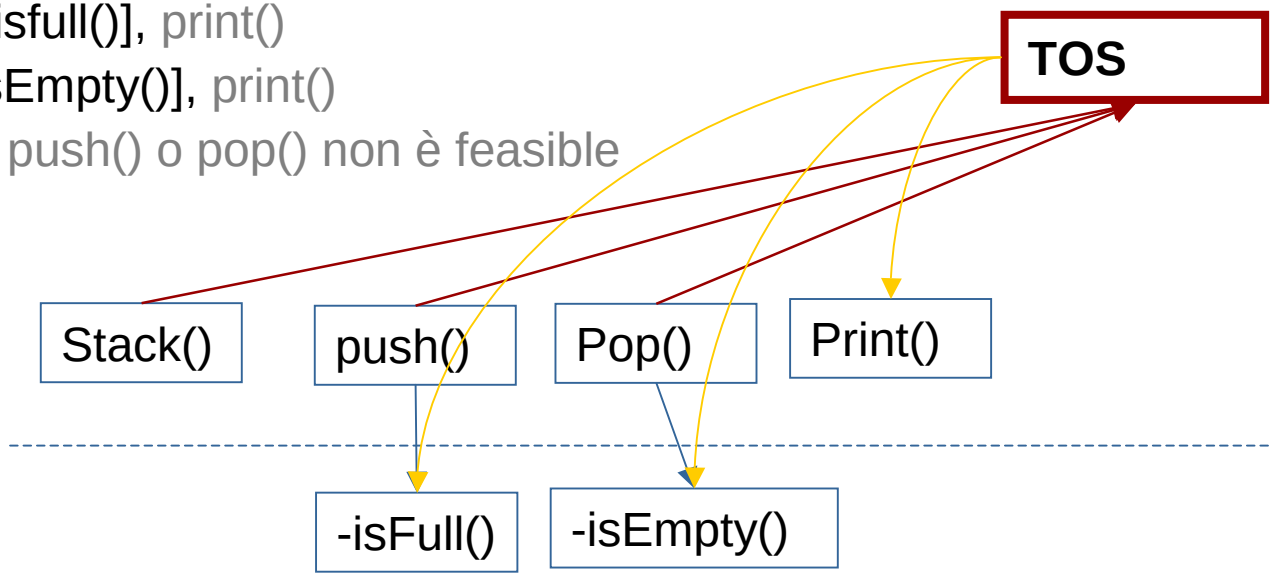




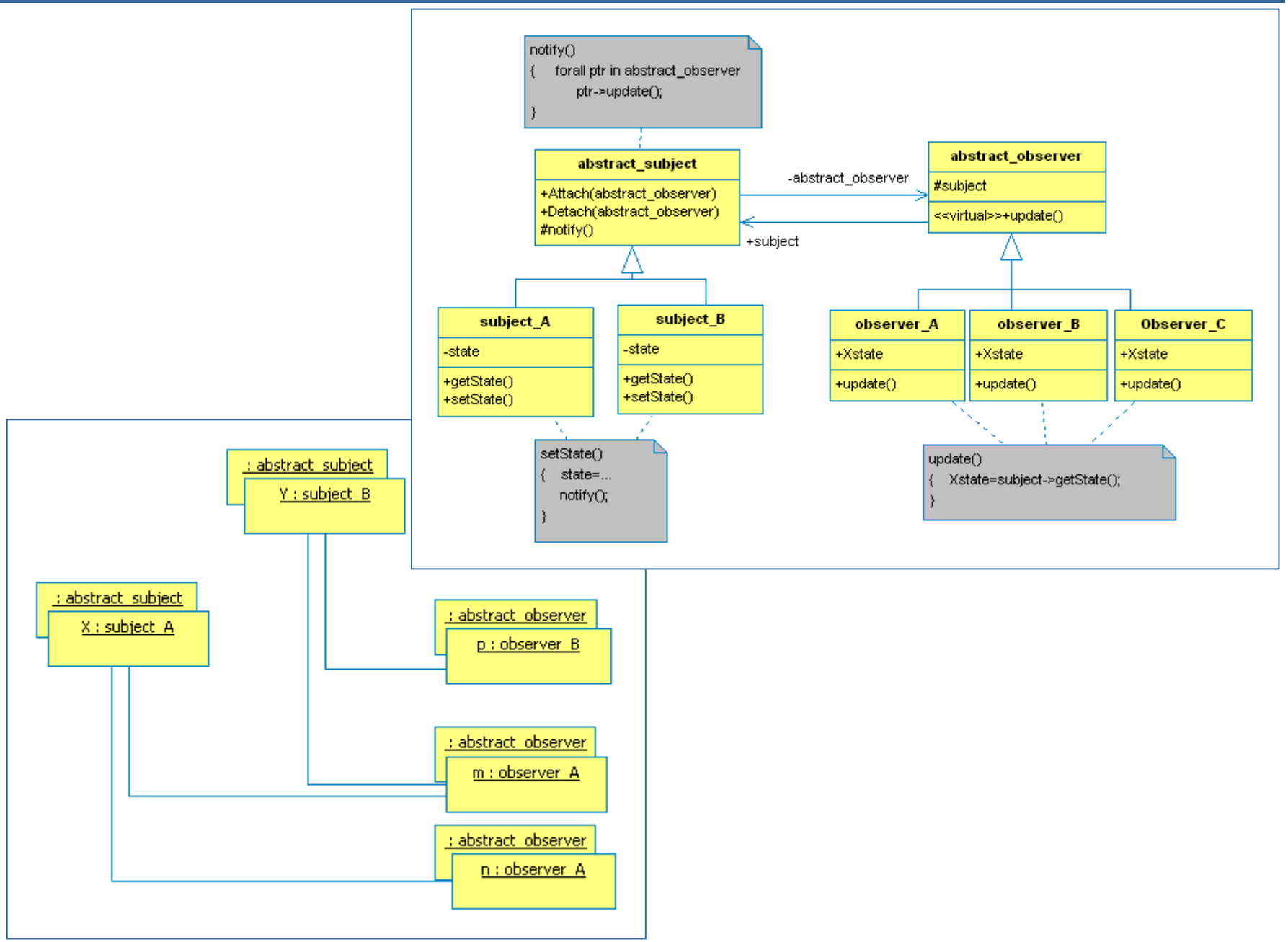
Il caso di uno stack

Copertura all-uses rispetto al TOS

- Stack(),print()
- Stack(), push() [isFull()], print()
  - Prolungo fino a print per osservare
  - L'uso in isFull() precede la def in push()
- Stack(),pop() [isEmpty()], print()
  - L'uso in isEmpty () precede la def in pop()
- …Push(),push() [isfull()], print()
  - La def del secondo push() è successiva a isFull()
- …Push(),pop() [isEmpty()], print()
- …Pop(),push() [isfull()], print()
- …Pop(),pop() [isEmpty()], print()
  - stack() dopo push() o pop() non è feasible

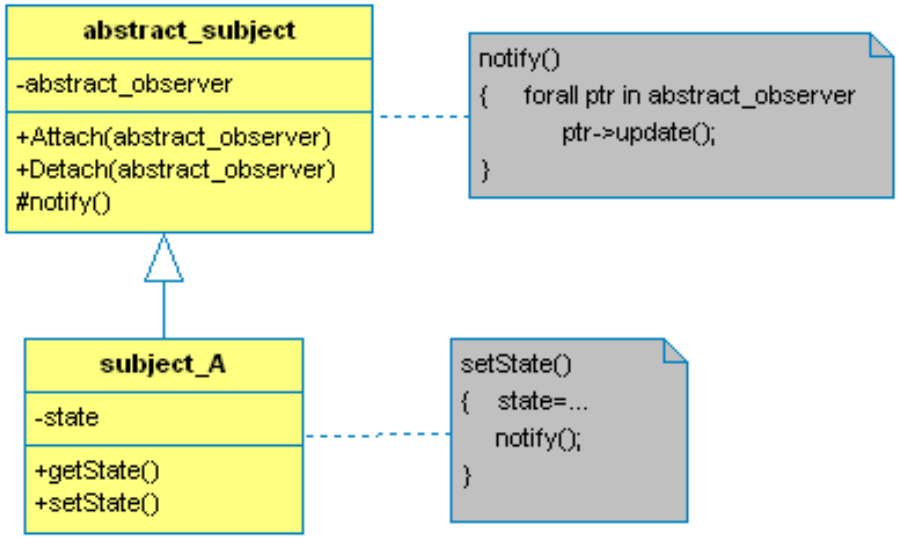


Integration testing su μarchitettura: il caso di un observer

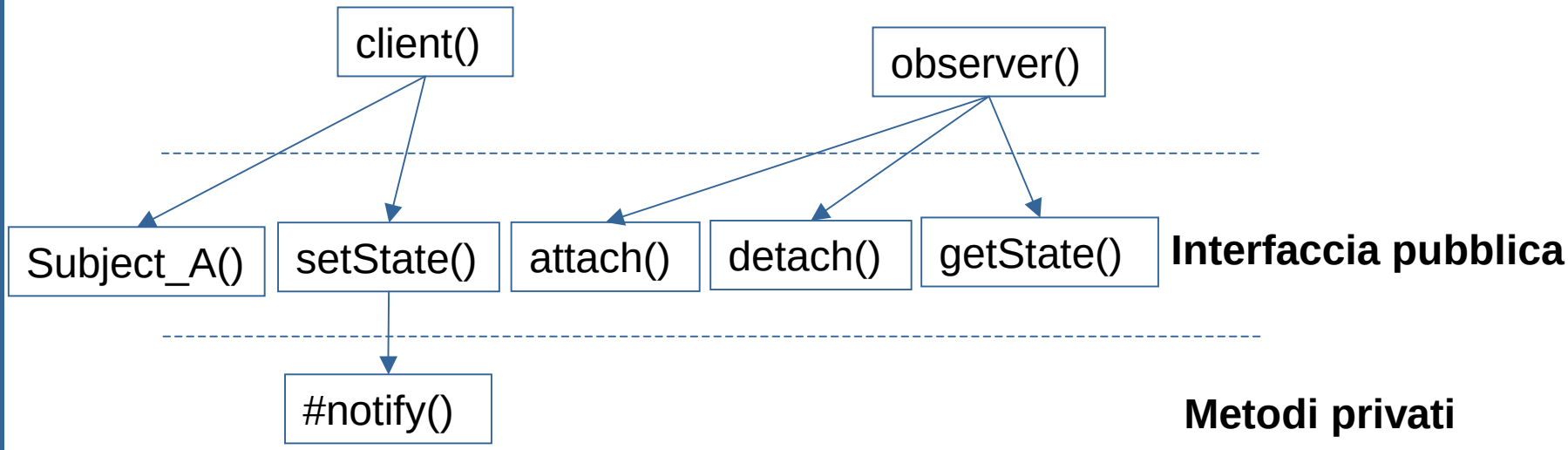


Il caso di un observer

- Test del subject



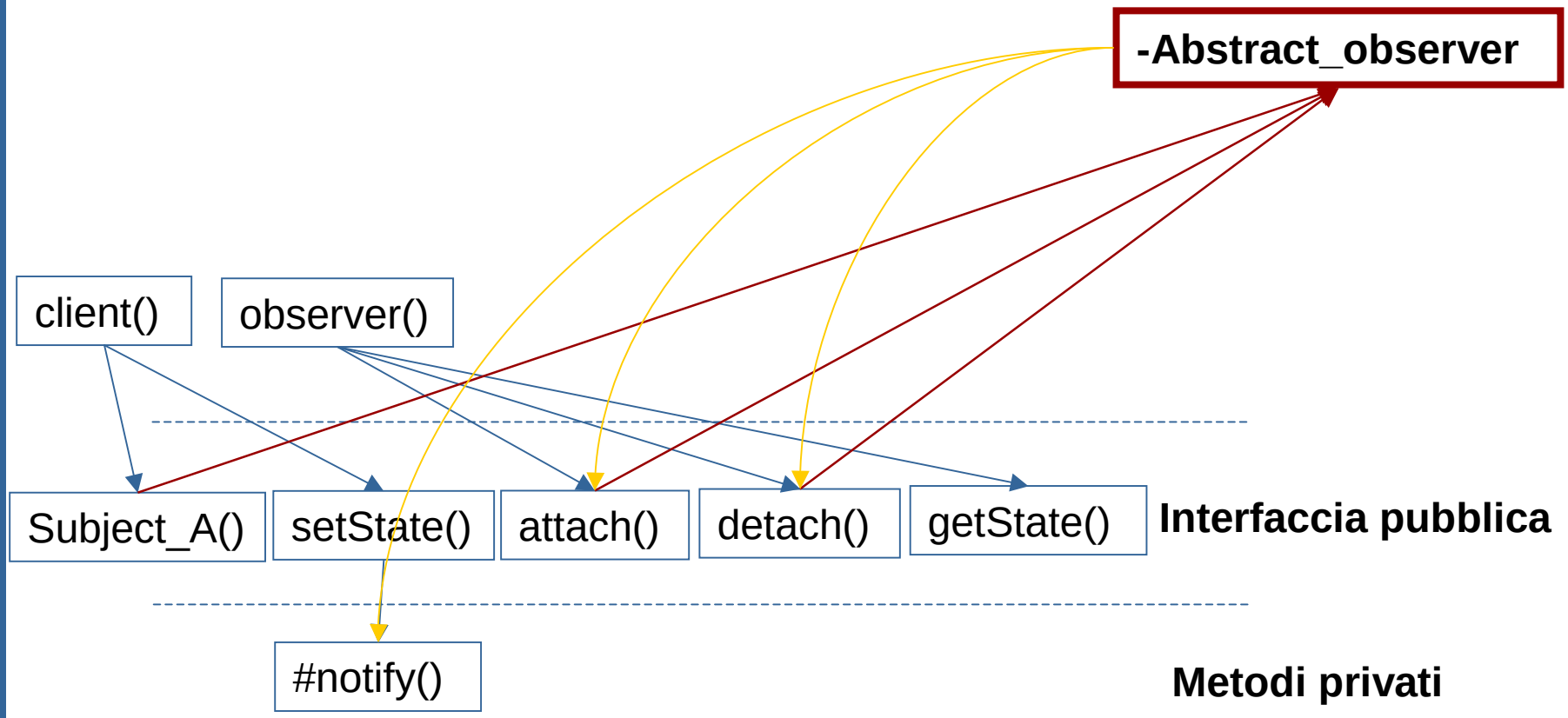
- Callgraph
  - Notify() è privato ma può essere esercitato attraverso setState()



Il caso di un observer

Annotazione dataflow del call graph

- Slicing sulle variabili che catturano gli elementi di compelssità
- Una variabile critica è la lista dei riferimenti -abstract_observer
- Anche -state sarebbe critico



Esempio: il caso di un observer

- Copertura all-def sulla lista degli osservatori (non dice molto)
  - Subject() setState() [Notify()]
  - Subject() Attach() setState() [Notify()]
    - SetState() e notify() rendono osservabile l'esito del test
  - Subject() detach() setState() [Notify()]
- Copertura all-uses (funziona)
  - Subject() setState() [Notify()]
  - Subject() Attach() setState() [Notify()]
  - Subject() detach() setState() [Notify()]
  - … Attach() attach() setState() [Notify()]
  - … Attach() detach() setState() [Notify()]
  - … detach() attach() setState() [Notify()]
- Miglioramento
  - La sequenza attach()attach() e attach()detach() dovrebbe essere distinta a seconda di come è risolta la condizione interna al secondo metodo (i.e. se esso agisce sullo stesso oggetto su cui ha agito il primo)
  - Per rendere la cosa osservabile sul call graph dovrei avere due funzioni sotto la guardia dell'if

3/3) il problema della topologia degli oggetti

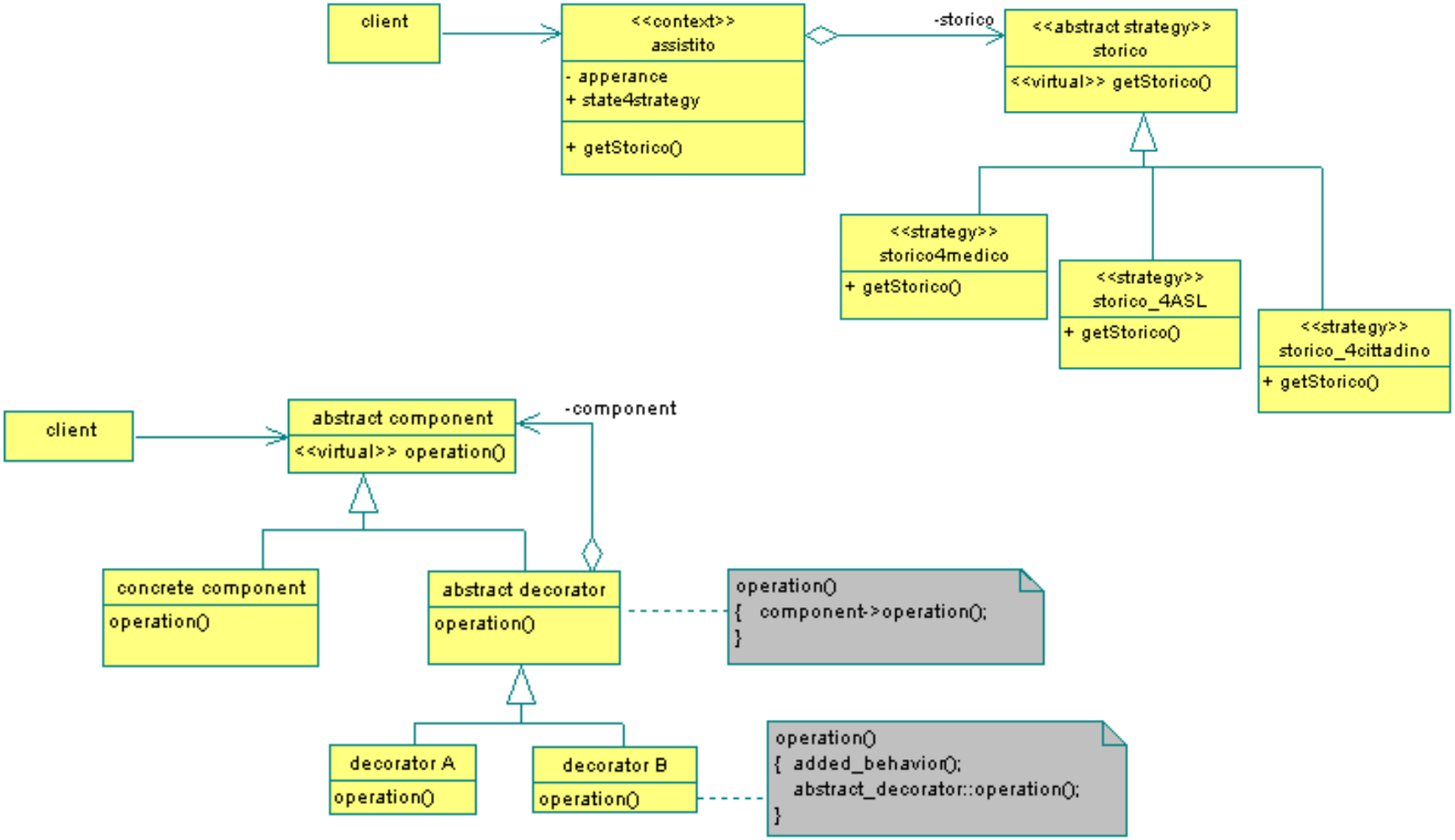
OOP: composizione dinamica della rete degli oggetti

- In SP la gerarchia è sostanzialmente statica
  - A meno di usare puntatori a funzione,
    che spesso non è una buona idea dove si vuole usare il c
- In OOP la topologia della rete degli oggetti è largamente dipendente da scelte prese al tempo di esecuzione
  - Oggetti istanziati in modo dinamico
  - In tipi diversi
  - Configurati e composti in modo diverso
  - La programmazione nello stile dei “design patterns” crea astrazione proprio sul modo con cui viene poi configurata la rete degli oggetti
  - Il prevalere dell’uso della delega rispetto all’ereditarietà esacerba il problema

Polimorfismo

La stessa operazione può avere implementazioni polimorfiche

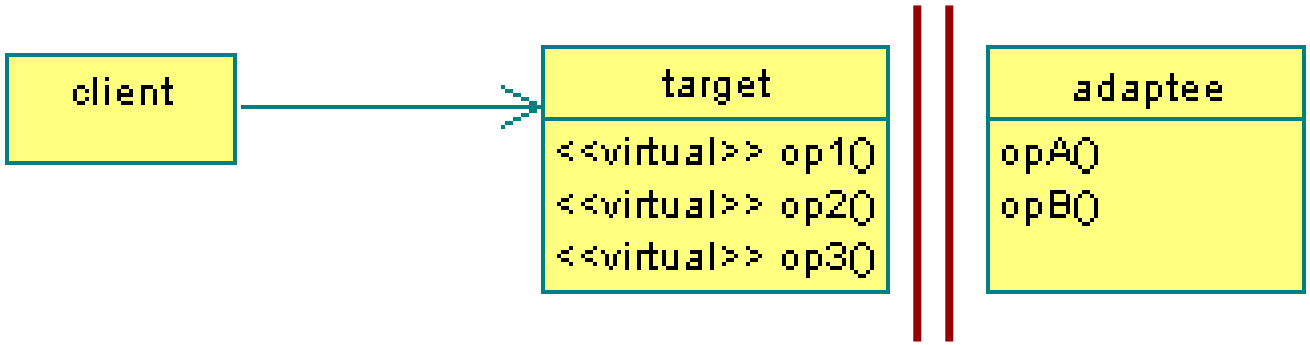
- Diverse forme nell'istanza concreta della classe (e.g. strategy)
  - Riduzione del costrutto IF
- Diverse configurazioni delle deleghe (e.g. decorator)



Da swe: Adapter - (structural)

- Converte l'interfaccia di una classe già implementata per adattarla alla interfaccia attesa da un cliente

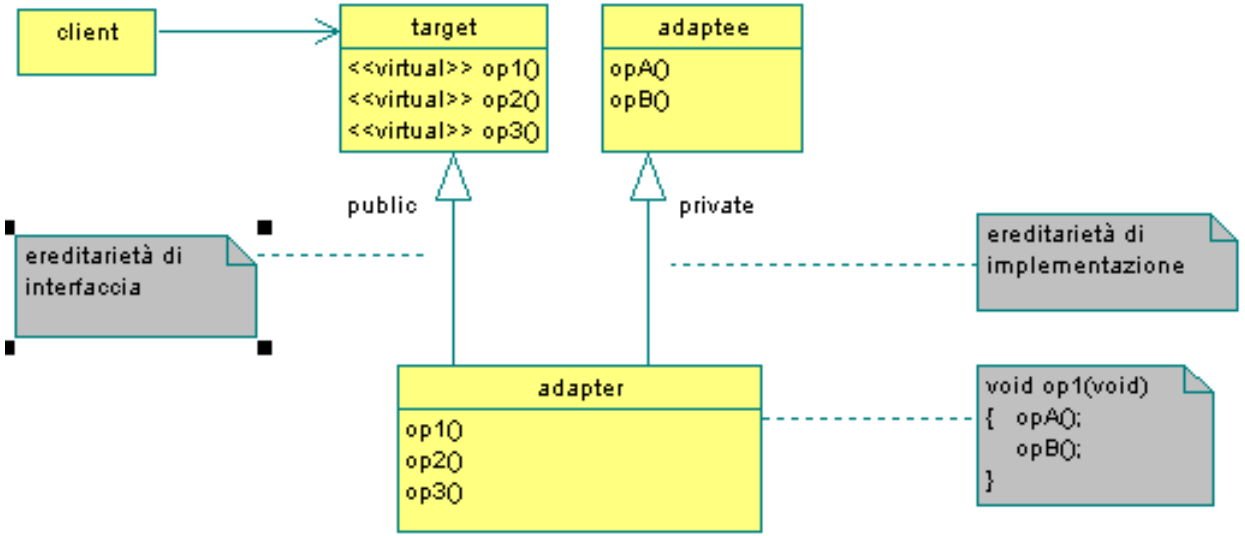
- Motivazione:
  - Una applicazione usa una interfaccia target che non è implementata
  - E' disponibile una interfaccia Adaptee che realizza in sostanza le stesse responsabilità ma in modo non conforme
    - Mismatch sui nomi, sui parametri, sulla granularità delle operazioni



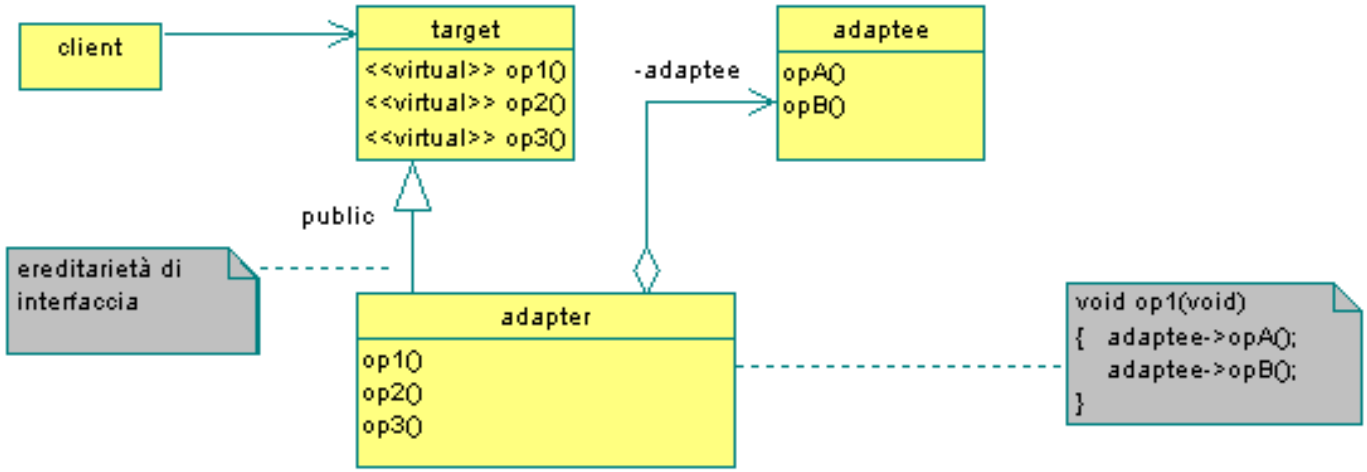
- Esempio
  - Adattamento verso una interfaccia già disponibile realizzata da una precedente applicazione

Da SWE: Adapter

- Class adapter
  - ereditarietà

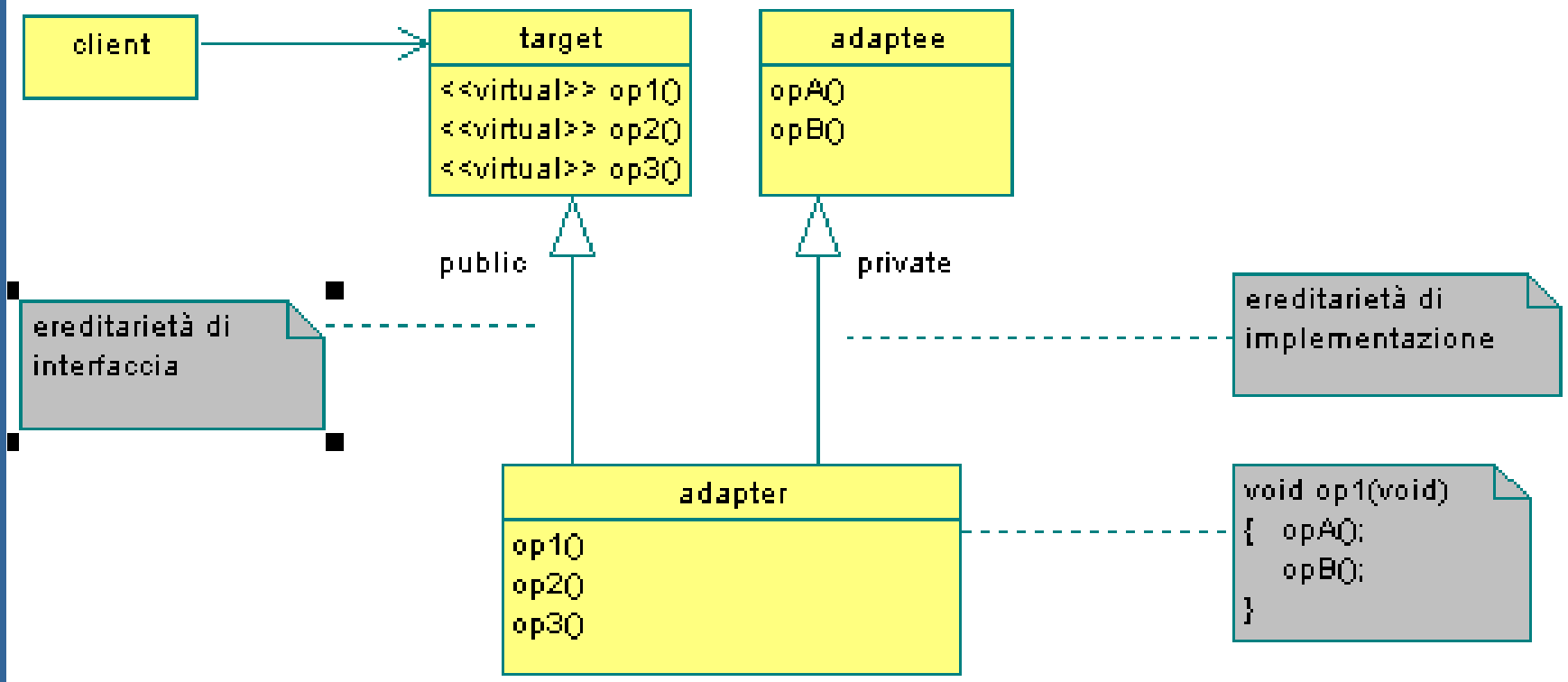


- Object adapter
  - composizione



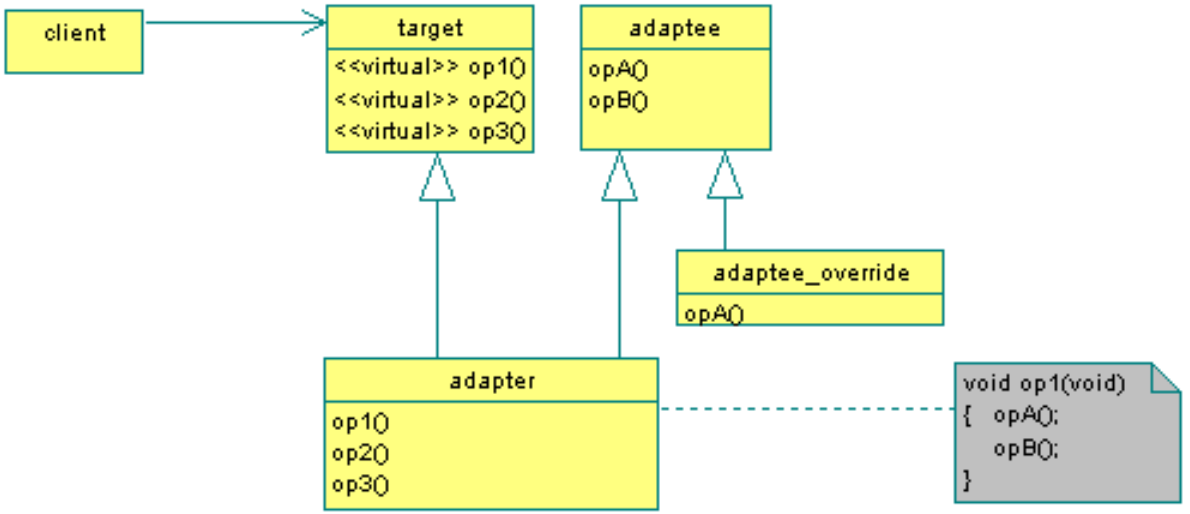
Da SWE: Class adapter

- Eredita l'interfaccia del target e l'implementazione dell'adaptee

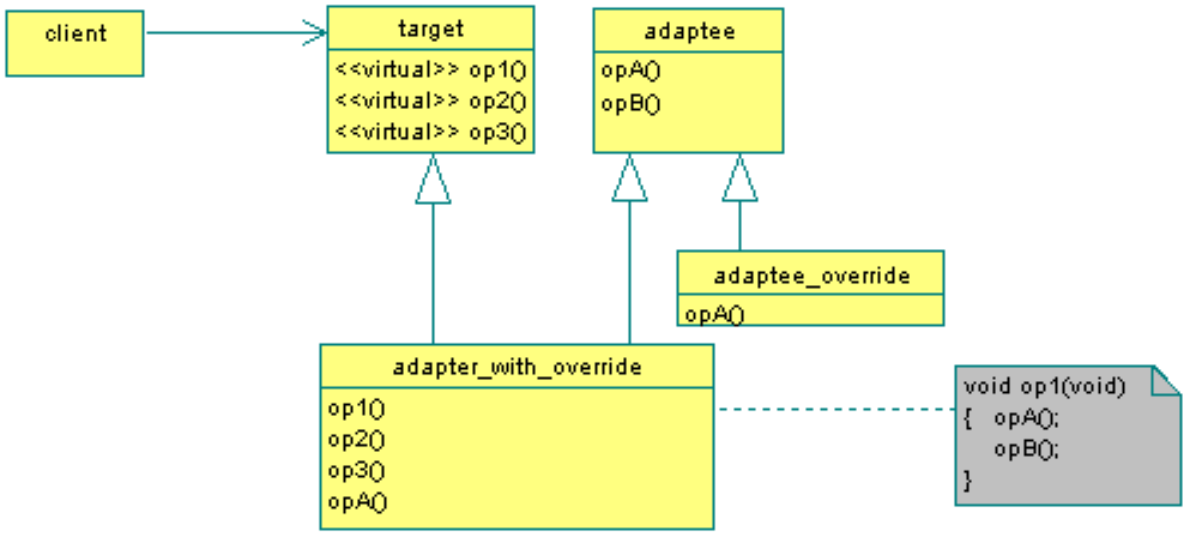


Da SWE: Class adapter

- Un solo oggetto :-)
- L'adapter non eredita un'eventuale override dell'adaptee :-(

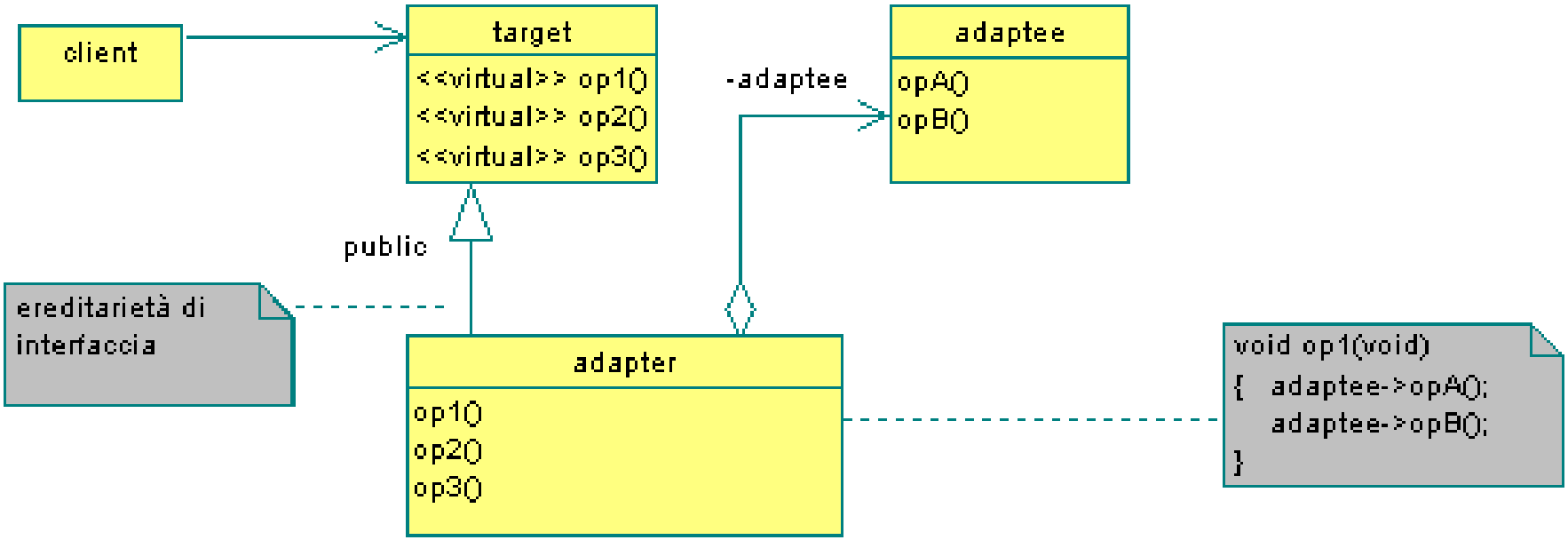


- Ma l'adapter può effettuare l'override :-|



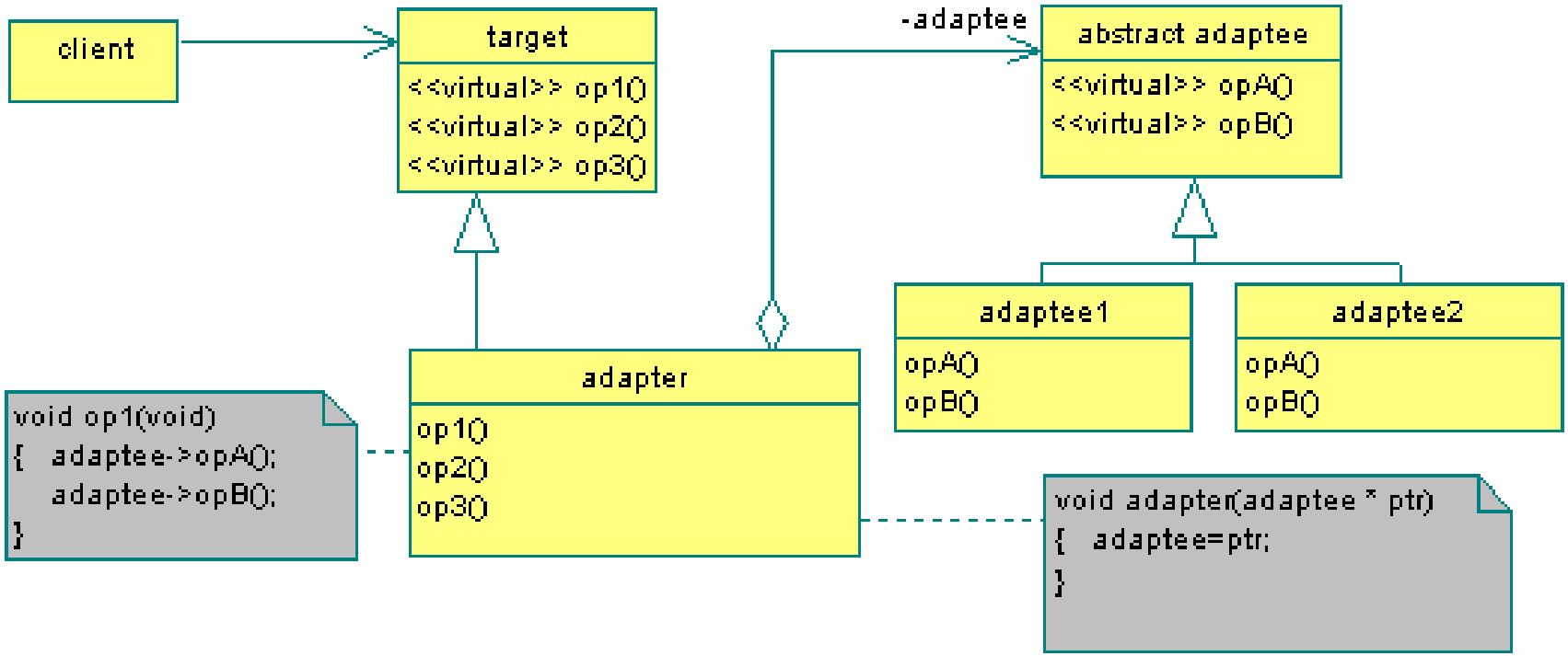
Da SWE: Object adapter

- Implementa l’interfaccia del target delegando all’adaptee



Da SWE: Object adapter

- Può utilizzare implementazioni diverse dell'adaptee :-)
  - Il costruttore dell'adapter deve installare l'adaptee concreto specifico



- Ci sono due oggetti e l'installazione è più complessa :-(
- In principio l'adaptee può essere sostituito dinamicamente :-)

- In ultimo è il trade-off nella scelta tra ereditarietà e composizione

Il problema delle configurazioni della topologia

- Nel caso di tipi con singola istanza
  - Dunque ortogonale rispetto all'approccio precedente
  - Test di una funzione con implementazione polimorfa per effetto di varietà nella configurazione delle deleghe o nella forma concreta che implementa le interfacce
  - Esempio: strategy o decorator

Class dependency graph

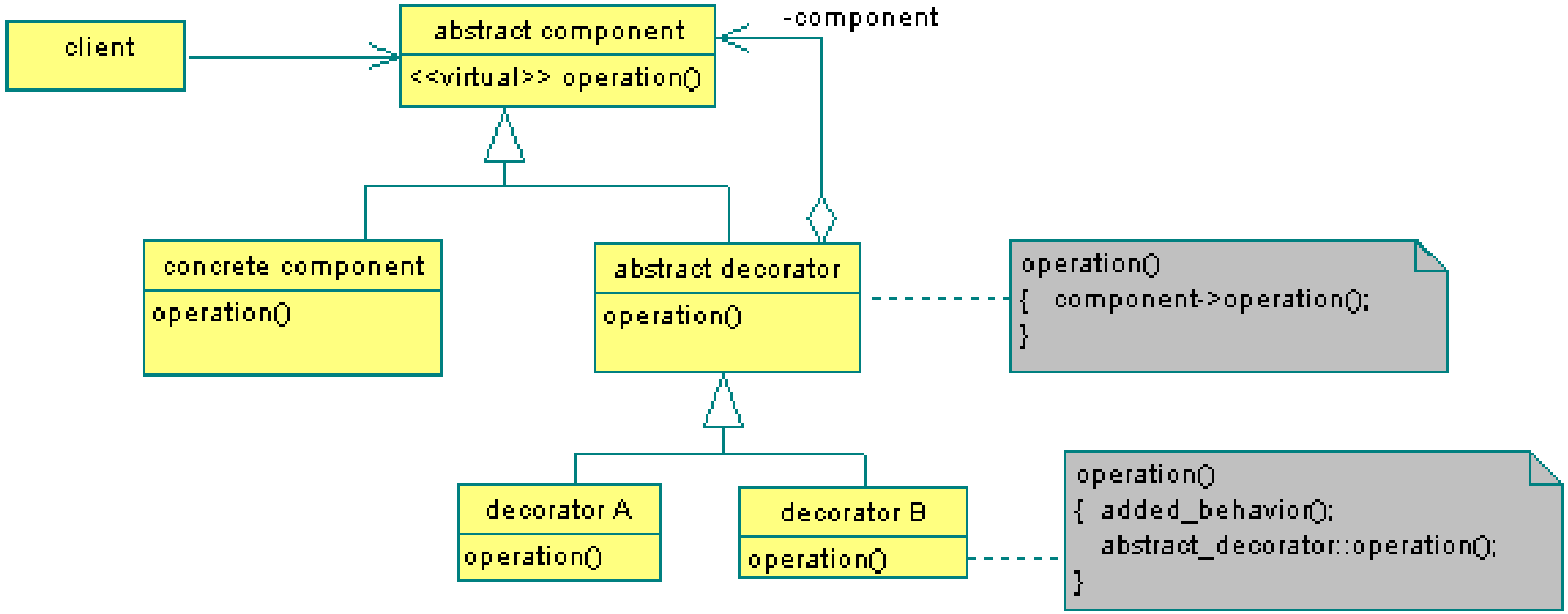
Class dependency graph

- I vertici sono classi
- Gli archi sono dipendenze
  - Di uso: un metodo nella classe client usa un metodo nella classe server
  - Di implementazione: un metodo nella classe base è implementato nella classe derivata
  - Di ereditarietà: un metodo nella classe derivata è ereditato dalla classe base
  - Di upcall esplicita: un metodo overridden

Da SWE: Decorator - (behavioral)

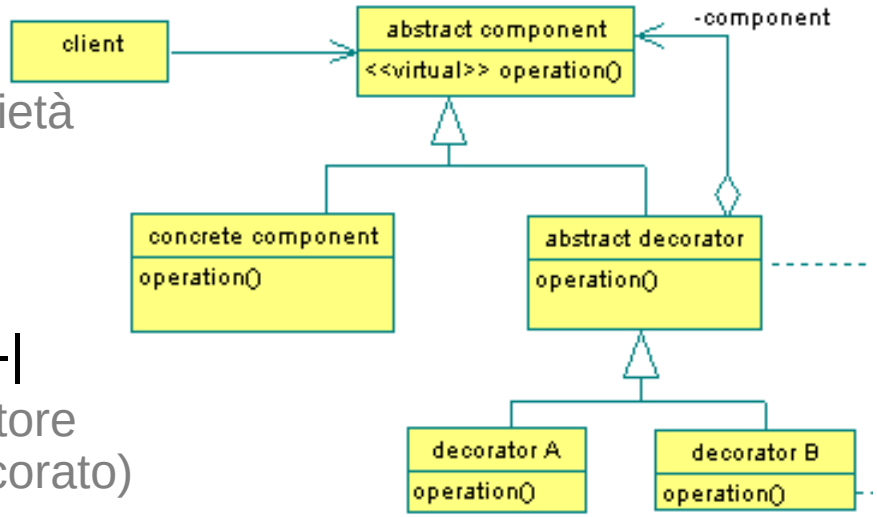
Aggiunge dinamicamente responsabilità ad un oggetto in modo da estenderne la funzionalità

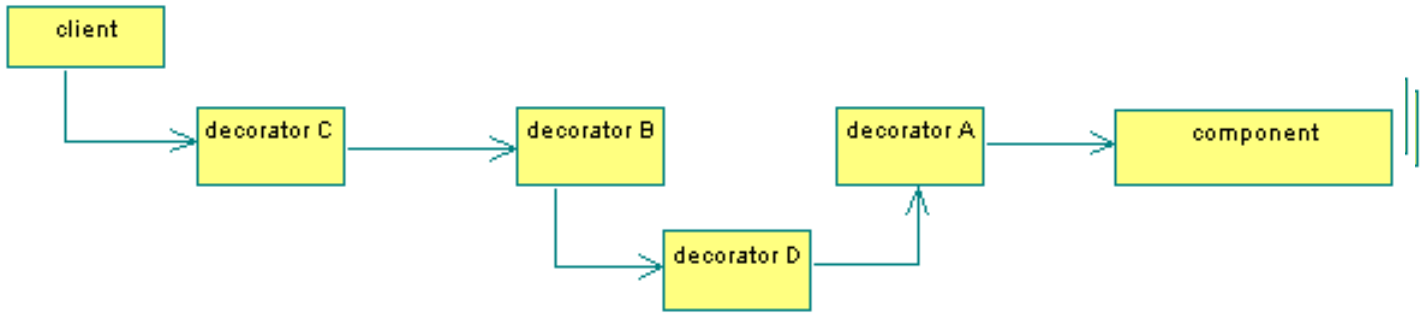
- Evoluzione del proxy verso una struttura ripetitiva



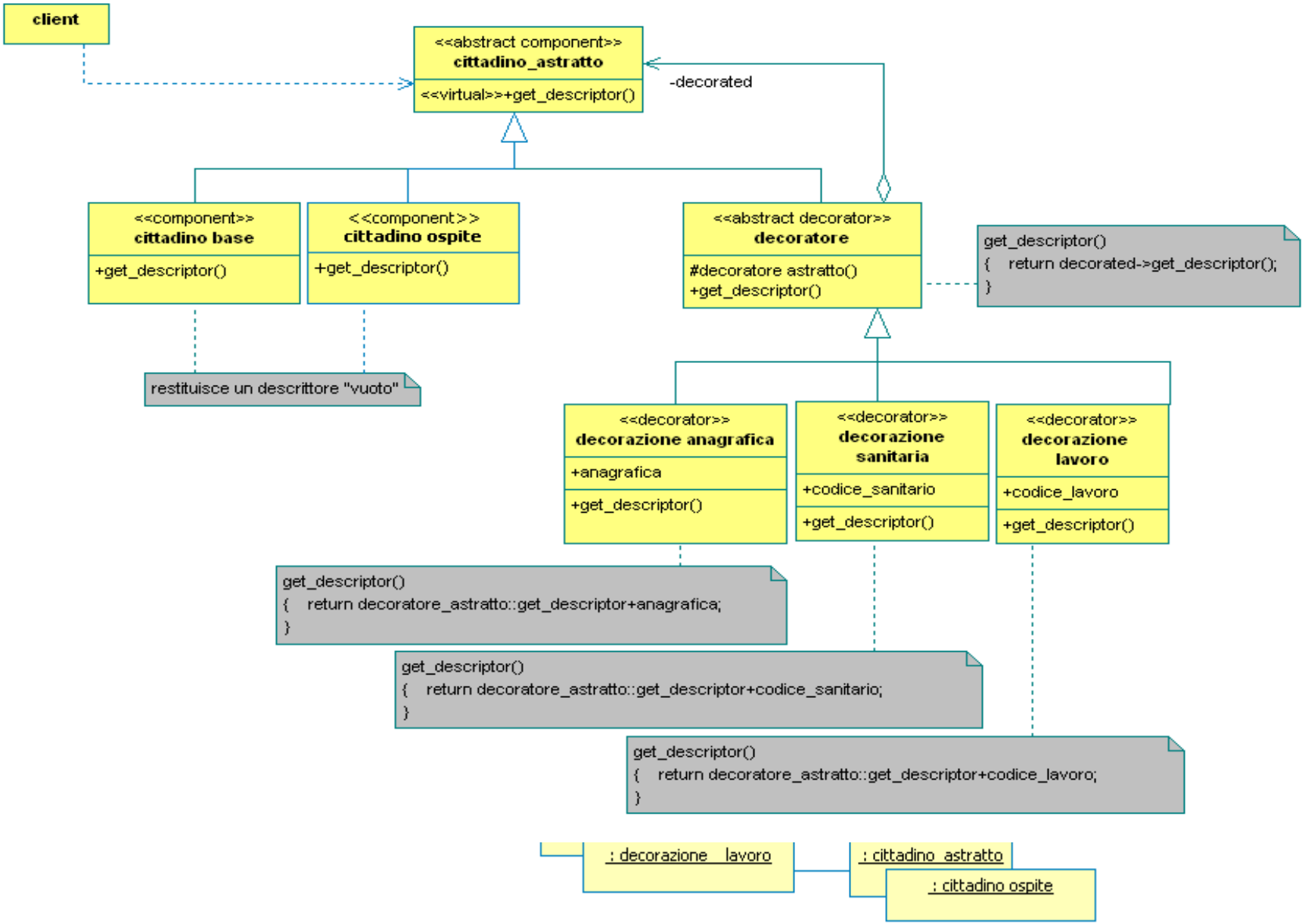
Da SWE: Decorator - conseguenze

- Il numero delle classi scala in modo lineare :-)
  - In uno schema basato sull'ereditarietà scalerebbe in modo quadratico
- N+1 oggetti :-(
  - Difficoltà nel debug
- Necessaria l'installazione iniziale :-|
  - A carico del costruttore del decoratore (riceve il riferimento all'oggetto decorato)
- Efficienza :-(
  - Forwarding
  - Le classi alte devono essere leggere
- Configurazione modificabile dinamicamente :-)
  - Addition and withdraw

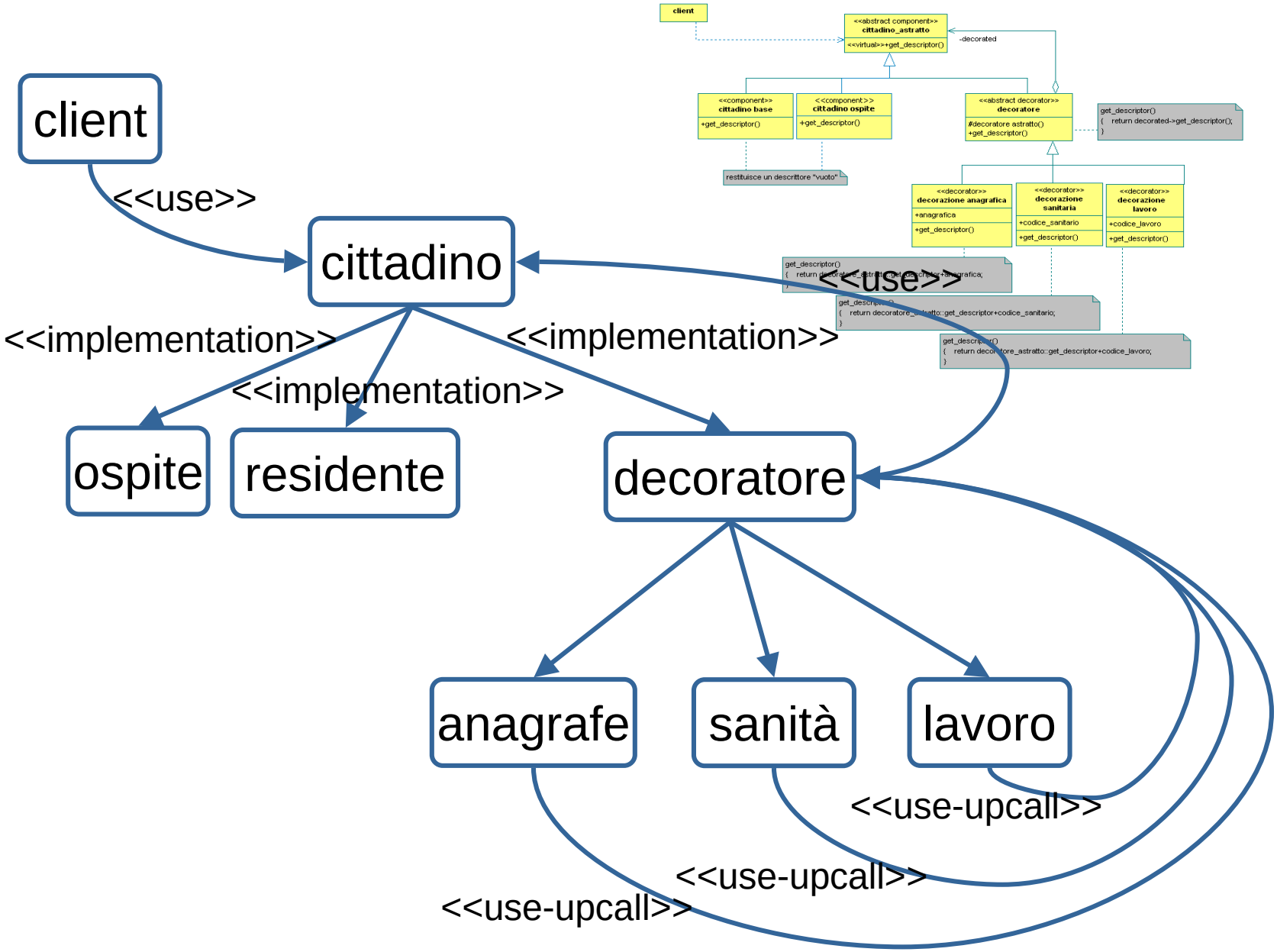




Class dependency graph: esempio nel caso di un decorator

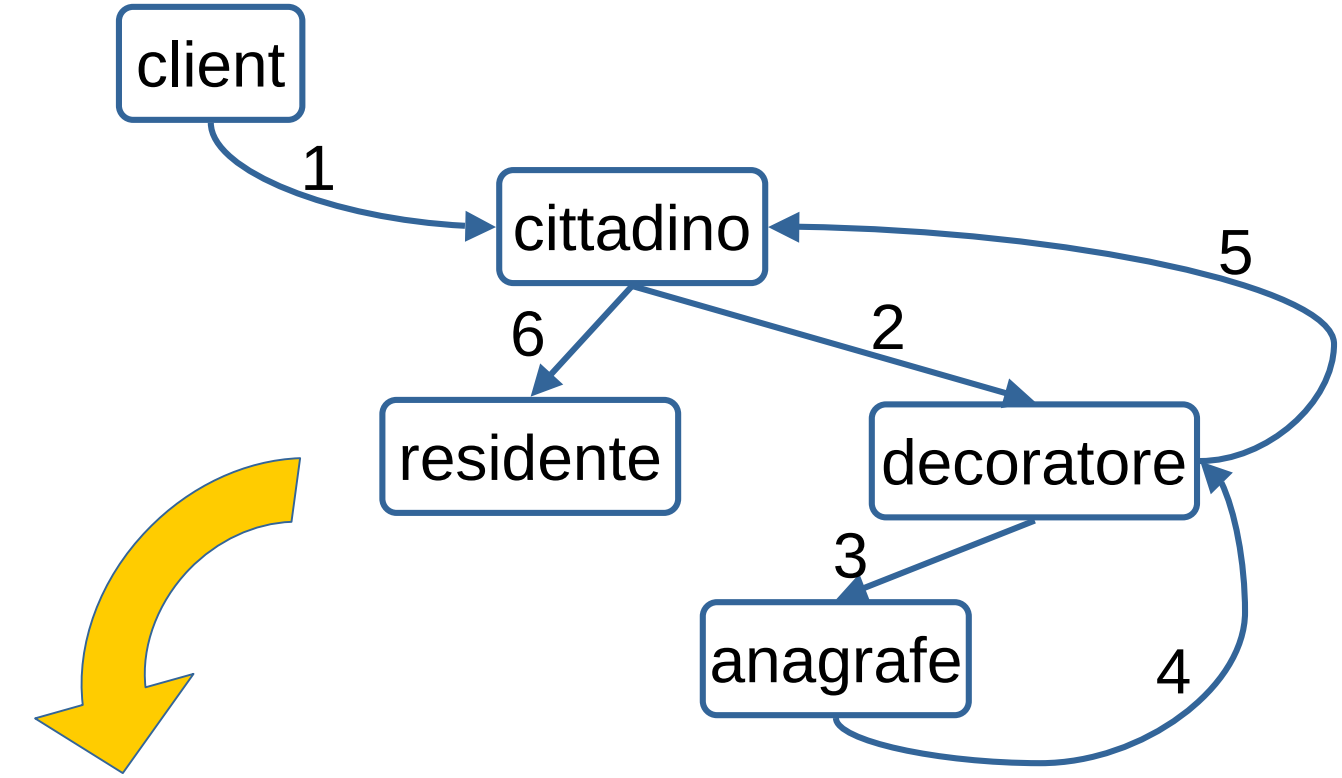


Class dependency graph

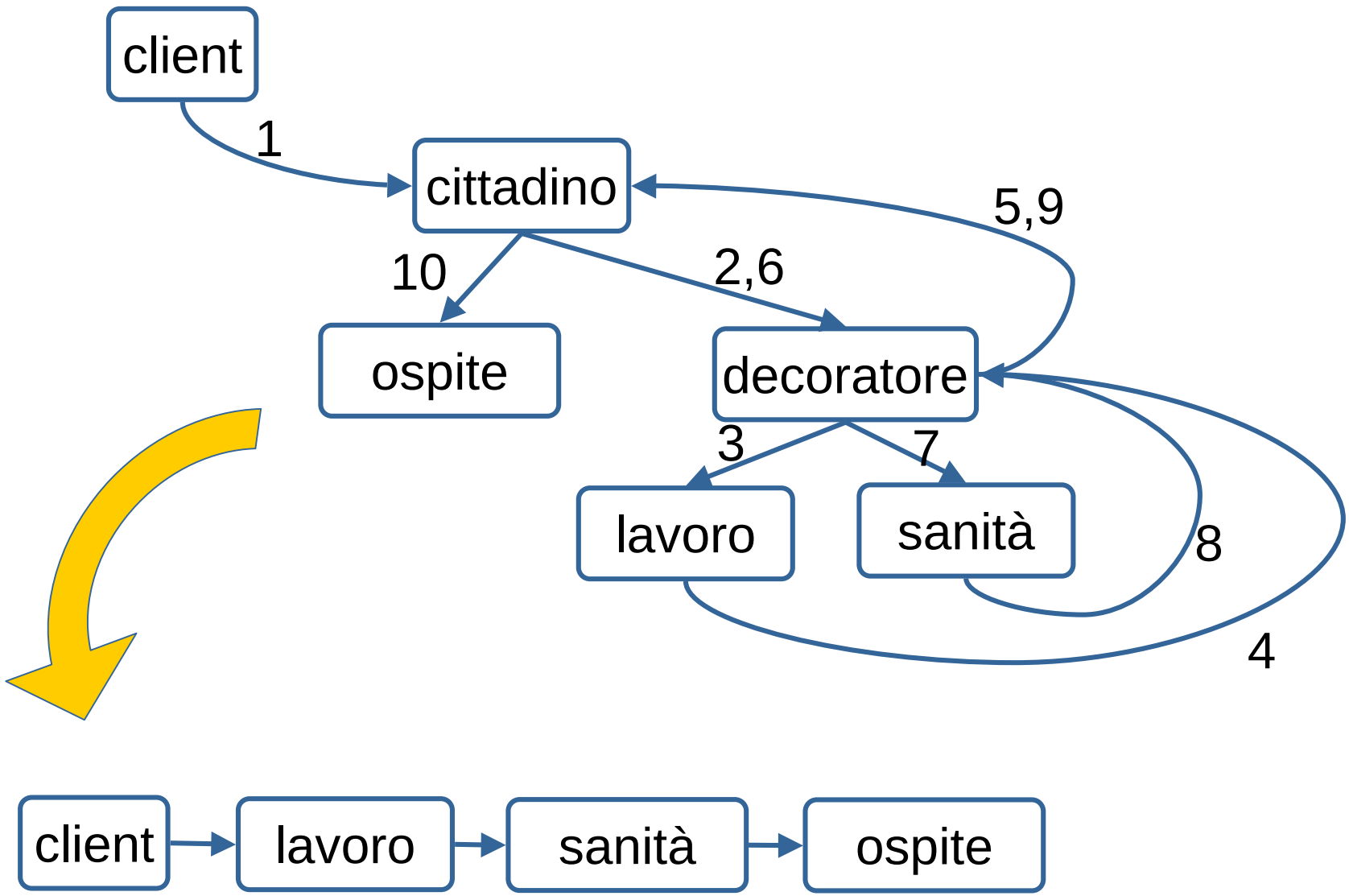


Cammini nel grafo: dalle classi agli oggetti

- Un path nel grafo rappresenta una catena di dipendenze in una configurazione determinata da
  - Oggetti in vita
  - Configurazione delle loro deleghe



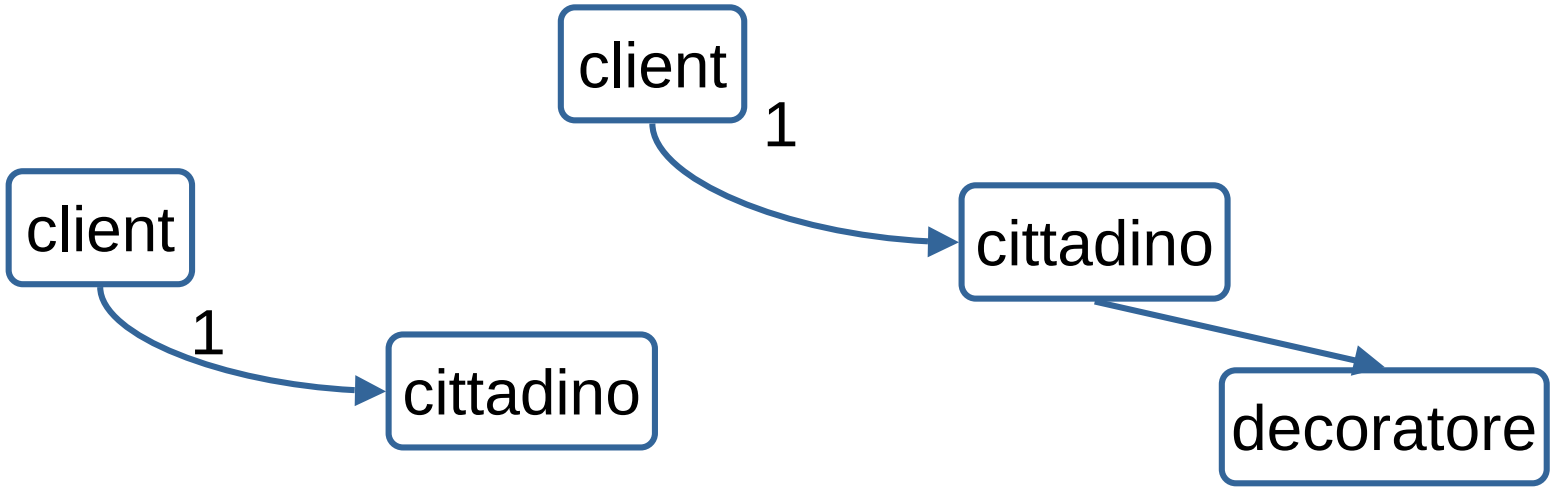
Cammini nel grafo



Cammini nel grafo

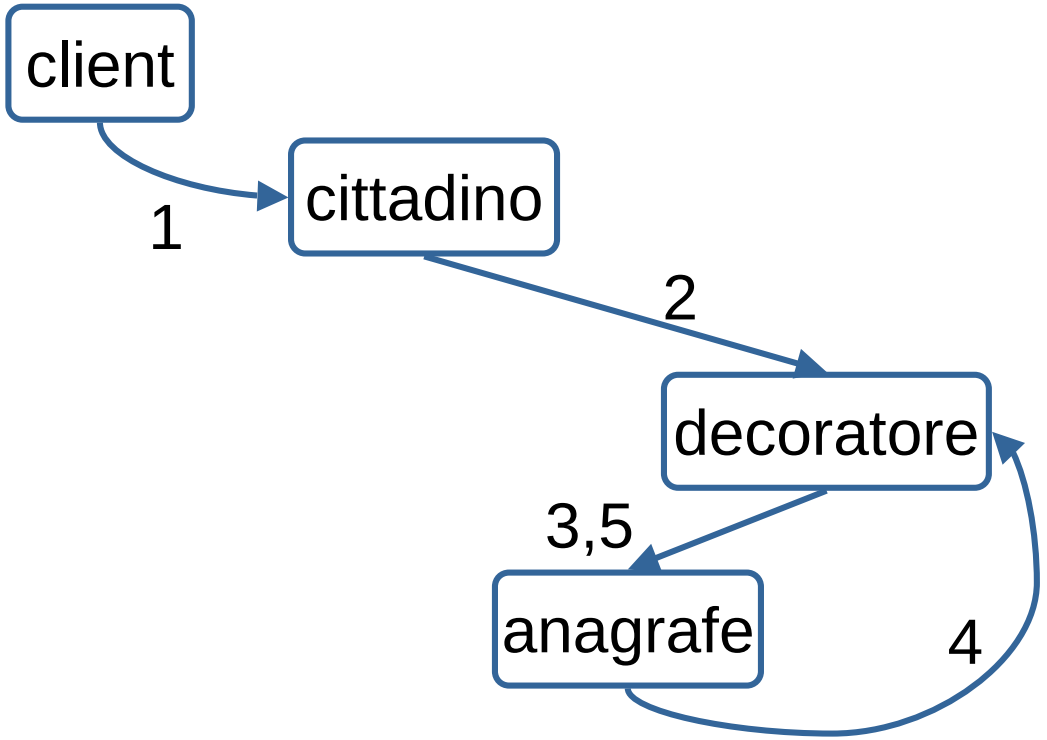
- Non tutti i paths sono fattibili per via dei vincoli definiti dalla politica di dynamic binding del linguaggio

- Un oggetto parzialmente astratto non può essere istanziato
  - Implica che le gerarchie di ereditarietà devono essere discese fino a raggiungere una classe completamente concreta
  - E in questa prospettiva non sono istanziabili oggetti con costruttore protected o private



Cammini nel grafo

- Su un upcall esplicito non viene applicato l'override della classe derivata



- Una classe estesa in più una classi derivate è tuttavia sempre implementato in una sola delle forme possibili
  - Implica che non può esistere un broadcast lungo una gerarchia di implementazione
  - Viceversa puo' esistere una relazione uno a molti lungo una relazione di uso
  - Su questi potrebbe esistere un annotazione di alternativa

Significato di alcune coperture sul class dependency graph

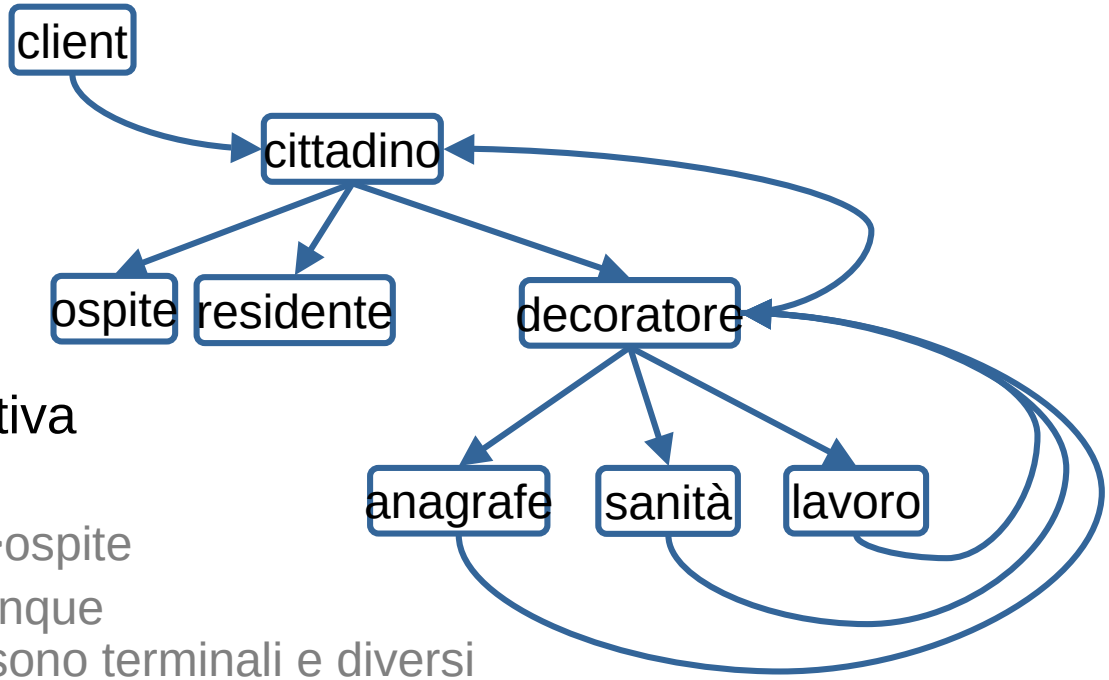
- All nodes:
garantisce di avere esercitato ogni classe concreta almeno una volta

- All edges: garantisce di avere esercitato tutte le relazioni di delega e tutte le implementazioni diverse
  - Perché risulti diverso da all nodes occorre avere allo stesso tempo polimorfismo e concorrenza (branching e confluenza nella vista delle classi)
  - Il polimorfismo fa sì che un oggetto abbia alternative nella dipendenza (il barnch). E' un caso comune. Lo strategy è un buon esempio.
  - La concorrenza fa sì che un oggetto possa essere usato da più oggetti (la confluenza). E' meno frequente, ma capita comunque. Ad esempio nell'observer il subject è usato da più observer concreti per quanto attraverso la facciata di un comune observer astratto.

- All paths: copre tutte le configurazioni della topologia delle dipendenze

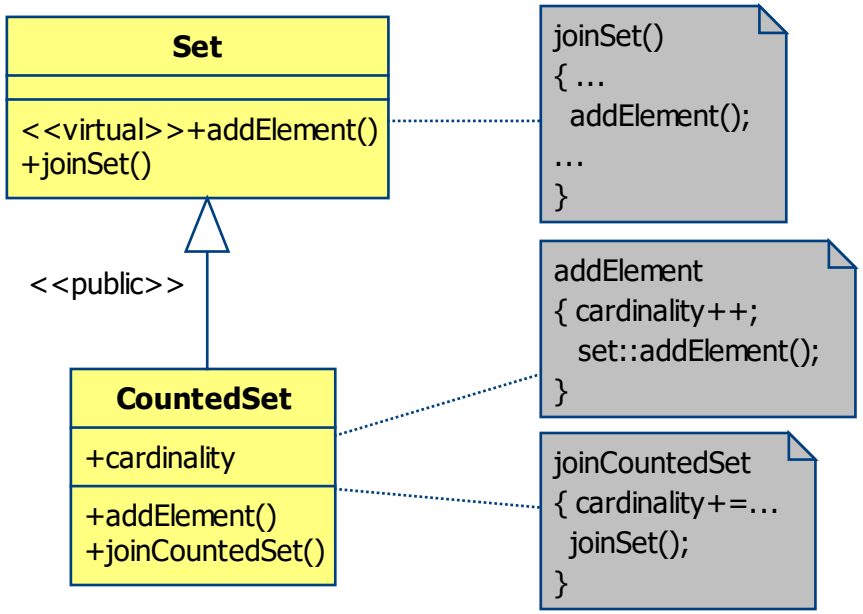
Esempio: coperture sul class dependency graph di un decorator

- Esempi di paths fattibili



- Copertura all nodes
  - Client->lavoro->residente
  - Client->sanità->ospite
  - Client->anagrafica->ospite
- Copertura all nodes alternativa
  - Client->lavoro->residente
  - Client->sanità->anagrafe->ospite
  - Due casi mi servono comunque
    perché residente e ospite sono terminali e diversi
- La copertura all edges coincide con all nodes
  - Mancano condizioni di concorrenza che risultino in una confluenza sul grafo

Il caso della fragile base class



- self-loop sul class dependency graph