



# Design Pattern

Ingegneria del Software

# Argomenti

- ~ Idee di base nell'applicazione dei pattern
- ~ Pattern GRASP e pattern GOF (Gamma)

# Perché i Pattern

- ~ Un sistema software ben progettato è facile da comprendere, mantenere ed estendere
  - le scelte fatte consentono buone opportunità di riusare i suoi componenti software in applicazioni future
- ~ Per esempio: sviluppo iterativo
  - il software viene costantemente modificato, esteso, manutenuto
  - comprensibilità e semplicità facilitano le attività evolutive
  - caratteristiche importanti non solo nello sviluppo iterativo

# Perché i Pattern

~ Le qualità di un buon sistema software

· & comprensibilità

· & modificabilità

· & impatto ai cambiamenti

· & flessibilità

· & riuso

· & semplicità

~ Progettazione modulare

# Progettazione modulare

- ~ Il software deve essere decomposto in elementi software (moduli) coesi e debolmente accoppiati
- ~ Coesione
  - Misura di quanto le operazioni sono accoppiate all'interno di un modulo
- ~ Accoppiamento
  - misura di quanto più moduli sono dipendenti tra loro

# Progettazione modulare

- ~ Garantire la qualità del software significa affrontare **problemi ricorrenti**
- ~ Individuare un insieme di oggetti
  - & correttamente individuato
  - & il più possibile riusabile
  - & definire al meglio le relazioni tra classi e gerarchie di ereditarietà

# Pattern - definizione

“Ogni pattern descrive un problema che si ripete più e più volte nel nostro ambiente, descrive poi il nucleo della soluzione del problema, in modo tale che si possa riusare la soluzione un milione di volte, senza mai applicarla alla stessa maniera.”

Christopher Alexander - Architetto -

# Pattern caratteristiche

- ~ Rappresentano soluzioni a problemi ricorrenti
- ~ Permettono di organizzare un **progetto OO** favorendo **riuso**
- ~ Permettono di imparare dal lavoro degli altri
- ~ Evitano al progettista di “reinventare la ruota”
- ~ Permettono di definire un linguaggio comune e semplice per la comunicazione tra i progettisti

# Pattern caratteristiche

- ~ Portano di norma ad buona progettazione
- ~ Permettono di scrivere codice che usa strutture valide
- ~ Non risolvono tutti problemi!



# Due tipi di Pattern

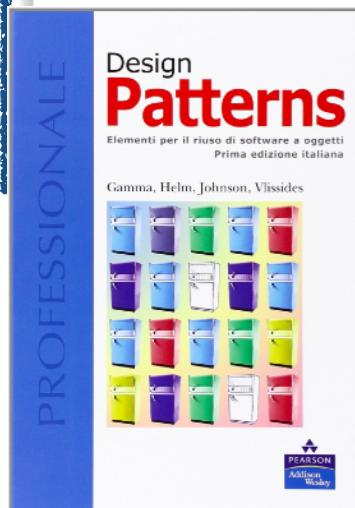
~ Pattern GRASP - General Responsibility Assignment Software Pattern

· per assegnare la responsabilità agli oggetti

~ Design Pattern GOF

· Per le idee di progettazione più avanzate

# Riferimenti bibliografici



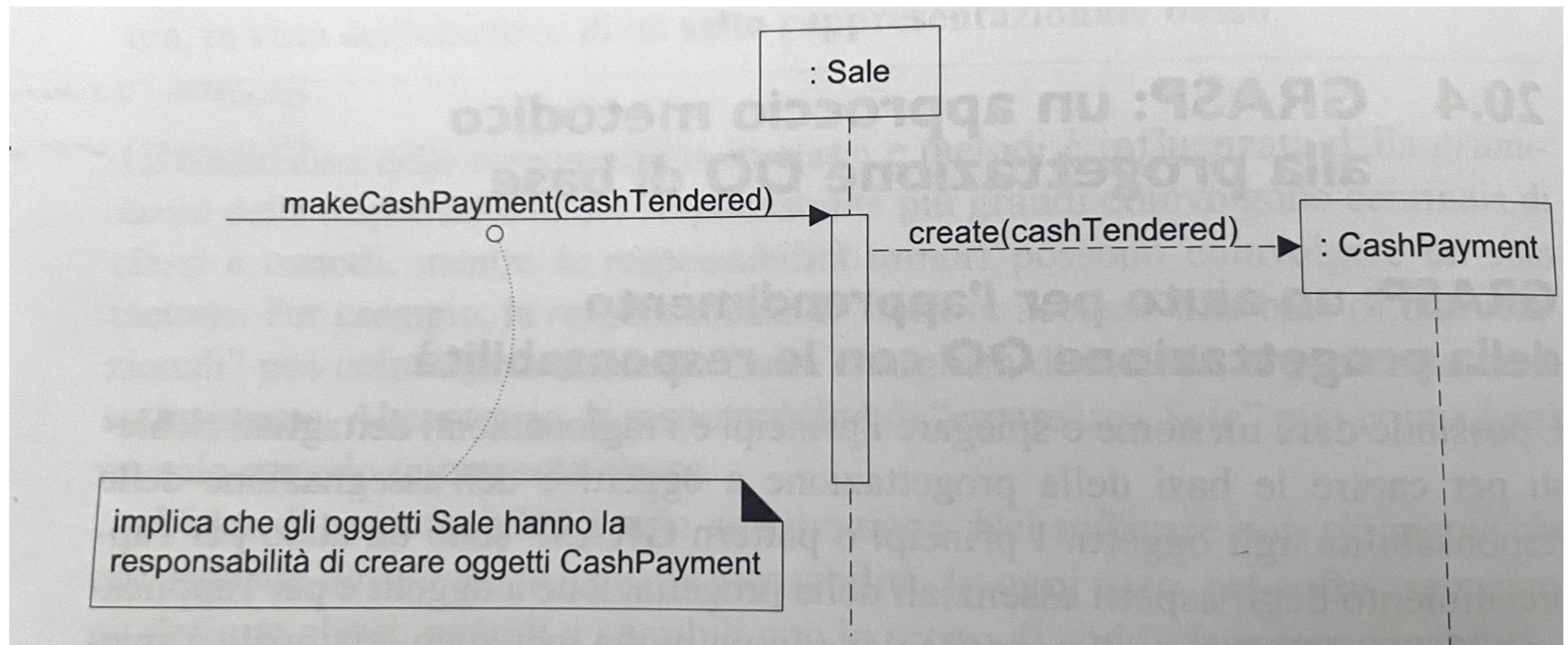
- ~ E. Gamma, R. Helm, R. Johnson, J. Vlissides (GOF, Gang of Four), Design Patterns – Elementi per il riuso di software a oggetti, Addison Wesley
- ~ Craig Larman Applicare UML e i Pattern, Prentice Hall



# Che cosa sono i pattern

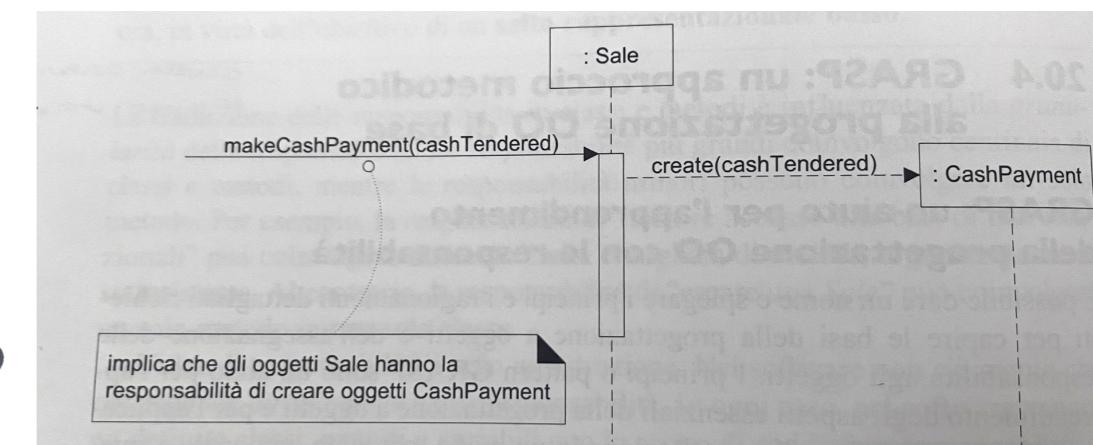
- ~ Principalmente usare pattern significa **assegnare responsabilità** agli oggetti (si rimanda al libro di testo)
- ~ La stesura dei diagrammi di interazione diventa l'occasione per considerare tali responsabilità.
- ~ La responsabilità agli oggetti possono essere assegnate mentre si esegue la codifica oppure durante la modellazione

# Che cosa sono i pattern - esempio



# Che cosa sono i pattern - esempio

- ~ Agli oggetti **Sale** è stata assegnata una responsabilità di creare oggetti **CashPayment**
- ~ questa responsabilità viene concretamente richiamata con un messaggio **MakeCashPayment**
- ~ gestita con un corrispondente metodo **MakePayment**
- ~ l'adempimento di questa responsabilità richiede una collaborazione per creare l'oggetto **CashPayment** e chiamare il suo costruttore



# Che cosa sono i pattern

- ~ Gli sviluppatori OO esperti e altri sviluppatori del software hanno accumulato un repertorio contenente sia **principi generali** che **soluzioni idiomatiche**
  - e guidano nella creazione del software
  - questi principi e idiomì se codificati in un formato strutturato che descriva il problema e la soluzione e a cui è assegnato un nome possono essere chiamati **pattern**
  - per esempio un pattern può essere il seguente



# Che cosa sono i pattern

Nome del pattern

Information Expert

Problema

Qual è un principio di base con cui assegnare responsabilità agli oggetti?

Soluzione

Assegna una responsabilità alla classe che ha le informazioni necessarie per soddisfarla

# Che cosa sono i pattern

- ~ Nella progettazione object oriented un pattern è una descrizione con un nome di un problema di progettazione ricorrente e di una soluzione ben provata che può essere applicata a nuovi contesti
- ~ un pattern da consigli su come applicare la sua soluzione in circostanze diverse e considera i vantaggi e i compromessi
- ~ molti pattern, data una categoria specifica di problemi, guidano l'assegnazione di responsabilità agli oggetti



# Semplicemente

- ~ Un pattern è una coppia problema/soluzione **ben conosciuta** e con un **nome** che può essere applicata in nuovi contesti con consigli su come applicarla in situazioni nuove e con una discussione sui relativi compromessi, implementazioni, variazioni e così via



# Un po' di storia

- ~ L'idea di pattern con nome deriva da Kent Beck (l'ideatore dell'xp programming)
- ~ metà anni 80
- ~ il 1994 fu momento fondamentale nella storia dei libri sul pattern, sulla progettazione OO e sulla progettazione del software
- ~ in quell'anno fu pubblicato il libro estremamente autorevole e di larga diffusione **Design Patterns** di Gamma, Helm, Johnson e Vlissides
- ~ Questo libro è considerato il libro dei libri sui design pattern, descrive 23 pattern di progettazione (esempi: Strategy, Adapter)
- ~ Questi 23 pattern creati dai 4 autori sono chiamati GOF - Gang of Four

# Esempio ed idee principali

- ~ Pattern GRASP (General Responsibility Assignment Software Patterns)
- ~ I pattern GRASP sono 9
- ~ Nelle prossime slide vediamo degli esempi generici del
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion



# Creator

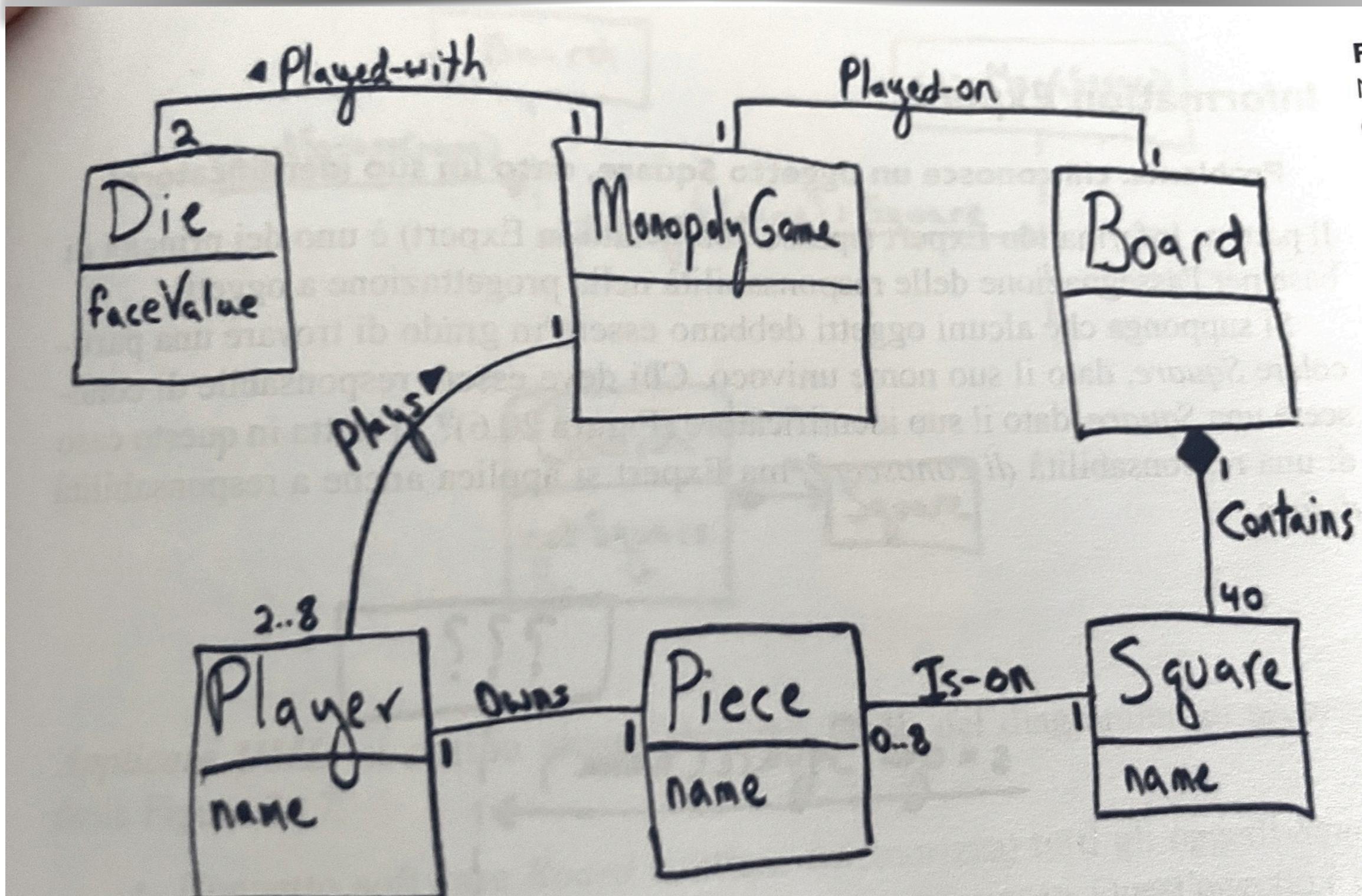
- ~ Problema: chi crea gli oggetti Square (riferimento esempio del Monopoly sul libro di testo)?
- ~ Uno dei problemi comuni che occorre affrontare nella progettazione oggetti è: chi crea l'oggetto X?
- ~ si tratta di una responsabilità "del fare", per esempio nel caso studio del Monopoly, chi crea un oggetto software Square?
- ~ nel caso di java questa domanda può voler dire: in quale classe deve essere scritto new Square ()...?



# Creator

- ~ È ovvio che qualsiasi oggetto può creare un oggetto Square ma quale sarebbe la scelta più idonea? E perché?
- ~ Per esempio, come creatore si potrebbe scegliere un oggetto Dog (cioè una classe arbitraria)
- ~ questa scelta però non sarebbe opportuna perché non si richiama al nostro modello mentale del dominio
- ~ nell'esempio del libro di testo è l'oggetto Board che deve creare gli oggetti Square

# Creator





# Creator

~ La definizione del Creator (semplificata) ->  
Concrete Factory ed Abstract Factory

Nome	Creator
Problema	Chi crea l'oggetto A?

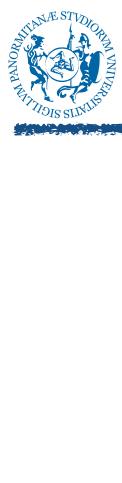
Soluzione  
(consiglio)

Assegna alla classe B la responsabilità di creare una classe A se una delle seguenti condizioni è vera (più sono vere e meglio è)

- B “contiene” o aggrega con composizione oggetti di tipo A
- B registra A
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A

# Creator

- ~ Linee di principio per l'assegnazione di responsabilità:
  - se Premessa-> B ed A fanno riferimento ad oggetti software che possono essere già stati creati
  - si tenta di applicare Creator degli oggetti software esistenti che soddisfano il ruolo di B
  - se la progettazione è appena iniziata si ricorre al modello di dominio
  - nell'esempio è possibile notare che Board contiene oggetti Square
  - in questo caso è un punto di vista concettuale e non software



# Creator

## ~ Considerazioni generali:

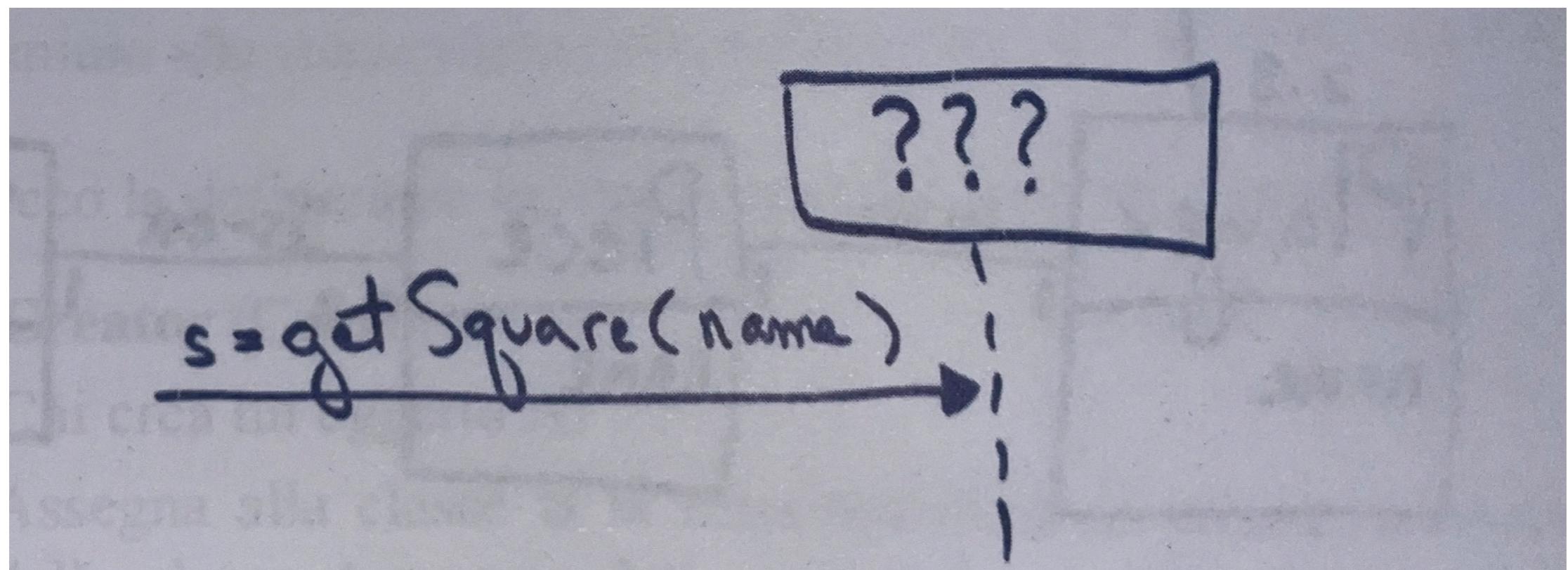
- Si ricordi che una pratica della modellazione agile consiste nel creare in parallelo modelli comportamentali, dinamici e statici degli oggetti
- viene disegnato sia un diagramma di sequenza che è un diagramma delle classi entrambi parziali
- si abbina perfettamente ai consigli per applicare i pattern

# Information Expert

- ~ Problema: Chi conosce un oggetto Square, dato un suo identificatore?
- ~ Il pattern Information Expert (spesso abbreviato in Expert) è uno dei principi di base per l'assegnazione delle responsabilità nella progettazione a oggetti
- ~ si supponga che alcuni oggetti debbano essere in grado di trovare una particolare Square dato il suo nome univoco
- ~ chi deve essere responsabile di conoscere una Square dato il suo identificatore?
- ~ si tratta in questo caso di una responsabilità "di conoscere" ma Expert si applica anche a responsabilità "di fare"

# Information Expert

~ responsabilità di "conoscere" significa questo:





# Information Expert

Nome

Information Expert

Problema

Qual è un principio di base per assegnare responsabilità agli oggetti?

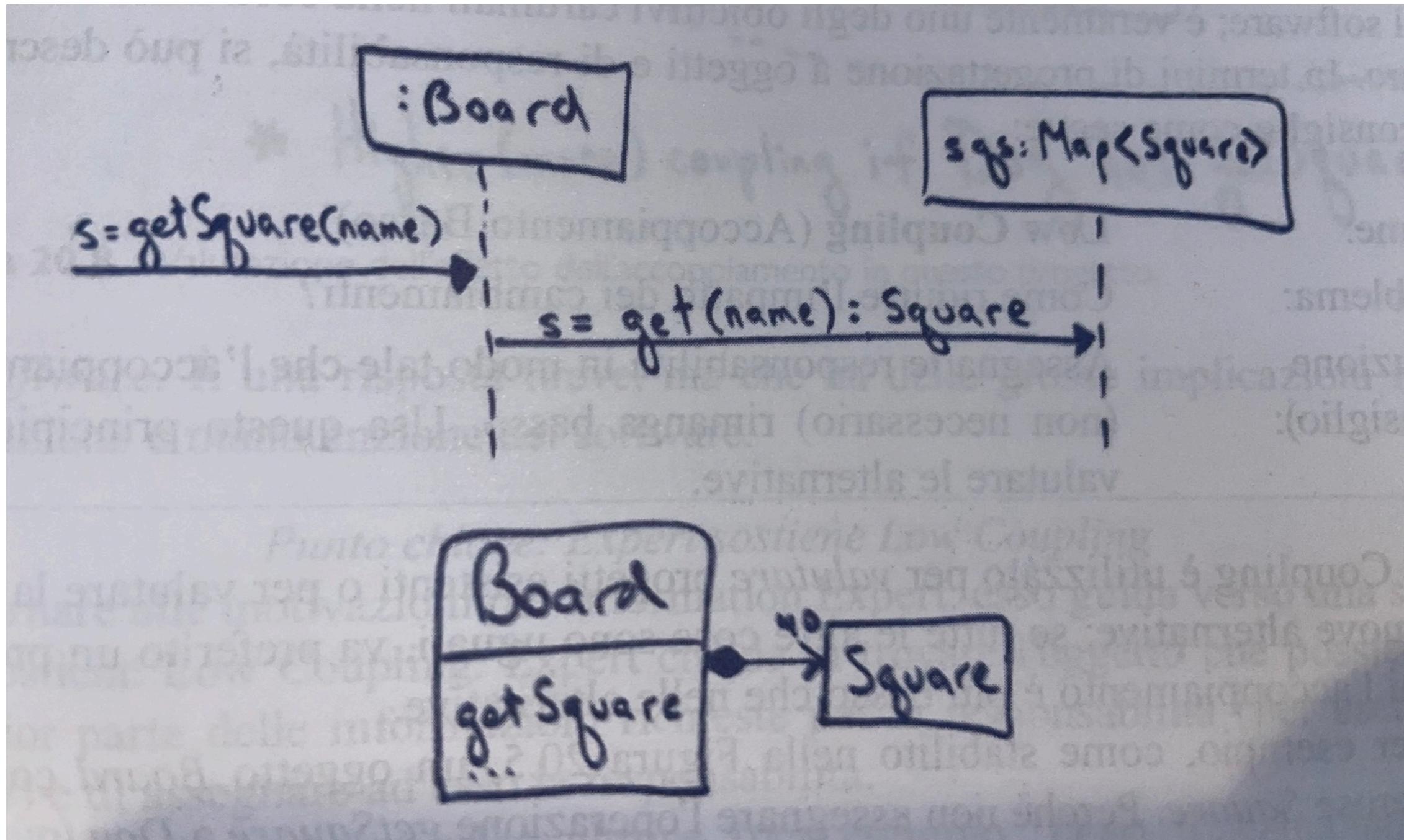
Soluzione  
(consiglio)

Assegna una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla

# Information Expert

- ~ una responsabilità necessita di informazioni per essere soddisfatta
  - su altri oggetti
  - sullo stato di un oggetto
  - sul mondo che circonda l'oggetto
  - informazioni che l'oggetto può ricavare e così via
- ~ per potere recuperare o presentare un qualsiasi Square (dato il suo nome) qualche oggetto deve conoscere o avere informazioni su tutti gli oggetti Square
- ~ come mostrato prima, un oggetto software Board aggrega tutti gli oggetti Square pertanto Board possiede le informazioni necessarie per soddisfare questa responsabilità

# Information Expert



# Information Expert

- L'oggetto software *Board* contiene (memorizza) tutti gli oggetti *Square* mediante l'utilizzo dell'oggetto *sqs : Map<Square>*. Quest'oggetto è una collezione di tipo *Map* che contiene oggetti *Square*. L'uso di una mappa consente di effettuare in modo efficiente accessi sulla base di una chiave di ricerca (per esempio, il nome della casella). Il nome *sqs* è una contrazione di *squares* (*caselle*).
- La variabile *s* nel messaggio iniziale *getSquare* e la variabile *s* restituita dal messaggio successivo *get* fanno riferimento allo stesso oggetto.
- L'espressione messaggio *s = get(name) : Square* indica che il tipo di *s* è un riferimento a un'istanza di *Square*.

# Low Coupling

- ~ Problema: Perché Board e non Dog?
- ~ Il pattern Expert suggerisce di assegnare, nell'esempio di prima, la responsabilità di conoscere una particolare Square all'oggetto Board
- ~ Perché?
- ~ La risposta si trova nel concetto di Low coupling
- ~ L'accoppiamento è una misura di quanto fortemente un elemento è connesso ad altri elementi, lì conosce o dipende da essi

# Low Coupling

~ Accoppiamento o una dipendenza

- quando l'elemento da cui si dipende subisce una modifica ciò può ripercuotersi sull'elemento dipendente
- per esempio una sottoclasse è fortemente accoppiata ad una superclasse
- un oggetto A che richiama le operazioni dell'oggetto B ha un accoppiamento ai servizi di B

~ il principio di LOW COUPLING si applica a molte attività dello sviluppo software

- obiettivo cardine della costruzione del software



# Low Coupling

Nome

Low Coupling

Problema

Come ridurre l'impatto dei cambiamenti?

Soluzione  
(consiglio)

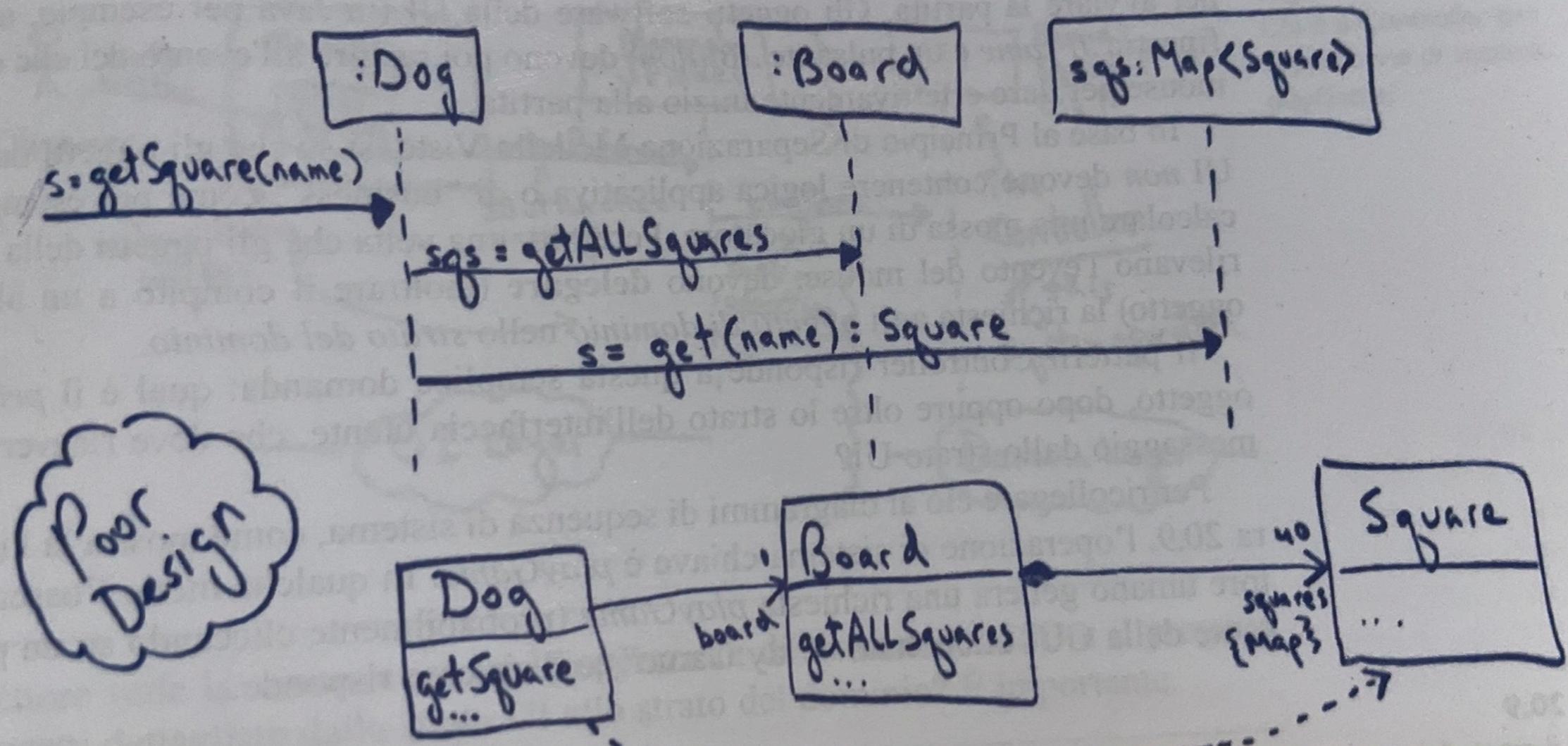
Assegna la responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso, usa questo principio per valutare le alternative



# Low Coupling

- ~ Il Low Coupling è un principio di base per valutare progetti esistenti o per valutare la scelta tra nuove alternative
- ~ Se tutte le altre cose sono uguali va preferito un progetto in cui l'accoppiamento è più basso che nelle alternative
- ~ Le due alternative dell'esempio su Square sono:
  - assegnare l'operazione getSquare a Dog (cioè una classe arbitraria)
  - assegnare l'operazione a Board

# Low Coupling



\* Higher (more) coupling if Dog has `getSquare`!

# Low Coupling

- ~ Se Dog ha getSquare deve collaborare con Board per ottenere la collezione di tutte le Square contenute nella Board
  - sono probabilmente memorizzate in un oggetto collezione map che consente la ricerca tramite una chiave
  - il Dog può accedere ad una particolare Square tramite la chiave name e poi restituirla
- ~ L'accoppiamento complessivo di questo progetto con Dog prevede che sia Dog che Board devono conoscere gli oggetti Square
- ~ Accoppiamento complessivo è più basso nel progetto precedente e quindi è migliore in termini di sostegno dell'obiettivo di Low Coupling

# Controller

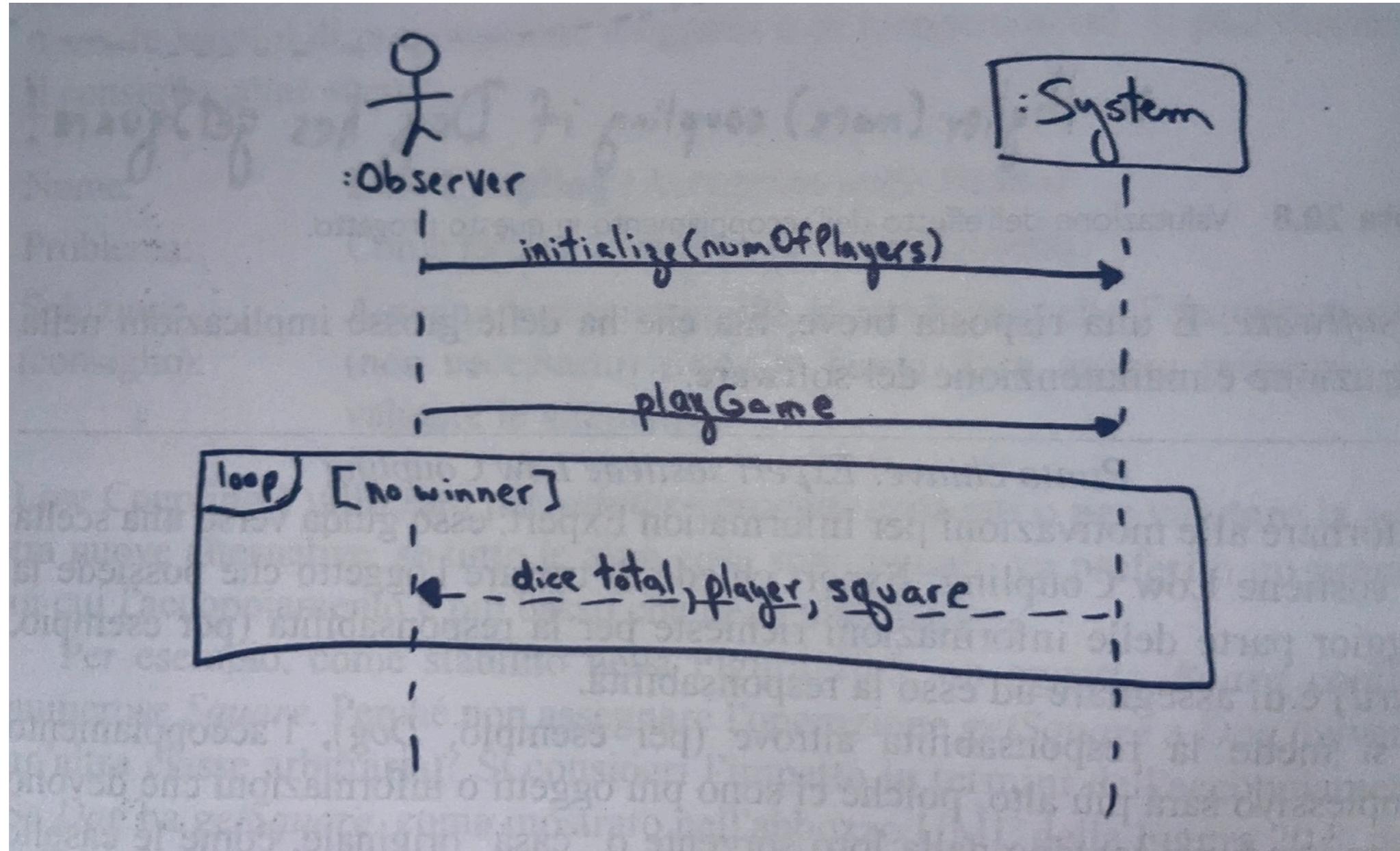
- ~ Una architettura a strati ha, tra gli altri, uno strato per l'interfaccia utente e uno strato del dominio
- ~ Gli attori, come l'osservatore umano nella partita del Monopoly, generano eventi UI come cliccare su un pulsante con il mouse per avviare la partita
- ~ Gli oggetti software della UI (in Java per esempio sono una finestra JFrame o un pulsante JButton) devono poi reagire all'evento del clic del mouse per dare effettivamente inizio alla partita
- ~ In base al principio di separazione Modello-vista, gli oggetti della UI non devono contenere logica di business (per esempio calcolare la mossa di un giocatore)
- ~ Ogni volta che gli elementi della UI rilevano l'evento del mouse, devono delegare la richiesta agli oggetti di dominio nello strato di dominio



# Controller

- ~ Problema: Qual è il primo oggetto, dopo (oppure oltre) lo strato dell'interfaccia utente, che deve ricevere il messaggio dallo strato UI?
  - ~ Nell'esempio del Monopoly, l'osservatore umano (l'attore) genera una richiesta PlayGame
    - cliccando su un qualche pulsante ed il sistema risponde (figura seguente)

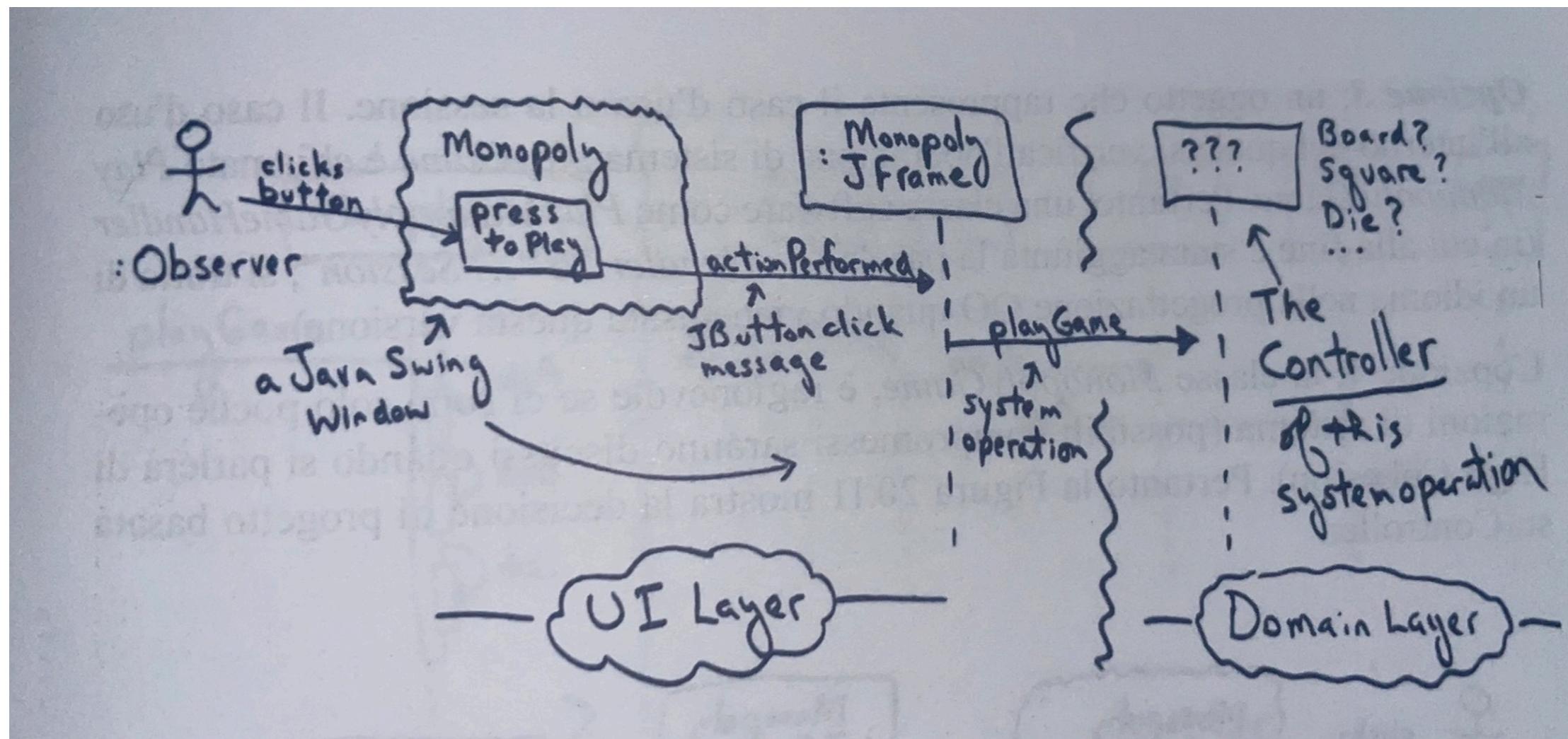
# Controller



# Controller

- ~ Ipotizzando l'esistenza di una finestra java swing Jframe della GUI e di un pulsante JButton la situazione è quella della figura successiva
- ~ cliccando su un JButton viene inviato un messaggio ActionPerformed ad un oggetto, spesso alla finestra Jframe stessa
- ~ successivamente la finestra Jframe deve adattare quel messaggio a qualcosa di semanticamente più significativo come un messaggio playGame e delegare il messaggio actionPerformed ad un oggetto di dominio nello strato del dominio

# Controller





# Controller

Nome	Controller
Problema	Qual è il primo oggetto oltre allo strato UI a ricevere e coordinare un'operazione di sistema?
Soluzione (consiglio)	<p>Assegna la responsabilità a un oggetto che rappresenta una delle seguenti scelte :</p> <ul style="list-style-type: none"><li>• rappresenta il sistema complessivo, un oggetto radice, un dispositivo all'interno del quale viene eseguito il software, un punto di accesso al software o un sottosistema principale (sono tutte varianti di un facade controller)</li><li>• rappresenta uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di sistema (un controller di caso d'uso controller di sessione)</li></ul>

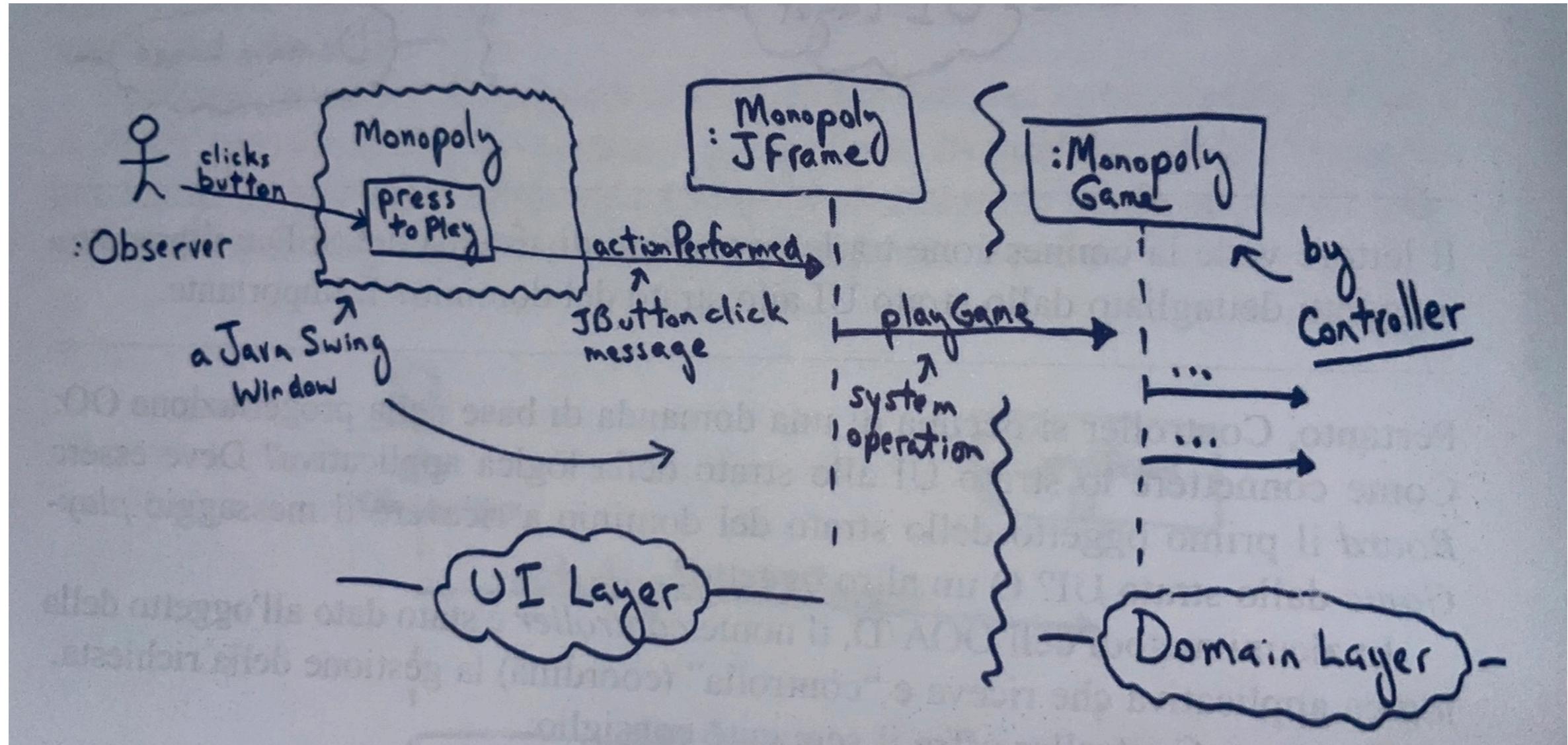
# Controller

~ Ci sono tre opzioni:

- 1. Un oggetto che rappresenta il sistema complessivo o un oggetto radice per esempio un oggetto chiamato Monopoly Game
- 2. Un oggetto che rappresenta un dispositivo all'interno del quale è seguito il software per esempio hardware specializzati come un telefono o un registratore di cassa (questa opzione non è applicabile in questo caso)
- 3. Un oggetto che rappresenta il caso d'uso o la sessione. Il caso d'uso all'interno del quale si verifica l'operazione di sistema play games chiamato play MonopolyGame

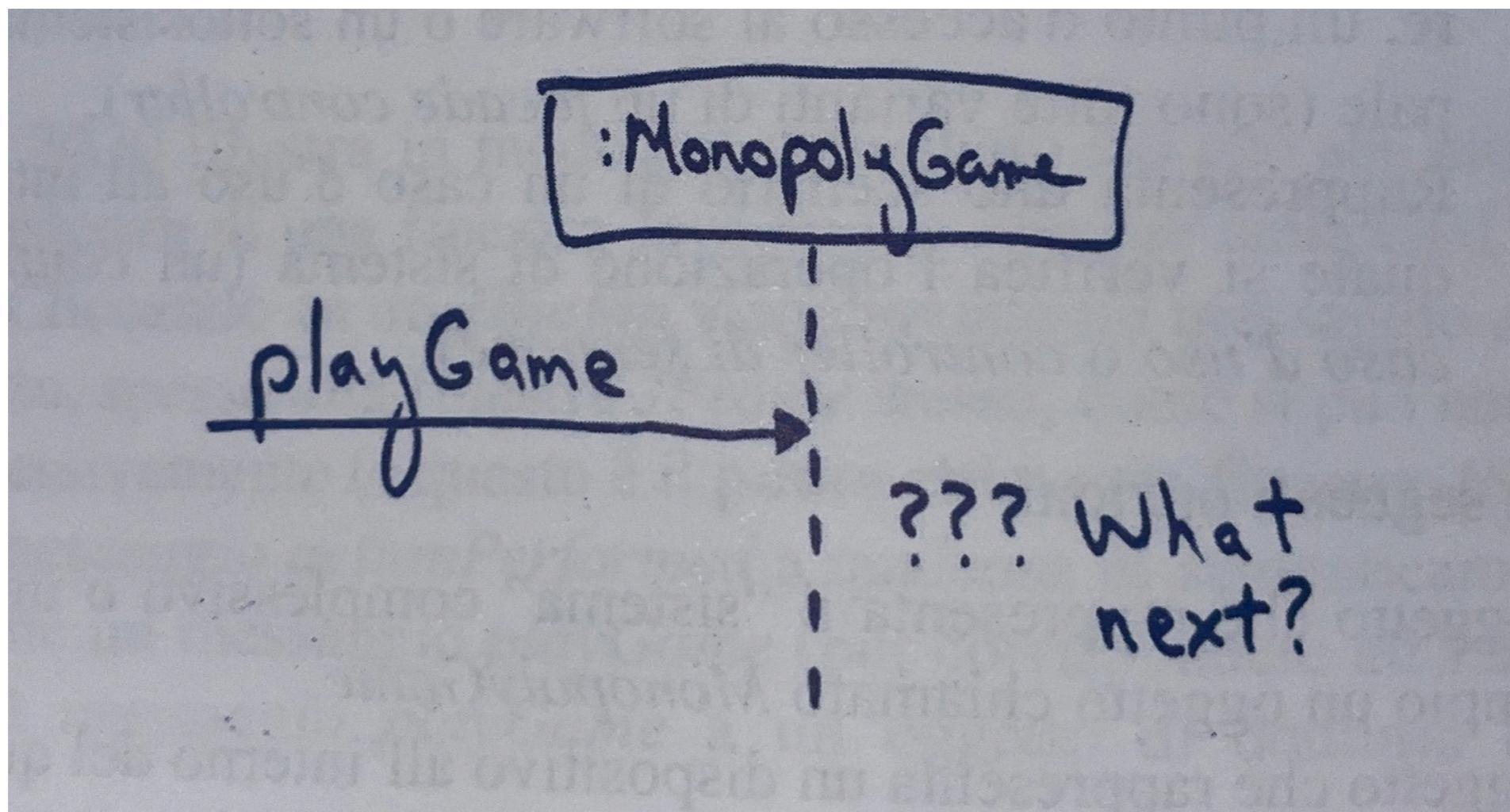
~ In questo caso l'opzione 1 è la più adatta, la più ragionevole perché ci sono poche operazioni di sistema

# Controller



# High Cohesion

- ~ Supponiamo di essere al punto di progettazione del pattern precedente



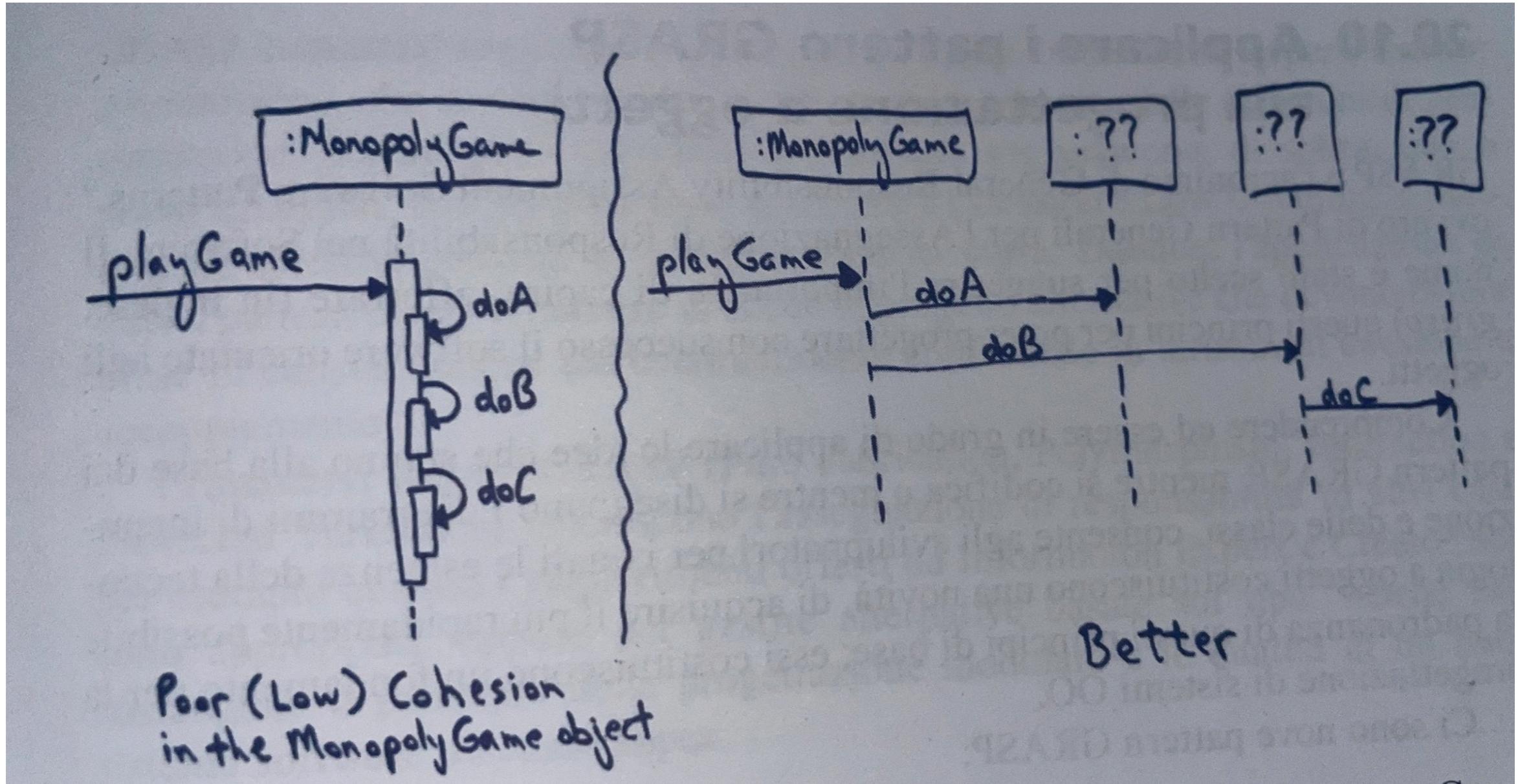
# High Cohesion

- ~ In base alla decisione suggerita dal controller la domanda è cosa fare dopo?
- ~ Ci sono due possibilità
  - s. L'oggetto MonopolyGame esegue tutto il lavoro
  - s. delega e coordina il lavoro per la richiesta play games
- ~ Il punto chiave in questo momento è il concetto di **coesione**

# High Cohesion

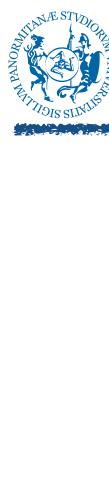
- ~ La **coesione** è una misura di quanto sono correlate le operazioni di un elemento software e da un punto di vista funzionale
- ~ una misura di quanto lavoro sta eseguendo un elemento software
- ~ Per esempio: un oggetto con 100 metodi e 2000 linee di codice fa più lavoro di un oggetto con 10 metodi e 200 linee di codice
  - se però i 100 metodi hanno molte aree di responsabilità diverse allora l'oggetto ha meno coesione funzionale
  - dobbiamo guardare sia la quantità di codice che la correlazione al codice

# High Cohesion



Poor (Low) Cohesion  
in the `:MonopolyGame` object

Better



# High Cohesion

Nome	High Cohesion
Problema	Come mantenere gli oggetti focalizzati comprensibili e gestibili e come effetto collaterale sostenere l'high cohesion?
Soluzione (consiglio)	Assegna la responsabilità in modo tale che la coesione rimanga alta usa questo principio per valutare le alternative

# High Cohesion

- ~ Nella figura di prima la versione di sinistra ha una coesione peggiore di quella della versione sulla destra
  - la versione sulla sinistra fa eseguire tutto il lavoro all'oggetto MonopolyGame anziché delegare e distribuire il lavoro tra gli oggetti
  - questo porta al principio di high cohesion che va utilizzato per valutare scelte di progetto diverse
  - se tutte le altre cose sono equivalenti va preferito un progetto con una coesione più elevata



# In conclusione

- ~ Comprendere ed essere in grado di applicare le idee che stanno alla base dei pattern (abbiamo visto solo i GRASP), sia mentre si codifica che mentre si progettano i diagrammi di interazione e delle classi, consente agli sviluppatori, per i quali le esigenze della tecnologia ad oggetti costituiscono una novità, di acquisire il più rapidamente possibile la padronanza di questi principi di base.

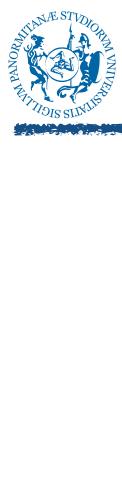
# Design pattern GoF

- ~ I GoF sono stati descritti per la prima volta nel libro design patterns
  - è un lavoro importante ed estremamente diffuso
  - 23 pattern utili durante la progettazione a oggetti
  - ciascun design pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente
  - i design pattern GoF sono classificati in base al loro scopo che può essere creazionale, strutturale o comportamentale ed in base a quello che si può chiamare raggio di azione

# Design pattern GoF

## ~ Classificazione 1:

- Creazionali: i pattern di questo tipo sono relativi alle operazioni di creazione di oggetti.
- Strutturali: sono utilizzati per definire la struttura del sistema in termini della composizione di classi ed oggetti. Si basano sui concetti OO di ereditarietà e polimorfismo.
- Comportamentali: permettono di modellare il comportamento del sistema definendo le responsabilità delle sue componenti e definendo le modalità di interazione.



# Design pattern GoF

## ~ Classificazione 2:

- Classi: pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate prevalentemente sul concetto di ereditarietà e sono quindi statiche (definite a tempo di compilazione).
- Oggetti: pattern che definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi più dinamiche.

# Catalogo dei pattern GOF

- ~ 1. Adapter
- ~ 2. Façade
- ~ 3. Composite
- ~ 4. Decorator
- ~ 5. Bridge
- ~ 6. Singleton
- ~ 7. Proxy
- ~ 8. Flyweight
- ~ 9. Strategy
- ~ 10. State
- ~ 11. Command
- ~ 12. Observer
- ~ 13. Memento
- ~ 14. Interpreter
- ~ 15. Iterator
- ~ 16. Visitor
- ~ 17. Mediator
- ~ 18. Template Method
- ~ 19. Chain of Responsibility
- ~ 20. Builder
- ~ 21. Prototype
- ~ 22. Factory Method
- ~ 23. Abstract Factory

# Classificazione completa

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggio d'azione	Classi	<b>Factory Method</b>	<b>Adapter</b> (class)	Interpreter <b>Template Method</b>
	Oggetti	<b>Abstract Factory</b> Builder Prototype Singleton	<b>Adapter</b> (object) Bridge <b>Composite</b> <b>Decorator</b> Facade Flyweight Proxy	Chain of responsibility Iterator Mediator Memento <b>Observer</b> State <b>Strategy</b> Visitor