

Sviluppo delle applicazioni software

V. Bono

Capitolo 17

**GRASP: progettazione di oggetti
con responsabilità**

“Capire le responsabilita` e` fondamentale per una buona programmazione ad oggetti”

Martin Fowler

Progettazione Object-Oriented (OODesign)

- Identificare i requisiti: *fatto*
- Creare il modello di dominio: *fatto*
- Aggiungere i metodi alle classi appropriate: *da fare*
- Definire i messaggi fra gli oggetti per soddisfare i requisiti: *da fare*


Non ovvio!!!! Usare i *pattern*...



cosa sono i ***Principi o Pattern***

- Coppia problema-soluzione nota
 - Hanno un nome
 - Possono essere spiegati e applicati a situazioni nuove, con *pros* e *cons*
 - Modelli UML statico e dinamico portati avanti insieme (ma UML e` solo un linguaggio visuale per la progettazione, non una metodologia)
 - Cominciamo ad usarli quando abbiamo:
 - I casi d'uso (con il testo), i contratti delle operazioni, il glossario
 - I diagrammi di sequenza di sistema
 - Il modello di dominio
 - Le specifiche supplementari
- [Tutto utile, ma non necessario nell'UP...]

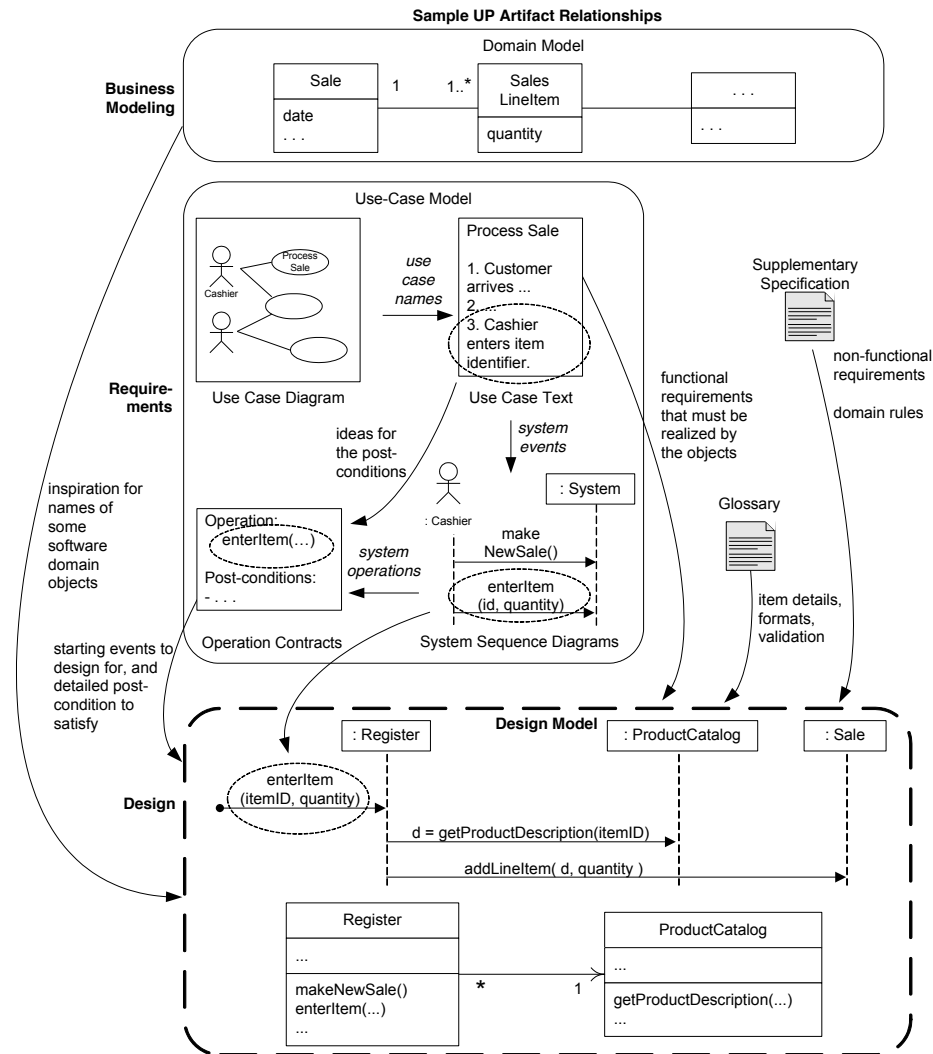
RDD

- Progettazione guidata dalle responsabilità
 (Responsability-Driven Development)
- Siamo sempre nell'UP, in cui le iterazioni riguardano tutte le fasi di sviluppo del software (dai requisiti al testing). Ogni iterazione raffina il “prototipo” della fase precedente: per evitare la “mentalità a cascata”, occorre passare presto alla codifica...

Output della RDD nella prima iterazione

- Diagrammi UML di interazione per le parti piu' difficili
- Specializzazione dei class diagram
- Abbozzi dell'interfaccia utente
- Modelli per i dati persistenti (uso dei *framework*, ne parleremo brevemente se avremo tempo)
- ...

Fig. 17.1 (POS NextGen)



RDD

- La responsabilita` e` un'astrazione del comportamento degli oggetti
- Responsabilita` di *fare*:
 - Creare un oggetto, fare un calcolo
 - Iniziare un'azione in altri oggetti
 - Coordinare attivita` di altri oggetti
- Responsabilita` di *conoscere*:
 - Conoscere i dati privati
 - Conoscere oggetti correlati
 - Conoscere cose calcolabili e/o derivabili

RDD

In POS NextGen:

- un oggetto *Sale* e` responsabile della creazione di oggetti *SalesLineItem* (fare)
- un oggetto *Sale* e` responsabile di conoscere il suo totale (conoscere)

Le responsabilita` sono assegnate alle classi.

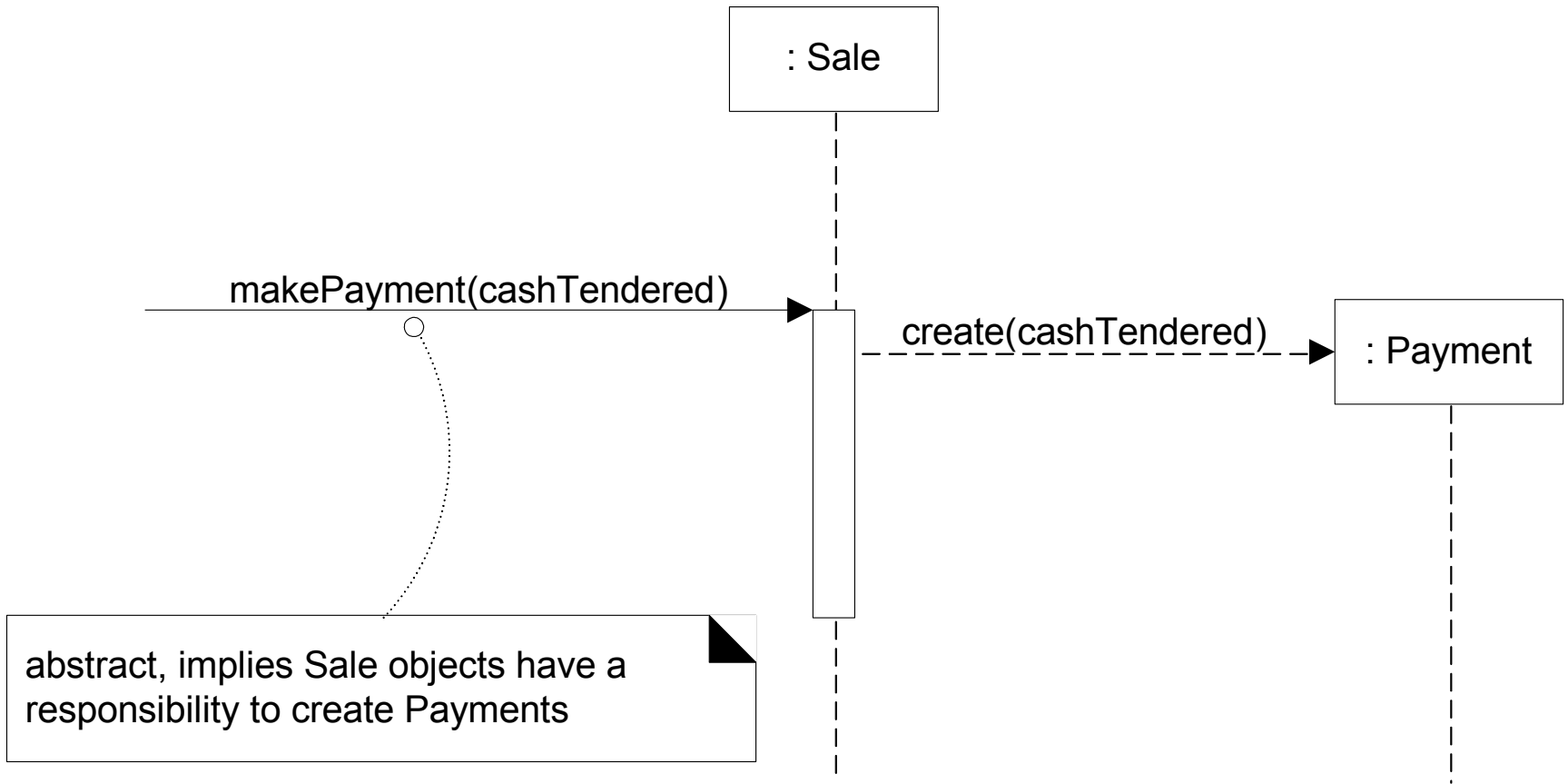
RDD

- Modello di dominio puo` ispirare le responsabilita` di conoscere: attributi e associazioni
- Una responsabilita` puo` essere condivisa da molte classi e metodi (es. gestione DB per dati persistenti)
- Responsabilita` diversa da metodo, anche se metodo soddisfa responsabilita`
- *Collaborazione* (es. metodo *getTotal* di *Sale* chiama metodo *getSubTotal* di *SalesLineItem* per ogni linea)

RDD e pattern

- Decisioni sulla responsabilita` durante la progettazione, ma anche durante la codifica
- Uso dei diagrammi UML di interazione (vedi Fig. 17.2) per visualizzare le responsabilita` durante la progettazione
- Pero` i pattern, specialmente i pattern *GRASP* di Larman, possono essere applicati direttamente alla codifica (ricordiamoci sempre che siamo in UP...)

Fig. 17.2



Pattern *GRASP*



- General Responsibility Assignment Software
- Danno un nome a e descrivono alcuni principi di base per assegnare le responsabilità alle classi
- E i famosi pattern *GoF*? Più “schemi di progettazione avanzata” che “principi”, infatti nel libro vengono introdotti dalla seconda iterazione in poi, ma noi ne vedremo qualcuno
- “*Cio` che per una persona rappresenta un pattern, per un'altra puo` costituire un blocco di costruzione elementare*” [Gang of Four]
- Scopi: accoppiamento basso, maggiore chiarezza, incapsulamento, quindi riusabilita`



I 9 pattern GRASP

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

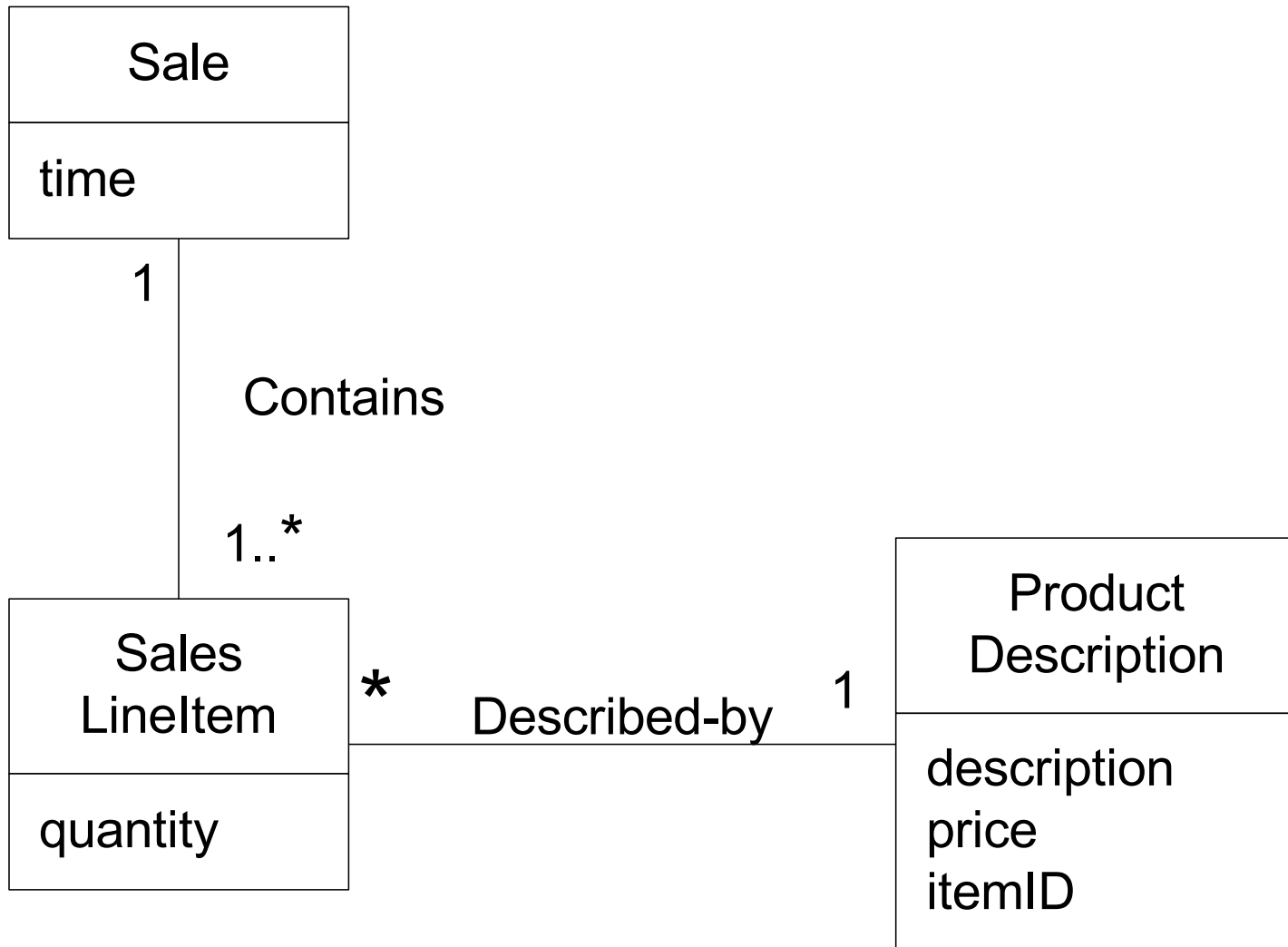
Creator

- Problema: Chi deve essere responsabile della creazione di un'istanza?
- Soluzione: Assegna a classe B la responsabilita` di creare istanza di classe A se (piu' cond. sono vere meglio e`):
 -  – B “contiene” o aggrega con una composizione di oggetti A
 - B “registra” A
 - B utilizza strettamente A
 -  – B “sa” i dati per l'inizializzazione degli oggetti di A (B e` un *Expert* rispetto a creazione di A)

Creator per POS NextGen (I)

- Chi crea istanze di *SalesLineItem*?
Usiamo il modello di dominio parziale di
Fig. 17.12...

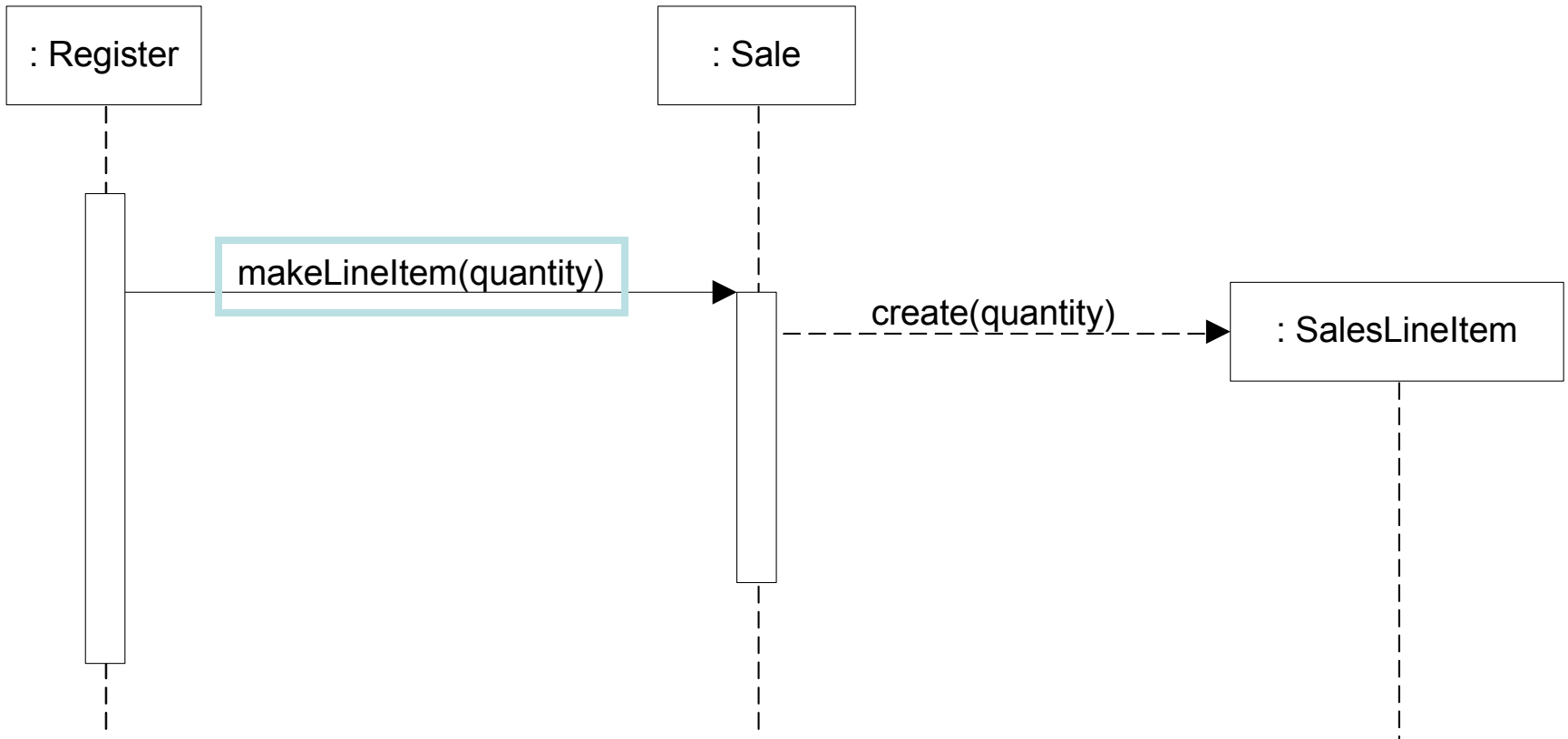
Fig. 17.12



Creator per POS NextGen (II)

- E' *Sale!* Vediamo Fig. 17.13: in *Sale* serve un metodo makeLineItem...
- Con il sequence di Fig. 17.13 si puo' arricchire il DCD (Design Class Diagram)

Fig. 17.13



Discussione su Creator

- Trovare creatore che abbia veramente bisogno di essere collegato all'oggetto creato (→low coupling)
- Nell'esempio precedente utilizzato composto/contenitore
- Si devono usare classi di supporto (ovvero pattern non-GRASP piu' complicati come le *Factory*) se la creazione puo` essere in alternativa a "riciclo" o se una proprieta` esterna condiziona la scelta della classe creatrice tra un insieme di classi simili
- Creator correlato a *Low Coupling*

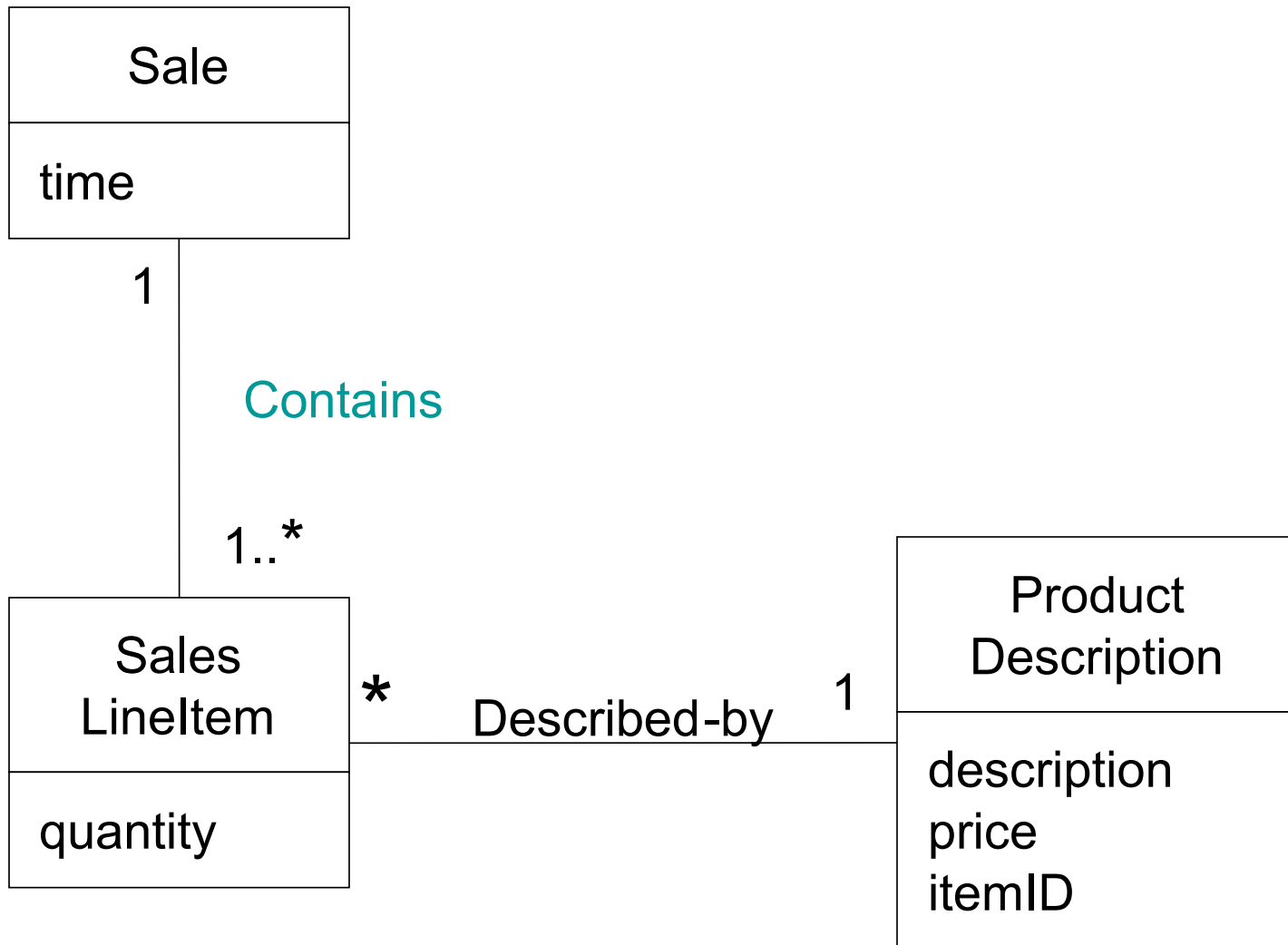
Information Expert (o Expert)

- Problema: Come scegliere le interazioni fra gli oggetti in modo che le responsabilita` siano ben distribuite tra le classi?
- Soluzione: da` una responsabilita` a chi conosce le informazioni necessarie per soddisfare le responsabilita`

Expert per POS NextGen (I)

- “Chi e` responsabile di conoscere il totale complessivo di una vendita”? Prima, avere chiara la responsabilita`
- Poi analizzare il modello di dominio (Fig. 17.14) da cui si capisce che *Sale* ha tutte le info necessarie...

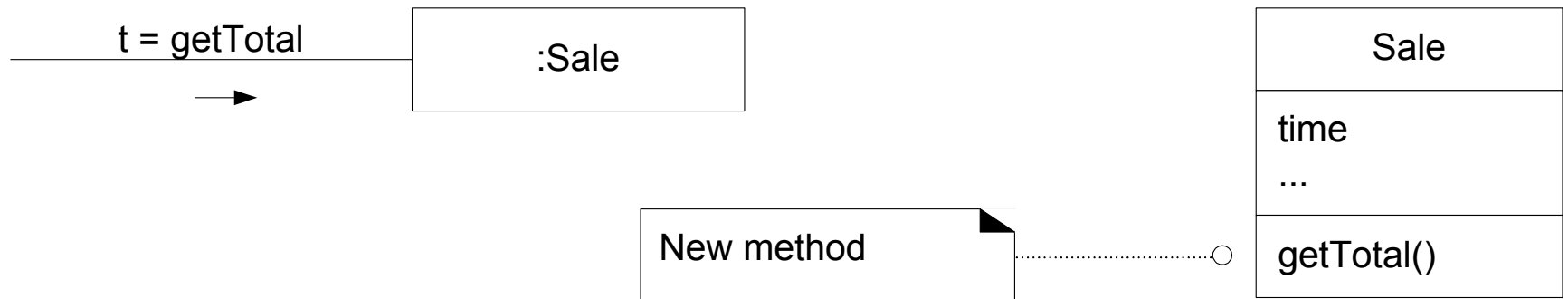
Fig. 17.14



Expert per POS NextGen (II)

- ... *Sale* deve avere metodo *getTotal()* (Fig. 17.15)
- Dal modello di dominio (mdd) al *modello di progetto* (mdp): un class diagram con l'indicazione dei metodi (*salto rappresentazionale basso*)
- NB Si puo` usare sia il mdd che il mdp per applicare Expert

Fig. 17.15



Expert per POS NextGen (III)

- *Sale*: conosce totale delle vendite
- *SalesLineItem*: conosce totale della riga di vendita di un item
- *ProductDescription*: conosce il prezzo del prodotto

→ Si ottengono in due passi i diagrammi di Figg. 17.16 e 17.17

Fig. 17.16

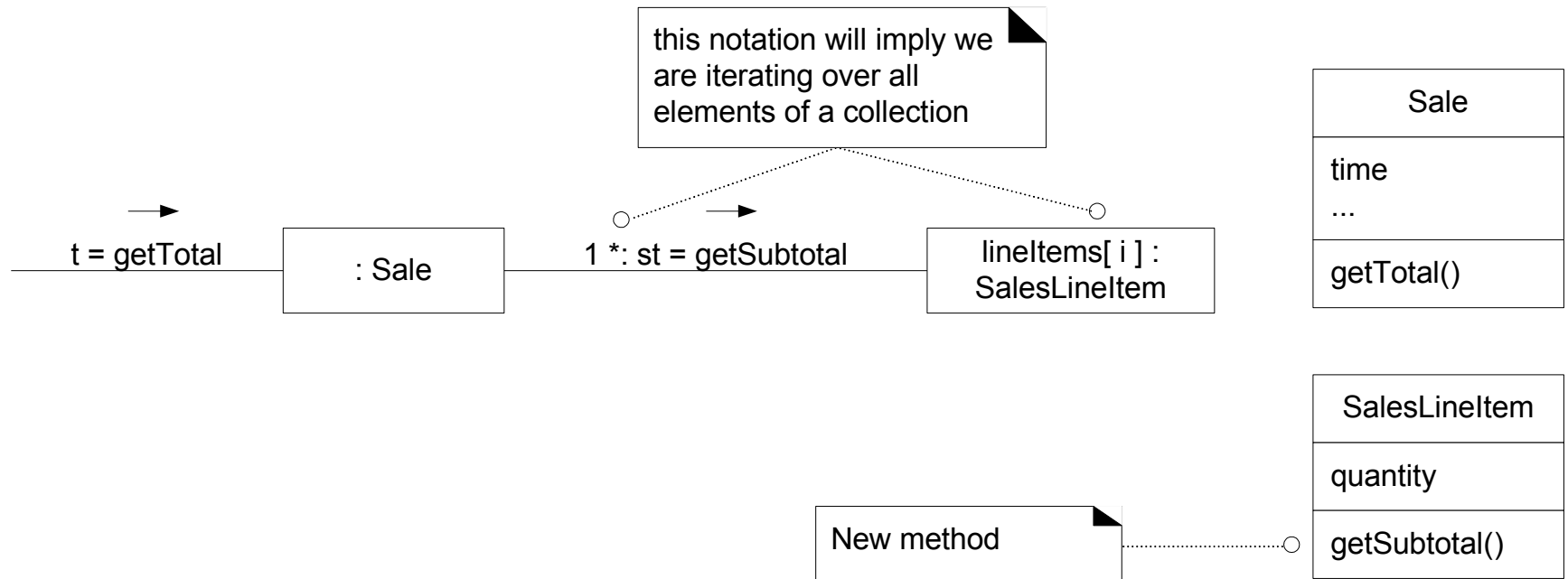
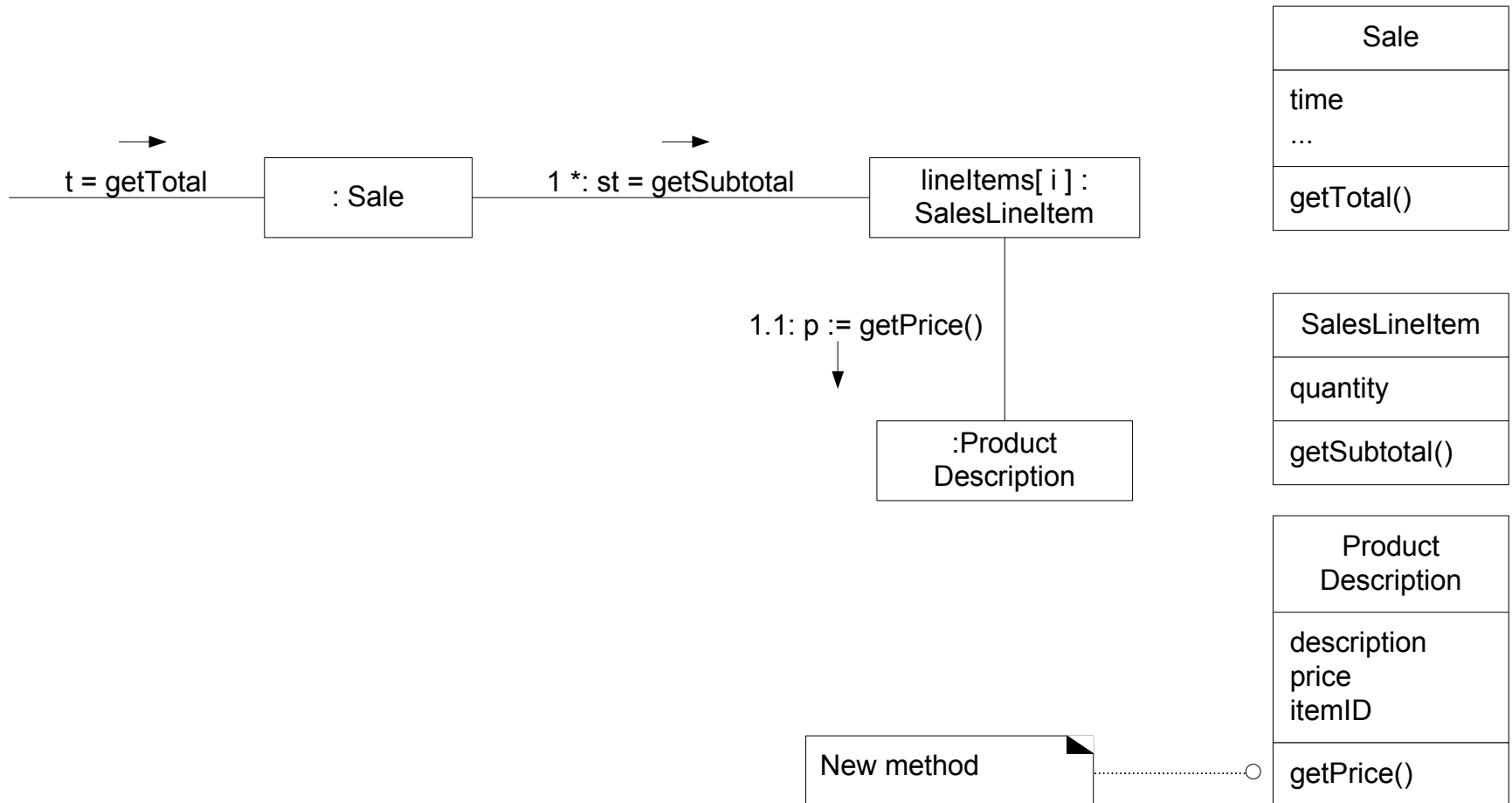


Fig. 17.17



Discussione su Expert (I)

- Si individuano informazioni parziali di cui classi diverse sono “esperte”: queste classi collaborano insieme per realizzare l'obiettivo → info distribuite, classi piu' leggere, senza perderne l'incapsulamento
- “*Do it myself*” [Peter Coad]: gli oggetti software, a differenza di quelli reali, hanno la responsabilita` di compiere delle “azioni” sulle cose che conoscono

Discussione su Expert (II)

- Chi salva un oggetto Sale in un DB?

Discussione su Expert (III)

- *Sale* (e così' tutte le classi salvano i loro oggetti)?!? NO!
- *Sale* avrebbe problemi di (1) coesione, (2) accoppiamento e (3) duplicazione: (1) non si occupa più' solo della logica applicativa (meno coesa); (2) accoppiata a classi di sistema e non solo a classi del mdd; (3) applicando la stessa idea a più' classi duplico la logica del DB
- Principio architetturale di base: separare le diverse logiche in sottosistemi separati → Expert non va sempre bene!



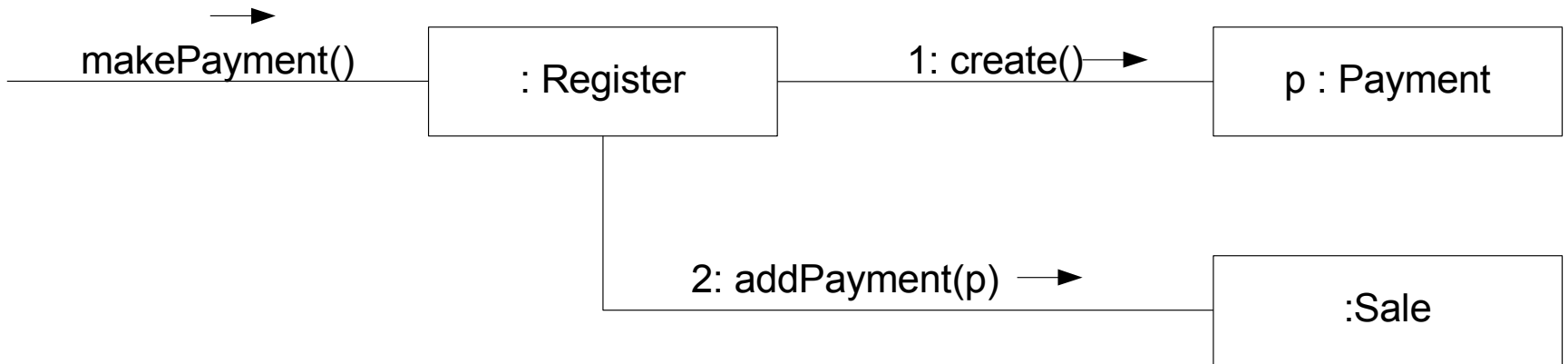
Low Coupling

- Problema: Come ottenere una dipendenza bassa, un impatto dei cambiamenti basso e un riuso maggiore?
- *Accoppiamento (coupling)* = misura di dipendenza da altri elementi (via connessioni/conoscenza)
- Soluzione: Evitare accoppiamento alto tra classi (se si puo`).
- NB Si usa per valutare alternative

Low Coupling per POS NextGen (I)

- Classi *Payment*, *Register*, *Sale*
- Con il Creator, si sceglie Register come creatore di Payment (Fig. 17.18): suggerito dalle responsabilità nel “mondo reale”
- Uso metodo *addPayment(p)* per comunicare con *Sale*

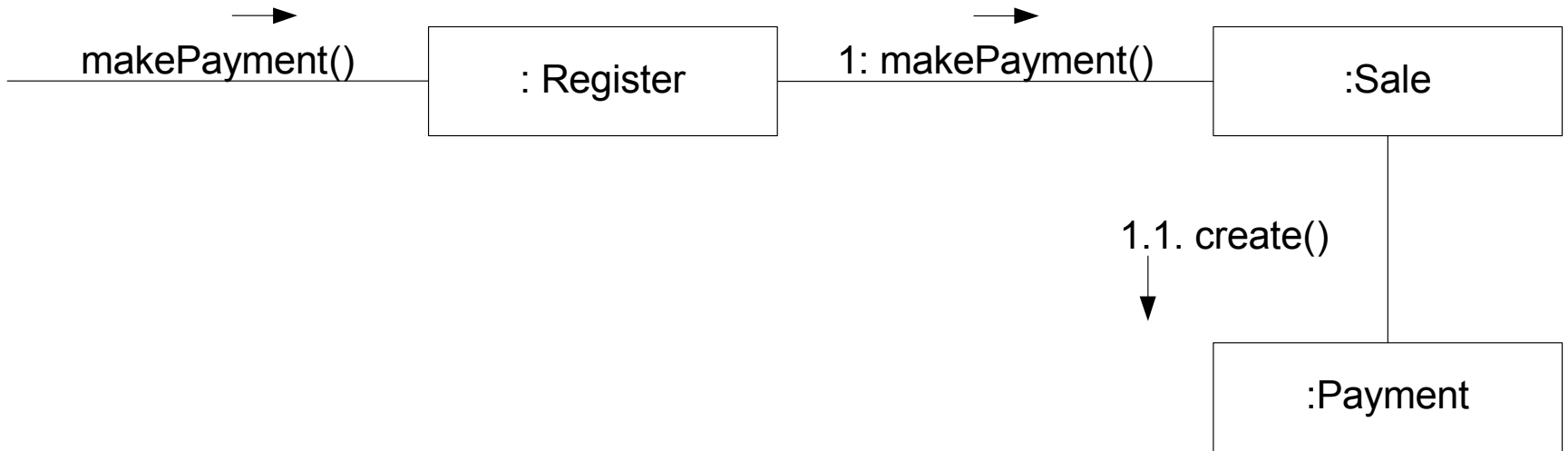
Fig. 17.18



Low Coupling per POS NextGen (II)

- Soluzione alternativa: *Sale* deve comunque conoscere *Payment*, per cui l'accoppiamento tra *Payment* e *Register* e' inutile
- Meglio il progetto di Fig. 17.19 dal punto di vista del Low Coupling

Fig. 17.19



Discussione su **Low Coupling** (I)

- E` principio di valutazione da utilizzare in parallelo ad altri pattern

Discussione su Low Coupling (II)

- Accoppiamento tra typeX verso typeY:
 - typeX ha un attributo typeY
 - oggetto typeX chiama servizi di oggetto typeY
 - typeX ha metodo che possiede un elemento di typeY (variabili locali, parametri, return)
 - typeX e` sottoclasse (diretta o no) di typeY (FORTE)
 - typeX implementa interfaccia typeY.
- Assegnare resp. in modo che non “aumenti troppo” l'accoppiamento (es. classi generiche devono avere accoppiamenti bassi), ma ci deve essere comunque collaborazione fra gli oggetti senno` sistema non o-o! **Attenzione in particolare all'accoppiamento con elementi instabili** (es. nell'interfaccia) **o tecnici** (es. db)



Controller (I)

- Problema: qual e' il primo oggetto oltre lo strato UI che riceve e coordina (controlla) un'*operazione di sistema*?
- Le *operazioni di sistema* sono gli eventi di input principali (es. evento che segue il premere del pulsante "End Sale")
- Soluzione: i Controller sono classi che ricevono/gestiscono i messaggi legati alle operazioni di sistema (coordinano, ma non fanno molto di piu', NON sono classi di dominio)



Controller (II)

Assegnare la responsabilita` di Controller a una classe che rappresenta:

- O il “sistema” complessivo, un dispositivo nel quale viene eseguito il sw o un sottosistema principale (*façade controller*)
- O uno scenario di un caso d’uso all’interno di cui si verifica l’evento, <UseCaseName>Handler o <UseCaseName>Coordinator o <UseCaseName>Session (*controller di caso d’uso*)



Opzione: un controller per ogni scenario

- Si utilizzi la stessa classe Controller per tutti gli eventi di sistema nello stesso scenario
- Informalmente una sessione e` un'istanza di conversazione con un attore
- Le classi “finestre”, “viste” e “documenti” non sono Controller, ma delegano la gestione degli eventi a una classe Controller
- Posso riusare la logica con diverse UI o in modo *batch (off-line)*
- Posso controllare che gli eventi avvengano in un ordine prestabilito (es. *makePayment* dopo *endSale*)



Controller per POS NextGen (I)

- Sistema stesso come una classe (Fig. 17.20): contiene le varie operazioni di sistema. Va bene a livello di analisi, non e` detto che questa idea sia propagata a livello del sw

Fig. 17.20

System
endSale() enterItem() makeNewSale() makePayment() ...

Controller per POS NextGen (II)

- Nel software chi deve controllare *enterItem* e *endSale*?
 - Un oggetto di classe façade controller:
Register (dispositivo con sw), *POSSystem*?
 - Un oggetto di classe controller di caso d'uso:
ProcessSalehandler, *ProcessSaleSession*?

Fig. 17.21: quale oggetto deve essere Controller per *enterItem*?

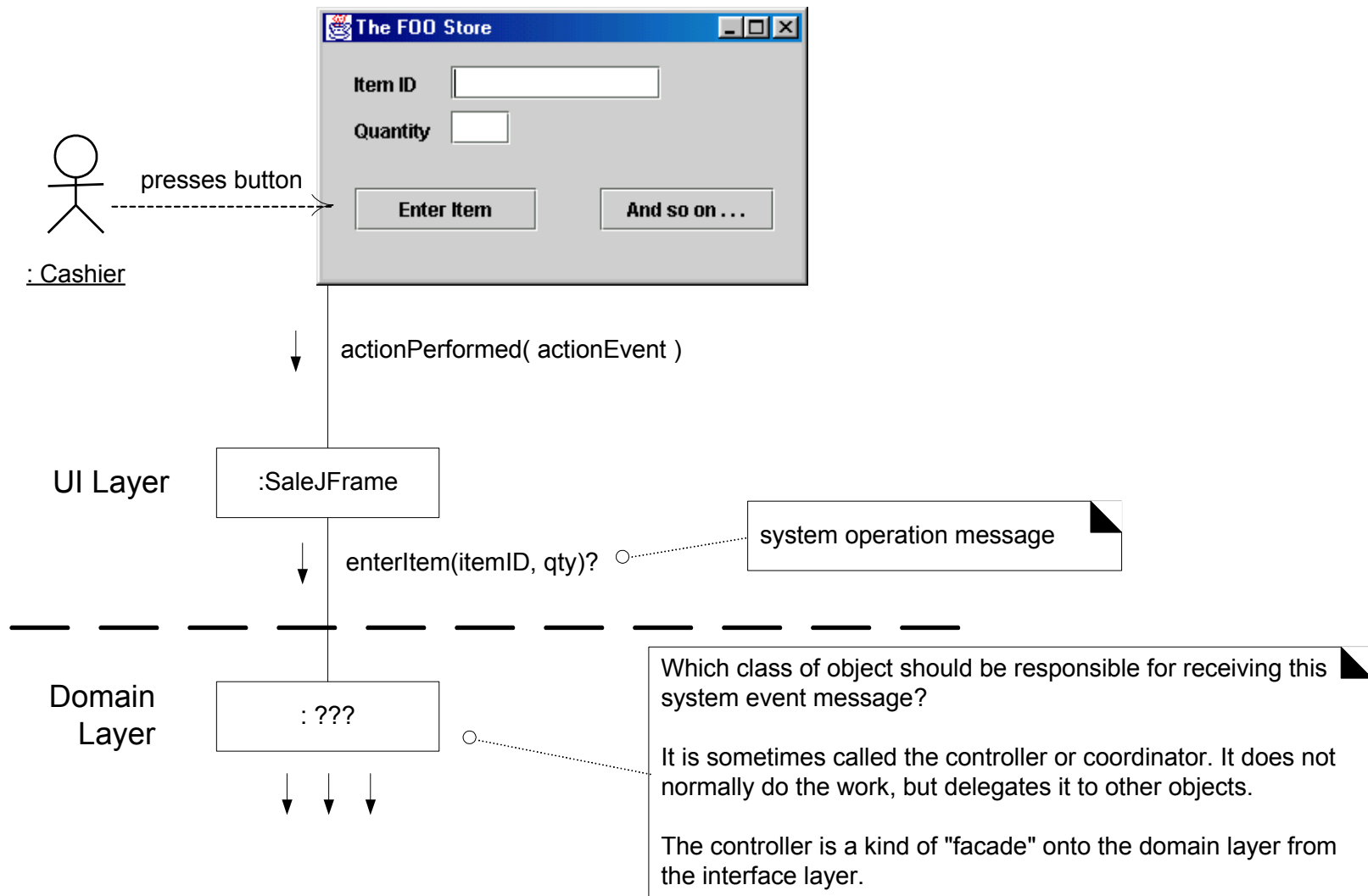
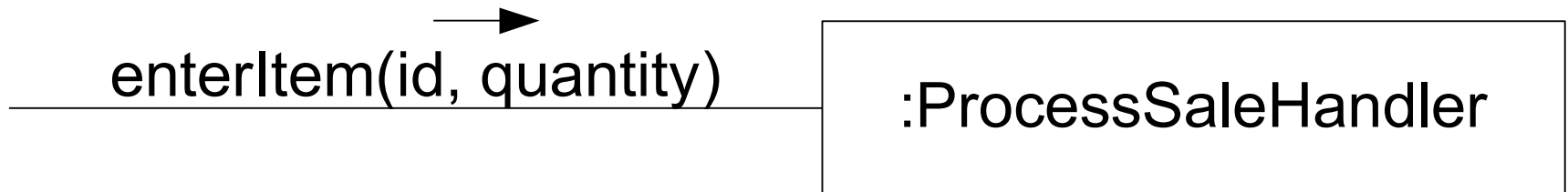
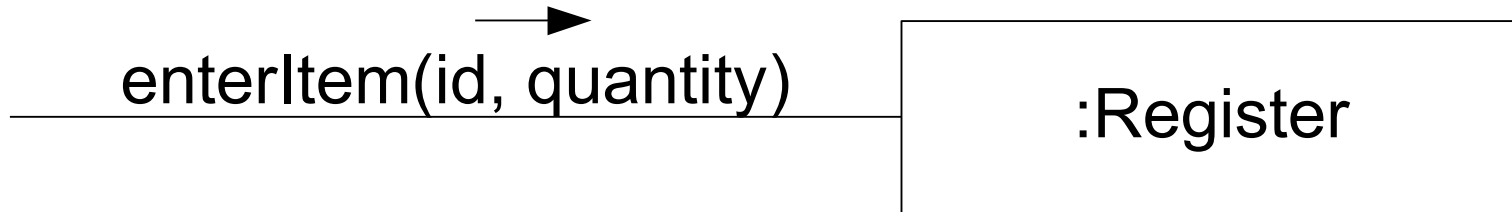


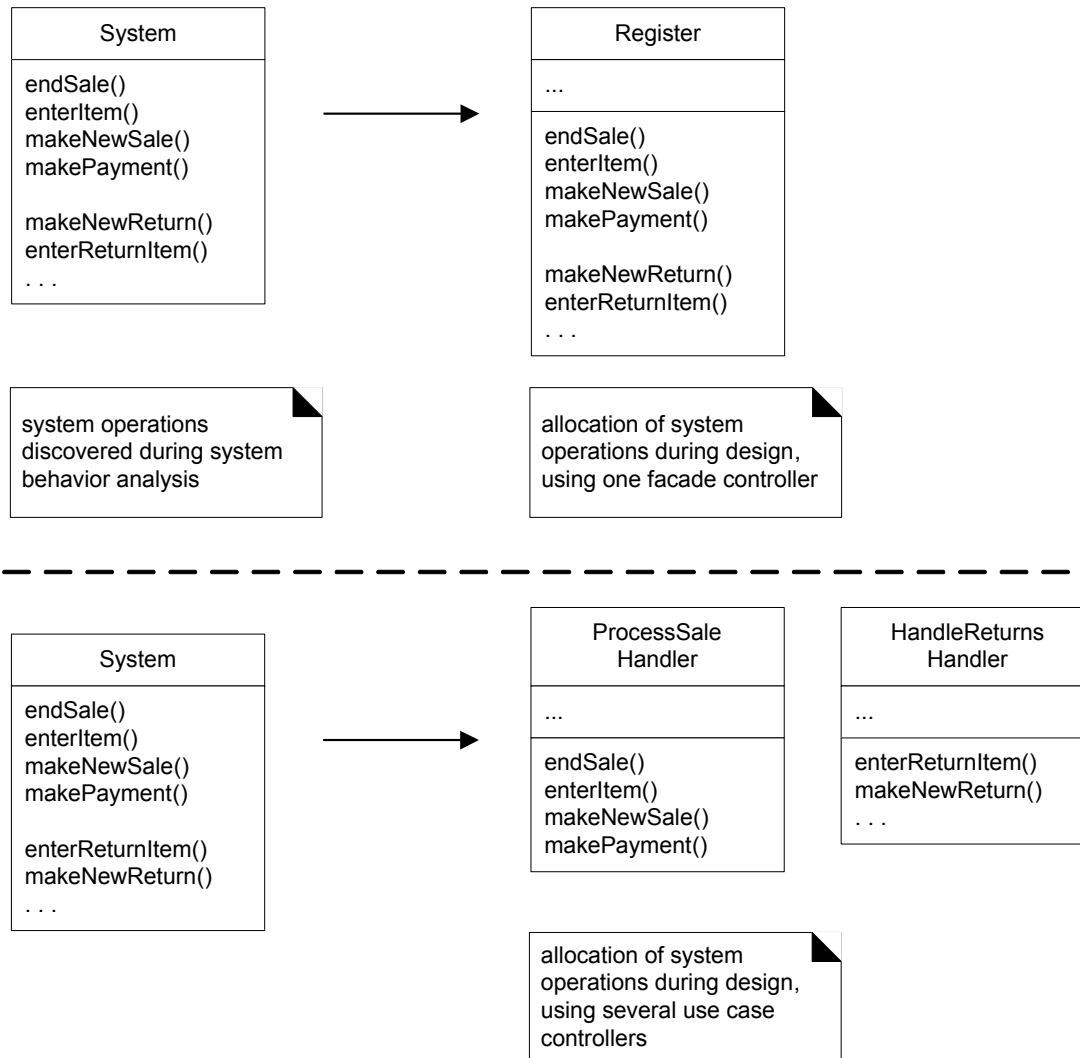
Fig. 17.22: due possibilita`



Controller per POS NextGen (II)

- La scelta fra le possibilità della Fig. 17.22 è influenzata da altri fattori, che esamineremo
- Fig. 17.23 riflette le due possibilità sul class diagram

Fig. 17.23



Discussione su Controller (I)

- Utile studiare esempi di codice
- Pattern di delega e di decoupling tra logica dell'interfaccia e logica del sistema: “facciata”
- Gestore di eventi esterni (spesso generati da attori umani via GUI o da sensori)
- Qualsiasi sia la scelta (façade controller, use case controller) il controller serve!

Fig. 17.24: lo strato UI non deve gestire gli eventi di sistema

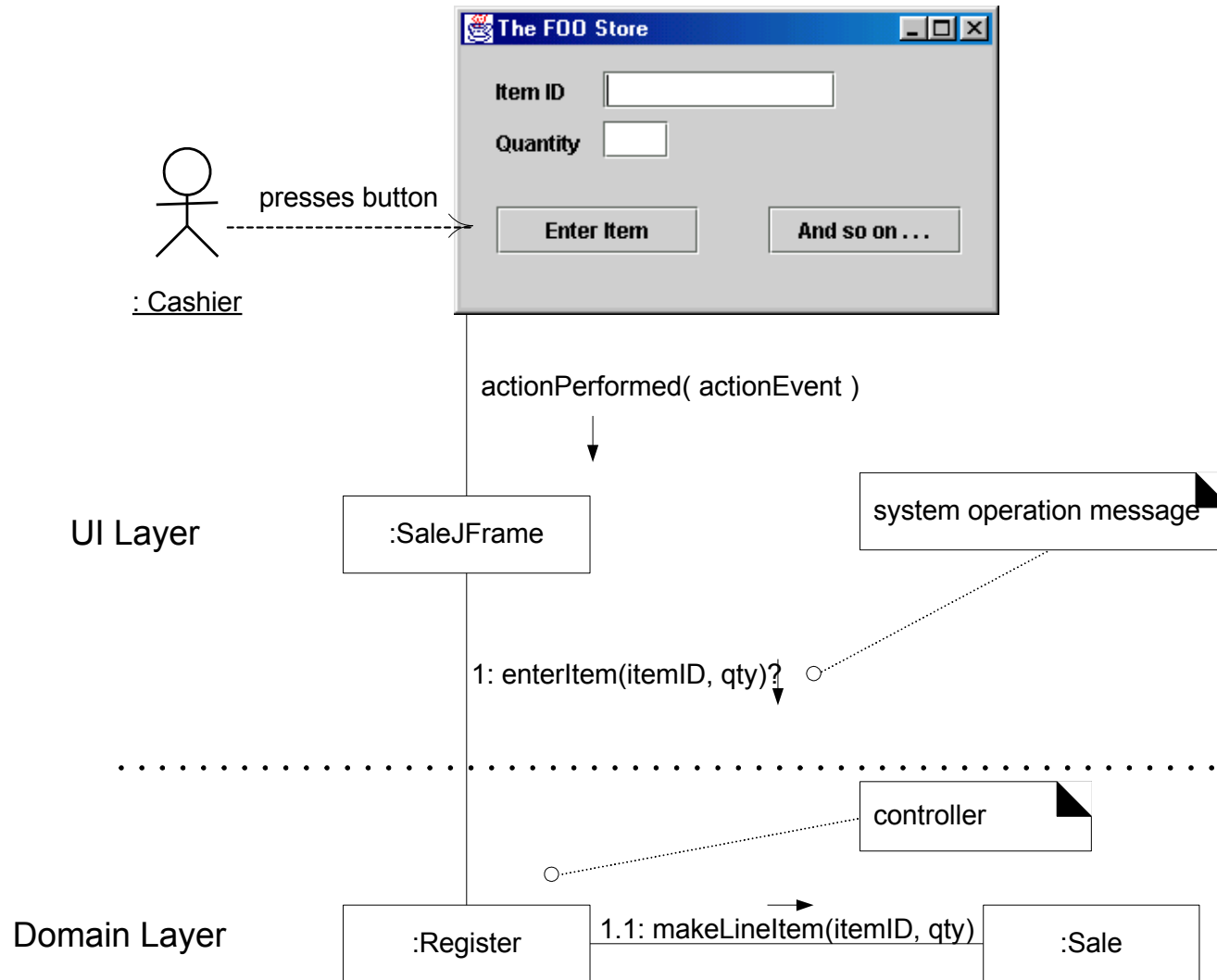
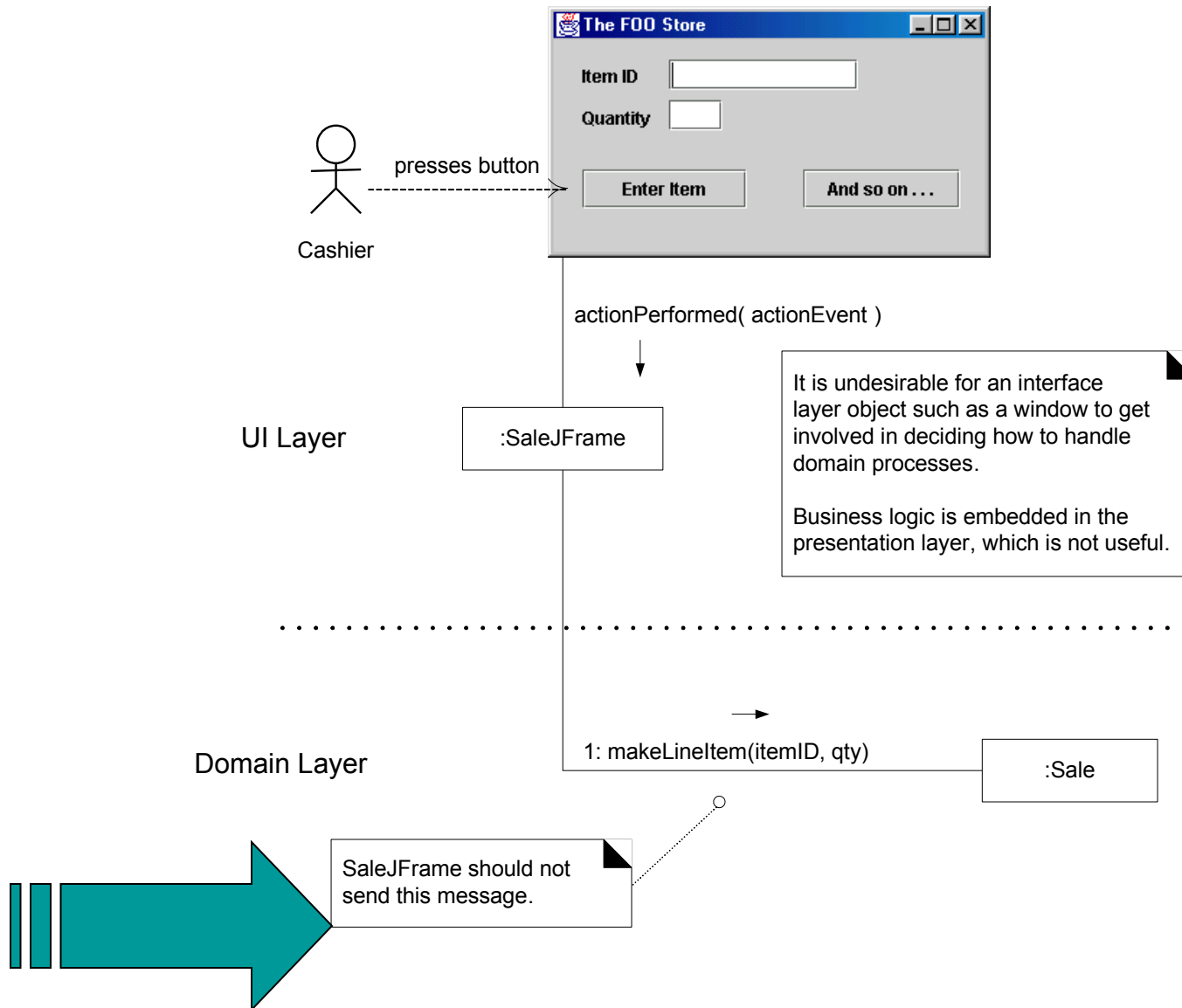


Fig. 17.25: soluzione meno opportuna



Discussione su Controller (II)

- Scelgo un *façade*: se non ci sono troppi eventi di sistema o se la UI non riesce a discriminare tra gli eventi di sistema
- Scelgo uno *use case*: se un *façade* fosse troppo “gonfio” di responsabilità (coesione bassa e accoppiamento alto), molti eventi di sistema diversi in processi diversi

Discussione su **Controller** (III)

- GoF pattern *Command*: specializza l'idea del Controller, ad es. nel caso di sistema per la gestione di messaggi o server che riceve messaggi da processi diversi c'è un oggetto Command che lo rappresenta e lo gestisce
- Pattern GRASP *Pure Fabrication*: creazione arbitraria del progettista, non è ispirata dal modello del dominio. Così è il Controller

High Cohesion

- Problema: Come mantenere gli oggetti focalizzati, comprensibili e gestibili, quindi sostenere Low Coupling come *side-effect*?
- Cohesion = misura delle correlazioni e concentrazioni di responsabilità di un elemento (classe, sottosistema). Responsabilità altamente correlate e quantità di lavoro non eccessiva → alta coesione
- Soluzione: assegnare resp. in modo che la coesione rimanga alta

High Cohesion per POS NextGen (I)

- Riprendiamo l'esempio di Low Coupling (Figg. 17.26 e 17.27)
- Assumiamo di creare istanza di *Payment* (in contanti) e associarla alla *Sale* corrispondente. Quale classe e' responsabile di cio'?
- *Register*? (Fig. 17.26) Puo' essere ok, ma se si aggiungono altre resp. essa perde in coesione
- *Sale*? (Fig. 17.26) Meglio anche dal punto di vista della coesione

Fig. 17.26: *Register crea Payment*

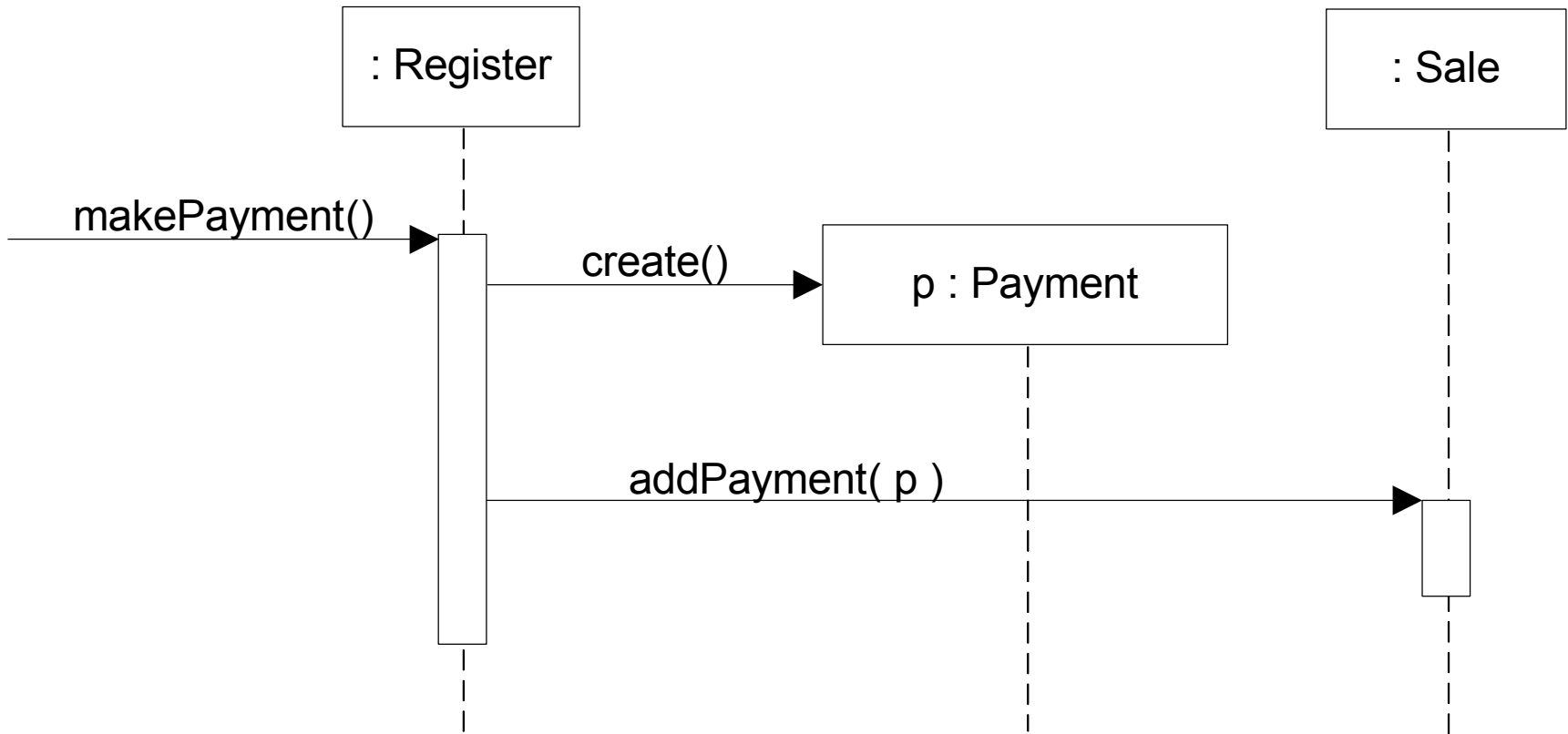
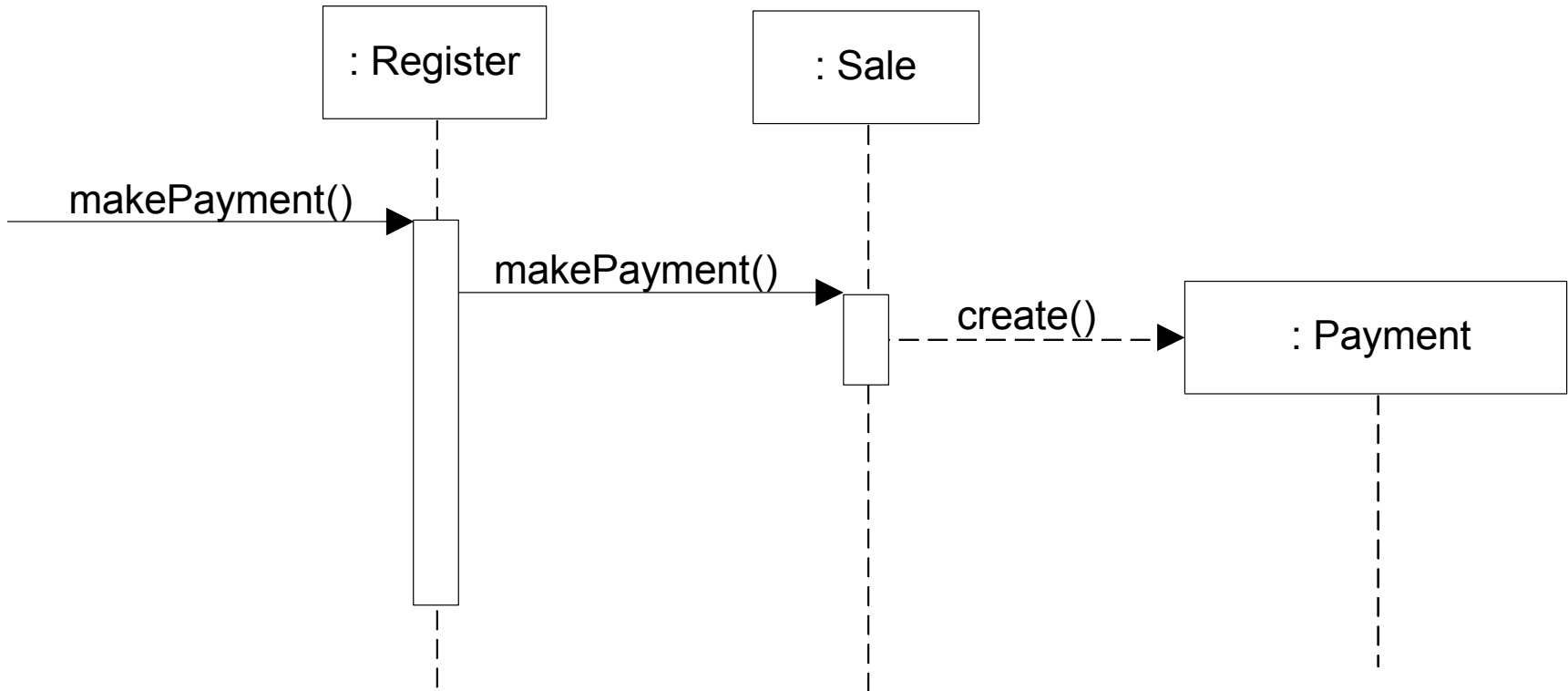


Fig. 17.27: *Sale crea Payment*



Discussione su High Cohesion (I)

- Low Coupling e High Cohesion molto importanti nelle decisioni di progetto: modularita` [Booch]
- Booch dice che esiste una coesione funzionale alta quando gli elementi di una componente “lavorano tutti insieme per fornire un comportamento ben circoscritto”

Discussione su High Cohesion (II)

- Coesione molto bassa: una classe e` sola responsabile di molte cose in aree funzionali molto diverse → piu' famiglie di classi
- Coesione bassa: una classe e` sola responsabile di molte cose in una sola area funzionale → piu' classi, piu' leggere
- **Coesione alta**: classe con responsabilita` leggere in una sola area che collabora con altre classi (di solito pochi metodi mirati)
- Coesione moderata: classe da sola con responsabilita` leggere in aree diverse, legate al concetto rappresentato dalla classe ma non l'una all'altra

Discussione su **High Cohesion** (III)

Quando e' meglio una coesione piu' bassa?

- Se per un determinato compito di programmazione/manutenzione ci vuole un esperto (es. raggruppare tutte le istruzioni SQL di un sistema in una sola classe)
- Per gli oggetti distribuiti lato sever (in caso di RMI o middleware affini): per avere migliori prestazioni