Illustrating Design Patterns

- POST example will be used to show how to apply the design patterns to a system under development
- Intentionally detailed, to exhaustively illustrate that responsibility assignment and the choice of object interactions can be rationally explained and learned
- Recall the steps in creating interaction diag.

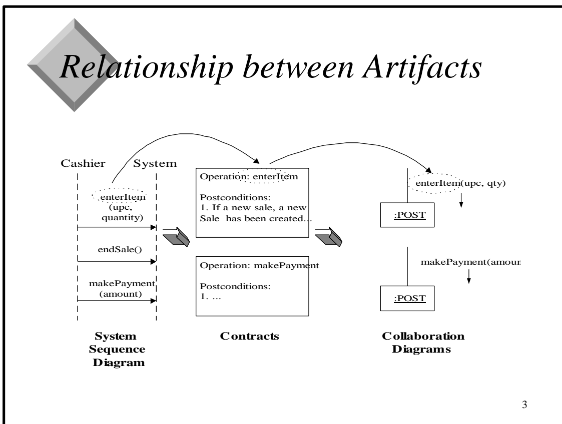
1

Steps to Make Collaboration Dia.

- Create a separate diagram for each system operation under development in current dev. Cycle
  - For each system message, make a diagram with it as the starting message
- Split complex diagrams into smaller ones
- Using the operation contract responsibilities, post-conditions, and use case descriptions, design interacting objects to fulfill the tasks. Apply GRASP & other patterns to improve design.

2

Relationship between Artifacts



3

Relationships Explained

- Use cases suggest the system events that are explicitly shown in system sequence diagrams
- An initial best guess at the effect of the system events is described in system operation contracts
- System events represent msgs that initiate inter-action diagrams, which show how objects interact to fulfill the required tasks
- Interaction diags involve msg interaction between objects of the conceptual model & other classes

4

Use Cases & System Events

- Two use cases in this development cycle:
  - Buy Items - version 1
    - enterItem()
    - endSale()
    - makePayment()
  - Start Up
    - startUp()
- For each event, create a collaboration diag. w/system event being the starting msg.

5

Using the Controller Pattern

- The POST class is chosen as the controller

enterItem() → :POST 1: ???() →

endSale() → :POST 1: ???() →

makePayment() → :POST 1: ???() →

startUp() → :POST 1: ???() →

6

Contracts

- Using the contract responsibilities, post-conditions and use cases, design system of interacting objs.
- Contract terms on p. 220
- Given this description: the collaboration diagram becomes:

enterItem(upc, qty) :POST 1: [new sale] create() :Sale

7

Lessons to Remember

- The spirit of iterative development is to capture a reasonable degree of info in each phase
- New types of objects are discovered by this process, and is to be expected
- Never feel that any one diagram captures all the detail needed in the system
- See *enterItem* collaboration diag. p. 223

8

Choosing a Controller

- The choices for *enterItem* are:
  - represent entire system (POST)
  - represent business org. (Store)
  - represent real world obj. (Cashier)
  - represent handler of a use case (BuyItemsHdlr)

Since only a few sys. ops. are present, we'll choose POST as the controller class.

by Controller enterItem(upc, qty) :POST

9

Making a New Sale

- Two of the contract post-conditions state:
  - If a new sale, then a *Sale* obj. is created (instance creation)
  - If a new *sale*, its associated with the POST (association formed)
- The GRASP creator pattern suggests _____ as the creator.
  - Answer: POST

10

What does Sale require?

- When a *Sale* is created, it must create a container to record all future *SalesLineItem* instances
- This collection will be maintained by *Sale*, so the Creator pattern implies that *Sale* should create it
- The container is represented by a Mulit-object in the collaboration diagram

11

Collaboration Diagram

by Controller by Creator enterItem(upc, qty) :POST 1: [new sale] create() :Sale by Creator 1.1: create() An empty collection that will eventually hold SalesLineItem instances. :SalesLineItem

12

Other enterItem Post-conds state:

- A *SalesLineItem* was created (instance creation)
- A *SalesLineItem* was associated with the *Sale* (association formed)
- *SalesLineItem.quantity* was set (attribute modification)
- *SalesLineItem* was associated with ProductSpecification, based on UPC match (associaction formed)

13

Who Creates SalesLineItem?

- A *Sale* contains a *SalesLineItem*
  - hence, by the Creator pattern, *Sale* is creator
- A *SalesLineItem* needs a quantity
  - so POST must pass it along to *Sale*, which in turn passes it as a parameter to *SalesLineItem*
    - Thus, by Creator, a *makeLineItem* msg is sent to *Sale*, which uses it to create *SalesLineItem* and then stores the new instance in its collection
    - quantity is a parameter to *makeLineItem*

14

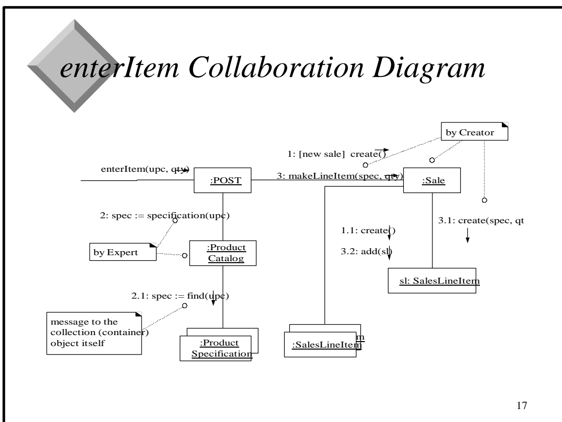
- *SalesLineItem* needs to be associated with *ProductSpecification*
- Who should be responsible for knowing a ProductSpecification, based on UPC match?
- Neither Creator or Controller apply here!
- Expert suggests the class that has the info
- Who knows all about ProductSpecification?
  - Answer: itself and ProductCatalog
  - Should ProductSpecification be creating objects?
    - Answer: NO, its been a passive object so far
    - ProductCatalog would be a better choice anyway.
- Who should send the specification msg to ProductCatalog asking for a ProductSpecification?

15

- Its reasonable to assume that POST and *Product-Catalog* were created during the *StartUp* use case
- So there is a permanent connection between POST and *ProductCatalog*
- Thus, POST can send the *specification* msg to *ProductCatalog*
- This requires POST to be able to "see" *Product-Catalog* (have visibility)
- We'll assume all of *ProductSpecification* can fit into memory. (Keep it simple)
  - Chapter 38 deals with persistence and DBs
- The *enterItem* collaboration diagram follows:

16

enterItem Collaboration Diagram



17

endSale Collaboration Diagram

- The *endSale* op. occurs when a Cashier presses a button indicating end of sale
- Contract appears on p. 229
- A collaboration diagram will be constructed to satisfy the post-conditions of *endSale*
- Who's responsible for the *endSale* system op?
  - The contract post-conditions state: `Sale.isComplete` has to be set to *true* (attribute modification)
  - Since it is not a Creator or Controller problem, Expert should dictate the responsibility

18

- By Expert, it should be *Sale* itself, since it owns and maintains the `isComplete` attribute
- Thus, POST will send a *becomeComplete* msg to *Sale* in order to set `isComplete` to *true*
  - Notice how pseudocode in the note conveys this:

```
becomeComplete()
{
  isComplete := true
}
```

endSale() :POST 1: becomeComplete() :Sale

by Controller by Expert

19

Calculating the Sale Total

- No single class currently knows the sales total
- Objects will need to collaborate to satisfy this
- As always, Expert should be considered, unless Creator or Controller apply (which they don't)
- So the *Sale* class has this responsibility
- *SalesLineItem* will supply subtotals to *Sale*
- *ProductSpecification* will supply price to *SalesLineItem*

20

Sales Total Collaboration Diag.

```
Sale--total()
{
  tot := 0
  for each SalesLineItem, sli
     tot := tot + sli.subtotal()
  return tot
}
```

1*: [for each] sli := next() :SalesLineItem

tot := total() :Sale 2: st := subtotal() sli: SalesLineItem

by Expert by Expert 2.1: pr := price

prodSpec: ProductSpecification

```
SalesLineItem--subtotal()
{
  return quantity * prodSpec.price()
}
```

21

makePayment Collaboration

- Occurs when Cashier enters the amount of cash tendered for payment
- Contract details, p. 234, Post-conditions include:
  - A *Payment* object is created (instance creation)
  - *Payment.amountTendered* is set (attribute modification)
  - *Payment* is associated with *Sale* (association formed)
  - The *Sale* is associated with the *Store*, to add it to the historical log (association formed)
- The Controller pattern applies to the POST class

22

Post-conditions Require

- A *Payment* is created
  - Who records, aggregates, uses, or contains *Payment*?
    - The POST
  - Who is the expert w/r/t initializing data?
    - The POST
  - However, *Sale* closely uses *Payment*, so its a candidate
  - Consider the Low Coupling/High Cohesion with both the POST and *Sale* classes
    - If *Sale* is selected, a lighter POST results, and POST doesn't need to know about Payment objs. Since its recorded indirectly via the *Sale*

23

Collaboration Diagram

- This diagram satisfies the contract post-conditions Payment is created, associated w/Sale, and amountTendered is set

by Controller by Creator, Low Coupling

makePayment(cashTendered) :POST 1: makePayment(cashTendered) :Sale

1.1: create(cashTendered)

:Payment

24

© 1998, Jeffrey Blessing

Logging the Sale

Sale
...
...
Logs-completed
*
1
Store
...
addSale(s : Sale)
...

Store is responsibile for knowing and adding completed Sales.

Acceptable in early development cycles if the Store has few responsibilities.

Sale
...
...
Logs-completed
*
1
SalesLedger
...
addSale(s : Sale)
...

SalesLedger is responsibile for knowing and adding completed Sales.

Suitable when the design grows and the Store becomes uncohesive.

25

Calculating the Balance

- Again, no class directly knows the balance, so objects will need to collaborate
- Using Expert, who knows the sale total, and cash tendered?
  - *Sale* and *Payment* are partial Experts
  - If Payment is responsible, it needs visibility to Sale. Since it doesn't currently know Sale, this increases Coupling (which is bad)
  - If Sale is responsible, it already has visibility of Payment, so no increase in Coupling (which is good)

26

Balance Collaboration Diagram

Sale--balance()
{ return pmt.amount() - total() }

bal := balance()
:Sale
1: amt := amount()
pmt: Payment
2: t := total()

27

StartUp Collaboration Diagram

- Should be done last in the develop. Cycle
- Must create an Initial Domain Object
- GUI or some other Actor (OS) starts the application
- POST starts up when a Manager powers on the machine and SW loads
- We'll say a create msg is sent to create the Initial Domain Object

28

Who Receives the create() msg?

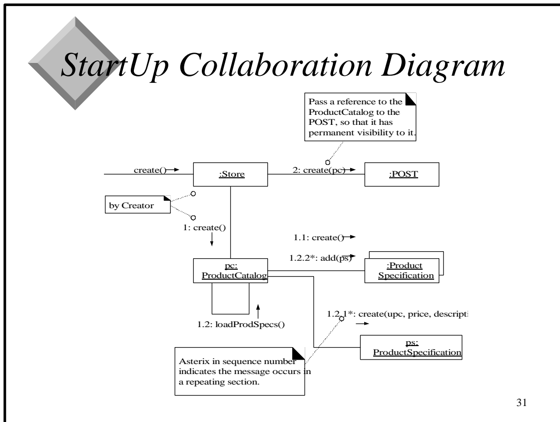
- Candidates are:
  - class representing the whole system
  - class representing the business organization
- Here we choose *Store* as the initial object
- Persistent objects: ProductSpecification
  - We'll assume objects are loaded on demand into memory since later development cycles deal with databases

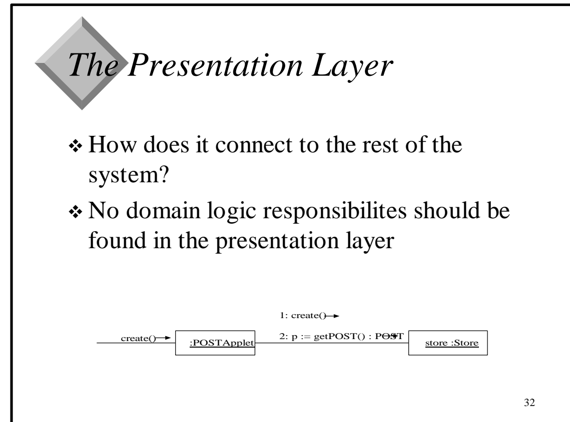
29

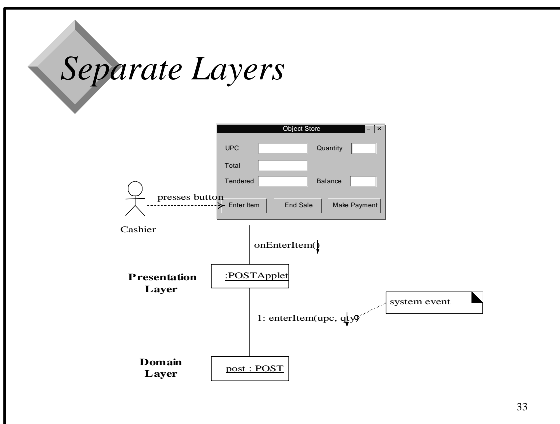
StartUp Contract

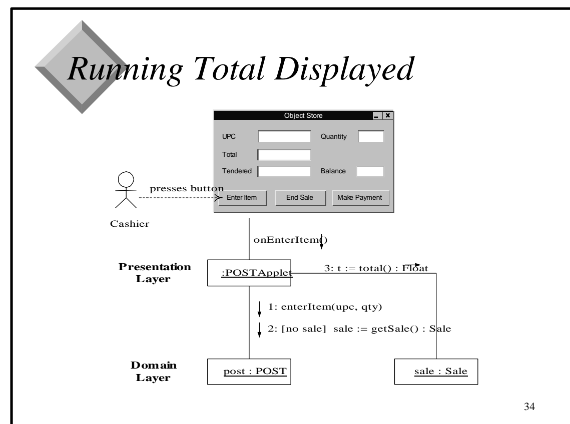
- Post-conditions:
  - Store, POST, ProductCatalog, and ProductSpecification have been created (instance creation)
  - ProductCatalog is associated with ProductSpecification (association formed)
  - Store is associated with ProductCatalog (association formed)
  - Store is associated with POST (association formed)
  - POST is associated with ProductCatalog (association formed)

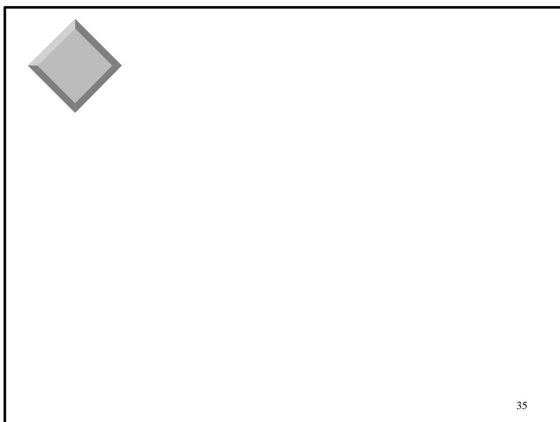
30


StartUp Collaboration Diagram
Pass a reference to the ProductCatalog to the POST, so that it has permanent visibility to it.
create()
:Store
2: create(pc)
:POST
by Creator
1: create()
1.1: create()
1.2.2*: add(ps)
pc: ProductCatalog
:Product Specification
1.2: loadProdSpecs()
1.2.1*: create(upc, price, descripti
ps: ProductSpecification
Asterix in sequence number indicates the message occurs in a repeating section.
31


The Presentation Layer
❖ How does it connect to the rest of the system?
❖ No domain logic responsibilites should be found in the presentation layer
1: create()
create()
:POSTApplet
2: p := getPOST() : POST
store :Store
32


Separate Layers
Object Store
UPC
Quantity
Total
Tendered
Balance
Enter Item
End Sale
Make Payment
presses button
Cashier
onEnterItem()
Presentation Layer
:POSTApplet
system event
1: enterItem(upc, qty)
Domain Layer
post : POST
33


Running Total Displayed
Object Store
UPC
Quantity
Total
Tendered
Balance
Enter Item
End Sale
Make Payment
presses button
Cashier
onEnterItem()
Presentation Layer
:POSTApplet
3: t := total() : Float
1: enterItem(upc, qty)
2: [no sale] sale := getSale() : Sale
Domain Layer
post : POST
sale : Sale
34


35