PROGRESS IN SOFTWARE DESIGN

SOFTWARE DESIGN,
DESIGN PATTERNS,
BUSINESS LAYER OF EAA
FROM APPLICATION TO DESIGN AND TO IMPLEMENTATION

Analysis
Investigation of the problem

Design
Logical solution

Construction
Code

Book
(concept)

Domain concept
Representation in
analysis of concepts

Book

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
</tr>
<tr>
<td>print()</td>
</tr>
</tbody>
</table>

public class Book {
  public void print();
  private String title;
}

Representation in an object-oriented programming language
INTERACTION DIAGRAM NOTATION

COMMUNICATION VS. SEQUENCE DIAGRAMS
INTERACTION DIAGRAMS 2 TYPES

- illustrate how objects interact via messages.
- **Communication** vs **Sequence** diagrams

![Diagram showing communication and sequence interactions between ClassAInstance and ClassBInstance with messages message1(), message2(), and message3().]
COMMUNICATION DIAGRAMS illustrate object interactions in a graph or network format.

ClassAInstance
  ↓ message1( )
  1: message2()
  ↓ 2: message3()

ClassBInstance
SEQUENCE DIAGRAMS

Illustrate interactions in a kind of fence format.

Set of all operation contracts defines system behavior.

Create an interaction diagram for each operation contract.
SEQUENCE DIAGRAMS

X-Axis (objects)

- member: LibraryMember
- book: Book
- :Book Copy

Y-Axis (items)

- borrow(book)
- ok = mayBorrow()
- [ok] borrow(member)
- setTaken(member)

Message

Condition

Life Line

Object

Activation box
EXAMPLE COMMUNICATION DIAGRAM: MAKEPAYMENT

makePayment(cash Tendered) → :Register

1: makePayment(cash Tendered) → :Sale

1.1: create(cash Tendered) → :Payment

parameter

link line

object creation

first message

instance
HOW TO READ THE MAKEPAYMENT COMMUNICATION DIAGRAM

1. The message `makePayment` is sent to an instance of `Register`. The sender is not identified.

2. The `Register` instance sends the `makePayment` message to a `Sale` instance.

3. The `Sale` instance creates an instance of a `Payment`.

```
makePayment(cashTendered)

:Register

↓ 1: makePayment(cashTendered)

:Sale

↓ 1.1: create(cashTendered)

:Payment
```
EXAMPLE SEQUENCE DIAGRAM: MAKEPAYMENT

:Register

makePayment (cashTendered)

:Sale

create (cashTendered)

:Payment

makePayment (cashTendered)
To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined. Additionally a class name should be preceded by a colon.

An instance name can be used to uniquely identify the instance.
MESSAGES TO “SELF” OR “THIS”

A message can be sent from an object to itself. This is illustrated by a link to itself, with messages flowing along the link.

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
CREATION OF INSTANCES

The language independent creation message is create, being sent to the instance being created.

The create message may include parameters, indicating passing of initial values.
CREATION OF INSTANCES

An object lifeline shows the extend of the life of an object in the diagram.

Note that newly created objects are placed at their creation height.
A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to the iteration clause.

The message is sent only if the clause evaluates to true.
CONDITIONAL MESSAGES
OLD VERSION

message1():
A
[color=red]calculate():
B
CONDITIONAL MESSAGES

message1():

\[
\text{calculate()}
\]

\text{[color=red]}

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
MUTUALLY EXCLUSIVE CONDITIONAL PATHS

unconditional after either msg2() or msg4()  

1a and 1b are mutually exclusive conditional paths.
Both are sequence number 1 since either could be the first internal message.

unconditional after either msg2() or msg4()
MUTUALLY EXCLUSIVE CONDITIONAL MESSAGES OLD VERSION

message1():

A: [x<10] calculate()

B: [x>15] calculate()

C:

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
MUTUALLY EXCLUSIVE CONDITIONAL MESSAGES

message1():

A: B

C:

\[x < 10\]

\[x > 15\]

alt

\(\text{calculate}()\)

\(\text{calculate}()\)
CONDITIONAL MESSAGES

- **Sender**
- **receiver1**
- **receiver2**

**alt**
- **[condition1]**
  - message1(parameters)
  - message2(parameters)
  - Interaction occurs if condition1 is met

- **[condition2]**
  - message3(parameters)
  - otherwise, this interaction occurs if condition2 is met

- **[else]**
  - message4(parameters)
  - otherwise, this interaction occurs
Iteration is indicated by following the sequence number with a star (*).

This expresses that the message is being sent repeatedly, in a loop, to the receiver.

It is also possible to include an iteration clause indicating the recurrence values.
MUTUALLY EXCLUSIVE CONDITIONAL MESSAGES

: A

message1()

: B

: C

calculatePart()

loop

[x<10]

calculatePart()
EXAMPLE

Clerk

:Violations
Dialog

:Violations
Controller

:Violations
DBProxy

lookup

viewButton()

id=getID()

display(v)

getViolation(id)

<<create>>

v:Traffic
Violation

May be a pseudo-method

DB is queried and the result is returned as an object

EXAMPLE

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
EXAMPLE

Client

Active object

:PrintServer

:Queue

:Printer Proxy

print(doc, client)

enqueue(job)

job=dequeue()

[job] print(job.doc)

[job] done(status)

Repeated forever with 1 min interludes

Printing A Document

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
GRASP*:
DESIGNING OBJECTS WITH RESPONSIBILITIES

* General Responsibility Assignment Software Patterns
DEMONSTRATION
EXAMPLE

Point of Sale
• Grocery store

Roles
• Customer – shops items
• Cashier – charges customer
• Manager – starts register

Register
• What was soled
• What is the item price
• What to pay / payments
• Receipt
## DEMONSTRATION EXAMPLE

<table>
<thead>
<tr>
<th>Název</th>
<th>Počet</th>
<th>Cena</th>
<th>Celkem</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td></td>
<td>28,00</td>
<td></td>
</tr>
<tr>
<td>horká čokoláda</td>
<td></td>
<td>38,00</td>
<td></td>
</tr>
<tr>
<td>mattoni 0.33</td>
<td></td>
<td>29,00</td>
<td></td>
</tr>
<tr>
<td>nestea 0.2</td>
<td>2x</td>
<td>29,00</td>
<td>58,00</td>
</tr>
<tr>
<td>čaj</td>
<td>3x</td>
<td>32,00</td>
<td>96,00</td>
</tr>
</tbody>
</table>

Celkem: Kč 249,00
**DEMONSTRATION EXAMPLE**

<table>
<thead>
<tr>
<th>Název</th>
<th>Počet</th>
<th>Cena</th>
<th>Celkem</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td></td>
<td>28,00</td>
<td></td>
</tr>
<tr>
<td>horká čokoláda</td>
<td></td>
<td>38,00</td>
<td></td>
</tr>
<tr>
<td>mattoni 0.33</td>
<td></td>
<td>29,00</td>
<td></td>
</tr>
<tr>
<td>nestea 0.2</td>
<td>2x</td>
<td>29,00</td>
<td>58,00</td>
</tr>
<tr>
<td>čaj</td>
<td>3x</td>
<td>32,00</td>
<td>96,00</td>
</tr>
</tbody>
</table>

**Celkem:** Kč 249,00
Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
DOMAIN MODEL FOR EXAMPLES

- Payment
  - amount
- Customer
- Sale
  - date
time
- Sales
  - LineItem
    - quantity
- Contained-in
- Logs-completed
- Captured-on
- Records-sale-of
- Product
  - Catalog
    - Described-by
      - Specification
        - description
        - price
        - itemID
    - Contains
      - Item
        - Contains
      - Described-by
        - Sales
          - LineItem
            - quantity
          - date
time
          - date
          - Customer
          - Payment
            - amount
          - Paid-by
            - Initiates-by
          - Sales
            - Sale
              - date
time
              - Sales
                - Customer
                - Payment
                  - amount

The focus of object design is to:

1. **Identify classes and objects**
2. Decide what **methods** belong **where**
3. And **how** these **objects** should **interact**.

**Responsibilities**

- Obligations of an object in terms of its behavior.

**Two types of responsibilities:**

- **Doing:**
- **Knowing:**
RESPONSIBILITIES AND METHODS

Two types of responsibilities:

- **Doing:**
  - Doing something itself
    - e.g. creating an object, doing a calculation
  - Initiating action in other objects.
  - Controlling and coordinating activities in other objects.

- **Knowing:**
  - Knowing about private encapsulated data.
  - Knowing about related objects.
  - Knowing about things it can derive or calculate.
RESPONSIBILITIES AND METHODS

Responsibilities assigned to classes during object design.

For example:

• “object Sale is responsible for creating SalesLineItems” (doing)
• “object Sale is responsible for knowing its total” (knowing)

Responsibilities - “knowing” often deduced from the Domain Model

• because of the attributes and associations it illustrates
RESPONSIBILITIES AND METHODS

The granularity influences the translation of responsibilities into classes and methods.

- For example, “provide access to relational databases” may involve dozens of classes and hundreds of methods, whereas “create a Sale” may involve only one or few methods.

A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities. Methods either act alone, or collaborate with other methods and objects.
RESPONSIBILITIES AND INTERACTION DIAGRAMS

Within the UML implemented as methods in interaction diagrams.

Sale objects have been given the responsibility to create Payments, handled with the makePayment method.
PATTERNS

Emphasizing principles to **guide choices** where to assign responsibilities.

A **pattern** = named description of a problem with a solution applicable to new contexts;

**Pattern name:** Information Expert

- **Problem:**
  - What is the most basic principle by which to assign responsibilities to objects?

- **Solution:**
  - Assign a responsibility to the class that has the information needed to fulfil it.
Problem: What is a general principle of assigning responsibilities to objects?

Solution: Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.

If you know then answer!

e.g.

• **who should be responsible for knowing the grand total of a sale?**

• *By Information Expert we look for a class that has the information needed to determine the total.*
**INFORMATION EXPERT (OR EXPERT)**

**Expert:** allocate a responsibility to a class that has the information

Knowledge includes having the data, and also knowing who has got the data
It is necessary to know about all the SalesLineItem instances of a Sale and the sum of the subtotals.

A Sale instance contains these

- it is an information expert for this responsibility.
INFORMATION EXPERT (OR EXPERT)

This is a partial interaction diagram.

\[
t := \text{getTotal()}
\]

:Sale

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sale</td>
</tr>
<tr>
<td>date</td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td>total()</td>
</tr>
</tbody>
</table>
What information is needed to determine the line item subtotal?

- quantity and price.

SalesLineItem should determine the subtotal.

This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.
INFORMATION EXPERT (OR EXPERT)
INFORMATION EXPERT (OR EXPERT)

To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.

The ProductSpecification is the information expert on answering its price.
INFORMATION EXPERT (OR EXPERT)
INFORMATION EXPERT (OR EXPERT)

To fulfil the responsibility of knowing and answering the sale’s total,

- three responsibilities were assigned to three design classes

The fulfillment of a responsibility often requires information that is spread across different classes of objects.

This implies that there are many “partial experts” who will collaborate in the task.

<table>
<thead>
<tr>
<th>Class</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sale</td>
<td>Knows Sale total</td>
</tr>
<tr>
<td>SalesLineItem</td>
<td>Knows line item total</td>
</tr>
<tr>
<td>ProductSpecification</td>
<td>Knows product price</td>
</tr>
</tbody>
</table>

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
Problem: Who should be responsible for creating a new instance of some class?

Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:

1. B aggregates A objects.
2. B contains A objects.
4. B has the initializing data that will be passed to A when it is created
   • thus B is an Expert with respect to creating A.
CREATOR

Creator: creating an object should be the responsibility of a class that is closely related to the created object.
In the considered application, who should be responsible for creating a `SalesLineItem` instance?

Since a `Sale` contains many `SalesLineItem` objects, the Creator pattern suggests that `Sale` is a good candidate.
This assignment of responsibilities requires that a `makeLineItem` method be defined in `Sale`.
LOW COUPLING

**Coupling**: measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.

A class with high coupling depends on many other classes (libraries, tools).

**Problems because of a design with high coupling:**

- Changes in related classes force local changes.
- Harder to understand in isolation; need to understand other classes.
- Harder to reuse because it requires additional presence of other classes.

**Problem**: How to support low dependency, low change impact and increased reuse?

**Solution**: Assign a responsibility so that coupling remains low.
LOW COUPLING

Assume we need to create a Payment instance and associate it with the Sale.

What class should be responsible for this?

By Creator, Register is a candidate.
Register could then send an `addPayment` message to Sale, passing along the new Payment as a parameter.

The assignment of responsibilities couples the Register class to knowledge of the Payment class.

Sale also coupled to knowledge of a Payment.
An alternative solution is to create **Payment** and associate it with the **Sale**.

No coupling between **Register** and **Payment**.
LOW COUPLING

Some of the places where coupling occurs:

- Attributes: X has an attribute that refers to a Y instance.
- Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
- Subclasses: X is a subclass of Y.
- Types: X implements interface Y.

There is no specific measurement for coupling, but in general, classes that are generic and simple to reuse have low coupling.

There will always be some coupling among objects, otherwise, there would be no collaboration.
LOW COUPLING

Classes are easier to maintain
Easier to reuse
Changes are localized
Easier to understand
LOW COUPLING

Afferent coupling:
• Number of responsibilities

Efferent coupling:
• Number of dependencies

Instability:
• Ratio of efferent to total coupling \((\text{Afferent} + \text{Efferent})\).
LOW COUPLING

Automation

• Sonar
  • LCOM4
  • https://blog.codecentric.de/en/2012/03/pros-and-cons-of-the-lcom4-metric-in-sonar/

• Cyclomatic complexity

• Cohesion metrics
  • http://www.aivosto.com/project/help/pm-oo-cohesion.html
HIGH COHESION

Cohesion: measure of how strongly related and focused the responsibilities of an element are.

A class with low cohesion does many unrelated activities or does too much work.

Problems because of a design with low cohesion:

- Hard to understand.
- Hard to reuse.
- Hard to maintain.
- Delicate, affected by change.

Problem: How to keep complexity manageable?

Solution: Assign a responsibility so that cohesion remains high.
Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?

By Creator, Register is a candidate.

Register may become bloated if it is assigned more and more system operations.
HIGH COHESION

An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.

This design supports high cohesion and low coupling.

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
HIGH COHESION

Scenarios that illustrate varying degrees of functional cohesion

1. **Very low cohesion**: class responsible for many things in many different areas.
   - e.g.: a class responsible for interfacing with a database and remote-procedure-calls.

2. **Low cohesion**: class responsible for complex task in a functional area.
   - e.g.: a class responsible for interacting with a relational database.
HIGH COHESION

3. **High cohesion**: class has moderate responsibility in one functional area and it collaborates with other classes to fulfill a task.

- e.g.: a class responsible for one section of interfacing with a data base.

**Rule of thumb**: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn’t do much work. It collaborates and delegates.
HIGH COHESION

Classes are easier to maintain
Easier to understand
Often support low coupling
Supports reuse because of fine grained responsibility
HIGH COHESION
LOW COUPLING
HIGH COHESION
LOW COUPLING
CONTROLLER

Problem: Who should be responsible for handling an input system event?

Solution: Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

- Represents the overall system.
- Represents a use case scenario.
- A Controller is a non-user interface object that defines the method for the system operation. Note that windows, applets, etc. typically receive events and delegate them to a controller.

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
CONTROLLER

Where to assign system operations

Who is responsible?

Which class of object should be responsible for handling this system event message?

It is a controller.

enterItem(upc, quantity) → :???
Controller pattern, options:

- Point-of-Sale—overall “system”
- Store—business/organization
- Cashier—real-world entity participating in the task
- BuyItemsHandler—artificial handler to system operations for use case
Good choice has many factors: coupling / cohesion

• **Use façade controller**
  • pokud jsou se jedná o pár systémových událostí
• **UseCaseHandler in case**
  • pokud máme mnoho systémových událostí mezi různými procesy a façade kotrolery by se staly přeplněné
• **Use role controller carefully.**
  o Je snadné přiřadit veškerou práci person-like objektům na místo delegátů

• **Note:** Presentation layer does not handle system events
POSTApplet should not send this message.

It is undesirable for a presentation layer objects such as a Java applet to get involved in deciding how to handle domain processes.

Business logic is embedded in the presentation layer, which is not useful.
Benefits:

Increased potential for reuse.

Keeps external event sources and internal event handlers independent of each other’s type and behaviour.

Reason about the states of the use case.

• Ensure that the system operations occurs in legal sequence,

• Be able to reason about the current state of activity and operations within the use case.
WHAT'S THIS?

- Car uses normal engine to move
- Ford uses V engine to move
- Honda uses i-vtec technology to move
DEFINITION: POLYMORPHISM

- A property of object oriented software by which an abstract operation may be performed in different ways in different classes
- Exists when several classes, which each implement the operation either have a common superclass or implement an interface that contains the operation
- Gets power from dynamic binding
- One of the fundamental features of the object oriented paradigm
DEFINITION: DYNAMIC BINDING

- The **process** of binding a call to a particular **method**. This is performed dynamically at **run-time** due to the presence of **polymorphism**
- Gives power to polymorphism
- Is needed when the **compiler determines** that there are **multiple possible methods** that could be executed by a particular call
- Is a kind of process
- Prevents programmers from having to write **conditional statements** to explicitly choose which code to run
POLYMORPHISM

Problem: How to handle alternatives based on types?

Solution: When alternate behaviours are selected based on the type of an object, use polymorphic method call to select the behaviour, rather than using if statement to test the type.

Motivation

• Does alternatives differ by type? Assign responsibility of doing to polymorphic methods – generalize behaviour
• Do not test type of object, use condition to choose an alternative
POLYMORPHISM: EXAMPLE

By Polymorphism, each payment should authorize itself.
POLYMORPHISM

- Who is responsible for authorization of various payments?
- Cash, check, credit or debit payments
  - authorization differs!
- In Polymorphism, use polymorphic methods
  - and assign them the responsibility

(inherit interface from abstract Payment class)
POLYMORPHISM

• Easier and more reliable then using explicit selection logic
• Easier to add additional behaviors later on
• Increased the number classes in a design
• May make the code less easier to follow
Lessons from Polymorphism

• Future extensions required for unanticipated new variations are easy to add.
• Avoid “case /switch” statements or “if” conditionals on type.
• The objects work on themselves

– Example: payments authorise themselves
• Also known as:
• “Do it myself”, “Choosing Message” & “don’t ask what kind?”

POLYMORPHISM
PURE FABRICATION

Problem: To not violate High Cohesion and Low Coupling?

Solution: Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain, in order to support high cohesion, low coupling, and reuse.
Situation

• Object-oriented design has the characteristics to implement classes that reflect the concept of real world
• But what to do when the responsibility assignments it leads to high coupling and low cohesion?
Sale instances are persistent. Expert suggests - Sale is responsible.

What would be the impact?

– Place a lot of work related to DB operations to the object Sale
  • incohesive

– Sale class is then coupled to the interface DB [JDBC/JPA]
  • coupled

Do we want that?
**Sale** instances are persistent.

- Persistence of objects to DB is a general task reusable by multiple classes.
  
  • Assigning responsibility to **Sale** does not bring reuse

**Suggestion:**
- create new class responsible for DB persistence

<table>
<thead>
<tr>
<th>SaleDAO</th>
</tr>
</thead>
<tbody>
<tr>
<td>create()</td>
</tr>
<tr>
<td>update()</td>
</tr>
<tr>
<td>delete()</td>
</tr>
<tr>
<td>find()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PersistentStorageBroker</th>
</tr>
</thead>
<tbody>
<tr>
<td>save()</td>
</tr>
</tbody>
</table>
PURE FABRICATION EXAMPLE - ADAPTER
Benefits:

High cohesion is supported

• because responsibilities are factored into a class that only focuses on a very specific set of related tasks.

Reuse potential may be increased because of the presence of fine grained Pure Fabrication classes.
INDIRECTION

Problem: To avoid direct coupling?
To de-couple objects so that Low coupling is supported ..and..
.. Reuse potential remains high?

Solution: Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
INDIRECTION

Class

Indirection

Intermediate

Worker

.stackoverflow.com

/.../Java-file-open..

HTTP://

STACKOVERFLOW.COM/

QUESTIONS/10867734/JAVA-

FILE-OPEN-A-FILE-AND-

WRITE-TO-IT

DEVELOPERS LOVE INDIRECTION
Indirection: Assign a responsibility to a class that knows where to find an object that can complete the task,
In the example application, the payment terminal uses modem to send payment transactions.

- Operating system (OS) provides low-level API for networking.
- `CreditAuthorizationService` communicates with `modem`.
  - Add low-level API to the service? (highly coupled to OS)
  - What if OS changes?

- Instead a **mediator** class is provided.
  - Add new class called `Modem` that knows the low-level API.
  - It internally does the interaction, but provides interface to the service
  - This is known as **Proxy**.
INDIRECTION : EXAMPLE

By Indirection

Modem

By Indirection

Modem::dial(phoneNum)
{
  ::OS_OpenPort(1);
  ::OS_Dial(phoneNUM)
}

CreditAuthorizationService

authorize(payment) → CreditAuthorizationService 1::dial(phoneNum) → Modem
EXAMPLE:
PERSISTENT STORAGE BROKER

The Pure fabrication example of de-coupling the Sale from the relational database services through the introduction of a PersistentStorageBroker is also an example of assigning responsibilities to support Indirection.

The PersistentStorageBroker acts as a intermediary between the Sale and database.
LAW OF DEMETER: DON’T TALK TO STRANGERS

Problem: To avoid knowing about the structure of indirect objects?

Solution: If two classes have no other reason to be directly aware of each other or otherwise coupled, then the two classes should not directly interact.
LAW OF DEMETER: DON’T TALK TO STRANGERS

It states that within a method, messages should only be sent to the following objects:

The *this* object (or *self*)
A parameter of the method
An attribute of *self*
An element of a collection which is an attribute of *self*
An object created within the method
DON’T TALK TO STRANGERS!

**POST** instance of controller associates a *Sale*

*Sale* references *Payment*.

**POST** needs to return *paymentAmount*,

- charges for the *Payment*.

*Sale* can return the *Payment* (*payment()*), associated with *Sale*.
DON’T TALK TO STRANGERS
DON’T TALK TO STRANGERS

Introducing new class to POST
= new coupling
= degraded cohesion
DON'T TALK TO STRANGERS

Add the responsibility to direct associated for mediation of calls to Sale to return Payment amount

Promoting the interface
DON’T TALK TO STRANGERS !!!

Keeps coupling between classes low and makes the design more robust

Adds a small amount of overhead in the form of indirect method calls
WHAT DID WE DO?

Illustrates non-attribute visibility
### Use Cases

1. Customer arrives ...
2. Cashier makes new sale.
3. Cashier enters item identifier.
4.

### System Sequence Diagrams

- *Operation: makeNewSale*
  - Post-conditions:
    - "...

- *Operation: enterItem*
  - Post-conditions:
    - A SalesLineItem instance was created
    - "...

### Design Model

- *Register*
  - makeNewSale()
  - enterItem (itemID, quantity)
  - endSale()

- *ProductCatalog*
  - create()
  - spec := getSpecification(itemID)
  - addLineItem(spec, quantity)

- *Sale*
  - endSale()
USE CASE REALIZATIONS
USE CASE REALIZATIONS

A use-case realization describes how a use case is realized in terms of collaborating objects.

**UML interaction diagrams** are used to *illustrate* use case realizations.

Recall **Process Sale**: from main scenario we identified a number of system events (operations).

Each system event was then described by a contract.

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeNewSale()</td>
</tr>
<tr>
<td>addLineItem(itemID, quantity)</td>
</tr>
<tr>
<td>endSale()</td>
</tr>
<tr>
<td>makePayment()</td>
</tr>
</tbody>
</table>

**Contract CO1: makeNewSale**

**Operation:** makeNewSale()

**Cross References:** Use Cases: Process Sale.

**Pre-conditions:** none.

**Post-conditions:**
- A Sale instance `s` was created. (instance creation)
- `s` was associated with the Register (association formed)
- Attributes of `s` were initialized
OBJECT DESIGN: MAKE_NEW_SALE

We work through the postcondition state changes and design message interactions to satisfy the requirements.

:Register

Register creates a Sale by Creator.

By Creator, Sale creates an empty multiobject which will eventually hold SalesLineItem instances

By Controller.

Implied to take place within the constructor of Sale instance.

This is NOT a SalesLineItem Instance but a collection object.
Contract CO2: addLineItem

... 

Post-conditions:

- A **SalesLineItem** instance `sli` was created. (instance creation)
- `sli` was associated with the **Sale**. (association formed)
- `sli.quantity` was set to `quantity`. (attribute modification)
- `sli` was associated with a **ProductSpecification**, based on `itemID` match (association formed)
OBJECT DESIGN: ADDLINEITEM

Contract CO2: addLineItem

... 

Post-conditions:

- A SalesLineItem instance sli was created. (instance creation)
- sli was associated with the Sale. (association formed)
- sli.quantity was set to quantity. (attribute modification)
- sli was associated with a ProductSpecification, based on itemID match (association formed)

\[
\text{addLineItem}(\text{itemID}, \text{quantity}) \rightarrow \text{:Register}
\]
OBJECT DESIGN: ADDLINEITEM

Contract CO2: addLineItem

... 

Post-conditions:

- A SalesLineItem instance `sli` was created. (instance creation)
- `sli` was associated with the Sale. (association formed)
- `sli.quantity` was set to quantity. (attribute modification)
- `sli` was associated with a ProductSpecification, based on itemID match (association formed)

```
addLineItem(itemID, quantity)
```

\[ 
:\text{Register} \]

\[ 1: \text{getSpecification(itemID)} \]

:\text{ProductCatalog}
OBJECT DESIGN: ADDLINEITEM

Contract CO2: addLineItem

...  

Post-conditions:

- A SalesLineItem instance \( sli \) was created. (instance creation)
- \( sli \) was associated with the Sale. (association formed)
- \( sli.quantity \) was set to quantity. (attribute modification)
- \( sli \) was associated with a ProductSpecification, based on itemID match (association formed)

\[
\text{addLineItem}(\text{itemID}, \text{quantity}) \rightarrow \\
\text{:Register} \quad \downarrow 1: \text{getSpecification(} \text{itemID}\text{)} \\
\text{:ProductCatalog} \quad \downarrow 1.1: \text{spec:= find(} \text{itemID}\text{)} \\
\text{:ProductSpecification}
\]
OBJECT DESIGN: ADDLINEITEM

Contract CO2: addLineItem

...  

Post-conditions:

• A SalesLineItem instance \( sli \) was created. (instance creation)
• \( sli \) was associated with the Sale. (association formed)
• \( sli.quantity \) was set to quantity. (attribute modification)
• \( sli \) was associated with a ProductSpecification, based on itemID match (association formed)

\[\text{addLineItem}(\text{itemID}, \text{quantity}) \rightarrow \text{:Register} \rightarrow 2: \text{makeLineItem(spec, quantity)} \rightarrow \text{:Sale} \]

1: getSpecification(itemID)

1.1: spec := find(itemID)

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
OBJECT DESIGN: ADDLINEITEM

Contract CO2: addLineItem

... 

Post-conditions:

- A SalesLineItem instance sli was created. (instance creation)
- sli was associated with the Sale. (association formed)
- sli.quantity was set to quantity. (attribute modification)
- sli was associated with a ProductSpecification, based on itemID match (association formed)

```
addLineItem(itemID, quantity)

:Register

:ProductCatalog

:ProductSpecification

2: makeLineItem(spec, quantity)

:Sale

sli:SalesLineItem
```

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
OBJECT DESIGN: ADDLINEITEM

Contract CO2: addLineItem

... 

Post-conditions:

• A SalesLineItem instance \textit{sli} was created. (instance creation)
• \textit{sli} was associated with the Sale. (association formed)
• \textit{sli.quantity} was set to quantity. (attribute modification)
• \textit{sli} was associated with a ProductSpecification, based on itemID match (association formed)
OBJECT DESIGN: ADDLINEITEM

By Controller.

addLineItem(itemID, quantity)

By Expert.

:Register

1: getSpecification(itemID)

1.1: spec:= find(itemID)

:ProductCatalog

This is a multiobject collection. It contains many instances of ProductSpecification.

:ProductSpecification

:SalesLineItem

find and add are generic implementation-independent messages.

:sli:SalesLineItem

2: makeLineItem(spec, quantity)

2.1: create (spec, quantity)

2.2: add (sli)

By Creator.
OBJECT DESIGN: END_SALE

Contract CO3: endSale

... 

Post-conditions:

- Sale.isComplete became true (attribute modification)

```
public void becomeComplete() {
    isComplete = true;
}
```

UML notation for a constraint

```
{s.isComplete = true}
```

By Controller.

By Expert.

Tomas Cerny, Software Engineering, FEE, CTU in Prague, 2015
CREATING DOMAIN MODEL TO DESIGN

Identify all the classes participating in the software solution.
Do this by analyzing the interaction diagrams.
Draw them in a class diagram.
Duplicate attributes from the associated concepts in the Domain Model.

<table>
<thead>
<tr>
<th>Register</th>
<th>ProductCatalog</th>
<th>ProductSpecification</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>description</td>
<td>amount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>price</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>itemID</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Store</th>
<th>Sale</th>
<th>SalesLineItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>name</td>
<td>isComplete</td>
<td>isComplete</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>quantity</th>
<th>price</th>
<th>itemID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Add method names by analyzing the interaction diagrams.

If the message makeLineItem is sent to an instance of class Sale, then class Sale must define a makeLineItem method.

3: makeLineItem(spec, quantity)
## Creating Domain Model to Design

Add type information to the attributes and methods.

<table>
<thead>
<tr>
<th>Register</th>
<th>ProductCatalog</th>
<th>ProductSpecification</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>description</td>
<td>amount</td>
</tr>
<tr>
<td>endSale()</td>
<td>getSpecification()</td>
<td>price</td>
<td></td>
</tr>
<tr>
<td>addLineItem()</td>
<td>makeNewSale()</td>
<td>itemID</td>
<td></td>
</tr>
<tr>
<td>makePayment()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>Sale</td>
<td>SalesLineItem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>date</td>
<td>Quantity: Integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isComplete: Boolean</td>
<td>getSubtotal()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addSale()</td>
<td>becomeComplete()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>makeLineItem()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>makePayment()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTotal()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>subtotal()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BUSINESS LAYER OF EAA

presentation layer

business layer

persistence layer
BUSINESS LAYER OF EAA

This layer embeds business logic or domain logic. This logic is part of the application that encodes the real-world business rules that determine how data can be created, displayed, stored, and changed. It does much more..

Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation. Enforces constraints, security, validation, work-flow, transaction, concurrency, internal event handling, etc.

Link to details
BUSINESS LAYER OF EAA

Presentation
- RacerCarController

Business
- RacerCarService
  - RacerCarServiceImpl

Persistence
- PersonDAO
  - PersonDAOImpl
  - Person
- CarDAO
  - CarDAOImpl
  - Car
BUSINESS LAYER OF EAA

Presentation

RacerCarController

Business

RacerCarService

Persistence

PersonDAO

CarDAO

Person

Car
BUSINESS LAYER OF EAA

Presentation

RacerCarController

Business

RacerCarService

Persistence

PersonDAO

CarDAO

Person

Car

Constraints, business rules, validations, services, workflow, security, business processes

Tomas Cerhý, Software Engineering, FEE, CTU in Prague, 2015