References:

- Applying UML and patterns
  Craig Larman
Class Libraries vs. Frameworks vs. Patterns

**Definition**

- **Class libraries**
  - Self-contained, “pluggable” ADTs
- **Frameworks**
  - Reusable, “semi-complete” applications
- **Patterns**
  - Problem, solution, context
What are patterns?

- Principles and solutions codified in a structured format describing a problem and a solution
- A named problem/solution pair that can be applied in new contexts
- It is advice from previous designers to help designers in new situations
The idea behind design patterns is simple:
Write down and catalog common interactions between objects that programmers have frequently found useful.

Result:
Facilitate reuse of object-oriented code between projects and between programmers.
Some definitions of design patterns

• “Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.” (Pree, 1994)
• “Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design… and implementation.” (Coplien & Schmidt, 1995).
• “A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it” (Buschmann, et. al. 1996)
• “Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.” (Gamma, et al., 1993)
Characteristics of Good patterns

• It solves a problem
• It is a proven concept
• The solution isn't obvious
• It describes a relationship
• The pattern has a significant human component
Types of Design Pattern

- **Creational patterns**
  - Deal with initializing and configuring classes and objects

- **Structural patterns**
  - Deal with decoupling interface and implementation of classes and objects

- **Behavioral patterns**
  - Deal with dynamic interactions among societies of classes and objects
Types of patterns

Architectural Patterns
Expresses a fundamental structural organization or schema for software systems.

Design Patterns
Provides a scheme for refining the subsystems or components of a software system, or the relationships between them.

Idioms
An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
Describing patterns

Name: It must have a meaningful name.

Problem: A statement of the problem.

Context: This tells us the pattern's applicability.

Forces: A description of the relevant forces and constraints and how they interact/conflict with one another.

Solution: Static relationships and dynamic rules describing how to realize the desired outcome.

Consequences: Implications (good and bad) of using the solution.

Examples: One or more sample applications of the pattern.
GRASP Patterns

Which class, in the general case is responsible?

• You want to assign a responsibility to a class
• You want to avoid or minimize additional dependencies
• You want to maximise cohesion and minimise coupling
• You want to increase reuse and decrease maintenance
• You want to maximise understandability
• …..etc.
GRASP patterns
General Responsibility Assignment Software Patterns

• Expert
• Creator
• Low Coupling
• High Cohesion
• Controller
• Polymorphism
• Pure Fabrication
• Indirection
• Protected Variations
• Law of Demeter
Expert

Problem:
What is the most basic principle by which responsibilities are assigned in object-oriented design?

Solution:
Assign a responsibility to the class that has the information necessary to fulfil the responsibility.
Expert : Example

Who is responsible for knowing the grand total of a sale in a typical Point of Sale application?
Expert : Example

Need all \texttt{SalesLineItem} instances and their subtotals. Only \texttt{Sale} knows this, so \texttt{Sale} is the information expert. Hence

\begin{align*}
t &:= \text{total()} \\
: \text{Sale} \\
\text{New method} &\quad \text{Sale} \\
\text{date} \\
\text{time} \\
\text{total()}
\end{align*}
Expert : Example

But subtotals are needed for each line item (multiply quantity by price). By Expert, SalesLineItem is expert, knows quantity and has association with ProductSpecification which knows price.
Expert: Example

Hence responsibilities assign to the 3 classes.

<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 subtotal</td>
</tr>
<tr>
<td>ProductSpecification</td>
<td>knows product price</td>
</tr>
</tbody>
</table>
Expert

• Maintain encapsulation of information
• Promotes low coupling
• Promotes highly cohesive classes
• Can cause a class to become excessively complex
Creator

Problem:
Assign responsibility for creating a new instance of some class?

Solution:
Determine which class should create instances of a class based on the relationship between potential creator classes and the class to be instantiated.
Creator

who has responsibility to create an object?
By creator, assign class B responsibility of creating instance of class A if

B aggregates A objects
B contains A objects
B records instances of A objects
B closely uses A objects
B has the initializing data for creating A objects

where there is a choice, prefer
B aggregates or contains A objects
Who is responsible for creating \textit{SalesLineItem} objects? Look for a class that aggregates or contains \textit{SalesLineItem} objects.

![Diagram of Sale and SalesLineItem relationships](image)

**Sale**
- date
time

**SalesLineItem**
- quantity

**Product Specification**
- description
- price
- UPC

Contains 1..* Described-by
Creator: Example

Creator pattern suggests Sale.

Collaboration diagram is

\[ \text{Sale} \rightarrow \text{:SalesLineItem} \]

1: create(quantity)

\[ \text{Sale} \]

New method

\[ \text{makeLineItem(quantity)} \]

\[ \text{date time} \]

\[ \text{makeLineItem()} \]

\[ \text{total()} \]
Creator

• Promotes low coupling by making instances of a class responsible for creating objects they need to reference

• By creating the objects themselves, they avoid being dependent on another class to create the object for them
Low Coupling

Problem:
- To support low dependency and increased reuse?

Solution:
- Assign responsibilities so that coupling remains low.
In object oriented languages, common form of coupling from TypeX to TypeY include:

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- TypeX has a method which references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.
Low coupling

- Classes are easier to maintain
- Easier to reuse
- Changes are localised
Low Coupling

How can we make classes independent of other classes?

changes are localised
easier to understand
easier to reuse

Who has responsibility to create a payment?
Low Coupling

Two possibilities:

1. Post

2. Sale

Low coupling suggests Sale because Sale has to be coupled to Payment anyway (Sale knows its total).
High Cohesion

Problem:

To keep complexity manageable?

Solution:

Assign responsibilities so that cohesion remains high.
Some examples:

- Very Low Cohesion: A Class is solely responsible for many things in very different functional areas.
- Low Cohesion: A class has sole responsibility for a complex task in one functional area.
- High Cohesion: A class has moderate responsibilities in one functional area and collaborates with classes to fulfil tasks.
High cohesion

• Classes are easier to maintain
• Easier to understand
• Often support low coupling
• Supports reuse because of fine grained responsibility
High Cohesion

Who has responsibility to create a payment?

1. Post

looks OK if makePayment considered in isolation, but adding more system operations, Post would take on more and more responsibilities and become less cohesive.
High Cohesion

Giving responsibility to *Sale* supports higher cohesion in *Post*, as well as low coupling.

This design supports low coupling and high cohesion so it is desirable
Controller

Problem:
To assign responsibility for handling a system event?

Solution:
If a program receive events from external sources other than its graphical interface, add an event class to decouple the event source(s) from the objects that actually handle the events.
The Controller pattern provides guidance for generally acceptable choices.

Assign the responsibility for handling a system event message to a class representing one of these choices:

1. The business or overall organization (a façade controller).
2. The overall "system" (a façade controller).
3. An animate thing in the domain that would perform the work (a role controller).
4. An artificial class (Pure Fabrication representing the use (a use case controller)).
Benefits:
Increased potential for reuse. Using a controller object 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, or to be able to reason about the current state of activity and operations within the use case.
Controller: Example

System events in Buy Items use case
enterItem()
endSale()
makePayment()
who has the responsibility for enterItem()?
Controller : Example

By controller, we have 4 choices

the overall system Post
the overall business Store
someone in the real world Cashier
who is active in the task Cashier
an artificial handler of all system BuyItemsHandler

The choice of which one to use will be influenced by other factors such as cohesion and coupling
Who should be the controller for system events such as `enterItem` and `endSale`?

![Diagram of the control flow](image)

- **Cashier** presses button
- **Interface Layer**
  - `SaleJFrame`
    - `actionPerformed(actionEvent)`
    - `enterItem(itemId, qty)`
- **Domain Layer**
  - `???

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

It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

**Figure 16.14 Controller for enterItem?**
Good design
- presentation layer decoupled from problem domain
Bad design – presentation layer coupled to problem domain

Presentation Layer (Command object) → Cashier presses button

Object Store

<table>
<thead>
<tr>
<th>UPC</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>Tendered</td>
<td>Balance</td>
</tr>
</tbody>
</table>

| Enter Item | End Sale | Make Payment |

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.

Domain Layer

1: `makeLineItem(upc, qty)`

POSTApplet should not send this message.

:POSTCommand

:Sale
Controller

• Using a controller object keeps external event sources and internal event handlers independent of each other’s type and behaviour

• The controller objects can become highly coupled and uncohesive with more responsibilities
Polymorphism

Problem:
   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.
Polymorphism: Example

By Polymorphism, each payment should authorize itself.
Example: Polymorphism
Polymorphism

- Easier and more reliable than using explicit selection logic
- Easier to add additional behaviours later on
- Increased the number of classes in a design
- May make the code less easier to follow
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.
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.
Example
Suppose, in the point of sale example, that support is needed to save Sale instances in a relational database. By Expert, there is some justification to assign this responsibility to Sale class. However.

• The task requires a relatively large number of supporting database-oriented operations and the Sale class becomes incohesive.

• The sale class has to be coupled to the relational database increasing its coupling.

• Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the Sale class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.
Pure Fabrication : Example

- The Sale remains well design, with high cohesion and low coupling
- The PersistentStorageBroker class is itself relatively cohesive
- The PersistentStorageBroker class is a very generic and reusable object
Pure Fabrication

- Preserves low coupling and high cohesion of classes
- Improve reusability of 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.
Example: PersistentStorageBroker

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 an intermediary between the *Sale* and database.
Indirection: Example

By Indirection

Modem

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

CreditAuthorizationService

authorize(payment) → CreditAuthorizationService 1:dial(phoneNum) → Modem
• Assume that:
  - A point-of-sale terminal application needs to manipulate a modem in order to transmit credit payment request
  - The operating system provides a low-level function call API for doing so.
  - A class called CreditAuthorizationService is responsible for talking to the modem
• If CreditAuthorizationService invokes the low-level API function calls directly, it is highly coupled to the API of the particular operating system. If the class needs to be ported to another operating system, then it will require modification.
• Add an intermediate Modem class between the CreditAuthorizationService and the modem API. It is responsible for translating abstract modem requests to the API and creating an Indirection between the CreditAuthorizationService and the modem.
Indirection

- Low coupling
- Promotes reusability
Law of Demeter

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

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
Law of Demeter : Example
Violates Law of Demeter : Example

```cpp
POST::PaymentAmount()
{
    prnt:= m_sale->Payment()
    //Violates Law of DM
    return prnt->amountTendered();
}
```

Violates Law of DM

prnt is a 'Stranger' to POST
Support Law of demeter

Supports the Law of Demeter

```
amt:=paymentAmount():Float

POST:: PaymentAmount()
{
  return m_sale -> Payment();
}
```

```
:POST
1:prnt:=payment() : payment
1.1:amt:=amountTendered():Float

:Sale

:Sale

date : Date
isComplete : Boolean
time : Time

becomeComplete( )
makeLineitem( )
makePayment( )
payment( )
paymentAmount( )
total( )

prnt:Payment
Law of Demeter

- Keeps coupling between classes low and makes a design more robust
- Adds a small amount of overhead in the form of indirect method calls
Law of Demeter – Time totalling example
Time totalling example

Employee - Instances of the Employee class represent an employee.
PayrollRules – The rules for paying an employee vary with the laws that apply to the location where the employee works. Instances of the PayrollRules class encapsulate the pay rules that apply to an employee.
PayPeriod – Instances of the Payperiod class represent a range of days for which an employee is paid in the same pay slip.
Shift – Instances of the Shift class represent ranges of time that the employee worked.
TimeTotaller – The Timetotaller class is an abstract class that the PayPeriod class uses to break the total hours worked during a pay period into normal and overtime minutes.
C1TimeTot,C2TimeTot,C3TimeTot – Concrete subclasses for different location of TimeTotaller that encapsulate the rules for breaking total minutes worked into normal and overtime minutes worked.
The following interaction must occur:

- The pay period must become associated with an instance of the subclass of TimeTotaller appropriate for the employee when the PayPeriod object is created.
- The TimeTotaller object must be able to examine each shift in the pay period to learn the number of minutes worked in each shift.
Bad time-totalling collaboration

PayPeriod class has no reason to know anything about the PayrollRules class

For TimeTotaller to have direct access to the collection of shifts that it needs implies violation of the Shift class’s encapsulation of how it aggregates Collection of shifts -- resulting in higher level of coupling
Good time-totalling collaboration

To preserve the level of cohesion and coupling a less direct interaction may be used.

This is done as shown by the following collaboration diagram and the creation of additional methods.
Good time-totalling collaboration
Law of Demeter – Time totalling example with added operations
Sample UP Artifact Relationships for Use-Case Realization

**Domain Model**

- **Sale**
  - date
  - ...
- **SalesLineItem**
  - quantity
  - ...

**Use-Case Model**

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

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

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

- **Operation: endSale**
  - **Post-conditions:**
  - ...

**System Sequence Diagrams**

**Contracts**

- **Operation: makeNewSale**
  - Post-conditions:
  - A `SalesLineItem` instance `sli` was created
  - ...

- **Operation: enterItem**
  - Post-conditions:
  - ...

**Design Model**

- **Register**
  - `makeNewSale()`
- **ProductCatalog**
  - `create()`
- **Sale**
  - `...`

**Use Cases**

- **Some ideas and inspiration for the post-conditions derive from the use cases**

**Conceptual classes in the domain inspire the names of some software classes in the design**

**Domain objects**

- The domain objects, attributes, and associations that undergo state changes

---

69
Register makes a Sale by Creator:

- **create()**:
  - **Sales**
  - **LineItem**

**CAUTION:**
This is not a SaleLineItem instance. This is a collection object (such as a List) that can hold SalesLineItem objects.
Register

makeNewSale() by Controller
makePayment(cashTendered)

:Register

:Sale

1: makePayment(cashTendered)

1.1: create(cashTendered)

:Payment

by Controller

by Creator and Low Coupling
endSale() -> :Register

1: becomeComplete() -> s :Sale

by Controller

by Expert
tot := getTotal()

by Expert

:Sale

1 *: st := getSubtotal()

by Expert

: SalesLineItem

1.1: pr := getPrice()

recall this special notation to indicate iteration over the elements of a collection

:ProductSpecification
makePayment(cashTendered)

1: makePayment(cashTendered)

s :Sale

1.1: create(cashTendered)

:Payment

:Register

2: addSale(s)

completedSales: Sale

2.1: add(s)

:Store

note that the Sale instance is named 's' so that it can be referenced as a parameter in messages 2 and 2.1

by Expert
This find message is to the Map object (the multiobject), not to a ProductSpecification.

CAUTION:
This is a multiobject collection (such as a Map), not a ProductSpecification. It may contain many ProductSpecifications.

This enterItem message is to the Map object (the multiobject), not to a ProductSpecification.

CAUTION:
This is a multiobject collection (such as a Map), not a ProductSpecification. It may contain many ProductSpecifications.

This create message is to the Map object (the multiobject), not to a ProductSpecification.

CAUTION:
This is a multiobject collection (such as a Map), not a ProductSpecification. It may contain many ProductSpecifications.

This makeLineItem message is to the Map object (the multiobject), not to a ProductSpecification.

CAUTION:
This is a multiobject collection (such as a Map), not a ProductSpecification. It may contain many ProductSpecifications.

This create message is to the multiobject (e.g., List), not to a SalesLineItem.

CAUTION:
This is a multiobject collection (such as a List), not a SalesLineItem. It may contain many SalesLineItems.

2.1: create(spec, qty)

2.2: add(sl)

add the newly created SalesLineItem instance to the multiobject (e.g., List)
1.1: create(driversLicenseNum, total)

1.2: authorize()

by Do It Myself and Polymorphism

by Creator
1.1: makeCreditPayment(ccNum, expiryDate)
1.2: authorize()

:Register

:Sale

:CreditCard

:CreditPayment

by Creator

by Do It Myself and Polymorphism

create (ccNum, expiryDate)
create (ccNum, expiryDate, total)
Store is responsible for knowing and adding completed Sales.

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

SalesLedger is responsible for knowing and adding completed Sales.

Suitable when the design grows and the Store becomes uncohesive.