TODAY’S LECTURE

We will discuss 7 of the GRASP design patterns – cohesion and coupling were covered earlier.

These provide principles for evaluating and improving designs.
Understanding responsibilities is key to object-oriented design.  
Martin Fowler
RESPONSIBILITY-DRIVEN DESIGN

RDD is a metaphor for thinking about object-oriented design.

Think of software objects similar to people with responsibilities who collaborate with other people to get work done.

RDD leads to viewing an OO design as a community of collaborating responsible objects.
GRASP

General Responsibility Assignment Software Patterns or Principles (GRASP)

A learning aid for OO Design with responsibilities.

A collection of patterns/principles for achieving good design – patterns of assigning responsibility.

One such collection – others exist, e.g., SOLID
REALM OF GRASP

GRASP patterns refer to *software objects*.

Inspiration can of course be drawn from *domain objects*. 
RESPONSIBILITIES AND METHODS

Responsibilities are mostly to do with methods.

Applying GRASP can be done when drawing interaction diagrams and while coding.
LIST OF GRASP PATTERNS

Low coupling (already covered)

High cohesion (already covered)

Controller

Polymorphism

Indirection

Creator

Information expert

Pure fabrication

Protected variations
CREATOR
Who should be responsible for creating a SalesLineItem instance?
CREATING A SALELINEITEM

What about the ProductDescription?
DISCUSSION

Basic idea is to find a creator that needs to be connected to the created object in any event.

Also need initialisation data to be nearby – sometimes requires that it is passed into client. e.g., ProductDescription needs to be passed in.
CREATOR (SUMMARY)

Who creates? (Related to Factory design pattern.)

Assign class B the responsibility to create an instance of class A if one of these is true:

1. B contains A
2. B aggregates A
3. B has the initialising data for A
4. B records A
5. B closely uses A
Consider a LinkedList.

A client creates the objects stored in the list, but the LinkedList object creates the links of the list.

For complex construction, or when instance depends upon a specific external value, other options may need to be considered. Concrete Factory or AbstractFactory patterns.
INFORMATION EXPERT
INFORMATION EXPERT

What is general principle for assigning responsibilities to objects?

Consider that there may be 100s or 1000s of classes.

To which ones do we assign particular functionality?

Choose well and design will be easier to understand, maintain, extend, and reuse.
SOLUTION

Assign responsibility to the information expert—the class that has the information to fulfil the responsibility.

Start by clearly stating the responsibility.

Next, look to Design Model.

If that is not helpful, look to Domain Model and attempt to use (or expand) its classes to inspire the create of design classes.
Who should be responsible for knowing the grand total of a sale?
## RESPONSIBILITIES

<table>
<thead>
<tr>
<th>Design Class</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sale</td>
<td>knows sale total</td>
</tr>
<tr>
<td>SaleLineItem</td>
<td>knows line item subtotal</td>
</tr>
<tr>
<td>ProductDescription</td>
<td>knows product price</td>
</tr>
</tbody>
</table>
SOLUTION
DISCUSSION

Notice that it required information spread across different classes of objects.

Partial information experts collaborate in the task.

Sometimes other considerations overrule.

For instance, should a Sale object save itself in database?

If so, then all database code would be scattered across codebase.
CONTROLLER
PROBLEM

Which first object beyond the UI later receives and coordinates ("controls") a system operation?

Examples: "End sale" button, "Spell check" button.

A **Controller** is the first object beyond the UI later that is responsible for receiving and handling a system operation message.
SOLUTION

Assign the responsibility to a class representing one of the following choices:

- represents the overall “system”, a “root object”, a device that the software is running within, or a major subsystem.

- represents the use case scenario within which the system event occurs.

- use the same controller class for all system events in the same use case scenario.

- think in terms of sessions – instances of conversations with an actor.

Note that “view”, “window”, “document” classes are not on this list!
EXAMPLE

system operations discovered during behavioural analysis
ALLOCATION #1

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

Register
...
endSale()
enterItem()
makeNewSale()
makePayment()
makeReturnItem()
enterReturnItem()
....

system operations discovered during behavioural analysis

allocation of system operations during design, using one facade controller
ALLOCATION #2

allocation of system operations during design, using several use case controllers
DISCUSSION

This is simply a delegation pattern – the UI should not contain application logic.

Increases potential for reuse and pluggable interfaces.

Creates opportunity to reason about state of use case, for example, to ensure that operations occur in a legal sequence.
Problem: **Bloated Controllers**

- a single controller that receives all system events, does too much of the work handling events, has too many attributes (duplicating information found elsewhere), etc.

Remedies:

- Add more controllers.

- Design controller so that it primarily delegates the fulfilment of each system operation to other objects.
POLYMORPHISM
PROBLEM

How to handle alternatives based on types?

• if-then-else means variation, but nonextensible when new variations arise

How to create pluggable software components?

• view objects in client-server relationship: how can you replace one server component with another without affecting the client?
When related alternatives or behaviours vary by type, assign responsibility for the behaviour—using polymorphic operations—to types for which the behaviour varies.

Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.
EXAMPLE: SUPPORTING THIRD-PARTY TAX CALCULATORS?

Different possible tax calculators.

Each has a different interface.

Each supports different communication protocols (TCP/IP socket, SOAP, Java RMI).

What objects should be responsible for handling these varying external tax calculator interfaces?
POLYMORPHISM

By Polymorphism, multiple tax calculator adapters have their own similar by varying behaviours for adapting to different external tax calculators.

`getTaxes(Sale) : List<TaxLineItems>`

- `TaxMasterAdapter`
- `GoodAsGoldTaxProAdapter`
- `ShonkyTaxFraudAdapter`
REPRESENTING POLYMORPHISM IN UML
REPRESENTING
POLYMORPHISM IN UML
DISCUSSION

Interfaces or superclasses:

• Guideline: use interfaces when you want polymorphism without committing to a particular class hierarchy.

Future-proofing:

• if variability at a particular point is very probably, then expend the effort to accommodate flexibility.

• Avoid adding flexibility just because it is possible.
PURE FABRICATION
What objects should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Information Expert (for example) are not appropriate?
Even though classes often correspond to real-world domain objects, there are often cases where assigning responsibility to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential.
SOLUTION

Assign a highly cohesive set of responsibilities to a convenience class, not representing a problem domain concept.

Fabrication – made up.

Pure – keep it clean: high cohesion, low coupling.

“Pure fabrication” – English idiom that implies making something up.
EXAMPLE

Need to save *Sale* instances in a relation database.

Information Expert says assign functionality to *Sale*.

Implications:

- Task needs large number of supporting database-oriented operations, none related to the concept of a *Sale*. Incohesion!

- *Sale* becomes coupled to database interface, so coupling goes up.

- Saving objects in a database is a general task – many classes will need it.
**SOLUTION**

- **PersistentStorage**
  - insert( Object )
  - update( Object )

- Understandable concept.
- Pure software concept.
- Not in domain model.

- Sale unaffected.
- Cohesive concept.
- Generic and reusable.
DISCUSSION

Design of objects can be broadly divided into two categories:

• Representational decomposition. e.g., Sale

• Behavioural decomposition. e.g., TableOfContentsGenerator.

Pure Fabrications are often the result of behavioural decomposition.

Often highly cohesive with high reuse potential.

Avoid overuse: functions and algorithms generally should not be represented by objects.
INDIRECTION
PROBLEM

How to assign responsibility to avoid direct coupling between two (or more) things?

How to decouple objects so that low coupling is supported and reuse potential remains higher?
Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

The intermediary creates *indirection* between the other components.
EXAMPLE

The adaptor acts as a level of indirection to external systems.
PROTECTED VARIATIONS
PROBLEM

How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
SOLUTION

Identify points of predicted variation or instability

Assign responsibilities to create a stable interface around them.

“Interface” in broadest sense – not just Java interface.
The `ITaxCalculatorAdapter` interface (from Polymorphism) allows for future tax calculators that may not yet have been thought of.
Core protected variation mechanisms: data encapsulation, interfaces, polymorphism, standards, virtual machines, operating systems.

Service lookup: clients look up server with stable interface via technology such as Java JINI or UDDI for Web services.

*Law of Demeter:* objects never talk to objects they are not directly connected to.
TODAY’S LECTURE

We covered 7 of the GRASP design patterns.