ADVANCED SOFTWARE DESIGN
LECTURE 4
GRASP
Dave Clarke
TODAY’S LECTURE

We will discuss and apply the GRASP design patterns.

These provide principles for evaluating and improving designs.
General Responsibility Assignment Software Patterns or Principles (GRASP)

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

Refer to software objects not domain objects.
LIST OF GRASP PATTERNS

Low coupling
High cohesion
Creator
Information expert
Controller
Polymorphism
Indirection
Pure fabrication
Protected variations
LOW COUPLING

Problem: How to reduce the impact of change?

Advice: Assign responsibilities so that (unnecessary) coupling remains low.

Use this principle to evaluate alternatives.

Focus on points of realistic instability or evolution.

Drawback: may add additional layers of indirection.
**Problem:** How to keep objects focussed, understandable, and manageable, and as a side effect, support Low Coupling?

**Solution:** Assign responsibilities so that cohesion remains high.

*Use this to evaluate alternatives.*
**Problem**: Who creates an A?

**Advice**: Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- B “contains” or compositely aggregates A.
- B records A.
- B closely uses A.
- B has the initialising data for A.
**Problem:** What is the basic principle by which to assign responsibilities/functionality to objects?

**Advice:** Assign a responsibility to a class that has the information needed to fulfil it.
**Problem:** What first object beyond the UI layer receives and coordinates ("controls") a system operation?

**Advice:** Assign the responsibility to an object representing one of these choices:

- Represents the overall "system", a "root object", a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*).
- Represents a use case scenario within which the system operation

Avoid bloated controllers.
**Problem:** How to handle alternatives based on type? How to create pluggable software components?

**Solution:** When related alternatives or behaviours vary by type (class), assign responsibility for the behaviour —using polymorphic operations— to the types for which the behaviour varies.

Avoid adding flexibility just because you can.
INDIRECTION

**Problem:** Where to assign a responsibility, to avoid direct coupling between two (or more) classes? How to decouple objects so that low coupling is supported and reuse potential remains higher?

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

**Problem:** What object 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?

**Solution:** Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept—sometimes invented to support high cohesion, low coupling, and reuse.
PROTECTED VARIANTS

**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 is used in broadest sense – not just Java interfaces.*
SOME USE CASES

A. player finding an opponent

B. initialising a game

C. making a move (all of the steps)

D. finishing a game (when final move is made)

E. interaction between user interface and game (user presses button, what happens)

F. Restarting a game (from list of games)
LIST OF GRASP PATTERNS

Low coupling
High cohesion
Creator
Information expert
Controller
Polymorphism
Indirection
Pure fabrication
Protected variations
LONGER VERSION OF SLIDES
COUPLING AND COHESION
COUPLING AND COHESION

- **Coupling**: Amount of relations between sub-systems
  - Low coupling general design goal: Independence, supportability
  - May lead to extra layers of abstraction
- **Cohesion**: Amount of relations within a sub-system
  - High cohesion general design goal: Reduce complexity
  - Often a trade-off
PROPERTIES OF A GOOD ARCHITECTURE

minimises coupling between modules

• Goal: modules don’t need to know much about one another to interact
• Low coupling makes future change easier

maximises cohesion within modules

• Goal: the contents of each module are strongly inter-related
• High cohesion makes a module easier to understand

Applies also to classes
LOW COUPLING
How to support low dependency, low change impact, and increase reuse?
COUPLING

• Measure how strongly one element is connected to, has knowledge of or relies on other elements.

• An element with weak low (or weak) coupling is not dependent on too many other elements.
WHEN ARE TWO CLASSES COUPLED?

- Common forms of coupling from TypeX to TypeY:
  - TypeX has an attribute that refers to a TypeY instance.
  - A TypeX object calls on services of a TypeY object.
  - TypeX has a method that references an instance of TypeY (parameter, local variable, return type).
  - TypeX is a direct or indirect subclass of TypeY.
  - TypeY is an interface and TypeX implements that interface.
HIGH COUPLING (BAD)

• A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable and suffer from the following problems:
  
  • Forced local changes because of changes in related classes.
  
  • Harder to understand in isolation.
  
  • Harder to reuse because its use requires the additional presence of the classes on which it is dependent.
SOLUTION

- Assign responsibility so that coupling remains low.
- Use this principle to evaluate alternatives.
EXAMPLE

• Consider the following classes.

  Payment  Register  Sale

• We need to create a Payment instance and associate it with the Sale.

• Which class should be responsible?
POSSIBILITY #1

This is the solution Creator pattern (later) would suggest.
POSSIBILITY #2

makePayment() : Register

1:makePayment() : Sale

1.1:create() : Payment
WHICH IS BETTER?

Assume that each Sale will eventually be coupled with a Payment.

Design #1 has the coupling of Register and Payment, which is absent in Design #2.

Design #2 therefore has lower coupling.

Note that two patterns—Low Coupling and Creator—suggest different solutions.

Do not consider patterns in isolation.
DISCUSSION.

• Low coupling encourages designs to be more independent, which reduces the impact of change.

• Needs to be considered with other patterns such as Information Expert (later) and High Cohesion.

• Subclassing increases coupling – especially considering Domain objects subclassing technical services (e.g., PersistentObject)

• High coupling to stable “global” objects is not problematic – to Java libraries such as java.util.
PICK YOUR BATTLES

• The problem is not high coupling per se; it is high coupling to unstable elements.

• Designers can future proof various aspects of the system using lower coupling and encapsulation, but there needs to be good motivation.

• Focus on points of realistic high instability and evolution.
HIGH COHESION
PROBLEM

How to keep objects focussed, understandable, and manageable, and as a side-effect, support Low Coupling?

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

An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion.
LOW COHESION

A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly affected by change.
SOLUTION

• Assign responsibility so that cohesion remains high.

• Use this to evaluate alternatives.
EXAMPLE

- Game

play game

2 2 2

what next
LOW COHESION

Game is doing all the work. How related are tasks A, B, C? If not strongly related, then design has low cohesion.
HIGH COHESION

Delegates and distributes the work, presumably to objects with appropriately assigned responsibilities.
DISCUSSION

• **Very low cohesion**: a class that does two completely different tasks, e.g., database connectivity and RPC.

• **Low cohesion**: a class that has sole responsibility for a complex task in one functional area, e.g., one single interface responsible for all database access.

• **Moderate cohesion**: a lightweight class, solely responsible for a few logically related areas, e.g., *Company* that knows the employees and the financial details.

• **High cohesion**: a class with moderate responsibilities in one functional area that collaborates with other classes.
RULES OF THUMB

• For high cohesion, a class must
  • have few methods
  • have a small number of lines of code
  • not do too much work
  • have high relatedness of code.
CREATOR
SCENARIO: POS APPLICATION

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.

Need initialisation data nearby – sometimes requires that it is passed into client. e.g., ProductDescription needs to be passed in.

Creator applies to every object.
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
VARIATIONS

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.

Who creates a Concrete Factory?
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.
EXAMPLE: POS

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>ProductDescription</td>
<td>knows product price</td>
</tr>
<tr>
<td>SaleLinItem</td>
<td>knows line item subtotal</td>
</tr>
<tr>
<td>Sale</td>
<td>knows grand total</td>
</tr>
</tbody>
</table>
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.

Information Expert is about responsibilities, not location of data.
CONTROLLER
PROBLEM

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

Examples: After “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.

Central to Model-View-Controller.
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

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>endSale()</td>
</tr>
<tr>
<td>enterItem()</td>
</tr>
<tr>
<td>makeNewSale()</td>
</tr>
<tr>
<td>makePayment()</td>
</tr>
<tr>
<td>makeReturnItem()</td>
</tr>
<tr>
<td>enterReturnItem()</td>
</tr>
</tbody>
</table>

system operations
discovered during
behavioural analysis
ALLOCATION # 1

A bloated controller is an example of low cohesion.

- System operations discovered during behavioural analysis

- Allocation of system operations during design, using one facade controller

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

```
Register
...
endSale()
enterItem()
makeNewSale()
makePayment()
makeReturnItem()
enterReturnItem()
...
```
ALLOCATION #2

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

A bloated controller is an example of low cohesion.
POLYMORPHISM
How to handle alternatives based on types?

- if-then-else means variation, but non-extensible 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?
By Polymorphism, multiple tax calculator adapters have their own similar by varying behaviours for adapting to different external tax calculators.
REPRESENTING POLYMORPHISM IN UML SEQUENCE DIAGRAMS
REPRESENTING POLYMORPHISM IN UML SEQUENCE DIAGRAMS
Interfaces or superclasses:

- Guideline: use interfaces when you want polymorphism without committing to a particular class hierarchy.
- Liskov substitution principle – a value can be replaced by a subtype without changing important properties of program.

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?
IN DETAIL

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.

Example: an (x,y)-coordinate in a GIS application
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.

Most classes not appearing in the domain model will be pure fabrications.
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 data base interface, so coupling goes up.

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

<table>
<thead>
<tr>
<th>PersistentStorage</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert( Object )</td>
</tr>
<tr>
<td>update( Object )</td>
</tr>
</tbody>
</table>

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.

The intermediary is likely to be a pure fabrication.
EXAMPLE

The adaptor acts as a level of indirection to external systems.
PROTECTED VARIATIONS
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?
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 *ITaxCalculatorAdaptor* interface (from Polymorphism) allows for future tax calculators that may not yet have been thought of.
OTHER APPROACHES TO PROTECTED VARIATIONS

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.
OTHER DESIGN PRINCIPLES
(FOR YOUR INTEREST)
SINGLE RESPONSIBILITY

A class should have a single responsibility.
OPEN/CLOSED PRINCIPLE

Software entities should be open for extension but closed for modification.
LISKOV SUBSTITUTION PRINCIPLE

Objects should be replaceable with instances of their subtypes without altering correctness.

Related: design by contract.
Many client-specific interfaces are better than one general-purpose interface
DEPENDENCY INVERSION PRINCIPLE

Depend upon abstractions, not upon concretions.

Decoupling

Program to an interface, not an implementation.

Dependency injection is one method of following this principle.

Related to information hiding – provide stable interface to protect against changes to implementation
OTHERS
“You ain’t gonna need it.”

Implement only things that you will need.

Do the simplest thing that could possibly work.

From Extreme Programming.
“Don’t repeat yourself.”

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Not well supported in existing languages – relies heavily on code generators, build systems, scripting languages to enforce.
KISS

“Keep it simple, stupid” or

“Keep it simple stupid.”

Avoid unnecessary complication in systems.
OTHER PRINCIPLES

Abstraction – designs depend on higher level concepts, not implementation details, *an address book is not a list*

Encapsulation – group structural and behavioural elements together

Information hiding – avoid exposing implementation decisions

Minimise the accessibility of classes and members

Favour composition over inheritance