GRASP (General Responsibility Assignment Software Patterns)

Designing Objects with Responsibilities

- Learn to apply five of the GRASP principles or patterns for OOD.
- GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.

  UML Vs design principles
- The critical design tool for software development is a mind well educated in design principles.
- It is not the Unified Modeling Language, the Rational Unified Process, or any other methodology.

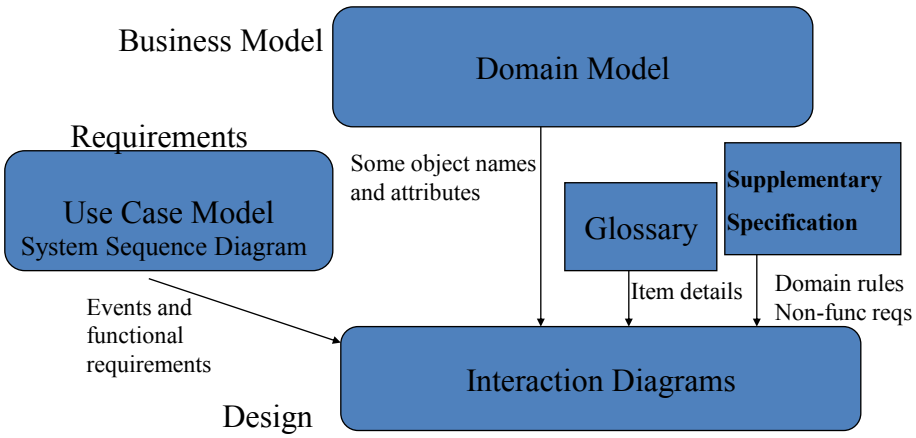
  What activities is design?
- As we switch from being analysts to designers, there are several different ways we can continue.

  1. Start developing, probably with test-first development (or)
  2. Start UML modeling for object design (or)
  3. Start with another modeling technique, such as CRC (class responsibility collaborated) cards.

Design activities

- During design and coding we apply various OO design principles, among them software design patterns such as the GRASP and GoF (Gang of Four) patterns.
- Our metaphor is Responsibility Driven Design (RDD), because our chief design emphasis is on assigning responsibilities to objects.

Design Model Relationships



6

Responsibilities and Methods

- Responsibilities are realized as methods
- Approach to understanding and using design principles is based on patterns of assigning responsibilities.
- Definition of Responsibility:

  "A contract or obligation of a classifier"

- Responsibilities are of the following types:
  - Knowing
  - Doing

Doing

Doing responsibilities of an object include :

  - Doing something itself , such as creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects

Knowing

- Knowing responsibilities of object include :
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate

Responsibilities and interaction diagrams

- Interaction diagrams show choices in assigning responsibilities to objects. When these diagrams are created, decisions in responsibility assignment are reflected in the messages sent to different classes of objects.

Patterns

- Principles and idioms codified in a structured format describing the problem, solution, and given a name are called **patterns.**
- A "New Pattern" is an oxymoron. Patterns are tried and true, established ways of doing things. They are not new things or new ways of doing things.

Definition of Pattern

- Pattern is a named problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussion of its trade-offs. Example.

Pattern name: information expert

Problem: what is a basic principle by which to assign responsibility to object?

Solution: assign a responsibility to the class that has the information needed to fulfill it.

Naming Patterns

- All Patterns have suggestive names. Naming a pattern, technique, or a principle has the following advantages :
  - It supports chunking and incorporating that concept into our understanding and memory
  - It facilitates communication.

Changing Emphasis

- Throughout analysis, we have referred to analyzing the problem domain, and postponing consideration of the solution domain. We emphasized domain objects over software objects.
- That changes in design. We are now concerned with software objects and the solution domain.

GRASP General Responsibility **A**ssignment **S**oftware **P**atterns

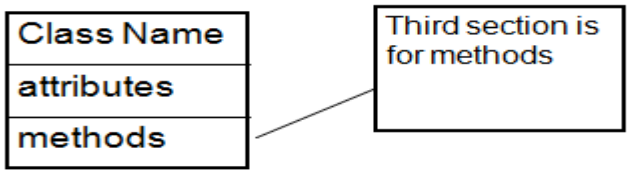
- The GRASP name was chosen to suggest the importance of grasping these principles to successfully design object-oriented software.
- It is an acronym for **G**eneral Responsibility **A**ssignment **S**oftware **P**atterns.
- They describe the fundamental principles of object design and responsibility assignment, expressed as patterns.

Five GRASP patterns :

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller

UML Class diagram notation

- Three sections:



3

Creator

- Problem: Who should be responsible for creating a new instance of some class ?
- The creation of objects is one of the most common activities in an object-oriented system.
- It is useful to have a general principle for assignment of creation responsibilities.
- Assigned well, the design can support low coupling increased clarity, encapsulation, and reusability.

Solution

- Assign class B the responsibility to create an instance of class A if one or more of the following is true :
  - B aggregates A objects .
  - B contains A objects.
  - B records instances of A objects.
  - B closely uses A objects.
  - B has the initializing data that will be passed to A when it is created
  - B is a creator of A objects.

Discussion

- Creator guides assigning responsibilities related to creation of objects.
- The basic intent of the Creator pattern is to find a creator that needs to be connected to the created object in any event.
  - Aggregator aggregates Part
  - Container contains Content
  - Recorder records.
- Composition is an excellent candidate considered by the creator pattern for creating object.

Contraindications

- Creation requires significant complexity, such as:

  - using recycled instances for performance reasons,
  - conditionally creating an instance from similar classes based upon some external property value
- In these instances, another pattern such as concrete factory or abstract factory may be preferred.
- Benefits : creator pattern supports low coupling. Due to this there are less dependencies between objects which enhances reusability.

Related patterns or principles

- Low coupling
- Abstract factory
- Aggression and composition

Monopoly Example

- Larman suggests in his example of a Monopoly Game, that since the game board contains the squares where activities take place, a Board software object is a logical place to create Square software objects.

Information Expert

- By Information expert we should look for the class of objects that has the information needed to determine the total.
- Problem : What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert – a class that has the information necessary to fulfill the responsibility.
- Such a class is known as information expert.

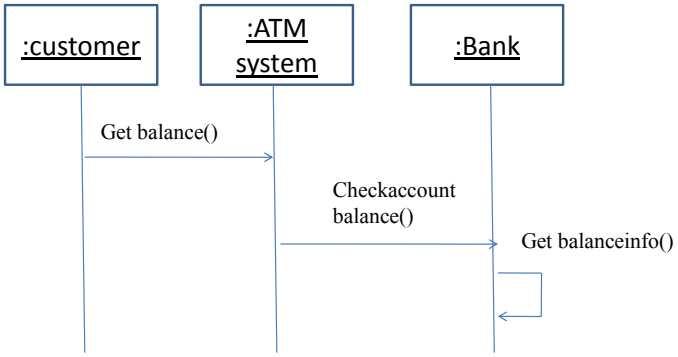
Discussion

- Information Expert is a basic guiding principle used continuously in object design.
- Expert is not meant to be obscure or fancy idea; it expresses the common "intuition" that objects do things related to information they have.
- Fulfillment of a responsibility often requires information that is spread across different classes of objects.
- Whenever information is spread across different objects, they will need to interact via messages to share the work.

Example

- ATM application – to know the balance amt we need
- 1. the acc no of a customer whose balance amt must be known.
- 2. The transactions till date for the same customer.

This information must be available in a class bank.
Therefore by information expert the bank class will return the acc no and balance details.



Contraindications

- There are situations where the solution suggested by expert is undesirable, usually because of problems in coupling and high cohesion.
- Keep the application logic in one place, database in another place and so forth, rather than intermingling different system concerns in the same component.

Benefits

1. It supports Information encapsulation because objects poses their information in order to perform the responsibilities.
2. it supports **low coupling** – so the system becomes more maintainable.
3. The behavior in distributed across the classes that need the related information. This helps in designing lightweight classes.
4. **High cohesion** is usually supported.

Related patterns 1. **low coupling 2. High cohesion**

Low Coupling

- Coupling is a measure of how strongly one object is connected to, has knowledge of, or relies on other object.
- An element with low coupling is not dependent upon too many objects.
- The classes with high coupling rely upon many classes and it leads to various problems.

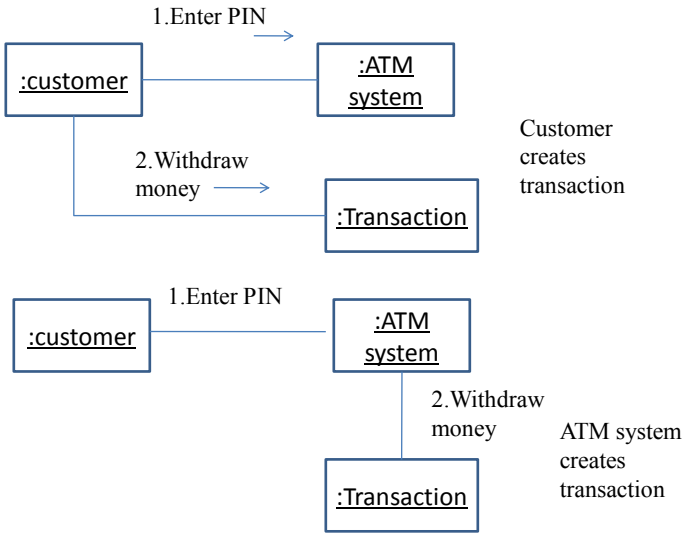
Low Coupling Pattern

Problem:

- How to reduce the impact of changes?
- How to understand the isolated object?
- How to reuse the object which is dependent on another object?

Solution:

- Assign a responsibility to objects in such a way that coupling between objects remains low.



Customer is related to ATMsystem class, and ATMsystem class performs the transaction leads to low coupling.

Discussion

- Low Coupling is a principle to keep in mind during all design decisions;
- It is an underlying goal to continually consider.

It is evaluative principle that a designer applies while evaluating all design decisions

Contraindications & Benefits

- (High coupling to stable elements and to pervasive elements is seldom a problem.)
- **Benefits of low coupling:**
- Impact of changes can be reduced.
- Objects are simple to understand in isolation
- Objects are convenient to reuse

Connections between Patterns

- In the example given, Information Expert supports Low Coupling.
- Result of identifying patterns from practices that have developed over years of experience.
- Often get the same result by starting with a different idea, because the ideas (patterns) support each other.

Controller

- A Controller is a non–user interface object responsible for receiving or handling a system event. It defines methods for system operation.
- When user generates some event using the UI objects then there is a first object beyond the UI layer who handles these events. This object is called controller.

Problem :

- Who should be responsible for handling an input system event?
- This assumes the Model-View-Controller architectural (not software) pattern that separates the user interface (Views) from the functionality of the rest of the system and uses controllers to access other functionality (Models).

Solution

- Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
  - Represent the overall system or root object or a device, or subsystem.
  - Represents a use case scenario within which the system event occurs. Such controller is called handler or coordinator or session.

Discussion

- Controller pattern provides guidance for generally accepted, suitable choices.
- It is desirable to use the same controller class for all the system events of one use case so that it is possible to maintain information about the state of the use case in the controller.

Monopoly Example

- The first system message is "Play Game." What object should receive this message?
- The author suggests that :MonopolyGame might be a suitable object if there are a limited number of system operations. Otherwise, we would have to consider High Cohesion if there are many operations.

Facade Controller

- Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
  - Represent the overall system or root object or a device, or subsystem.
  - These choices of controllers are called façade controller.

Use–Case controller

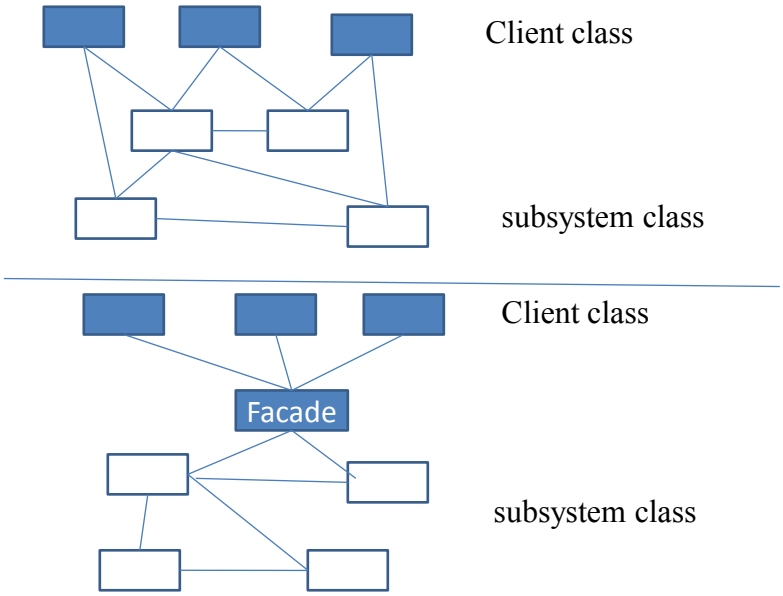
- When placing the responsibilities in a façade controller leads to design with low cohesion or high coupling, Use Case Controller is preferred as an alternative.
- When there are many system events across different processes;it factors their handling into manageable separate classes.

Benefits

- by delegating the system operations responsibilities to the controller supports the concept of reusability.
- If the responsibility of controller is handled in an interface object then it is possible to develop the pluggable interfaces.
- for occurring the system operations in some specific sequence use of controller pattern Is a reasonable choice.

Related Patterns

- In a message handling system, each message is handled separate command object.
- Façade – façade pattern is an object that provides the interface in a large sub system. When the client class wants to communicate to the subsystem classes then instead of having direct communication, the client class communicates with the façade interface .
- This interface delegates the request to the subsystem.



10

Layers & pure fabrication

- According to layer pattern each layer contains a specific logic. Ex UI layer contains the user interface logic.
- This a GRAPS pattern which does not represent the concept in the problem domain.
- It is an arbitrary creation of designer for achieving low coupling, high cohesion and reusability.

High Cohesion

- Cohesion is a measure of how strongly related and focused are the responsibilities of an element.
- High Cohesion: An object with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion.

High Cohesion Pattern

- **Problem:**
  - How can complexity be kept manageable?
- **Solution:**
  - Assign responsibility so that cohesion remains high.

A class with low cohesion may carry many unrelated things. It might face the following problems.

- these classes are hard to understand , harder to reuse .
- hard to maintain.

Discussion

Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to consider contantly

Monopoly Example

- Assuming, from the previous Monopoly slide, that :MonopolyGame is the controller:
- If :MonopolyGame performs most of the game functions within the software object, you have low cohesion.
- If the functions are delegated to other software objects, you have high cohesion.

Back to relationships

- Low Coupling and High Cohesion go together naturally. One usually leads to the other.
- Both are outcomes of the OO principle of "chunking," breaking up a large problem into manageable pieces.
- If the pieces are too simple there will be too many of them.
- If the pieces are too complex, each will not be understandable.

Cohesion examples

- Some scenarios illustrating varying degrees of varying degrees of functional cohesion:
- Very Low Cohesion – A class is solely responsible for many different functional areas.
- Low Cohesion – A class has sole responsibility for a complex task in one functional area.
- Moderate Cohesion – A class has light weight and sole responsibilities in a few different areas that are logically related to the class concept, but not to each other.
- High Cohesion – A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

Contraindications

- Grouping of responsibilities or code into one class or component to simplify maintenance by one person.
- Distributed server objects.
- It is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations.
- This approach is called coarse grained remote interface.

Benefits

- High cohesion pattern often supports Low coupling.
- Clarity and ease of comprehension of design is increased.
- Maintenance , modifications and enhancements are simplified.

An object classification schema

- The following analysis class types are used by Ivar Jacobsen in Objectory:
  - **Boundary objects** are abstractions of interfaces
  - **Entity objects** are application-independent domain software objects
  - **Control objects** are use case handlers
- Jacobsen was a contributor to UML, and the controller pattern reflects his approach.
- Increased potential for reuse, and pluggable interfaces.
- Provides information about the state of a use case.

Object Design and CRC Cards

- Another tool used to help assign responsibilities are CRC cards.
- CRC stands for Class-Responsibility-Collaborator
- They are index cards, one of each class, upon which the responsibilities of the class are briefly written, and a list of collaborator objects to fulfill these responsibilities.

Customer	
Responsibilities:	*Collaborators:*
Track work orders assigned to the customer	WorkOrder

WorkOrder	
Responsibilities:	*Collaborators:*
Track charge slips associated with the work order	ChargeSlip

Inventory	
Responsibilities:	*Collaborators:*
Manage a collection of parts with information specific to the boatyard	Part

ChargeSlip	
Responsibilities:	*Collaborators:*
Track charges Parts Additional Charges	ChargeSlip Part Charge

Part	
Responsibilities:	*Collaborators:*
Has a vendor cost, internal cost, quantity, and tax flag and belongs in inventory	Inventory Vendor
The part is associated with a work order when being purchases	WorkOrder PurchaseOrder
Parts go on charge slips	ChargeSlip
Parts are purchased from vendors	Vendor VendorPart

PurchaseOrder	
Responsibilities:	*Collaborators:*
Assign parts to a purchase order	Part
Reconcile the purchase order with the vendor's invoice	VendorInvoice Vendor
When the PO is closed, the parts are automatically billed to a charge slip	Customer WorkOrder ChargeSlip Part

Charge	
Responsibilities:	*Collaborators:*
Manage additional charges on a charge slip	ChargeSlip

Vendor	
Responsibilities:	*Collaborators:*
Manage Vendors	
Track parts associated with a particular vendor	Part

Invoice	
Responsibilities:	*Collaborators:*
Track additional charges	Charge
An invoice is associated with a purchase order	PurchaseOrder

Summary

- The skillful assignment of responsibilities is extremely important in object design. Determining the assignment of responsibilities often occurs during the creation of interaction diagrams , and certainly during programming.
- Patterns are named problem/solution pairs that codify good advice and principles often related to assignment of responsibilities.

Design Pattern
Designing for Visibility

Visibility

- Visibility: the ability of an object to "see" or have a reference to another object.
  - A resource (such as an instance) is within the scope of another.
  - Allows a sender object to send a message to a receiver object.
  - Example – the getproductDescription message sent from a **Register** to a product catalog means that **Productcatalog** instance is visible to **Register.**

Types of Visibility

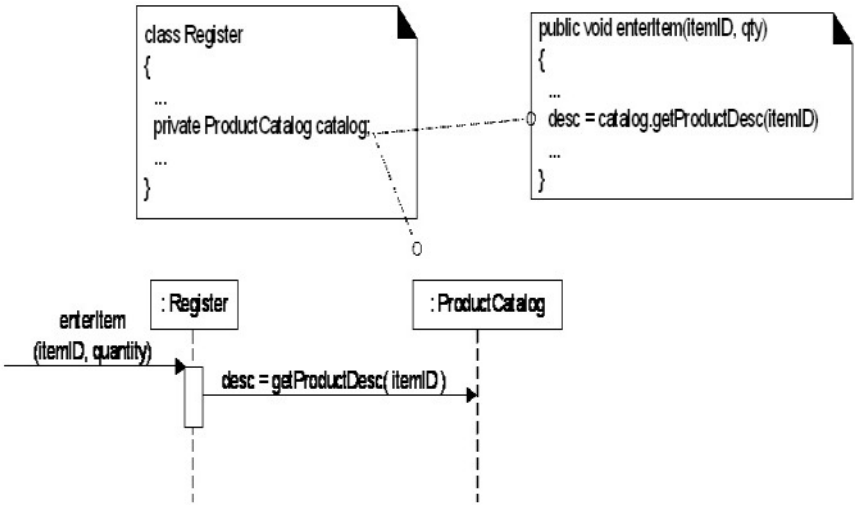
4 types of visibilities from object A to B are

- **Attribute visibility** – B is an attribute of A.
- **Parameter visibility** – B is a parameter of a method of A.
- **Local visibility** – B is a local object in a method of A (not a parameter).
- **Global visibility** – B is in some way globally visible.

Attribute visibility

- When obj2 is an attribute of obj1 then attribute visibility occurs.
- This kind of visibility is relatively permanent as long as obj1 and obj2 exists.
- Ex – consider POS system.

Attribute visibility



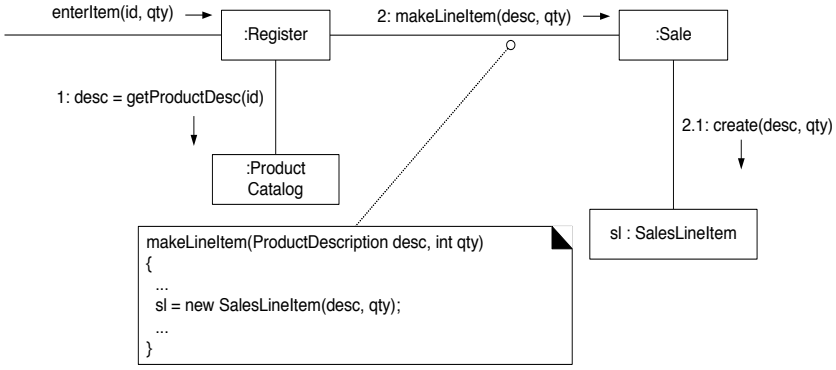
Register has attribute visibility to Productcatalog, because it is an attribute to Register.

Parameter visibility

- When obj2 is passed as a parameter to the method of obj1 then parameter visibility occurs.
- This is temporary visibility because it remains only within the scope of the method of obj1.

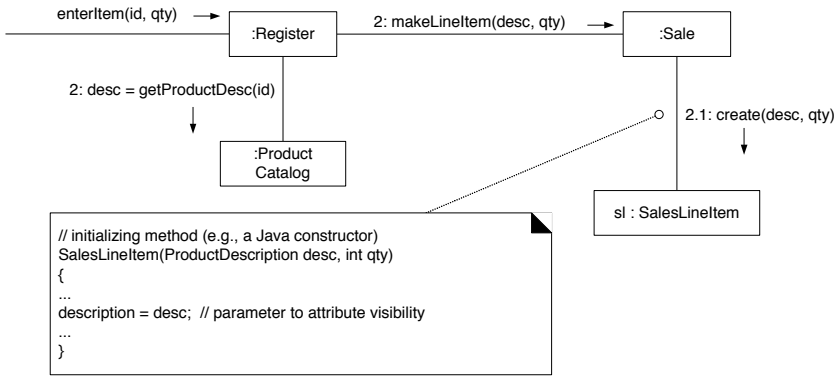
Parameter visibility

Within the scope of this method , object has the visibility



When makeLineITem is sent to a Sale instance, a ProductDescription instance is passed as a parameter.

Parameter to attribute visibility



- Parameter visibility is often transformed into attribute visibility by passing the parameter to a constructor when creating a new object.

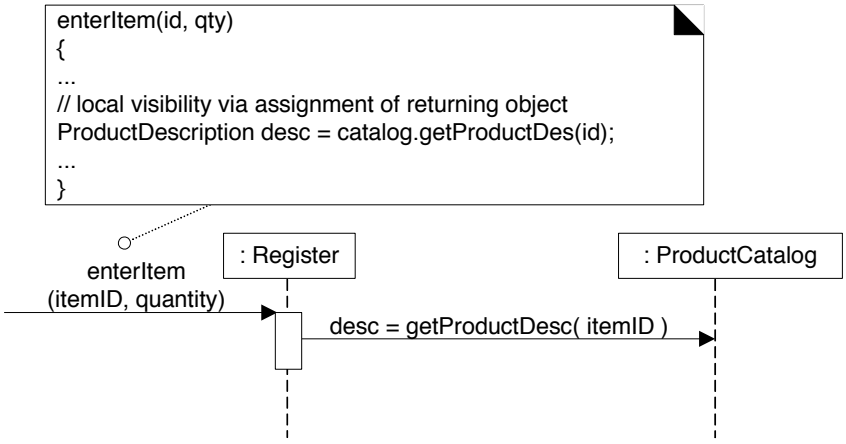
Local Visibility

- Two common means of achieving local visibility:
  - Create a new local instance and assign it to a local variable.
  - Assign the returning object from a method invocation to a local variable.
    - Returned object can also be anonymous:

anObject.getFoo().doBar();

- Like parameter visibility, commonly transformed to attribute visibility.
- It persists only within the scope of method

Local visibility



Global visibility

- When obj2 is global to obj1 the global visibility occurs from obj1 to obj2.
- For achieving the global visibility
  1. an instance is assigned to the global variable
  2. use of singleton pattern.