Software Engineering

Software Architecture for Enterprise Information Systems

Guido Menkhaus and Emilia Coste
Software Research Lab, University of Salzburg

References

References

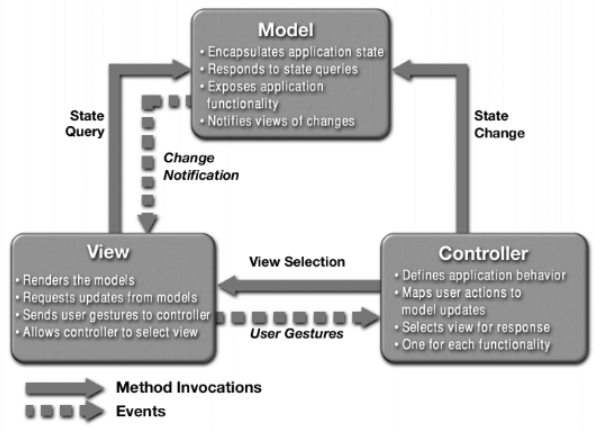
- **Floyd Marinescu,** *EJB Design Patterns*, John Wiley & Son, 2002
- **E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design,** *Design Patterns, Addison Wesley*, 1995
- **Sun,** *J2EE Patterns Catalog,* http://java.sun.com/blueprints/patterns/catalog.html
- **Leszek A. Maciaszek,** *Data Management in Designing Enterprise Information Systems,* http://www.comp.mq.edu.au/~leszek/ind_courses/

Overview

- Model - View - Controller
- Design Objectives
- Enterprise Information System Multi-Tier Architecture
- Design Principles
- Design Patterns
- Software Metrics

3

MVC



4

Design Objectives

- **A hierarchical layering / tiering of software modules that**
  - reduces complexity and
  - enhances understandability of module dependencies

  by disallowing direct object intercommunication between non-neighboring layers / tiers.

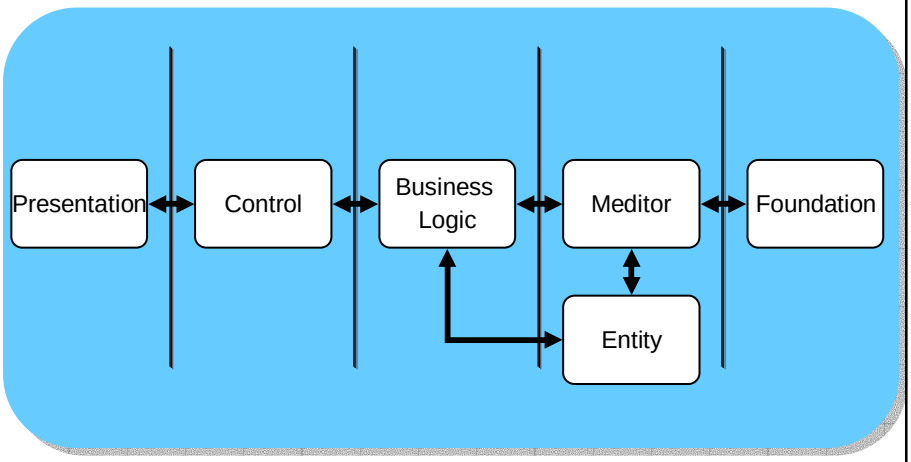
5

Design Objectives revisited

- **Architectural design is an exercise in managing module dependencies**
  - Module A depends on module B if changes to module B may necessitate changes to module A (→Open-Close Principle)
- **It is important that dependencies do not cross dependency firewalls**
  - In particular, dependencies should not propagate across non-neighboring layers / tiers and must not create cycles

6

EIS Multi-Tier Architecture



7

Multi-Tier Application Explained I

- **Presentation**
  - Classes that define UI objects. The presentation renders the contents of the application.
- **Control**
  - The controller translates interactions with the presentation into actions to be performed by the application logic.
  - In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests.
  - The actions performed include activating business processes. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

8

Multi-Tier Application Explained II

- **Application Logic**
  - The application logic represents the business rules that govern access to and updates of the data. Often the application logic serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the application logic.
- **Mediator**
  - Creates a level of independence between entity and foundation and between application logic and foundation
- **Entity**
  - Classes representing „business objects“
- **Foundation**
  - Responsible for all communication with the persistent data store

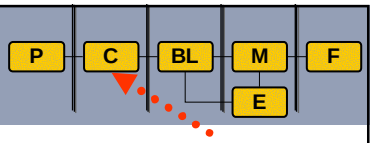
9

Principles

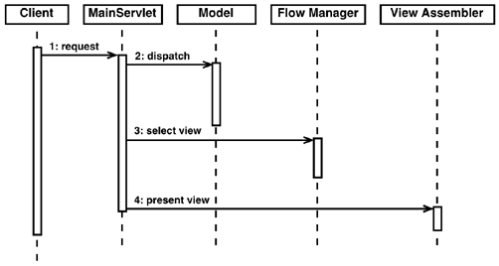
- **Downward (left-to-right) Dependency**
- **Upward (right-to-left) Notification**
- **Neighbor Communication**
- **Cycle Elimination**
- **Class Naming**

10

Front Controler

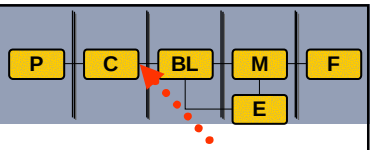


- The *Front Controller* pattern defines a single component that is responsible for processing application requests.
- A front controller centralizes functions such as view selection, security, and templating, and applies them consistently across all pages or views. Consequently, when the behavior of these functions need to change, only a small part of the application needs to be changed: the controller and its helper classes.

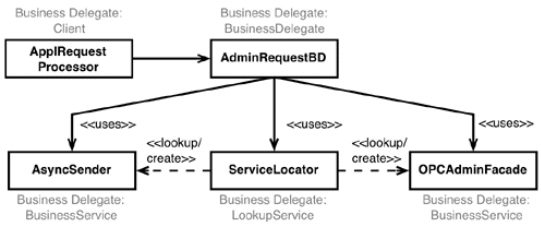


11

Business Delegate

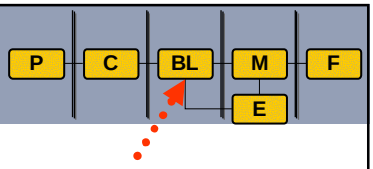


- In distributed applications, lookup and exception handling for remote business components can be complex. When applications use business components directly, application code must change to reflect changes in business component APIs.
- How can an intermediary between the presentation and the business logic be created to facilitate decoupling the presentation from the application logic tier ?



12

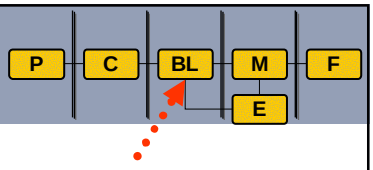
Observer



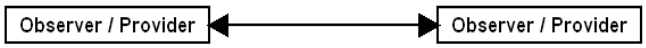
- ***Abstract coupling between Subject and Observer.***
  - *All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.*
- ***Unexpected updates.***
  - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

13

Observer Variation I

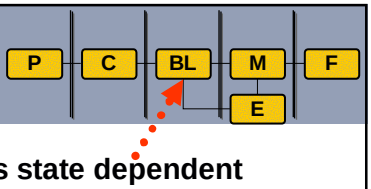


- **If both Components are Provider and Observer at the same time, they are mutual Provider and Observer for each other.**
- **Mutual Provider / Observer behavioral patterns require attention in implementation to prevent livelock runtime conditions.**

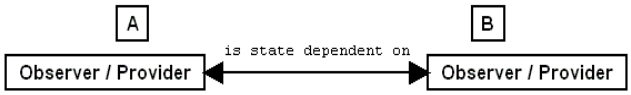


14

Observer Variation II

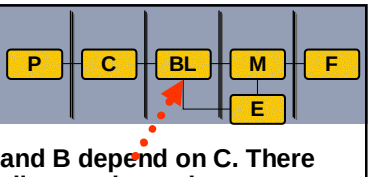


- A is state dependent on B and B is state dependent on A.
- There is a cycle in the dependency graph. In this case the Observer pattern is not helpful. The application might hang, since A and B might call each other's update method recursively.

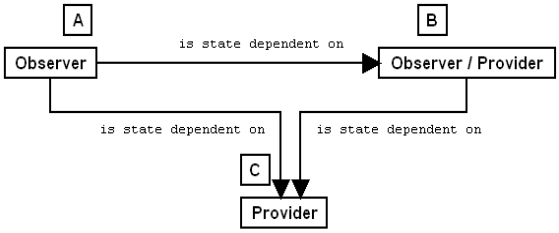


15

Observer Variations III

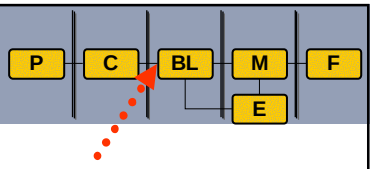


- Object A depends on object B and both A and B depend on C. There is a cycle in the dependency graph. Depending on the update strategy, this could result in incorrect results and / or redundancy updates as well.
- For example, if C changes and A is notified first and the subject A changes its state as part of the update implementation, A will receive an update call twice. There is no general solution to the problem of cylces in the dependency graph. Sophisticated Change Managers are not often useful, because they cannot be used in many situations and better and cheaper specialized solutions can frequently be found.

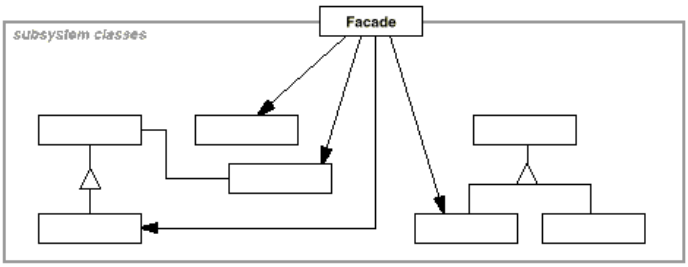


16

Facade

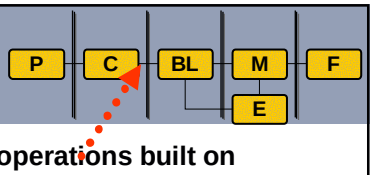


- How can the controler tier execute a use case‘s business logic in one transaction ?

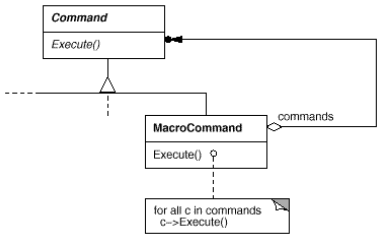


17

Command

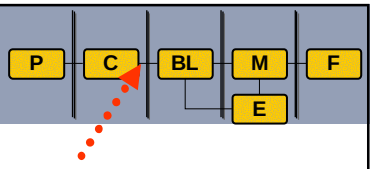


- Structure a system around high-level operations built on primitives operations.
- Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions

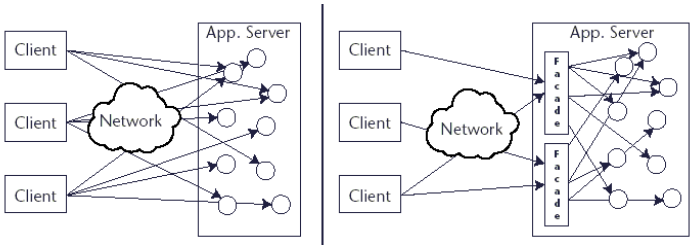


18

Session Facade

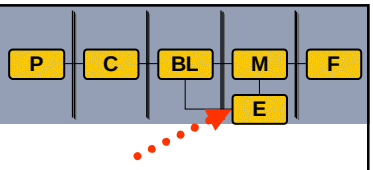


- Clean and strict separation of business logic from presentation tier.
- Low coupling
- Good reusability
- Good maintainability



19

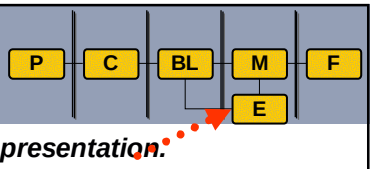
Entity Factory



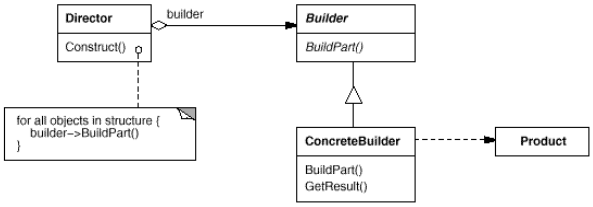
- How should entity object creation logic be implemented, in order to minimize the impact of frequent changes in the entity tier on the rest of the system ?
- Place the responsibility for creating entitiy objects in a entity object factory !

20

Builder

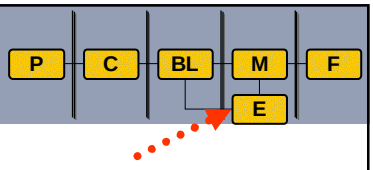


- ***It lets you vary a product's internal representation.***
  - The Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled. Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder.
- ***It isolates code for construction and representation.***
  - The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface.

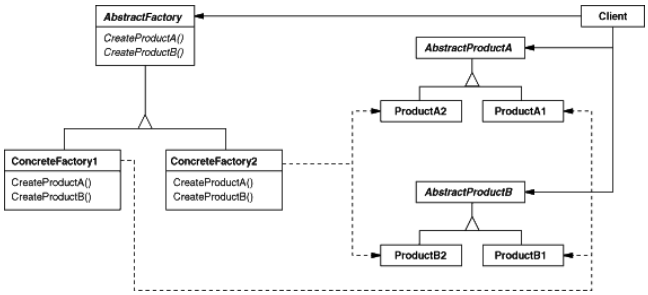


21

Abstract Factory

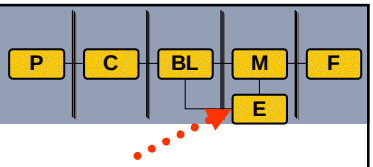


- **Provide an interface for creating families of related or dependent objects without specifying their concrete classes.**

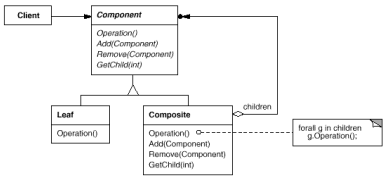


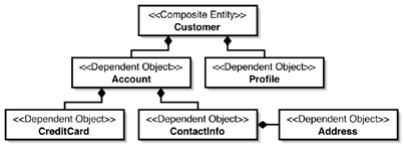
22

Composite Entity



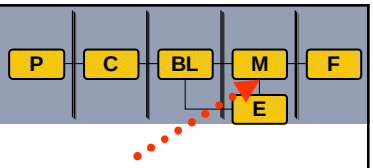
- **The *Composite Entity* design pattern offers a solution to modeling a networks of interrelated business entities. The composite entity's interface is coarse-grained, and it manages interactions between fine-grained objects internally. This design pattern is especially useful for efficiently managing relationships to dependent objects**



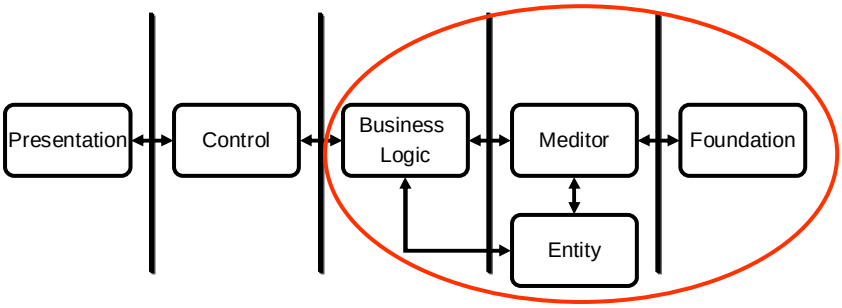


23

Mediator

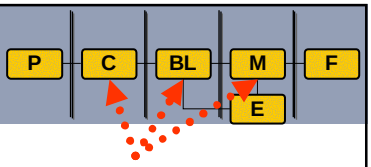


- ***It simplifies object protocols.***
  - A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain, and extend.
- ***It centralizes control.***
  - The Mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.

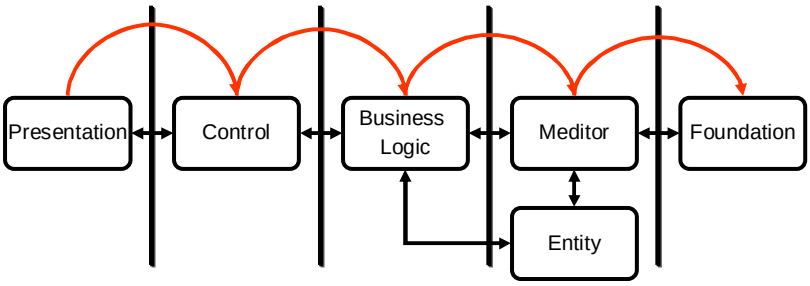


24

Chain

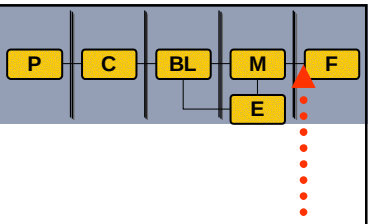


- When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

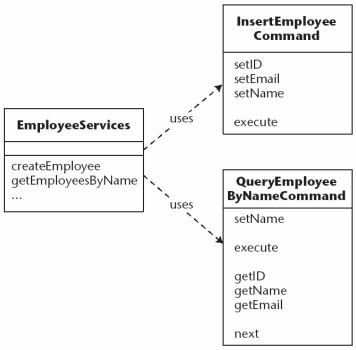


25

Data Acess Command I

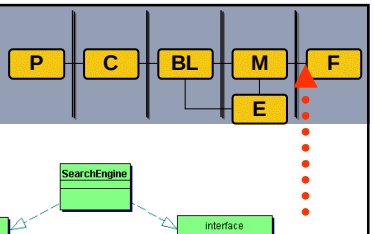


- How can persistence logic and persistent store be decoupled and encapsulated away from business logic ?
- Encapsulate persistence logic into data access command objects, which decouple business logic from all persistence logic !

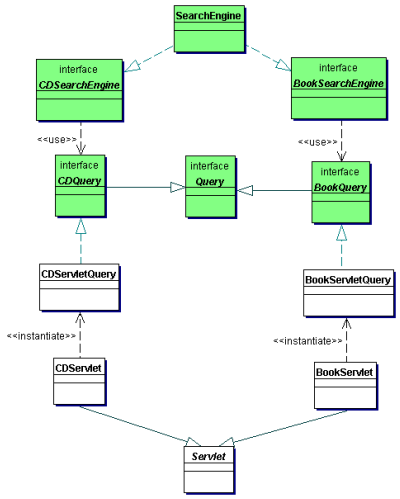


26

Data Access Command II

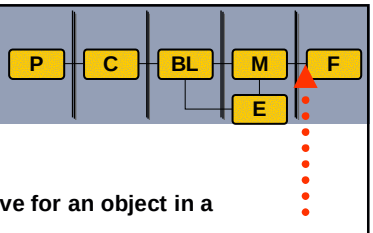


- **Clients should not be forced to depend upon interfaces that they do not use.**

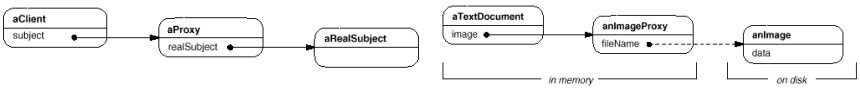


27

Proxy



- **A remote proxy provides a local representative for an object in a different address space.**
- **A virtual proxy creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.**
- **A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.**
- **A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed.**
  - counting the number of references
  - loading a persistent object into memory when it's first referenced.
  - checking that the real object is locked before it's accessed to ensure that no other object can change it.



28

Proactive and reactive Software Development

- **Architectural design takes an proactive approach to managing dependencies in software**
  - This is a **forward-engineering** approach – from design to implementation
  - The aim is to deliver software design that minimizes dependecnies by an architectural solutions to developers
- **Proactive approach must be supported by the reactive approach that aims at measuring dependencies in implemented software**
  - This is the **reverse-engineering** approach – from implementation to design
  - The implementation may or may not conform to the desired architextural design

29

Cohesion Of Methods

- **Metric 5: (LOCOM*) Lack Of Cohesion Of Methods (The definition of this metric was proposed by Henderson-Sellers in 1995)**
  - Measures the dissimilarity of methods in a class by attributes.
  - Consider a set of m methods, M_1, M_2, ... , M_m
  - The methods access a data attributes, A_1, A_2, ... , A_a
    - Let $m(A_k)$ = number of methods that access data A_k
  - Viewpoints:
    - Low value indicates good class subdivision implying simplicity and high reusability.
    - High lack of cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a} m(A_j)\right) - m}{1 - m}$$

30

Cohesion Of Methods

- **LCOM***
  - If each method accesses all attributes then m(Ak) = m so
  - At maximum cohesion LCOM* = 0

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a} m\right) - m}{1 - m} = \frac{(m) - m}{1 - m} = 0$$

- **LCOM***
  - If each method accesses only one attribute and a different attribute then we have:
  - At "minimum cohesion" LCOM* = 1

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j}^{a} 1\right) - m}{1 - m} = \frac{\left(\frac{1}{a}a\right) - m}{1 - m} = 1$$

31

Coupling between Objects

- **Metric 6: (CBO) Coupling between Objects (Chidamber & Kemerers(1994) modified the definition of CBO)**
  - CBO for a class is a count of the number of related couples with other classes. Represents the number of other classes to which a class is coupled
  - The Fan-out of a class, C, is the number of other classes that are referenced in C
  - A reference to another class, A, is a reference to a method or a data member of class A
  - In the fan-out of a class multiple accesses are counted as one access
  - The Fan-in of a class, C, is the number of other classes that reference in C
  - Definition CBO = fan-out of a class
- **CBO counts the number of reference types that are used in attribute declarations, formal parameters, return types, throws declarations and local variables, and types from which attribute and method selections are made. Primitive types, types from java.lang package and supertypes are not counted.**

32

Coupling between Objects

- **Viewpoints:**
  - High fan-outs represent class coupling to other classes/objects and thus are undesirable
  - High fan-ins represent good object designs and high level of reuse
  - It does not seem possible to maintain high fan-in and low fan outs across the entire system
  - Excessive coupling between objects is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.
  - A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

33