

"Package diagrams are a key tool for me in maintaining control over a system's overall architecture."
Martin Fowler

Lecture 10 Software Architecture

Lecturer: Tommy Yuan
Fall 2008 - Year 2

Object-Oriented Methods

Outline

- ◆ Extremes, abstractions and packages
- ◆ Software architectures
- ◆ Architecture styles

2

Extremes: one object

- ◆ You can have one object that does everything
 - a doItAll object
 - a monolithic solution
- ◆ Many attributes and operations are all bundled together
- ◆ A lump of spaghetti difficult to understand and to maintain
- ◆ All you can talk about are the system, attributes, and methods

3

Extremes: too many objects

- ◆ Everything is 'decomposed'
- ◆ Objects have perhaps one attribute and one domain method
- ◆ Still a lump of spaghetti
- ◆ All you can talk about are the system, attributes, and methods

4

In-between extremes lies good design

- ◆ Elusive, no "silver bullets"
- ◆ Iterate (do not just accept first-cut)
- ◆ **Robustness** to change is an excellent criteria of goodness
 - but is difficult to measure
- ◆ You want to be able to talk about lifts, doors, buttons, etc.
 - better levels of abstraction

5

Abstractions

- ◆ Levels at which you operate mentally
- ◆ Do you prefer to think about Java byte-code or Java objects?

6

Universal abstractions

- ◆ Universe
- ◆ Great clusters of galaxies
- ◆ A galaxy and its satellites
- ◆ Spiral arms
- ◆ Star groups
- ◆ A star



<http://www.janis.or.jp/users/kitahara/galaxy.html>

7

10s, 100s, 1000s of classes?

- ◆ Even with well designed classes software engineers need higher level abstractions to help:
 - break down a large system into smaller sub-systems
 - discuss overall design architecture
 - manage complexity

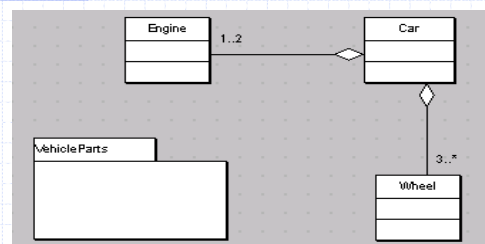
8

UML packages

- ◆ A general-purpose mechanism for organising elements into related groups
 - for example, grouping classes together
- ◆ A package forms a namespace
 - within a namespace, two different things cannot have the same name
- ◆ Do not group elements arbitrarily
- ◆ Well structured packages are **very cohesive** and **loosely coupled**

9

You can put classes into packages



The classes Engine and Wheel can be kept in the package VehicleParts. We can talk about "VehicleParts".

10

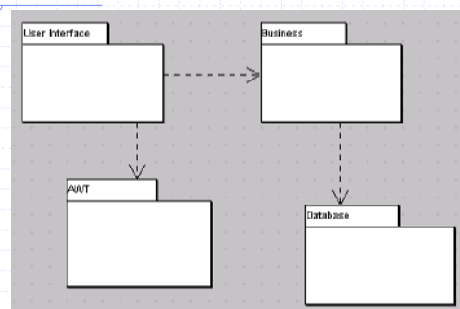
Dependencies (Martin Fowler)

"A dependency exists between two elements if changes to the definition of one element may cause changes to the other."

"A dependency between two packages exists if any dependency exists between any two classes in the packages."

11

A package diagram

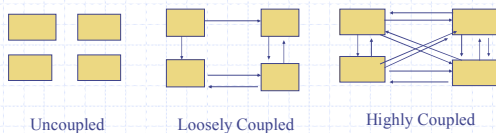


software architecture

12

Minimizing Coupling

- ◆ When a class changes, all classes which depend upon it generally need to be changed or recompiled
- ◆ More independent, more **robust** to changes



13

Encouraging Cohesion

- ◆ Group types (interface and classes) which fulfill a similar purpose, service or function.
- ◆ Internal package coupling, or relational cohesion can be quantified
 - $RC = \text{NumberOfInternalRelations} / \text{NumberOfTypes}$
 - The bigger of RC, the higher cohesion
 - Most useful for packages contain some implementation classes
 - Rarely needed
 - Less applicable to packages of most interfaces

14

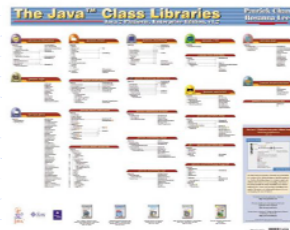
Java platform API

packages include:

- java.applet
- java.awt
- java.io
- java.math

```
import java.io.*;
import java.awt.*;
```

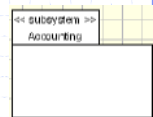
package statement in Java to organise classes into packages



15

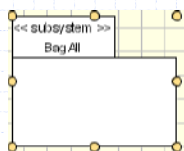
Package vs Sub-system

- ◆ Packages are just a general purpose mechanism for organizing elements into groups.
 - E.g. java.util package is not a subsystem
- ◆ Subsystems have clearly defined responsibilities, behavior, and interfaces that do work.
- ◆ A subsystem is flagged with a <<subsystem>> stereotype on a package



16

Sub-systems



- ◆ Should be cohesive
 - Avoid BagAll sub-systems which provide a collection of disparate services (coincidental cohesion)
- ◆ Should be as loosely coupled as possible
 - reduce dependencies and avoid large amounts of information passing between subsystems

17

High level design

Software Architecture

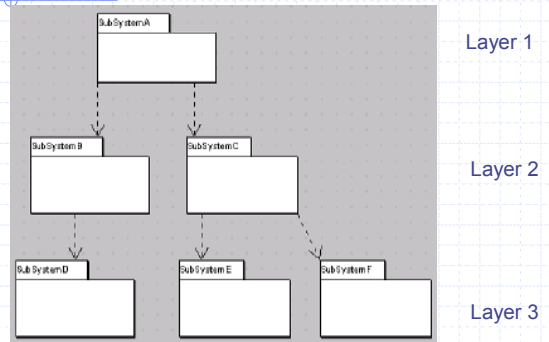
18

Issues in software architecting

- ◆ System decomposition
 - layering and partitioning
- ◆ Communication protocols between subsystems
 - Strong encapsulation
 - via an API (eg. operation signatures)
- ◆ Overall control flow
 - event driven?

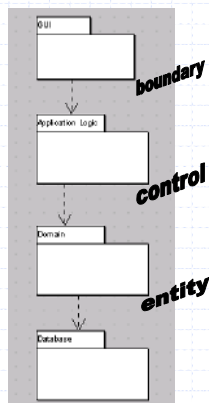
19

Layering and partitioning



20

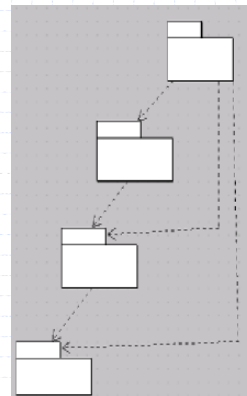
Closed architectures



- ◆ Each layer depends only on the layer below it
- ◆ But each layer introduces a speed & storage overhead
- ◆ 3-5 layers are typical

21

Open architectures



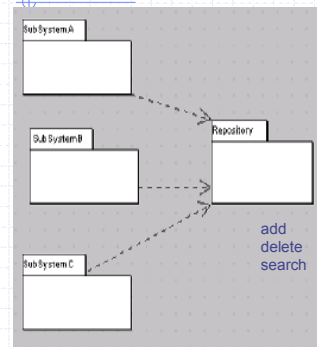
Can bypass layers and avoid performance bottlenecks.

trade-off

But suffer the penalty of increased coupling due to extra dependencies.

22

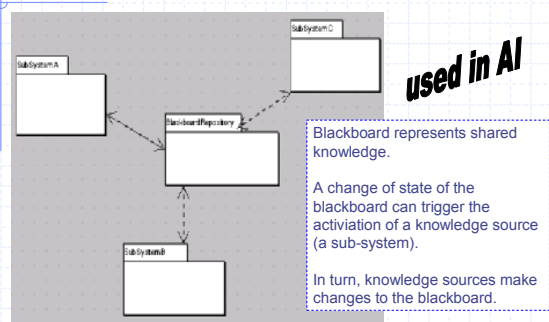
Repository architectures



e.g. DBMS
 Easy to add subsystems.
 But Repository can become a bottleneck.
 - queue requests

23

Blackboard repository architectures

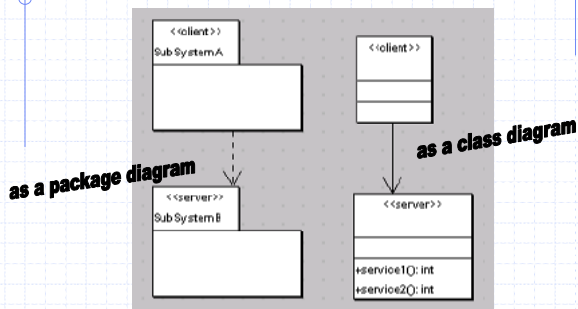


used in AI

Blackboard represents shared knowledge.
 A change of state of the blackboard can trigger the activation of a knowledge source (a sub-system).
 In turn, knowledge sources make changes to the blackboard.

<http://www.nb.net/~javadou/bb.htm>

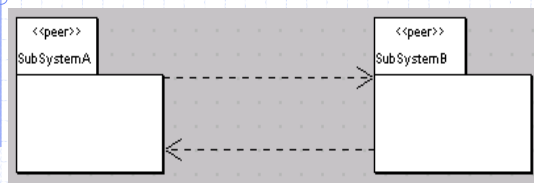
Client-server architectures



Client needs to know where the server is.

25

Peer to peer architectures

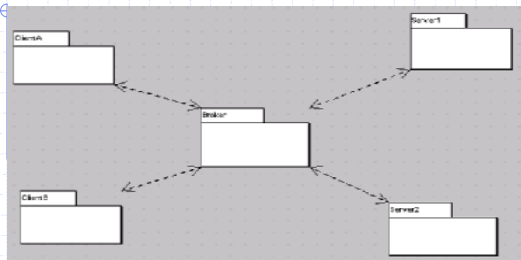


More coupling and control flow hazards, so more difficult to implement and to maintain.

Mechanism for inter-process communication?
Agent communication, maybe!

26

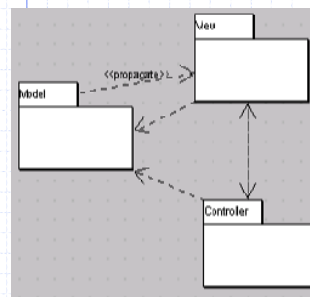
Broker architectures



Client does not need to know where the server is.
Distributed systems?
Multi-agents e-commerce system, maybe! JADE

27

Model View Controller (MVC) architectures



Model is independent of views and controllers.
- only needs to say "I've changed"

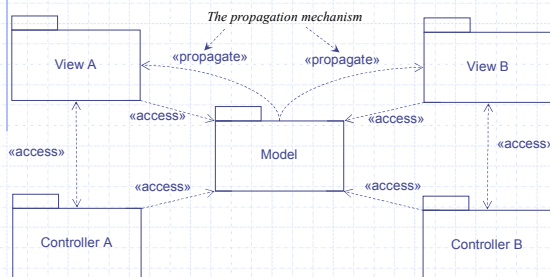
Multiple views are possible.



Subscribe/notify protocol needed to allow Views to update when Model changes.

28

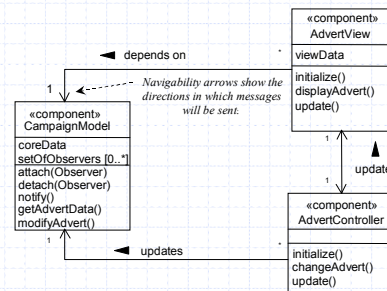
Model-View-Controller



(Adapted from Hopkins and Horan, 1995)

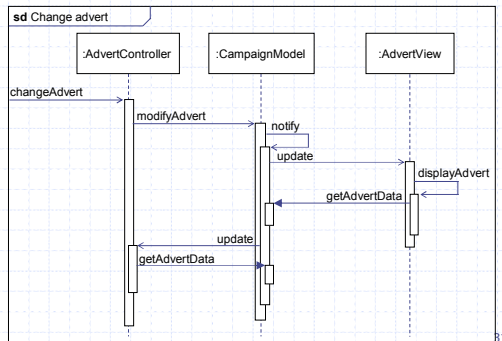
29

MVC applied to Adgate



30

MVC Component Interaction



MVC in Java

◆ Observer and Observable

An introduction to the Observer interface and Observable class using the Model/View/Controller architecture as a guide

<http://www.javaworld.com/javaworld/jw-10-1996/jw-10-howto.html>

32

Reading

◆ Text chapter 12

33