Chapter 9

Architectural Design

© Pearson Education Limited 2005

Topics

- Architectural layers and dependency management
  - Architectural modules
  - Dependencies between various objects
  - Interfaces
  - Event processing
  - Acquaintance

- Architectural frameworks
  - Model-View-Controller (MVC)
  - Presentation-Control-Mediator-Entity-Foundation (PCMEF)

- Architectural patterns
What is software architecture?

- **Software architecture** is the organization of software elements into a system aiming at addressing various concerns:
  - **Fundamental concern** – supportability (i.e. understandability + maintainability + scalability) of the software solution
  - Other (consequential) concerns:
    - managing object interdependencies and trade-offs
    - providing for generation and evaluation of (measuring) alternative solutions
    - organization of specific software modules (classes, packages, components)
    - assignment of behaviors to modules
    - determination of architectural patterns and principles

- **Architectural design** is the set of decisions aiming at efficient and effective software architecture together with the rationale for these decisions

Size and complexity

- **Legacy systems**
  - Monolithic, processing sequential and predictable
  - Complexity = size

- **Object systems**
  - Distributed, processing random and unpredictable
  - Complexity in wires

Facade pattern
Object systems → new legacy systems?

- Unsupportable system → legacy system
- Supportability = understandability + maintainability + scalability
- Properties of complex systems that are supportable:
  - Take the form of hierarchy and composition of objects
  - Intra-linkages of components stronger than inter-linkages
  - Dynamic links legalized as static associations
  - Complex systems that work are result of simple systems that worked (evolution)

Architectural design objectives

- **hierarchical layering** of software modules that reduces complexity and enhances understandability of module dependencies by disallowing direct object intercommunication between non-neighboring layers, and
- enforcement of programming standards that make module **dependencies** visible in compile-time program structures and that forbid muddy programming solutions utilizing just run-time program structures
Main points

- Measurably-supportable systems
- Supportable system → dependency metrics
- Module (object) A depends on module B if changes to module B may necessitate changes to module A
- Architecture (hierarchy of object layers) that minimizes (potential) dependencies
  - Dependencies must not cross dependency firewalls (should not propagate across non-neighboring layers and must not create cycles)
- Dependencies on classes, messages, events, inheritance
- Proactive approach (architecture → implementation) and reactive approach (implementation → architecture)
- Two aims of reactive approach:
  - Conformance to the architecture
  - Comparison of different implementations
- The issue of project management and availability of managerial tools

Design classes

- Business objects ≈
  - entity classes
  - domain classes
  - conceptual classes

- Design classes ≈
  - software classes
  - application classes
  - program classes
  - system classes
  - implementation classes
**Packages**

- **Package (UML)** is a grouping of modeling elements under an assigned name
  - package may contain other packages
  - package owns its members (elements) – removing the package from the model removes also its members
    - member (usually a class) can belong to one package only
  - package may have package imports to other packages
    - package A or element of package A can refer to package B or to its elements
    - class is owned by only one package but it can be imported to other packages

---

**Package notation**

- dependency relationship
  - A depends on B
- B owns X

- circle-plus notation
  - E owns F
  - F owns Y and Z
  - Class Y
  - Class Z

- package with no members revealed
- C owns D
Cycles between packages

Eliminating cycles between packages

- Package A
- Package B
- Package C
- Package D
- Package E
- Package A2
- Package C2

Circularly-dependable elements of Package A extracted into Package A2
Circularly-dependable elements of Package C extracted into Package C2
Layer dependencies

- Higher layers depend on lower layers
- Lower layers are required to be stable (any changes to them may have a ripple effect on higher layers)

Class dependencies

Layer 1 depends on Layer 2 because Class X depends on Class Y

Layer 1 depends on Layer 2 because Class X depends on Class Z

Package A depends on Package B because Class X depends on Class Y
Inheritance

- **Implementation inheritance** is a means of structural and behavioral sharing between a base class (or superclass) and its derived classes (subclasses), such that runtime service invocations can be given a subclass object in place of a superclass object
  - **polymorphism** - getting different behavior, depending on an object that happens to be servicing the request
    - polymorphic behavior is provided by **method overriding**
    - overriding ≠ **method overloading** (when a class provides a number of methods with the same name but different set of signatures)
  - **dynamic binding** (late binding) - the invocation of an appropriate method, conditional on the instantiated class at runtime

Compile-time inheritance dependencies

```
public void do1(){
    //do something
    do2();
}

public void do4(){
    //do something
    do1();
}
```

```
Object

A

\(\text{do1()}\)
\(\text{do2()}\)
\(\text{do1()}\)

B

\(\text{do2()}\)
\(\text{do3()}\)

C

\(\text{do4()}\)

```

wait()
Run-time inheritance dependencies

Inheritance without polymorphism

```java
public void doTest()
{
    Object o = new A();
    ...
    o.wait();
    ...
}
```

```java
public void do3()
{
    // do some stuff
    do2();
}
```
**Extension inheritance**

```java
public void do2()
   //do some stuff
   do3();
```

**Down-calls**

```java
public class B {
   public void do3() {
      //do some stuff
   }
}
```

```java
public class A {
   public void do2() {
   }
}
```

```java
public class X {
   public void doX() {
      A myA;
      public X(){
         myA = new B();
      }
      public void doX(){
         myA.do2();
      }
   }
}
```
Up-calls

```java
public void do3(){
//do some stuff
//refers to the parent's impl
super.do2();
}
```

Method dependencies

```java
public class CActioner {
  EEmployee emp;
  public void do1() {
    emp.do3();
  }
}

public class EOutMessage {
  EEmployee emp;
  public void do2() {
    emp.do3();
  }
}
```
In the UML 2.0, **interface** is a declaration of a set of features that is not directly instantiable, i.e. no objects of it can be directly created.

- The object that implements interface provides “a public façade that conforms to the interface specification” (UML, 2002, p.123).
- In UML 2.0 interface can declare attributes, not just operations.
  - By contrast, in Java an interface can contain data members but they must be constants (defined as static and final).

An **abstract class** is a class that contains at least one method, which is not (or cannot be) implemented by that class, and therefore it cannot be instantiated.

- A class is a class is a class. In languages that support only single implementation inheritance, like Java, a class can only extend one base class (abstract or concrete), but it can implement multiple interfaces. This is a huge practical difference.
- The related difference is that interfaces allow passing objects typed as interfaces in method calls.

---

### Interface notations

- `<Interface>`
- `Interface1`
- `Interface2`
- `Interface3`
- `AbstractClass1`
- `Class1`
**Implementation dependency**

```java
public interface Interface1 {
    private int a1;
    public void o1();
}

public class Class1 implements Interface1, Interface2 {
    public void o1() {
        //implementation code
    }
    public void o2() {
        //implementation code
    }
}
```

**Usage dependency**

This is permitted in UML 2.0, but not in Java, which only allows extension inheritance between interfaces.

```java
public class Class1 {
    Interface1 myInterface;
    public void do1() {
        myInterface.o1();
    }
}
```
cyclic dependencies, between classes and other structures (methods, packages, subsystems) unavoidable, but can be neutralized
- extra classes to reduce a network of calls to a hierarchy
- purposeful use of interfaces
Event processing

- Synchronous messages need to be considered separately from asynchronous communication where methods are “fired” to service asynchronous events.
- In event processing there is a separation between an event originator (publisher object) and various event listeners/observers (subscriber objects) that want to be informed of an event occurrence and take their own, presumably different, actions.
- In large systems, a separate registrator object performs the subscription, i.e. the “handshaking” between the publisher and subscribers.
- Usually, the publisher object creates an event object – the publisher translates the intended meaning of the event into an event object (called something like BCommandButtonEvent).
- The event object is passed (in a callback operation) to all subscriber objects that registered their interests in the mouse click on the button.

Dependencies in event processing

- In a callback, the publisher has no knowledge or interest in how the subscriber processes the event. The dependency exists but it is negligible from the viewpoint of the architectural design.
- The hand-shaking of subscribers and publishers causes a stronger dependency.
  - If a registrator object mediates the hand-shaking, then it depends on both the publisher and the subscriber.
  - If a subscriber object registers itself, then it depends on the publisher.
  - To loosen dependencies due to hand-shaking, subscribers can be passed to the registration methods in arguments typed as interfaces.
**Event processing – upward notification**

- upward communication that minimizes object dependencies
- lower layers rely on interfaces and event processing (publisher/subscriber protocols) to communicate with objects in higher layers

**Acquaintance**

- **Acquaintance** defines a situation when an object is passed another object in an argument to its method.
- More precisely, an object A gets acquainted with object B if another object C passes B to A in an argument of the message to A.
- Object communication due to acquaintance is one of the programming techniques legitimized in the Law of Demeter.
**Acquaintance package**

- separate layer of interfaces to support more complex object communication under strict supportability guidelines
- subsystem of interfaces only other objects in the system can use these interfaces, and pass them in arguments to method calls, instead of concrete objects → classes in non-neighboring subsystems can communicate without knowing the concrete suppliers of services (and, therefore, without creating dependencies on concrete classes).

**Model-View-Controller (MVC)**

- **Model objects** represent data objects – the business entities and the business rules in the application domain
- **View objects** represent GUI objects and present the state of the model in the format required by the user, typically on a graphic display
- **Controller objects** represent mouse and keyboard events (processing logic)
PCMEF architecture

Core J2EE tiers

Client Tier
- applications, UI presentation
- user interaction

Presentation Tier
- servlets, JSP
- session management, content management, format and delivery

Business Tier
- EJB
- business logic, transactions

Integration Tier
- JDBC, JMS, Connectors, Legacy resource adapters, external systems, rules engines, workflow

Resource Tier
- databases, external systems resources, data and external services

PCMEF layers

<<layer>> presentation

<<layer>> control

<<layer>> domain

<<layer>> foundation

...converting to PCMEF design
PCMEF layers (subsystems)

- **The presentation subsystem**
  - classes that handle the graphical user interface (GUI) and assist in human-computer interactions.

- **The control subsystem**
  - classes capable to understand what program logic is
    - searching for information in entity objects
    - asking the mediator layer to bring entity objects to memory from the database.

- **The entity subsystem**
  - manages business objects currently in memory
  - container classes
  - containers are linked

- **The mediator subsystem**
  - mediates between entity and foundation subsystems to ensure that control gets access to business objects
  - manages the memory cache and synchronizes the states of business objects between memory and the database

- **The foundation subsystem**
  - classes that know how to talk to the database
  - produces SQL to read and modify the database

PCMEF principles

1. **Downward Dependency Principle (DDP)**
2. **Upward Notification Principle (UNP)**
3. **Neighbor Communication Principle (NCP)**
4. **Explicit Association Principle (EAP)**
5. **Cycle Elimination Principle (CEP)**
6. **Class Naming Principle (CNP)**
7. **Acquaintance Package Principle (APP)**
CNP, NCP, EAP

- **CNP – class naming**
  - Name of each class and each interface in the system should identify the subsystem/package layer to which it belongs.
  - Ensuring that each class begins with a single letter identifying the PCMEF layer (i.e., P, C, etc.).
    - EVideo means that the class is in the entity subsystem.
    - IMVideo means that the interface is in the mediator subsystem.

- **NCP – neighbor communication**
  - Objects can communicate across layers only by using direct neighbors.
  - Chains of message passing.

- **EAP – explicit association**
  - Legitimizes run-time object communication in compile-time data structures.

CNP, NCP, EAP, DDP

- **CNP – class naming**
  - Name of each class and each interface in the system should identify the subsystem/package layer to which it belongs.
  - Ensuring that each class begins with a single letter identifying the PCMEF layer (i.e., P, C, etc.).
    - EVideo means that the class is in the entity subsystem.
    - IMVideo means that the interface is in the mediator subsystem.

- **NCP – neighbor communication**
  - Objects can communicate across layers only by using direct neighbors.
  - Chains of message passing.

- **EAP – explicit association**
  - Legitimizes run-time object communication in compile-time data structures.

- **DDP – downward dependency**
  - Higher PCMEF layers depend on lower layers.
  - Lower layers should be designed to be more stable.
**DDP, CEP**

- **DDP – downward dependency**
  - higher PCMEF layers depend on lower layers
  - lower layers should be designed to be more stable
    - difficult to change, but (paradoxically?) not necessarily difficult to extend

- **CEP – cycle elimination**
  - ensures that circular dependencies between layers, between packages and between classes within packages are broken
  - cycles can be resolved:
    - by creating a new package specifically to eliminate the cycle
    - by forcing one of the communication paths in the cycle to communicate via interface

---

**Acquaintance**

- **Acquaintance** defines a situation when an object is passed another object in an argument to its method.
- More precisely, an object A gets acquainted with object B if another object C passes B to A in an argument of the message to A.
- Object communication due to acquaintances is one of the programming techniques legitimized in the Law of Demeter.
### Event processing

- Synchronous messages need to be considered separately from **asynchronous communication** where methods are “fired” to service asynchronous events.
- In event processing there is a separation between an event originator (**publisher object**) and various event listeners/observers (**subscriber objects**) that want to be informed of an event occurrence and take their own, presumably different, actions.
- In large systems, a separate **registrator object** performs the subscription, i.e. the “handshaking” between the publisher and subscribers.
- Usually, the publisher object creates an **event object** – the publisher translates the intended meaning of the event into an event object (called something like BCommandButtonEvent).
- The event object is passed (in a callback operation) to all subscriber objects that registered their interests in the mouse click on the button.
Dependencies in event processing

- In a callback, the publisher has no knowledge or interest in how the subscriber processes the event. The dependency exists but it is negligible from the viewpoint of the architectural design.
- The hand-shaking of subscribers and publishers causes a stronger dependency.
  - If a registrator object mediates the hand-shaking, then it depends on both the publisher and the subscriber.
  - If a subscriber object registers itself, then it depends on the publisher.
  - To loosen dependencies due to hand-shaking, subscribers can be passed to the registration methods in arguments typed as interfaces.

UNP – upward notification

- promotes low coupling in bottom-up communication between layers
- can be achieved by using asynchronous communication based on event processing
  - objects in higher layers act as subscribers (observers) to state changes in lower layers
  - when an object (publisher) in a lower layer changes its state, it sends notifications to its subscribers
  - in response, subscribers can communicate with the publisher (now in the downward direction) so that their states are synchronized with the state of the publisher
PCMEF patterns

- PCMEF architecture is based on some well-known design patterns and on few new patterns specific to PCMEF.
- Main source of patterns for PCMEF are:
  - GoF (Gang of Four – [GAMM1995]),
  - PEAA (Patterns of Enterprise Application Architecture – [FOWL2003]),
  - Core J2EE [ALUR2003].
- Patterns particularly useful include: MVC, Façade, Abstract Factory, Chain of Responsibility, Observer, Mediator, Identity Map, Data Mapper, Lazy Load, OID Proxy.

Facade

- Facade: a higher-level interface that makes the subsystem easier to use.
- EEntity is the facade to the entity package.
Abstract Factory

- to provide “an interface for creating families of related or dependent objects without specifying their concrete classes”

Chain of Responsibility

- to “avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request”
- delegation
Observer (for UNP – upward notification)

- To define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also known as the Publish-Subscribe pattern

Mediator

- Promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently
Summary

- **Software architecture** – organization of software elements into a system
  - uses hierarchical layering of software objects (design classes)
  - ensures that dependencies between objects are minimized
    - three levels of structural dependencies: layer dependencies, package dependencies, and class dependencies
    - method dependencies and event dependencies are behavioral dependencies in a program (that lead to structural dependencies)

- Combining event processing and interfaces creates the most powerful mechanism to facilitate dependency management in software architectures

- The textbook applies architectural framework called PCMEF

- Patterns that most prominently feature in the PCMEF framework are: Façade, Abstract Factory, Chain of Responsibility, Observer, and Mediator