Developing Maintainable Systems –
Roundtrip Engineering with Metrics

© L.A. Maciaszek, B.L. Liong & Addison-Wesley

This presentation is based on:


Topics

- Application design objectives - PCMEF
- Tutorial case study – Email Management System
- Dependency management
- Generic architectural patterns
- Architectural patterns for business objects
- The supportability metrics
- Conclusions

The tutorial themes

"Applications come and go. Databases stay for ever."
Bob Epstein, Sybase (quoted from memory)

"Enterprise applications often have complex data – and lots of it – to work on, together with business rules that fail all tests of logical reasoning."

"The architectural pattern I like the most is that of layers...."
Martin Fowler
(in "Patterns of Enterprise Information Architecture")

Application design objectives

- a hierarchical layering of software modules that reduces complexity and enhances understandability of module dependencies by disallowing direct object intercommunication between non-neighboring layers, and
- an 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

MVC

© L.A. Maciaszek, B.L. Liong & Addison-Wesley
PCMEF application packages

**PCMEF**

### Meaning of P-C-M-E-F
- **The Presentation package**
  - classes that define UI objects
- **The Control package**
  - classes responsible for the program’s logic, algorithmic solutions, and main computations
- **The Mediator package**
  - creates a level of independence between entity and foundation and between control and foundation
  - responsible for object caching
- **The Entity package**
  - classes representing “business objects”
- **The Foundation package**
  - responsible for all communications with the persistent data store

### Consider non-PCMEF design
- CControl
- EEntity
- PPresentation
- MMediator
- FFoundation

### ...and PCMEF design
- CControl
- MMediator
- PPresentation
- EEntity
- FFoundation

### PCMEF principles
- Downward Dependency Principle (DDP)
- Upward Notification Principle (UNP)
- Neighbor Communication Principle (NCP)
- Explicit Association Principle (EAP)
- Cycle Elimination Principle (CEP)
- Class Naming Principle (CNP)
- Acquaintance Package Principle (APP)
**Tutorial case study**

- **EM – Email Management system**
  - part of an Electronic Document Management System
  - capture, indexing, classification and retrieval of all email business correspondence
-**Composing and emailing of messages stored in the database and scheduled to be sent to contacts (customers)**
  - a Java application that accesses an Oracle database via JDBC
  - upon employee’s login to the database, the application can retrieve from the database and display the list of messages to be emailed to contacts
  - the employee can then email chosen messages
  - after a successful email, the database is updated accordingly
  - the employee can create, update, delete messages
  - messages can be sorted and filtered

**Display email messages**

**Update email message**

**Email received**

**Conceptual class model**

**Relational database model**

© L.A. Maciaszek, B.L. Liong & Addison-Wesley
Let's slow down and revisit...

- 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
  - It is important that dependencies do not cross dependency firewalls (Martin, 2003)
    - In particular, dependencies should not propagate across non-neighboring layers and must not create cycles

Proactive and reactive approach

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

Packages

- Eliminating cycles between packages

Cycles between packages

- Eliminating cycles between packages
Layer dependencies

Layer 1

Layer 2

Package A

Package B

Package C

Package D

Package E

Layer 1 depends on Layer 2

Class dependencies

Class X depends on Class Y

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

Method dependencies

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

public class EOutMessage {
    EEmployee emp;
    public void do2() {
        emp.do3();
    }
}

public class EEmployee {
    MBroker brk;
    public void do3() {
        brk.do3();
    }
}

public class MBroker {
    FUpdater upd;
    public void do3() {
        upd.do3();
    }
}

public class FUpdater {
    public void do3() {
        // code for do3
    }
}

Interfaces

- 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.

UML notations

<<Interface>>

Interface1

Interface2

Interface3

AbstractClass1

Class1
Implementation dependency

Interface1
a1 o1();

Interface2
a2 o2();

Class1
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

Class1
do1();

Interface1
a1 o1();

Interface2
a2 o2();

<<uses>>

public class Class1 {
  Interface1 myInterface;
  public void do1() {
    myInterface.o1();
  }
}

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

Cycle… and breaking it

CActioner
do4();

PDialogBox
do3();

CInit
do1();

PPrimaryWindow
\do2();

Presentation

Control

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.

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.

Dependencies in event processing
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.

Interfaces and acquaintance

Acquaintance

Façade pattern

Abstract Factory pattern

Chain of Responsibility pattern

Observer pattern

Mediator pattern

Class design for EM

Instantiation model
**Identity Map pattern**

```
public class EIdentityMap {
    private Map OIDToObj; // OID -> Object
    private Map msgPKToOID; // Message PK -> OID

    public EIdentityMap() {
        OIDToObj = new HashMap();
        msgPKToOID = new HashMap();
    }

    public IAContact findContact(int contactOID) {
        return (IAContact) OIDToObj.get(new Integer(contactOID));
    }

    public void registerContact(IEObjectID oidObject) {
        OIDToObj.put(new Integer(oidObject.getOID()), oidObject);
    }

    public void unregisterContact(IEObjectID oidContact) {
        OIDToObj.remove(new Integer(oidContact.getOID()));
    }
}
```

**Data Mapper pattern**

**Identity Map**
- Load – check-out
- Unload – check-in

**Lazy Load**
- Lazy Initialization
- Virtual Proxy
- OID proxy
  - Navigation in Identity Map
  - Navigation in Entity Classes

**Unit of Work**

**Email Message** interaction

---

© L.A. Maciaszek, B.L. Liong & Addison-Wesley 2003

© L.A. Maciaszek, B.L. Liong & Addison-Wesley 2003
Lazy Load pattern

- Two kinds of retrieval operations:
  - Identity load
  - Predicate load

- Three loading strategies:
  - Closure load
  - Flat load
  - N-levels load

- Three approaches to Lazy Load:
  - Lazy Initialization
  - Virtual Proxy
  - OID Proxy

Lazy Initialization

Method `getContact()` in `EOutMessage`

```java
public IAContact getContact() {
    if (contactOID == null)
        contact = MDataMapper.retrieveContact(contactID);
    return contact;
}
```

Virtual Proxy

- Interaction

OID Proxy

- Internal programming mechanisms of ensuring the identity of objects loaded to memory

- These are no replacement for OIDs assigned explicitly by the program

- A singleton class that maintains maps of OIDs to objects can replace proxy classes:
  - If knows if an entity object is loaded
  - Upon loading, the object is given OID and all its data members are initialized (including FK values)
  - OID-based association links are initialized to null if the linked object has not been loaded yet.

- Dirty/clean status of an entity object:
  - When first loaded, the entity object is flagged clean
  - If its data content gets out of sync with database, it is flagged dirty
  - The entity object knows its clean/dirty status at all times
Navigation in Identity Map

**EIdentityMap**
- OIDToObj : Map
- msgPKToOID : Map

**findOutmessage(outmsgOID)**
**findContact(contactOID)**
**registerContact(OID)**

**MDataMapper**
- OID : List

**EOutMessage**
- OID : int
- <<PK>> outmsgID : String
- contactOID : int
- <<FK>> contactID : String
- dirty : boolean

**EContact**
- OID : int
- dirty : boolean

**Presentation and control in EM**

**Navigation in Entity Classes - interaction**

**Domain in EM**

**Mediator and foundation in EM**
Calculation of CCD assumes adherence to the architectural framework.

If the framework is found to be broken, the CCD is calculated as if a class can depend on any other class in the system.

- **Probability theory method** - the combinations counting rule
  
  - The CCD is the number of different combinations of pairs of dependent classes which can be formed from the total number of classes in the design multiplied by 2 (cycles)

\[
CDD = \frac{n!}{2!(n-2)!} \times 2
\]

**DEFINITION:**
Cumulative Class Dependency (CCD) is the total supportability cost over all classes \(C_1, ..., C_n\) in a system of the number of classes \(C_j(j=1, ..., n)\) to be potentially changed in order to modify each class \(C_i\).

Consider the PCMEF design with five classes and that the CCD for it is also 5.

For a corresponding unsupportable system, the CCD would be 20:

\[
5 \cdot \frac{5!}{2!(5-2)!} \times 2 = \frac{120}{12} \times 2 = 20
\]

The UF is therefore \(20/5 = 4\).

The UF factor serves as a modifier of the more detailed metrics computed for designs/systems that were found to be unsupportable.

**UF**

**DEFINITION:**
Unsupportability Factor (UF) is the result of the division of the CCD for an unsupportable system by the CCD for a corresponding supportable system, i.e., the system that conforms to a supportable architectural framework, such as PCMEF.

Consider a class \(C\) that contains two methods \(m1\) and \(m2\).

Consider further that \(m1\) calls \(m2\) (as the only thing that it does).

If \(m2\) is an empty method, then MDC for class \(C\) is equal 1 (because \(m1\) depends on \(m2\)).

If, however, \(m2\) contained calls (messages) to two other methods \(m3\) and \(m4\) in supplier objects within the same package, then MDC for class \(C\) would be equal 3 (because \(m1\) depends on \(m2\), and \(m2\) depends on \(m3\) and \(m4\)).

If supplier objects in a neighborhood package serviced \(m3\) and \(m4\), then MDC for class \(C\) would be 5.

If supplier objects in a non-neighborhood package (according to the PCMEF framework) serviced \(m3\) and \(m4\), then MDC for class \(C\) would further increase by the UF value.

**CMD**

**DEFINITION:**
Cumulative Message Dependency (CMD) is the total supportability cost over all Synchronous Messages (SM) within client objects of the costs associated with changes to methods \(M\), in supplier objects or responsible delegator objects that are accountable for servicing \(SM\). When calculating CMD, the dependency value for offending (unsupportable) messages is increased by the Unsupportability Factor (UF).

- If a responsible delegator object delegates the work to an object in another package then the cost of inter-package dependency is carried by the responsible delegator.

- Further delegation sequence does not result in an additional cost (i.e., non-responsible delegators do not carry a maintainability cost).

**CMD - calculation example**

Consider a class \(C\) that contains two methods \(m1\) and \(m2\). Consider further that \(m1\) calls \(m2\) (as the only thing that it does).

If \(m2\) is an empty method, then MDC for class \(C\) is equal 1 (because \(m1\) depends on \(m2\)).

If, however, \(m2\) contained calls (messages) to two other methods \(m3\) and \(m4\) in supplier objects within the same package, then MDC for class \(C\) would be equal 3 (because \(m1\) depends on \(m2\), and \(m2\) depends on \(m3\) and \(m4\)).

If supplier objects in a neighborhood package serviced \(m3\) and \(m4\), then MDC for class \(C\) would be 5.

If supplier objects in a non-neighborhood package (according to the PCMEF framework) serviced \(m3\) and \(m4\), then MDC for class \(C\) would further increase by the UF value.

**CMD - supportable**
**CMD - unsupportable**

- MDI = 1
- MDI = 2
- MDI = 2

This is an unsupportable model: 
CMD = 1 + (2*3) = 7

public class CActioner {  // client of FUpdater
    FUpdater upd;
    public void do1() {  // MD = 2
        upd.do3();
    }
}

public class EEmailMessage { // client of EEmployee
    EEmployee emp;
    public void do2() {  // MD = 1
        emp.do3();
    }
}

public class EEmployee {  // responsible delegator
    FUpdater upd;
    public void do3() {  // MD = 0
        upd.do3();
    }
}

public class FUpdater {  // supplier
    public void do3() {  // MD is null
        // code for do3
    }
}

**CED**

**DEFINITION:** Cumulative Event Dependency (CED) is the total supportability cost over all methods containing “fire event” messages FEi plus over all methods containing “process event” messages PEi within publisher object plus over all methods servicing these “process events” SEi within subscriber object. The PEi supportability cost is associated with changes to signatures of SEi methods. The SEi supportability cost is associated with changes to messages in the bodies of PEi methods. Messages within registrator object as well messages contained in bodies of SEi methods are excluded as they are computed as part of the CMD calculation. When calculating CED, the dependency value for offending (unsupportable) events is increased by the Unsupportability Factor (UF).

**DQ tool**

**Pictorial summary**

© L.A. Maciaszek, B.L. Liong & Addison-Wesley
**Additional references**