

Dependency Management for Web-based EISs



Workshop on “Web-based EISs”
Sydney University, 20-May-2005

Leszek A. Maciaszek

Macquarie University, Sydney, Australia

<http://www.comp.mq.edu.au/~leszek/>

© L.A.Maciaszek

Topics & background reading

- *Introduction (the problem)*
- *Measurably-supportable systems*
- *Supportable system → dependency metrics*
- *Architecture (hierarchy) that minimizes (potential) dependencies*
- *Dependencies on classes, messages, events, inheritance*
- *Proactive approach (architecture → implementation) and reactive approach (implementation → architecture)*
- *The issue of project management and availability of managerial tools*
- *Conclusions*

MACIASZEK, L.A. and LIONG, B.L. (2005): ***Practical Software Engineering. A Case Study Approach*** Addison Wesley, Harlow England, 864p.

MACIASZEK, L.A. (2005): ***Requirements Analysis and System Design, 2nd ed.*** Addison Wesley, Harlow England, 504p.

Introduction (the problem)

- **Supportable EIS-s** (*legacy systems is an alternative*)
- System development is about **modeling** (*program is an executable model*) of:
 - *functional requirements*
 - *non-functional system qualities – as opposed to ‘fit-to-purpose’ (incl. **supportability**)*
- **Meta-architecture** – *necessary condition of supportability*
- *Architectural modeling must be **roundtrip***

Past, present, future

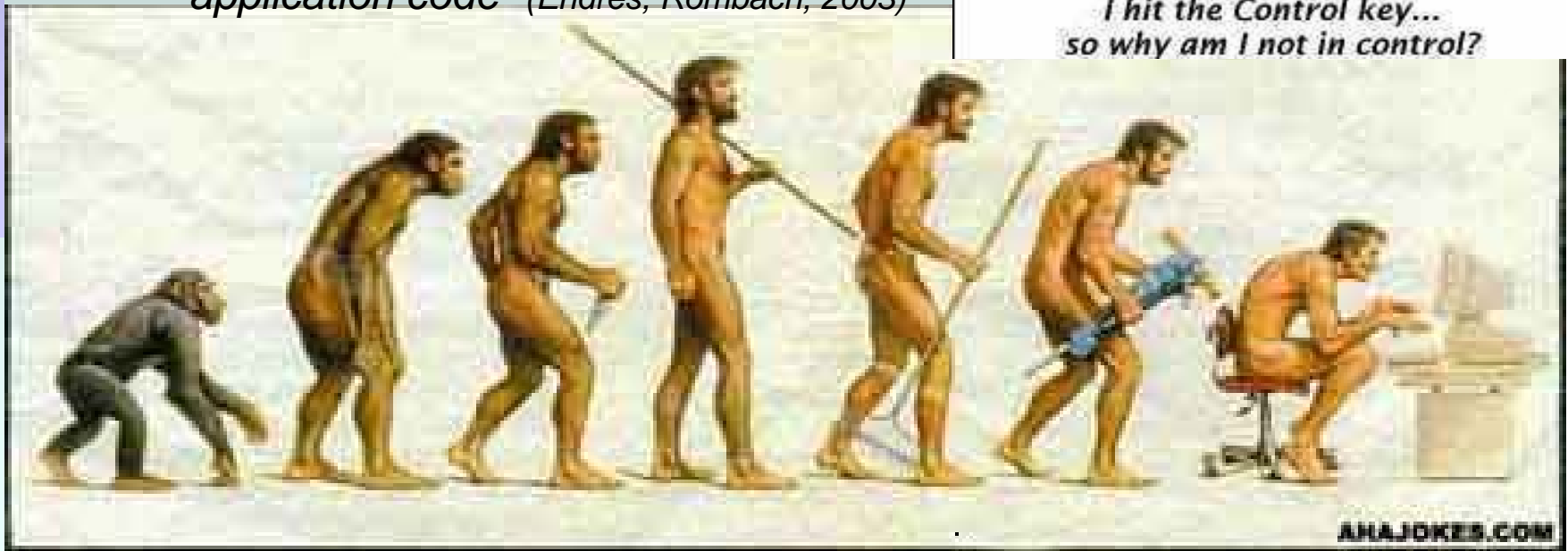
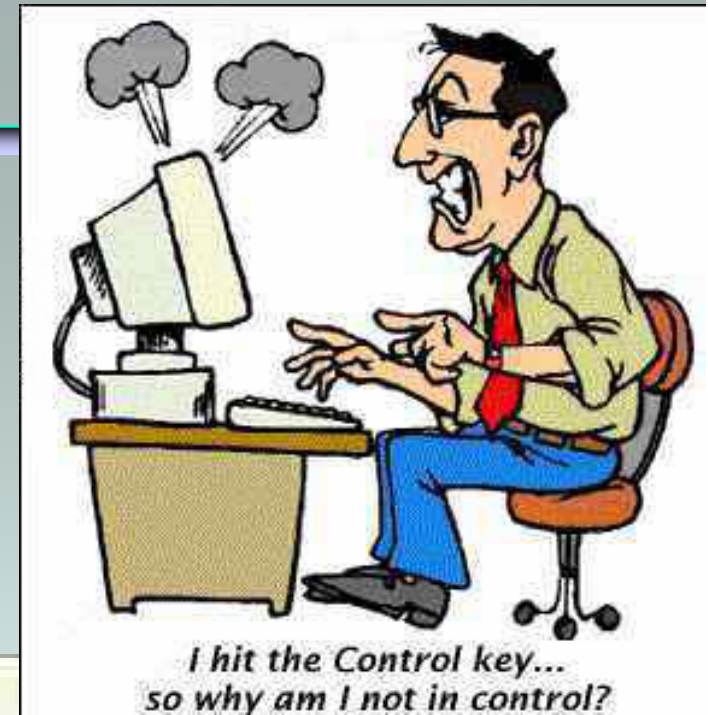
Legacy systems

- Monolithic, sequential and predictable
- Complexity = size


Object systems

- Distributed, user in control
- Complexity in wires

“cost of glue code is three times cost of application code” (Endres, Rombach, 2003)

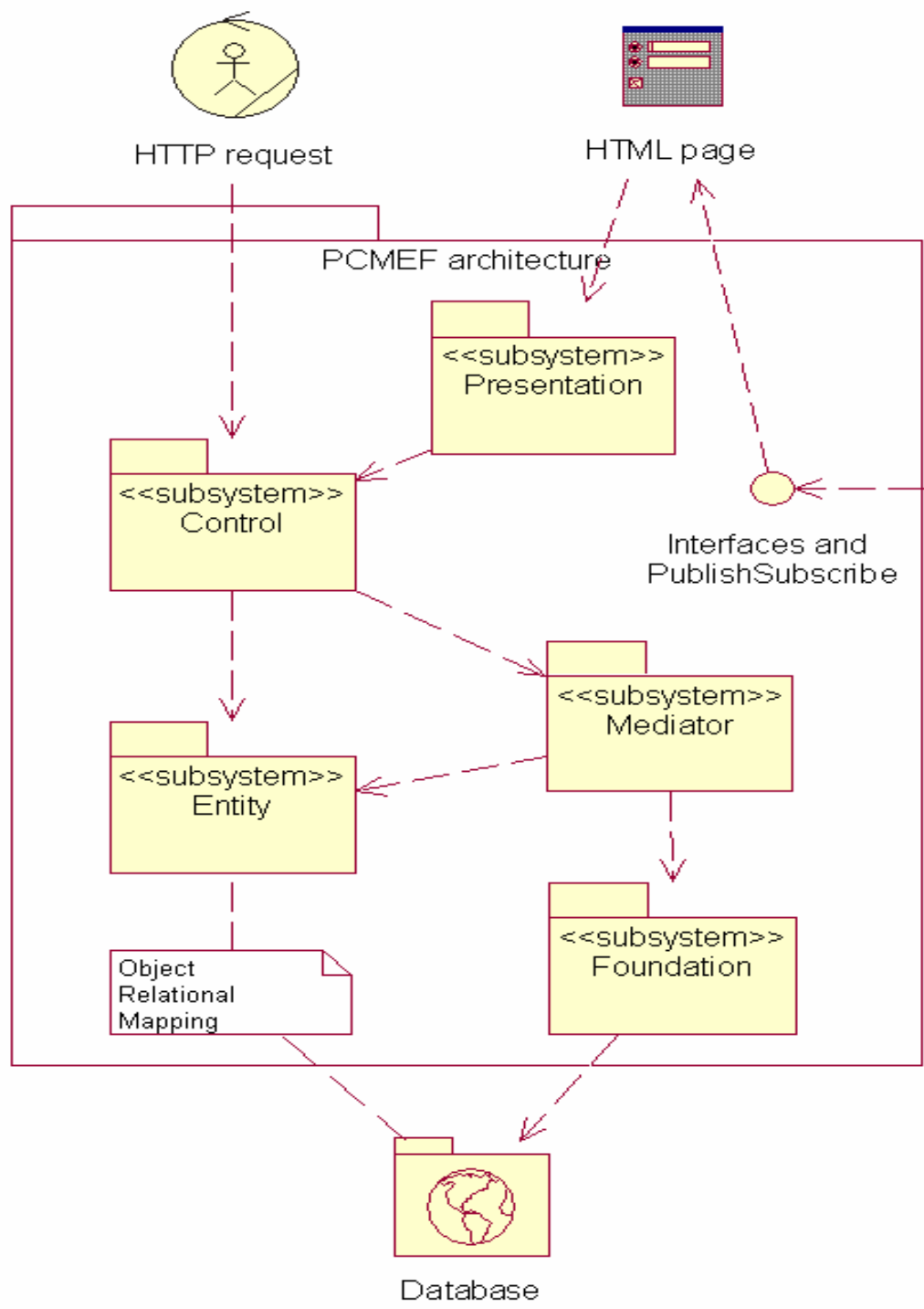


Object systems → new legacy systems?

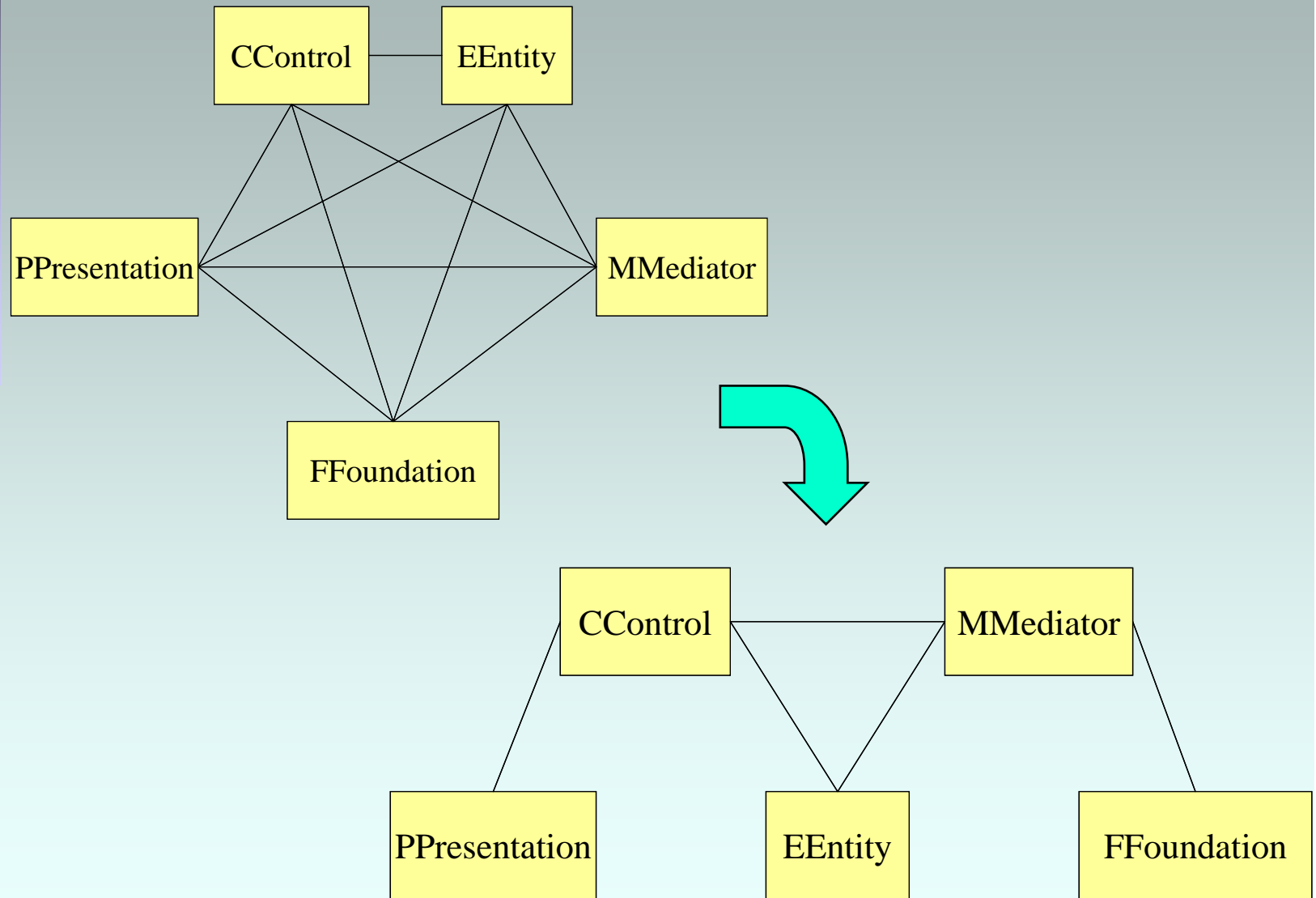
- **Supportability** = *understandability + maintainability + scalability*
- *Unsupportable system → legacy system*
- *Properties of complex systems that are supportable:*
 - *Take the form of  hierarchy and composition of objects*
 - *Intra-linkages of components stronger than inter-linkages*
 - *Dynamic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

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*



PCMEF



PCMEF 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*

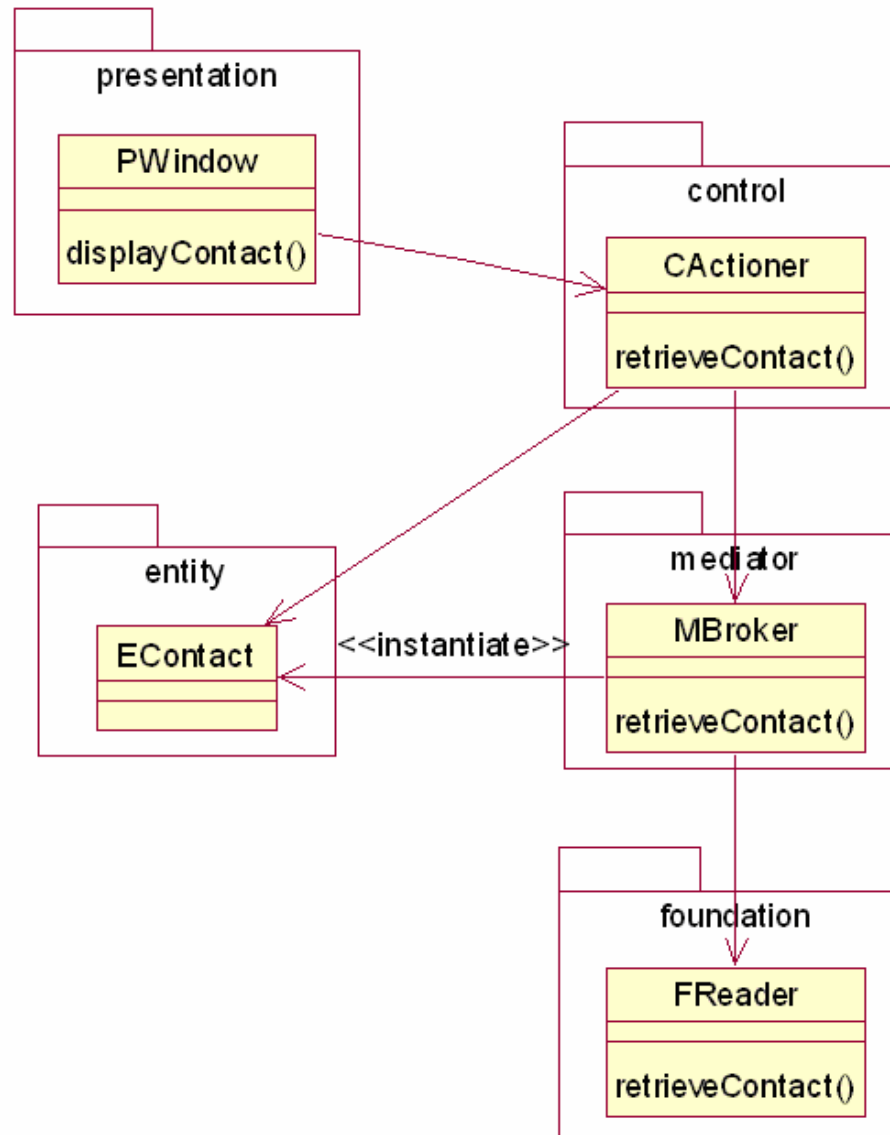
PCMEF principles

- *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)*

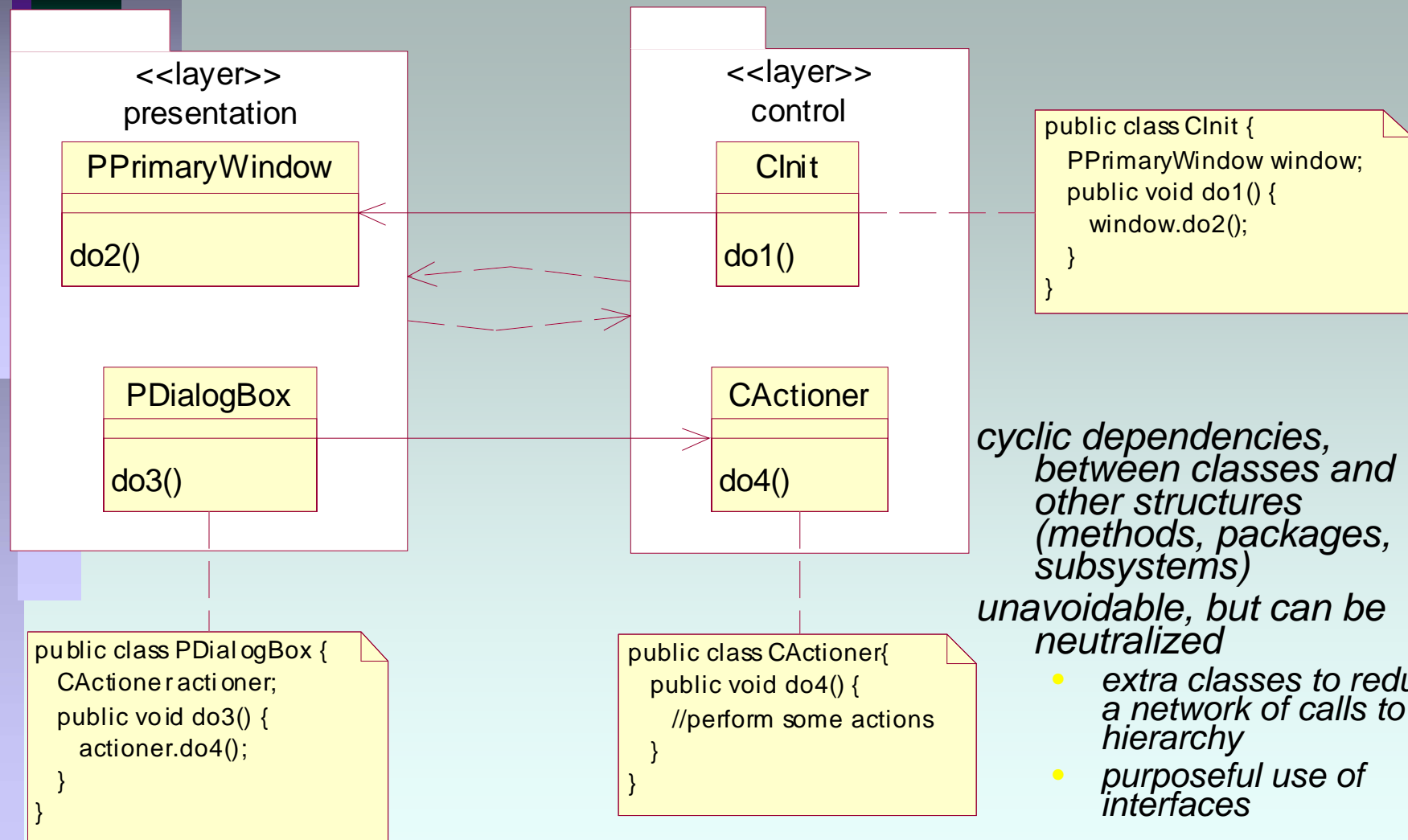
CNP, NCP, EAP, DDP

- *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*

Chain of responsibility pattern



CEP – cycle elimination

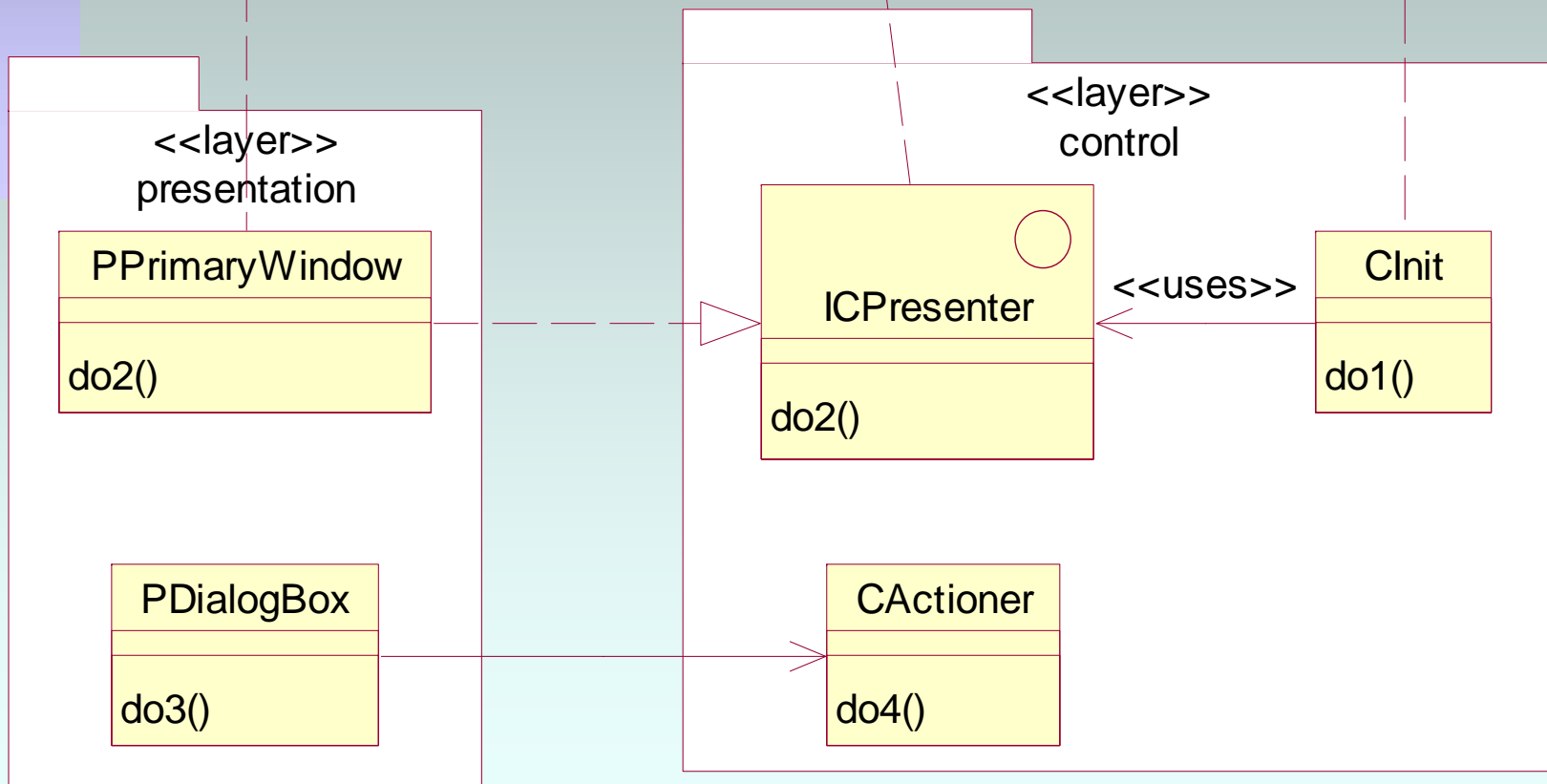


CEP

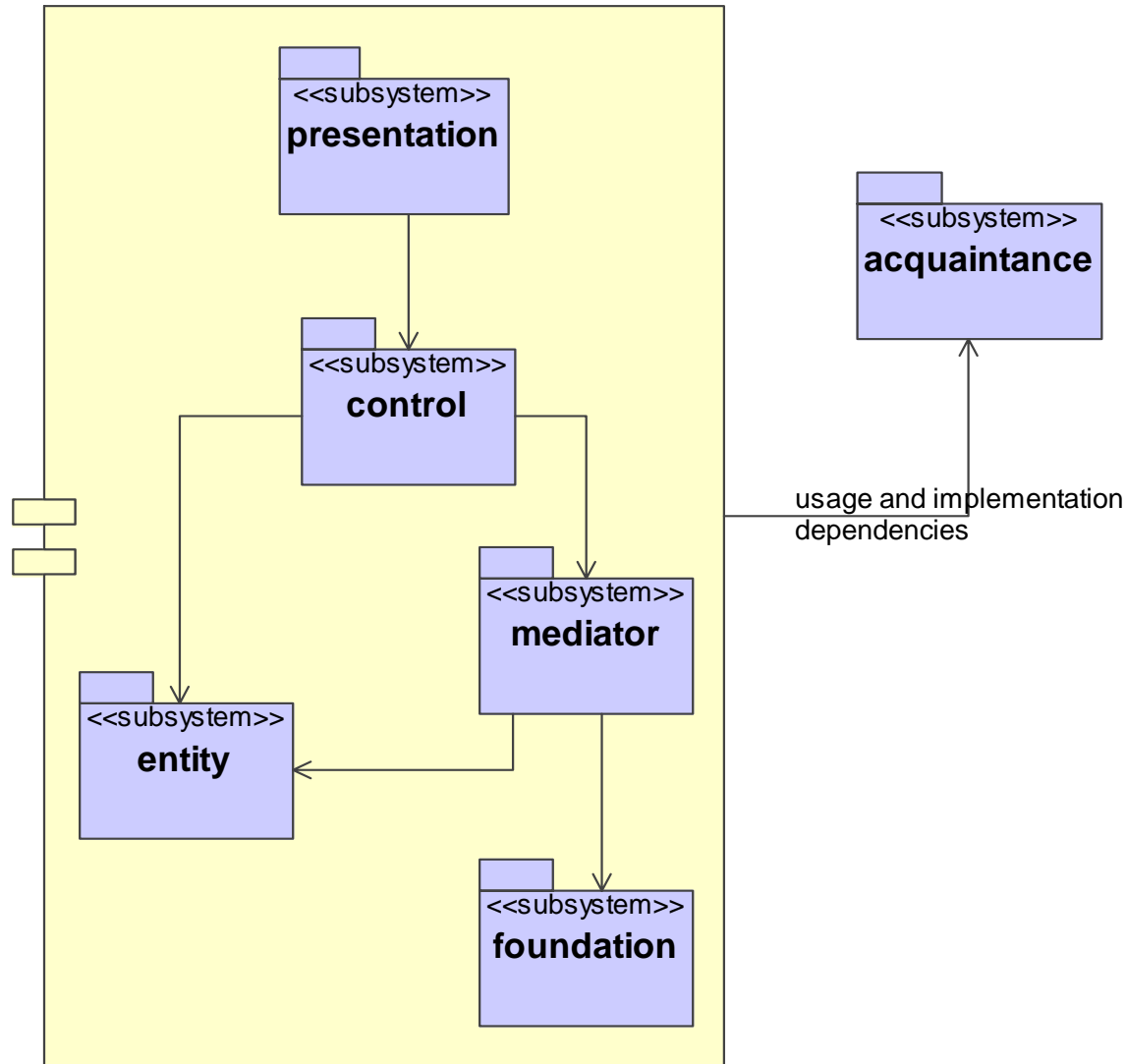
```
public class PPrimaryWindow  
implements control.ICPresenter {  
    public void do2() {  
        //implementation code  
    }  
}
```

```
public interface PController {  
    public void do2();  
}
```

```
public class CInit {  
    ICPresenter presenter;  
    public void do1(){  
        presenter.do2();  
    }  
}
```



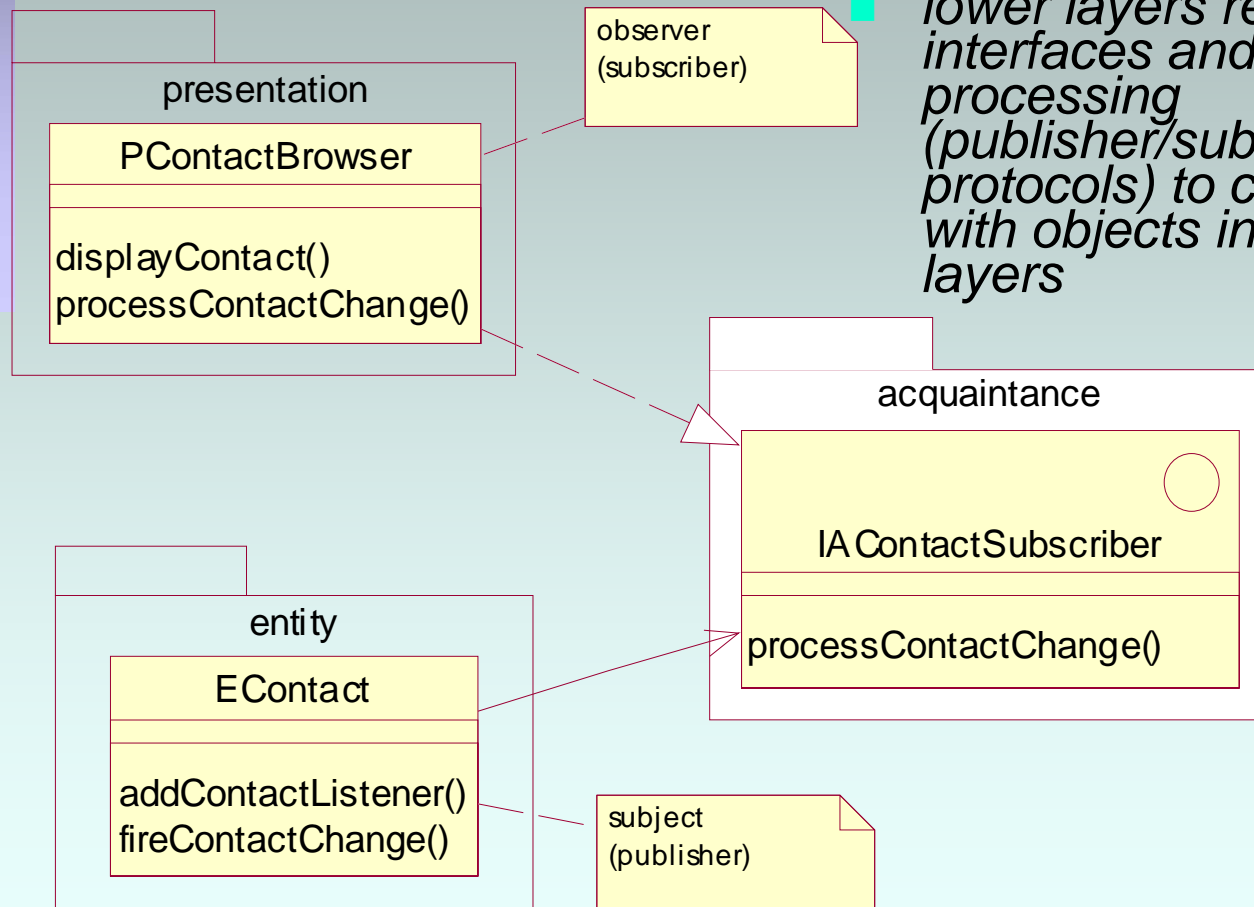
APP – acquaintance package



- *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).*

UNP – 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



PCMEF conformance verification

- *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.*
 - *The purpose is to show in numbers how much the implemented system is worse than a PCMEF solution (or other dependency-minimizing architecture)*

CCD

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

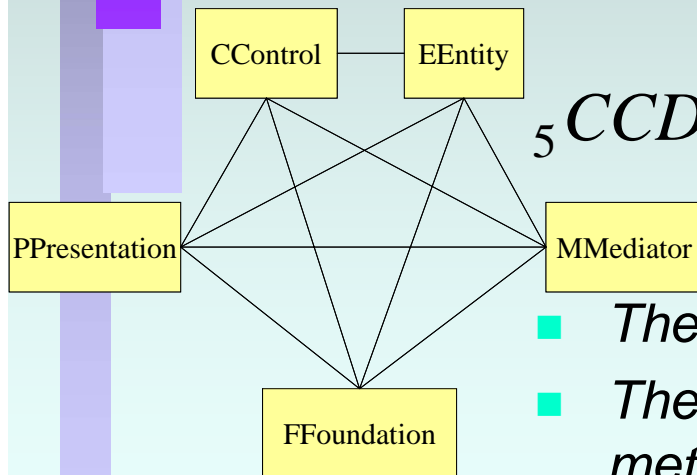
- *Calculation of CCD assumes adherence to the architectural framework.*
- *If the framework is found to be broken, the CCD is calculated as if a class can depend on any other class in the system.*
 - *probability theory method - the combinations counting rule*
 - *The CCD is the number of different combinations of pairs of dependent classes which can be formed from the total number of classes in the design multiplied by 2 (cycles)*

$${}_n CCD_2 = \frac{n!}{2!(n-2)!} \times 2$$

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 supportable architectural framework, such as PCMEF.

- 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:



$${}_5CCD_2 = \frac{5!}{2!(5-2)!} \times 2 = \frac{120}{12} \times 2 = 20$$

- 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.

CMD

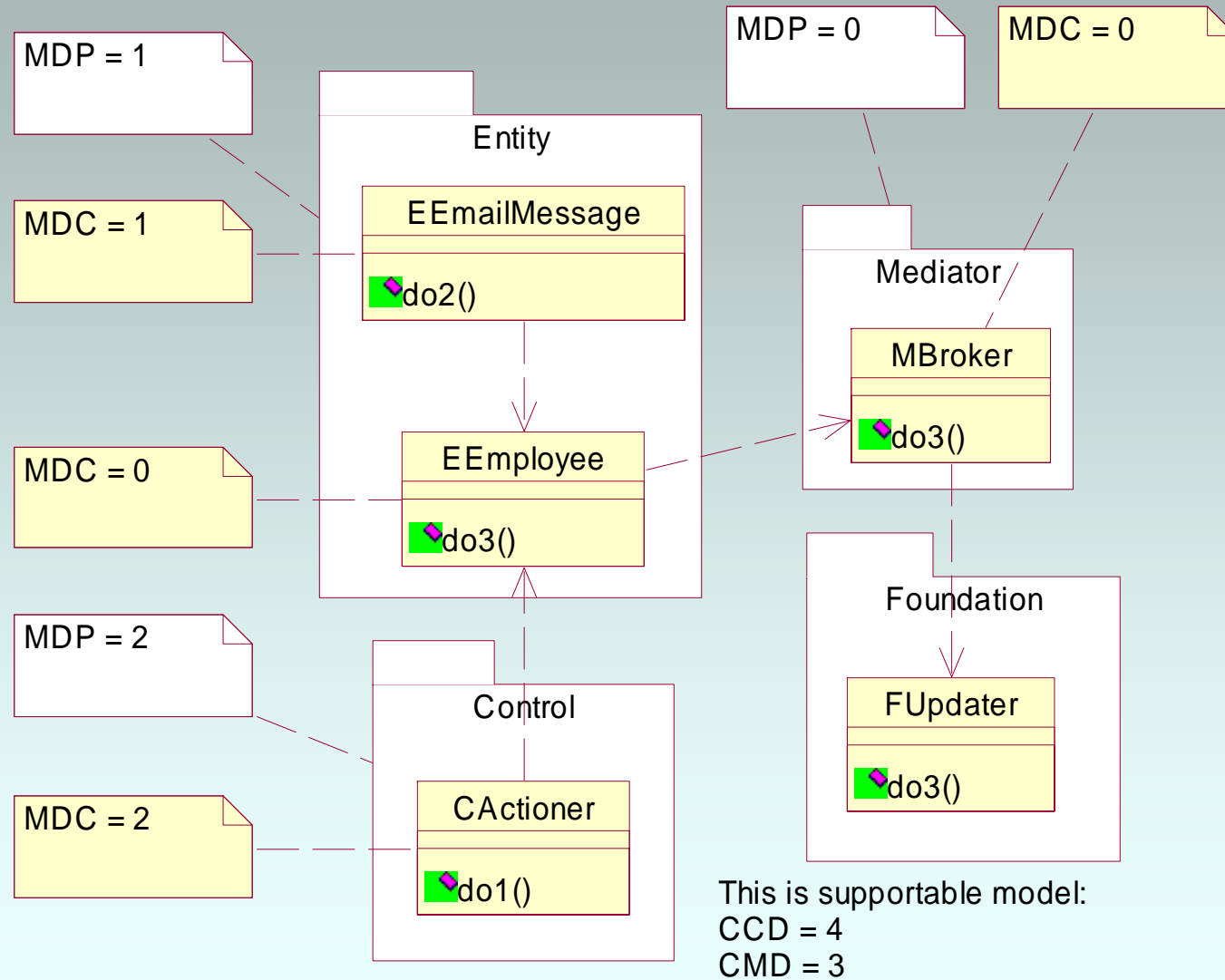
DEFINITION: **Cumulative Message Dependency (CMD)** is the total supportability cost over all Synchronous Messages SM_i within *client objects* of the costs associated with changes to methods M_j in *supplier objects* or *responsible delegator objects* that are accountable for servicing SM_i . When calculating *CMD*, the dependency value for offending (unsupported) 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

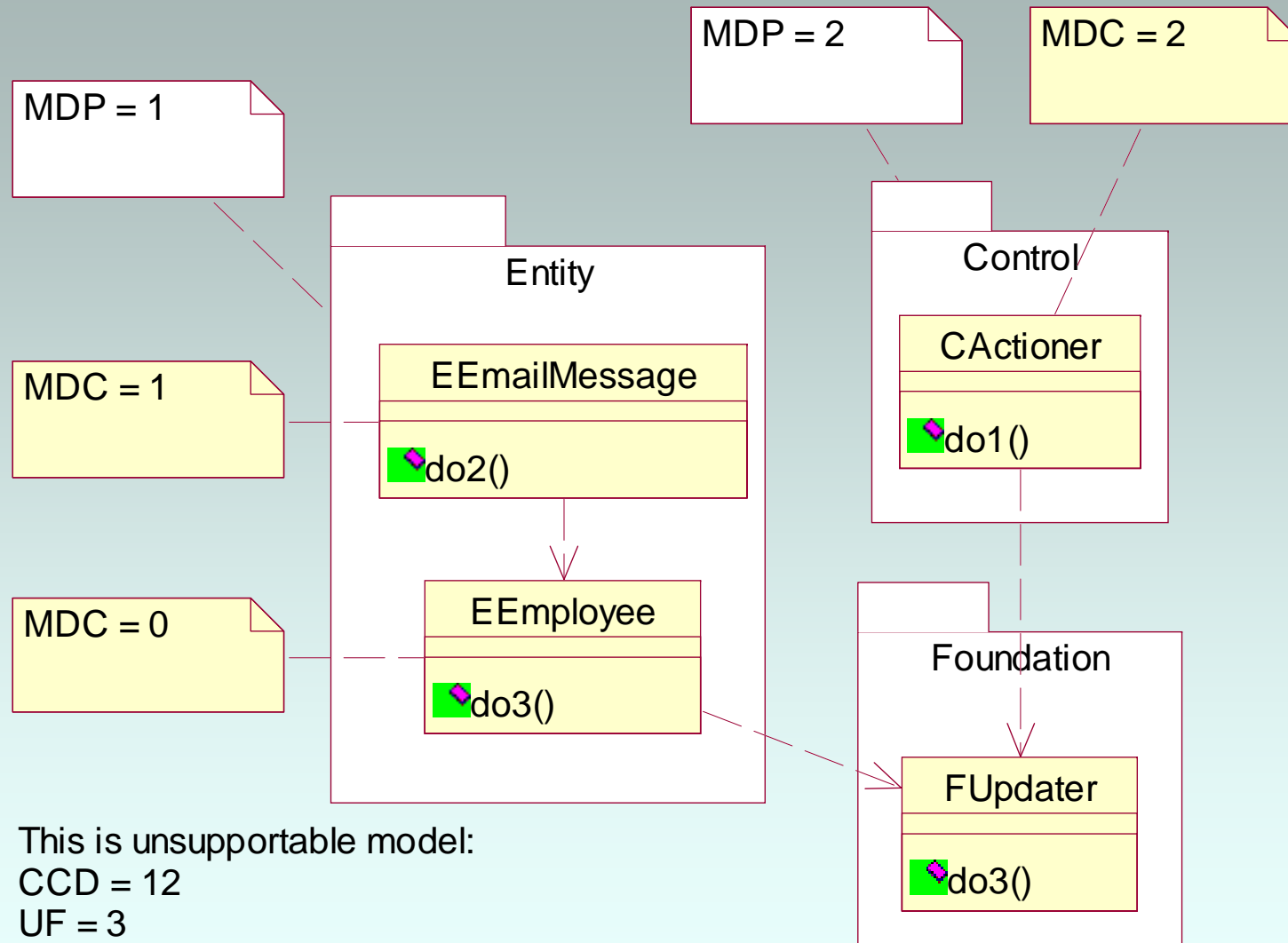


```

public class CActioner { // client of EEmployee
    EEmployee emp;
    public void do1() { // MD = 2
        emp.do3();
    }
}
public class EEmailMessage { // client of EEmployee
    EEmployee emp;
    public void do2() { // MD = 1
        emp.do3();
    }
}
public class EEmployee { // responsible delegator
    MBroker brk;
    public void do3() { // MD = 0
        brk.do3();
    }
}
public class MBroker { // delegator
    FUpdater upd;
    public void do3() { // MD = 0
        upd.do3();
    }
}
public class FUpdater { // supplier
    public void do3() { // MD is null
        // code for do3
    }
}

```

CMD - unsupportable



This is unsupportable model:
CCD = 12
UF = 3
CMD = 1+(2*3) = 7

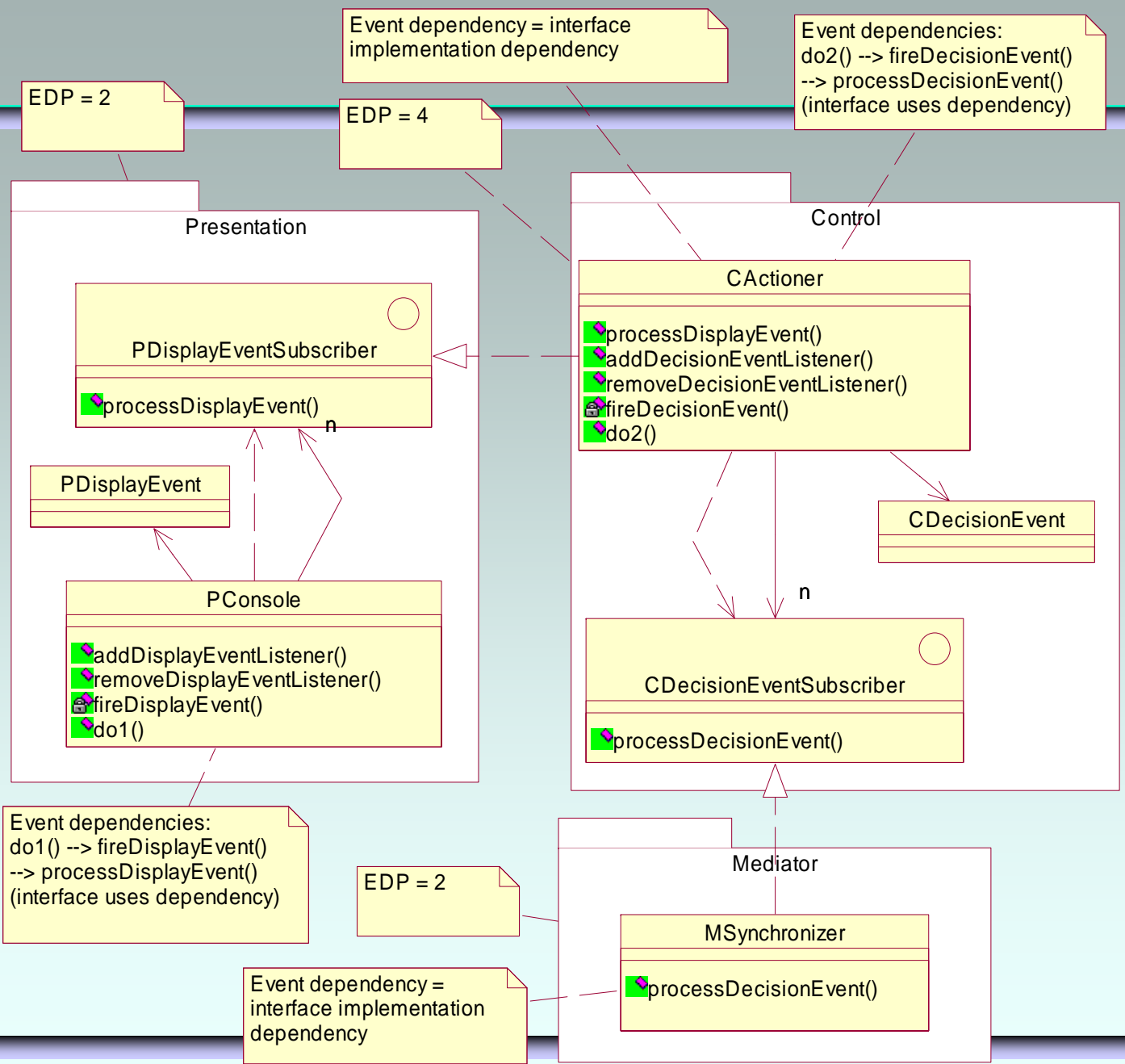
```

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
    }
}

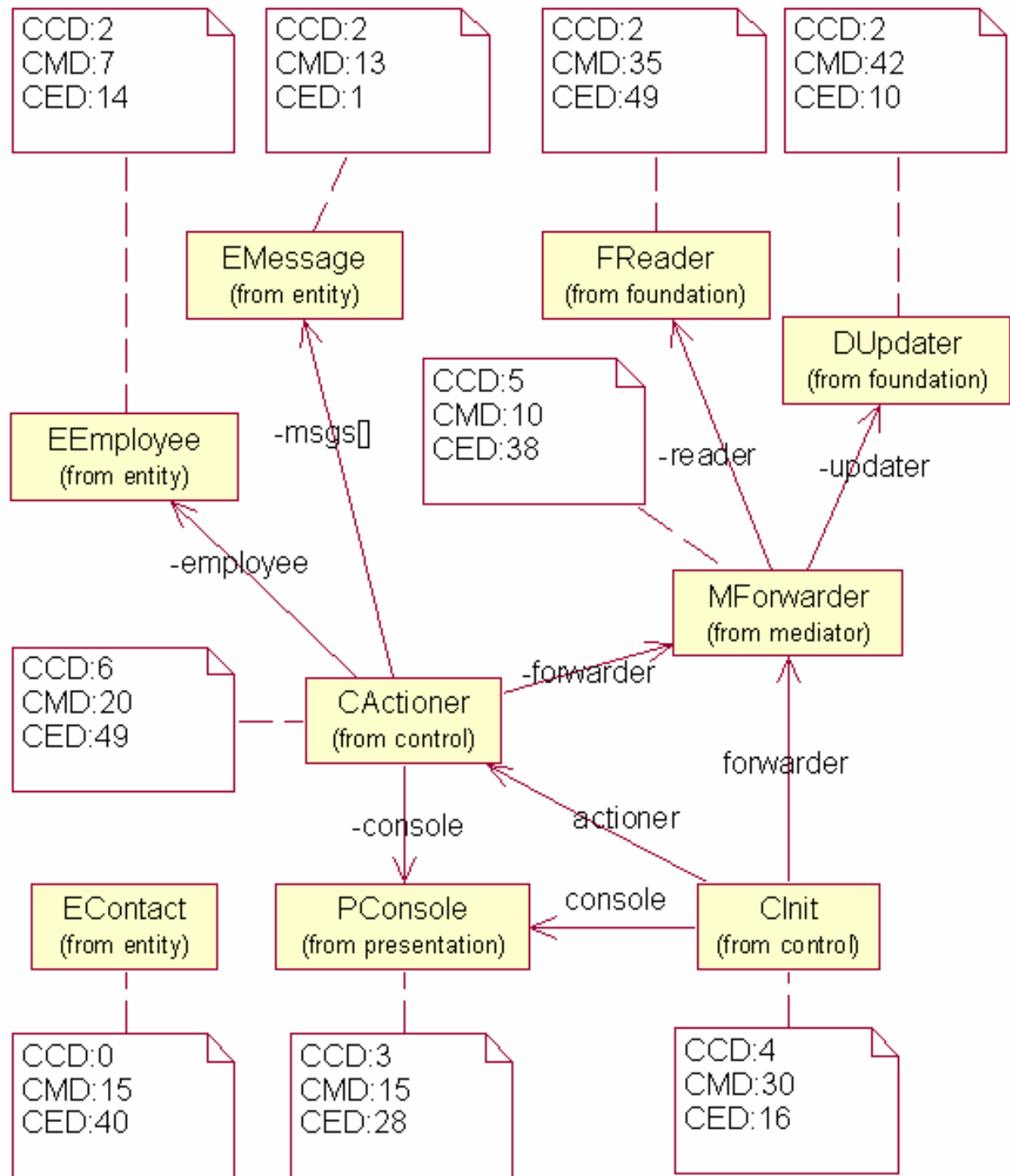
```

CED

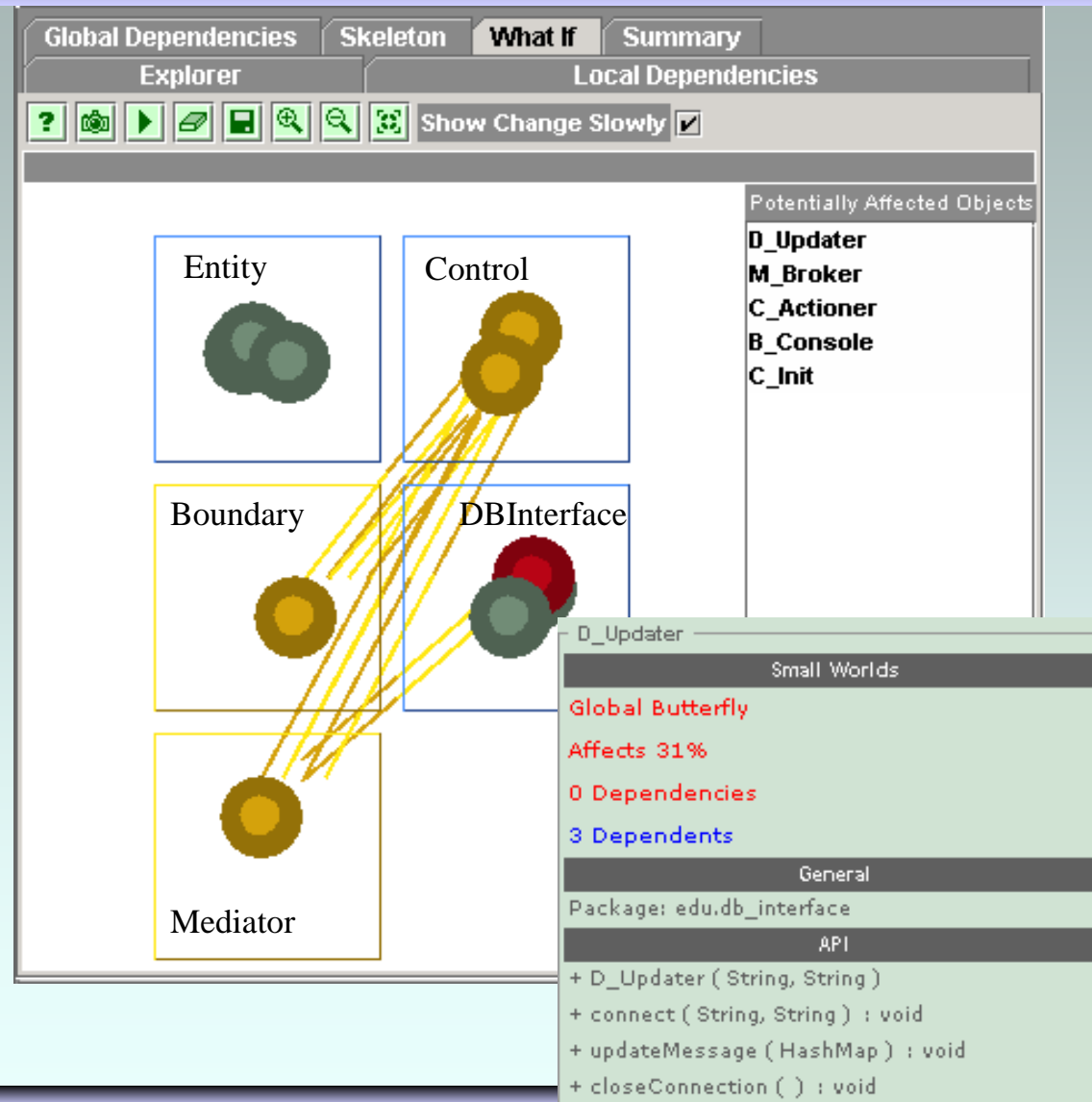
DEFINITION: **Cumulative Event Dependency (CED)** is the total supportability cost over all methods containing “fire event” messages FE_i plus over all methods containing “process event” messages PE_i within *publisher objects* plus over all methods servicing these “process events” SE_i within *subscriber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



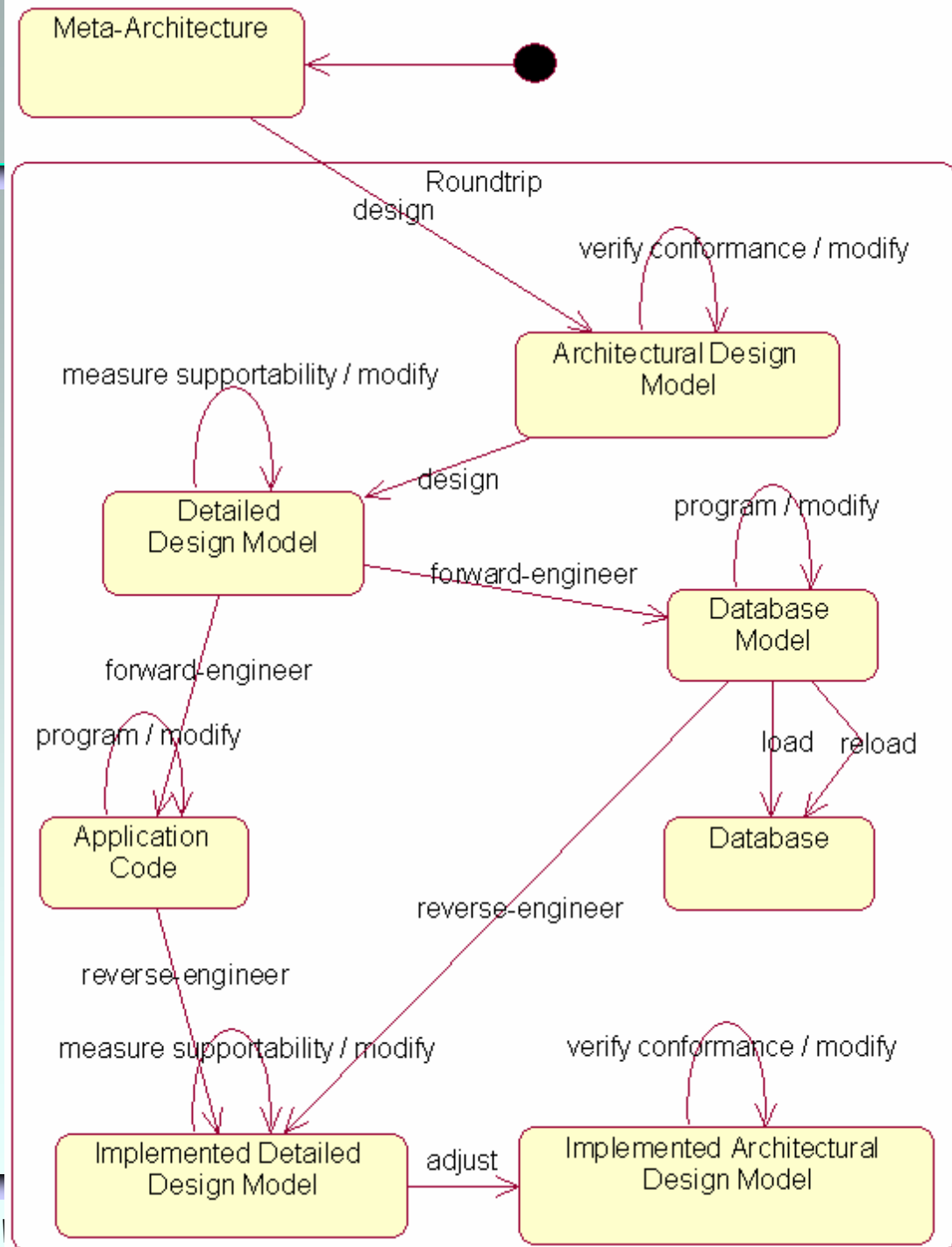
DQ tool



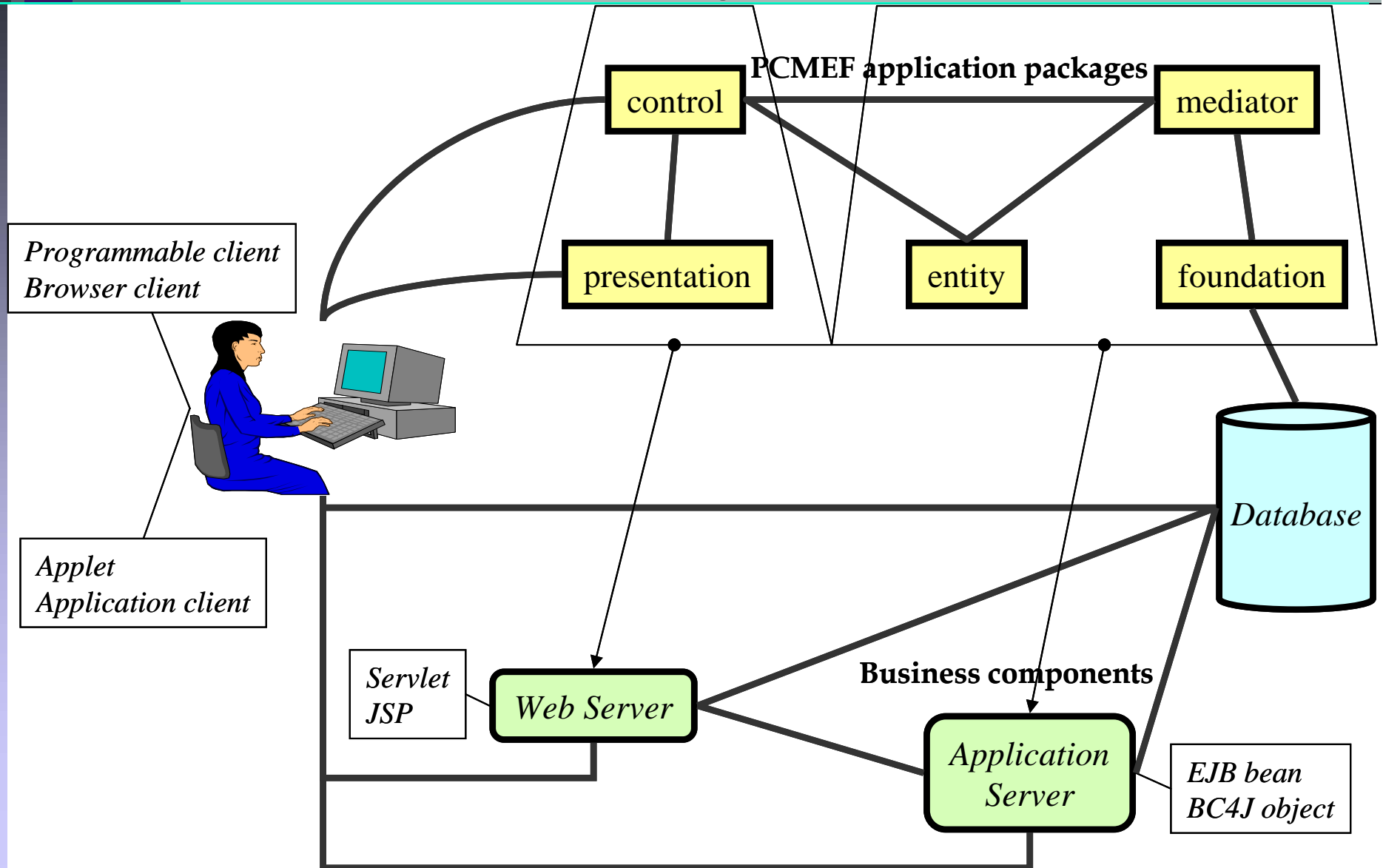
Metrics visualization



Process of roundtrip modeling



Pictorial summary



In the meantime

- *I design in UML*
- *I forward-engineer to Java and RDB, but cannot quite reverse-engineer from either*
- *I generate business components for Java and XML, but cannot quite plug into it my existing Java/Oracle applications*
- *I give clear architectural design to programmers but get a mess of intercommunicating objects that does not resemble the design*
- *I write regression tests that only work once in initial tests*
- *I establish traceability links from use cases to programs that are invalidated by the beginning of the 2nd project iteration*
- *...*

Counter-conclusion

- *“Whether we understand the world because it is hierarchic or it appears hierarchic because those aspects of it which are not, elude our understanding and observation” (Herb Simon, 1962)*
- *According to David Parnas, hirerachical structure is undefined unless we specify precisely what relationship exists between hierarchy layers*
 - *x contains y*
 - *x uses y*
 - *x has access to y*
 - *x gives work to y*
 - *x gives resources to y*
 - *x uses resources of y*

Additional references

- *FOWLER, M. (1999): Refactoring. Improving the Design of Existing Code, Addison-Wesley, 431p.*
- *FOWLER, M. (2003): Patterns of Enterprise Application Architecture, Addison-Wesley, 531p.*
- *GAMMA, E. HELM, R. JOHNSON, R. and VLISSIDES, J. (1995): Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, 395p.*
- *LARMAN, C. (2002): Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd ed., Prentice-Hall, 627p.*
- *MARTIN, R.C. (2003): Agile Software Development, Principles, Patterns, and Practices, Prentice-Hall, 529p.*