Chapter 9

Architectural Design

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

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

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
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
    - Conform 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
- Package A depends on Package B
- member (usually a class) can belong to one package only
- class is owned by only one package but it can be imported to other packages

Cycles between packages

- Eliminating cycles between packages
  - 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

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.

Run-time inheritance dependencies

Inheritance without polymorphism

Compile-time inheritance dependencies

Object
    \begin{verbatim}
    public void do1()
    { //do something
      do2();
    }
    
    public void do2()
    { //do something
      do1();
    }
    \end{verbatim}

A
    \begin{verbatim}
    public void do1()
    { //do something
      do3();
    }
    \end{verbatim}

B
    \begin{verbatim}
    public void do2()
    { //do something
      do3();
    }
    \end{verbatim}

C
    \begin{verbatim}
    public void do4()
    { //do something
      do3();
    }
    \end{verbatim}
### Extension Inheritance

- **A**
  - do2()
  - do3()

- **B**
  - do3()
  - do2()

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

### Down-calls

- **A**
  - do2()

- **B**

```java
public void do2(){
    //do some stuff
    super.do2();
}
```

### Up-calls

- **A**
  - do2()

- **B**
  - do3()
  - do2()

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

### Method Dependencies

- **CActioner**
  - `do1()`
    - `emp.do3();`

- **EOutMessage**
  - `do2()`
    - `emp.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.**

### Interface Notations

- **<<Interface>>**
  - Interface1
  - Interface2
  - Interface3

- **AbstractClass1**
  - Class1
### Implementation dependency

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

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

### Usage dependency

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

### Cycle

Cyclic dependencies between classes and other structures (methods, packages, subsystems) are unavoidable, but can be neutralized with extra classes to reduce a network of calls to a hierarchy. Purposeful use of interfaces allows for:

- Extra classes to reduce a network of calls to a hierarchy.
- Purposeful use of interfaces.

### Cycle elimination

In a callback, the publisher has no knowledge or interest in how the subscriber processes the event. The dependency exists but 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 in method calls, instead of concrete objects
  - 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)
The entity subsystem
- classes that know how to talk to the database
- manages the memory cache and synchronizes the states of business objects between memory and the database

The presentation subsystem
- containers are linked
- produces SQL to read and modify the database

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 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

Object communication due to acquaintance
- Acquaintance defines a situation when an object is passed another object in an argument to its method.
- More precisely, an object A in an argument of the message to A.
- Acquaintance is one of the programming techniques legitimized in the Law of Demeter.
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-adjacent subsystems can communicate without knowing the concrete suppliers of services (and, therefore, without creating dependencies on concrete classes).

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

- to define "a higher-level interface that makes the subsystem easier to use"
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"

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"

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
  - new levels of structural dependencies: layer dependencies, package dependencies, and class dependencies
  - method dependency and event dependency 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