Chapter 15
Architectural Refactoring

What is refactoring?

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" (Fowler)

- Refactoring is about cleaning up the code after it has been written
- Refactoring targets are potential problem areas ("bad smells in code") in the design so far
- Refactoring integrates very well with the agile development, but it can (and should) be used with other processes

Topics

- Refactoring targets
- Refactoring methods
  - Extract class
  - Subsume method
  - Extract interface
- Refactoring patterns
  - Identity map
  - Data mapper
  - Lazy load

Refactoring targets

- duplicated code – the same pieces of code in multiple places,
- long method – a method that does too much,
- large class – a class that does too much and/or has too many data members,
- long parameter list – too much data passed in parameters (rather than asking other objects for the data),
- divergent change – when a class has to be changed as a result of more than one kind of change,
- shotgun surgery – when the same change affects many classes,
- feature envy – a method that accesses many other objects with get messages in order to get data for its own computation,
- data clumps – data items (data members, parameters) that tend to be used together in many places and should be made into an object.

Refactoring methods

- Refactoring methods (or simply refactorings) are basic principles and best practices of changing the code to improve its understandability, maintainability and scalability
- Many CASE and IDE-s assist in performing refactorings
- We’ll discuss just three refactoring methods:
  - Extract Class
  - Subsume Method
  - Extract Interface

Extract Class

- Consider large class as refactoring target
- Applicable methods are:
  - Extract Class and
  - Extract Interface
- Extract Class – “Create a new class and move the relevant fields and methods from the old class into the new class”
  - Once fields and methods are relocated to new class (classes), an association link should be established from the old to the new class
Refactoring patterns are architectural patterns used in code refactoring.

Discussed here:
- Identity Map
- Data Mapper
- Lazy Load
- Virtual Proxy
- Navigation in Identity Map
- Unit of Work
Identity Map

- "Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them."
- An Identity Map object maintains one or more maps (e.g. hash maps) that map an object identifier to an object.
- Objects in an explicit identity map are accessed (get), registered (put) and unregistered (remove) with distinct methods for each class of cached objects.
  - getEmployee (new Integer(empOID));
- Objects in a generic identity map are accessed (get), registered (put) and unregistered (remove) with a single method for all classes (a parameter of the method determines the class).
  - get("EEmployee", new Integer(OID));

```java
public class EIdentityMap {
    private Map OIDToObj; //OID -> Obj
    private Map msgPKToOID; //msgPK -> OID

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

    // Get the stored contact
    public IAContact findContact(int contactOID) {
        return (IAContact) OIDToObj.get(new Integer(contactOID));
    }

    // Store a contact with the OID indicated
    public void registerContact(IEObjectID oidObject) {
        OIDToObj.put(new Integer(oidObject.getOID()), oidObject);
    }

    // Unregister a registered contact
    public void unregisterContact(IEObjectID oidContact) {
        OIDToObj.remove(new Integer(oidContact.getOID()));
    }
}
```

Data Mapper

- "A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself."

Load – check-in

- Memory objects are unloaded to DB.

Load – check-out

- DB records are retrieved (loaded) and transformed to memory objects.

MDataMapper – example

- MDataMapper - fragment

```java
class MDataMapper {
    private Map msgPKToOID; //msgPK -> OID

    public MDataMapper() {
        msgPKToOID = new HashMap();
    }

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

    public void unregisterContact(IEObjectID oidContact) {
        msgPKToOID.remove(new Integer(oidContact.getOID()));
    }
}
```
Many data mappers

Metadata mapping

Lazy Load

Lazy Initialization

Virtual Proxy - interaction

Metadata mapping

Where to keep metadata?
- in the source code of the application
- in an external file, preferably an XML file
- in the database

How to embed mapping into the running code?
- code generation
- reflective program

Lazy Load

- Two kinds of retrieval operations:
  - Identity load
  - Predicate load
- Three loading strategies:
  - Closure load (eager load)
  - Flat load
  - N-levels load
- Three approaches to Lazy Load:
  - Lazy Initialization
  - Virtual Proxy
  - OID Proxy

Lazy Initialization

- On request from a client object, Data Mapper – which is responsible for maintaining the entity cache – searches the cache for data and, if the data is not there, loads it from the database.

Method getContact() in EOutMessage

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

Virtual Proxy

- Placeholder object that stands for real object (called real subject in the pattern)

Virtual Proxy - interaction
OID Proxy

- Internal programming mechanisms of ensuring the identity of objects loaded to memory are no replacement for OIDs assigned explicitly by the program.
- A singleton class (EIdentityMap) that maintains maps of OIDs to objects can replace proxy classes:
  - It 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 gets out of sync with database, it is flagged dirty
  - The entity object knows its clean/dirty status at all times
- Two variants of OID proxy:
  - navigation in identity map
  - navigation in entity classes

Navigation in Identity Map

- Association link in entity object is represented:
  - by FK obtained from the database, and
  - by an OID to a related object in memory.
- A null in OID means that associated object is not loaded

Navigation in Identity Map - interaction

- The responsibility to navigate between entity objects placed on these objects

Navigation in Entity Classes - interaction

Summary

- Refactoring is the process of cleaning up and improving the internal structure of the code without changing its external behavior.
- There are two main refactoring targets: elimination of “bad smells in code” (cleaning up) and structural improvements to the code resulting in a better code architecture.
- The Identity Map pattern assigns object identifiers to objects and maintains maps to find in-memory objects based on their object identifiers.
- The Data Mapper pattern decouples in-memory objects from their persistent representation in the database and is responsible for maintaining memory caches of entity objects.
- The Lazy Load pattern loads only selected objects from database to memory but it can load remaining and related objects when needed.