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.
Extract Class - example

<table>
<thead>
<tr>
<th>CActioner</th>
<th>CMfgSender</th>
</tr>
</thead>
<tbody>
<tr>
<td>login()</td>
<td>sendMessage()</td>
</tr>
<tr>
<td>getEmployee()</td>
<td></td>
</tr>
<tr>
<td>exit()</td>
<td></td>
</tr>
<tr>
<td>logout()</td>
<td></td>
</tr>
<tr>
<td>sendMessage()</td>
<td></td>
</tr>
<tr>
<td>retrieveMessages()</td>
<td></td>
</tr>
<tr>
<td>retrieveMessage()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CMfgSeeker</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSeeker</td>
</tr>
<tr>
<td>setEmployee()</td>
</tr>
<tr>
<td>retrieveMessages()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CMfgSender</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgSender</td>
</tr>
<tr>
<td>sendMessage()</td>
</tr>
</tbody>
</table>

CAdmin

getMsgSeeker()
getMsgSender()
login()
getEmployee()
exit()
logout()

Subsume Method

- **Subsume Method** – eliminates a method by including its functionality into another existing method
  - addresses “duplicated code” (the same pieces of code in multiple places)

- **Example – View Unsent Messages and Display Text of a Message**
  - retrieving many outmessages and retrieving a single outmessage involves very similar processing
**Subsume Method – after refactoring**

- `getUserInput()`
- `displayMessages()`
- `retrieveMessages()`
- `retrieveUnsentMessages()`
- `query()`
- `displayMessageText()`
- `retrieveMessages()`
- `displayLogin()`
- `displayMenu()`
- `display()`
- `prepareMessage()`

**Extract Interface**

- "Several clients use the same subset of a class’s interface, or two classes have part of their interfaces in common" → “Extract the subset into an interface”
- **PConsole** is a large class that does too much and performs distinct tasks
  - For example, it may need to display a list of outmessages or details of a single outmessage
Architectural patterns

- **Refactoring patterns** are architectural patterns used in code refactoring
  - Fowler calls them Patterns of Enterprise Application Architecture.
  - **Discussed here:**
    - Identity Map
    - Data Mapper
      - 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
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
  - getEEmployee (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));

Identity Map - example

```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."

control package: CAdmin

mediator package: MDataMapper

entity package: EIdentityMap

foundation package: FReader

give me entity object (e.g., EContact)

let’s find it

retrieve it from the database

find it by OID

where is it?

get it from the cache

is it clean?

check if clean

[dirty]

[not in cache]

[not in cache]

[dirty]

[dirty]

[dirty]

[dirty]

entity package: EContact

Load – check-out

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

1. retrieveEmployeeByName(username: String)
   1.1. query(sql: String)
   1.2. createEmployee()
   1.2.1. EEmployee(_: IEmployee)
   1.3. closeResultSet(rs: java.sql.ResultSet)
   1.4. registerEmployee(_: Object)
Load – check-in

- Memory objects are unloaded to DB

1. sendMessage : (from : String, to : String, subject : String, body : String)

2. updateMessage : (msg : acquaintance, IAOutMessage)
   2.1. storeMessage : (msg : acquaintance, IAOutMessage)
      2.1.1. update(sql : String)
      2.1.2. unregisterMessage : (oldMessage : domain.entity, EObjectID)
          2.1.2.1. flagCache(flag : boolean)

MDataMapper - example

MDataMapper

- MDataMapper()
- retrieveMessages()
- retrieveEmployeeByOId()
- storeMessage()
- retrieveEmployeeByName()
- retrieveEmployeeByOId()
- retrieveContactByOId()
- retrieveMessage()
- doRetrieveMessages()
- createEmployee()
- createOutMessage()
- createContact()
- getEmployeeOId()
- getContactOId()
- getMessageOId()

FReader

- FReader()
- setConnection()
- query()
- closeResult()

FWriter

- FWriter()
- setConnection()
- update()
- closeStatement()
### Many data mappers

- `getByOID()` – get an entity object if given its OID; retrieve it from database if cache is dirty.
- `retrieve()` – retrieve an entity object from the cache (if it is there) or from the database; if the latter, create a new entity object and put it in the cache.
- `insert()` – convert a new entity object (created by the application) to raw data and insert it into the database.
- `update()` – save the changes to an entity object to the database.
- `delete()` – delete an entity object from the database and remove it from the cache.

### Metadata mapping

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

- **Assuming one-to-one correspondence between table and classes**
- **Assuming default RDBMS-Java type mappings, no separate class for type mappings**

- **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

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

**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.

```java
Method getContact() in EOutMessage
public IAContact getContact() {
    if (contactOID == null) {
        contact = MDataMapper.retrieveContact(contactID);
    }
    return contact;
}
```
Virtual Proxy - interaction

1. retrieveContactForOutMessage(outmsgID)
   1.1. getContact()
   1.2. getRealSubject()
   1.3. getFirstName()
   1.4. getFamilyName()

If EContact not loaded, retrieve it from the database and instantiate.

EContactProxy - interaction

1. retrieveContactForOutMessage(outmsgID)
   1.1. getContact()
   1.2. getFirstName()
   1.3. getFamilyName()

returns proxy to start with

EContactProxy

realSubject : IAContact
contactID : String
getRealSubject()
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
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 “bed 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 entity 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.