DATA ACCESS OBJECT DESIGN

Pavel Mička
mickapa1@fel.cvut.cz

Fundamental concepts

- Encapsulation
  - Replaceability
  - Modifiability
- Don’t repeat yourself (DRY principle)
- You ain't gonna need it (YAGNI principle)
- Single responsibility principle
- Reusability
- Testability

JPA NAMED QUERIES

The prototyping way

Entity

```
@Entity
@Table(name = "book", uniqueConstraints = {
    @UniqueConstraint(columnNames = {"title"})})
@NamedQueries({
    @NamedQuery(name = Book.Q_FIND_ALL_BOOKS, query = "SELECT b FROM Book b"),
    @NamedQuery(name = Book.Q_FIND_BOOK_BY_ISBN, query = "SELECT b FROM Book b
WHERE b.isbn = :isbn"),
    @NamedQuery(name = Book.Q_FIND_BOOK_BY_TITLE, query = "SELECT b FROM Book b
WHERE b.title = :title"),
    @NamedQuery(name = Book.Q_FIND_BOOK_BY_EDITION, query = "SELECT b FROM Book b
WHERE b.edition = :edition"),
    @NamedQuery(name = Book.Q_FIND_BOOK_BY_OWNER, query = "SELECT b FROM Book b
WHERE b.owner = :owner"),
    @NamedQuery(name = Book.Q_REMOVE_BOOK_BY_ISBN, query = "DELETE FROM Book b
WHERE b.isbn = :isbn")})
public class Book implements Serializable {
    <ENTITY CODE HERE>
}
```

Call

```
public class BookLogic implements BookLogicLocal {

    @PersistenceContext
    EntityManager em;

    @Override
    public List<Book> getAllBooks() {
        return em.createNamedQuery(Book.Q_FIND_ALL_BOOKS).getResultList();
    }
```

JPA Named Queries

Pros

- Generated by IDE
- Rapid programming
  - Prototyping

Cons

- Violates encapsulation
  - Exhibits data access implementation to service layer (Entity manager) and business objects (annotations)
  - No modifiability, everything in annotations
- Violates Single responsibility principle
  - Service layer is now responsible for data access caching and logging
- Violates DRY principle
  - Y.Q_FIND_Y_BY_X
    - For all X, X is a property
    - For all Y, Y is an entity

JPA Named Queries (4)

Pros

Cons

- Violates YAGNI
  - Generates all queries, even those, which are not needed
- No reusability
- Low testability
  - Every named query should be tested
  - Because of massive duplication, there might be hundreds of them

SIMPLE DAO

Improving encapsulation

Simple DAO

```
public class BookDAO implements BookDAOIface {
    public List<Book> getAll() {
        return getEntityManager().createQuery("SELECT b FROM Book b").getResultList();
    }

    public List<Book> getByProperty(String property, Object value) {
        String queryString = "SELECT e FROM Book e WHERE e." + property + " = :value";
        return getEntityManager().createQuery(queryString).setParameter("value", value).getResultList();
    }
    <MORE data access methods here>
}
```

Simple DAO

Pros

- Encapsulates database technology
- One method for all getByX calls

Cons

- Violates DRY principle
  - New DAO with the same generic functionality for all entities
- No reusability
- Low testability (because of code duplication)
  - But better than with named queries

GENERIC DAO

Improving reusability

Generic DAO

- Improvement of Simple DAO
- Generic functionality contained in genericly typed object

```
public class GenericDAO implements GenericDAOIface {

    public <ENTITY> List<ENTITY> getAll(Class<ENTITY> clazz) {
        return getEntityManager().createQuery("SELECT e FROM " + clazz.getSimpleName() + "
                e").getResultList();
    }

    public <ENTITY> List<ENTITY> getByProperty(String property, Object value, Class<ENTITY>
clazz) {
        String queryString = "SELECT e FROM " + clazz.getSimpleName() + " e WHERE e." + property +
" = :value";
        return getEntityManager().createQuery(queryString).setParameter("value", value).getResultList();
    }

    <MORE data access methods here>

}
```

Generic DAO problem

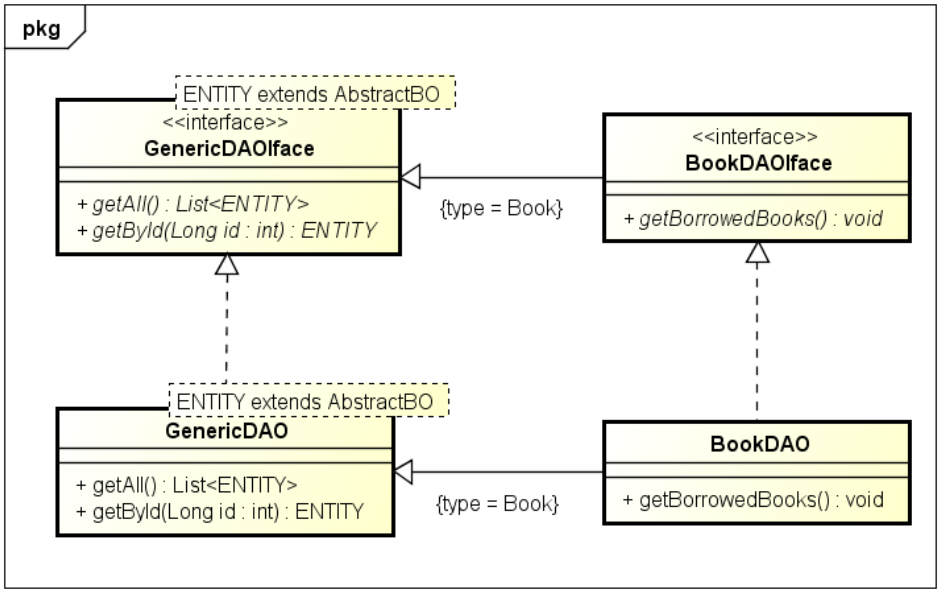
- Where to place the specific functionality?
  - Into the generic object
    - Violates single responsibility principle
    - Creates huge poorly maintainable object with low cohesion
  - Into separate object
    - Better solution
    - BUT, what to do, if the specific call must override generic call?
      - Inconsistent API
  - Inheritance strategy
    - Modify the code (previous slide) in such a way that for every class there exists one subclass of the generic DAO (next slide)
    - Generic DAO should be an abstract base class

Generic DAO – inheritance strategy

```
public class GenericDAO<ENTITY extends AbstractBusinessObject> implements GenericDAOIface<ENTITY> {
    public List<ENTITY> getAll() {
        return getEntityManager().createQuery("SELECT e FROM " + clazz.getSimpleName() + "  e").getResultList();
    }

    public List<ENTITY> getByProperty(String property, Object value) {
        String queryString = "SELECT e FROM " + clazz.getSimpleName() + " e WHERE e." + property + " = :value";
        return getEntityManager().createQuery(queryString).setParameter("value", value).getResultList();
    }

    <MORE data access methods here>
    <Clazz parameter resolver, it can be inferred
                from the generic type using reflection>
}
```



What is AbstractBusinessObject (1)

```
@MappedSuperclass
public abstract class AbstractBusinessObject implements Serializable {
    @Id
    @GeneratedValue(generator="system-sequence")
    @GenericGenerator(name="system-sequence", strategy = "sequence")
    protected Long id;
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public boolean equals(Object obj) {
        if(obj == null) return false;
        else if(!(obj instanceof AbstractBusinessObject)) return false;
        return ((AbstractBusinessObject)obj).getId().equals(this.getId());
    }

    public int hashCode() {
        int hash = 7;
        hash = 97 * hash + (this.id != null ? this.id.hashCode() : 0);
        return hash;
    }
}
```

What is AbstractBusinessObject (2)

- Abstraction of saved objects
  - In conventional databases all objects are identified only by identifiers
  - Abstract BO guarantees, that the entity has ID and assignes one to entity on persistence (in example code transparently using JPA)
  - Provides all subclasses with hashcode and equals methods

Generic (inherited) DAO

Pros

- Encapsulates database technology
- Generic functionality implemented only once
- Programmer can consistently override generic functionality
- Good reusability

Cons

- Majority of calls uses only the generic implementation, but this architecture requires to create for all entities specific subclasses
  - These subclasses will be almost always empty and will have an empty specific interface
  - Violates YAGNI

DAO DISPATCHER

My way of doing it ☺. Getting rid of empty classes, creating a robust DAO module reusable in different projects.

DAO Dispatcher

- Extends Generic DAO in a way of composition
- Polymorphism is done via explicit call, not implicitly
- Actors of the pattern
  - Registry/DAO dispatcher
  - Generic DAO implementation
  - Abstract Specific DAO
  - Specific implementations

Dsipatcher/Registry (1)

- Central point of the pattern is a dispatcher/registry, which maps Entity classes to actual DAO implementations
- The registry itself implements GenericDAO and should be used as en entry point for generic calls, when no specific functionality exist
- The registry guarantees that when specific DAO is registered, always its implemetation of generic operations will be performed (acts as a dispatcher)
- Registry itself does not (and should not) provide any DAO functionality
- By default dispatcher delegates all data access calls to the generic implementation

Dispatcher/Registry (2)

```
public class GenericDAODispatcher implements DAODispatcherIface {

    private Map<Class, SpecificDAOIface> registry;
    private GenericDAOIface genericDAO;

    public GenericDAODispatcher() {
        this.registry = new HashMap<Class, SpecificDAOIface>();
    }
    public <ENTITY extends AbstractBusinessObject> void register(SpecificDAOIface<ENTITY> specificDAO,
                                                                  Class<ENTITY> clazz) {
        registry.put(clazz, specificDAO);
    }

    public <ENTITY> List<ENTITY> getAll(Class<ENTITY> clazz) {
        if (registry.containsKey(clazz)) {
            return registry.get(clazz).getAll();
        } else {
            return genericDAO.getAll(clazz);
        }
    }

    <MORE data access methods here, unregister method>
}
```

Generic DAO implementation

- Actual worker class
- Should never be contacted directly, only using a dispatcher
- The implementation is the same as Generic DAO in the previous chapter

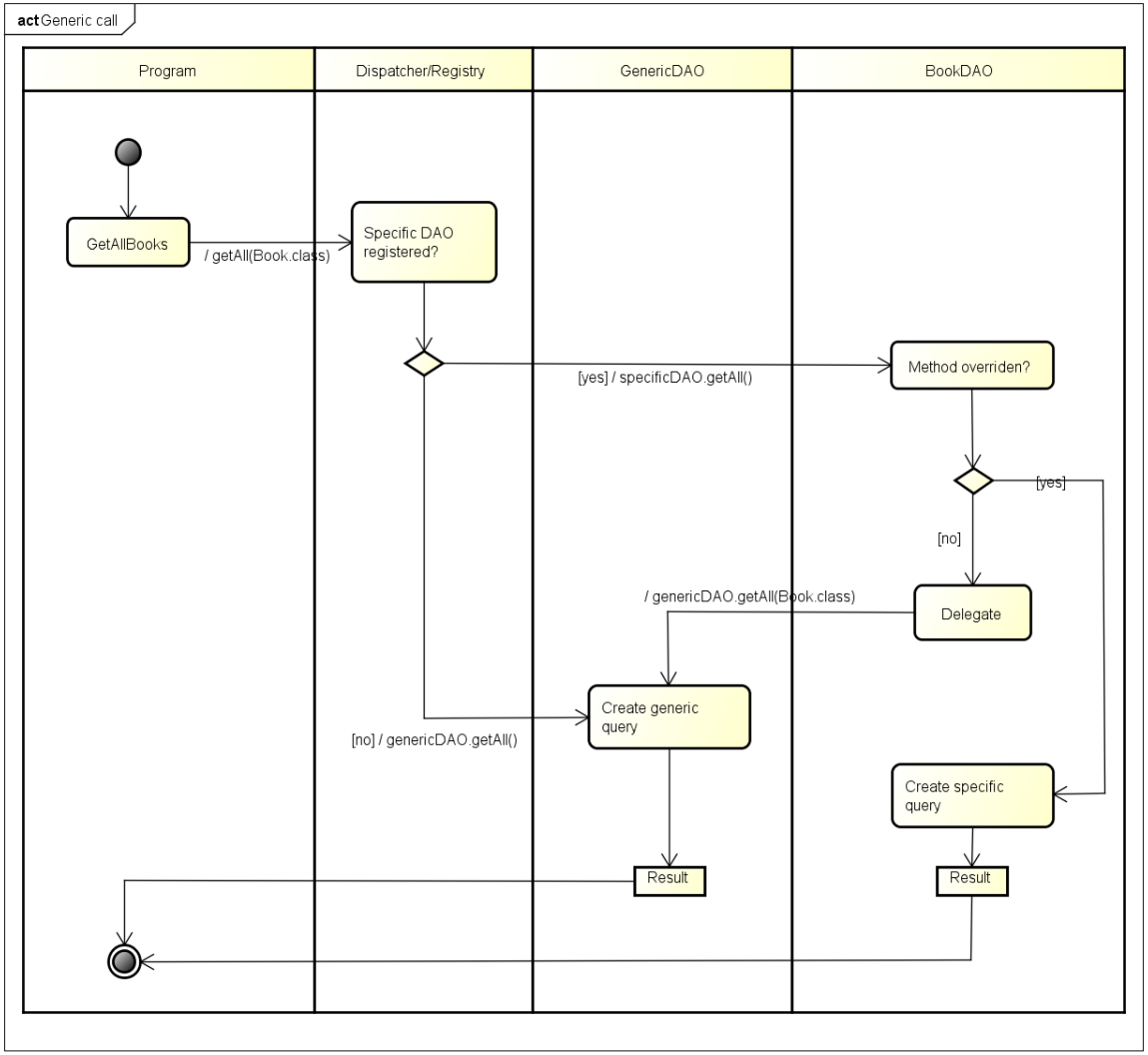
Abstract specific DAO

- Implemens SpecificDAO interface (same as the generic one, but without already known class parameters)
- As it was stated, for most of entities only generic calls are needed
  - Hence this base class by default delegates all calls back to the GenericDAO
- Automatically registers itself to the registry

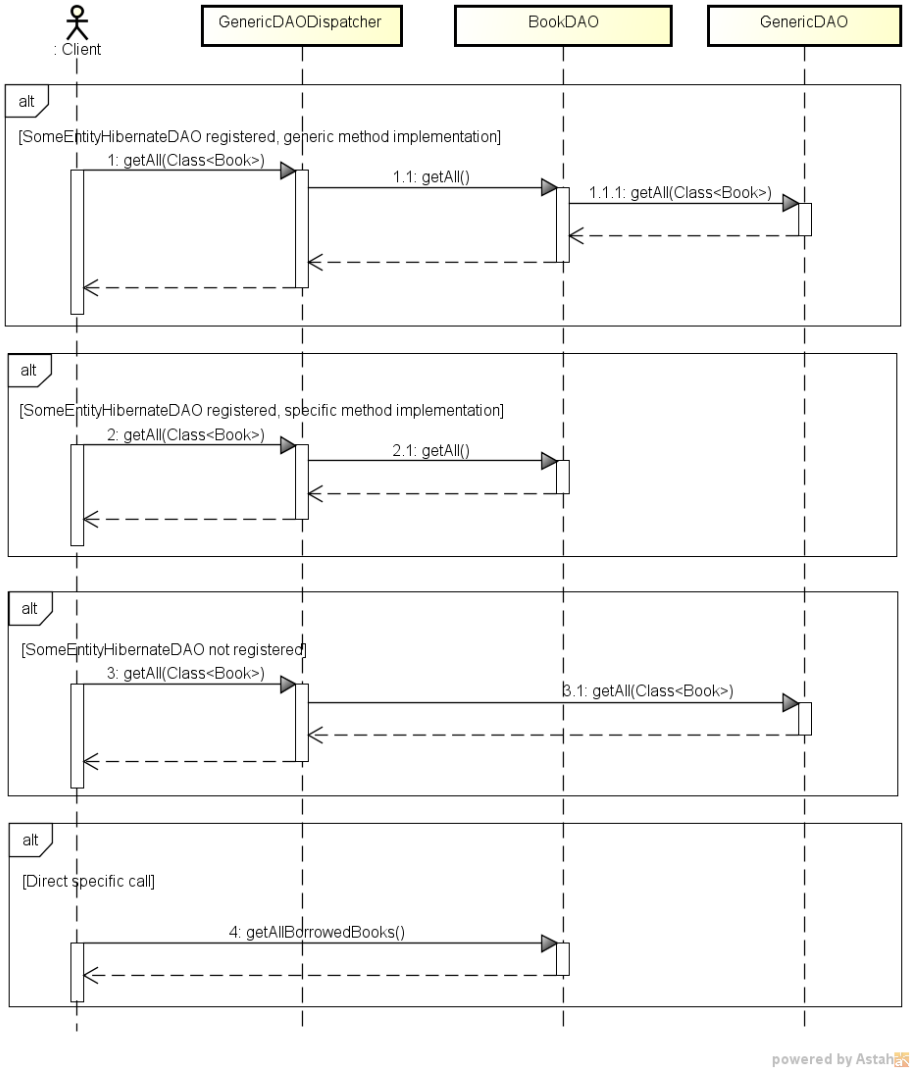
Concrete specific DAO

- For example BookDAO
- Might be called directly or using the dispatcher
- Extends AbstractSpecificDAO, hence is automatically registered in the dispatcher/registry
- If it does not override generic functionality, all calls are automatically delegated to the GenericDAO
- By overriding the inherited methods, all calls (even those already existing and using dispatcher) are altered
- Might (and should) declare its own specific methods, which are called directly on this instance

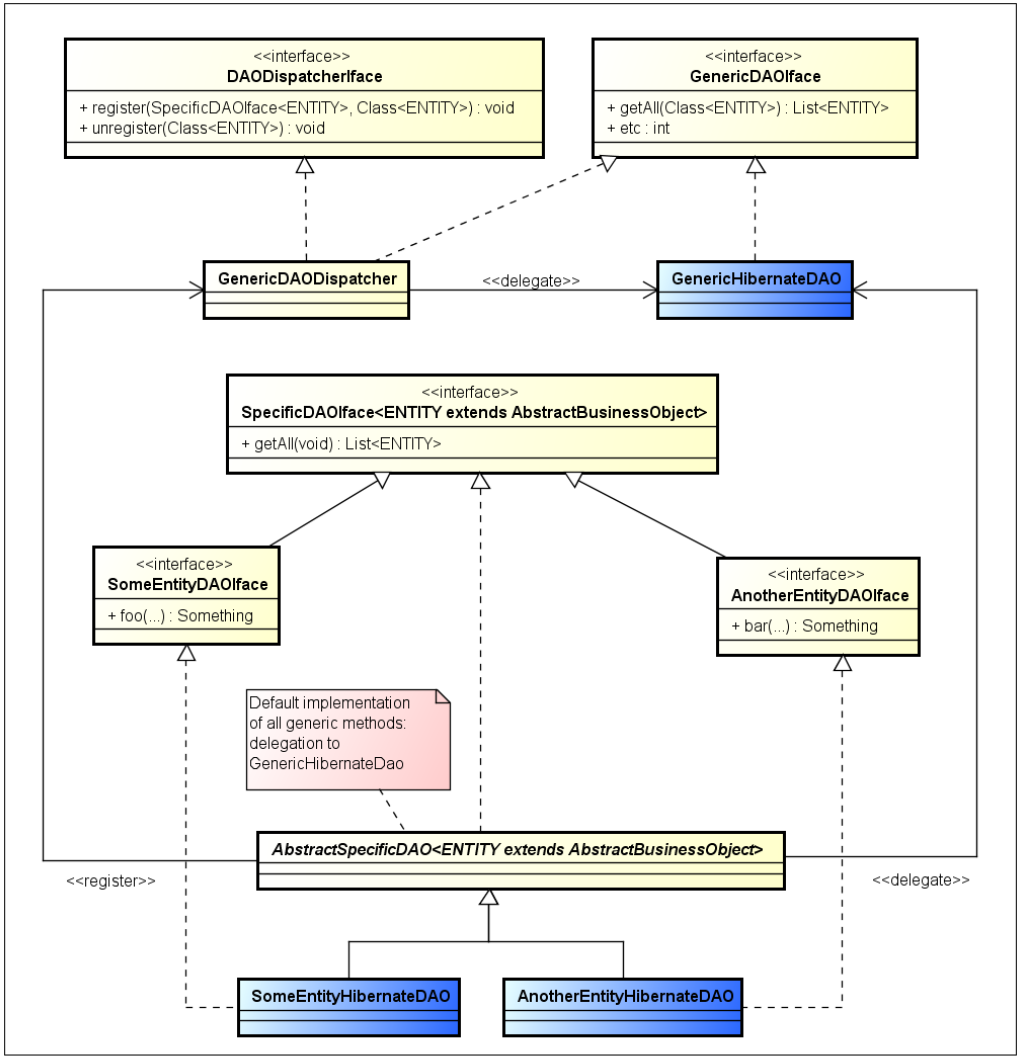
Dispatcher decisions (generic call)



Transaction diagram



The big picture



DAO Dispatcher

Pros

- Encapsulates database technology
- Generic functionality implemented only once
- Programmer can consistently override generic functionality
- Higly reusable
- Good testability
- Write only what you really need
- Perfectly supports standard use case
  - Simple app does not need many specific calls
  - With growing app, programmer can easily switch to specific DAOs and more complex DB logic

Cons

- Complex
  - Not easy to implement
  - New programmer must grasp the concept to fully utilize the DAO

References

- http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming) – Encapsulation at Wikipedia
- http://c2.com/cgi/wiki?DontRepeatYourself – Don’t repeat yourself definition
- http://martinfowler.com/eaaCatalog/registry.html - Registry pattern described by Martin Fowler
- http://www.hibernate.org/ - Hibernate JPA provider
- http://www.xprogramming.com/Practices/PracNotNeed.html - You’re NOT gonna need it

Questions