BOF5619 - Lean Beans (are made of this): Command pattern vs. MVC

Michael Bar-Sinai (@michbarsinai)
http://www.iq.harvard.edu/people/michael-bar-sinai
http://mbarsinai.com

Philip Durbin (@philipdurbin)
http://www.iq.harvard.edu/people/philip-durbin
http://greptilian.com

IQSS
The Institute for Quantiative Social Science
HARVARD UNIVERSITY

Agenda

1. Intro, real world challenges
2. MVC, MVC in Java EE, classic command
3. Command pattern adaptations
    - Java EE
    - Modern programming
    - Permission system
4. The Lean Bean Design Pattern

Slides:
https://github.com/IQSS/javaone2014-bof5619

Code:
https://github.com/IQSS/dataverse

Two real world challenges

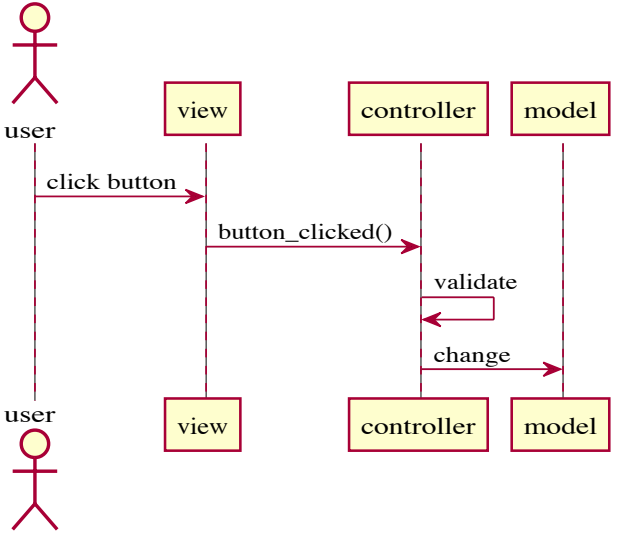
Dataverse 4.0 requirements

1. Maximal code re-use between API and GUI
2. Host sensitive data with granular permissions

https://github.com/IQSS/dataverse

Classic MVC

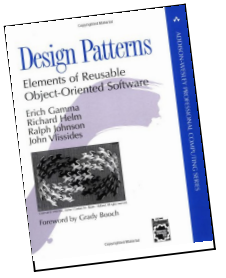
Goal: **Separate data from its representation**



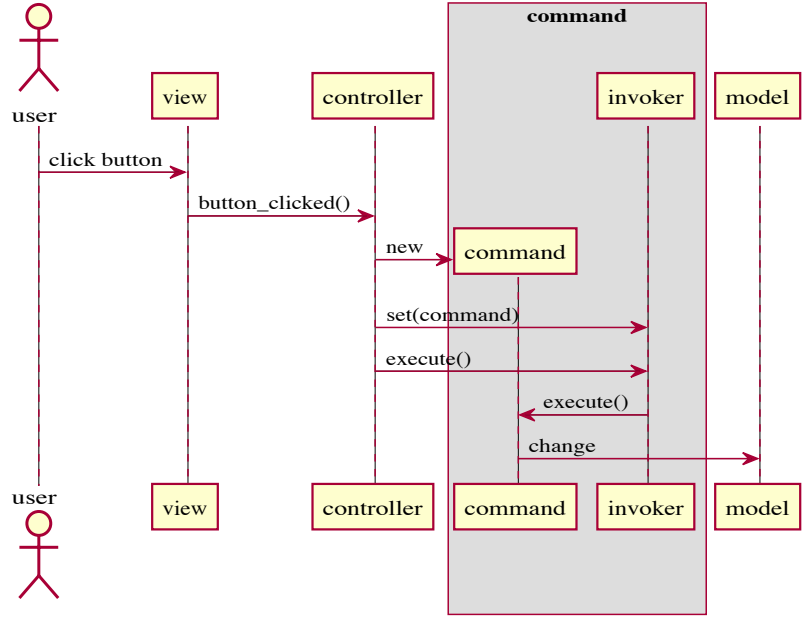
- Model: Business objects
- View: What the users sees
- Controller: Manipulates the model according to inputs from the view

Developed by Trygve Reenskaug at Xerox PARC in 1979. Not a GoF pattern.

The Command Pattern



Goal: **Capture operations on the model as data**



A request is a first-class object.

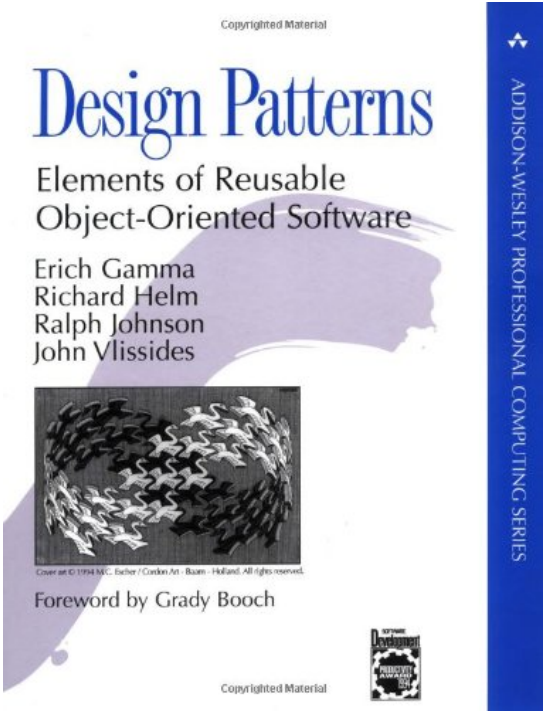
Official GoF definition

Command Pattern:

Encapsulate a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

Objects in play:

- Client
- Command
- Receiver
- Invoker



Waitresses and Cooks

- Customer
- Order
- Waitress
- Cook

You Are In Command Now

(Admiral Piett)

Giving Commands

1. stop people with droids
2. check identification

The Command interface

```
public interface Command {
    void execute();
}
```

Complete CheckIdCommand

```
public class CheckIdCommand implements Command {

    private Suspect suspect; // receiver

    public CheckIdCommand(Suspect suspect) {
        this.suspect = suspect;
    }

    public void execute() {
        try {
            System.out.println("Id for " + suspect.getName()
                                    + " is " + suspect.getId());
        } catch (Exception ex) {
            System.out.println("Move along, move along.");
        }
    }

}
```

The Client

```
System.out.println("# Mos Eisley checkpoint");

Suspect obiwan = new Jedi("Obiwan"); // receiver
Command checkIdCommand = new CheckIdCommand(obiwan);

StormTrooper stormTrooper = /* Recruit trooper here */
stormTrooper.setCommand( checkIdCommand ); // invoker
stormTrooper.execute(); // Move along, move along.
```

- ties all the object together
- command bound to receiver
- invoker is given a command to execute, and told to execute it

One Invoker, Two Commands

```
StormTrooper stormTrooper = /* TX-421 */

System.out.println("# Mos Eisley checkpoint");
Suspect obiwan = new Jedi("Obiwan"); // receiver
Command checkIdCommand = new CheckIdCommand(obiwan);
stormTrooper.setCommand(checkIdCommand);
stormTrooper.execute(); // Move along, move along.

System.out.println("# Death Star hangar");
Ship falcon = new Ship("Millenium Falcon"); // receiver
Command checkShipCommand = new CheckShipCommand(falcon);
stormTrooper.setCommand(checkShipCommand); // same invoker, new command
stormTrooper.execute(); // No one on board.
```

Same stormtrooper given different commands.

Stormtrooper/Invoker

```
public class StormTrooper {

    private Command command;

    public StormTrooper(Command command) {
        this.command = command;
    }

    public void setCommand(Command command) {
        this.command = command;
    }

    public void execute() {
        command.execute();
    }

}
```

Stateful: setCommand, another call to execute.

Suspect/Receiver

```
public class Suspect {
    // fields, constructors, getters

    public String getId() {
        return id;
    }
}
```

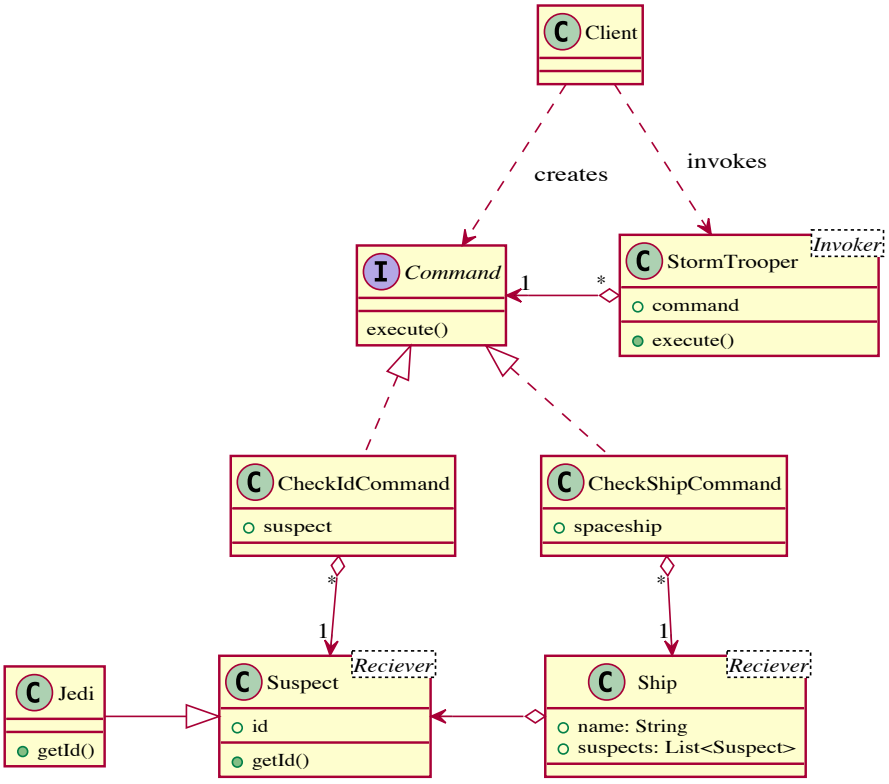
Jedi were always in a class of their own.

```
public class Jedi extends Suspect {

    @Override
    public String getId() {
      throw new RuntimeException("Jedi mind trick!");
    }
}
```

Another Command Example

```
public class CheckShipCommand implements Command {

    Ship ship; // receiver

    public CheckShipCommand(Ship ship) {
        this.ship = ship;
    }

    public void execute() {
        System.out.println(
            ship.getSuspects().isEmpty() ? "No one on board."
                    : "Found " + ship.getSuspects().size()
                                + " suspects in " + ship.getName());
    }

}
```

Star Wars Class Diagram



But what's in it for us?

- Why not call `execute()` on the command from the client code?
- Why not put the functionality of `execute()` in a normal method instead of creating new objects?
- *Yet another* level of indirection? Don't we have enough of these?

Code as Data

The command pattern allows us to treat a block of code (and its parameters) as data. Benefits:

- Reuse methods from different controllers
- Store commands in data structures
- Queue commands
- Create command macros, a sequence
- Commands calling other commands
- Test better
- Log better, show command history
- Easy to extend to support undo operations

We can do all this (and more!) with commands.

But there's one very special thing we can also do with commands...

Ignore Them.

Why We Chose Command

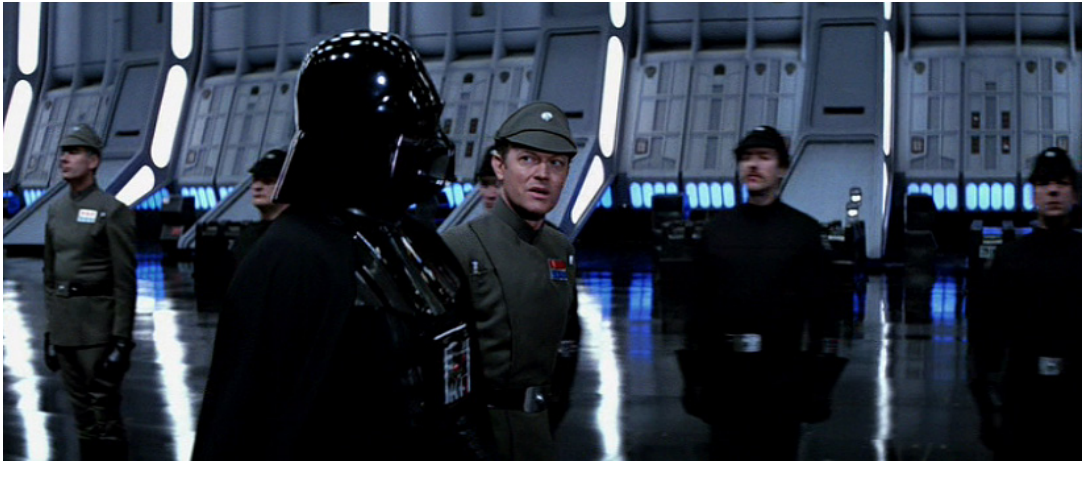
For Dataverse 4.0, a Java EE 7 app which needs to support sensitive data and a full API, we wanted:

1. Maximal code re-use between the API and the UI
2. By-design, permission-based security.

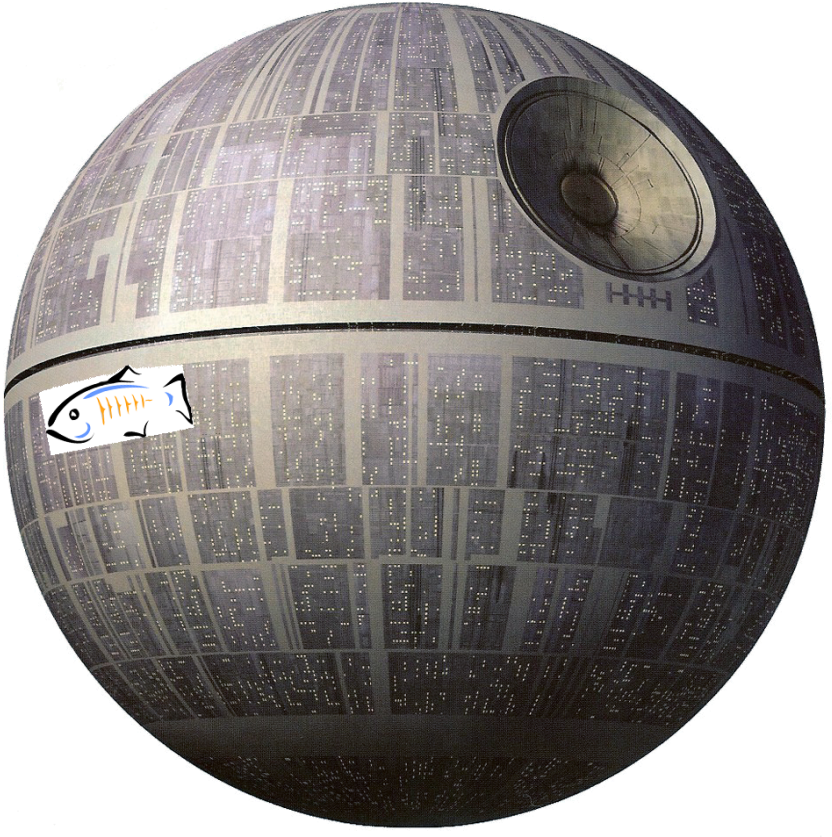
With some extensions and infrastructure (shown later) - we got just that.

Back on schedule

You may dispense with the pleasantries, Commander. I'm here to put you back on schedule.

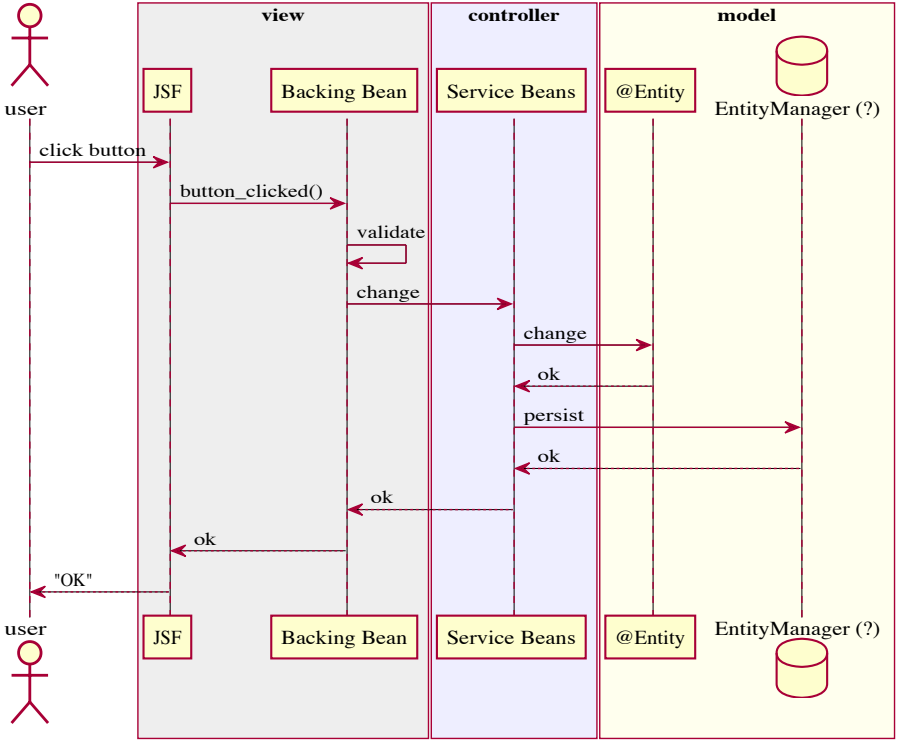


Back to Java EE



MVC in Java EE

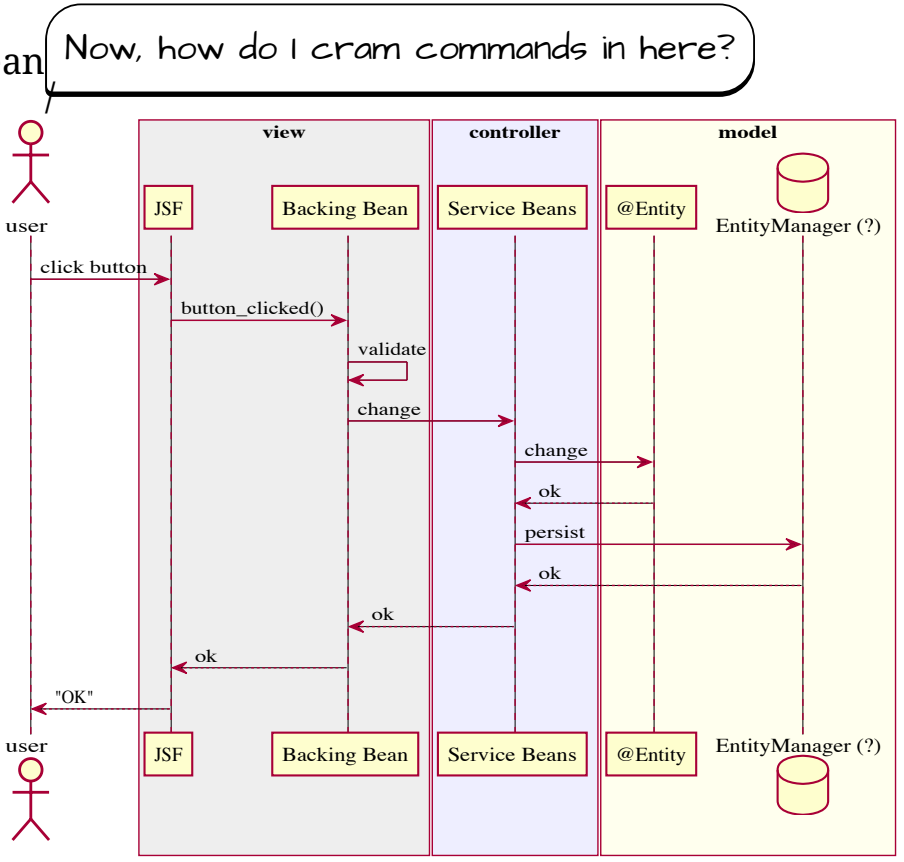
Balancing clean design and practicality.



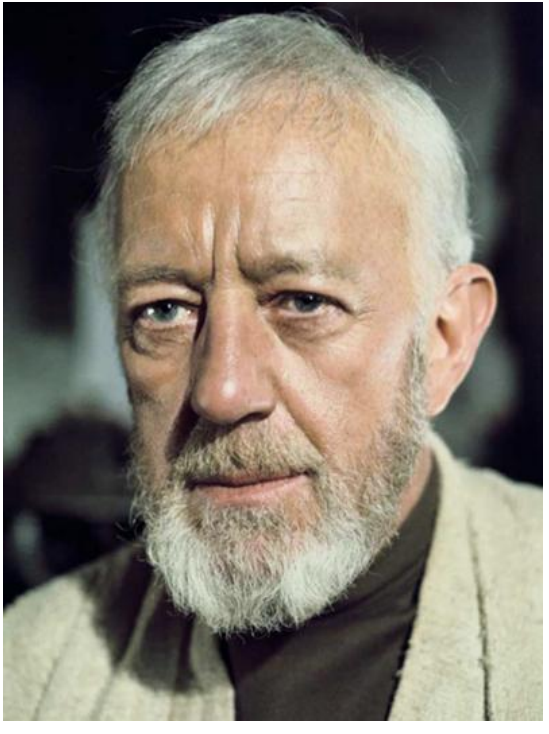
Note: We show one interpretation here - there are other interpretations as well.

MVC in Java EE

Balancing clean



Note: We show one interpretation here - there are other interpretations as well.



I feel a great disturbance in the source.

Nothing New Under The Sun

EJB Design Patterns: Advanced Patterns, Processes, and Idioms by Floyd Marinescu, 2002

"Use the Command pattern to wrap business logic in lightweight command beans that decouple the client from EJB, execute in one network call, and act as a façade for the EJB layer." (p. 19)

Command vs. Session Façade + Business Delegate

No invoker: "Applied to EJB, the Command Server class is a stateless session bean that accepts a command as a parameter and executes it locally." (p. 22)

Struts™

http://www.theserverside.com/news/1369776/Free-Book-EJB-Design-Patterns

Command, Adapted

We have adapted the Command Pattern to support permissions and execute in a Java EE environment

```
public interface Command<R> {

  public R execute( CommandContext ctxt ) throws CommandException;

  public Map<String,DvObject> getAffectedDvObjects();

  public User getUser();

}
```

- Modern touches
  - `execute` is an expression (not a statement)
  - Generics
  - Command objects can be (and mostly are) immutable
- The `CommandContext` parameter is used to allow the command access to server resources
  - No dependency injection needed → Test using standard JUnit!

Real code, comments removed.

Command, Adapted

We have adapted the Command Pattern to support permissions and execute in a Java EE environment

```
public interface Command<R> {

  public R execute( CommandContext ctxt ) throws CommandException;

  public Map<String,DvObject> getAffectedDvObjects();

  public User getUser();

}
```

- A command acts on one or more receivers of type `DvObject`
  - Think of these as "files" and "directories".
- A command must be issued by a `User`

Real code, comments removed.

Command, Adapted

We have adapted the Command Pattern to support permissions and execute in a Java EE environment

```
public interface Command<R> {

  public R execute( CommandContext ctxt ) throws CommandException;

  public Map<String,DvObject> getAffectedDvObjects();

  public User getUser();

}
```

`CommandException` has 3 sub-classes:

- `IllegalCommandException` - Command makes no sense
  - e.g. move a parent to its descendant
- `PermissionException` - Issuing user doesn't get to perform this operation over the affected receivers
- `CommandExecutionException` - Oops, our bad

Real code, comments removed.

Command Invoker, Adapted

We called the invoker *Engine*, as the term is more familiar.

```
public interface DataverseEngine {

  public <R> R submit( Command<R> aCommand ) throws CommandException;

}
```

- Modernized invoker - replaced the `set`→`execute`→`get` sequence with a single method call.
- `submit` is a generic method, allowing type-safe execution of any command.

Real code, comments removed.

Command Engine in the Wild

Creating a Dataset from the API (JAX-RS `@Path` bean).

```
@EJB
protected EjbDataverseEngine engineSvc;
...
Dataset ds = ... // get dataset here
Users u = ... // get the user here
try {
    Dataset managedDs =
          engineSvc.submit( new CreateDatasetCommand(ds, u) );
    return okResponse( "created dataset " + managedDs );

} catch ( XXXCommandException ex) {
    //...
}
```

Code adapted for slide. Original file at: https://github.com/IQSS/dataverse/blob/master/src/main/java/edu/harvard/iq/dataverse/api/Dataverses.java#L123

Command Engine in the Wild #2

Same command used from a JSF backing bean.

```
@EJB EjbDataverseEngine commandEngine;
...
Command<Dataset> cmd;
try {
    if (editMode == EditMode.CREATE) {
        cmd = new CreateDatasetCommand(dataset, session.getUser());
    } else {
        cmd = new UpdateDatasetCommand(dataset, session.getUser());
    }
    dataset = commandEngine.submit(cmd);
    ...
} catch ( CommandException ex) {
    ...
}
return "/dataset.xhtml?id=" + dataset.getId() + ...
          + "&faces-redirect=true";
```

Code adapted for slide.

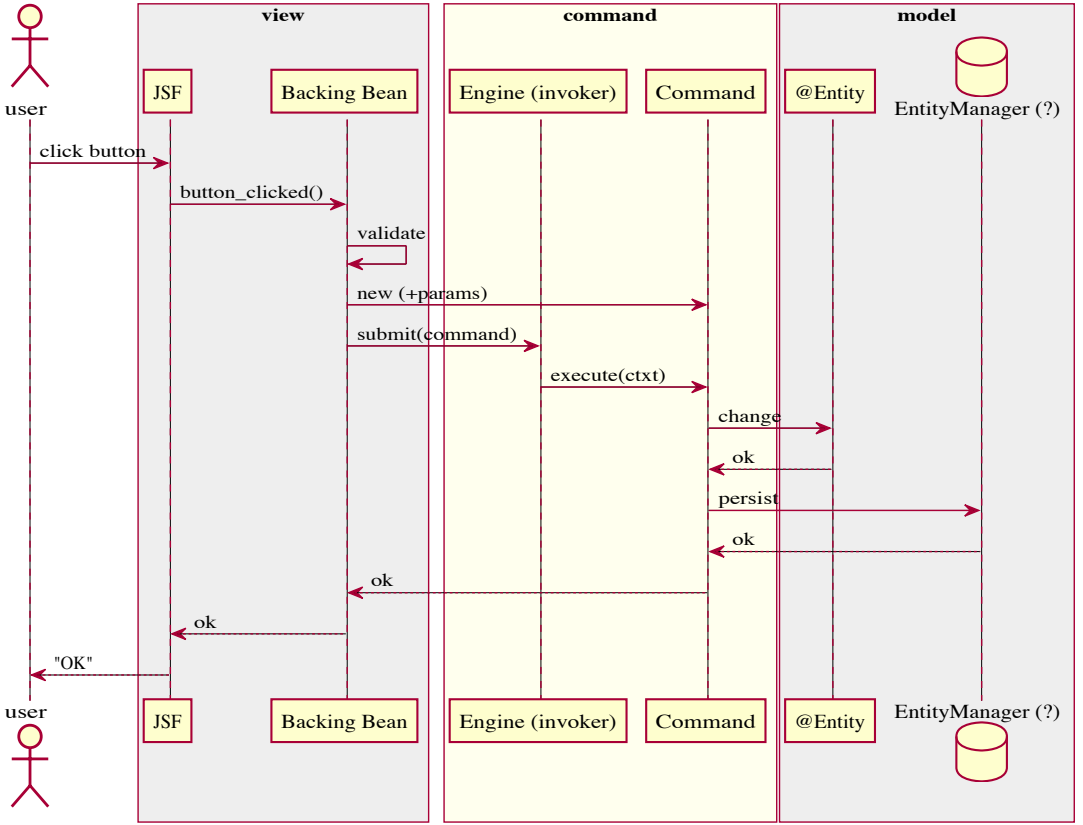
Command Engine in a Wild Loop

Listing the content of a dataverse object (think `ls`).

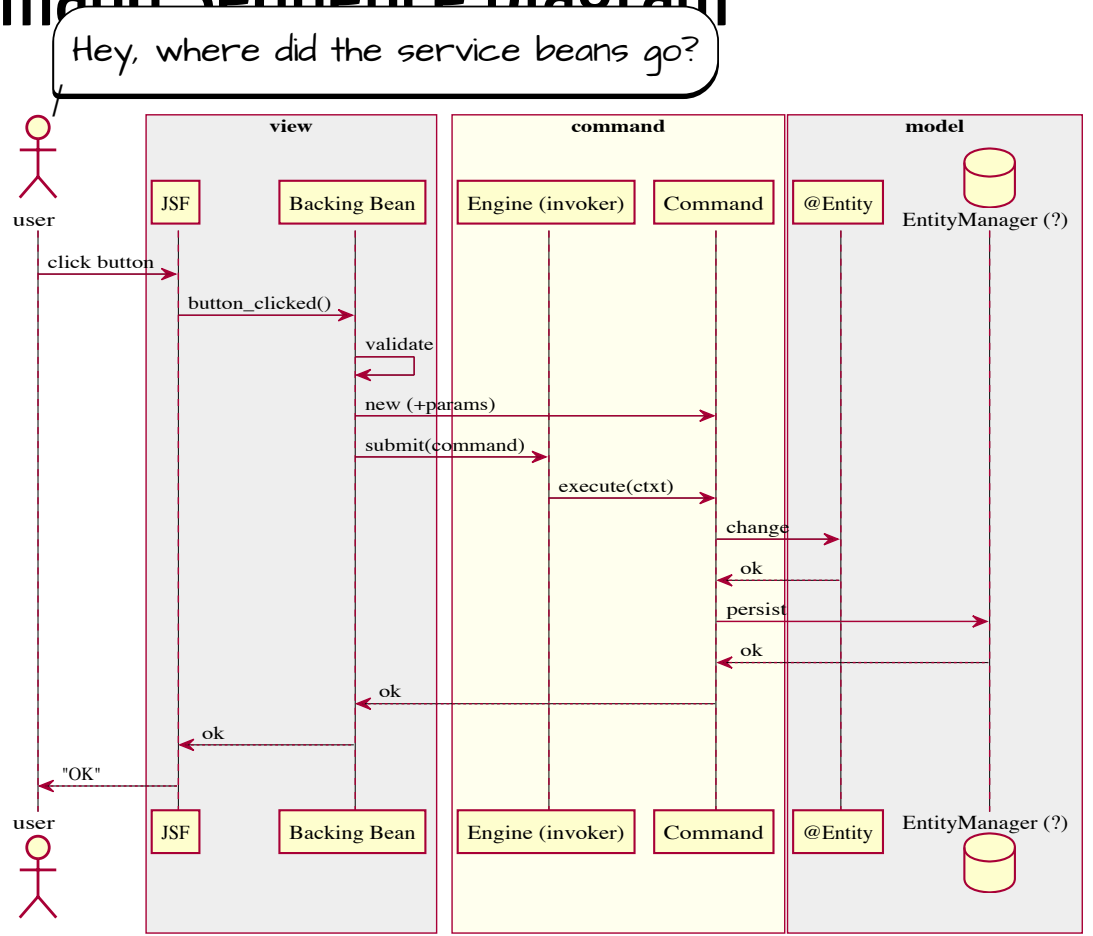
```
try {
  for ( DvObject o :
      engineSvc.submit(new ListDataverseContentCommand(u, dataverse)) ) {
    // add o to the output
  }
} catch (IllegalCommandException ex) {
    return errorResponse( Response.Status.FORBIDDEN, ... );

} catch (PermissionException ex) {
    return errorResponse(Response.Status.UNAUTHORIZED, ... );

} catch (CommandException ex) {
    logger.log(Level.SEVERE, "Error while " + messageSeed, ex);
    return errorResponse(Status.INTERNAL_SERVER_ERROR, ... );
}
```

Code adapted for slide. Actual code has some extra neat stuff outside the scope of this BOF. See https://github.com/IQSS/dataverse/blob/master/src/main/java/edu/harvard/iq/dataverse/api/AbstractApiBean.java#L123

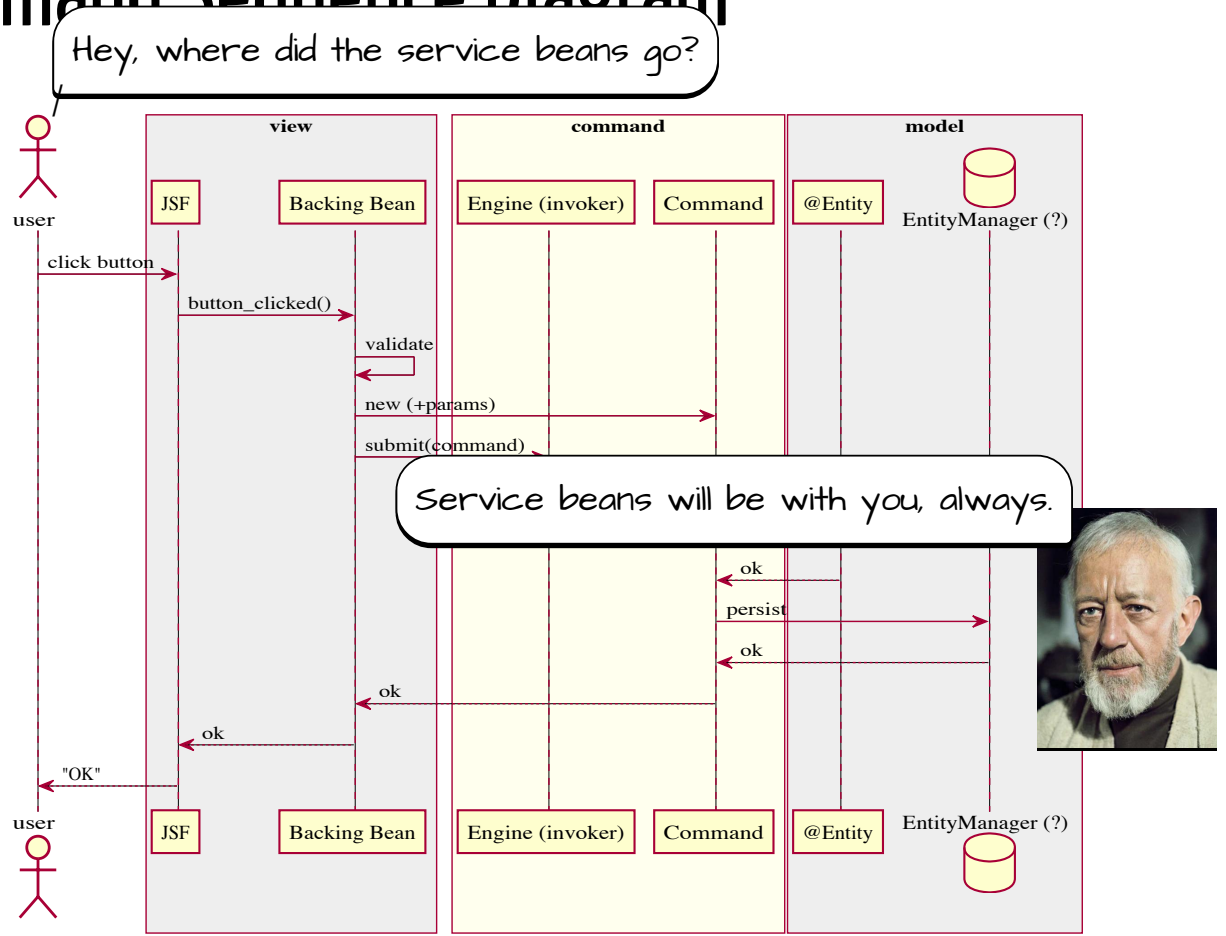
Command Sequence Diagram



Command Sequence Diagram



Command Sequence Diagram



Sample Command: Rename a Dataverse

```
@RequiredPermissions( Permission.UndoableEdit )
public class RenameDataverseCommand
          extends AbstractCommand<Dataverse>{
  private final String newName;
  private final Dataverse renamed;
  public RenameDataverseCommand( User aUser,
                    Dataverse aDataverse, String aNewName ) {
    super( aUser, aDataverse );
    newName = aNewName;
    renamed = aDataverse;
  }

  @Override
  public Dataverse execute(CommandContext ctxt) throws CommandException {
    if ( newName.trim().isEmpty() ) {
      throw new
          IllegalCommandException("Dataverse name cannot be empty", this);
    }
    renamed.setName(newName);
    return ctxt.dataverses().save(renamed);
  }
}
```

Permissions and Commands

- In code, permissions live in an enum. Each permission:
  - Holds a basic descriptive text.
  - States which objects it applies to.
- In the database, permission live in a bit field
  - Very fast, but must be kept under 64.
  - No DB joins needed
- Testing if a permission exists in a permission set is a bitwise operation.

```
public enum Permission {
  Discover("See and search content", DvObject.class),
  Download("Download the file", DataFile.class),
  AccessUnpublishedContent("Access unpublished content",
                           DvObject.class),
  AccessRestrictedMetadata("Access metadata marked as\"restricted\"",
                           DvObject.class),
  UndoableEdit("Edits that do not cause data loss", DvObject.class),
  DestructiveEdit("Edits that cannot be reversed, such as deleting data",
                  DvObject.class),
// ...
```

Supporting Multiple Receivers

When commands involve more than a single receiver, the `RequiredPermissionMap` annotation can be used.

```
@RequiredPermissionsMap({
  @RequiredPermissions( dataverseName = "moved",
                        value = {Permission.UndoableEdit, Permission.GrantPe
  @RequiredPermissions( dataverseName = "source",
                        value = Permission.UndoableEdit ),
  @RequiredPermissions( dataverseName = "destination",
                        value = Permission.DestructiveEdit )
})
public class MoveDataverseCommand extends AbstractVoidCommand {
  // ...
  public MoveDataverseCommand( User aUser,
                               Dataverse moved, Dataverse destination ) {
    super(aUser, dv("moved", moved),
           dv("source",moved.getOwner()),
           dv("destination",destination) );
    this.moved = moved;
    this.destination = destination;
  }
```

Command Composition

A dataset has a published version, accessible by everyone, and a draft version, accessible by the team only. We composed existing commands to get the latest version accessible to the `User` issuing the `Command`:

```
@RequiredPermissions( Permission.Discover )
public class GetLatestAccessibleDatasetVersionCommand
       extends AbstractCommand<DatasetVersion>
// ...
@Override
public DatasetVersion execute(CommandContext ctxt) throws CommandException {
    DatasetVersion d = null;

    try {
        d = ctxt.engine()
                .submit(new GetDraftDatasetVersionCommand(u, ds));
    } catch(PermissionException ex) {}

    if ( d == null ) {
        d = ctxt.engine()
                .submit(new GetLatestPublishedDatasetVersionCommand(u,ds));
    }

    return d;
}
```

Easy Testing

Since the command and the context are POJOs, we can mock them easily.

```
//...
@Before
public void setUp() {
  testEngine = new TestDataverseEngine( new TestCommandContext(){...});
//...
@Test
public void testValidMove() throws Exception {
  testEngine.submit(
      new MoveDataverseCommand(null, childB, childA));

  assertEquals( childA, childB.getOwner() );
  assertEquals( Arrays.asList(root, childA), childB.getOwners() );
}

@Test( expected=IllegalCommandException.class )
public void testInvalidMove() throws Exception {
  testEngine.submit(
      new MoveDataverseCommand(null, childA, grandchildAA));
  fail();
}
```

Given that:

- Service beans act on model objects,
- Commands act on model objects, and
- Commands work well for us,

Can we remove all service beans?

Can we remove all service beans?

Yes

- Command context can give direct access to the entity manager, JMS resources and the like, so commands could use them directly.
- There's just one problem…

Can we remove all service beans?

Yes

- Command context can give direct access to the entity manager, JMS resources and the like, so commands could use them directly.
- There's just one problem…

It's the Wrong Question

Given that:

- Service beans act on model objects,
- Commands act on model objects, and
- Commands work well for us,

~~Can~~ Should we remove all service beans?

Should We Remove All Service Beans?

Probably Not

We tried that. Didn't work well, since the commands became too detailed.

Current Status - a more balanced approach

Commands deal with: Operations on model objects.

Service Beans deal with: save, update, delete and various lookups of model objects (e.g. findById).

Should We Remove All Service Beans?

Probably Not

We tried that. Didn't work well, since the commands became too detailed.

Current Status - a more balanced approach

Commands deal with: Operations on model objects.

Service Beans deal with: save, update, delete and various lookups of model objects (e.g. findById).

Hence, we call this:

the

Lean Bean Design Pattern

Lean Beans (are made of this)

- Actions on models done by Command objects
- CRUD done by lean beans

Benefits

- Code as data
- Reuse commands from various places
- Permission validation baked into the system
- Commands are POJOs:
  - Reusable outside of Java EE
  - Testable using JUnit
    - Since we use beans, which are easier to mock than EntityManagers
- Easy to find functionality - look at the class' name

Lean Beans (are made of this)

- Actions on models done by Command objects
- CRUD done by lean beans

Downside

- Some infrastructure needed
  - Engine
  - Permission annotations
- Requires some learning - not a mainstream solution

Also, we're just starting this - so not a lot of experience yet.

Future Work

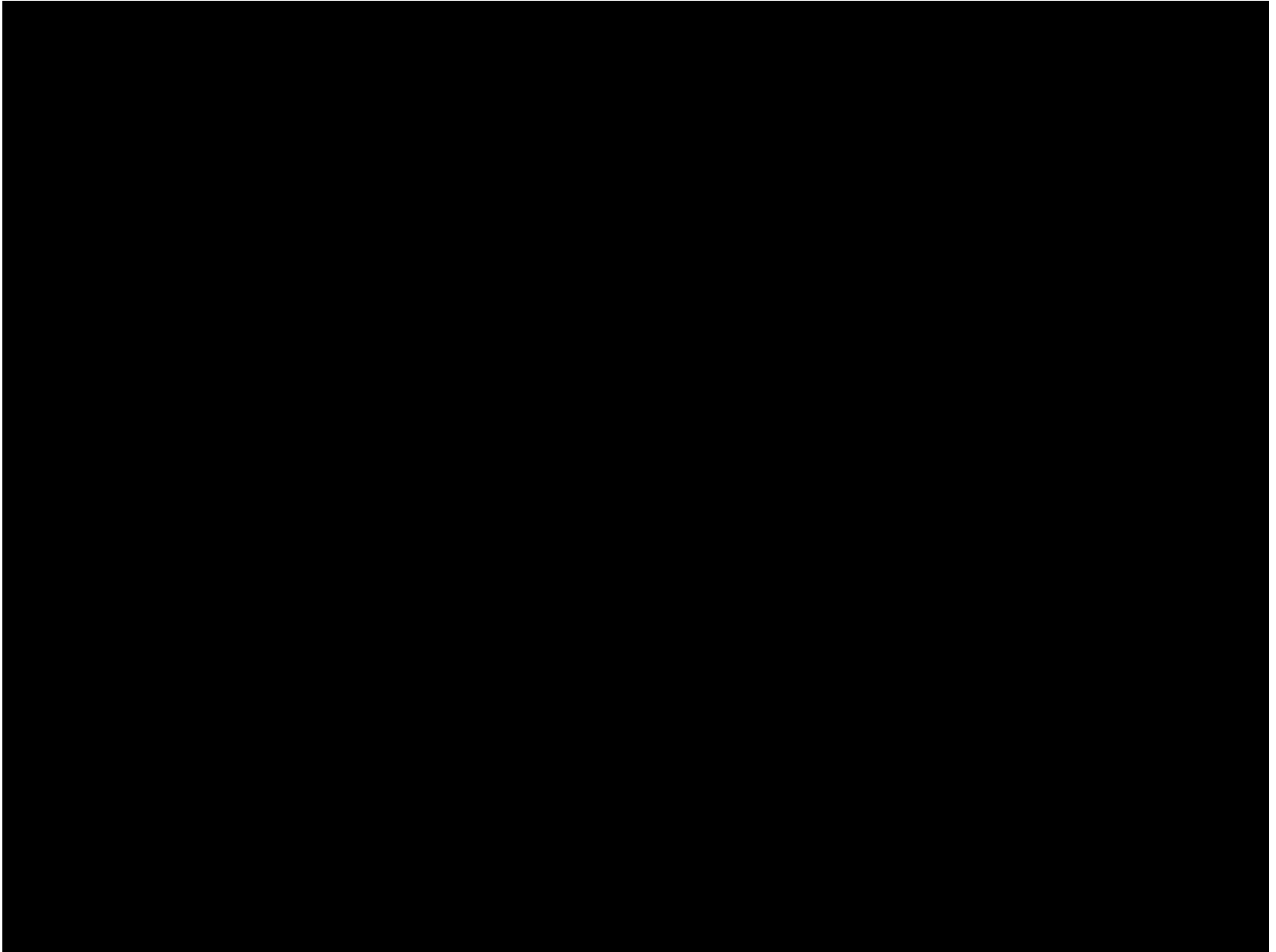
Some issues we already found out, and will deal with soon:

As Annotations are static, required permissions can't be dynamic

This conflicts with, e.g. The Decorator pattern. We will use a static-dynamic combo, where the basic command implementation uses reflection to return the required permissions, but subclasses can override this behavior.

Permission pre-flight check

Current implementation requires an actual object to work on, but the database layer allows for permission checks using the entity's id only - no real need to retrieve the object. When it makes sense, we need to take advantage of this. Somehow.



Thanks

Visit the IQSS data science team at http://datascience.iq.harvard.edu

Dataverse project @ GitHub: https://github.com/IQSS/dataverse

Slides and sample code from this talk: http://iqss.github.io/javaone2014-bof5619

Next up from IQSS:
Mike Heppler on JSF, PrimeFaces and Boostrap - Right here at Plaza A