

# Java Rich Internet Application Patterns

Daniel Pfeifer, Bruno Schäffer – Canoo



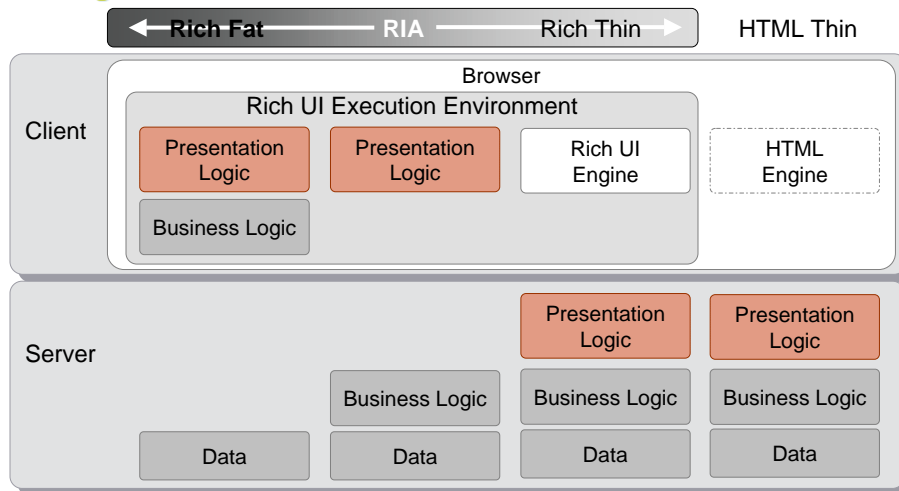
Daniel Pfeifer, Bruno Schäffer – Canoo  
**Java Rich Internet Application Patterns**

## Overview

- Rich Internet Architectures
- Presentation model
- Component factories
- Validation
- Accessing business logic

# canoo

- Company
  - Based in Basel / Switzerland
  - About 40 employees & contractors
  
- Business
  - Services:
    - RIA consulting
    - Custom Rich Internet & Web application development
  - Products:
    - Java RIA library
    - Web application testing

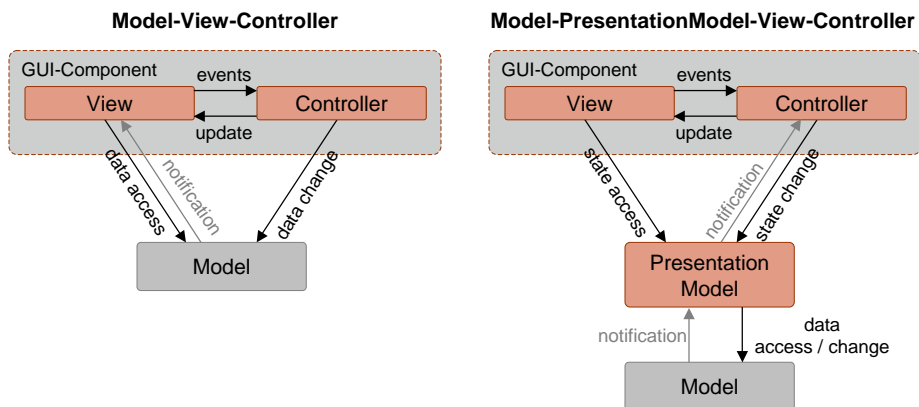


## RIA – User Interaction Patterns

- Master / Detail view, two panel selector
- Instant enabling / disabling, responsive disclosure
- Type lookahead
- Instant validation (syntactical / semantic)
- Silent error handling
- Drag & Drop
- Multiple (synchronized) windows /views

4

## Separation of Concerns

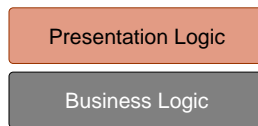


5

## Presentation Models

- Typical structure of GUI libraries:
  - Trees of component objects represent GUI
  - Event notification to react to user activity
  - E.g. GWT, SWT, Swing, ULC, ...

- Typical software layering:



- This simple layering does not support modular GUI building
- Presentation models help

## Qualities of Complex GUIs (1)

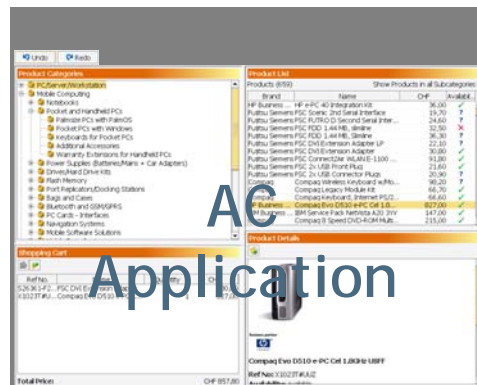
- Hierarchical component structures
- Hierarchy contains so-called application components (ACs)
  - Application-related, coarse-grained GUI elements
  - Consist of base components from GUI library
  - GUI developers invest in ACs
    - 1 AC  $\cong$  1 public (outer) class
    - Useful granularity for task assignments in team

## Example: ACs of the Online Shop



8

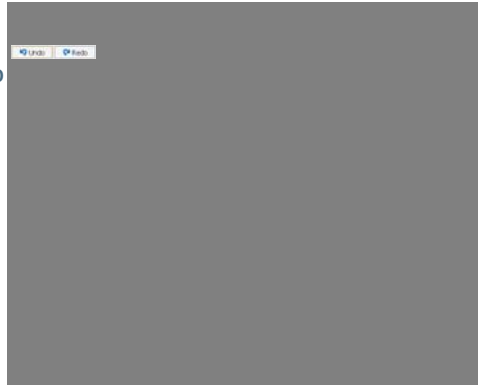
## Example: ACs of the Online Shop



9

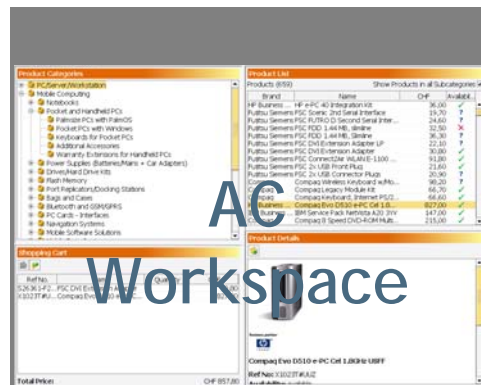
## Example: ACs of the Online Shop

AC  
Undo / Redo



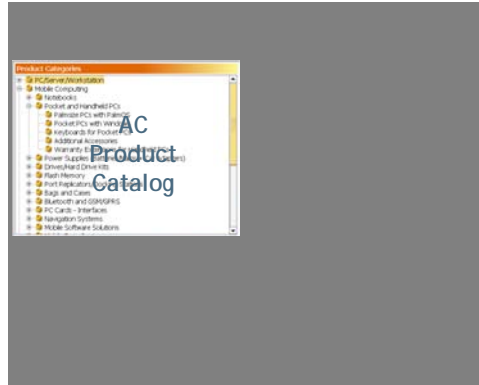
10

## Example: ACs of the Online Shop



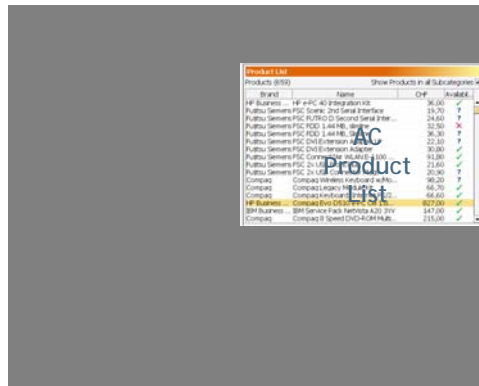
11

## Example: ACs of the Online Shop



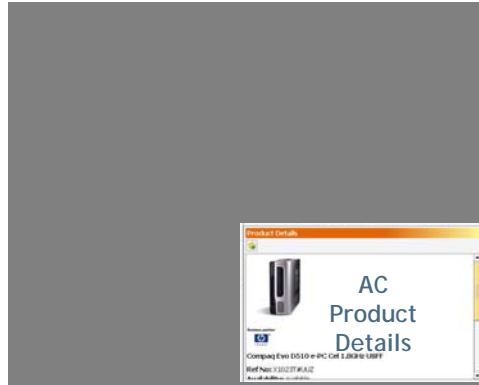
12

## Example: ACs of the Online Shop



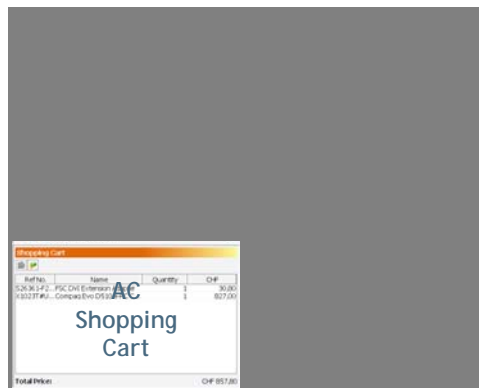
13

## Example: ACs of the Online Shop



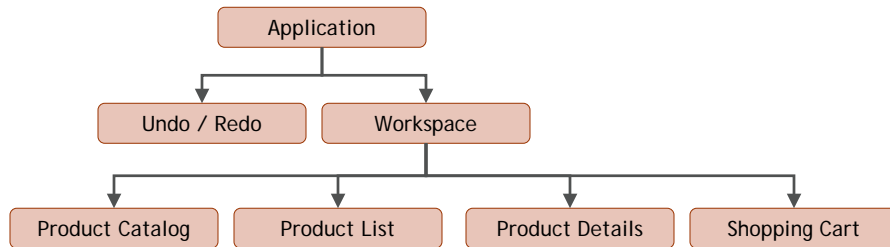
14

## Example: ACs of the Online Shop



15

## Resulting AC Tree



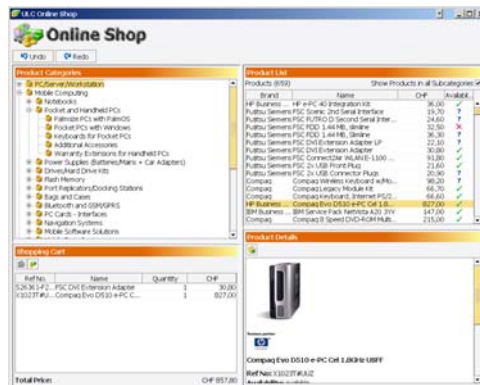
16

## Qualities of Complex GUIs (2)

- GUIs allow for redundant access to system parts
  - Different ACs display shared state
- The MVC pattern synchronizes ACs
  - Common state is represented by a separate object - the model
  - ACs trigger and observe model changes and update themselves accordingly

17

## Demo: Example of Shared AC State



18

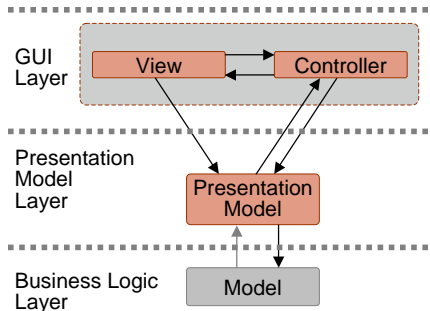
## Issues with Pure MVC

- MVC only applied to special GUI components (e.g. tables)
- Model represents business state
  - Presentation state not modelled
- Models and ACs get mixed up
  - E.g.: AC Product List must reference AC Product Catalog in order to access the selected categories
- Result: ACs are not decoupled
  - E.g.: AC Product List cannot be developed / tested / used without AC Product Catalog

19

## Solution: The Presentation Model

- Idea: Fully apply MVC to every single AC!
- Leads to the presentation model as a separate software layer
- But:
  - How to structure a related model?
  - What states to store in a model?
  - What further model qualities?
  - How to structure ACs then?
  - What framework support is useful and necessary?



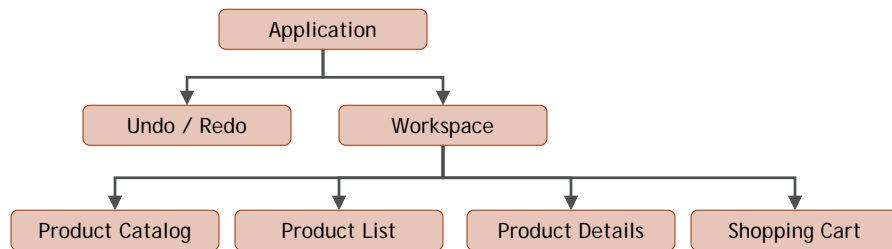
20

## Structure of Presentation Models

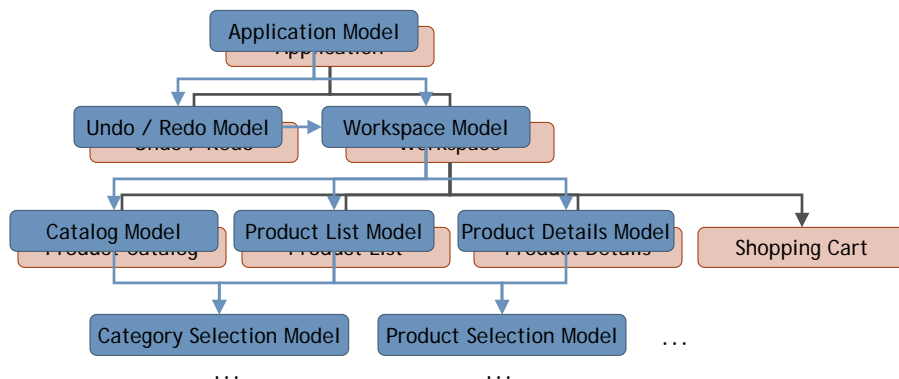
- ACs form trees
  - A root corresponds to a desktop window
- All (relevant) state of an AC is kept in a single model (instance), but
  - Models may share referenced submodels
- Presentation model instances form a directed acyclic graph (DAG)
  - Unique mapping from AC nodes to model graph nodes (typically not injective or surjective)

21

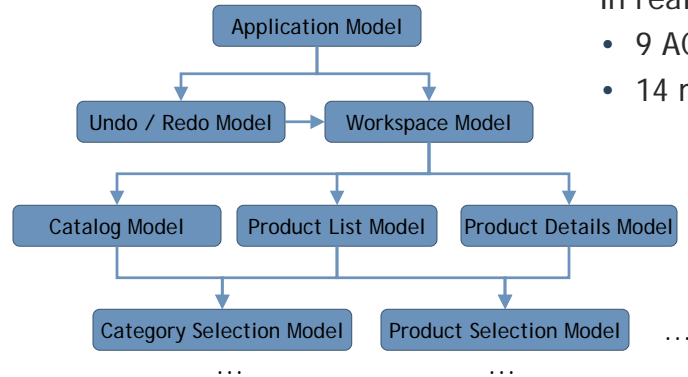
## Example: AC Tree (for Online Shop)



## Example: AC Tree + Model Graph



## Example: Model Graph Only



In reality:

- 9 AC classes
- 14 model classes

24

## States in Presentation Models

- Only mutable GUI states must be reflected, e.g.
  - A button's background color is (typically) immutable
  - Selection of a check box is (typically) mutable
- Abstracting state model, e.g.
  - Current windows size is (typically) irrelevant (handled by layout man.)
  - Selection of check box is relevant
- State representation boils down to
  - Primitive values for simple GUI states, e.g.
    - A boolean for checkbox selection
  - Submodel for complex ones, e.g.
    - For a list of selected elements

25

## Further Qualities of Pres. Models

- Model state must be fully observable
  - Via event notification
- An AC observes the mapped model and updates itself accordingly (MVC)
- When a model is initially bound to an AC it must synchronize its entire state with the AC
  - The model triggers a set of events to completely sync the AC

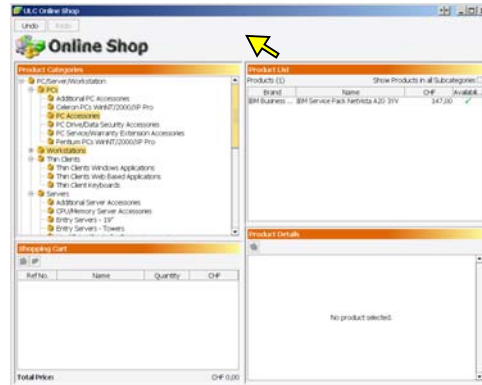
26

## Structure of ACs

- AC obtains the mapped model at construction time
- ACs implement 3 parts:
  - Code to build its GUI subtree from base components
  - Controller to propagate base component changes to the presentation model
  - Controller to propagate model changes to base components
- Two-way state update between an AC and its model

27

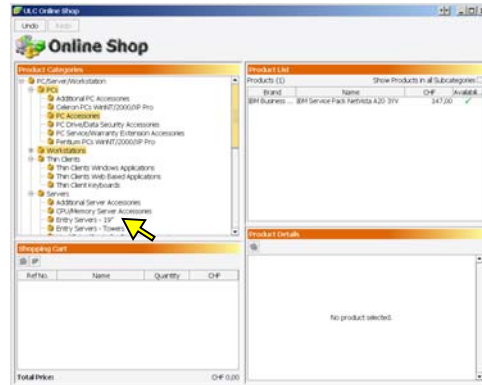
## Example: Two-Way Update



**Category Selection Model**

PCs  
 PC Accessories  
 Workstations

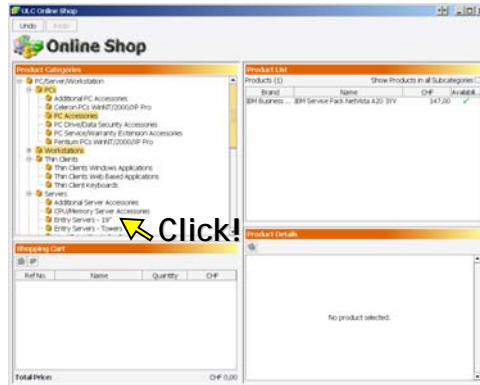
## Example: Two-Way Update



**Category Selection Model**

PCs  
 PC Accessories  
 Workstations

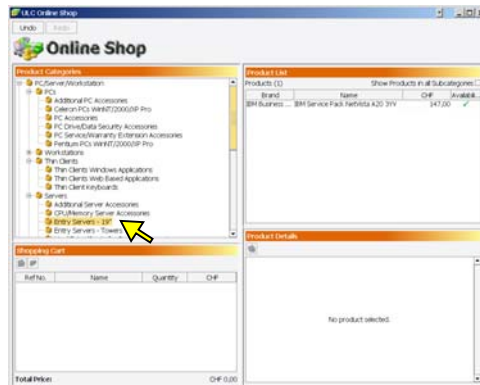
## Example: Two-Way Update



Category Selection Model

- PCs
- PC Accessories
- Workstations

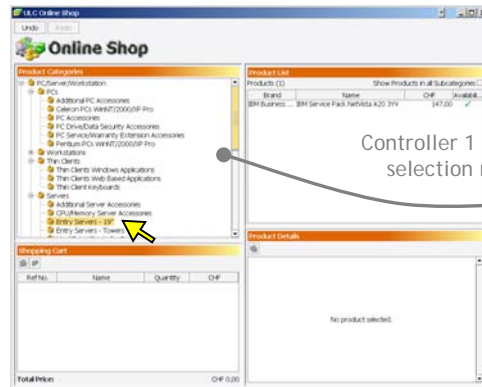
## Example: Two-Way Update



Category Selection Model

- PCs
- PC Accessories
- Workstations

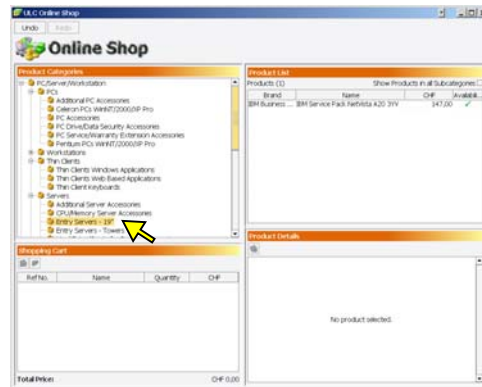
## Example: Two-Way Update



**Category Selection Model**

- PCs
- PC Accessories
- Workstations

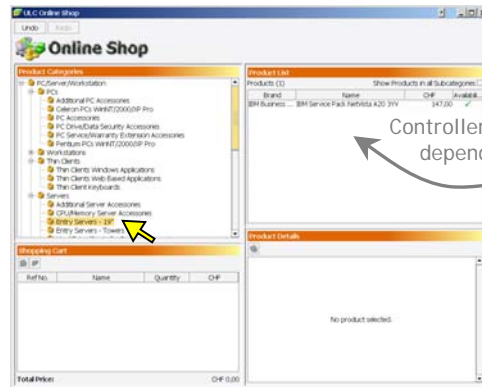
## Example: Two-Way Update



**Category Selection Model**

Entry Servers -19''

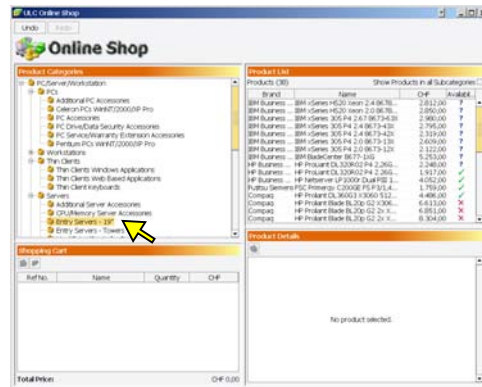
## Example: Two-Way Update



Controller 2 updates dependent ACs

Category Selection Model  
Entry Servers -19''

## Example: Two-Way Update



Category Selection Model  
Entry Servers -19''

## Framework Support

- A framework is useful!
- Templates for model structures:
  - To ease / standardize implementation of models
    - Abstract super classes / interfaces for models
    - List models, undo / redo models, ...
- Templates for ACs
  - Abstract classes with standard functionality supporting
    - I18N
    - Factory methods for base components
    - Factory methods for controllers

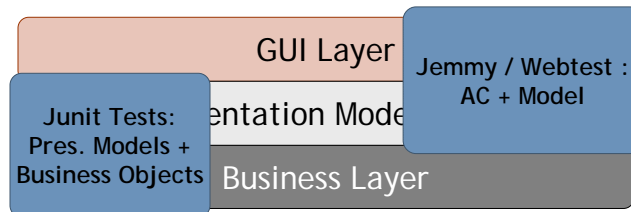
36

## Pros & Cons for Developers

- Cons:
  - Extra work for designing / implementing presentation layer
  - Extra work for additional controllers
- Pros:
  - ACs can be implemented / tested / used independently
  - Very uniform GUI code → eases team work and maintenance
  - Undo / redo almost for free
  - Dynamic locale changes very easy (via locale model)
  - Easier to exchange GUI library
  - Potentially cleaner business layer

37

## Improved Testability



- Independent testability of ACs and models
  - Freely choose width of test (according to granularity of ACs)
  - Test a lot without GUI (less Jemmy and Webtest)
  - Test a lot without DB layer (dummy models or business objs.)

38

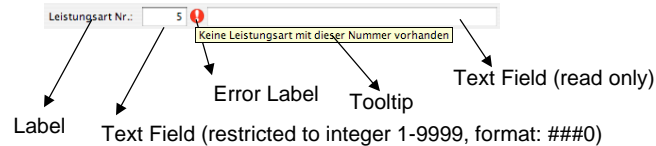
## What Else Do We Need?

- GUI component creation
- User input validation
- Interacting with business logic

39

## Component Factories

- Form components don't map well to GUI components



- GUI components are still too low-level
- Consistency is hard to achieve
- Changes are a nightmare

## Component Factories

- Component factories can help a lot
  - Factorize guy creation code
  - Two levels:
    - Domain independent factory, e.g. `createDateField()`
    - Domain specific factory, e.g. `createCustomerIdField()`
- Keeping consistency is way easier
- Global changes are a breeze
- Builder pattern emulates optional parameters

## Validation

- Syntactic validation
  - Can be handled by
  - Component → component factory configures component
  - Formatter → value objects
  - Example
    - Date: correct format and data according to the Gregorian calendar
- Semantic validation
  - A lot more challenging!

42

## Semantic Validation

- RIAs feature extensive semantic validation in the GUI
- Examples

Klient Nr: 4711

wöchentlich

1.1.2007 01.01.2006

Datum "Bewilligt von" muss kleiner oder gleich "Bewilligt bis" sein

- Challenges
  - Semantic validation is intrinsic to the business logic layer
  - Duplication of validation logic in presentation layer is expensive
  - Delegating validation to the business logic layer is costly
  - Semantic validation in the GUI needs subset but more
  - How to communicate validation errors?
  - Validation for RIA is both on the attribute and object level

43

## Semantic Validation

- How to factorize semantic validation?
  - Business object validates itself
    - Is business object available in presentation layer?
    - Might only be a data transfer object
  - Validation class (e.g. Spring validator)
  - Descriptive validation (e.g. commons validator)
  - Rule-based validation
- Referential validation best done in the business layer
  - Requires service access
  - Can only be done in the presentation layer for small data sets
- Validation method allows to specify attribute to be validated

44

## Semantic Validation

- How to communicate validation errors?
- Exceptions
  - Validation errors can be modelled by the exception
  - Flexibility to catch the validation error along the call stack
  - Any property setter can just throw the exception
  - Are exceptions the right way to signal common events?
- Error objects
  - Error object accumulates all validation errors
  - Must be passed as an additional parameter to the validator
  - Direct caller must handle it or pass it on

45

## Accessing Business Logic

- Kind of object the presentation logic should work with?
  - Business objects?
  - Data transfer objects?
- Business object model vs. services?
- Service granularity?

46

## Business Objects

- Pros
  - No need to define data container for presentation logic
  - Reuse validation logic
- Cons
  - Extends transaction context into presentation layer
    - Detach objects from the DB!
  - Lack of encapsulation
    - Reveals more business logic than is needed
  - Transparent lazy loading might be error-prone
    - Error-prone (if references are resolved in a detached object)
    - Navigational access might lead to unintended lazy loading
      - Must ensure that all required objects are loaded

47

## Data Transfer Objects

- Pros
  - Decouples from business logic
  - Deals with incomplete object graphs
- Cons
  - DTOs are dumb (e.g. no validation)
  - DTOs have to be implemented / generated / maintained
  - DTOs implemented as maps
    - More convenient to implement and maintain
    - Less convenient to access and loss of type safety

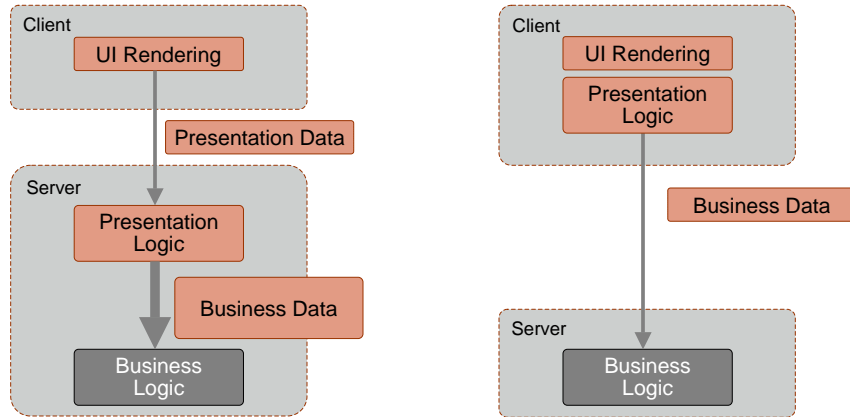
48

## Service Data Objects

- Addresses common problems of service data
  - Disconnected data graphs
  - Logging changes to the data graph
  - Generic and statically typed access to properties
  - Metadata available
  - Frameworks and tools included
  - Independent of environment (programming language)

49

## Service Granularity



50

## Service Granularity

- Rich Internet Applications
  - Need more frequent access to data
    - Validation, interaction patterns such as drill down
  - Remote clients cannot afford to transfer lots of data
- Coarse granularity promotes reusability
- Solutions:
  - Implement adapter services tailored to presentation needs
  - Generic DTOs & filter

51

## Business Service – Caching

- Rich Internet Applications
  - Need more frequent access to data
    - Validation, interaction patterns such as drill down, master-detail
  - Try to avoid displaying stale data
- Server-side caching
  - More useful for server-side presentation logic
  - Easier to avoid stale data
  - Cache can be shared
- Client-side caching
  - Stale data can hardly be avoided
  - More cache memory available

52

## Handling Service Data

- Service data format may be inconvenient
  - String is very common for all sorts of types
  - Different services may return different formats for the same type (even in the same company!)
- Consider using value objects
  - Fundamental abstractions in a business domain
  - Examples: monetary amount, account number, entry date

53

## Value Objects

- Key properties:
  - Values are abstractions of a problem domain
  - Values have no lifecycle
  - Values have no alterable state
  - Values are referentially transparent (i.e. no side effects)
- Implementation:
  - Internal representation (logical or canonical format)
  - Deals with invalid and unknown values
  - Implements `getValue()`, `toString()`, `equals()`, `hashCode()`, `isValid()`
  - Factory method and private constructor

54

## Infrastructure for Value Objects

- Formatter
  - Formats value object (according to locale)
  - Validates (incomplete) string syntactically
  - Parses string and converts it to logical format
- Converter
  - Converts proprietary format to logical format and vice versa
  - Used for service data which does not conform to the logical format.

55

## Summary

- POWAs (Plain Old Web Applications)
  - Easy on developers but a hassle for users
  - Server-side presentation layer simplifies a lot
- RIAs
  - A lot more challenging to developers
  - Presentation model is the pivotal concept
  - Additional infrastructure required
  - Partitioning has quite some impact

## Resources / Links

- Presentation model:  
<http://www.martinfowler.com/eaDev/PresentationModel.html>  
<http://http://www.jgoodies.com/articles/patterns-and-binding.pdf>
- Service Data Objects:  
<http://www.bea.com/dev2dev/assets/sdo/Next-Gen-Data-Programming-Whitepaper.pdf>
- Value Objects:  
<http://www.riehle.org/computer-science/research/2006/plop-2006-value-object.pdf>
- Validation:  
<https://validation.dev.java.net/>  
<http://http://www.springframework.org/docs/reference/validation.html>



## The Model Interface

```
public interface Model {
    // Common environment for all models:
    public ModelContext getModelContext();
    // Basic enabling of being observable:
    public void addPropertyChangeListener(String propertyName,
        PropertyChangeListener listener);
    public void addPropertyChangeListener(PropertyChangeListener listener);
    public void removePropertyChangeListener(String propertyName,
        PropertyChangeListener listener);
    public void removePropertyChangeListener(PropertyChangeListener listener);
    // To visit model graph (required for synchronization):
    public void visit(ModelVisitor modelVisitor);
    // To initially synchronize dependent Acs:
    public void synchronize();
    ...
}
```

60

## Pres. Model Code Sample

```
public class SimpleModel extends GenericPropertyModel {
    public static final String MY_FLAG_PROPERTY = "myFlag";

    public SimpleModel(ModelContext modelContext) {
        super(modelContext);
        defineProperty(MY_FLAG_PROPERTY, boolean.class);
    }

    public boolean isMyFlag() {
        return (Boolean)getProperty(MY_FLAG_PROPERTY);
    }

    public void setMyFlag(boolean value) {
        setProperty(MY_FLAG_PROPERTY, value);
    }
}
```

- A model with a single boolean state
  - Observation and synchronization feature covered in super class

61

## Code Sample: Associate AC

```
public class SimpleAK extends ModeledJComponent<SimpleModel> {
    private JCheckBox checkBox;

    public SimpleAK(SimpleModel model) {
        super(model);
    }

    protected JComponent createComponent(SimpleModel model, ...) {
        return checkBox = new JCheckBox("My Flag");
    }

    protected void createComponentChangeListeners(final SimpleModel model, ...) {
        checkBox.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                model.setMyFlag(checkBox.isSelected());
            }
        });
    }

    protected void createModelChangeListeners(SimpleModel model, ...) {
        model.addPropertyChangeListener(new PropertyChangeListener() {
            public void propertyChange(PropertyChangeEvent evt) {
                checkBox.setSelected(((Boolean)evt.getNewValue()));
            }
        });
    }
}
```

62