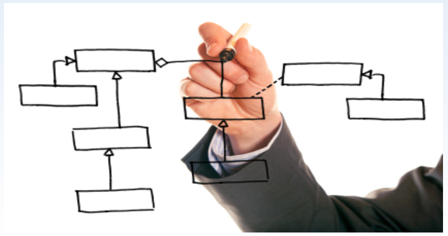
Employing a View Model API Based on Dynamic Proxies

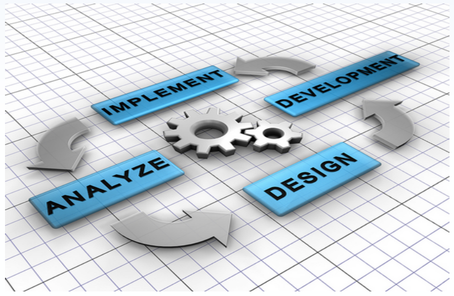
(Decoupling Persistent Domain Objects from UI)

harezmi it solutions

Who is Kenan Sevindik?

- Over **15 years** of enterprise application development **experience**
- Involved in creation and **development of architectures** in various enterprise projects
- Has **extensive knowledge and experience** about enterprise Java technologies, such as Spring, Spring Security, Hibernate, Vaadin...





Who is Kenan Sevindik?

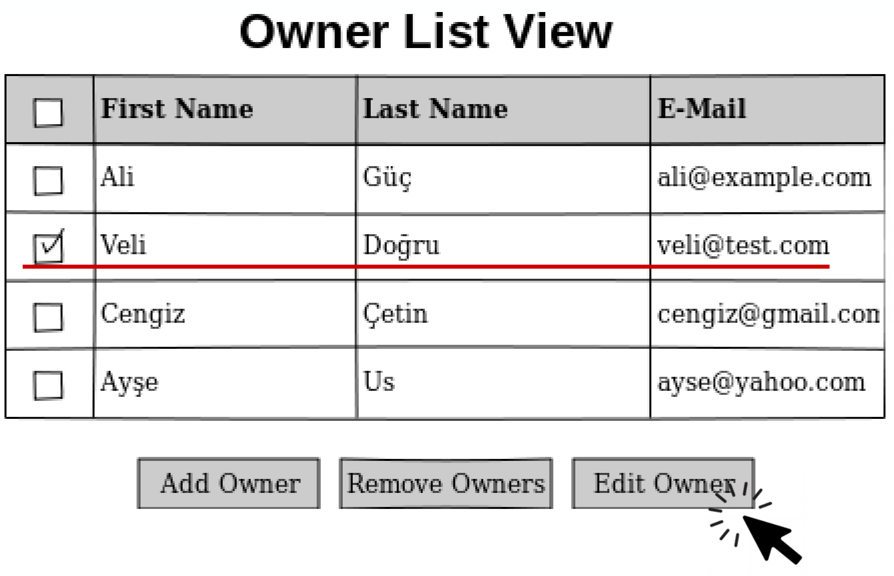
- Co-author of **Beginning Spring** Book (published in 2015)
- Founded **Harezmi IT Solutions** in 2011
- What Harezmi does?
  - Involves in **development** of enterprise Java applications
  - Offers **consultancy** and mentoring services
  - Organizes enterprise Java technologies related **trainings**

What is the Problem?

Several problems arise in case persistent **domain objects** fetched via JPA/Hibernate **are directly bound to UI** layer!

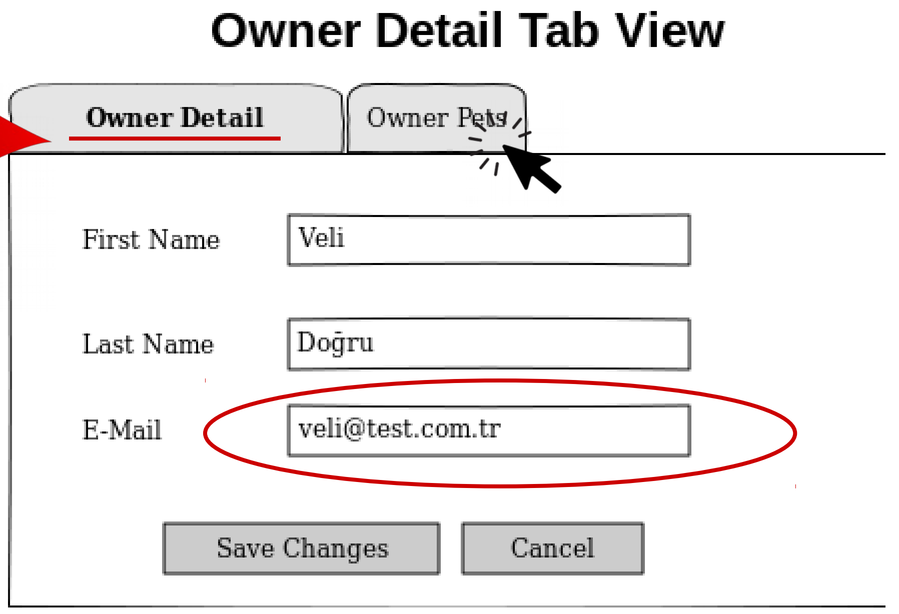
Scenario 1

- Let's assume we have a typical **master-detail** UI in which **Owner** and his/her **Pets** are displayed
- And assume, Owner – Pets association is **fetched on demand** (lazy)


Owner List View

☐	First Name	Last Name	E-Mail
☐	Ali	Güç	ali@example.com
☑	Veli	Doğru	veli@test.com
☐	Cengiz	Çetin	cengiz@gmail.con
☐	Ayşe	Us	ayse@yahoo.com

User lists a group of Owner entities, selects an Owner within the list, and navigates into the detail view

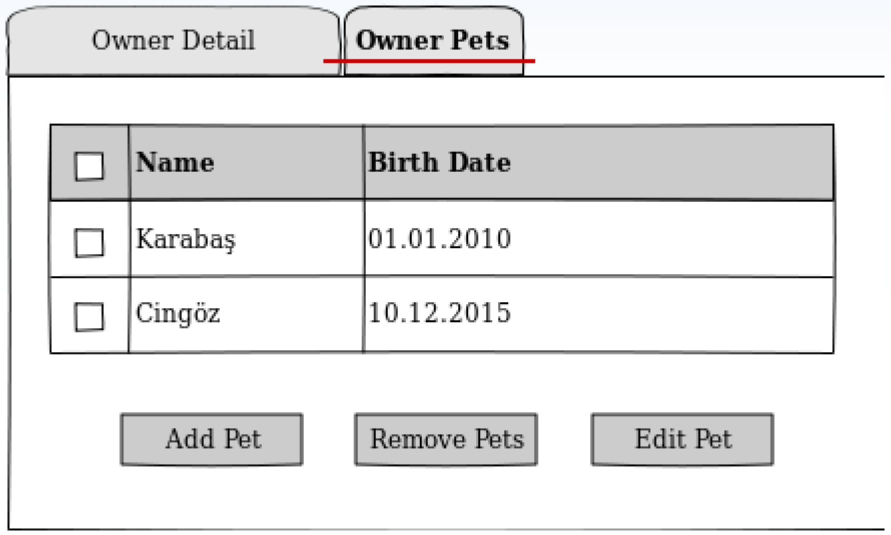

Owner Detail Tab View

First Name	Veli
Last Name	Doğru
E-Mail	veli@test.com.tr

User may change some of the properties of selected Owner entity

Scenario 1

Owner Pets Tab View



☐	Name	Birth Date
☐	Karabaş	01.01.2010
☐	Cingöz	10.12.2015

When he switches into the “Owner Pets” tab in order to list Pet records, **detached owner is re-attached to the persistence context in order to fetch contents of the pets collection from DB**. During this re-attachment, however, those **changes performed previously are flushed into DB** as a side effect of this re-attachment

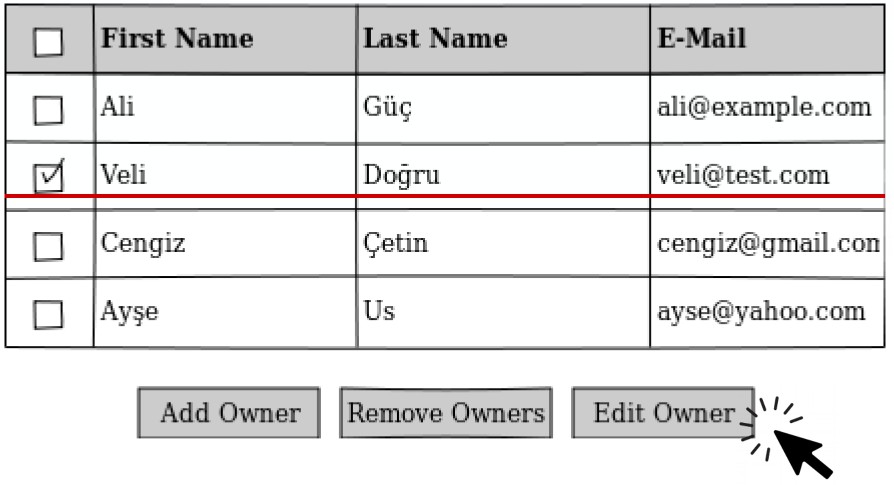
Scenario 1: Overview

- **Lazy association** should have been handled on its own
- Any **changes performed on the detached entity** should NOT have been reflected into the DB as a side effect of this handling of lazy association

harezmi it solutions

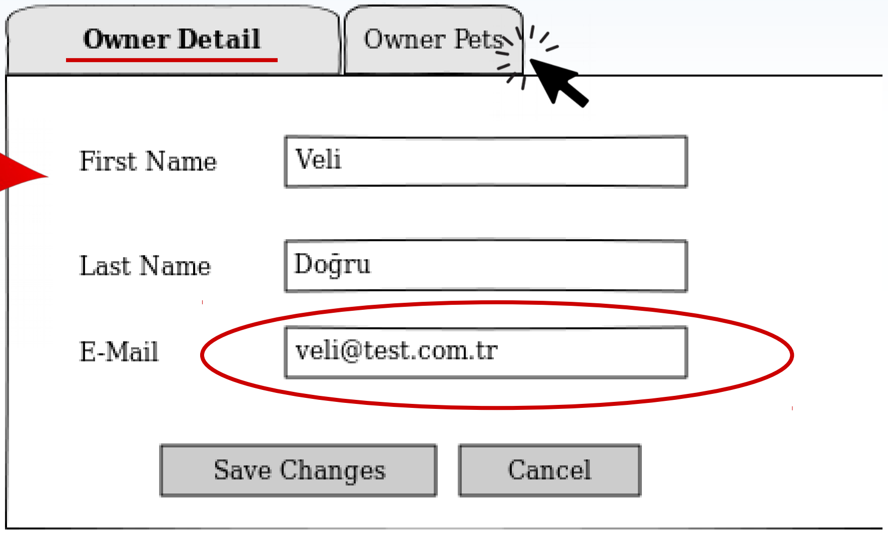
Creates Value...

Scenario 2



Owner List View

☐	First Name	Last Name	E-Mail
☐	Ali	Güç	ali@example.com
✓	Veli	Doğru	veli@test.com
☐	Cengiz	Çetin	cengiz@gmail.con
☐	Ayşe	Us	ayse@yahoo.com



Owner Detail Tab View

First Name: Veli

Last Name: Doğru

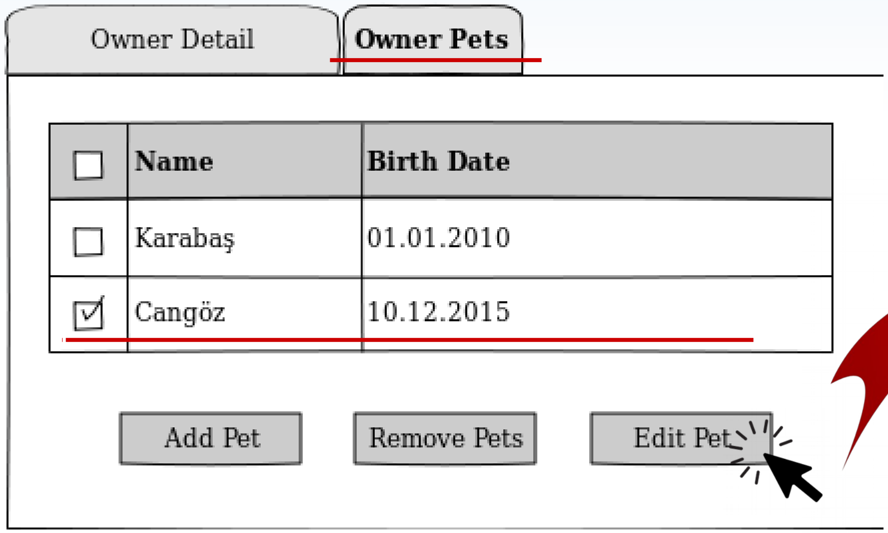
E-Mail: veli@test.com.tr

User lists a group of Owners, and selects an Owner from the list, then selects one and switches into the detail view

He performs changes over some of the properties of selected Owner, but **doesn't click “Save Changes”** button yet

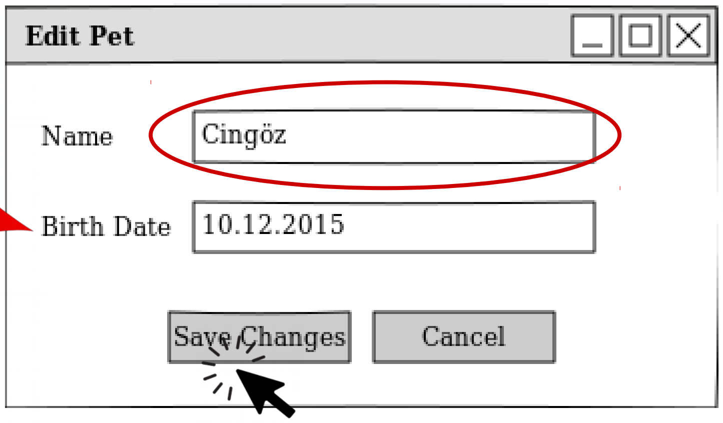
Scenario 2

Owner Pets Tab View



☐	Name	Birth Date
☐	Karabaş	01.01.2010
✓	Cangöz	10.12.2015

Edit Pet Dialog



Name: Cingöz
Birth Date: 10.12.2015

Later, without saving those changes in Owner entity, he switches into Owner Pets tab, and starts editing a Pet instance

User edits the Pet instance, and **clicks "Save Changes" button**. Depending on Cascade definitions, or attachment status of Owner entity, changes made on it might also be reflected into the DB

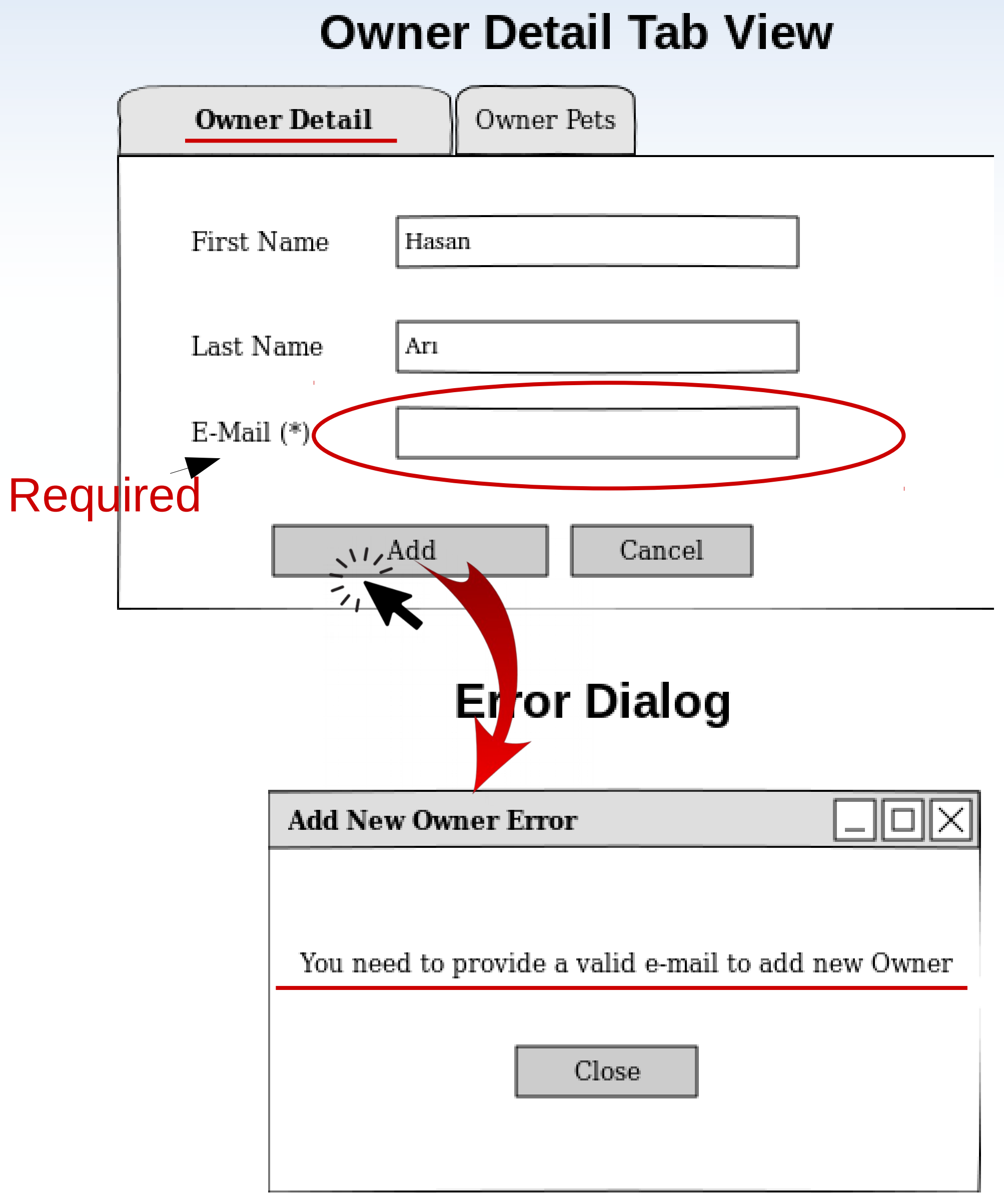
Scenario 2: Overview

- Both **changes in Owner and Pet entities** are reflected into DB when Pet's save button clicked
- User was not able to **cancel out changes made on Owner**, and reflect only those changes made on Pet
- Similar cases to this scenario may occur like **addition of a new Pet** entity
- or **deletion of an existing Pet** entity unintentionally

Scenario 2: Overview

- It is necessary that **old property values** or **additions/removals** from collections should be **stored somewhere else**, so that user **changes could be reverted** back whenever user decides to cancel out his operation

Scenario 3



First Name: Hasan
Last Name: Arı
E-Mail (*):

You need to provide a valid e-mail to add new Owner

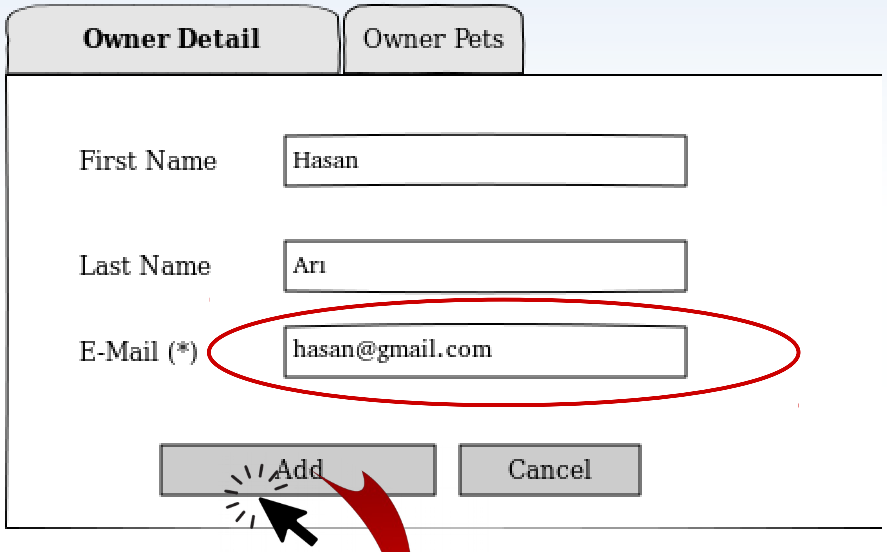
User enters details in order to create **a new Owner entity** in DB and **clicks “Add” button**

However, an **error is rised from within business layer** because of errornous data entered by the user

Creates Value...

Scenario 3

Owner Detail Tab View



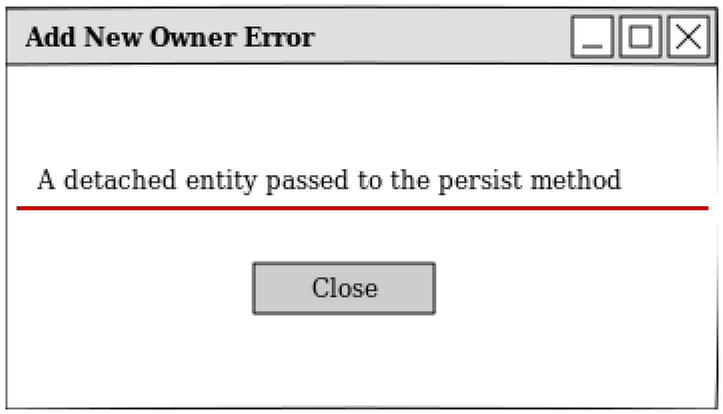
First Name: Hasan

Last Name: Arı

E-Mail (*): hasan@gmail.com

User corrects his fault, and **clicks “Add” button again**

Error Dialog



A detached entity passed to the persist method

Unfortunately, this time **persistence layer fails because of the state change** (identifier assignment etc) occured within domain object during the first attempt

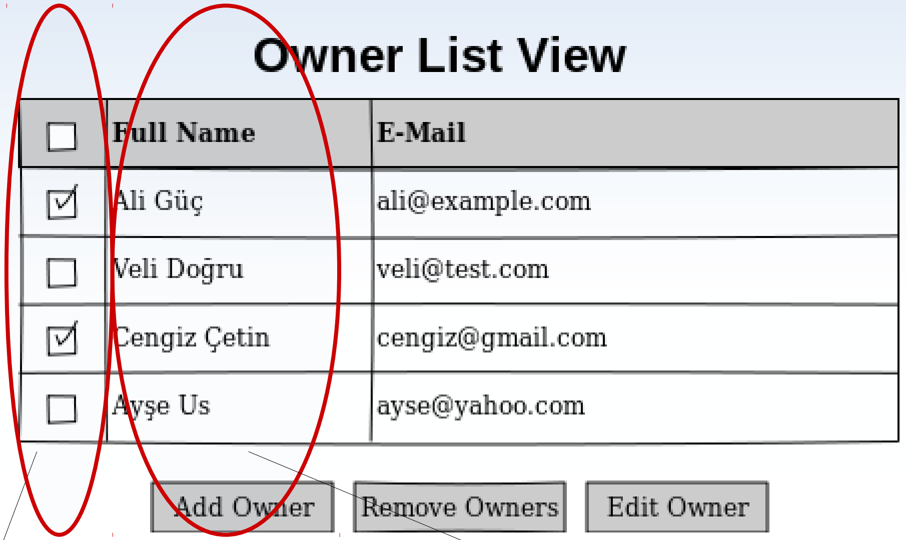
Scenario 3: Overview

- Persistence layer **assigns PK value** to identifier property of Owner entity which is in transient state during persist operation
- However, **transaction is rollbacked** and Owner is not saved into DB due to validation error occured
- During the second attempt, persistence layer thinks that Owner is in **detached state because of the assigned PK value** and fails the persistence operation

Scenario 3: Overview

- State of the domain object should have been **reverted back to its initial values** when transaction rolled back after the first attempt

Scenario 4



Owner List View

☐	Full Name	E-Mail
☑	Ali Güç	ali@example.com
☐	Veli Doğru	veli@test.com
☑	Cengiz Çetin	cengiz@gmail.com
☐	Ayşe Us	ayse@yahoo.com

UI specific changes in domain classes

It is required to store user selection information of Owner records somewhere in the application. The most practical place for this is Owner entity itself. For this purpose, a property is added into Owner class and used only to store that selection information. It is not related with the business logic at all.

Users may ask for Owners' names to be listed together as “fullName”, instead of firstName and lastName separated. Again, the easiest way to achieve this looks like adding a method as getFullName() to return firstName and lastName concatenated. This method has nothing to do with business logic, either.

Scenario 4: Overview

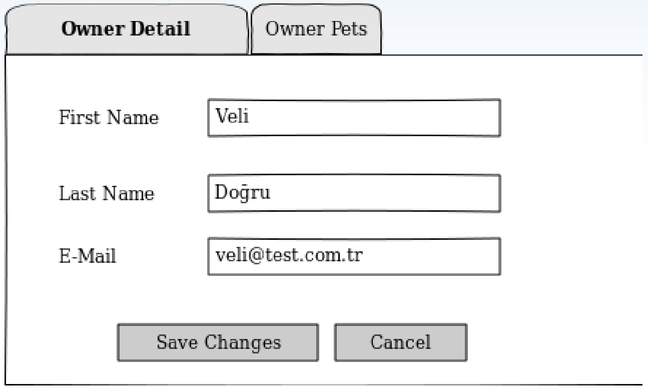
- **Properties and methods** which have **nothing to do with business logic** have been added into domain classes
- Those **domain classes** would be aimed to be used **as reusable units** in different applications
- In such a case, adding such properties and methods would **pollute domain model**

Problem Summary

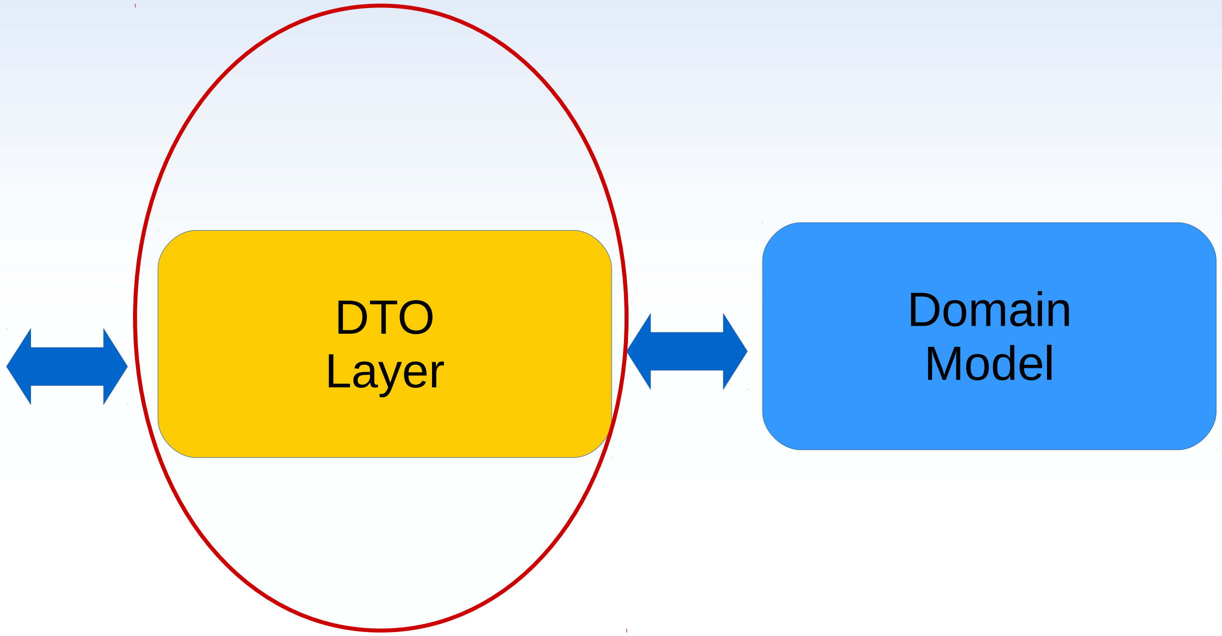
- All those problems arise because of **direct binding of persistent domain objects with UI** layer
- Persistent domain objects **should not be directly used within UI** layer
- Instead **there should be a separate layer** to handle displaying of UI specific information obtained from domain objects, grab user input and pass it into the service layer as well

Solution !: DTO Layer

Owner Detail Tab View



First Name	Veli
Last Name	Doğru
E-Mail	veli@test.com.tr



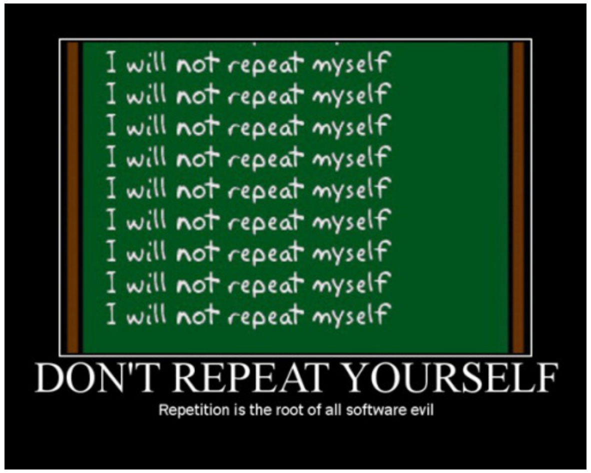
- **Data** needed by the UI is **obtained from domain objects** and **put into DTOs** possibly transformed by those DTOs to make it suitable for UI rendering
- **Use input** is first **accumulated within DTOs** as UI components are bound to DTOs
- The input accumulated within DTOs are **transferred into domain** instances **at appropriate times** and used within business logic execution

DTO, Wasn't It An Anti-Pattern?

- DTO predates back to **Value Object pattern**
- In the early days of Java EE application development, DTOs were mainly used to **transfer data between layers separated by network**
- Because **EJB method calls** were remote only, and those remote procedure calls were **causing performance problems**
- **DTOs** were then employed in order **to reduce communication overhead** of those RPCs

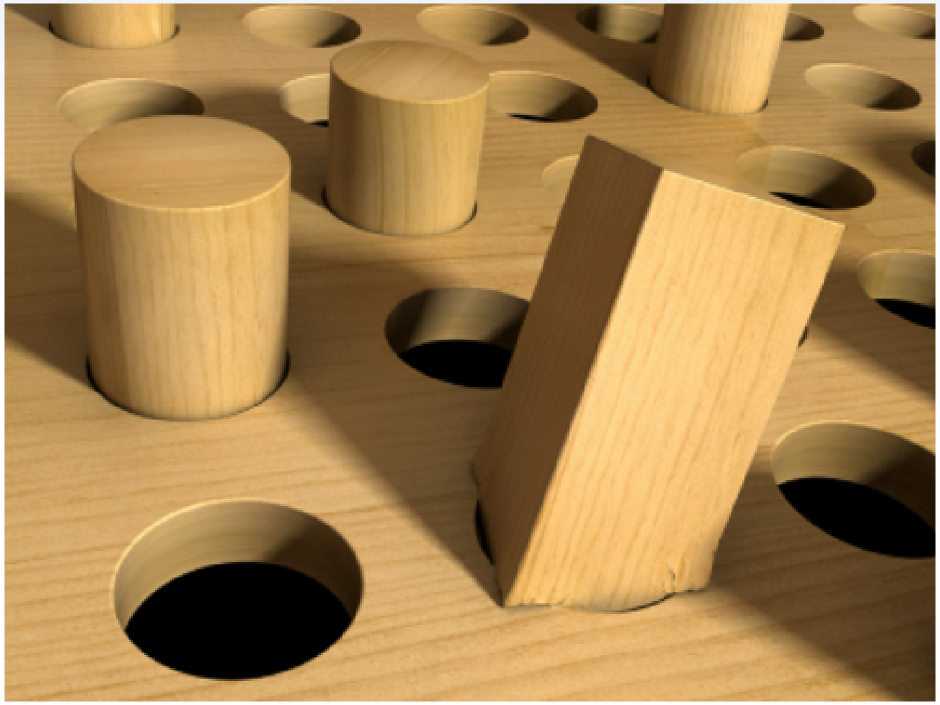
DTO, Wasn't It An Anti-Pattern?

- The most criticized aspect of DTO pattern is its **violation of DRY principle**
- According to DRY (dont repeat yourself) principle, **a task should only be implemented once and at only one single place** in the system
- Most of the time, many of the **properties and methods in domain classes are simply repeated in DTO** classes as well
- Apart from such repetition, only few other **properties and methods specific to DTOs** are added



DTO, Wasn't It An Anti-Pattern?

- DTOs, today are mostly considered **as anti-pattern** because of
  - such repetition in two different places
  - and strong encouragement by several popular UI and persistence frameworks to use domain classes directly in UI



Today's Common Situation

- Nowadays, domain instances are usually **fetched from DB**, using a persistence framework, such as JPA/Hibernate
- Afterwards, they are **directly bound to UI** components which are developed using a UI framework, like JSF or Vaadin
- Hence, transferring **user input from over domain objects directly into the DB** and vice versa is a mainstream approach

Revision in Naming: **View Model**

- Unfortunately, we've thought that such a naming like DTO or Value Object may cause underestimation to the need of **separating UI and domain layers** from each other
- Therefore, entitling our solution with **a different name** might be useful in terms of revealing its real benefits in our enterprise application architectures
- Our preference was to use **”View Model”** as it reveals its direct relationship with UI layer more than the word, DTO

Problem with DRY Still Exists!

- However, revision in the naming alone doesn't help us to get away from the **core of the problem**
- How such a separate View Model layer can be implemented **without violating DRY principle**?

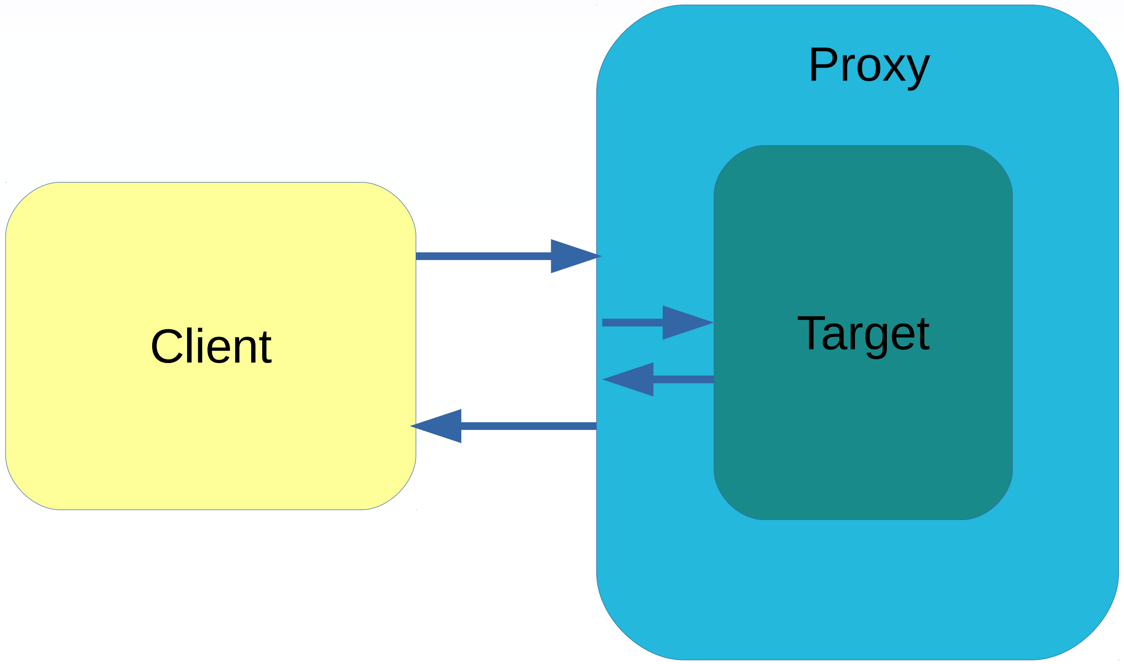
Solution : Dynamic Proxy Class Generation !

- View Model classes can be generated out of domain classes dynamically at runtime using **Proxy pattern!**

Proxy Pattern

Proxy is of same type with its target, and it intercepts method calls occuring between client and the target

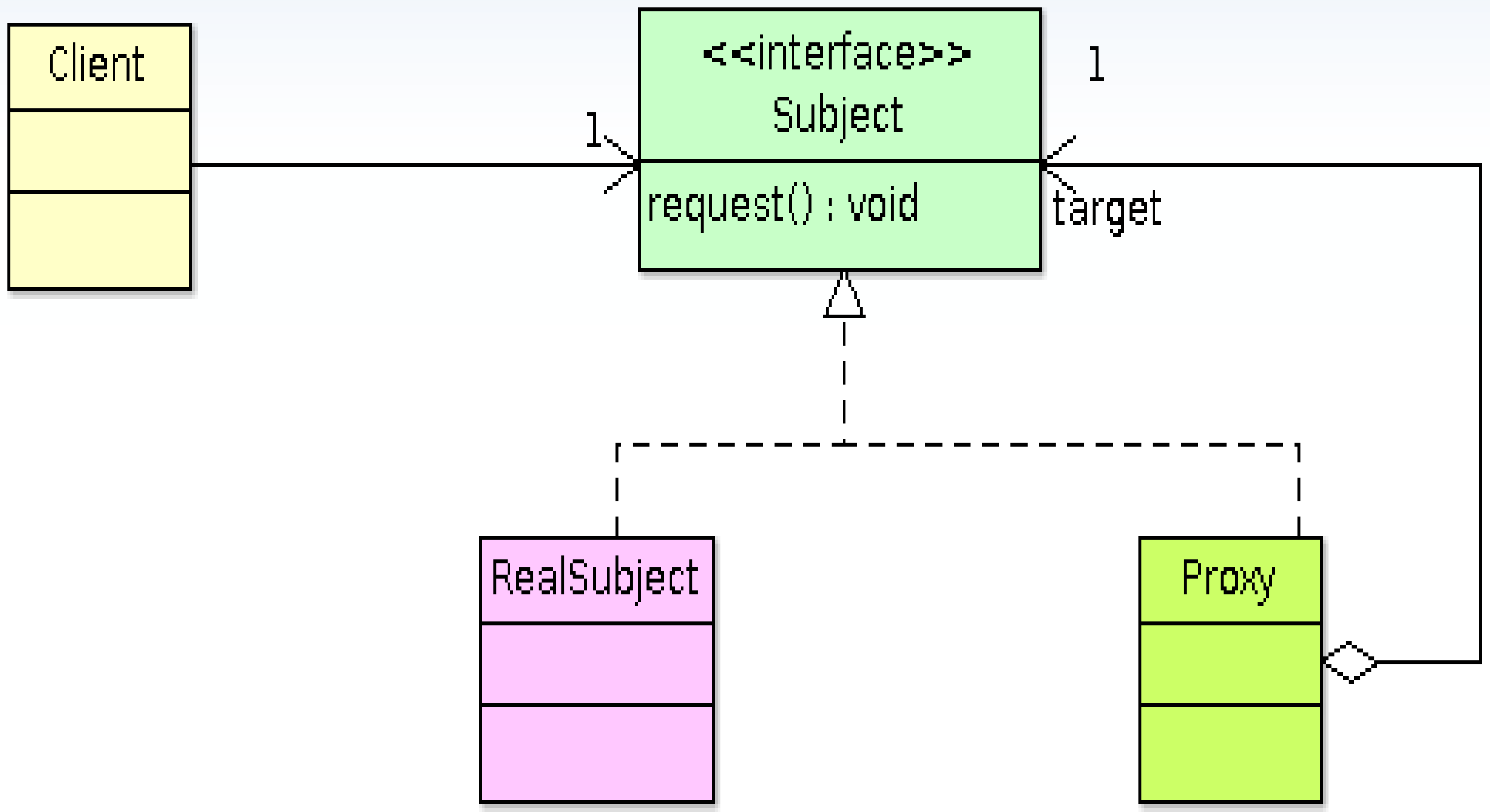
Client, on the other hand, is not aware of its interation with the proxy instance



Method calls from client first arrive at proxy instance before reaching their target destination

Proxy, before and after those method calls can perform tasks related with the call

Proxy Class Diagram

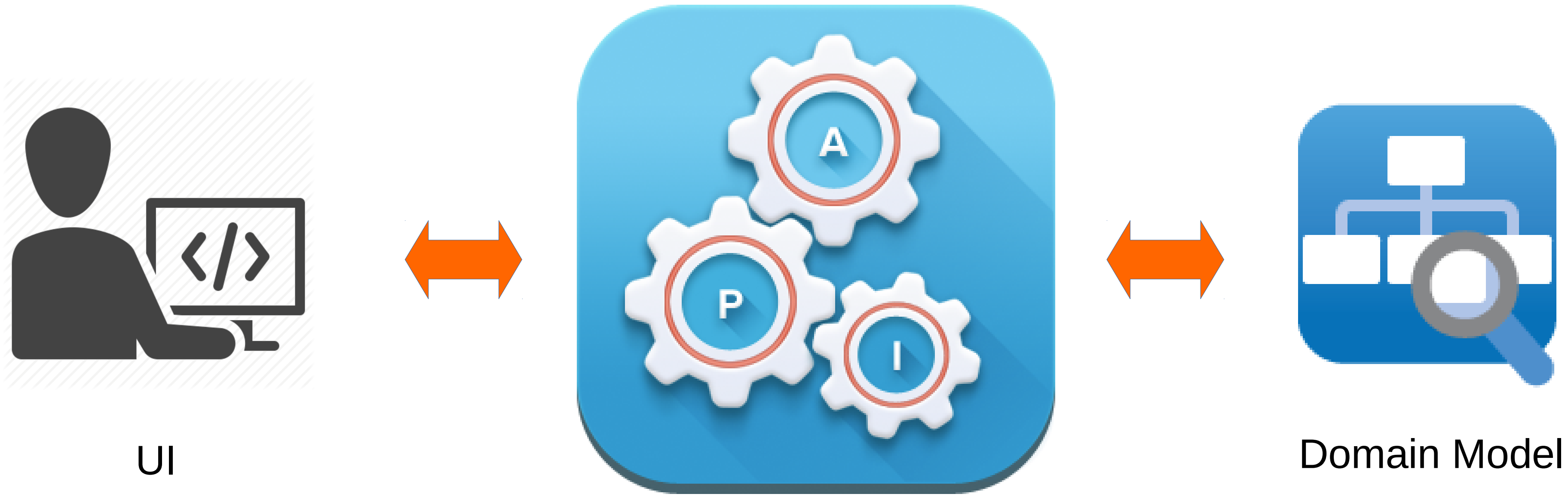


Proxy Generation Strategies

- **Interface Proxy**
  - Interfaces implemented by the actual model class are used to generate proxy class
  - Known as JDK proxy
- **Class Proxy**
  - Domain class is extended to generate proxy class
  - Known as CGLIB or Javassist proxy

View Model API

- An separate API, in the role of a **bridge between UI and persistent domain objects** is necessary to operate
- **Proxy classes** generated from those domain classes should also **implement this API** as well



View Model API

- **getModel**
  - Allows acces to the wrapped domain model instance
- **flush**
  - Reflects changes accumulated in the view model instance into the wrapped target domain model instance
- **refresh**
  - Reverts state of the view model into domain model instance's initial state
- **savepoint(id)/rollback(id)**
  - Allows to save current state of view model by associating it with a given identifier, so that view model state can be rolled back to the state identified by that id at a later time

View Model API

- **isDirty**
  - Detects if view model state has been changed or not
- **isSelected/setSelected**
  - Helps to identify if view model instance is selected through the bounded UI component, and to mark it as selected
- **isTransient**
  - Helps to check if domain model instance wrapped by the view model is persisted into DB before or not
- **replace(Object model)**
  - Replaces given domain model instance with the already wrapped target domain model instance in the view model

View Model API

- **addedElements(propertyName)**
  - Returns elements which are added into the collection property identified by the given propertyName
- **removedElements(propertyName)**
  - Returns elements which are removed from the collection property identified by the given propertyName
- **dirtyElements(propertyName)**
  - Returns elements whose state has been changed in the collection property identified by the given propertyName

harezmi it solutions

Creates Value...

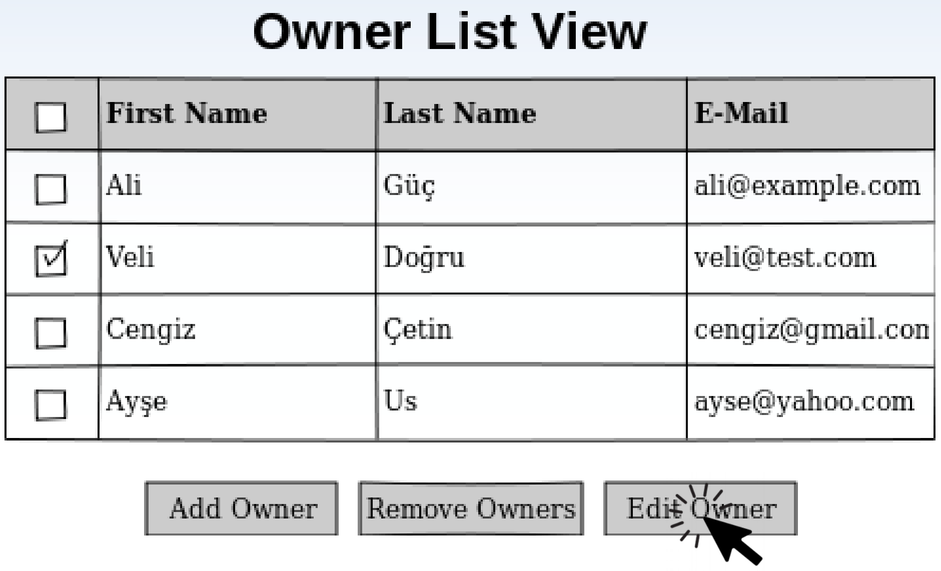
View Model API in Action

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

List<Owner> owners = em.createQuery(
            "from Owner").getResultList();

List<Owner> viewModels = new
            ArrayList<Owner>(owners.size());
for(Owner model:owners) {
    Owner viewModel = viewModelCreator
            .create(Owner.class, model);
    viewModels.add(viewModel);
}
```

list owners



Owner List View

☐	First Name	Last Name	E-Mail
☐	Ali	Güç	ali@example.com
☑	Veli	Doğru	veli@test.com
☐	Cengiz	Çetin	cengiz@gmail.con
☐	Ayşe	Us	ayse@yahoo.com

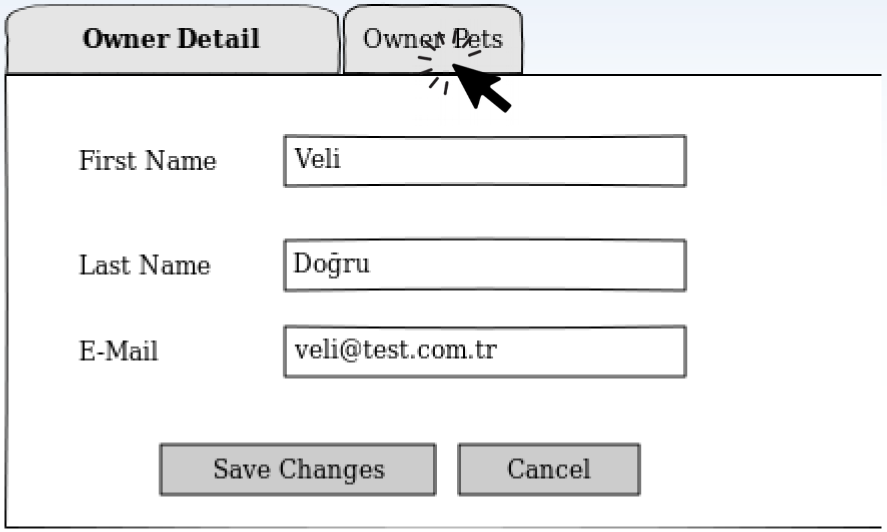
```
Owner selectedOwner = null;
for(Owner viewModel:viewModels) {
    if(((ViewModel<Owner>)viewModel)._isSelected_()) {
        selectedOwner = viewModel;
        break;
    }
}
```

select an owner and check if it is selected

Creates Value...

View Model API in Action

Owner Detail Tab View



First Name	Veli
Last Name	Doğru
E-Mail	veli@test.com.tr

edit owner

```
selectedOwner.setEmail("veli@test.com.tr");

...

((ViewModel<Owner>) selectedOwner)
        ._savepoint_("pets_tab_view");
```

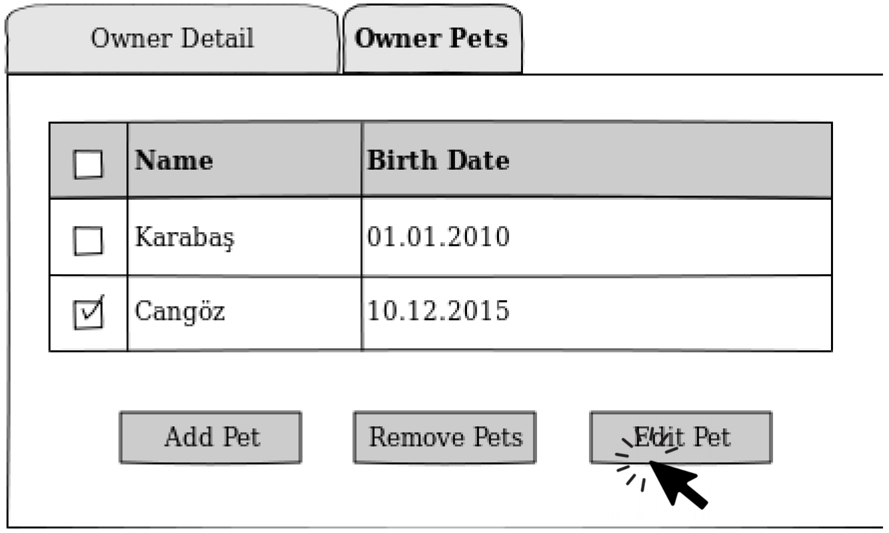
create a savepoint before switching into pets tab view

Owner Pets Tab View

```
Pet selectedPet = null;

for(Pet pek:selectedOwner.getPets()) {
    if(((ViewModel<Pet>)pek)._isSelected_()) {
        selectedPet = pek;
        break;
    }
}
```

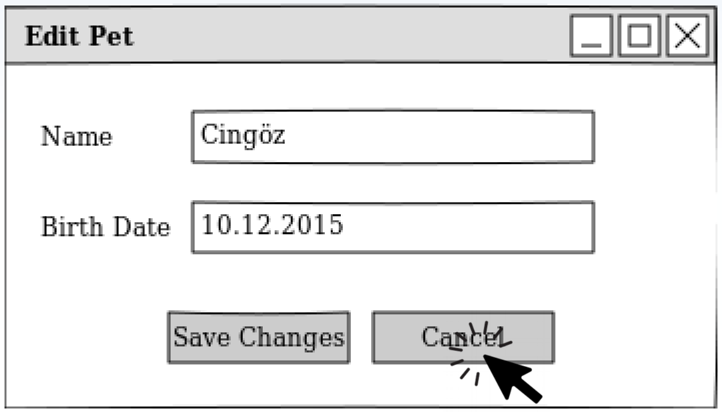
select a pet and check if it is selected



☐	Name	Birth Date
☐	Karabaş	01.01.2010
☑	Cangöz	10.12.2015

View Model API in Action

Edit Pet Dialog


Edit Pet

Name	Cingöz
Birth Date	10.12.2015

edit pet

```
selectedPet.setName("Cingöz");

...

((ViewModel<Owner>)selectedOwner)
              ._rollback_("pets_tab_view");
```

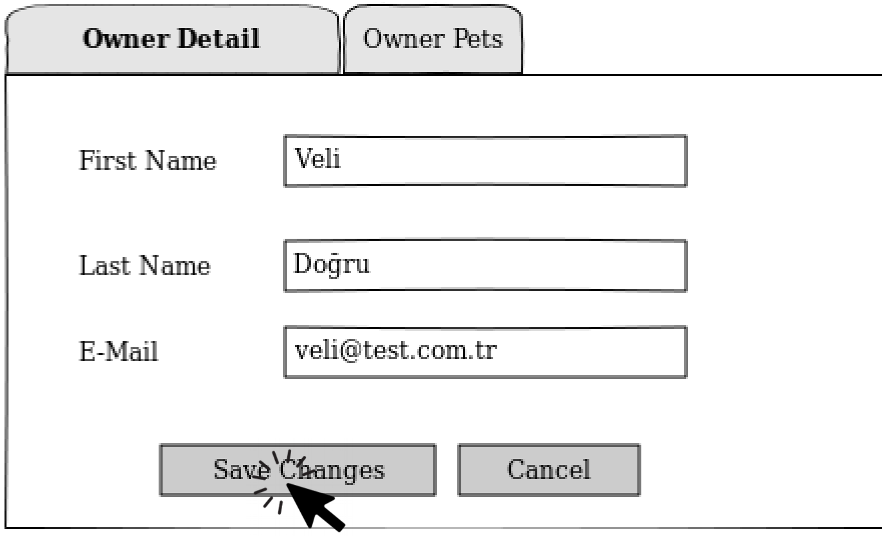
rollback changes made in pets tab view

Owner Detail Tab View

```
((ViewModel<Owner>) selectedOwner)._flush_();
```

flush changes accumulated in view model into the target owner

```
em.getTransaction().commit();
em.close();
```



First Name	Veli
Last Name	Doğru
E-Mail	veli@test.com.tr

harezmi it solutions

Creates Value...

View Model API in Action:

Adding UI Specific Fields & Methods

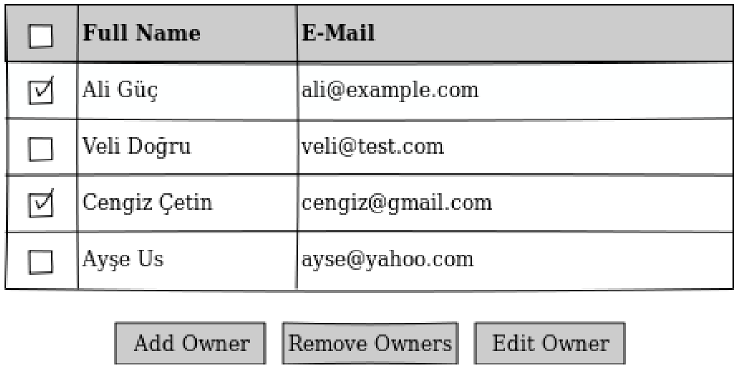
declare a new interface to handle UI specific interactions

```
public interface OwnerViewModel {
    public String getFullName();
}

public class OwnerViewModelImpl
            extends ViewModelImpl<Owner>
                    implements OwnerViewModel {
    public OwnerViewModelImpl(Owner model,
            ViewModelDefinition definition) {
        super(model, definition);
    }

    @Override
    public String getFullName() {
        String firstName = _getModel_().getFirstName();
        String lastName = _getModel_().getLastName();
        String fullName = "";
        if (StringUtils.isNotEmpty(firstName)) {
            fullName += firstName;
        }
        if (StringUtils.isNotEmpty(lastName)) {
            if (StringUtils.isNotEmpty(fullName)) {
                fullName += " ";
            }
            fullName += lastName;
        }
        return fullName;
    }
}
```

Owner List View



☐	Full Name	E-Mail
☑	Ali Güç	ali@example.com
☐	Veli Doğru	veli@test.com
☑	Cengiz Çetin	cengiz@gmail.com
☐	Ayşe Us	ayse@yahoo.com

View Model API in Action:

View Model Definitions

```
public class PetClinicViewModelDefinitionProvider
                implements ViewModelDefinitionProvider {


    @Override
    public Collection<ViewModelDefinition> getViewModelDefinitions() {

        ViewModelDefinition ownerDef =
            new ViewModelDefinition(Owner.class,OwnerViewModelImpl.class);

        ViewModelDefinition petDef = new ViewModelDefinition(Pet.class);
        ownerDef.addDefinition("pets", petDef);

        return Arrays.asList(ownerDef, petDef);
    }
}
```

Conclusion

- Reusing persistent domain objects within the UI layer causes several **persistence related problems** in enterprise applications
- An **intermediate layer placed in between UI and domain models** is required
- **An API to execute operations** through this intermediate layer becomes a bridge between UI and domain models
- Such a layer, which is called as **“View Model”** can be created by employing **dynamic proxy class generation** method without violating DRY principle!

Questions & Answers

Contact

- Harezmi IT Solutions
- http://www.harezmi.com.tr
- info@harezmi.com.tr