

M3.KISKIS

Ahmed Amohamed – 9736859

Harpreet Singh Sokhal – 6417965

Jia He - 6224601

Mary Garzon – 9719148

Nagabandi Kalyan Kumar – 6417221

Class Diagram of Actual System (5 points)

Our ideal Architecture VS KisKis's Actual Architecture

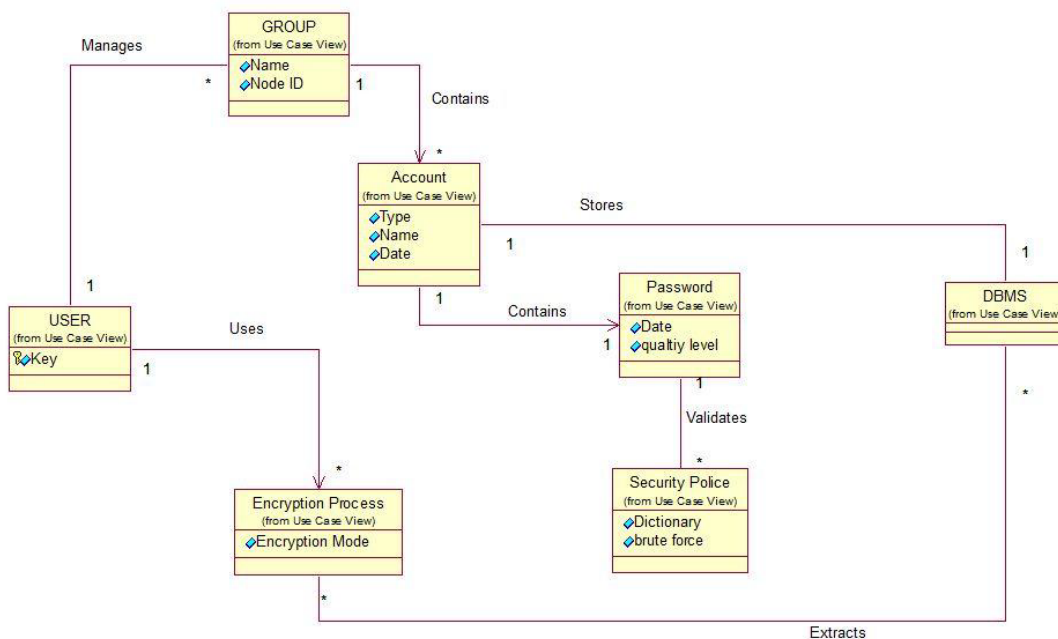


Figure 1 . Our Ideal Diagram from M2

In the actual class, all the operations are controlled by users via the GUI, so all the GUI classes refer to the conceptual class **USER**. By applying GUI, the user interface is friendlier and the structure of stored passwords is clearer.

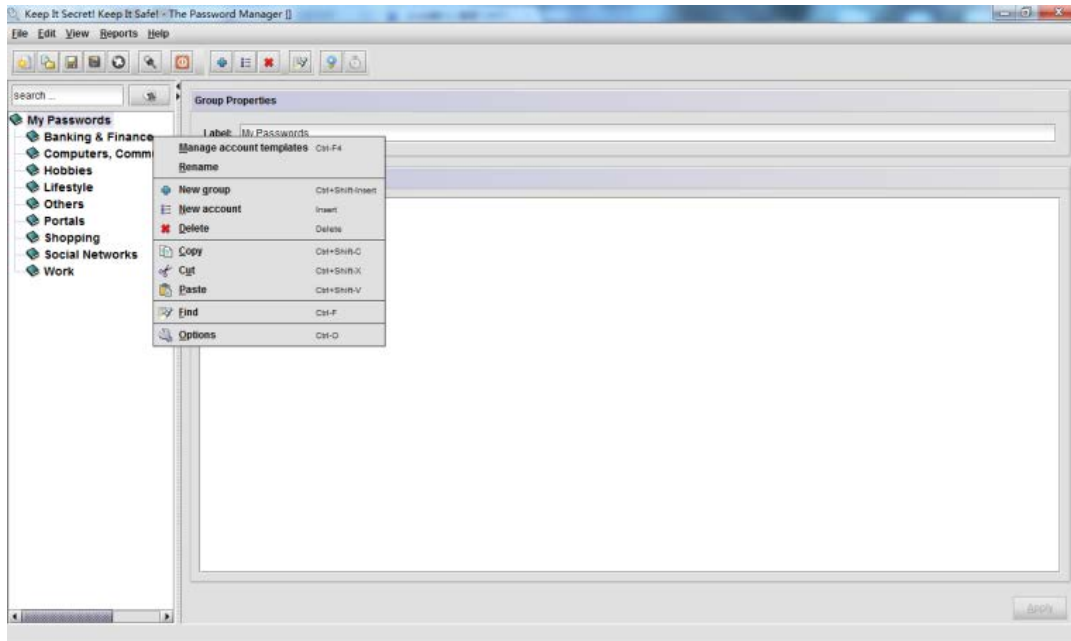


Figure 2. User Interface

In our ideal architecture, the instances of **ACCOUNT** class are distinguished by the value of attribute “Type” and the **GROUP** which the account belongs to represent the categorization of that account. But in the actual architecture, the **GROUP** class implements an interface **ModelNode**, since all the groups and accounts are shown as tree nodes in the user interface. For the conceptual class **ACCOUNT**, the actual architecture applies a Factory pattern. There is an abstract class **SecuredElement**, which includes most of the general properties of an account, such as name, create date, password, and so on. Then, 5 final classes inherit this abstract class and define the specific properties of each different account type. This gives the system a highly flexible and lower difficulty in evolvement.

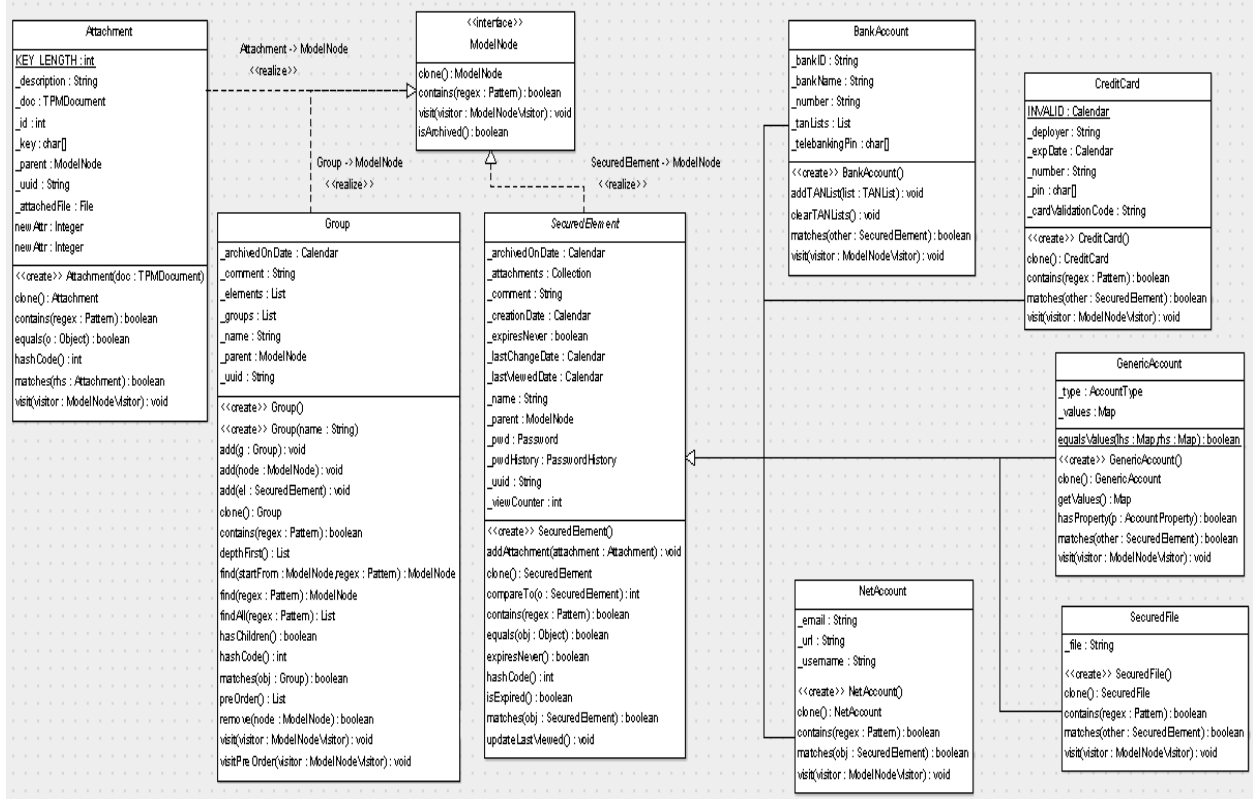


Figure 3. KISKIS's architecture on GROUP and ACCUONT

For example, if we want to create a password for Bank Account. Besides the general properties, we also need the information about bank name, account number, and so on.

Abstract Class *SecuredElement*

```

public abstract class SecuredElement implements Comparable<SecuredElement>,
ModelConstants, Cloneable, ModelNode, Archivable {
    private static final long serialVersionUID = 1L;
    private Calendar _archivedOnDate;
    private Collection<Attachment> _attachments;
    private String _comment;
    private Calendar _creationDate;
    private boolean _expiresNever;
    private Calendar _lastChangeDate;
    private Calendar _lastViewedDate;
    private String _name;
    private transient ModelNode _parent;
    private Password _pwd;
    private PasswordHistory _pwdHistory;
    private String _uuid;
    private int _viewCounter;
    protected SecuredElement() {
        _uuid = IdGenerator.generate();
        _name = "unnamed";
        _pwd = new Password();
        _pwdHistory = new PasswordHistory();
    }
}

```

```

        _expiresNever = true;
        _comment = "";
        _creationDate = DateUtils.getCurrentDateTime();
        _lastChangeDate = DateUtils.getCurrentDateTime();
        _lastViewedDate = DateUtils.getCurrentDateTime();
        _viewCounter = 0;
        _attachments = new Vector<Attachment>();
    }

    .....
}

```

Final Class BankAccount : (inherit the general properties from SecuredElement and define the specific properties for BankAccount type.)

```

public final class BankAccount extends SecuredElement {
    private static final long serialVersionUID = 1L;
    private String _bankID;
    private String _bankName;
    private String _number;
    private final List<TANList> _tanLists;
    private char[] _telebankingPin;
    public BankAccount() {
        super();
        _number = "";
        _bankName = "";
        _bankID = "";
        _tanLists = new ArrayList<TANList>();
        _telebankingPin = new char[0];
    }

    .....
}

```

To sustain the relationship between **GROUP** and **ACCOUNT**, an attribute of type **List<SecuredElement>** is defined in order to keep the information of accounts belonged to this group. There also is an attribute of type **List<Group>** is defined in order to keep the information of the sub-group belonged to this group. In the class **GROUP**, polymorphism is applied to define the method add in order to insert sub-group and account into the defined lists and arrange the tree structure of groups and accounts.

Final class GROUP:

```

public final class Group implements ModelNode {
    private final List<SecuredElement> _elements;
    private final List<Group> _groups;

    public Group() {
        this("unnamed");
    }

    public Group(final String name) {
        _uuid = IdGenerator.generate();
        _name = name;
        _comment = "";
        _groups = new ArrayList<Group>();
        _elements = new ArrayList<SecuredElement>();
    }
}

```

```

    }

    private void add(final Group g) {
        assert g != null;

        _groups.add(g);
        Collections.sort(_groups);
    }

    public void add(final ModelNode node) {
        assert node != null;
        if (node.getParent() != null) {
            ((Group) node.getParent()).remove(node);
        }

        if (node instanceof Group) {
            this.add((Group) node);
        } else {
            add((SecuredElement) node);
        }
        node.setParent(this);
    }

    private void add(final SecuredElement el) {
        _elements.add(el);
        Collections.sort(_elements);
    }
}

.....
}

```

Besides the structure of groups and accounts, the password validation is another part of interest. In our ideal diagram, a class **SecurityPolice** is defined for doing this. However, a strategy pattern is applied to create three different password validators, **EmptyPasswordValidator**, **WeakPasswordValidator**, and **DictionaryPasswordValidator**. All of them implement the interface **IPasswordValidator**, which has only one operation **Boolean validatePassword(char[] pwd)**.

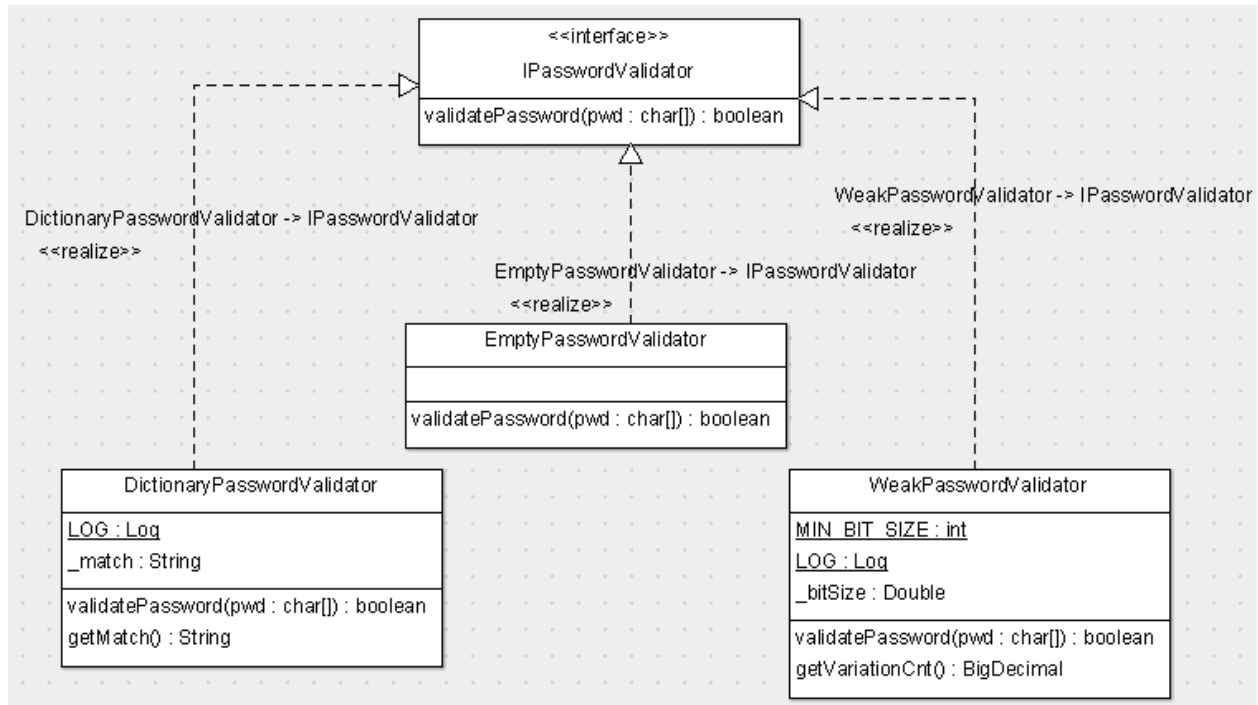


Figure 4. Actual architecture of Password Validator

The use of strategy pattern makes the source code reusable and maintainable, because they also can be used in the other parts of the system or even another system without code duplication. Moreover, the change of the source code in the function **validatePassword** would not affect the client side source code.

Finally, when an account is created, a new instance of class **Password** will be concreted. This refers to the relationship “**Contains**” between conceptual classes **ACCOUNT** and **Password** in our ideal architecture diagram. Since all types of account should do this in their constructor, this part of code is included in the abstract class **SecuredElement**.

Final class SecureElement

```

public abstract class SecuredElement implements Comparable<SecuredElement>,
ModelConstants, Cloneable, ModelNode, Archivable {
protected SecuredElement() {
.....
    _pwd = new Password();
.....
}
.....
}

```

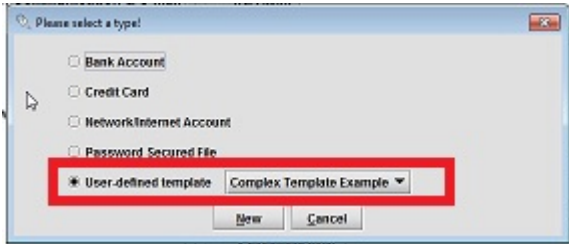
However, unlike our ideal diagram, there is no relationship between class **Password** and the classes **PasswordValidator**. The password is validated just after its input, so the validator classes are invoked by the GUI code and then will show the result to user immediately.

Description of ArgoUML

ArgoUML is an UML diagramming application written in Java and released under the open source Eclipse license.^[1] It supports forward engineering and reverse engineering. For analyzing KISKIS, ArgoUML helps us to understand the actual architecture faster, since the UML class diagram will be generated automatically after the java files are imported. To analyze a specific module, only the related classes are needed.

Code Smell and Possible Refactoring (5 marks)

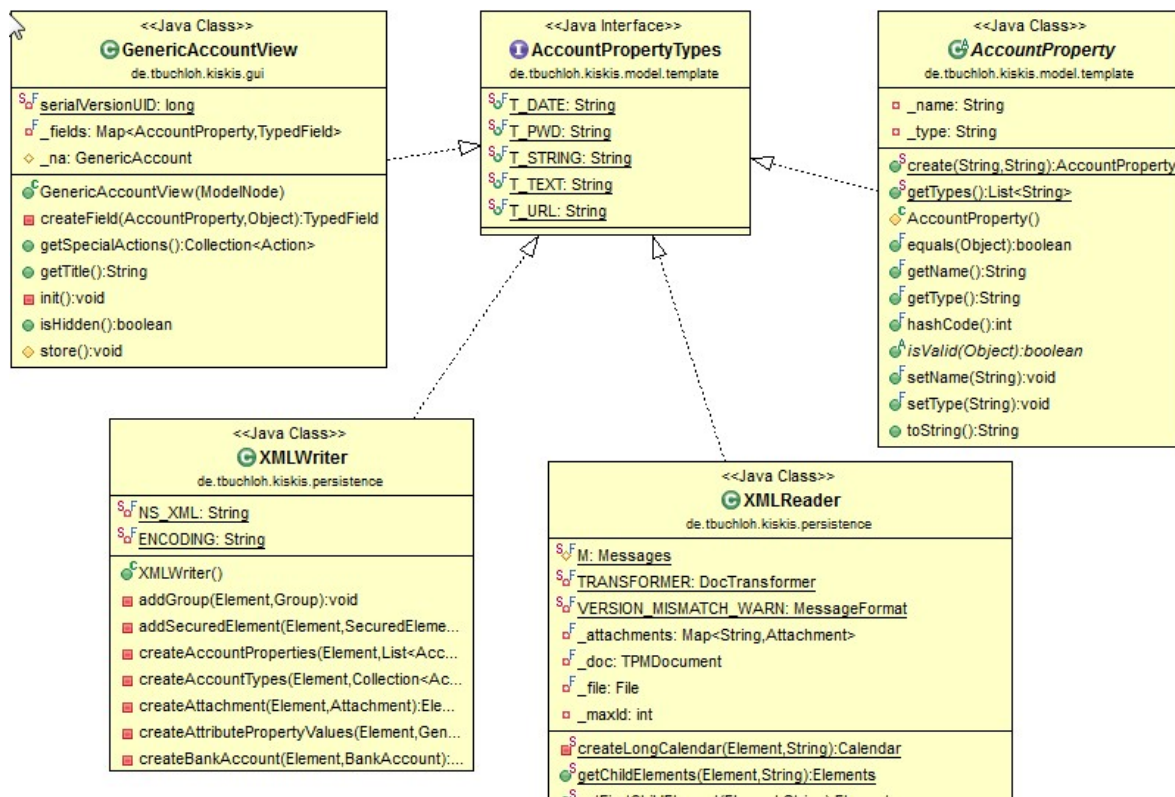
The source code of <kiskis> has evolved during last eight years; this explains the high quality code of each one of the modules (GUI, Model, persistence). Furthermore, finding code smells in this project was a difficult task and a big challenge. The following table summarizes the most important code smells detected; the refactoring we suggest and the different steps to succeed and avoid the code smell.

DESCRIPTION OF SUBSYSTEM		
<p>Template functionality in kiskis. This subsystem contains classes located in the Model module. The subsystem is used in case a new account of type 'User Defined Template – Complex Template Example' is selected.</p> 		
CODE SMELL	REFACTORING	PROPOSED SOLUTION
<p>ShotGunSurgery was detected in the interface <i>AccountPropertyTypes</i>, because this interface provides only static variables of type String, thus a change on these variables requires modifications on every one of the four classes that implements the mentioned interface. Also we found inappropriate use of interface in this class because it does not have any methods that require implementation on the subclasses.</p> <p>Duplication code was found in the method <i>create()</i> of the class <i>AccountProperty</i>, that uses the constants from <i>AccountPropertyTypes</i>.</p> <p>MiddleMan was detected because the interface <i>AccountPropertyTypes</i> delegates all the responsibilities regarding the</p>	<p>Move field, Move method, Consolidate conditional expression, Remove Middle Man</p>	<p>In Java it is possible to define a class of type Enumeration is a type like class and interface and can be used to define a set of Enum constants. In this order of ideas we think it is possible to replace the actual interface <i>AccountPropertyTypes</i> by an Enum class and define the methods that will be used by the classes that make use of this new type. Furthermore, it is not possible to create a superclass because the current classes that implements <i>AccountPropertyTypes</i> already extends a superclass.</p> <ol style="list-style-type: none">1. Create a new class of type Enum called <i>AccountPropertyTypes</i> to avoid unnecessary coupling between the interface and six classes. Indeed, the constants will be declared only in one place and the responsibility of manipulate them are only in the Enum class, thus we prevent duplication code.

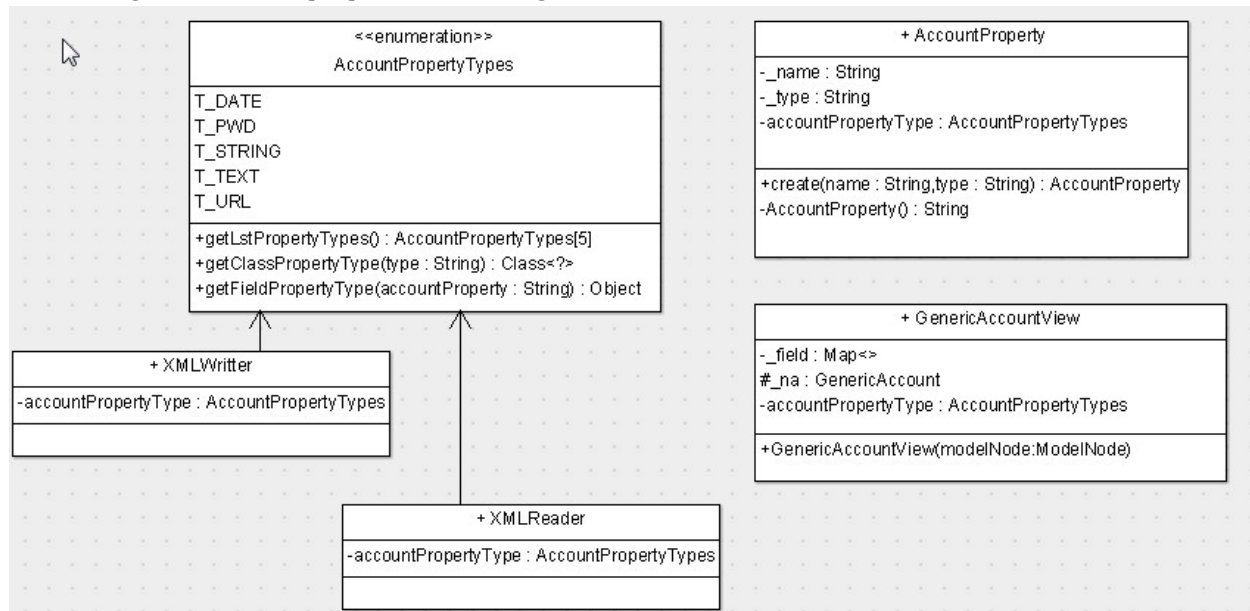
[1]. <http://en.wikipedia.org/wiki/ArgoUML>

<p>constants to the implementing classes: <i>AccountProperty</i> and <i>GenericAccountView</i></p>		<ol style="list-style-type: none"> 2. Using Move field to move the features from the initial interface to the Enum class. So, the five static final variables will be moved to the Enum class. 3. Using Move Method in order to copy the method <i>getTypes()</i> from <i>AccountProperty</i> to the Enum class <i>AccountPropertyTypes</i>, thus improving cohesion. We decided to rename the method to <i>getLstPropertyTypes()</i> for better comprehension. 4. Delete the interface. 5. In the declaration of classes: <i>AccountProperty</i>, <i>GenericAccountView</i>, <i>StandardDocumentationFactory</i>, <i>XMLReader</i>, <i>XMLWriter</i> will not be necessary to do any implementation of <i>AccountPropertyTypes</i> interface. Finally, we ensure low coupling between those classes and the Enum responsibilities. 6. A Consolidate Conditional expression is performed in method <i>create()</i> from class <i>AccountProperty</i>, because there are many conditionals with the same results. In this case we avoid code duplication. 7. Using RemoveMiddleMan to define the constants control and operations in the Enum class. In the Enum class a new method <i>getClassPropertyType()</i> is created to replace the logic used in <i>create()</i> method on the <i>AccountProperty</i> class. Thus the code from <i>create()</i> is moved to <i>getClassPropertyType()</i> using Move Method to the Enum class and a reference to this method is added in <i>create()</i>. With this step we are enforcing the cohesion. 8. As same as in step 7 with RemoveMiddleMan refactoring in the Enum class adding a new method <i>getFieldPropertyType()</i> to replace the logic in method <i>createField()</i> from <i>GenericAccountView</i> class and the code from <i>createField()</i> is moved to <i>getFieldPropertyType()</i> to provide the responsibility to determinate the type of field for each Enum value. Thus high cohesion and the single responsibility design are implemented. (TODO:is possible to implement a factory of FieldType!!) 9. Replace the old references in the source classes to the Enum class. Now, if any small modification is made in the Enum class it is not necessary to perform changes in the other four classes that uses the constants and the methods provides by the Enum class. Thus, a more cohesive set of classes.
--	--	---

The next figure depicts the current design of the Template package, and the related classes from the view and persistence module.



The next figure shows the proposed refactoring:



This is the actual source code in the interface 'AccountPropertyTypes':

```
public interface AccountPropertyTypes {

    public static final String T_DATE = "Date";

    public static final String T_PWD = "Password";

    public static final String T_STRING = "String";

    public static final String T_TEXT = "Text";

    public static final String T_URL = "URL";

}
```

This is the actual source code in the class 'AccountProperty':

```
public static AccountProperty create(final String name, final String type)
    throws ElementNotFoundException {
    AccountProperty p = null;
    if (T_DATE.equals(type)) {
        p = new SimpleProperty(Calendar.class);
    } else if (T_PWD.equals(type)) {
        p = new SimpleProperty(String.class);
    } else if (T_STRING.equals(type)) {
        p = new SimpleProperty(String.class);
    } else if (T_TEXT.equals(type)) {
        p = new SimpleProperty(String.class);
    } else if (T_URL.equals(type)) {
        p = new SimpleProperty(String.class);
    } else {
        // TODO: localize me
        throw new ElementNotFoundException("No such property type: " + type);
    }
    p.setName(name);
    p.setType(type);
    return p;
}
```

And this is for the class 'GenericAccountView':

```
private TypedField createField(final AccountProperty p, final Object value) {
    TypedField comp = null;
    if (T_PWD.equals(p.getType())) {
        comp = new PasswordField();
    } else if (T_STRING.equals(p.getType())) {
        comp = new StringField();
    } else if (T_DATE.equals(p.getType())) {
        comp = new DateField();
    } else if (T_TEXT.equals(p.getType())) {
        comp = new TextField();
    } else if (T_URL.equals(p.getType())) {
        comp = new UrlField();
    } else {
        throw new Error("unknown property type: " + p.getType());
    }
    comp.setValue(value);
    comp.addContentListener(this);
    return comp;
}
```