Faculty of Informatics and Information Technologies
Slovak University of Technology

Design Pattern Support Based on The Source Code Annotations and Feature Models

Peter Kajsa and Pavol Návrat
{kajsa,navrat}@fiit.stuba.sk

Program of Presentation

- Introduction
- Open Problems
- Our Ideas
- Method of Design Pattern Support in The Source Code
  - Proposal of Annotation for Design Patterns
  - Support of Design Pattern Instantiation and Evolution
  - Realization and Implementation of the Method
- Elimination of Manual Annotation of The Source Code
- Conclusion and Future Work

Introduction

- Design Patterns
  - represent abstracted, generalized and verified solutions of non-trivial problems of software design that occur repeatedly.
  - the idea of applying verified pattern solutions to common recurring problems has very quickly attracted considerable attention also in the software engineering
  - patterns provide especially effective ways to improve the quality of software systems
- Nowadays there exist many approaches to the support of design patterns.
- Many of them are based on pattern modeling languages, pattern ontologies (e.g. [4]), UML profiles (e.g. [5]), pattern templates (e.g. [6]) or model transformations (e.g. [7]), etc.

4. Kampffmeyer, H., et al.: Finding the pattern you need: The design pattern intent ontology. LNCS, vol. 4735, pp. 211–225. Springer, Heidelberg. (2007)
5. Debnath, N.C., et al.: Defining Patterns Using UML Profiles. In: IEEE International Conference on Computer Systems and Applications, pp. 1147-1150, Washington. (2006)
6. Marko, V.: Template Based, Designer Driven Design Pattern Instantiation Support. LNCS, vol. 3255, SOFSEM 2004, Springer-Verlag, pp. 144-158, (2004)
7. Dong, J., Yang, S., Zhang, K.: A model transformation approach for design pattern evolutions. ECBS '06, pp. 80-92, Washington, DC, USA, IEEE Computer Society. (2006)

Open Problems

- Most of the approaches are focused on the support of design patterns at the design level (i.e. model) and as a result, developers have available wide tool-based support of many pattern aspects at the design level.
- However, the support becomes less robust by the transition to the source code and pattern instances become almost invisible in the huge amount of the source code lines.
- The evolution of the existing instances of patterns in the source code is very difficult without any tool-based support, because a developer does not have a good view of all the participants of pattern instances in the source code.
- Moreover, due to the inability to identify the individual participants of pattern instances in the source code, they may be modified in an incorrect way during the system evolution and maintenance, and this may result in the breakdown of the pattern and the loss of the benefits gained by its application in the software system.

Open Problems (2) and our Ideas

- It is necessary to mark explicitly and make visible the higher-level (i.e. design) intentions also in the source code.
- The missing semantics can be added into the source code via annotations expressing the semantics and design intentions of the annotated code fragments
- Source code annotations as metadata information about the intention of the source code fragments have been introduced in several approaches.
- All of the approaches, however, require a manual annotation of source code fragments by a developer.
- The manual annotation of the source code by developers is very lengthy and tiresome. Therefore our method does not require the manual annotation of the source code by a human, instead the method works on the idea of the architectural and design information propagation and expansion from higher levels of abstraction (i.e. models) into the source code.
- Our idea is to support also the instantiation and evolution of the pattern instances in the form of the context assistant in the source code working via annotations and feature models.

Our Ideas (2)

- The semantics of patterns introduced into the source code by annotations emphasizes the visibility of pattern instances
- Annotations makes identification of pattern participants in the source code quite easy.
- In consequence, the support of the pattern detection, instantiation and evolution in the source code can be achieved in a very suitable form of a source code context assistant.
- Thanks to annotations, the support mechanism will be able to identify the pattern participants already implemented, and subsequently it will be able to offer an option to generate any missing pattern participant or to perform possible pattern evolution in the given context, etc.

Method of Design Pattern Support in The Source Code

- Proposal of annotation for design patterns

```
public @interface DesignPattern{
    PatterNames patterName();
    String instanceAlias();
    RoleNames roleName();
    String variant() default "DEFAULT";

    enum PatterNames{ Observer; //...
    }
    enum RoleNames{Subject, attach, detach, notifyObservers,
        update, observers, Observer, ConcreteObserver; //...
    }
```

- *patternName* expresses the name of the pattern e.g. Observer, Mediator, Command, etc.
- Because one pattern (for example Observer) may have more different instances applied, the pattern *instanceAlias* is necessary for the recognition among these instances.
- *roleName* expresses the name of the pattern participant e.g. Subject, ConcreteSubject, attach, etc.
- Some participants of the pattern instances may have more possible variants and therefore the *variant* attribute is also necessary

Method of Design Pattern Support in The Source Code

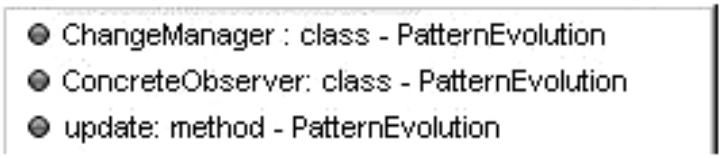
- An example of annotated source code

```
@DesignPattern(patterName = PatterNames.Observer, instanceAlias = "obs1",
        roleName = RoleNames.Subject)
public abstract class AbstractUserModel {
    // ...
    @DesignPattern(patterName = PatterNames.Observer, instanceAlias = "obs1",
            roleName = RoleNames.observers)
    ArrayList<View> views;
    // ...
    @DesignPattern(patterName = PatterNames.Observer, instanceAlias = "obs1",
            roleName = RoleNames.attach)
    public void addView(View v){
        views.add(v);
    }
    //...
    @DesignPattern(patterName = PatterNames.Observer, instanceAlias = "obs1",
            roleName = RoleNames.notifyObservers, variant="modelOfNotification = callback")
    public void updateViews(){
        for(View v : views) v.refresh(this);
    }
    //...
}
```

Method of Design Pattern Support in The Source Code

- The support of the pattern instantiation and evolution is realized in form of the source code context assistant with the consequent source code generation driven by typed annotation and its location.
- In a nutshell, a developer write an annotation in a specific location and the tool generate source code of an element in accord to the typed annotation
- For example:

```
3  @DesignPattern( patterName = PatterNames.Observer,
4          instanceAlias = "Observer1",
5          roleName = RoleNames.
6
7
```



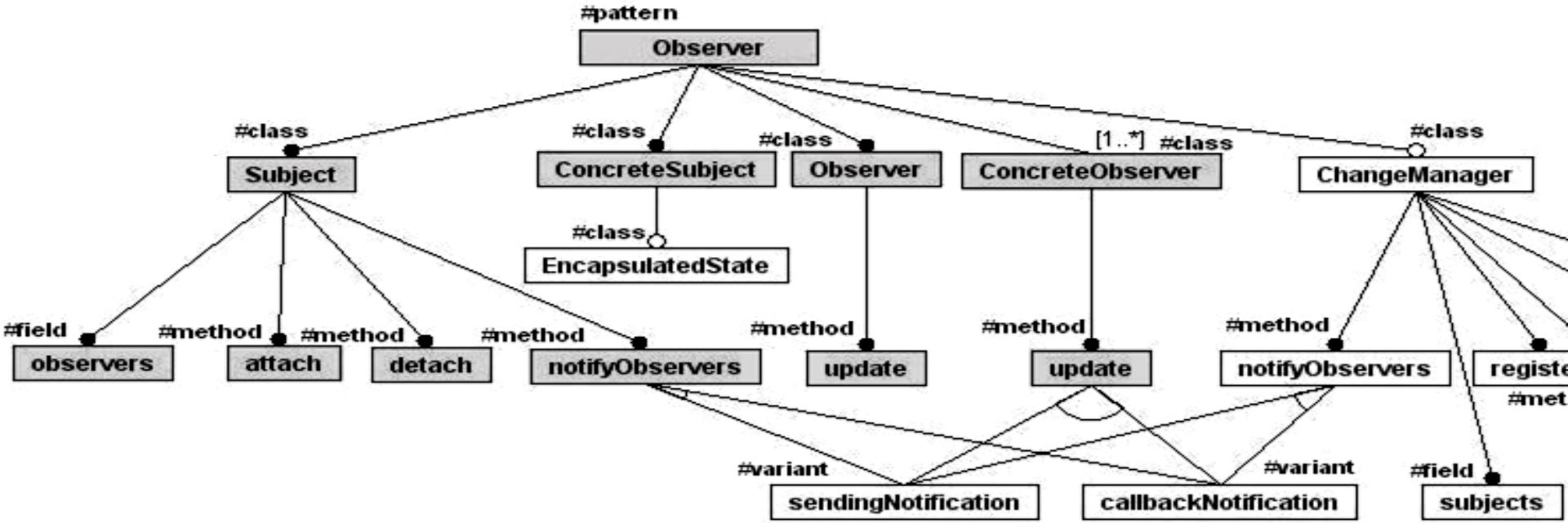
Method of Design Pattern Support in The Source Code

- The method is described in the following steps.
  1. In the first step, the developer begins with the writing of a pattern annotation in the desired location in the code. When the developer writes @DesignPattern(patternName... , the context assistant offers the set of names of supported patterns. The developer, for example, chooses PatternNames.Observer.
  2. Next the developer continues with the writing of the annotation and writes instanceAlias. So the annotation looks as follows: @DesignPattern(patternName = PatternNames.Observer, instanceAlias = ... Now the context assistant searches all the existing instances of the pattern with the given name i.e. PatternNames.Observer and it offers the developer the set of aliases of all the existing instances of Observer pattern in the project. Because of the suitable annotation structure this search is very straightforward.
  3. The developer chooses an instanceAlias from the offered set or writes a new, unique alias.
  4. When the developer writes a new, unique instance alias, the support mechanism deduces that the developer desires a creation of a new pattern instance. Otherwise, when the developer chooses one of the offered existing instance aliases, the support mechanism deduces that the developer desires evolution of the pattern instance identified by the chosen alias and the pattern name. According to the developer's choice pattern instantiation or evolution follows.

Support of Instantiation

5. When the developer writes a new, unique instance alias, the instantiation of specified pattern is performed. So the support mechanism loads feature model of the specified pattern and it selects all mandatory features at the first level (i.e. classes) and generates them into the source code.
6. If one of the mandatory features has more possible variants, the developer is asked for selection of its variant via dialogue during the instance generation.

An example of feature model of Observer pattern (mandatory features are filled with gray color):



Support of Instantiation (2)

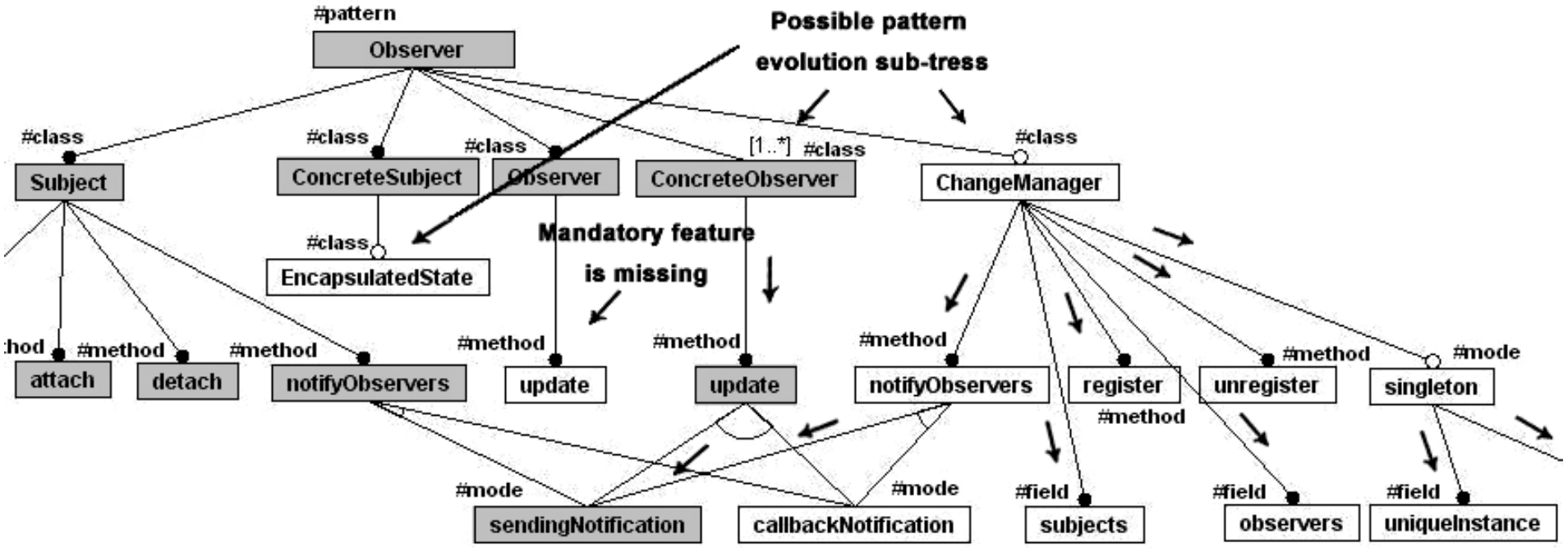
- The first mandatory class is generated at the position of the entered annotation in the current file, therefore in case of the pattern instantiation the developer should write the annotation in a new empty file.
- Other mandatory classes are generated into new automatically created empty files in the current package of the project.
- Of course, an element is always generated with all its mandatory sub-elements.

Support of Evolution

5. When the developer selects alias from offered set of the existing instance aliases, the evolution of specified pattern instance is performed. So the support mechanism creates a feature model configuration of the specified pattern instance. Thanks to the annotations, the recognition of the pattern instance participants present in the source code is quite easy.
6. The support mechanism loads the feature model of the pattern.
7. The created feature model configuration of the pattern instance is compared with the loaded feature model of the pattern. In consequence, the options of possible evolution of the pattern instance are detected.
8. The support mechanism offers the detected set of possible instance evolution options in form of the code assistant.

Support of Evolution (2)

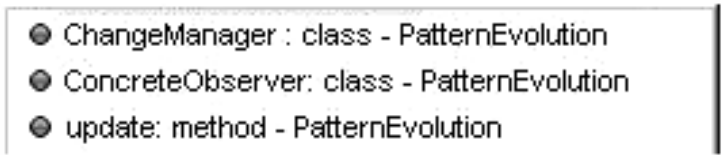
- An example of comparison of the feature model configuration of an existing Observer instance with the feature model of Observer pattern (existing participants - features are filled with gray color). The possible options of pattern instance evolution are illustrated by the arrows.



Support of Evolution (3)

- Example of detected set of possible instance evolution options offered in form of the code assistant.

```
3  @DesignPattern( patterName = PatterNames.Observer,
4          instanceAlias = "Observer1",
5          roleName = RoleNames.
6
7
```



- It is important to remark that only the roots of possible instance evolution sub-trees are offered to the developer, because generation of child elements (e.g. methods) has no sense as long as the parent element (e.g. class) does not exist in the source code.
- The selected element with all its mandatory sub-elements is generated at the position of the entered annotation in the current file. So the method supposes at least basic knowledge of patterns.
- Within the scope of the pattern evolution also the detection of missing mandatory features is supported. This way the basic check of the pattern instance validity is achieved.

Realization

- Each element – feature of the pattern feature model has its own code template attached.
- Each code template of an element includes subsequent templates of all related mandatory sub-elements of the element in accord with the feature model of the pattern.
- Therefore an element is always generated with all its mandatory sub-elements.
- For an example the Subject template includes observers, attach, detach and notifyObservers templates.

```
<%@ jet class="Observer-SubjectTemplate" package="dp.anot.jet" startTag="<%" endTag="%>"
<% Map context = (Map)argument;
String instanceAlias = (String)context.get("instanceAlias");
String curr_package = (String)context.get("package");
//...
%>
package <%=curr_package%>;
import java.util.*;

@DesignPattern( patterName = PatterNames.Observer, instanceAlias =  <%=instanceAlias%>,
        roleName = RoleNames.Subject )
public class Subject<%=instanceAlias%> {

  /*included and generated subparticipants */
  <%= includeParticipant("observers") %>
  <%= includeParticipant("attach") %>
  //...
  <%= includeParticipant("notifyObservers") %>
}
```

Realization (2)

- If an element - feature has more possible variants, the template of such element contains the source code for both variants distinguished by annotations.
- The following notation has been introduced for the variant attribute of annotations: [~]Attribute_name = value[;] (“~” expresses negation and more values can be joined via “;”)
- When the element - feature has more than one possible variant, the developer’s selection is compared with annotations in the template and in consequence, the desired variant of element – feature is generated.
- Example of notifyObservers template which contains two different variants distinguished by annotations (notice difference of variant attributes of annotations).

```
<%@ jet class="Observer-notifyObserversTemplate" package="dp.anot.jet" startTag="<%" endT
<% Map context = (Map)argument;
String instanceAlias = (String)context.get("instanceAlias");
String observersClassName = "";
if(instanceEvolution)
 observersClassName = getObserverClassName(PatterNames.Observer,
         instanceAlias,RoleNames.Observer);
if(observersClassName.equals(""))
 observersClassName = "Observer"+instanceAlias;
//...
%>
@DesignPattern( patterName = PatterNames.Observer, instanceAlias =  <%=instanceAlias%>,
        roleName = RoleNames.notifyObservers, variant = "modelOfNotification = callback")
    public void notifyObservers<%=instanceAlias%>() {
        for (<%=observersClassName%> o : <$=observersCollectionRefName>)
      o.<%=updateMethodName%>(this);
    }

@DesignPattern( patterName = PatterNames.Observer, instanceAlias =  <%=instanceAlias%>,
        roleName = RoleNames.notifyObservers, variant = "modelOfNotification = sending")
    public void notifyObservers<%=instanceAlias%>() {
        for (<%=observersClassName%> o : <$=observersCollectionRefName>)
      o.<%=updateMethodName%>(this.get<%=subjectStateClassName>();
    }
```

Realization (3)

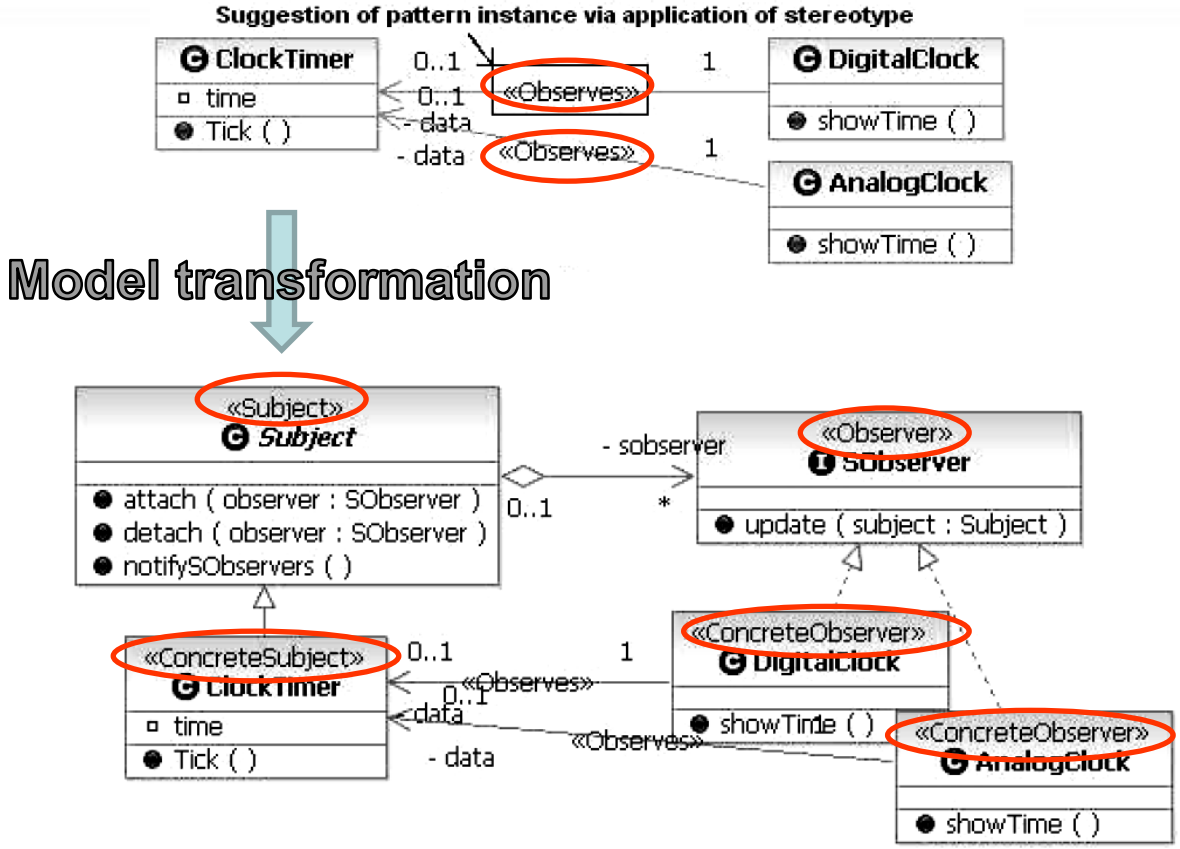
- The whole method is based on the defined name conventions.
- The names of feature models are identical to the PatternNames used in the source code annotations
- The feature names are identical to the RoleNames used in the source code annotations.
- The templates are named as follows: PatternName-RoleNameTemplate.
- In consequence, the support mechanism is able to automatically deduce from the annotations typed by the developer in the source code which feature model and which templates it should load and generate.
- The presented realization is intended for Java platform, but it can be simply adjusted also for other platforms, even if they do not support source code annotations. In such case the annotations may be enclosed in comments.

Elimination of Manual Annotation of the Source Code

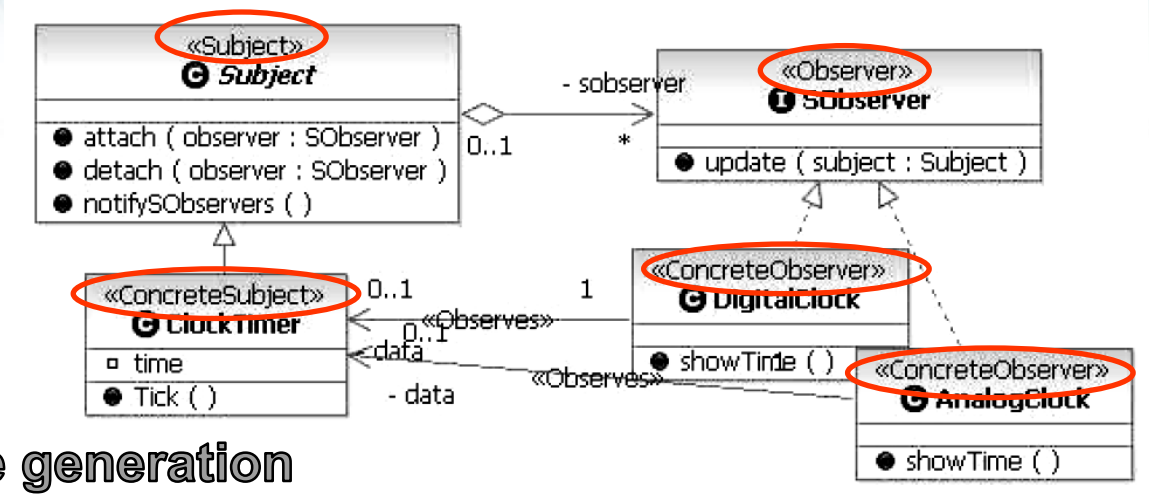
- The instances of patterns applied into the source code via the presented method are automatically generated into the source code with the annotations of all instance participants. So the evolution of these instances is supported by the method directly, without any need of manual annotation.
- For other instances of patterns we propose a mechanism of architectural and design information propagation and expansion from higher levels of abstraction (i.e. models) into the source code.
- For the purpose of architectural and design decision suggestion in the model we use the UML profile
- UML profile enables to define semantic extension of UML in form of semantic marks (i.e. stereotypes) and their meta-attributes (i.e. tagged values), enumerations and constraints.
- As a consequence, the developer is able to suggest architectural or design decisions or patterns via application of the stereotypes from the UML profile onto specific model elements.
- Suggested instances of patterns are concretized by the transformation of a model to a model in the next step.
- The transformation generates missing structural participants of pattern instances and it also marks each pattern participant with the appropriate stereotype. So the transformation propagates and expands applied marks on particular elements in the concretization process as the pattern instances are concretized.

Elimination of Manual Annotation of the Source Code (3)

- An example of Observer pattern application and visibility propagation



Elimination of Manual Annotation of the Source Code (4)



Source code generation

```
@DesignPattern( patterName = PatterNames.Observer,
        instanceAlias = "obs1",
        roleName = RoleNames.Subject,
        variant = "encapsulateSubjectState = false; managerType = noManager")
public class Subject {

    @DesignPattern( patterName = PatterNames.Observer,
            instanceAlias = "obs1",
            roleName = RoleNames.ObserversCollection)
    ArrayList<SObserver> observers = new ArrayList<SObserver>();
    //...

    @DesignPattern( patterName = PatterNames.Observer,
                    instanceAlias = "obs1",
                    roleName = RoleNames.Attach)
    public void attach(SObserver observer){
        observers.add(observer);
    }
    //...
```

Conclusion and Future Work

- The presented method fits in wider context of pattern support based on semantics and subsequent model transformations or source code generation.
- The proposed definition of annotations introduces the semantics and clear visibility of pattern instances in the source code and in consequence it opens new opportunities to support various aspects of patterns, or even for correct reverse transformations of the code with the pattern detection.
- Available feature models of patterns also enable a possibility of live detection of pattern instances advanced defects.
- Because the manual annotation of the source code by developers is very lengthy and tiresome, we have proposed the approach of manual annotation elimination based on the idea of design information propagation and expansion from models into the source code.
- Although it does not deal with the problem of existing or legacy software systems, it provides a very useful way how to propagate and expand design information and how to prevent the problem of pattern instances invisibility in the source code towards the future.
- Besides, it does not have to be used only for patterns, but it can be simply adjusted for other architectural or design decisions as well.

Thanks for your attention