

# Solving Software Engineering Problems with Generic Patterns

Kai Koskimies  
Tampere University of Technology  
{imed.hammouda,kai.koskimies}@tut.fi  
<http://www.cs.tut.fi/~kk>  
<http://practise.cs.tut.fi>



## 1 Introduction

Software architecture is inherently multi-dimensional: any decomposition of a system hides other important structures.

Cross-cutting concerns:

- variation points
- maintenance tasks
- comprehensibility issues
- features, use cases
- global concerns (like persistence, distribution, etc.)
- design patterns, platform-specific patterns
- other design dependencies



# How are cross-cutting concerns managed?

## Industry

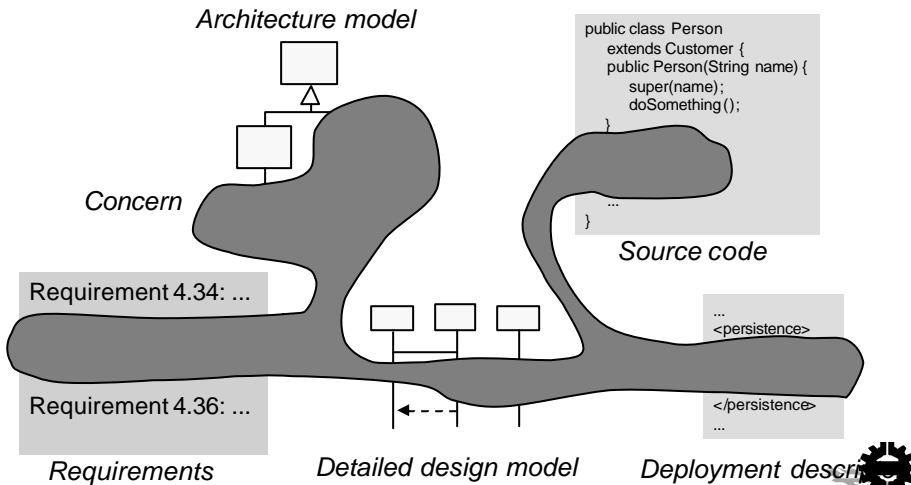
- documenting
- commenting
- programming conventions (e.g. naming)
- special tracing facilities
- personal communication, meetings
- metadata annotations
- ... but mostly not managed...

## Academia

- AOP, SOP, Multi-dimensional separation of concerns
- AspectJ, Hyper/J
- separate concern descriptions merged



# Challenge: multiple dimensions in heterogeneous software artifacts



## 2 Generic patterns

- Idea: use a pattern-like structural description to capture a slice consisting of system artifact elements participating in a concern (a *generic pattern*)
- An instance of a generic pattern is bound to elements in system artifacts
- A generic pattern:
  - can represent any concern
  - is not tied to particular artifact type or notation
  - is generative: supports the producing of artifacts

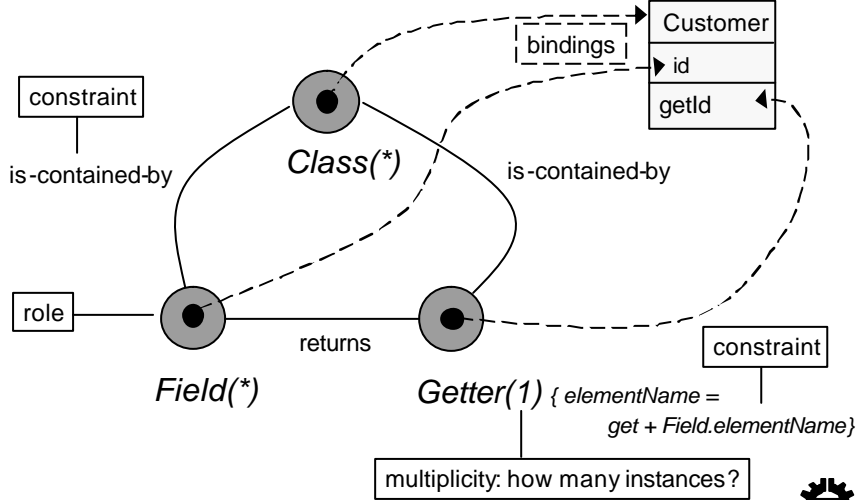


## What is a generic pattern?

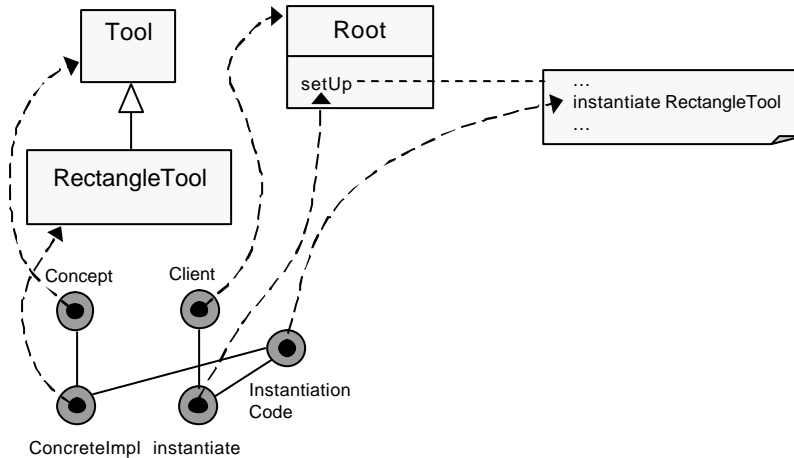
- A generic pattern is defined by:
  - a set of roles that can be bound to artifact elements
  - a set of constraints on the roles
  - descriptions of default elements to be bound to some roles
  - informal explanations about the pattern
- Constraints are used for:
  - stating the relationships that must be followed by the elements bound to different roles
  - enforcing certain attributes on the bound element
- Informal explanations:
  - instruct the pattern user to perform bindings
  - give information about the meaning of a role in the pattern
  - are adapted to current bindings



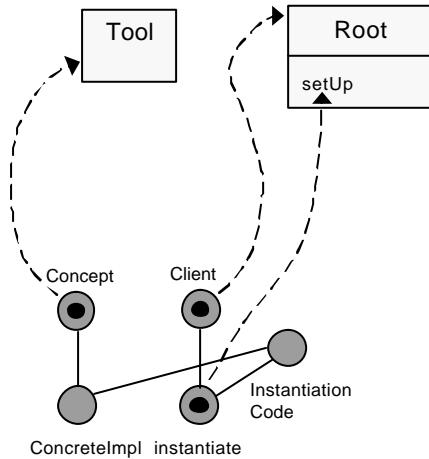
## Simple example: minipattern for a class



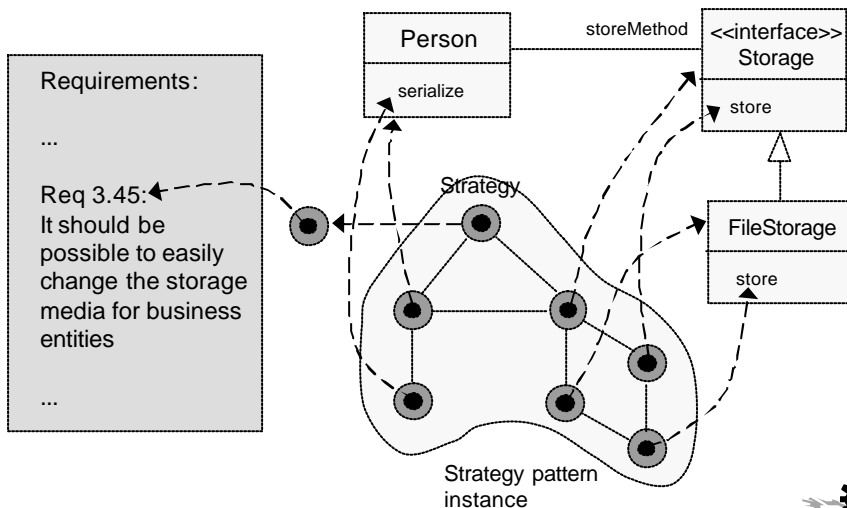
## Example: extension point



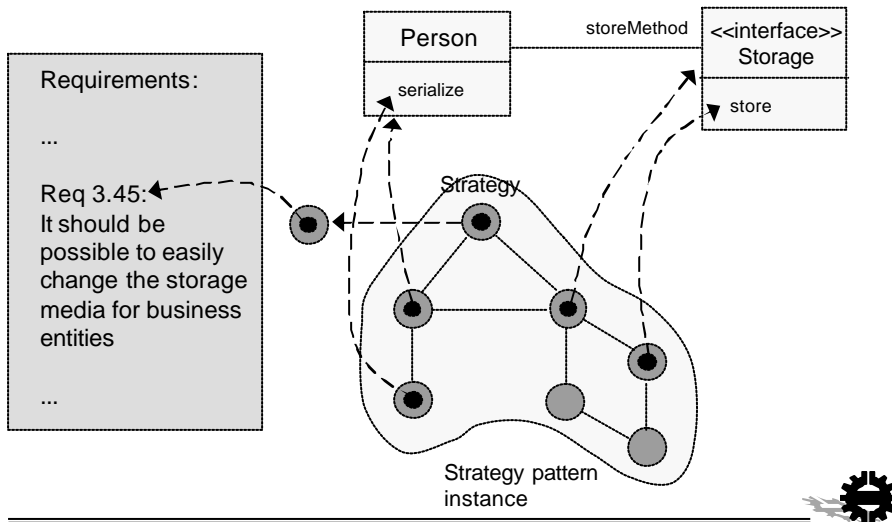
## Example: extension point



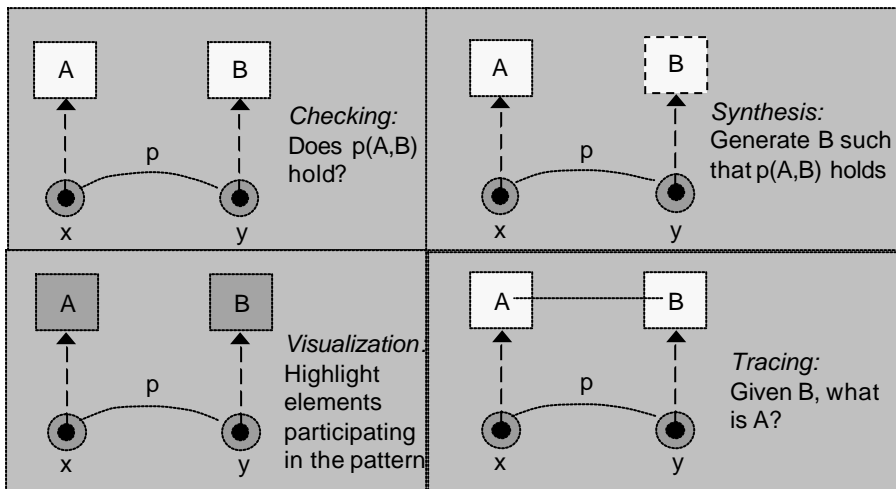
## Example: linking requirements with architectural solution



## Example: linking requirements with architectural solution



## Using generic patterns



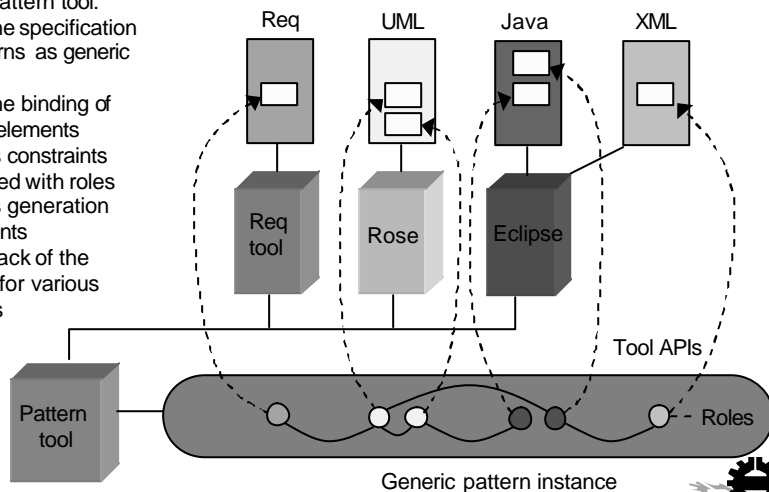
## Nice things about generic patterns

- The existence of a concern is completely detached from the system representation: the system is untouched
- Concerns presented as generic patterns can be arbitrarily overlapping: the same system element can play different roles
- An unbound or partially bound generic pattern instance can be transformed (by the tool) into a task list with explanations
- The binding process (or "weaving") is controlled by the designer: no magic
- The same mechanism can be used to a variety of purposes, depending on what entities are bound to the roles
- The mechanism is in principle neutral to artifact types and notations



## 3 Tool support for generic patterns

- Generic pattern tool:
- allows the specification of concerns as generic patterns
  - allows the binding of roles to elements
  - enforces constraints associated with roles
  - supports generation of elements
  - keeps track of the concern for various purposes



# User interface

Rose

A UML model being constructed according to a pattern

The used pattern library

A view to an instance of a pattern (roles)

A task list to carry out the bindings at this point

Instructions for doing a binding task

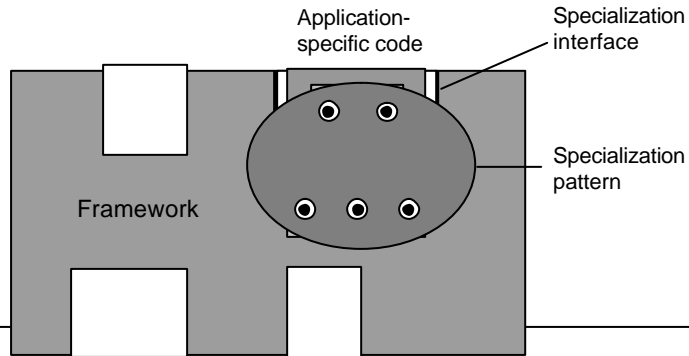
Eclipse based GUI

## 4 Application examples

- Framework specialization environment
- Maintenance support environment
- System learning environment
- Variability management environment
- Open MDA environment
- ...

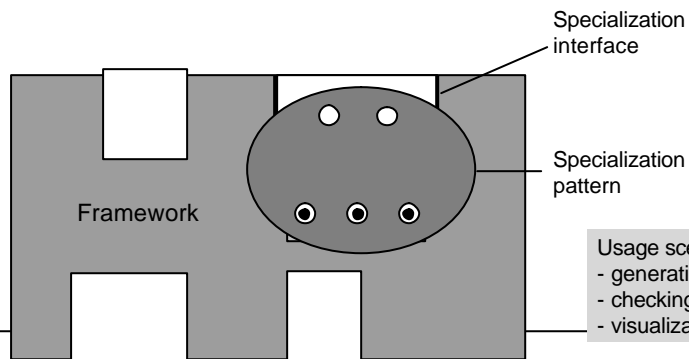
# Framework specialization

Framework = A collection of classes implementing the basic architecture for a family of software systems, to be *specialized* into an application by adding application-specific code.



# Framework specialization

Framework = A collection of classes implementing the basic architecture for a family of software systems, to be *specialized* into an application by adding application-specific code.



# Maintenance support

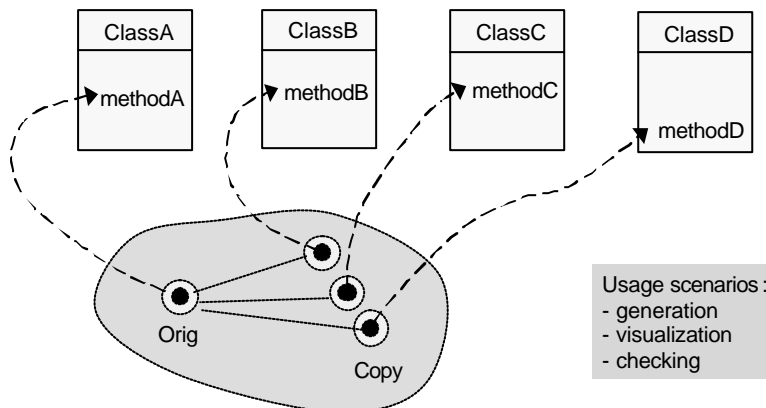
- Documenting maintenance concerns is valuable
  - for recording logical connection between different parts of the system that is relevant from the maintenance viewpoint
  - for repeating similar maintenance tasks
  - for preventing new maintenance tasks from corrupting the effect of the previous actions
  - for undoing maintenance tasks

Generic patterns can be used to capture the software elements related to a particular maintenance concern, with informal explanations concerning the related maintenance action ("maintenance patterns").



## Example: copy-and-paste problem

Implementation of one method copied (and possibly modified) into other methods :

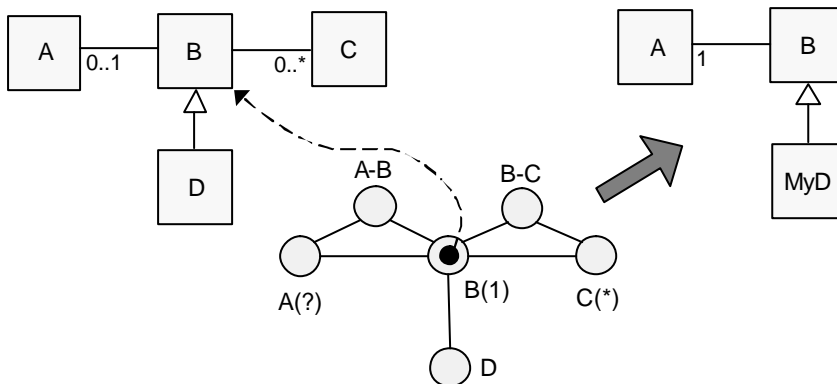


# System learning

- To employ an existing software library, its structure should be first learned and the required elements should be identified.
- The understandability of a complex system can be improved by graphical representations (e.g. UML).
- A graphical model (e.g UML class diagram) may contain information not relevant to the purpose of the learner.
- The learning process can be facilitated by a stepwise, guided construction of a model customized for the learner.



## Using generic patterns for model synthesis & customization





# Feature-driven variability management

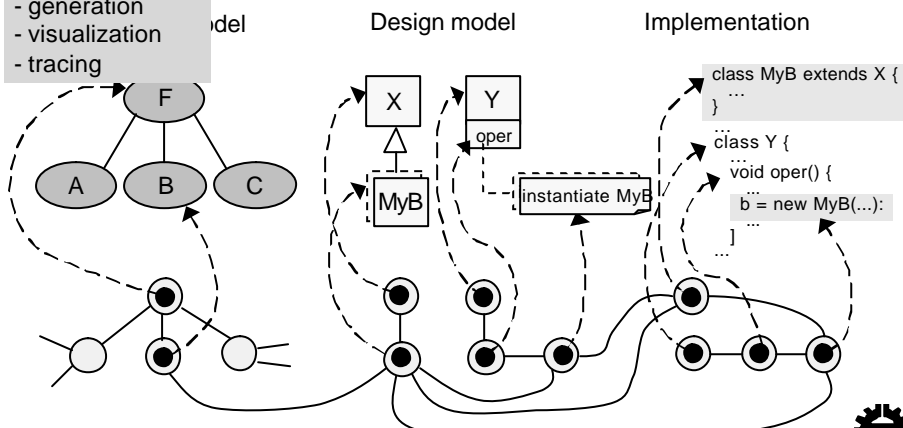
- A key issue in using product platforms: how to manage and use the variance offered by the platform to produce new applications.
- Ideally, given a set of required features (and qualities?), a tool could generate the application, or at least assist in the design.
- Black-box generation problem: the application designer loses track of the application structure.
- Generic patterns can be used to capture the design decisions related to a particular choice of features, and guide the designer through the application development.



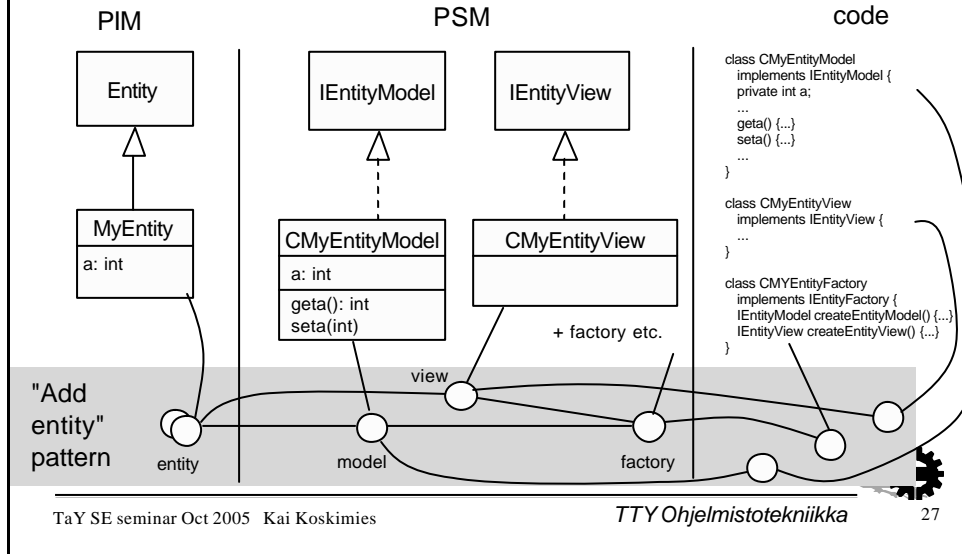
# Using generic patterns for feature-driven application development

Usage scenarios:

- generation
- visualization
- tracing



# "Open" MDA using generic patterns



## Literature

### Framework specialization:

Hakala M., Hautaniemi J., Koskimies K., Paakki J., Viljamaa A., and Viljamaa J.:  
Generating application development environments for Java frameworks.  
*GCSE 2001*. Springer LNCS 2186, 2001, 163-176.

### Maintenance:

I. Hammouda, M. Harsu: Documenting maintenance tasks using  
maintenance patterns. *International Conference on Software Maintenance &  
Reengineering 2004*.

### System comprehension:

I. Hammouda, O. Guldogan, K. Koskimies, and T. Systä: Tool-supported  
customization of UML class diagrams for learning complex system models.  
*International Workshop on Program Comprehension 2004*.

### MDA model transformation:

M. Siikarla, K. Koskimies and T. Systä: Open MDA using transformational patterns.  
*MDAFA 2004*, Linköping, Sweden, June 2004.

### Feature-driven variability management:

I. Hammouda, J. Hautaniemi, M. Pussinen and K. Koskimies: Managing variability  
using heterogeneous feature variation patterns. *FASE 2005*, Edinburgh, April 2005,  
Springer LNCS 3442, 145-159.

## 5 Conclusions

- The relatively simple concept of a generic pattern has turned out to be a surprisingly versatile mechanism for a variety of purposes.
- A light-weight infrastructure for representing concerns: systems are still developed using conventional methods.
- But: if you have a hammer, everything looks like a nail.
- Is it worthwhile to use this concept (and tool support) in practice? Some experiments have been made, but we are still investigating the usefulness of the approach in industrial context.
- Limitations: relies on good APIs and easy ways to identify "elements" in an artifact. Backtracking is still a problem.



## Metadata in Java

```
@Stateful public class CartBean implements ShoppingCart {
    @Resource(name="myDB") // type is inferred from variable
    private DataSource database;

    @EJB private ProductService products;

    @Inject public UserTransaction utx;
    @Inject SessionContext sessionContext;

    private float total;
    private List<Product> shoppingCart;
    public void buy() {...};
    ...
    @PreDestroy endShopping() {...};
}
```



## Related work

There are many approaches which address the same or similar problems

Characteristic properties of our approach:

- language & artifact neutrality (e.g. AspectJ, Hyper/J, framed aspects)
- working tool support (e.g. MDSoc)
- relies on UI rather than formalisms (e.g. Batory)
- generation of small-grained tasks from patterns (AOP, Batory)
- "open" weaving of an aspect (AOP)
- actual system is conventional and untouched (AOP)
- no links between elements in artifacts (requirements tracing tools)
- solution for multiple problem types: checking, generation, tracing, visualization (e.g. Reiss)

