PATTERN - ORIENTED FRAMEWORK ENGINEERING USING FRED

Markku Hakala*, mh@cs.uta.fi
Juha Hautamäki*, csjuha@uta.fi
Jyrki Tuomi*, jjt@cs.uta.fi
Antti Viljamaa†, antti.viljamaa@cs.helsinki.fi
Jukka Viljamaa†, jukka.viljamaa@cs.helsinki.fi

Abstract

Application frameworks are reusable architectures used to improve the software development process. Although design patterns may be used in designing, implementing, and documenting architectural constructs, lacking a systematical approach frameworks are still hard to design and reuse. This paper presents the methodology and the set of supporting tools that provide a methodical practice for applying design patterns in software engineering, especially when constructing reusable designs such as application frameworks.

Keywords

Framework, Design Pattern, FRED, CASE Tool, Software Engineering, Java

INTRODUCTION

An application framework is a set of objects that captures the special expertise in some application domain to a reusable form [Lew95]. An application can be specialized from this skeleton by adding new features to it.

Frameworks are hard to design and reuse. However, design patterns [GHJ95, CoS95] can be used to capture the knowledge of framework experts in an easily accessible way. Design patterns name and explain architectural constructs that may be found in number of architectures. A design pattern is an abstract solution to a general design problem, and may be reused by individual developers as building blocks for new frameworks and applications. [Joh92, BMR96]

FRED (Framework Editor) is a development environment, especially designed for framework development and specialization. In addition of being a development tool, FRED introduces a uniform model of a software architecture and software development that makes heavy use of generalization of design patterns.

The FRED environment has been implemented with Java programming language [ArG98] and is used in developing Java applications and frameworks. However, the model behind the environment is not tied to any specific programming language or tool set.

FRED is an ongoing project between the departments of Computer Science at the University of Tampere and University of Helsinki, supported by TEKES (Technology Development Centre, Finland) and several Finnish industrial partners.

FRED MODEL

Both frameworks and applications are software architectures. FRED, as a development environment, is a tool for creating such architectures. In FRED, an architecture is always created based on another architecture or architectures. A typical example is an application that is derived from an application framework.

Designated by the object-oriented domain, each architecture eventually consists of classes and interfaces, which in turn contain fields (Java synonym for attributes) and methods. Also, the term *data type* is used to refer to both classes and interfaces.

Data types alone are insufficient to represent architectural constructs when reusability is essentially required. They do not provide enough documentation for the architecture, nor control the specialization of the architecture. To meet these two requirements, *pattern* is hereby defined as a description of an arbitrary relationship between number of classes and interfaces. Patterns range from generic design patterns to domain and even application specific patterns. A pattern is an architectural description, but needs not to be general. In this context, general constructs such as those listed by Gamma *et al.* [GHJ95] are called *design* patterns. No distinction between patterns on the basis of their generality is made in FRED.

* University of Tampere, Department of Computer Science, P.O. Box 607, FIN-33101 Tampere, Finland
† University of Helsinki, Department of Computer Science, P.O. Box 26, FIN-00014 University of Helsinki, Finland

Patterns are used to couple together arbitrary data types that participate in a particular design decision or architectural feature. This kind of coupling of data types provides structural documentation for the architecture. Any data type may participate in more than one pattern, in which case it plays several roles in the architecture.

Structures

An architecture is a complex construction of patterns, data types and both their static and dynamic relations. Structural elements of an architecture, such as patterns, data types, methods and fields are called *structures*. Also, the architecture itself is a structure. Adopting the general concept of structure simplifies both the discussion and internals of the FRED environment.

Structures are divided into composite and leaf structures. Architectures, patterns and data types are composite structures, which may contain other structures. An architecture contains patterns, patterns contain data types, and data types in turn contain methods and fields, which are leaf structures in the sense that they do not contain other structures.

Based on these relations, a directed acyclic graph can be presented for an architecture. Structures of the architecture are nodes of this graph, and edges range from composite structures towards the leaf structures. The graph has single source, namely the architecture node, and multiple sinks representing the leaf structures. As a data type can belong to a number of patterns, the graph is not a tree. In FRED development environment, the graph is however visualized with a tree-like notation. In this notation, the root equals the source of the graph. Furthermore, a data type contained in multiple patterns appears in the tree at each containing pattern, but only methods and fields relevant to a particular pattern are shown. Based on this notation, a structure containing another structure is called a *parent* of the latter. The parent of a structure is naturally not necessarily unambiguous.

An example architecture is shown in figure 1 with both graph representation and the corresponding FRED notation.

Templates

All structures may be classified as *implementations* or *templates*.

An implementation is a structure that is actually implemented in the architecture. In a conventional application, all structures are essentially implementations. From the developer's point of a view, an implementation is a "normal" structure, in contrast to the concept of template.

A template defines a set of possible implementations, but does not specify the actual implementation. In a way a template is a blueprint of an implementation. Providing a template in an architecture means defining a gap that must be filled in by the developer deriving from the architecture.

Templates are structures just like implementations. An architecture template contains patterns, a pattern template contains data types, and data type templates contain methods and fields. Architecture and pattern templates may contain both templates and implementations, but data type template contains only templates. If a structure contains a template, it is itself a template.

In FRED, a structure (including architecture) is always based on other structures, namely templates; every structure is based on corresponding meta-structure and in addition to some other templates. There is a meta-structure for each type of structure. Meta-structures are based on themselves.

Templates are used in creating new structures. This is called *instantiating* the template. The instantiated template is called a *model* in relation to its *instance*. For instantiation purposes a template provides the following properties:

1. Free-form hyperlinked documentation that guides in creating an instance for the template.
2. Parameterized default implementation that automatically adjusts to the instantiation environment.
3. Tools for instantiating the template.
4. Constraints that all its instances must conform to.

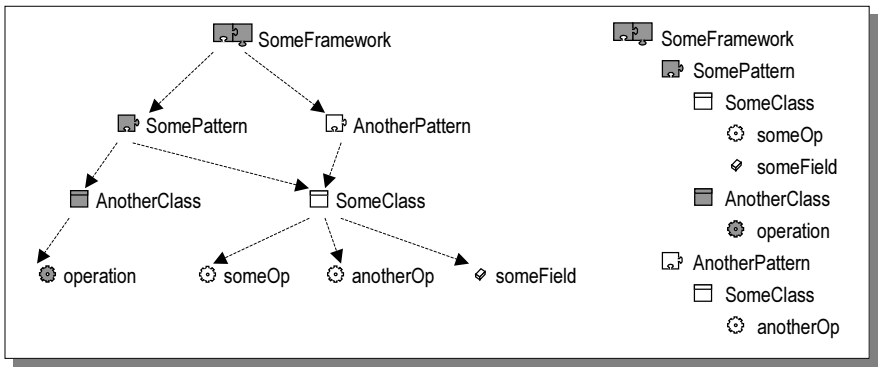


Figure 1. An example architecture as a directed graph and using FRED tree-like notation.

Template is instantiated by introducing a structure (an implementation or a new template) that is conformable to the model. This instance can be created using the documentation, default implementation, and tools provided by the template. By default the FRED environment includes tools for creating and editing structures, but the architecture developer may also provide specialized domain-specific tools.

Once a structure is created the default tool or tools suggested by the template can be used to modify the structure. Constraints are used to ensure that the instance conforms to the template. For example, a constraint may state how many times the template must be instantiated with respect to its parent template (cardinality constraint). See appendix A for more examples on constraints.

An *automatic* template is a template that is instantiated automatically whenever its parent structure is instantiated. Instantiating an automatic template may however require user interaction, e.g., when all the parameters of the default implementation cannot be resolved. Automatic or tool-generated structures are updated according to the modifications in surrounding code. The developer cannot access the automatically maintained code fragments unless explicitly requested.

Patterns Using Templates

By the earlier definition, a pattern describes a relationship between participating data types. In FRED, a pattern is described using templates.

A pattern template couples together data type templates and data type implementations. The constraints of the contained templates define the required relationships between collaborating structures. The default implementation makes it easy to instantiate a pattern in a software architecture. Hyperlinked documentation attached to the templates provides the required documentation for the pattern and its collaborations. In addition, specialized tools may be provided. For example a template representing the Interpreter design pattern [GHJ95] may be accompanied with a tool that takes a grammar as input and parameterizes the default implementation with it.

Instantiating a (design) pattern means binding the domain-specific vocabulary and implementation. Frameworks usually provide only partial implementations for design patterns and leave specific parts to be supplemented by the specializer. In FRED this means providing templates that instantiate the original templates of the pattern. This instantiation chain may be arbitrary long for any structure. Constraints of a template apply to all following instances in the chain. Thus constraints cumulate and the set of possible implementations becomes smaller in every instantiation. This implies kind of inheritance hierarchies for frameworks and design patterns. Layered frameworks are discussed for example by Koskimies and Mössenböck [KoM95].

A BRIEF EXAMPLE

To illustrate how the FRED model works in practice, a simple framework is implemented and an application is derived from it. The example is based on the sample *framelet* presented by Pree and Koskimies [PrK98]. In their paper they define framelets as small architectural building blocks that can be easily understood, modified, and combined. Here the example framelet appears somewhat modified. The reader may consult appendix B when going through the following example.

Many graphical applications provide list boxes together with buttons to add items to the list box, and to modify and remove them. Typically a separate dialog is used to add and modify items. The associated programming task can be packed into a small self-contained framework, called List Box Framelet. The framelet class diagram and user interface are shown in figure 2.

The framelet provides a window (ListBoxFrame) to display and maintain a list of items. An application developer provides item and dialog classes. Item classes must implement the Item interface and dialog classes must extend the Display base class. Transformer class is used internally by the framelet to manage the data transfer between FieldAccessor instances, namely items and displays.

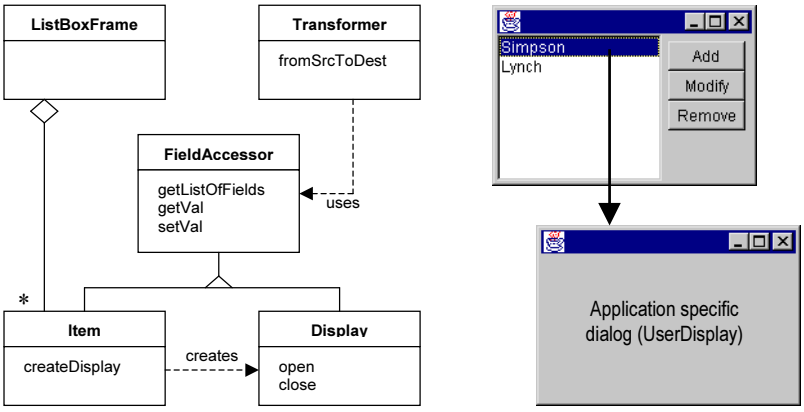


Figure 2. Class diagram and user interface for List Box Framelet.

Creating the Framelet

In FRED List Box Framelet must be presented with structures. In the following example, names of structures are printed with sans serif font (e.g. ListBoxFramelet).

As an architecture, List Box Framelet must be based on another architecture. FRED environment provides a special architecture called PatternCatalog, which collects arbitrary design patterns by several authors. PatternCatalog can be expanded by individual developers. The root structure of the framelet, an architecture called ListBoxFramelet, is based on PatternCatalog.

Among other things the catalog contains patterns AbstractFactoryPattern, SingletonPattern, and BeanComposition. The first two are representations of design patterns Abstract Factory and Singleton, which are discussed by Gamma *et al.* [GHJ95]. The BeanComposition in turn stands for the Bean Composition pattern, which represents a composition of Java bean components [Sun97] and is developed within FRED project.

After browsing the documentation of the catalog, the developer chooses to instantiate AbstractFactoryPattern to implement the framelet's behavior. The developer names the instance as ItemPattern. To implement the created pattern, all of the templates contained in the model pattern must be instantiated according to their cardinality. AbstractFactoryPattern contains data type templates AbstractFactory, ConcreteFactory, AbstractProduct and ConcreteProduct. The developer creates the Item interface and the Display class based on the AbstractFactory and AbstractProduct templates. Implementation of ConcreteFactory and ConcreteProduct the developer wants to leave to the specializer. For this purpose, the developer creates templates UserItem and UserDisplay, respectively.

All structures are also based on the corresponding meta-templates. Similarly, ItemPattern is based on meta-pattern. For this reason, the developer can augment ItemPattern by introducing FieldAccessor interface by instantiating it from meta-datatype. Likewise, a number of methods and fields can be created that are not based on method and field templates contained in AbstractFactoryPattern.

Data types that are not involved in any specific pattern, are contained in a special OtherClasses pattern, which is automatically generated for an architecture. The developer creates the Transformer class in that pattern, but soon realizes that SingletonPattern may be used here. The developer creates a new SingletonPattern instance called TransformerPattern and associates the existing Transformer class as an instance of the Singleton template. Members required by the Singleton template may be pointed out similarly from the existing code. The created structures and their relations to their models are shown in figure 3.

After creating most of the framelet logic, the developer begins to implement the user interface. This can be done by instantiating BeanComposition pattern, which is equipped with a FRED Java Beans builder tool that generates appropriate code for the user interface.

Because the framelet contains a template (ItemPattern) it is itself a template which can be instantiated. The developer chooses to make ItemPattern automatic implying that once the framelet is instantiated, the ItemPattern template is also instantiated. Finally, the developer provides guidance for the framelet specialization by documenting the contained templates.

Specializing the Framelet

Any architecture that is a template can be derived from by another architecture. E.g., List Box Framelet can be used to create an application to create, modify, and remove customer contact entries. The Contact Manager application may be based on multiple architectures, but for simplicity, only the List Box Framelet –based portion is discussed here.

The developer names the root structure of the application as ContactManager. At first the application developer makes it an instance of ListBoxFramelet. As ItemPattern is an automatic template, the environment auto-

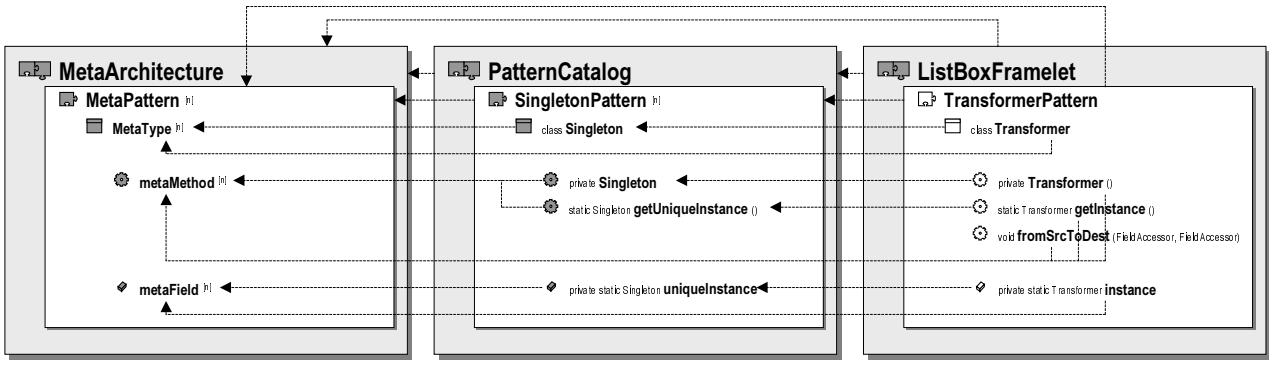


Figure 3. TransformerPattern is based on SingletonPattern. All structures are based on the corresponding meta-structures.

matically creates an instance of it. The developer changes the name of the instance to ContactPattern.

ItemPattern contains template classes UserItem and UserDisplay. These are not instantiated automatically. Hence, the application developer creates the Contact class based on UserItem and ContactDisplay based on UserDisplay. Substructures (methods and fields) of the templates are instantiated similarly. Template specific documentation and constraints guide the developer during this instantiation process.

To implement the user interface, the application developer may use the BeanComposition pattern. To do this, the developer will need to point out the application as an instance of PatternCatalog and instantiate the pattern. The developer uses the pattern to create user interface for ContactDisplay. For this purpose ContactDisplay is made an instance of the Composition template. Thereafter ContactDisplay is an instance of two templates. The Java Beans builder tool is used to generate the user interface –specific code for the ContactDisplay class.

As a result, the Contact class represents a contact entry and the ContactDisplay dialog is used by the framelet to add and modify contacts. Because there are no violated constraints, the developer chooses to test Contact Manager by implementing a class with the required main method. The developer creates the class in the OtherClasses pattern.

FRED Tool

Although the FRED model is not tied to any specific tool, the model requires a sophisticated user interface in order to lessen cognitive load on the developer. The user interface should adjust to the level of abstraction of the architecture by hiding the internals of the architecture and using more specific vocabulary when necessary. For this reason, FRED is as much of a user interface issue as it is a conceptual model, though only the latter has been discussed here.

However, part of the user interface for FRED development environment is shown in figure 4. The window is divided into three parts. Hierarchies on the left are used to browse the architecture from different aspects. Structures are modified on the desktop area with the template tools, and the lower part displays documentation for the selected hierarchy item. The same environment can be used in developing architectures ranging from applications to frameworks and even abstract pattern catalogs.

In figure 4 the developer is implementing the ContactManager architecture. The ContactDisplay class has been selected on the left. Models of the selected structure are listed below with their associated tasks. *Tasks* are user interface representations of unimplemented templates and broken constraints. UserDisplay is listed as a model because ContactDisplay is based on it. The associated tasks state that the developer has to instantiate the getVal and setVal method templates.

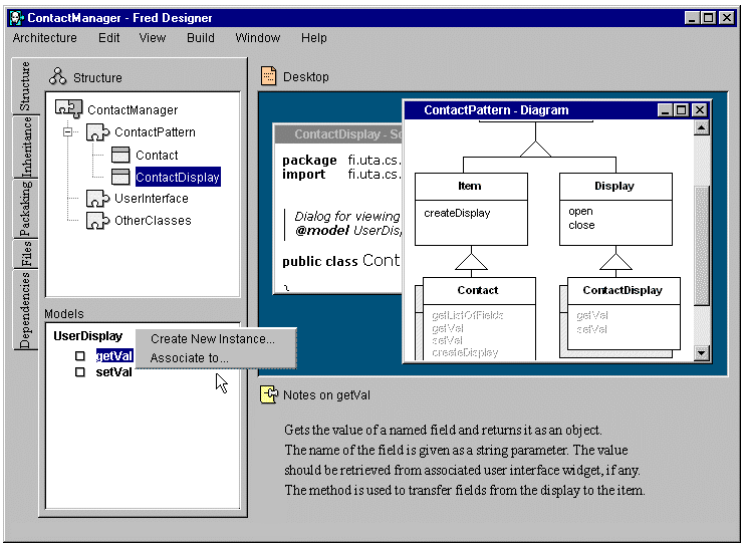


Figure 4. The user interface of the FRED development environment in action.

CONCLUSION AND RELATED WORK

Many design pattern tools (see, e.g., [BFV96], [ACL96], and [Wil96]) use macro expansion mechanisms to generate implementation code. This implies design – implementation gap [MDE97]: changing generated code breaks the connection between design patterns and the implementation and any changes that involve regeneration of code will force the user to manually reintegrate the pattern code to the rest of the system. We think that a better way is to use explicit representation of (design) patterns that stays at the background all the way from design to implementation.

Furthermore, mere code generation is not enough. It is essential to be able to combine multiple pattern instances and to annotate existing code with patterns. Our implementation of this role binding functionality is influenced by Kim and Benner's Pattern-Oriented Environment (POE) where one can manage and validate pattern instances and their role mappings [KiB96].

Another similar tool is represented by Florijn, Meijers, and van Winsen [FMW97]. Their model is based on cloneable fragments that represent design elements (design patterns, charts), implementation entities (classes, methods), or other objects (comments, arbitrary

text). The system being developed is represented as a graph of interrelated fragments of different types.

Besides supporting development of frameworks, the FRED environment also aids framework users. Instantiating any structure in FRED involves customization of default implementations within the bounds of constraints associated with the structure (in a specialized *visual* or *active text* [MöK95] editor) guided by the associated documentation. This makes, e.g., deriving an application from a framework a systematic process.

One major shortcoming of the adopted model is that constraints defined in the templates apply only to method and field signatures, data types, and larger constructs of the architecture. E.g., FRED environment cannot verify that the code within a method body is valid while it can ascertain that the method's return type is correct.

As simplicity for the end-user was one of the main concerns of FRED, the model is more than adequate. Extensive systems involving formalisms like predicate logic tend to chase developers away. FRED model hopefully provides a relatively easy approach to pattern-oriented software engineering.

Some of the functionality referred to in this paper is not implemented in the current version of the FRED environment. In addition to making the FRED environment fully operational, future directions of project include:

- Pattern selection wizard to assist the user in finding a solution to a design problem.
- Support for pattern mining by associating an appropriate tool for a pattern template.
- An automatic mechanism for creating customized wizards to assist the user in framework specialization.
- Importing Java standard extensions and similar popular frameworks (by JavaSoft) into the FRED model.
- Visualization of both static and dynamic relations within an architecture.

ACKNOWLEDGEMENTS

FRED project is supported by TEKES (Technology Development Centre, Finland) and Finnish industrial partners: Dycom, ICL Data, Major Blue Company, Nokia Telecommunications, Nokia Research Center, Novo Group, Profit, Stonesoft, Sun Microsystems, Telecom Finland, TT Tieto, and Valmet Automation.

REFERENCES

ACL96 Alencar P., Cowan D., Lichtner K., Lucena C., Nova L.: Tool Support for Design Patterns. Internet: ftp://csg.uwaterloo.ca/pub/ADV/theory/fmsp96.ps.gz.

ArG98 Arnold K., Gosling J.: *The Java Programming Language*, 2nd ed. Addison-Wesley, 1998.

BFV96 Budinsky F., Finnie M., Vlissides J., Yu P.: Automatic Code Generation from Design Patterns. *IBM Systems Journal 35*, 2, 1996, 151-171.

BMR96 Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *Pattern – Oriented Software Architecture, A System of Patterns*, John Wiley & Sons Ltd, 1996.

CoS95 Coplien J., Schmidt D. (eds.): *Pattern Languages of Program Design*. Addison-Wesley, 1995.

FMW97 Florijn G., Meijers M., van Winsen P.: Tool Support for Object-Oriented Patterns. *Proc. ECOOP '97 European Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 1997, Lecture Notes in Computer Science 1241, Springer-Verlag, 1997, 472-495.

GHJ95 Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Joh92 Johnson R.: Documenting Frameworks with Patterns. In: *Proc. OOPSLA '92, ACM SIGPLAN Notices 27*, 10, 1992, 63-76.

KiB96 Kim J., Benner K.: An Experience Using Design Patterns: Lessons Learned and Tool Support. *Theory and Practice of Object Systems (TAPOS)* 2, 1, 1996, 61-74.

KoM95 Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalization. In: Proc. of ESEC'95, Lecture Notes in Computer Science 989, Spinger-Verlag, 1995, 479-497.

Lew95 Lewis T. (ed.): *Object-Oriented Application Frameworks,* Manning Publications Co., 1995.

MDE97 Meijler T., Demeyer S., Engel R.: Making Design Patterns Explicit in FACE — A Framework Adaptive Composition Environment. In: *Proc. 6th European Software Engineering Conference*, Zurich, Switzerland, September 1997, Lecture Notes in Computer Science 1301, Springer-Verlag, 1997, 94-110.

MöK95 Mössenböck H., Koskimies K.: Active Text for Structuring and Understanding Source Code. *Software Practice & Experience 26(7)*, July 1996, 833-850.

PrK98 Pree W., Koskimies K.: Framelets - Small and Loosely Coupled Frameworks. Manuscript, submitted for publication, 1998.

Sun97 Sun Microsystems Inc.: JavaBeans Documentation. Internet: http://java.sun.com/beans/docs

Wil96 Wild F.: Instantiating Code Patterns — Patterns Applied to Software Development. *Dr. Dobb's Journal 21*, 6, 1996, 72-76.

APPENDIX A – FRED NOTATION AND CONSTRAINTS

In FRED, the structural graph of an architecture is presented with a tree-like notation consisting architecture, pattern, data type, method, and field nodes. Each node contains an image indicating the type of the structure and the name. Templates come with dark gray images and implementations with white ones. Other information, such as implementation information or constraints may also be presented with a node.

In the following table, notation for template nodes and some of their most important constraints are presented. Notation for implementations follows closely the syntax of Java language and is not discussed. Also, more specific constraints and parameterized default implementations are bypassed as they are not included in this notation.

Some of the constraints are interpreted differently in an overriding situation. For this purpose, c*onstraint signature* for a method is defined as collection of constraints for return type, parameters, and exceptions. When a constraint signature of a method template equals a signature or constraint signature of another method in any of the supertypes, the method is said to override the other method. The signature of its instance should be exactly the same as that of the overridden method (or its instance).

	NOTATION	ALLOWED MODIFIERS
Architecture	**name**	
Pattern	modifiers **name** [cardinality]	automatic
Data Type	modifiers kind **name** [cardinality] supertypes	automatic, public, pckgprivate, abstract, final
Method	modifiers return_type **name** [cardinality] (parameters) exceptions	automatic, public, pckgprivate, protected, private, static, abstract, final, synchronized, native
Field	modifiers return_type **name** [cardinality]	automatic, public, pckgprivate, protected, private, static, final, transient, volatile

cardinality

Cardinality states how many times a template must be instantiated with respect to its parent template. By default, cardinality is one and need not to be shown. The notation [n] is a shortcut for [0..n] and means that the template can be instantiated any number of times.

modifiers

Modifiers are equal to Java-modifiers, supplemented by a few additional keywords. However, the meaning of a modifier for a template is different from the meaning of a modifier for an implementation; the existence of a modifier in template sets a constraint on the structure. E.g., *automatic* keyword means that the structure is instantiated automatically once its parent is instantiated, and *private* means that the instantiated structure must be private. If there are no modifiers present, no associated constraints are applied.

kind

A special modifier for a data type is its kind. A kind may be *class* or *interface*. It defines whether the instance data type should be class or interface, respectively. If missing, the instance can be either one, assuming that other constraints are not violated.

supertypes

Required supertypes for an instance may be indicated by *extends*, *implements*, and *inherits* clauses. By using the latter alternative, the type of inheritance is not fixed and subtypes of the mentioned data type are also accepted.

return_type

Both methods and fields can be associated with a return type. Existence of a return type states that the instance should return the mentioned type or any of its subtypes.

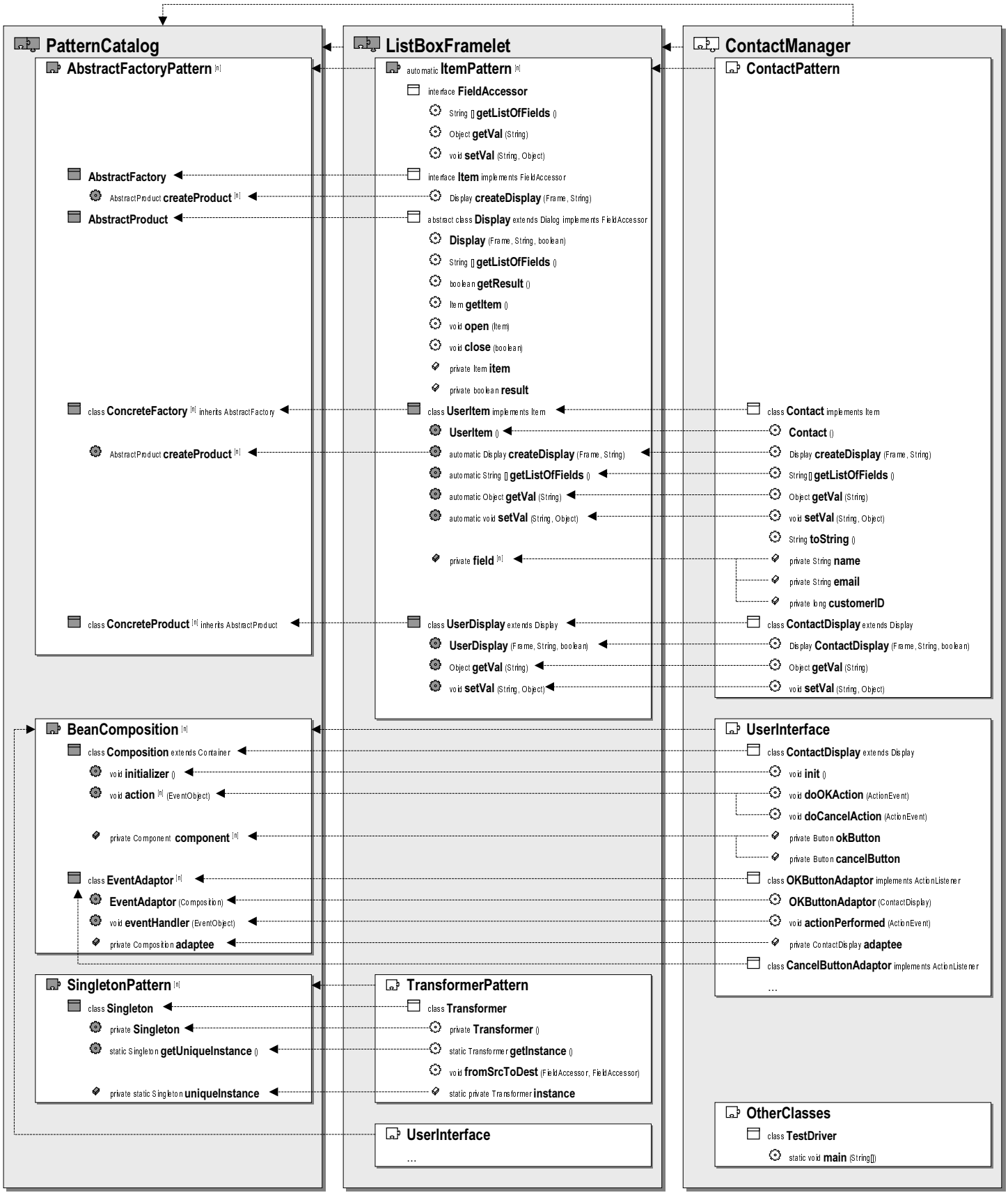
parameters

Formal parameters may be left unconstrained for a method template. Similar to the return type constraint, existence of a formal parameter states that the method should accept a parameter of the mentioned type or any of its subtypes. At the same time the order of parameters is fixed.

exceptions

Like parameters, a method can be constrained for the exceptions it must introduce in its throws clause. Exceptions-constraint follows the syntax of throws-clause in Java-language.

APPENDIX B – EXAMPLE ARCHITECTURES



For simplicity, meta-structures are not shown in this example.