Design-Decision-oriented Programming

A Paradigm for Property-based Selection of Reusable Implementations

Tobias Dürschmid[a]

a Hasso Plattner Institute, University of Potsdam, Germany

Abstract A proper architecture supports software evolution. Therefore, architectural design decisions have a large influence on the changeability and extensibility of software. However, developers often do not pay much attention to the non-functional properties of a system since they focus on implementing features rather than explicitly making design decisions.

Hence, the resulting implementation might lack quality, context tailoring, and requirements traceability. Existing approaches for reusing standard implementations for reoccurring design questions let developers explicitly choose and build upon design decision implementations. Therefore they do not facilitate changeability of the design decisions.

To this end, a new paradigm, *Design-Decision-oriented Programming (DDOP)*, is proposed, that decouples the specification of required properties for a design question from the selection of the best-suited design decision implementation out of a design decision library. This shifts the focus of programming from implementation to design decision making.

DDOP targets the reduction of implicit assumptions about design decision implementations, changeability of the non-functional requirements, and transparent forward-compatibility to improved solutions of reoccurring challenges. Therefore, DDOP should reduce the accidental complexity of sophisticated software systems.

A case study is presented that exemplarily applies DDOP to caching and the involved design questions. It provides a proof of concept and demonstrates how DDOP can be implemented without introducing new programming language features.

DDOP aims for popularizing programming by enabling less experienced developers to create and maintain high-quality software.

ACM CCS 2012

- **Software and its engineering → Language types; Software development techniques;**

Keywords design decision, design question, non-functional property, software design, modularity, reusability

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Modularity and separation of concerns, General-purpose programming

1 Introduction

During software development, many design questions arise, for example, which replacement policy to use for caching, which encryption algorithm to use for network communication, or which culling technique to use for 3D rendering (see figure 1). Each design question can be answered by a corresponding set of design decisions (e.g., all kinds of replacement policies, all kinds of encryption algorithms, or all kinds of culling techniques). Each design decision offers different Non-functional Properties (NFPs) (e.g., performance, memory consumption, security). Hence, architectural design decisions and their traceability have a large influence on the changeability and extensibility of software [33]. However, developers often do not pay much attention to the NFPs of a system since they focus on implementing features rather than explicitly making and communicating rational design decisions [41, 47, 59]. Hence, the resulting implementation might face problems that are described in the following.

Implementation Effort Re-implementation of common design decisions can be error-prone, wastes time of the developer and usually does not reach the same quality as standardizated solutions implemented by domain experts. Therefore, treating design questions and their implementation as first-class citizens is desirable [29], motivated by the following example.

In the past, developers had to implement the collection data structures on their own. Since the first solution was implemented, the temptation to reuse it without rethinking about its application to other contexts was high. Nowadays, a lot of different implementations are part of many standard libraries (e.g., `Stack`, `Vector`, `HashSet`, `LinkedList`, `ArrayList`, `CopyOnWriteArrayList` in Java). The same happened to trees, graphs, and common algorithms such as sorting, hashing, and encryption. Since the domain of algorithms and data structures provides reusable standard implementations, developers just decide which implementation is best suited for their use case. They know about the trade-offs that are implied by using one of these solutions, or they can easily look them up in the documentation. It is easy to change a design decision just by changing the type of the list or implementing the corresponding interface. However, crosscutting concerns such as software patterns and other super-imposed roles lack support for these kinds of reusable, refinable standard libraries.

Required Expertise However, to make a good decision, developers have to know all solutions and their implications to the software's NFPs. This requires extensive knowledge of the domain and recent implementations. For example, to ensure thread safety of a Java list, developers have to know `CopyOnWriteArrayList` or `SynchronizedList`. When researchers develop new techniques, to solve a reoccurring problem, developers have to hear about them and learn them before their applications can use them.

Requirements Traceability The traceability of past design decisions and their underlying non-functional requirements might be challenging [20]. To communicate the reason why a design decision was made, manual documentation is needed because different problems can result in the same solution. To respond to changing or new

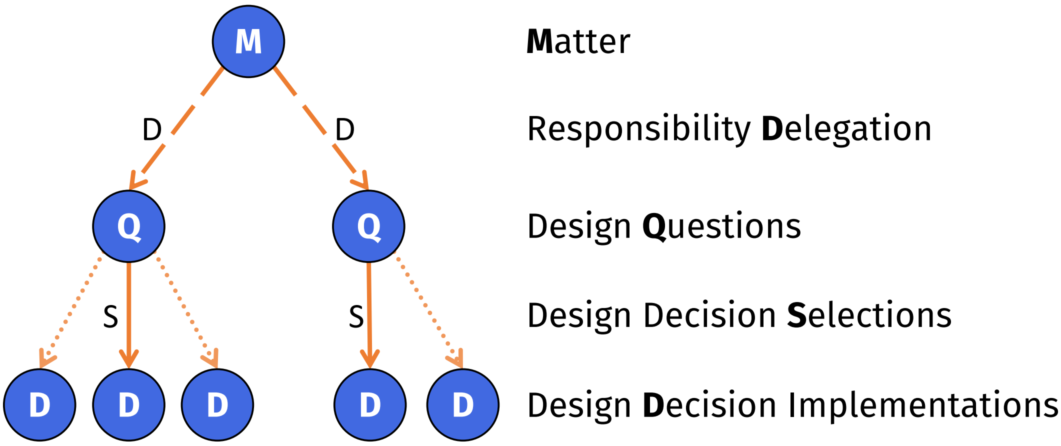


Figure 1 Structural overview of the approach. A *matter* is a concern that can delegate responsibilities to design questions. A *design question* is a pending design decision that are automatically selected based on desired non-functional properties. A *design decision* is a concrete option of answering a design question.

requirements, developers have to understand the reasons, why the current implementation was chosen and find an implementation, that serves the new requirements as well as the reverse engineered present requirements.

1.1 Approach: Design Decisions as first-class citizens

We propose Design-Decision-oriented Programming (DDOP), a new paradigm that automates the selection of the best-suited design decision by declaratively defining the required properties. A structural overview of the involved concepts is given in figure 1. The usage of DDOP might happen like this scenario: The application developer Alice implements a client-server system. She decides to use caching to decrease the server load and the response time of the system. Therefore, she just declares the requests as cacheable entities of the client and ranks response time as the most important NFP. She adds data confidentiality and memory consumption as medium-priority requirements. Hence, the optimal matching implementation is chosen out of a design decision library. Some time later, the caching expert Bob develops a new caching strategy that reduces the memory consumption without affecting performance. Thus, without Alice knowing about this new technique, her system uses the improved algorithm.

Therefore, DDOP makes developers think of design questions and design decisions as first-class citizens and it provides the infrastructure to offer reusable design decision implementations. The best-suiting decision is automatically selected according to the developer-defined properties of the present context.

1.2 General Applicability

DDOP is applicable, if all of these requirements are supported:

R1) **The design question provides a fixed, general contract that is fulfilled by each design decision implementation.** A design decision implementation that needs further configuration or more information about the context cannot be used instead of the other ones since it does not conform to the *Liskov Substitution Principle* [36, 37]. Therefore, the design question has to be as generic as possible but as simple as possible to be used by clients. This requirement enables a suitable potential for reuse of the design decision implementations.

R2) **Each design decision implementation can be modularized in one reusable module.** This requirement targets the feasibility of creating an implementation that can be used in a wider context. Criteria for choosing programming languages or language extensions that are suitable for archiving this requirement are discussed in section 4.1.

R3) **Tailoring the DDOP implementation to the context should be easy.** An implementation that cannot be tailored to a specific context limits its applicability. Therefore, the mechanisms to bind abstract roles of the matter to meta-objects (e.g., classes, methods, attributes) should support large variability (e.g., dealing with static and non-static meta-objects, access and ignore the sender / receiver of a call). Furthermore, the configuration of each design question should be easier then re-implementing its simplest design decision, because otherwise, developers have no direct incentive to use the DDOP implementation.

1.3 Contributions

This paper makes the following contributions:

1. We propose a new paradigm targeting implicit design decisions and those that have not been considered yet. By separating the definition of required NFPs from the selection of the best-suited implementation, this approach facilitates changing requirements by reducing the assumptions that can be made against concrete implementations.
2. Our prototypical DDOP implementation shows the feasibility and demonstrates the core concepts of DDOP by applying it to the caching domain. The case study use ObjectTeams/Java [23, 24, 25] to bind the abstract roles to domain classes.

1.4 Structure of the Paper

The remainder of the paper is structured as follows: At first, section 2 summarizes the aspects of modularity relevant to our approach. This lays the foundation on which the concepts of DDOP, described in section 3, build upon. Subsequently, section 4 gives details on how DDOP can be implemented in Object-oriented Programming (OOP). Possible practical applications are presented in section 5. Implementation examples of a case study on caching is given in section 6. The benefits and liabilities of DDOP are discussed in section 7. Then, section 8 compares DDOP with other approaches. Considerations of future work are given in section 9 and the main insights are concluded in section 10.

Furthermore, the whole source code of the presented DDOP framework can be found in appendix A. The source code of the caching case study is given in appendix B.

2 Background

This section introduces the basic concepts that constitute the rationale of how we propose to implement DDOP.

2.1 Modularity

Modularity principles are the foundation of decomposing a matter into smaller design decisions.

In 1972, David Parnas [45] argued, that decomposing systems into modules should be done according to difficult design decisions or design decisions that are likely to change. Thus, each module should hide one of these design decisions from the others. This concept, known as information hiding, supports *changeability* (changing requirements lead to changing only one module), *independent development* (modules can be implemented in parallel), and *comprehensibility* (developers are able to understand one module at a time). Some years later, he showed that this information hiding concept still works for complex, hard real-time systems by applying it to the Onboarding Flight Program of the A-7E aircraft [48].

The separation of conceptual behavior from concrete implementations is the main purpose of abstractions [36]. Encapsulation supports the change and exchange of implementations without the need for changing the dependents [36]. Hence, reducing the assumptions that modules make about each other is one important goal in software design [46]. This can be archived by *Design by Contract*, a principle for declaring obligations and benefits for connections between modules in fixed interfaces [42]. Similarly, the *Liskov Substitution Principle* requires that each object of a subclass should be substitutable for their base class objects. This means, that sub types should provide at least all post-conditions and require not more than the pre-conditions of their base types [37]. This principle is important for implementing many design decisions that conform to the same design question.

2.2 Aspect-oriented Programming

Aspect-oriented Programming (AOP) [31] supports the reuseability and extensibility of common software aspects by modularizing cross-cutting concerns. Therefore, it assists the implementation of reusable tailorable design decisions.

Cross-cutting Concerns (CCCs) are concerns that are architecturally orthogonal to each other. The structure of CCCs involves *scattering* (one concern is spread over multiple modules) and *tangling* (multiple concerns are interleaved in one module) [3, 56]. In the context of this paper, a concern is considered cross-cutting if it scatters and tangles the domain responsibilities of the application. Hence, example of typical CCCs are logging, resource management, security, and fault tolerance. Furthermore, collaborations (e.g., design patterns such as the Observer) can create superimposed roles (e.g., Subject and Observer roles) that crosscut the domain functionality of the involved classes [22, 24]. AOP is a common mechanism to modularize crosscutting concerns in *aspects* [4, 8, 21, 22, 31, 34]. Aspects are development-time programmatic units that are weaved into the base modules (e.g., domain classes). AOP can improve separation of concerns of a system containing CCCs and enables reuse of the modularized aspects [31]. Therefore, it supports implementing reusable design decisions.

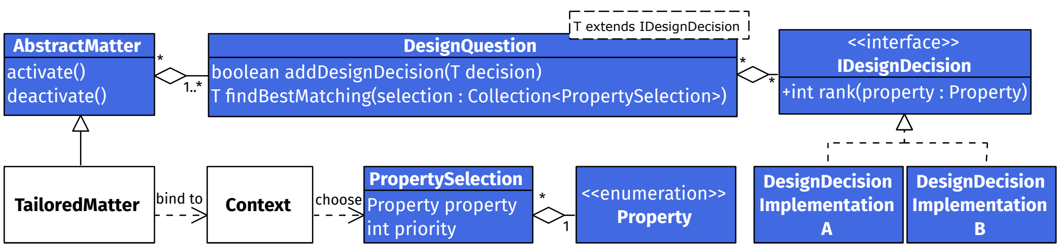
3 Design-Decision-oriented Programming

This section introduces the generic concepts of the DDOP paradigm on a language-independent level. An overview of the object-oriented reference implementation is shown in figure 2.

Formal Description Suppose, a system is strictly designed according to Parnas' information hiding principle [45]: Each design decision is encapsulated in one module and the interfaces of all modules M contain only the least possible details. Hence, each module can be exchanged with any other module that implements a design decision for the same design question q. Assuming, that q is very general and is raised in many different contexts C_q, then modules M_q that give different answers to q could be reused in all $c \in C_q$. Each module $m \in M_q$ has q-specific implications/properties. Developers performing DDOP decide which $m \in M_q$ to use just by specifying the desired properties $P_{\mu,c}$ for a matter μ in c. A design decision selection mechanism selects for each q the best-suiting $m \in M_q$ according to $P_{\mu,c}$.

3.1 Design Question

During software development, many design questions arise (e.g., which collection implementation to use for observers of a subject, which encryption algorithm to use for network communication, or which culling technique to use for 3D rendering).



■ **Figure 2** The structural overview of an object-oriented DDOP framework (blue) and its generic usage (white). Notation: UML 2.5 class diagram.

Each design question can be answered by a set of design decisions (e.g., all kinds of collection implementations, all kinds of encryption algorithms, or all kinds of culling techniques). However, to provide changeability, developers should not be able to assume a concrete implementation of a design decision. Instead, each design question can automatically select the best matching design decision according to a set of required properties (e.g., fast insert, quick decryption and high security, or highest rendering quality) with associated priority as defined in the source code. Thus, developers can explicitly weight the requirements, so that the design decision implementation that is optimal in this context will be selected.

> A *design question* is the programmatic representation of a pending generic design decision whose implementation can be assessed by a set of required prioritized properties.

The Dependency Inversion Principle ("Depend upon abstractions. Do not depend upon concretions") [39, 40] can in OOP be realized using interfaces. They serve as specifications of classes that agree on a common contract. Using interfaces ensures that developers assume only the least common properties (the *abstraction*) of all implementations by keeping the realization (the *concretions*) hidden. Similarly, design questions apply the Dependency Inversion Principle to design decisions by removing dependencies to concrete design decision implementations. Thus, design questions reduce design assumptions and therefore, facilitate changing requirements.

3.2 Design Parameter

Developers that want to use a reusable caching implementation have to decide whether the cache is kept in memory only or stored on a hard drive. This design decision is essential for the software, because it changes its external visible functionality. In contrast to design questions, these kind of pending design decisions have to be carried out by the developer. They are called design parameters.

> A *design parameter* is the programmatic representation of a pending generic design decision whose implementation is explicitly selected by the developer.

3.3 Matter

Since different design questions share common requirements, DDOP suggests to modularize them in a concern that abstracts their common intention. This simplifies binding many design questions to a concrete, complex context. For example caching involves design questions concerning the replacement policy, the memory management, and the cache creation. All of them should respond to the same set of requirements that configures the whole caching matter.

A *matter* is a concern that binds a semantically coherent set of design questions and design parameters to a concrete context.

A matter defines abstract superimposed roles all targeting one specific concern (e.g., caching, culling, encryption). The abstract roles are assigned to concrete classes inside the tailored matter that binds the concern to the concrete context.

An example of a tailored caching matter is shown in **??**.

Transparent Matter Some matters (e.g., caching) have no influence on the correctness of the resulting software. A browser with caching has be same behavior as without. Thus, the caching matter can be considered transparent.

A *transparent matter* is a matter, that is not required or referenced by other modules, and thats presence or absence influences NFPs only.

Hence, a transparent matter provides a binary configuration of a software system on a higher level of abstraction than a design question. The activation or deactivation is a decision that is manually made by the developer

Transparent matters can be used to enlarge the variety of NFPs of Software Product Lines (SPLs). Furthermore, if their implementation allows to activate and deactivate them even during run-time, transparent matters can be used to dynamically respond to changing environments. Thus, transparent matters can support the implementation of self-adaptive systems.

3.4 Design Decision Implementation

A design question can be answered by many different design decisions. By giving an answer to the design question, a design decision implementation offers NFPs across various dimensions (e.g., performance, memory consumption, security). An implementation provides some of these properties to a higher degree than other properties. Hence a property rating function is used to assess the design decision's suitability for a concrete context.

A *design decision implementation* is a concrete answer to a corresponding design question. It characterizes its implications as function that describes the degree of achievement for each property.

A design decision implements the interface of the corresponding abstract design decision associated with the design question to answer. By giving an answer to the design question, it offers NFPs that can be requested in order to assess the suitability of the design decision according to a concrete context.

3.5 Property

> A *property* is one dimension for characterizing a design decision implementation.

Examples are performance, memory consumption, security, or domain-specific properties such as the softness of a shadowing algorithm. The raking of properties for design decision implementations must be objective. In the simplest form, this is done by the design decision developer, future work discusses automated property assignment.

3.6 Design Decision Selection

The selection of best matching design decision $d^* \in D_q$ for a design question q happens by ranking the design decision implementations D_q according to a set of property selections S. The rank $r(d,S)$ of a design decision $d \in D_q$ is computed using this formula:

$$r(d,S) := \sum_{s \in S} (d.rank(s.property) * s.priority)$$

Thus, d^* is the design decision implementation with the maximum $r(d^*,S)$.

4 Object-oriented Implementation

This section describes how DDOP can be implemented in object-oriented languages. A structural overview of the concepts is shown in figure 2

Matters use the strategy design pattern [17] to delegate responsibilities to design decisions which are instantiated using a design question. Each design questions holds a collection of concrete design decision implementations that answer this design question. To ensure type safety, it is parameterized by the a sub type of the design decision interface that must be implemented by each of the corresponding design decisions. Thereby, a design decision can answer many design questions by implementing their corresponding design decision interface.

4.1 Suitable Programming Languages

As stated in R2, DDOP demands programming languages that enable developers to modularize each design decision in one module. Many of the collaborative design decisions involve crosscutting concerns. These can be modularized using AOP [31], for example AspectJ, AspectS or AspectC. We decided to use ObjectTeams [23, 24, 25] for our implementation, because it provides advanced configuration of reusable collaborations.

4.2 Property-based Design Decision Selection

In order to select one design decision per design question, the developer should be able to declaratively define properties, the demanded design decision should have.

Design decision selection during *start-up-time* increases the flexibility, because the library of design decisions can be deployed separately. However, a run-time error occurs if no suitable design decision implementation was found. This delays the feedback of the software correctness.

Design decision selection during *compile-time* statically binds the implementations. It ensures the correctness of the software and can provide shorter start-up time. But the implementation requires a modified compiler. Therefore, our prototypical implementation uses the simpler start-up-time design decision selection. The compile-time version is left as future work.

5 Applications

In general, DDOP applies to design decisions that differ in the external quality attributes (e.g., performance, security, fault tolerance) only and that answer a reoccurring design question. It is well suited for domains that are well understood. If the community agrees on common interfaces and unified concepts, it is more easy to establish design question interfaces with promising future. New technologies with ongoing controversial discussions and a polarized community might have problems with agreeing on common standard implementations.

Furthermore, DDOP is not suited for domains which require a proof of correctness of the resulting system, because the concrete design decision implementations are unknown. Operating system kernels might not benefit from DDOP, since the complexity introduced by DDOP makes reasoning about the system difficult. Run-time binding of design decisions is not suited for hard real-time systems, because it introduces a new unknown dependency for irregularity.

However, there are many applications that are well suited for DDOP. Some examples are given in the following.

Data Types The question whether to use a floating point number, fixed point number, (non-negative) integer, fraction or linked list representation for extremely large, and precise numbers depends on the intended usage of the number. Developers consider criteria such as rounding errors, memory consumption, domain range, hardware support as well as compatibility with other modules. Using DDOP, developers can tailor the design decisions according to the requirements by declaring which parts of the system should be optimized for which NFP. The same concept applies to collections, trees, graph representations, and spatial data structures.

Resource Management Design decisions in resource managements strongly influence the NFPs of a system. Furthermore, they are crosscutting concerns and a well-understood techniques [32]. The following examples are potential design questions for which DDOP applies well: Whether the access to a resource should be cached is one of the most common decisions in resource management. Caching itself involves many common design decisions, e.g., how many entries should be cached? Which replacement policy should be used? Does the capacity scale according to the number of cache misses or should it be static? Which strategy should be used to decide which entry should be removed if the capacity is reached? A case study of a DDOP implementation of caching can be found in figure 6.

Security Often, developers use security frameworks incorrectly, because the usage of them is hard for non-experts of the security domain [28]. However, since security issues have been modularized in aspects [27, 43, 55], they can be used as design decisions in DDOP by assigning them fine granular properties of the security domain. Thereby, developers would not require extensive knowledge of the security domain to create software that facilitates a medium level of security. They would just rank security as important NFP in the critical components.

Design Patterns Design patterns discuss sets of reusable solutions for reoccurring design questions [17]. Therefore, they are good candidates to be general design questions.

The implementation of a design pattern often includes substantial boiler-plate code that is scattered over multiple classes [22, 53]. Furthermore the developer can make mistakes while implementing a design pattern (e.g., forgotten synchronization in a threaded-safe Singleton implementation or ignoring object equality in a Decorator implementation).

Noureddine and Rajan [44] proposed design pattern implementations that are optimized for energy consumption. Dougherty et al. [13] discuss secure implementations of Factory, Strategy Factory, Builder Factory, Chain of Responsibility, State Machine, and Visitor. Fernandez-Buglioni [15] proposed secure implementations of Broker, Pipes and Filters, Blackboard, Adapter, Model-View-Controller and other security patterns.

Using DDOP, the usage of design patterns involves exposed design question (such as object equality in Decorators) decided by developers and hidden design questions that are automatically decided based on the selection of required properties. The exposed design questions force the developers to think about them while the hidden design questions offer flexibility and ease of use.

Computer Graphics Systems In 3D rendering systems, culling techniques such as backface culling [60], view frustum culling [2], occlusion culling [10] and detail culling [35] can be used to optimize performance of the scenes. Different shadowing approaches such as shadow mapping [12, 58] are suited for real-time soft shadows while shadow volumes [11] provide pixel-exact shadows. Because new shadowing improvements are continuous invented, usual graphics developers might not have an overview of the current implementations and their trade-offs. Analogously, techniques for bump mapping[1] (e.g., Blinns method [5], relief mapping [50], and displacement mapping [57]), culling and other graphical design decisions are continuously improved. Using DDOP, computer graphics researchers could add their improved implementations to the design decision library as supplemental material of their papers. Hence developers would instantly get the optimal implementation for their context without inquiry of recent techniques.

6 Case Study

Caching [32] is a transparent matter that involves multiple design decisions (e.g., which replacement policy to use, which data structure to use for the cache, how to determine the capacity of the cache) and design parameters (e.g., which resources to cache, which requests to cache). It can increase the performance of an application while increasing the consumption of memory or storage.

Creation of a reusable caching library The a structural overview of our DDOP caching implementation can be found in figure 3. When the capacity of the cache is reached and one element should be added to the cache, the implementation has to decide which element to remove from the cache. Therefore, we created an interface for replacement policies and implemented First In First Out (FIFO) as well as Least Recently Used (LRU). The abstract caching matter delegates the request for replacing

[1] Increasing geometrical details using textures

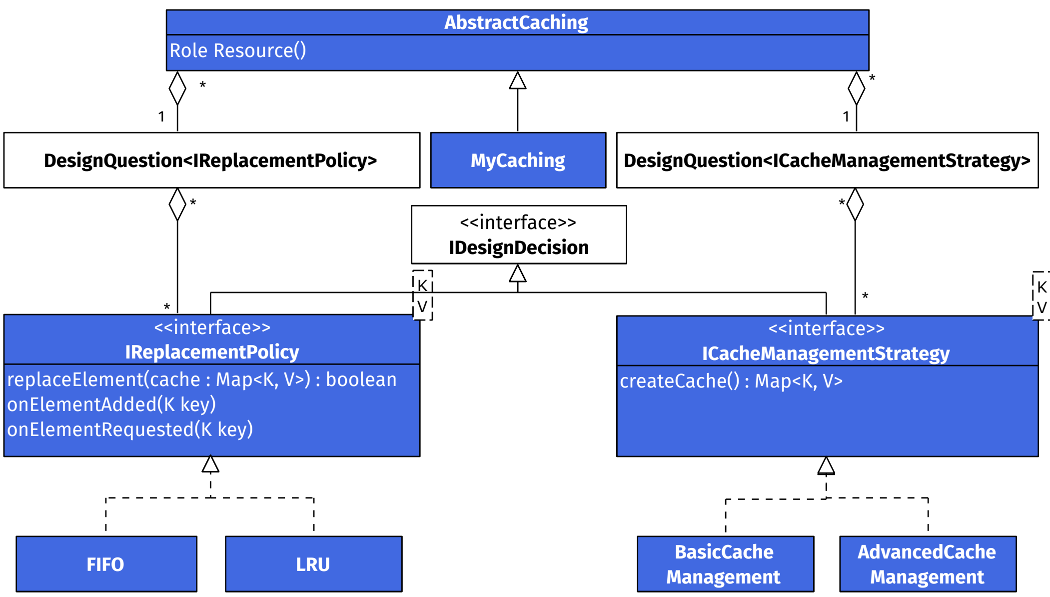


Figure 3 Structural view of a DDOP implementation of caching (blue) and the DDOP framework (white). Notation: UML 2.5 class diagram.

an element to its replacement policies (see listing 1). Similarly, the cache creation it delegated to the cache management strategy design decision.

Listing 1 Usage of the replacement policy inside the abstrace caching class. (The complete class can be found in listing 6 in appendix B).

```
public abstract team class AbstractCaching {
    [...]
    protected boolean replaceElement() {
        return mReplacementPolicy.replaceElement(getCache());
    }
}
```

Usage of the caching library The integration of caching into an application is limited to binding the abstract caching matter to the contextual domain classes using subclassing (listing 2). The configuration of the design parameters happens by assigning the abstract roles to concrete meta objects.

Listing 2 Subclassing of the abstract matter AbstractCaching. The abstract role ResourceRole is assigned to the concrete class MyResource using the playedBy relation of ObjectTeams. The requestResource method is bound to the getElement method of MyResource using a callin.

```
public team class MyCaching extends AbstractCaching {
    [...]
    public class ResourceRole playedBy MyResource {
        String requestResource(String id) <— replace String getElement(String id);
    }
}
```

To ensure, that a property selection is made, it is passed as argument to the constructor of the matter (listing 3). This explicit statement of the desired NFPs ensures traceability of the requirements.

Furthermore, when requirements change (e.g., security becomes more important), the locality of code changes is limited to the property selection. Thereby, changeability is facilitated.

Listing 3 Property selection.

```
final Collection<DesignQuestion.PropertySelection> selection = new ArrayList<>(2);
selection.add(new DesignQuestion.PropertySelection(10, IDesignDecision.Property.
    ↪ PERFORMANCE));
selection.add(new DesignQuestion.PropertySelection(2, IDesignDecision.Property.SECURITY));

MyCaching caching = new MyCaching(selection);
caching.activate();
```

7 Discussion

DDOP can have the following advantages:

(A1) Popularization of Programming DDOP makes it easier to develop complex systems by reducing the required knowledge about the design decision implementations. Therefore, it popularizes programming by enabling less skilled people to create high-quality software.

(A2) Forward-Compatibility Developers need not to know the recent technologies. If they are added to the database connected to the design question, they will automatically be selected without any action of the developer.

(A3) Tailored Solutions DDOP simplifies making design decisions by explicitly offering all alternatives. By specifying the required properties of the solution, it causes developers to think about the non-functional requirements in order to improve the overall quality of the software.

(A4) Traceability The resulting source code communicates the traceability of the main design decisions, because they are explicitly part of the code and do not require additional documentation.

(A5) Changeablity Design decisions are easy to change. Therefore, iteration cycles in finding optimal solutions are shortened and changing requirements do not result in changing a lot of code.

(A6) Avoiding Reinvention By offering an infrastructure to reuse common design decision, DDOP causes developer to use existing implementations instead of re-implementing them.

(A7) Reduced Error-proneness Since developers use tested standard solutions, they avoid bugs.

(A8) Reusability Each user can contribute to the design decision variability, because each adoption of standard implementations can become part of the library.

However, DDOP can have the following liabilities or challenges:

(L1) Thoughtlessness Developers might use implementations they do not know. This might reduce the understanding of the internal behavior.

(L2) Code complexity DDOP requires advanced modularization techniques which might call for further programming education of developers, like all new programming paradigms.

(L3) Execution variations The observable properties of the resulting system might differ, since newer versions in the design decision library can result in different design decisions selections. In most cases this is the intended behavior of DDOP. However, it might result in unexpected changes of the system, if the interface of the design decisions are not designed carefully. For example, if two design decision implementations use a different persistence format, a data has to be migrated when switching between these decisions.

(L4) Performance The execution time for the design decision selection scales with the number of design decision implementations. Hence, design decision selection during load-time or run-time can slow down the execution of the program.

While the selected design decision influences external quality attributes of the client system, the internal architectural properties of the client (e.g., modularity) are not modified.

8 Related Work

8.1 Concern-Oriented Software Design

The modeling technique *Concern-Oriented Software Design (COSD)* [1] targets a similar vision by focusing on concerns as units of reuse.

In contrast to COSD, developers performing DDOP do not explicitly select one concrete design decision implementation for each design question. So, while COSD helps developers during the decision making process, DDOP helps developers to express the intention behind the decision and enforces late binding of the concrete implementation. Therefore, COSD provides static binding which results in less execution variations (L3). DDOP enforces information hiding since developers never know which implementation will be chosen to support changeability (A5). COSD enables developers to build on the decision that was made and might introduce assumptions which simplify the software but decrease the changeability. Furthermore, DDOP provides tractability (A4) of design decisions by direct manifestation of the requirements in the source code.

In conclusion, COSD targets ease of software construction only, while DDOP supports flexible evolution of the design decision library.

8.2 Other Paradigms

DDOP is orthogonal to many other programming paradigms.

Declarative programming Declarative programming enables developers to define the required outcome without specifying how to get there. This leads to abstract programs which decouple problems from implementations. Therefore, the implementations can be reused in a wider context. Similar to this concept, DDOP decouples design questions from design decision implementations by declaratively defining the required properties. While DDOP uses some declarative concepts, it should not be considered a sub-concept of declarative programming, because the remaining parts of a DDOP program might be implemented in imperative programming languages.

Object-oriented programmng Our example implementation uses OOP, because it fits most of the contemporary software. However, DDOP programs might be written without objects or classes.

Aspect-oriented programming AOP is one mechanism to modularize design decisions and weave them into the main program. Therefore, DDOP implementations might make extensive use of AOP concepts. However, AOP is not essential for DDOP, since future modularization techniques might be used to separate the crosscutting concerns.

8.3 Non-functional Property Selection

SOA In the field of Service Oriented Architecture (SOA) approaches for selecting services based on NFPs have been developed [51].

EJB and CORBA Enterprise Java Beans (EJB) and the Common Object Request Broker Architecture (CORBA) are component frameworks that allow late binding of components. Göbel et al. [19] proposed a component model that enables to separately specify the non-functional properties and the functional properties of required components.

8.4 Reuse of Design Pattern Implementations

Since the publication of design patterns [17], many researchers tried to find a way to reuse their implementations. One approach to modularize design patterns is AOP [6, 21, 34]. *Hannemann and Kiczales* [22] describe design pattern implementations in AspectJ [30]. Some of their implementations (e.g., Observer, Composite, Iterator) could be reused, but they do not provide the ability of specialization or configuration of the design pattern implementations. Injecting design patterns into classes at run-time can be done by expressing design decisions as applications of design operators to be carried out at run-time [26].

Automatic code generation [7, 14, 16, 38] for design patterns simplifies the implementation and allows refining, but produces more code that has to be maintained. This can lead to the round-trip problem [9]: modified code will be overriden by regeneration. In contrast, our concept uses the weaving of aspect-oriented programming [31] in order to keep the pattern code out of the domain code.

The concept of ObjectTeams [23, 24, 25] facilitates refinable reusable implementations of multi-object collaborations. It modularizes crosscutting collaborations by introducing a "team" as new refinable first-class language construct. DDOP enables to reuse multiple design pattern implementations that support a set of desired NFPs.

8.5 Call by Meaning

Call by Meaning [52] is a vision aiming for calling methods by semantic specifications instead of names in order to simplify finding a solution matching a requirement.

In contrast, DDOP targets mainly NFPs rather than functional requirements.

8.6 End-User Architecting

End-User Architecting [18] is a concept that enables users to compose functionality resulting in a new program. It targets people who create and execute programs in support of their professional goals, but not as their primary job function. It offers a graphical user interface with which the end user can connect base components from a functional view. Similar to DDOP, End-User Architecting enables the client to decide In contrast, DDOP targets professional programmers by offering a design view. Furthermore, the focus of DDOP lays on selecting one solution out of a set of alternatives whereas End-User Architecting does not necessarily provide multiple modules that serve the same purpose in different ways.

9 Future Work

DDOP is a new paradigm. Hence, there are many questions that have not been answered in this paper.

Linting implicit Design Questions The proposed paradigm causes developers to think about the trade-off that have to be made during implementation of common design questions. However, it does not raise the design question. Future work targeting this challenge might develop techniques for suggesting developers to think of the design questions that are implicitly answered using classical implementations. This might include hints for refactoring to DDOP.

Refactoring to DDOP Tool support for refactorings that transform implicit design decisions to DDOP implementations would lower the barrier to use DDOP in legacy projects or to create design decision specifications for existing design decision implementations.

Compile-time Design Decision Selection In order to ensure the correctness of the resulting software, the ability to map the selected properties to design decision implementation during compile-time is required. This gives developers early feedback, if no design decision implementation matches the requirements.

Software Product Lines (SPLs) One challenge in SPL engineering is to steer the NFPs of the resulting products (e.g., to optimize performance or energy consumption) [54]. We believe, that a dynamic DDOP approach simplifies this by changing the selection criteria per concrete product. However, this claim needs to be evaluated.

Coarse-grained Property Selection The proposed implementation enforces the developer to select a set of NFPs for each design question. In order to provide a more abstract and coarse-grained property selection, developers should be able to define them for larger modules. In a preprocessing step, the framework would recursively add them to the existing selected properties for each submodule or contained design question.

Automated Property Assignment In the proposed realization the properties of a design decision implementation need to be added by the developer as meta data of the implementation. This limits the application of the design decision implementation, because its development has to think about all possible advantages and liabilities of the implementation. In order to scale this approach, the assignment of properties could be done on demand according to the requested properties of the client. The environment automatically generates quality tests assessing the possible implementations, executes the tests on real production data and selects the best-fitting implementation.

10 Conclusion

Design-Decision-oriented Programming (DDOP) has been proposed as a new programming paradigm which shifts the focus from implementing code to making traceable design decisions. It separates the explicit definition of required properties of a design question from the implementations of the design decisions. Therefore, DDOP aims for simplifying the development of complex systems by hiding the concrete implementations of the required properties.

Basic DDOP implementations can be done in ObjectTeam/Java without custom language constructs, as shown in the prototypical implementation.

We have shown that DDOP applies for a wide range of domains and therefore, can be considered as a general-purpose paradigm.

Acknowledgements I thank Alexander Riese for supporting the implementation of a previous version of this work and Patrick Rein for giving feedback on the paper.

A Source Code of the DDOP Framework

A.1 Design Question Implementation

Listing 4 The design decision interface

```java
package aspect;

import java.util.Arrays;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;

/**
 * Design question that should be automatic solved by picking the best matching {@link
     ↪ IDesignDecision}
 * for given property preferences.
 *
 * @param <T> Design decisions that can solve this design question.
 */
public class DesignQuestion<T extends IDesignDecision> {

  /**
```

```
 * Class to store the prioritized property preferences.
 */
public static class PropertySelection{
  /** The desired property. */
  public IDesignDecision.Property mProperty;

  /** The priority of the desired property. */
  public int mPriority;

  /**
   * Pairs a property with a priority.
   *
   * @param priority Indicates how important the given property is.
   * @param property The property that should be valued.
   */
  public PropertySelection(int priority, IDesignDecision.Property property){
    mProperty = property;
    mPriority = priority;
  }
}

/** Collection of {@link IDesignDecision} that can solve this design question */
private List<T> mDesignDecisions;

/**
 * Constructor to instantiate a design question with {@link IDesignDecision}.
 *
 * @param designDecisions Design decisions to solve this design question.
 */
public DesignQuestion(final T... designDecisions){
  mDesignDecisions = Arrays.asList(designDecisions);
}

/**
 * Constructor to instantiate a design question without {@link IDesignDecision}.
 */
public DesignQuestion(){
  mDesignDecisions = new LinkedList<>();
}

/**
 * Adds a {@link IDesignDecision} to this design question.
 *
 * @param designDecision Design decision that should be added.
 * @return True if the operation was successful, false otherwise.
 */
public boolean addDesignDecision(final T designDecision) {
  if(!mDesignDecisions.contains(designDecision)) {
    mDesignDecisions.add(designDecision);
    return true;
  }
  return false;
}
```

```java

    /**
     * Rate a single design decision in context to given property preferences.
     *
     * @param designDecision Design decision that should be rated.
     * @param properties Property preferences used to rate the design decision.
     * @return Rating value how well the design decision fits to the property preferences.
     */
    private static int rateProperties(IDesignDecision designDecision, Collection<PropertySelection
        ↪ > properties){
      int result = 0;
      for(PropertySelection property: properties){
        result += designDecision.rankProperty(property.mProperty) * property.mPriority;
      }
      return result;
    }

    /**
     * Returns the most suitable {@link IDesignDecision} for a given property preference.
     *
     * @param selection Property preferences used to rate the design decisions.
     * @return The most suitable {@link IDesignDecision} for the given property preference.
     */
    public T findBest(final Collection<PropertySelection> selection){
      //In Java 9: mDDs.stream().max(Comparator.comparingInt((IDesignDecision decision) —>
          ↪ rankProperties(decision, d))).ifPresent(IDesignDecision::activate);

      if(mDesignDecisions.isEmpty()) {
        return null;
      }

      T maximum = mDesignDecisions.get(0);
      for(T decision: mDesignDecisions) {
        if(rateProperties(decision, selection) > rateProperties(maximum, selection)){
          maximum = decision;
        }
      }
      return maximum;
    }
}
```

A.2 Design Decision Interface

Listing 5 The design decision interface

```java
package aspect;

/**
 * A specific design decision to answer a {@link DesignQuestion}.
 */
public interface IDesignDecision{

```

```
  /** Properties that can be used to assess design decisions. */
  public enum Property{
    PERFORMANCE,
    SECURITY
  }

  /**
   * Returns the satisfaction of the design decision for a given property.
   *
   * @param property Property that should be ranked.
   * @return The rank of the property.
   */
  int rankProperty(final Property property);
}
```

B Source Code of the Caching Case Study

B.1 Matter: Abstract Caching

Listing 6 The implementation of the abstract caching matter

```
package caching;

import java.util.Collection;
import java.util.Map;

import aspect.DesignQuestion;
import aspect.DesignQuestion.PropertySelection;
import caching.strategies.BasicCacheManagement;
import caching.strategies.FIFO;
import caching.strategies.ICacheManagementStrategy;
import caching.strategies.IReplacementPolicy;
import caching.strategies.LRU;

/**
 * Abstract team class to bundle caching behavior.
 */
public abstract team class AbstractCaching{

  /** Strategy which defines how the cache get managed */
  private ICacheManagementStrategy<String, String> mCacheManagement;

  /** Strategy which defines when and how elements get replaced */
  private IReplacementPolicy<String, String> mReplacementPolicy;

  /** Constructor which instantiate the most suitable caching strategies for a given property
      ↪ preference. */
  public AbstractCaching(final Collection<PropertySelection> selection) {
    mCacheManagement = getCacheManagementStrategy(selection);
    mReplacementPolicy = getReplacementStrategy(selection);
  }
```

```

/**
 * Accessor for the cache management strategy.
 * @return The current cache management strategy.
 */
  public ICacheManagementStrategy<String, String> getCacheManagement() {
  return mCacheManagement;
}

  /**
 * Accessor for the cache management strategy.
 * @param cacheManagement The new cache management strategy.
 */
public void setCacheManagement(ICacheManagementStrategy<String, String>
    ↪ cacheManagement) {
  mCacheManagement = cacheManagement;
}

/**
 * Accessor for the cache replacement policy.
 * @return The current cache replacement policy.
 */
public IReplacementPolicy<String, String> getReplacementPolicy() {
  return mReplacementPolicy;
}

  /**
 * Accessor for the cache replacement policy.
 * @param cacheManagement The new cache replacement policy.
 */
public void setStrategy(IReplacementPolicy<String, String> policy) {
  mReplacementPolicy = policy;
}

/**
 * Abstract role class for the resources that should be cached.
 */
public abstract class ResourceRole{

  /** Cache that is in use. */
    private Map<String, String> mCache;

    /** Number of elements currently stored in the cache. */
    private int mElementCount = 0;

    /** Number of cache misses. */
    private int mCacheMissesCount = 0;

    /** Number of cache hits. */
    private int mCacheHitsCount = 0;

    /**
     * Accessor for the number of elements.
```

```
     * @return the number of elements in the cache.
     */
    public int getElementCount() {
        return mElementCount;
    }

    /**
     * Accessor for the number of elements.
     * @param elementCount the number of elements in the cache.
     */
    public void setElementCount(int elementCount) {
        mElementCount = elementCount;
    }

    /**
     * Accessor for the number of cache misses.
     * @return The current number of cache misses.
     */
    public int getCacheMissesCount() {
        return mCacheMissesCount;
    }

    /**
     * Accessor for the number of cache misses.
     * @param cacheMissesCount The new number of cache misses.
     */
    public void setCacheMissesCount(int cacheMissesCount) {
        mCacheMissesCount = cacheMissesCount;
    }

    /**
     * Accessor for the number of cache hits.
     * @return The current number of cache hits.
     */
    public int getCacheHitsCount() {
        return mCacheHitsCount;
    }

    /**
     * Accessor for the number of cache hits.
     * @param cacheHitsCount The new number of cache hits.
     */
    public void setCacheHitsCount(int cacheHitsCount) {
        mCacheHitsCount = cacheHitsCount;
    }

    /**
     * Accessor for the cache.
     * @return The current cache.
     */
    public Map<String, String> getCache() {
        return mCache;
    }
```

```

    /**
     * Setup of the cache. Should be called before using it.
     */
    protected void setup() {
    mCache = mCacheManagement.createCache();
    mReplacementPolicy = getReplacementPolicy();
    mCache = mReplacementPolicy.optimizeCache(mCache, mCacheManagement.getCapacity());
    }

    /**
     * Adds an element to the cache.
     * @param id ID to access the element inside the cache.
     * @param value Value that should be stored inside the cache.
     * @return True if the operation was successful, false otherwise.
     */
    protected boolean addElement(String id, String value) {
    if(mElementCount < mCacheManagement.getCapacity() || replaceElement()){
      mCache.put(id, value);
      mElementCount++;
      mReplacementPolicy.onElementAdded(id);
      return true;
    }
    return false;
  }

    /**
     * Replaces an element to free space.
     * @return True if the operation was successful, false otherwise.
     */
  protected boolean replaceElement() {
    System.out.print(getCacheMissesCount() + " ");
    return mReplacementPolicy.replaceElement(getCache());
  }

  /**
   * Clears the cache and resets the counter.
   */
  protected void clearCache() {
    mCache.clear();
    mCacheHitsCount = 0;
    mCacheMissesCount = 0;
  }

  /**
   * Returns the resource for a given ID.
   * @param id ID of the requested resource.
   * @return Resource for the given ID.
   */
  @SuppressWarnings("basecall")
  callin String requestResource(final String id){
    mReplacementPolicy.onElementRequested(id);
```

```java
      if(mCache.containsKey(id)){
        mCacheHitsCount++;
        return mCache.get(id);
      } else {
        mCacheMissesCount++;
        final String result = base.requestResource(id);
        addElement(id, result);
        return result;
      }
    }
  }

 /**
  * Returns most suited {@link ICacheManagementStrategy} for a given property preference.
  *
  * @param selection Property preferences used find the most suitable design decisions.
  * @return Best fitting cache management strategy.
  */
  protected ICacheManagementStrategy<String, String> getCacheManagementStrategy(final Collection<PropertySelection> selection) {
    DesignQuestion<ICacheManagementStrategy<String, String>> cacheManagementQuestion = new DesignQuestion<>();
  cacheManagementQuestion.addDesignDecision(new BasicCacheManagement<>());

  return cacheManagementQuestion.findBest(selection);
  }

  /**
  * Returns most suited {@link IReplacementPolicy} for a given property preference.
  *
  * @param selection Property preferences used find the most suitable design decisions.
  * @return Best fitting replacement policy.
  */
  protected IReplacementPolicy<String, String> getReplacementStrategy(final Collection<PropertySelection> selection) {
    DesignQuestion<IReplacementPolicy<String, String>> replacementDesignQuestion = new DesignQuestion<>();
  replacementDesignQuestion.addDesignDecision(new FIFO<>());
  replacementDesignQuestion.addDesignDecision(new LRU<>());
  return replacementDesignQuestion.findBest(selection);
  }
}
```

B.2 Tailored Matter: MyCaching

Listing 7 The implementation of a tailored caching matter

```java
package caching;

import java.util.Collection;

import aspect.DesignQuestion.PropertySelection;
```

```

import base core.Resource;

/**
 * Basic team class to realize {@link AbstractCaching}.
 */
public team class MyCaching extends AbstractCaching {

  /** Constructor which instantiate the most suitable caching strategies for a given property
       ↪ preference. */
    public MyCaching(Collection<PropertySelection> selection) {
        super(selection);
  }

  /**
   * Binds the {@link Resource} with the {@link ResourceRole}.
   */
  public class ResourceRole playedBy Resource{
          String requestResource(String id) <— replace String get(String id);
          void setup() <— after Resource();
    }
}
```

B.3 Design Decision: Replacement Policy

Listing 8 The interface for replacement policies

```
package caching.strategies;

import java.util.Map;

import aspect.IDesignDecision;

/**
 * Strategy to define how and when elements in the cache should be replaced.
 *
 * @param <K> Keys that can be used to access the resources.
 * @param <V> Values that can be saved.
 */
public interface IReplacementPolicy<K, V> extends IDesignDecision {

  /**
   * Replace a element in the given cache.
   * @param cache Cache where an element should be replaced.
   * @return True if the operation was successful, false otherwise.
   */
  boolean replaceElement(final Map<K, V> cache);

  /**
   * Method that is called when an element is added.
   * @param key Key of the added element.
   */
```

```
  void onElementAdded(K key);

  /**
   * Method that is called when an element is requested.
   * @param key Key of the requested element.
   */
  void onElementRequested(K key);

  /**
   * Optimizes the cache for this replacement policy.
   * @param cache Cache that should be optimized.
   * @param capacity Capacity of the cache that should be optimized.
   * @return Optimized cache for this replacement policy.
   */
  Map<K, V> optimizeCache(Map<K, V> cache, int capacity);
}
```

Listing 9 The implementation of the first in first out replacement policy

```
package caching.strategies;

import java.util.Map;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;

public class FIFO<K, V> implements IReplacementPolicy<K, V> {

  /** Queue that stores the element to easily add new elements and remove the oldest one. */
  private Queue<K> mElements = new LinkedBlockingQueue<>();

  /**
   * Optimizes the cache for this replacement policy.
   * @param cache Cache that should be optimized.
   * @param capacity Capacity of the cache that should be optimized.
   * @return Optimized cache for this replacement policy.
   */
  @Override
  public Map<K, V> optimizeCache(Map<K, V> cache, int capacity) {
    return cache;
  }

  /**
   * Replace a element in the given cache.
   * @param cache Cache where an element should be replaced.
   * @return True if the operation was successful, false otherwise.
   */
  @Override
  public boolean replaceElement(final Map<K, V> cache) {
    try{
      final K displacedElement = mElements.remove();
      cache.remove(displacedElement);
      return true;
    } catch (Exception e) {
      return false;
```

```
    }
  }

  /**
   * Method that is called when an element is added.
   * @param key Key of the added element.
   */
  @Override
  public void onElementAdded(K key) {
    mElements.add(key);
  }

  /**
   * Method that is called when an element is requested.
   * @param key Key of the requested element.
   */
  @Override
  public void onElementRequested(K key) {
    //Do nothing.
  }

  /**
   * Returns the satisfaction of the design decision for a given property.
   *
   * @param property Property that should be ranked.
   * @return The rank of the property.
   */
  @Override
  public int rankProperty(Property property) {
    switch (property) {
      case SECURITY: return −1;
      case PERFORMANCE: return 2;
      default: return 0;
    }
  }
}
```

Listing 10 The implementation of the lest recently used replacement policys

```
package caching.strategies;


import java.util.LinkedHashMap;
import java.util.Map;

public class LRU<K, V> implements IReplacementPolicy<K, V> {

    /**
     * Optimizes the cache for this replacement policy.
     * @param cache Cache that should be optimized.
     * @param capacity Capacity of the cache that should be optimized.
     * @return Optimized cache for this replacement policy.
     */
    @Override
```

```java
    public Map<K, V> optimizeCache(Map<K, V> cache, int capacity) {
        int cacheSize = capacity;
        float loadFactor = 0.75F;
        boolean accessOrder = true;
        return new LinkedHashMap<K, V>(cacheSize, loadFactor, accessOrder){

            @Override
            protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
                return size() > cacheSize;
            }
        };
    }

    /**
     * Replace a element in the given cache.
     * @param cache Cache where an element should be replaced.
     * @return True if the operation was successful, false otherwise.
     */
    @Override
    public boolean replaceElement(final Map<K, V> cache) {
        return true;
    }

    /**
     * Method that is called when an element is added.
     * @param key Key of the added element.
     */
    @Override
    public void onElementAdded(final K key) {
        // Do nothing
    }

    /**
     * Method that is called when an element is requested.
     * @param key Key of the requested element.
     */
    @Override
    public void onElementRequested(final K key) {
        // Do nothing
    }

    /**
     * Returns the satisfaction of the design decision for a given property.
     *
     * @param property Property that should be ranked.
     * @return The rank of the property.
     */
    @Override
    public int rankProperty(Property property) {
        switch (property) {
            case SECURITY: return -1;
            case PERFORMANCE: return 3;
            default: return 0;
```

```
        }
    }
}
```

B.4 Design Decision: Cache Management

Listing 11 The interface for cache management strategies

```java
package caching.strategies;

import java.util.Map;

import aspect.IDesignDecision;

/**
 * Strategy to define how the cache should be managed.
 *
 * @param <K> Keys that can be used to access the resources.
 * @param <V> Values that can be saved.
 */
public interface ICacheManagementStrategy<K, V> extends IDesignDecision {

    /**
     * Returns the number of elements that can be stored.
     * @return the current capacity of the cache.
     */
    int getCapacity();

    /**
     * Creates the cache.
     * @return The created cache data structure.
     */
    Map<K, V> createCache();

}
```

Listing 12 The implementation of a simple cache management strategy

```java
package caching.strategies;

import java.util.HashMap;
import java.util.Map;


/**
 * Basic realization of {@link ICacheManagementStrategy}.
 *
 * @param <K> Keys that can be used to access the resources.
 * @param <V> Values that can be saved.
 */
public class BasicCacheManagement<K, V> implements ICacheManagementStrategy<K, V> {

```

```
  /**
   * Returns the satisfaction of the design decision for a given property.
   *
   * @param property Property that should be ranked.
   * @return The rank of the property.
   */
  @Override
  public int rankProperty(Property property) {
    return 0;
  }

  /**
   * Returns the number of elements that can be stored.
   * @return the current capacity of the cache.
   */
  @Override
  public int getCapacity() {
    return 10; //constant, might scale with the number of cache misses or request the free
        ↪ memory.
  }

  /**
   * Creates the cache.
   * @return The created cache data structure.
   */
  @Override
  public Map<K, V> createCache() {
    return new HashMap<K, V>(getCapacity());
  }
}
```

References

[1] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. "Concern-Oriented Software Design". In: *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems*. MODELS'13. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pages 604–621. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_37.

[2] Ulf Assarsson and Tomas Moller. "Optimized View Frustum Culling Algorithms for Bounding Boxes". In: *Journal of Graphics Tools* 5.1 (2000), pages 9–22. DOI: 10.1080/10867651.2000.10487517.

[3] Klaas van den Berg and José María Conejero Manzano. "A conceptual formalization of crosscutting in AOSD". In: (2005). URL: https://www.researchgate.net/publication/228857937_A_conceptual_formalization_of_crosscutting_in_AOSD (visited on 2017-07-06).

[4] Mario Luca Bernardi and Giuseppe A Di Lucca. "Improving Design Pattern Quality Using Aspect Orientation". In: *Proceedings of the IEEE International

Workshop on Software Technology and Engineering Practice. STEP '05. 2005, pages 206–218. DOI: 10.1109/STEP.2005.14.

[5] James F. Blinn. "Simulation of Wrinkled Surfaces". In: *SIGGRAPH Comput. Graph.* 12.3 (Aug. 1978), pages 286–292. ISSN: 0097-8930. DOI: 10.1145/965139.507101.

[6] Jacob Borella. "The observer pattern using aspect oriented programming". In: *Proceedings of the Viking Pattern Languages of Programs*. Viking PLOP '03 (2003). URL: https://pdfs.semanticscholar.org/e744/9b80e7ac485cdfb0628b92bcf2b10aac4b28.pdf (visited on 2017-07-06).

[7] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. "Automatic Code Generation from Design Patterns". In: *IBM Systems Journal* 35.2 (May 1996), pages 151–171. ISSN: 0018-8670. DOI: 10.1147/sj.352.0151.

[8] Nelio Cacho, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thais Batista, and Carlos Lucena. "Composing Design Patterns: A Scalability Study of Aspect-oriented Programming". In: *Proceedings of the 5th International Conference on Aspect-oriented Software Development*. AOSD '06. Bonn, Germany: ACM, 2006, pages 109–121. ISBN: 1-59593-300-X. DOI: 10.1145/1119655.1119672.

[9] Craig Chambers, Bill Harrison, and John Vlissides. "A Debate on Language and Tool Support for Design Patterns". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '00. Boston, MA, USA: ACM, 2000, pages 277–289. ISBN: 1-58113-125-9. DOI: 10.1145/325694.325731.

[10] Satyan Coorg and Seth Teller. "Real-time Occlusion Culling for Models with Large Occluders". In: *Proceedings of the 1997 Symposium on Interactive 3D Graphics*. I3D '97. Providence, Rhode Island, USA: ACM, 1997, pages 83–99. ISBN: 0-89791-884-3. DOI: 10.1145/253284.253312.

[11] Franklin C. Crow. "Shadow Algorithms for Computer Graphics". In: *SIGGRAPH Comput. Graph.* 11.2 (July 1977), pages 242–248. ISSN: 0097-8930. DOI: 10.1145/965141.563901.

[12] William Donnelly and Andrew Lauritzen. "Variance Shadow Maps". In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. Redwood City, California: ACM, 2006, pages 161–165. ISBN: 1-59593-295-X. DOI: 10.1145/1111411.1111440.

[13] Chad Dougherty, Kirk Sayre, Robert Seacord, David Svoboda, and Kazuya Togashi. *Secure Design Patterns*. Technical report CMU/SEI-2009-TR-010. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2009. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9115 (visited on 2017-07-06).

[14] Amnon H Eden, Amiram Yehudai, and Joseph Gil. "Precise specification and automatic application of design patterns". In: Nov. 1997, pages 143–152. DOI: 10.1109/ASE.1997.632834.

[15] Eduardo Fernandez-Buglioni. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013. ISBN: 978-1119998945.

[16] Gert Florijn, Marco Meijers, and Pieter Van Winsen. "Tool support for object-oriented patterns". In: *Proceeding of the European Conference on Object-Oriented Programming*. ECOOP '97. Berlin, Heidelberg: Springer, 1997, pages 472–495. ISBN: 978-3-540-69127-3. DOI: 10.1007/BFb0053391.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 978-0201633610.

[18] David Garlan, Vishal Dwivedi, Ivan Ruchkin, and Bradley Schmerl. "Foundations and Tools for End-User Architecting". In: *Large-Scale Complex IT Systems. Development, Operation and Management*. LNCS volume 7539. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pages 157–182. ISBN: 978-3-642-34059-8. DOI: 10.1007/978-3-642-34059-8_9.

[19] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. "The COMQUAD Component Model: Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects". In: *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. AOSD '04. Lancaster, UK: ACM, 2004, pages 74–82. ISBN: 1-58113-842-3. DOI: 10.1145/976270.976281.

[20] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE International Conference on Requirements Engineering*. ICRE'94. Apr. 1994, pages 94–101. DOI: 10.1109/ICRE.1994.292398.

[21] Ouafa Hachani and Daniel Bardou. "Using aspect-oriented programming for design patterns implementation". In: *Proceeding of the Workshop Reuse in Object-Oriented Information Systems Design*. OOIS '02. 2002. URL: https://pdfs.semanticscholar.org/28a5/1cd8c99c5b55a18cd8d9be2673115eb76b2a.pdf (visited on 2017-07-06).

[22] Jan Hannemann and Gregor Kiczales. "Design Pattern Implementation in Java and aspectJ". In: *SIGPLAN Notices* 37.11 (Nov. 2002), pages 161–173. ISSN: 0362-1340. DOI: 10.1145/583854.582436.

[23] Stephan Herrmann. "A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java". In: *Applied Ontology* 2.2 (Apr. 2007), pages 181–207. ISSN: 1570-5838. URL: http://content.iospress.com/articles/applied-ontology/ao033 (visited on 2017-07-06).

[24] Stephan Herrmann. "Object Teams: Improving Modularity for Crosscutting Collaborations". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. LNCS volume 2591. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pages 248–264. ISBN: 978-3-540-36557-0. DOI: 10.1007/3-540-36557-5_19.

[25] Stephan Herrmann, Christine Hundt, and Marco Mosconi. "ObjectTeams/Java Language Definition". In: *Technical Report 2007/03, Technical University Berlin* (2007). URL: http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2007/2007-03.pdf (visited on 2017-07-06).

[26] Robert Hirschfeld and Ralf Lämmel. "Reflective designs". In: *IEE Proceedings - Software* 152.1 (Feb. 2005), pages 38–51. ISSN: 1462-5970. DOI: 10.1049/ip-sen:20041097.

[27] Minhuan Huang, Chunlei Wang, and Lufeng Zhang. "Toward a reusable and generic security aspect library". In: *AOSD:AOSDSEC '04: AOSD Technology for Application-level Security* 4 (2004). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.1688 (visited on 2017-07-06).

[28] Soumya Indela, Mukul Kulkarni, Kartik Nayak, and Tudor Dumitraș. "Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: ACM, 2016, pages 180–196. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986024.

[29] Anton Jansen and Jan Bosch. "Software Architecture as a Set of Architectural Design Decisions". In: *5th Working IEEE/IFIP Conference on Software Architecture*. WICSA'05. 2005, pages 109–120. DOI: 10.1109/WICSA.2005.61.

[30] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ". In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. ECOOP '01. Berlin, Heidelberg: Springer, 2001, pages 327–353. ISBN: 978-3-540-45337-6. DOI: 10.1007/3-540-45337-7_18.

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *Proceeding of the European conference on object-oriented programming*. ECOOP '97. Springer. Berlin, Heidelberg, 1997, pages 220–242. ISBN: 978-3-540-69127-3. DOI: 10.1007/BFb0053381.

[32] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*. Volume 3. John Wiley & Sons, 2013. ISBN: 978-0470845257.

[33] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang Cai. "A Study on the Role of Software Architecture in the Evolution and Quality of Software". In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, May 2015, pages 246–257. ISBN: 978-0-7695-5594-2. DOI: 10.1109/MSR.2015.30.

[34] Nicolas Lesiecki. *AOP@Work: Enhance design patterns with AspectJ*. 2005. URL: http://www.ibm.com/developerworks/java/library/j-aopwork5/ (visited on 2016-08-05).

[35] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. "Real-time, Continuous Level of Detail Rendering of Height Fields". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pages 109–118. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237217.

[36] Barbara Liskov. "Data Abstraction and Hierarchy". In: *SIGPLAN Notices* 23.5 (Jan. 1987), pages 17–34. ISSN: 0362-1340. DOI: 10.1145/62139.62141.

[37] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pages 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383.

[38] David Mapelsden, John Hosking, and John Grundy. "Design Pattern Modelling and Instantiation Using DPML". In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. CRPIT '02. Sydney, Australia: Australian Computer Society, Inc., 2002, pages 3–11. ISBN: 0-909925-88-7. URL: http://dl.acm.org/citation.cfm?id=564094 (visited on 2017-07-06).

[39] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002. ISBN: 978-0135974445.

[40] Robert Cecil Martin. *Design principles and design patterns*. 2000. URL: http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf (visited on 2017-07-06).

[41] Pete McBreen. *Questioning Extreme Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201844575.

[42] Bertrand Meyer. "Applying 'design by contract'". In: *Computer* 25.10 (Oct. 1992), pages 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279.

[43] Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. "An Aspect-oriented Approach for the Systematic Security Hardening of Code". In: *Comput. Secur.* 27.3-4 (May 2008), pages 101–114. ISSN: 0167-4048. DOI: 10.1016/j.cose.2008.04.003.

[44] Adel Noureddine and Ajitha Rajan. "Optimising Energy Consumption of Design Patterns". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, May 2015, pages 623–626. DOI: 10.1109/ICSE.2015.208.

[45] David Lorge Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dec. 1972), pages 1053–1058. DOI: 10.1145/361598.361623.

[46] David Lorge Parnas. "Software Engineering Principles". In: *INFOR: Information Systems and Operational Research* 22.4 (1984), pages 303–316. DOI: 10.1080/03155986.1984.11731932.

[47] David Lorge Parnas and Paul C. Clements. "A rational design process: How and why to fake it". In: *IEEE Transactions on Software Engineering* SE-12.2 (Feb. 1986), pages 251–257. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312940.

[48] David Lorge Parnas, Paul. C. Clements, and David M. Weiss. "The Modular Structure of Complex Systems". In: *IEEE Transactions on Software Engineering* SE-11.3 (Mar. 1985), pages 259–266. DOI: 10.1109/TSE.1985.232209.

[49] Srini Penchikala. *Implementing Object Caching with AOP*. Sept. 2004. URL: http://www.theserverside.com/news/1364528/Implementing-Object-Caching-with-AOP (visited on 2017-03-30).

[50] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. "Real-time Relief Mapping on Arbitrary Polygonal Surfaces". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. Washington, District of Columbia: ACM, 2005, pages 155–162. ISBN: 1-59593-013-2. DOI: 10.1145/1053427.1053453.

[51] Stephan Reiff-Marganiec, Hong Qing Yu, and Marcel Tilly. "Service Selection Based on Non-functional Properties". In: *Service-Oriented Computing - ICSOC 2007 Workshops*. LNCS volume 4907. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 128–138. ISBN: 978-3-540-93851-4. DOI: 10.1007/978-3-540-93851-4_13.

[52] Hesam Samimi, Chris Deaton, Yoshiki Ohshima, Alessandro Warth, and Todd Millstein. "Call by Meaning". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! '14. Portland, Oregon, USA: ACM, 2014, pages 11–28. ISBN: 978-1-4503-3210-1. DOI: 10.1145/2661136.2661152.

[53] André L. Santos and Duarte Coelho. "Java Extensions for Design Pattern Instantiation". In: *Software Reuse: Bridging with Social-Awareness*. ICSR '16. Cham: Springer International Publishing, 2016, pages 284–299. ISBN: 978-3-319-35122-3. DOI: 10.1007/978-3-319-35122-3_19.

[54] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. "SPL Conqueror: Toward optimization of non-functional properties in software product lines". In: *Software Quality Journal* 20.3 (Sept. 2012), pages 487–517. ISSN: 1573-1367. DOI: 10.1007/s11219-011-9152-9.

[55] Kotrappa Sirbi and Prakash Jayanth Kulkarni. "Stronger Enforcement of Security Using AOP and Spring AOP". In: *Journal of Computing* 2 (2010), pages 99–105. URL: https://www.semanticscholar.org/paper/Stronger-Enforcement-of-Security-Using-AOP-and-Spr-Sirbi-Kulkarni/94729a8ab666158103dfadb81c1e1663d15294c0 (visited on 2017-07-06).

[56] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. "N Degrees of Separation: Multi-dimensional Separation of Concerns". In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE '99. Los Angeles, California, USA: ACM, 1999, pages 107–119. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302457.

[57] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. “View-dependent Displacement Mapping”. In: *ACM Trans. Graph.* 22.3 (July 2003), pages 334–339. ISSN: 0730-0301. DOI: 10.1145/882262.882272.

[58] Lance Williams. “Casting Curved Shadows on Curved Surfaces”. In: *SIGGRAPH Comput. Graph.* 12.3 (Aug. 1978), pages 270–274. ISSN: 0097-8930. DOI: 10.1145/965139.807402.

[59] Carmen Zannier, Mike Chiasson, and Frank Maurer. “A model of design decision making based on empirical results of interviews with software designers”. In: *Information and Software Technology* 49.6 (2007). Qualitative Software Engineering Research, pages 637–653. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2007.02.010.

[60] Hansong Zhang and Kenneth E. Hoff III. “Fast Backface Culling Using Normal Masks”. In: *Proceedings of the 1997 Symposium on Interactive 3D Graphics*. I3D ’97. Providence, Rhode Island, USA: ACM, 1997, 103–ff. ISBN: 0-89791-884-3. DOI: 10.1145/253284.253314.