Trade-off-Oriented Development: Making Quality Attribute Trade-offs First-Class

Tobias Dürschmid
Carnegie Mellon University
duerschmid@cmu.edu

Eunsuk Kang
Carnegie Mellon University
eskang@cmu.edu

David Garlan
Carnegie Mellon University
garlan@cs.cmu.edu

***Abstract*—Implementing a solution for a design decision that precisely satisfies the trade-off between quality attributes can be extremely challenging. Further, typically quality attribute trade-offs are not represented as first-class entities in development artifacts. Hence, decisions might be suboptimal and lack requirements traceability as well as changeability. We propose Trade-off-oriented Development (ToD), a new concept to automate the selection and integration of reusable implementations for a given design decision based on quality attribute trade-offs. Implementations that vary in quality attributes and that solve reoccurring design decisions are collected in a design decision library. Developers declaratively specify the quality attribute trade-off, which is then used to automatically select the best fitting implementation. We argue that thereby, software could satisfy the trade-offs more precisely, requirements are traceable and changeable, and advances in implementations automatically improve existing software.**

***Index Terms*—Design decision, software architecture, quality attribute, non-functional property, reuse, software design**

I. Introduction

Software developers have to make trade-offs between quality attributes, such as security, performance, power consumption, reliability, and availability that are implicit in the implemented code [1]. For example, there are many considerations in implementing a pipe [2, pp. 53–70] to connect two components: Should the data stream be encrypted and signed to ensure confidentiality, which could decrease performance as side effect, and if yes, which algorithms to use? Should a ping / echo tactic or heartbeat tactic [1, pp. 87–89] be used to ensure availability of both components, which could reduce the bandwidth, and if yes, how often should the ping or heartbeat occur, and which implementation should be used? Should the data stream be compressed to increase the bandwidth, which could add a small performance overhead at both components?

To make optimal design decisions, developers have to be experts in all of these domains to know and evaluate a large set of alternatives [3]. However, nowadays, software is often developed by non-domain-experts. Studies in the security domain have shown that developers often lack necessary domain knowledge and struggle with finding and using a library for security features [4]. Therefore, they demand would benefit from techniques to specify and reason about at a higher level of abstraction [4]. Furthermore, since these trade-offs have to be made at various places of the software, quality attribute implementations are scattered over the code [5]. Additionally, design decisions that involve quality attribute trade-offs are typically implicit and not directly observable in the code. So it is often difficult to trace quality attributes and trade-offs among them to implementations or adjust the software if those requirements change [5], [6]. Cross-module changes, for example, are especially error-prone [7].

This research was supported by the Siebel Energy Institute award A022598.

To address these problems, we propose a new development concept called ToD that makes quality attribute trade-offs first class entities of programming by making them explicitly part of the code and design. ToD reuses existing implementations to a reoccurring design decision in a design decision library created by domain experts; enables non-expert developers to declaratively specify quality attributes, optionally with a priority, for each design decision; and automatically selects the best-suited implementation from those in the library.

We argue that this concept has three main advantages: (1) developers need less domain knowledge to create software with the demanded quality attributes, because ToD simplifies the reuse of expert solutions; (2) the resulting system more easily responds to changing requirements, because developers need only to adjust the declaratively specified quality attributes; and (3) if a new implementation that better suits the defined requirements is added to the design decision library, the software automatically uses the new implementation and, therefore, improves the overall software quality attributes.

II. Related Work

Feature-Oriented Programming [8] is an approach for flexible composition of objects and systems from a set of features. It provides variability in the functional dimension of the requirements by activating and deactivating features. While quality attributes can be affected by the activation of a feature, the focus is on changing the functionality. In contrast, the proposed concept provides variability in the non-functional dimension while keeping the functionality fixed. By changing the implementation of each feature, the intent is to specifically control the quality attributes of the system.

Aspect-Oriented Programming (AOP) [9] is a technique to modularize cross-cutting concerns (e.g., logging, monitoring, security). However, AOP is not designed to modularize quality attributes that are intently scattered (e.g., performance, memory consumption). So AOP is not intended to provide a first-class representation of all quality attribute trade-offs.

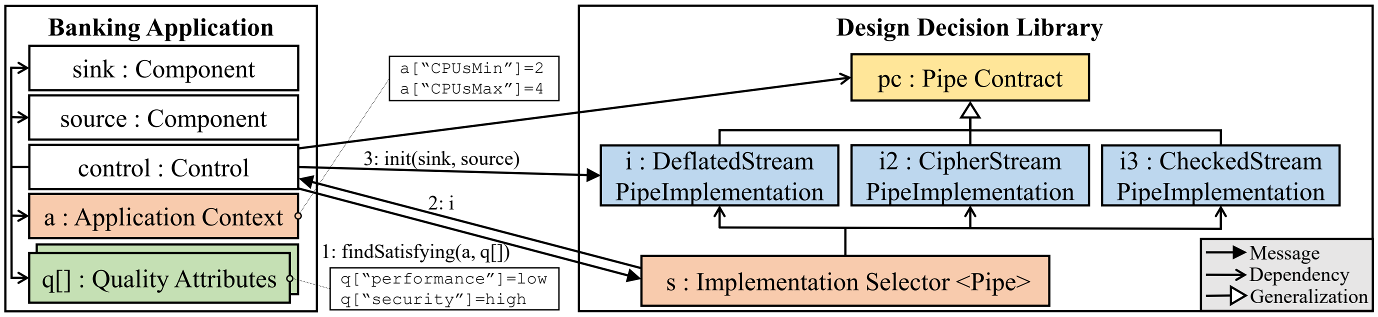

Fig. 1. Overview of ToD applied to the pipe example. The colors of the concepts are used in all following figures. Quality attributes are green, contracts are yellow, and implementations are blue. Other ToD concepts are red, while classes belonging to the example are white. The control needs a pipe implementation. The control forwards the quality attribute trade-off and the application context to the implementation selector (*findSatisfying*), which determines the best matching implementation. Afterwards the control initializes (*init*) the returned pipe implementation to connect the source component with the sink component.

The modeling technique *Concern-Oriented Software Design* [10] supports developers in selecting a reusable implementation by offering model-driven tools for exploring a hierarchical design space. Similarly, *End-user architecting* [11] is a concept that enables non-professional programmers to compose functionality to create programs. In these approaches, developers explicitly make each design decision by selecting one solution for each variation point without having to know the details of the low-level implementation. This guides developers in creating high quality software. However, the software does not express the trade-offs behind each decision. Therefore, it lacks requirements traceability. Furthermore, since the developers select one implementation, they can build upon implicit assumptions. Hence, later it is hard to change a decision if these requirements change. Our proposed approach targets these challenges by explicitly stating the required quality attributes while keeping the decision transparent.

TradeMaker [12] is an approach that automatically analyses trade-off spaces of design decisions and synthesizes an implementation from an abstract specification of object-relational mappers. It is focused on one domain and its performance-related quality attributes. So ToD generalizes this approach. Further, ToD builds on reuse rather than program synthesis.

III. Trade-off-oriented Development (ToD)

This section introduces the generic concepts of ToD, as visualized in Figure 1. As an example for ToD, to implement a pipe for connecting two components of a banking application, a developer would have to (1) search for the functional contract (explained below) of a pipe in the design decision library, (2) prioritize the involved quality attributes (e.g., data confidentiality, data integrity, and reliability), and (3) bind the inputs and outputs of the pipe contract to the application. Then the implementation that best satisfies this trade-off is automatically selected and used. Thereby, the trade-off of the required quality attributes is first-class represented in the code and easily changeable. Additional details of the use case scenarios can be found in Section IV.

A) Contract. A *contract* is a functional interface for design decision implementations that vary in quality attributes. Each contract can be implemented by a set of implementations

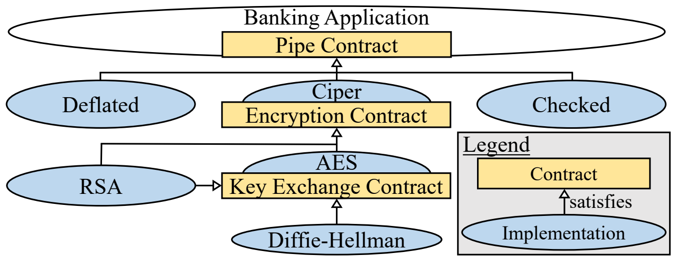

Fig. 2. Tree of design decisions. Implementations of contracts can delegate responsibilities to other contracts. Since quality attribute trade-offs are forwarded to the children, the peripheral application controls all decisions.

(e.g., encryption algorithms or pipe implementations). However, to provide changeability, developers should not make assumptions on a concrete implementation of a design decision. Instead, the best matching implementation is automatically selected according to the required quality attributes with associated priorities 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 and the trade-off becomes first class.

B) Design Decision Implementation. A *design decision implementation* is a concrete implementation of a corresponding contract. It is characterized by the degree to which it supports certain quality attributes. A contract can be implemented by many different implementations. Each implementation offers quality attributes 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 utility function is used to assess the design decision's suitability for a concrete context.

Implementations can delegate responsibilities to other contracts created by other domain experts, as shown in Figure 2. The quality attribute trade-offs are forwarded to the child decisions to tailor them to the peripheral application. Thereby, implementations can be reused while hiding the hierarchy from the user and delaying all inner decisions to the target application. Hence, a tree of design decision can be created that provides multiple levels of abstraction and a larger amount of context-tailored reuse.

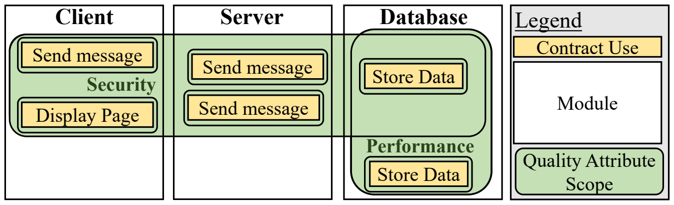

Fig. 3. The scope of quality attributes can cross-cut multiple modules (e.g., security), be limited to one single module (performance in the database module), or influence only a singe contract. Quality attribute scopes can overlap with each other.

C) Quality Attributes. A *quality attribute* is one dimension for characterizing an implementation of a contract. Examples are performance, memory consumption, security, or domain-specific properties, such as precision or fault-tolerance.

The application developer then specifies the trade-off between quality attributes by creating a utility function that weights the quality attributes. This could be a linear combination of the quality attributes, or quality attribute scenarios [1], or it could also include conditional logic (e.g, *as long as the computation is faster than one second, I do not care about performance*). By explicitly specifying the utility function that evaluates the implementations, the trade-off becomes fist class.

Each quality attribute has a scope to which it applies, as shown in Figure 3. This can be the use of a single contract, but it can also be a whole module (such as the networking component), or a cross-cutting feature of the system (such as all messages sent to one specific receiver). For example, if one part of the system becomes a bottleneck, the performance requirements can be prioritized higher for this specific part of the system. So there is no need to consider performance for other system parts that do not constitute a bottleneck. To implement scopes, quality attributes can be expressed as separate artifacts that, similarly to pointcuts in AOP, abstractly specify the contracts, modules, or features to which they apply. Further, reoccurring configurations of quality attribute trade-offs can be defined as presets (e.g., high security, high availability, and low scalability for the `SmallWebApp` preset) to be reused and optionally refined conveniently and frequently.

D) Application Context. Information about the hardware on which the software is expected to run (e.g., memory, CPU speed, amount of cores, and GPU properties) and the environment (e.g., speed, reliability and security of the network) can influence the decision about which implementation is best suited. Therefore, developers can optionally specify information about the application context in a standardized format (e.g., in a key-value schema like #CPU cores → 2–4, or as model of the runtime architecture). This information usually is uniform across the whole application and will be used for all trade-offs.

E) Implementation Selector. The *implementation selector* automatically chooses an implementation for a contract. To select the implementation that best satisfies the trade-off, the selector evaluates each implementation according to the priorities of quality attributes and the context. The implementation selector uses, if possible, automated analysis, such as performance metrics as carried out by TradeMaker [12] and security metrics, or otherwise an assessment functions $q : context \times qualityAttributes \rightarrow \mathbb{R}$ manually defined by domain experts, of the implementations to determine which best satisfies the requirements. The optional information of the application context can be used to tailor the evaluation to the execution environment of the software.

IV. Use Case Scenarios

ToD mainly targets these scenarios: (A) implementing a feature without being an expert in the domain, (B) adapting the software according to changing quality attribute requirements, and (C) updating to new advancements in the domain.

A) Implementation. Junior developer Alice needs a pipe implementation for her distributed banking system. She browses the design decision library for the pipe contract by searching for the term “pipe” in the index. Because sensitive data is transfered, she highly prioritizes data confidentiality, data integrity, and reliability. She thinks availability deserves medium priority, because it is more important to ensure that the data is transfered correctly. Performance is assigned the lowest priority, since delays are acceptable. So she creates a utility function that weights these quality attributes according to these priorities and passes it to the design decision library. Since she knows the hardware properties and the network capacity, she adds this information to the application context, as recommended in the documentation of the pipe contract.

B) Changing Requirements. After some time of testing, Alice recognizes that there happens some communication between the two components that does not involve sensitive data, which became a performance bottleneck because of the encryption overhead. Since the quality attribute trade-off is represented in the code as a first-class entity, she can directly trace the location where this decision was made. So she reduces the scope of data confidentiality to sensitive data exchange only and increases the priority of performance at the bottleneck. Internally, different implementations are used for the corresponding types of communication and the system is now tailored to the changed requirements.

C) Implementation Update. A security researcher just published a new encryption algorithm that provides a higher level of security while still being faster to compute than existing algorithms. This attributes are determined by automated tests that are attached to the implementation and the assessment of other domain experts that know all encryption algorithms in the library. Alice does not know about this new advancement, but since the implementation became part of the design decision library, her application automatically selects the improved version. After the next deployment, her banking application runs faster and provides a higher level of security. So Alice as a non-expert in security can reuse the high-quality implementation created by a security researcher. In general the design decision library is supposed to be developed and maintained by a community of domain experts while the target audience is mainly non-expert developers.

V. Research Challenges

To make ToD usable in practice, some further research questions have to be addressed:

How to deal with implementations that do not match the interface of the contract? ToD is limited to implementations that share a common interface, the contract. If an implementation requires other inputs, it is incompatible with the general contract. A contract with the additional inputs can be created that automatically includes the general contract using a wrapping implementation. However, since this workaround works for a limited amount of input sets only, it remains to be explored how to deal with large varieties of input sets. However, the general problem of adaptation to interface mismatch is still an open problem [13].

How to evaluate quality attributes? ToD relies on an objective and comparable assessment of quality attributes of implementations. In the optimal case, this is done automatically. This is already possible for performance [12] and partially for specific network security attacks [14], but hardly for quality attributes such as reliability and availability. So future work should research techniques to evaluate all quality attributes.

How to specify quality attribute trade-offs? In ToD quality attribute trade-offs need to be specified using a precise, formal notation to enable automatic implementation evaluation. Most of the prior works on interface specification have focused on functional requirements, although there are recent works on specification of certain quality attributes such as reliability (e.g.,[15]). However, further research is needed to devise specification mechanisms to express a wide range of quality attributes and their relations to each other.

How to optimize the quality attributes globally? In ToD each decision point locally optimizes quality attribute trade-offs. However, the overall goal of software architecture is to optimize the quality attributes of the whole system. So when applying ToD, one still needs an architect who maintains an overview of the whole system and defines constraints for each component. For example, to ensure that the overall system does not consume more than x memory, the implementation for this contract should not consume more than y memory, where y depends on x. This directly results in the question of how to globally optimize the decisions of the contracts. Since this exploration space is potentially exponential, solve this efficiently is a non-trivial task.

VI. Discussion and Conclusions

We propose ToD, a new concept to make quality attribute trade-offs first class. We argue that thereby, requirements are more easily traceable and changeable. Furthermore, developers will concentrate more on quality attributes. ToD has the potential to change the way that software is developed by facilitating rapid cycles of development, testing, and adjusting requirements. This in turn will enable developers to deliberately control the quality attributes of each part of the system separately. For example, performance optimizations would be limited to bottlenecks and security optimizations would be limited to sensitive parts of the system. Moreover, newly developed solutions that more precisely satisfy the trade-offs will automatically be integrated into the software due to the decoupling of trade-off specifications from the solutions. Thereby, advances in research and development of new solutions might have more direct and immediate impact on practice. Finally, since less expertise is needed to develop high quality systems, this concept is one step towards the goal of making software development more accessible to non-experts.

ToD can be applied to, for example, architectural connectors, resource management, persistence, computer graphics, data structure, security, and other domains with unified contracts and varieties of implementations. So ToD is mainly intended for domains that yet well understood and thats community has agreed on a common problem definitions and common standard implementations.

ToD has the potential to dramatically change the way that code is reused. Currently either a small, negligible portion of software is reused, or a larger portion that includes many decisions that might not fit the context of the reusing software and, therefore, limits the applicability of the reused code. ToD combines both advantages to delay all decisions that are made in reused code to the client that uses it. So developers would not decide to use a specific security library, but state the decisions that are made during the development of the security library. Thereby it is one steps towards the vision of mass customized modules.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed.

[2] F. Buschmann, R. Meunier, H. Rohnert, S. Peter, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*, vol. 1.

[3] D. Tofan, M. Galster, and P. Avgeriou, "Difficulty of architectural decisions – a survey with professional architects," in *Software Architecture*, K. Drira, Ed., Springer Berlin Heidelberg, 2013.

[4] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?" In *Proc. ICSE 2016*.

[5] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Proc. WICSA 2005*.

[6] M. P. Robillard, "Sustainable Software Design," in *Proc. FSE 2016*.

[7] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai, "A study on the role of software architecture in the evolution and quality of software," in *Proc. MSR 2015*.

[8] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proc. ECOOP 1997*, M. Akşit and S. Matsuoka, Eds.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. ECOOP 1997*.

[10] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *Proc. MODELS 2013*.

[11] D. Garlan, V. Dwivedi, I. Ruchkin, and B. Schmerl, "Foundations and tools for end-user architecting," in *Large-Scale Complex IT Systems. Development, Operation and Management*, ser. LNCS 7539. 2012.

[12] H. Bagheri, C. Tang, and K. Sullivan, "TradeMaker: Automated Dynamic Analysis of Synthesized Tradespaces," in *Proc. ICSE 2014*.

[13] C. Canal, J. M. Murillo, and P. Poizat, "Software Adaptation," *L'Objet*, vol. 12, no. 1, 2006.

[14] A. Ramos, M. Lazar, R. H. Filho, and J. J. P. C. Rodrigues, "Model-Based Quantitative Network Security Metrics: A Survey," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, 2017.

[15] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," *SIGPLAN Not.*, vol. 49, no. 10, Oct. 2014.