Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager

Thomas Leich, Sven Apel and Gunter Saake

Department of Computer Science,
Otto-von-Guericke-University Magdeburg
{*leich*, *apel*, *saake*}@iti.cs.uni-magdeburg.de

Abstract. In recent years the deployment of embedded systems has increased dramatically, e.g. in the domains of sensor networks or ubiquitous computing. At the same time the amount of data that have to be managed by embedded systems is growing rapidly. For this reason an adequate data management support is urgently needed. Current database technologies are not able to cope with the requirements specific to embedded environments. Especially the extreme resource constraints and the diversity of hardware plattforms and operating systems are challenging. To overcome this tension we argue that embedded database functionality has to be tailored to the application scenario as well as to the target platform. This reduces the resource consumption and customizes the data management to the characteristices of the plattform and the application scenario. We show that component techniques and feature-oriented programming help to face the mentioned limitations without focusing on special-purpose software. We present the design and the implementation of a database storage manager family. We discuss how feature-oriented domain analysis and feature-oriented programming help to do this task. Our evaluation criteria are the number of features and the flexibility to combine these features in different valid variants.

1 Introduction and Motivation

The domain of embedded systems is growing rapidly [15]. Approximately 98 % of all computer devices are deployed as embedded systems [34]. It is expected that pervasive and ubiquitous computing will push this trend in future [36]. Due to the low cost of embedded hardware, software-development on embedded systems is a hard challenge. The limitations on hardware, e.g. CPU-power, memory capacities or battery constraints make high demands on software-development. The result of these limitations is that applications developed as special-purpose software are tailored to a specific application scenario. Modern software-engineering methods, known from other domains are rarely used. We argue that component techniques, *Feature-Oriented Programming (FOP)* [8], and *Mixin Layers* [31] can help to reduce the devoplement cost and the time-to-market. Moreover, this software-engineering methods help to face the ressource restrictions without focusing on special-purpose software. Several promising studies [5, 1, 4, 12]

show that FOP and mixin layers are appropriate to implement such layered, step-wise refined architectures. A further problem is the absence of standard infrastructure, e.g., database services. This makes developing embedded system applications more complicated. Due to the enormous number of variants of hardware and operating systems, software developers are swamped with finding the right vendor of infrastructure services. Since a few years the idea of product-lines is discussed in this context. Product-lines are supposed to maximize the reuse of existing components as well as to increase extensibility and customizability. This paper focuses on product-line technology for embedded data management infrastructure services.

In this contribution we present our first results towards a flexible, lightweight storage manager for embedded systems. The key idea is to implement the storage manager as a highly configurable program family [29][1]. Different family members (a.k.a configurations) satisfy the needs of different application scenarios: e.g. several embedded sensors require different data management functionality than data collectors or a mobile measurement unit in form of a PDA [37]. Furthermore, the high degree of configurability as well as the well thought design of the family allow to develop a highly portable storage manager. To implement a highly configurable program family we utilize feature-oriented domain analysis [22], feature-oriented programming [3] and mixin layers [31]. It is not obvious how the combination of these methods, integrated into the domain engineering process [16], leads to configurable, reusable and extensible data management software.

The article is structured as follows: Section 2 introduces a sensor network scenario and points to problems regarding embedded storage management functionality. Section 3 reviews the relevant software engineering methods used here. The subsequent sections present our storage manager architecture. In Section 5 we discuss our implementation results and review related work. Finally, we conclude in Section 6.

2 An Application Scenario

This section sketches an application scenario for embedded data management. Thereupon, we point out challenges of embedded data management.

2.1 A Sensor-Network Application Scenario

Due to the advances in wireless sensor-network technologies previous research focused on in-network aggregation and query processing. Most existing sensor applications rely on a centralized system for collecting data. Centralized data collection and analysis should provide cheap sensor nodes and minimal resource consumption. However, there are still a lot of problems: Sensor-networks are

[1] Although there is a subtle difference between program and product families (see [16]) we use these terms synonymously.

often intended for long-term deployment. Therefore, they underlay extreme resource constraints. One consequence of limited resources is that they are highly communication constrained and therefore data buffering on the sensor node is required. Another problem is that pre-aggregation and centralized systems lack flexibility because data are extracted in a predefined way. However, an pre-aggregation of raw data on nodes is possible only if the features of interest are known a priori. This is not often the case in practice. Thereby an lightweight and efficient buffering and access on raw data is essential for an ad hoc aggregation on sensor nodes.

The following example scenario is borrowed in parts from [37, 19]. We focus on sensor-networks used in scientific applications, e.g. micro-climate and habitat monitoring. Low-end sensor nodes are detecting environmental parameters, e.g. temperature or light intensity. These modern sensors do not only respond to physical signals to produce data, they also embed computing capabilities for independent activity. Data collectors are special nodes to gather data from affiliated sensors to provide data for in- and out-network analyses. The different node types are ordered in a hierarchical way and have widely varying requirements on storage management services. For our scenario we point out two different categories of heterogeneous devices:

– The first category are simple *sensor nodes* that only need a data structure to store data. Because of the hardware restrictions, all data are stored in the main memory. The size of a data record is known. The data structure needs only efficient insert, update, and lookup operations. Usually, sensor data are measurements. Therefore simple integrity checks of data are needed.
– The second category are *data collectors* that collect and aggregate data of different simple sensor nodes. To store data persistently the collector-node uses a secondary storage device (an additional flash chip). To optimize the processing of data a caching manager is required. Also complex integrity checks are needed.

2.2 Problems Occurring

In the scenario introduced certain problems occur: Common database implementations cannot provide the full range of the required functions by attending the strong resource limitations. The monolithic system structure prevents the reuse of logical device-independent functionality. These general-purpose systems are not scalable[2] enough to satisfy the resource restrictions. The features of database services are not tailorable in such a fine-grained sense. The main reason is well known as crosscutting concerns. Special-purpose data management services dealing with strong resource restrictions are not flexible enough to provide services to all kinds of sensor-node types presented in the scenario. Application developers have to choose the embedded database fitting best to their application, hardware and software requirements. This is a difficult, time consuming

[2] In the sense of scale their memory footprint.

and costly process, with lots of compromises. An adequate solution could be the concept of program-family architecture that can be tailored and optimized to the application scenario. Our goal are summarized as follows:

- systematic and detailed analysis of the domain of embedded data management
- customizability, reusability and extensibility through fine-grained features
- lightweight and portable implementation
- seperating crosscutting concern

3 Software Engineering Background

This section introduces the software engineering methods that we have used to analyze the domain of embedded storage management as well as to design and implement a program family of storage management. Following this idea we have used for the domain analysis feature-oriented domain analysis (FODA) [22]. Thereupon, we have designed a program family based on step-wise refinements and feature-oriented programming (FOP) [31]. Mixin layers are used as implementation technique.

3.1 Feature-Oriented Domain Analysis

With the domain analysis feature modeling is an appropriate software engineering method [22]. The goal of FODA is to analyze the considered target application scenarios and to derive the required and optional features. Since the focus of FODA is on a domain of applications the resulting features are chosen with regard to a whole family of systems. The results of feature modeling are feature models that describe the features, their relations, constraints, and dependencies [16]. These models express variation points and commonalities of the target-programs in an abstract and implementation independent way. Features are organized in a hierarchical way (see Fig. 1). Features are *mandatory*

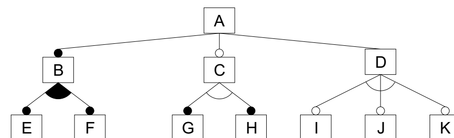

Fig. 1. Example feature tree

or *optional* stated by filled *(e.g. feature B)* and empty circles *(e.g. feature C,D)*. Moreover, they can be related in two ways: *alternative (e.g. feature G,H)*, connected by an empty arc and *or (e.g. feature E,F)*, connected by a filled arc. Feature models are one appropriate basis for designing and implementing program families [16].

3.2 Program Families and Step-Wise Refinements

Parnas [29] introduced program families first. The idea is to build software incrementally, using minimal building blocks and starting from a minimal base. This procedure is also known as *step-wise refinement* [31]. Exchanging, adding and removing such building blocks, also called *layers*, yields reusability, extensibility, and customizability. Batory et al. have mapped this concept to the object-oriented world [5, 31]. They have observed that a new *software feature* often extends or modifies numerous existing classes. Based on this observation, they perceive features as *collaborations of class/object fragments*, also referred to as *roles*. Figure 2 shows a stack of collaborations. Classes are arranged vertically

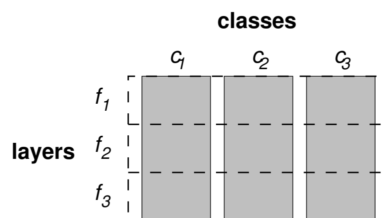

Fig. 2. Stack of collaborations

(c_1 – c_3). Collaborations are arranged horizontally and span several classes (f_1 – f_3). Several features of a software system result in a stack of collaborations. In our context, examples of features are supported data types or caching strategies. Collaborations with the same interfaces are easily exchangeable. They are an instance of large-scale components [5]. In the sense of *Feature-Oriented Programming (FOP)* [31], a collaboration of objects implements a feature and is part of a layered stack.[3]

3.3 Mixin Layers

Mixin layers are one appropriate implementation technique to implement features in form of collaborations [31]. A mixin layer is a static component encapsulating fragments of several different classes (mixins) so that all fragments are composed consistently. Mixin layers are an approved implementation technique for component-based layered designs. Advantages are the high degree of modularity and the easy composition [31]. *AHEAD (Algebraic Hierarchical Equations for Application Design)* is an architectural model for FOP and a basis for large-scale compositional programming [7]. The *AHEAD Tool Suite (ATS)*[4], including the *Jak* language, implements AHEAD for Java.

[3] We use the terms feature and layer as synonym for collaboration.
[4] http://www.cs.utexas.edu/users/schwartz/Hello.html

3.4 Seperating Crosscuting Concerns

Pioneer work on software modularity was made by Dijkstra [17] and Parnas [29]. They have proposed the principle of *separation of concerns*. The idea is to separate each concern of a software system in a separate modular unit. They argue that this lead to maintainable, comprehensible software, which can be easily reused, customized and extended. Since a few years *Aspect-Oriented Programming (AOP)* and FOP are discussed as solutions of this problem. AOP was introduced by Kiczales et al. [23]. The aim of AOP is to separate crosscutting concerns. Common object-oriented methods fail in this context [23, 16]. The idea behind AOP is to implement so called orthogonal features as *Aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using join point specifications (*pointcuts*), an aspect weaver brings aspects and components together.

There are several discussions of pros and contras on separating crosscutting concerns using on AOP and FOP [25, 24]. In this paper we are concentrating on heterogeneous crosscutting concerns. Heterogeneous crosscuts are distributed over several join points but apply varying code. That means different pieces of code are added to lots of different places. Homogeneous crosscutting concerns are distributed over several join points, but apply the same code fragments, e.g. locking or logging. Therefore the same piece of code is added to lots of different places. Current AOP languages focus on homogeneous concerns whereas FOP languages deal with heterogeneous concerns.

4 Storage Manager Design and Implementation

This section presents the domain analysis, design and implementation of the storage manager. Due to the limitations of space, we only focus on essential characteristics that are related to the presented scenario (see Section 2).

4.1 FODA

Figure 3 shows a subset of the feature model as result of FODA that describes the variability of our storage manager family.

The grey boxes symbolize features that have not displayed sub-features. This is because of the space limitations. The storage manager is separated in four mandatory features: (1) *Data Type (DT)* that represents the supported data types, (2) a *Buffer Manager (BM)* for storage data in primary or secondary memory, and managing the free space, (3) a *Storage Organisation (SO)* for structuring and accessing data and (4) *Records (Rec)* which represents the data in our database. Optional features are the *Integrity Checks (IC)* and supported *File Types (FT)*.

The overall feature-model of our small storage manager family has 93 features. We have not investigated in special data types, transaction management,

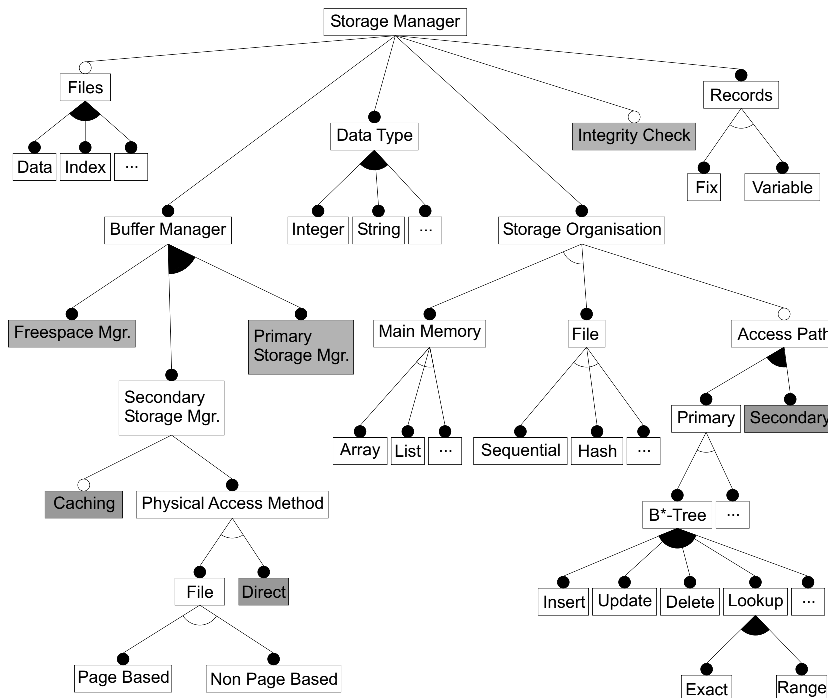

Fig. 3. Feature model of the storage manager family

recovery or specialized data structures for highly restricted application scenarios like smartcards [10]. An more extended analysis would produce hundreds of additional features for a storage management system.

Table 1 depicts variable parameters, e.g. number of supported data types (first two colums). The third column depicts the values for our experimental evalutation, e.g. for calculation of the permitted variants of the storage manager we assumed four different data types. Calculating the theoretical number of variants (cf. Table 1), we have determined 8.164.800 possible configurations (using a GenVoca grammar [6]).

$$\#SM = \underbrace{(2^f - 1)}_{FT} * \underbrace{(15)}_{BM} * \underbrace{(2^n - 1)}_{DT} * \underbrace{((m + s) * (6 * a * (2^o - 1)))}_{SO} * \underbrace{(2)}_{IC} * \underbrace{(2)}_{Rec}$$

The high amount of amount of feature combinations reforces the diversity of the database domain. The abstract description of variants and commonalities can be exploited to build a highly configurable database program family. The most combination differ only in a few details, e.g., the number of supported data types. However, we argue that only this fine-grained design can lead to optimally tailored database services.

parameter	description	# for caluculation
d	data types	4
f	file type	2
m	main memory organisation	2
s	data file structures	2
a	access structure	2
o	B^* Tree	6

Table 1. Adjustable parameters.

4.2 Design and Implementation

In order to evaluate our approach, we have implemented the storage manager, using the AHEAD Tool Suite which supports FOP for Java. It is also feasible to use the C++ template mechanism, nested classes and parameter-based inheritance [1, 31, 2]. Because of missing tool support and several problem regarding C++, we decided to utilize Java and AHEAD to prove our concept.

Figure 4 depicts a subset only. Mainly, the layers concerning the B^*-Tree access structure are depicted in bottom up order. Starting from the basic layers, which implement records, page storages and caching, the layer stack is refined to the B^*-Tree structure and several operations (Fig. 4 depicts the insert-operation only). The layer stack crosscuts about 26 classes and a couple of help-classes. In average, we have refined 3 classes per layer. To implement for instance the *Page Based Storage*, we had to refine three classes (*File, FreeSpaceMgr, SecStorageMgr*) and added one new class (*Page*).

5 Results and Experiences

This section discusses our results and experiences in implementing the prototype. Therefore, we use the scenario introduced in Section 2.

5.1 Configuration

The configuration process is easy.To convey the ease of the configuration procedure and the flexibility of the implementation, we have derived several storage managers:

Sensor Node. We have configured two different storage manager versions for our sensor nodes. Both configurations use main memory management. The main memory allocation is static, because of the fixed record length. The first sensor node type uses only basic data types (integer, number) and a simple array to store data. The resulting storage manager is step-wise refined by 11 layers. For the second category, we have configured a hash-map instead of an array. Thus the update and lookup functionalities are efficient supported. Furthermore, we have added an integrity check on the records. This storage manager is created by 16 layer refinement.

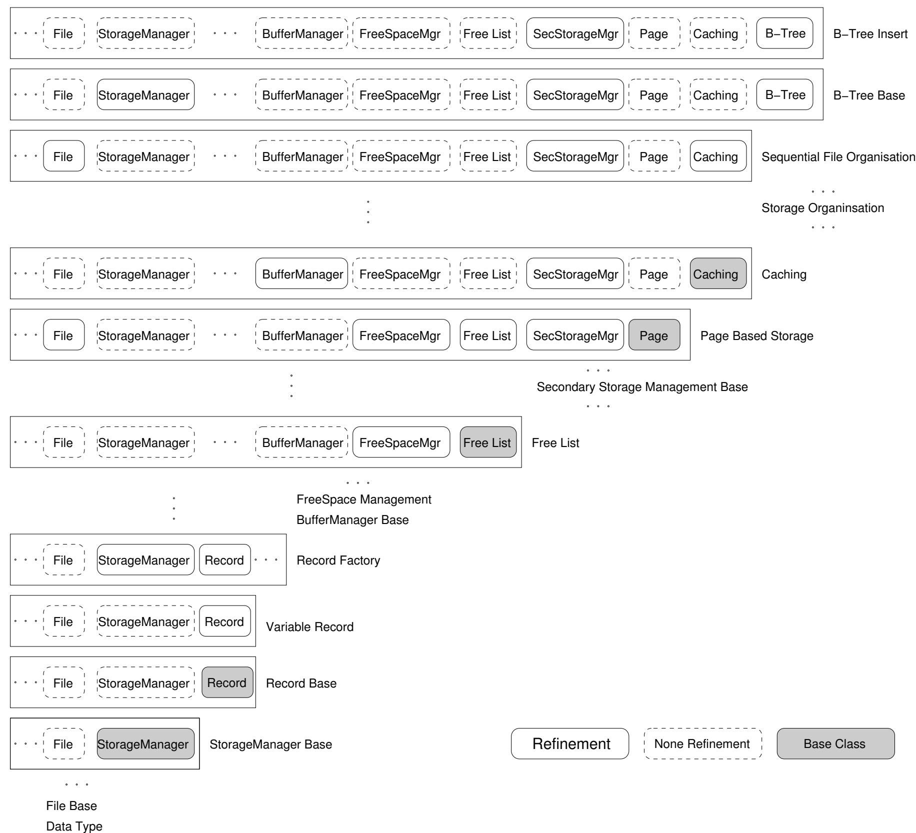

Fig. 4. Subset of implementes mixin layer

Data Collector. Due the application scenario the functionality of the data collector is more complex (cf. Sec. 2). Because of the availability of a secondary storage device, we have configured a secondary storage manager using a file oriented storing and an internal page based organisation. The file is sequentially ordered and for the access path we have used a B^*-Tree. To improve the performance we have chosen a cache management. The data record has variable length. Due to the task of the data collector we integrated a special integrity check on records. The total number of layers is 38 layers.

The correct syntactical composition of layers for a particular configuration is determined by equation files. The semantic correctness is ensured by DRC. We figured out that approximately 60 % of all possible configuration were excluded using DRC. We omit a detailed discussion because that is out of scope of this paper.

5.2 Discussions and Comparison to Related Approches

Incremental software development is an adequate process of building programs from simple ones by successively adding programmatic details. We have used these development methods to build tailored storage manager support for resource-restricted devices. Our implementation has shown that decomposition of storage manager into fine-grained components is possible. FODA, FOP and mixin layers are adequate software engineering methods to achieve highly scalable and lightweight software.

Direct comparisons e.g. performance analysis, code metrics, to other database solutions, e.g. COMET DBMS [26], Berkeley DB [28], are not meaningful at the current state of the work for several reasons:

- The set of implemented features of our storage manager is different to other approaches.
- We have not implemented any error-handling or logging functions, so that an objective performance analysis would be adulterated. We are going to implement this functions in future work.
- Our approach is implemented in Java. To the best of authors known, there is no other approach focussing on such a fine-grained tailorability of components by using Java implementing database functionalities.

For these several reasons we are comparing our approach to other solutions only on the concept level. First we compare our approach to Berkeley DB as known system in this area: Berkeley DB [28] is a common embedded database system, which is implemented in C and Java. The Berkeley DB consists of the following sub-systems: access methods, memory pool, transactions and locking. Hence, Berkeley DB is configurable on the sub-system level. However the components are coarser structured, as in our presented approach. An exchange of, e.g., the access methods, is complex due to a high degree of dependency of the sub-systems. This prevents an easy exchanging and extending of the database system. This fact is also confirmed by Tesanovic et. al [33]. They investigated on homogeneous crosscutting concerns in the Berkeley DB. With separating and implementing failure detection and synchronisation through aspects a code reduction up to 57 % was showed. This fact proves that crosscutting concern in Berkeley DB complicates tailorability and extensibility. Moreover Tesanovic et. al showed that there is a trade-off between the tailorability and maintainability of the system when aspects are used.

For the second comparison we choose COMET DBMS [26, 32]. COMET DBMS is a component-oriented DBMS for embedded real-time systems. The research focuses on applying aspect-oriented and component-based software development to real-time system development. COMET is decomposed into seven basic components. These are: user interface component, transaction scheduler component, locking component, indexing component recovery and logging component, memory handling component, and transaction manager component. Furthermore the system is decomposed in tree types of aspect: run-time, composition, and application aspect. One of the application aspects is the concurrency

control aspect. This aspect crosscuts four basic components, namely the user interface component, transaction scheduler component, locking component, and transaction manager component. A clean separation of this aspect helps reconfiguring COMET to support locking or non-locking transaction execution. The COMET-project has shown that especially in real-time scenarios a lot of code is distributed as homogeneous crosscutting concern over several implementation units. Our approach has shown that in very fine-grained decomposed storage management systems heterogeneous concerns are challenging problems. Remaining on the discussion in [25, 24] both types of concerns are common in today's system. Consequently, our objective in future work is to enhance our prototype with AOP features to deal with homogeneous crosscutting concern as well as with heterogeneous crosscutting concerns.

Finally, we have a closer look to more generalized results of our work. Choosing feature components in large scale database management software from a program family the system complexity can be reduced. This helps to reduce the maintenance overhead and new feature like automatic tuning can be easy evaluated and included in less complex software [14].

5.3 Related Work

Extensibility, customizability and flexibility on database systems are a research area that has been actively studied. An overview and classifications on extensibility can be find in [18, 20]. Prototype systems like GENESIS [9], STARBURST [30], KIDS [21], EXODUS [13], etc, are commonly known in this research area. The XXL-library [11]is another prominent approaches to achieve extensibility and customizability based on using object-oriented design patterns. A more specific overview on embedded systems and real-time data management can found in Tesanovic et. al [32]. Olson points out in how to find the right database systems for embedded system environments [27]. Typical special-purpose database management solutions for embedded systems are e.g. GnatDB [35] for digital right management or PICO DBMS [10] for data management support on smartcards.

6 Conclusion and Further Research

Feature-oriented software methods and step-wise refinements advance the design and implementation of database functionality for embedded systems. In this article we have proposed a combination of FODA, FOP and mixin layers as feasible software engineering methods to implement a storage manager as a program family. A subset of basic features of a storage manager has been analysed and implemented, to show a high degree of flexibility and tailorability of our approach. Therefore we have presented an application adopted scenario from sensor networks, which shows different requirements on storage management in this area. Through an easy configuration process we derived three different variants

form our storage manager product family tailored to the different application scenario.

As future work, we want to investigate and integrate more features. Our tokens of interests are special purpose algorithms resource restrict devices, transaction management, real-time feature and query processor. Furthermore, we want to investigate the performance and the memory footprint and how to encourage the configuration process for data management through deriving information from application scenario automatically. Furthermore we want to investigate our new language FeatureC++ an extension to C++ that supports FOP [2]. Moreover FeatureC++ improve the problem of crosscutting modularity by combining traditional FOP concepts with concepts of AOP.

References

1. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In Cecilia Mascolo and Thomas Gschwind, editors, *Software Engineering and Middleware Fourth International Workshop, SEM 2004, Linz, Austria*, volume 3437 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2005. to appear.
2. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical Report Preprint Nr. 3, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
3. D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering*, pages 702–703. IEEE Computer Society, 2004.
4. D. Batory, Lou Coglianese, M.Goodwin, and S. Shaver. Creating Reference Architectures: An Example from Avionics. In *Symposium on Software Reusability (SSR)*, Seattle Washington, 1995.
5. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), 1992.
6. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.
7. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proc. of the 25th Int. Conf. on Software Engineering*, 2003.
8. D. Batory, J. N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.
9. D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: an Extensible Database Management System. In *Readings in object-oriented database systems*, pages 500–518. Morgan Kaufmann Publishers Inc., 1990.
10. C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *VLDB 2000, Proceedings of 26th International 2000, Cairo, Egypt*, pages 11–20, Los Altos, CA 94022, USA, 2000. Morgan Kaufmann Publishers.

11. M. Cammert, C. Heinz, J. Krämer, M. Schneider, and B. Seeger. "a status report on xxl - a software infrastructure for efficient query processing". *IEEE Data Eng. Bull.*, 26(2):12–18, 2003.
12. R. Cardone et al. Using Mixins to Build Flexible Widgets. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, 2002.
13. M. J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In K. R. Dittrich, U. Dayal, and A. P. Buchmann, editors, *On Object-Oriented Database Systems*, Topics in Information Systems. Springer, 1991.
14. S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *The VLDB Journal*, pages 1–10, 2000.
15. Business Communications Company. Future of Embedded Systems Technology, 2000. BCC Press release on market study RG-229.
16. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
17. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
18. K. Dittrich and A. Geppert. Component Database Systems: Introduction, Foundations, and Overview. In K. R. Dittrich and A. Geppert, editors, *Component Database Systems*, pages 1–28. dpunkt.verlag, San Francisco u.a., 2001.
19. D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An Evaluation of Multi-resolution Storage for Sensor Networks. In *Proceedings of the ACM SenSys Conference*, pages 89–102, Los Angeles, California, USA, November 2003. ACM.
20. A. Geppert. Methodical Construction of Database Management Systems. *GI Datenbank Rundbrief*, 14:62, 1994.
21. A. Geppert, S. Scherrer, and K. Dittrich. KIDS: Construction of Database Management Systems based on Reuse. ifi-97.01, Department of Computer Science, University of Zurich, January 9 1997.
22. K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, cmu/sei-90-tr-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.
23. G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.
24. R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. extended report. Technical Report CS-TR-05-16, The University of Texas at Austin, Department of Computer Sciences, 1 2005. Thu, 19 May 105 18:18:04 GMT.
25. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.
26. D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04)*, Edinburgh, Scotland, May 2004. IEEE Computer Society Press.
27. M. A. Olson. Selecting and Implementing an Embedded Database System. *IEEE Computer*, 33(9):27–34, 2000.
28. M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999.
29. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions On Software Engineering*, SE-5(2), March 1979.

30. P. M. Schwarz, W. Chang, J. Freytag, G. M. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the starburst database system. In Klaus R. Dittrich and Umeshwar Dayal, editors, *1986 International Workshop on Object-Oriented Database Systems, September 23-26, 1986, Asilomar Conference Center, Pacific Grove, California, USA, Proceedings*, pages 85–92. IEEE Computer Society, 1986.
31. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 11(2), 2002.
32. A. Tesanovic, D. Nystrom, J. Hansson, and C. Norstrom. Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach. Technical report, Linkoping University, Mlardalen University, 2002.
33. A. Tešanović, K. Sheng, and J. Hansson. Application-Tailored Database Systems: a Case of Aspects in an Embedded Database. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS'04)*, Coimbra, Portugal, July 2004. IEEE Computer Society Press.
34. J. Turley. The Two Percent Solution. *Embedded Systems Programming*, 2002. http://www.embedded.com/story/OEG20021217S0039.
35. R. Vingralek. GnatDb: A Small-Footprint, Secure Database System. In *VLDB*, pages 884–893, 2002.
36. M. Weiser. Hot Topics: Ubiquitous Computing. *IEEE Computer*, 26(10), 1993.
37. A. Woo, S. Madden, and R. Govindan. Networking support for Query Processing in Sensor Networks. *Commun. ACM*, 47(6):47–52, 2004.