A Theory of Modularity for Automated Software Development (Keynote)

Don Batory
Department of Computer Science
University of Texas at Austin, Texas, USA
batory@cs.utexas.edu

Abstract

Automated Software Development (ASD) are technologies for developing customized programs automatically and compositionally from modules. The foundations of ASD are domain-specific algebras, where each program in the target domain maps to a unique expression. Algebraic identities are used to optimize programs automatically. In this keynote, I trace the history of ASD and present a general theory of modularity for ASD that follows from its tenets.

Categories and Subject Descriptors D.2.2 Design Tools and Techniques
General Terms design
Keywords module composition, categories, commuting diagrams, functors

1. Introduction

I have worked in modeling and modularity for almost 40 years. I started with the modular construction of extensible database systems [3], which is now an early example of *Software Product Lines (SPLs)* [8]. From there, it was short steps to the modular construction of *Domain Specific Languages (DSLs)* [7] and then to *Model Driven Engineering (MDE)* [33]. My current interest is automatically deriving high-performance software libraries [24] using a correct-by-construction approach. This keynote presents a theory of modularity that is appropriate for *Automated Software Development (ASD)*.[1]

Why ASD? I believe it to be a grand challenge in *Software Engineering (SE)*. One needs to master (at least) three distinct subjects:

1. **Domain**. You must be an expert in the target domain, e.g., *Dense Linear Algebra (DLA)*;
2. **Software Engineering**. You must be an expert in writing efficient programs for that domain; and
3. **Modeling**. You must recognize the atomic modules of software construction for that domain. What form that modules take (graph rewrites) may be very different from the form you have experienced in the past (mixin layers, aspects).

It is non-trivial to acquire and integrate all three areas of expertise. If you are lucky (and I have been on occasion), one person assumes all three roles. More typically, the required expertise is provided by a team of experts.

Modules for ASD must satisfy more constraints than typical modules (domain atomicity, composability, and reusability which I document later). But does this make the problem of module design harder? Honestly, I am not sure. I have found the constraints of ASD to remove degrees of freedom that only complicate solutions. I will leave the simplicity or difficulty of this form of modularity for others to decide.

A recent chat with a colleague revealed his thoughts and benefits of modularity, which I take as typical of SE practitioners today:

- Modules for the sake of modules are uninteresting,
- Modules are created to improve performance,
- Modules are created for adaptability,
- Modules are created for reasons of understandability, etc.

I was bewildered by a list of properties that modules *should have*, as it told me nothing about what modules *should be*, which to me was a more fundamental and interesting question. It is also a question that is, for many reasons, is difficult to answer:

- Goals for modularity may be application-specific;
- Different perspectives and different goals lead to different conclusions and different expectations;
- Our education imprints us to view problems in different ways;

[1] There is no Related Work section in this paper: the entire paper is a Related Work section. For space reasons, I limit citations to the most critical papers.

MODULARITY Companion'15, March 16–19, 2015, Fort Collins, CO, USA.
Copyright © 2015 ACM 978-1-4503-3283-5/15/03...$15.00.
http://dx.doi.org/10.1145/

- There is far too much emphasis on concrete thinking and too little on abstraction (which to me is what design is all about);
- There are pitfalls: we (including myself) tend to generalize from too few domains; and
- It takes time to understand and appreciate the viewpoints of others.

I could not have written this paper in my first 10 years, nor 20. Maybe 30. The contributions of this keynote are to:

- Review fundamental results on modularity that imprinted my world view of ASD;
- Explain concepts that are fundamental to ASD modules;
- Present technical results that led me to this position; and
- Sketch a general Theory of Modularity for ASD

from the perspective of decades of hindsight.

2. Future Software Development Paradigms

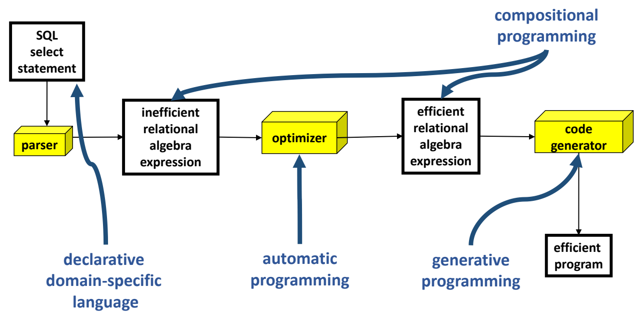
The 1980's were replete with prognostications on future paradigms for software development, particularly ASD. The primary four were:

- **Compositional Programming** – to develop software by composing modules;
- **Generative Programming** – to have programs generate programs;
- **Domain-Specific Languages** – to elevate languages to domain-specific notations; and
- **Automatic Programming** – to translate declarative specifications into efficient programs automatically.

It was clear back then that a simultaneous advance on all fronts was needed to make a significant impact. Astonishingly, an example of this futuristic vision was realized a decade earlier around the time when many AI researchers gave up on automatic programming. It was *Relational Query Optimization (RQO)* [27], which in my opinion is *the* most significant result in ASD. Ironically, RQO is rarely mentioned in typical textbooks and papers in SE, software design, modularity, product lines, DSLs, and software architectures, almost as if RQO never existed.

To refresh everyone's memory, Figure 1 illustrates the basic ideas. An SQL select statement is translated by a parser to an inefficient relational algebra expression. An optimizer applies algebraic identities to optimize the expression. A code generator translates the optimized expression into an efficient data retrieval (Java) program.

SQL is a prototypical declarative DSL. Relational algebra is an exemplar of compositional programming: relational operations are composed into expressions to define data retrieval programs. The code generator is a contribution to generative programming, and the optimizer is the key to automatic programming (more on this later).



compositional programming
SQL select statement
parser
inefficient relational algebra expression
optimizer
efficient relational algebra expression
code generator
efficient program
declarative domain-specific language
automatic programming
generative programming

Figure 1. Relational Query Optimization Paradigm.

RQO simplified database usage and revolutionized a fundamental area in Computer Science. It automated the development of data retrieval programs which were notoriously hard-to-write, hard-to-optimize, and even harder-to-maintain. Modules were operations, compositions of operations were expressions, each of which represented a unique program, and algebraic identities could be used to optimize programs/expressions automatically.

In short, RQO provided me a framework in which to think about ASD. It led me to make the following assumption that I believe is correct to this day: all domains have fundamental "shapes" or "modules" or "operations" from which their programs could be assembled. I illustrated this insight in my first tutorial on ASD in 1994 using Figure 2. In (a) I cut out of paper two shapes, a triangle and rectangle, and chose an attendee to use scissors to partition each shape (which represented the source code of a program) into modules. I might get (b) as a decomposition. Then I revealed that I wanted to build program (c), a parallelogram. By overlaying modules from each of the decomposed programs onto the parallelogram as in (d), it was easy to show that a lot of hacking would be needed to build the parallelogram using ad hoc decompositions. Then I showed (e) which demonstrated that a "domain-atomic" decomposition could enable the composition of pieces `x` and `y` of Figure 2e to build (c) in no time.

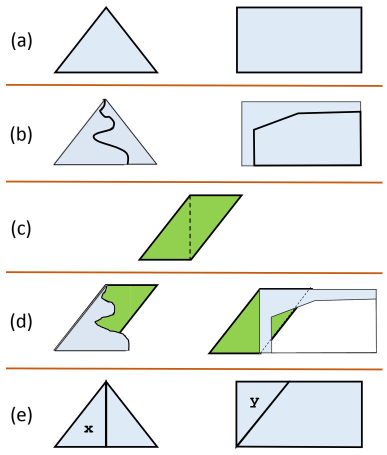


Figure 2. Fundamental Shapes in a Domain.

I viewed the process of *domain analysis* (the activity of studying a domain of programs to find its atomic shapes) to be the analog of creating an atomic theory in science. Starting with a set of disparate but representative programs (indicated by ○ in Figure 3), I abstracted to a theory of modular construction so that I could explain these programs in an elegant way by composing "atoms" (fundamental shapes/modules) and to predict new phenomena/programs (indicated by ●) that hadn't been seen or built before.

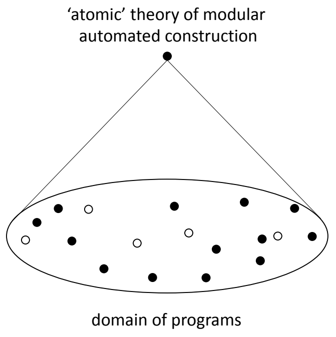


Figure 3. Domain Analysis

Further, RQO derived a set of semantically equivalent programs given a seed program. In Figure 4, programs that are equivalent to the initial program (the blue dot) are derived by applying algebraic identities, where an arrow $\mathtt{A} \rightarrow \mathtt{B}$ says program B is derived from A. That is, replacing equals with equals yields program B that is semantically equivalent to A.

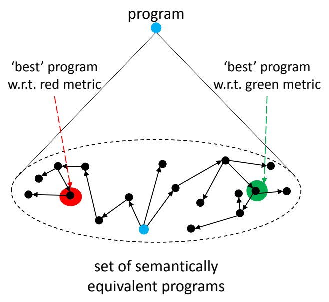

program
'best' program w.r.t. red metric
'best' program w.r.t. green metric
set of semantically equivalent programs

Figure 4. Deriving Semantically Equivalent Programs

Given an estimate of the performance of each program, RQO selects the "best". Using a one metric – say red – one program would be proclaimed the "best"; using another metric – say green – a different program may be selected (see Figure 4). Note that the graph of semantically equivalent programs derived from identities in Figure 4 remains invariant w.r.t. chosen metrics.

So how does RQO select the "best" program? Answer: It maintains different representations for each operation. So given a relational expression:

$$\mathtt{P} = \sigma(\mathtt{A}) \bowtie \sigma(\mathtt{B})$$

to estimate P's red performance, RQO composes $\text{red}_\mathtt{r}$ performance representations of each operation and relation [27]:

$$\mathtt{P}_\mathtt{r} \;=\; \sigma_\mathtt{r}(\mathtt{A}_\mathtt{r}) \bowtie_\mathtt{r} \sigma_\mathtt{r}(\mathtt{B}_\mathtt{r})$$

and evaluates $\mathtt{P}_\mathtt{r}$ to determine P's red efficiency. It composes $\text{green}_\mathtt{g}$ performance representations to estimate P's green efficiency:

$$\mathtt{P}_\mathtt{g} \;=\; \sigma_\mathtt{g}(\mathtt{A}_\mathtt{g}) \bowtie_\mathtt{g} \sigma_\mathtt{g}(\mathtt{B}_\mathtt{g})$$

And to produce the $\text{source}_\mathtt{s}$ code for P, it composes source representations:

$$\mathtt{P}_\mathtt{s} \;=\; \sigma_\mathtt{s}(\mathtt{A}_\mathtt{s}) \bowtie_\mathtt{s} \sigma_\mathtt{s}(\mathtt{B}_\mathtt{s})$$

To me, this was supremely elegant – granted that I understood/created this explanation only a decade ago. The symmetry that one finds in Nature was obvious in this explanation, so it had the right look and feel.

In short, my database upbringing imprinted me to think about ASD in terms of algebras, that "compositional" meant following the tenets of high-school mathematics – not any ad hoc means which often passes for gospel in SE literature [22].

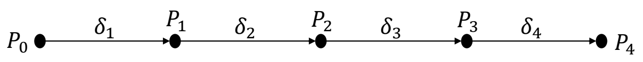
3. ASD Modularity Diagrams: Part 1

We are all familiar with UML class diagrams. They allow designers to express relationships among program entities, such as classes, interfaces, methods, and so on. These relationships are *declarative*: they can be implemented in *lots* of ways in Java, C#, C, and even assembly languages.

ASD uses different kinds of entities and relationships and also benefits from declarative diagrams. Figure 5 shows an ASD "modularity" diagram. A dot (●) represents a program or artifact. A series of changes ($\delta_\mathtt{i}$) is applied to $\mathtt{P}_0$ to produce $\mathtt{P}_4$. These deltas could be edits, refactorings, reformattings, etc. In normal programming, deltas are realized manually by programmers. In ASD, all deltas are automatically executed. For now, think of a delta as adding a module. So program $\mathtt{P}_1$ is produced from $\mathtt{P}_0$ by adding module δ_1. Given this description, we can formulate the relation between $\mathtt{P}_4$ and $\mathtt{P}_0$ as the expression:

$$\mathtt{P}_4 = \delta_4 \cdot \delta_3 \cdot \delta_2 \cdot \delta_1 \cdot \mathtt{P}_0$$

where the $\delta_\mathtt{i}$ are domain operations to be composed.[2]



$$\begin{tikzcd} P_0 \arrow[r, "\delta_1"] & P_1 \arrow[r, "\delta_2"] & P_2 \arrow[r, "\delta_3"] & P_3 \arrow[r, "\delta_4"] & P_4 \end{tikzcd}$$

Figure 5. Declarative Modularity Diagrams in ASD.

Let's use these ideas to examine a modular development of Section 2 of this keynote. Figure 6 shows its ASD modularity diagram. The dot on the far left denotes the null or empty section; the dot on the far right represents the completed section. The remaining dots represent intermediate stages of that section's development. An edge ($\mathtt{X} \rightarrow \mathtt{Y}$) represents a modular increment to X that produces Y.

"Design" is a noun: it represents the structure and relationship among formal elements within an artifact or set of artifacts. "Design" is also a verb: it is a sequence of steps to produce a (noun) design. Figure 6 represents both static and sequence relationships. A *design process* is a path connecting dots in Figure 6. I took the "high-road" by adding

[2] Even $\mathtt{P}_0$, a base document, can be considered a base operation – it has no parameters.

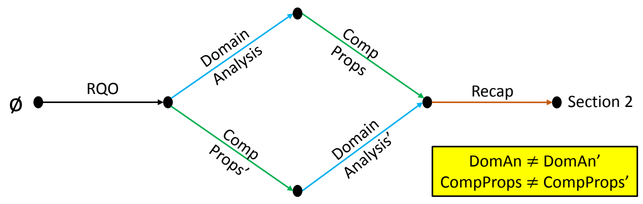
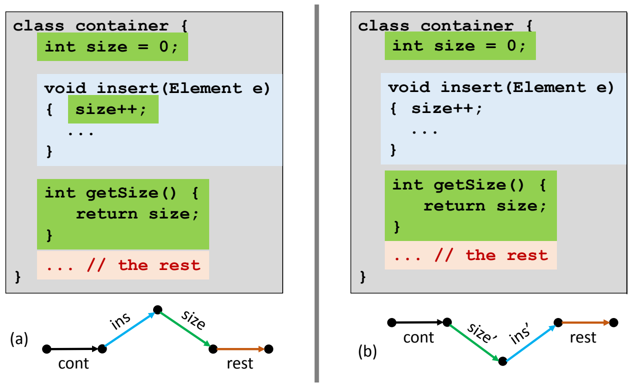


Figure 6. Modular Design of Section 2.

the module for `DomainAnalysis` before `CompProperties` when I created Section 2. But I could have proceeded in another way, using slightly different modules, first using `CompProperties'` and then `DomainAnalysis'`, to produce *exactly the same end result*. I see this modular structure all the time in software design [2, 33]. Here are examples.

Example 1. Consider the tiny code example of Figure 7a [1]. Starting with the null class, I first create a `container` class (the region in gray) [21]. Then I add a module that introduces an `insert()` method (the region in light blue). Next I add a module that maintains the size of a container (the regions in green): it introduces a method `getSize`, a `size` field and a modification of `insert()` to increment the container size when an element is added. This last/green module *must be aware* that `insert` exists to make the appropriate reparation of `insert`.[3] Finally, I add a module (orange) that supplies the remaining members.



```
class container {
  int size = 0;

  void insert(Element e)
  { size++;
    ...
  }

  int getSize() {
     return size;
  }
  ... // the rest
}
```

```
class container {
  int size = 0;

  void insert(Element e)
  { size++;
    ...
  }

  int getSize() {
     return size;
  }
  ... // the rest
}
```

Figure 7. A Tiny Code Example.

Figure 7b is another design. The difference is that I add a size module first (the regions in green), and then a module that introduces `insert()` (the region in light blue). Note that the last/blue module *must be aware* that container size exists to introduce the correct `insert` method.[3] Observe the green and blue modules in Figure 7a-b are *not* identical, but the *results of their compositions are indistinguishable* [1].[4]

Example 2. Now consider a larger example in Figure 8a. Starting with an empty code base, I introduce an IDE as a container of language-based tools. Next I add code bases for a compiler and AST[5] creation (outlined in blue, ignoring green files). Then I add a refactoring engine (green), which also modifies the AST codebase to permit AST manipulation. This refactoring/green module *must be aware* of the AST codebase to make the correct changes.[3] Lastly, I add a module (orange) that introduces remaining tools.

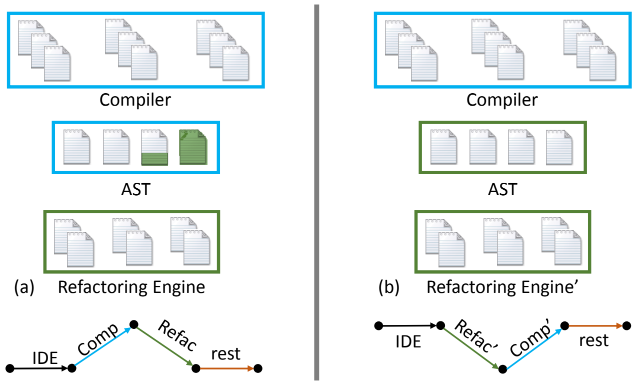


Figure 8. A Larger Code Example.

Figure 8b shows a different design. This time I had foresight when writing my AST module to include AST manipulation capabilities. So my initial module, `Refac'` (green), already has the needed infrastructure for later tools. Then I add my compiler (blue), which *must be aware* that the required AST codebase exists for it to work correctly.[3] Observe that the green and blue modules in Figure 8a-b are *not* identical, but the *results of their compositions are indistinguishable*.[6]

Example 3. A "theory" of modularity should not be limited to code. The artifact being created could be a latex file, powerpoint file, or pdf of my keynote. The ideas behind Figure 6 are quite general.[7]

The name given to the equality relationship in Figure 6 is a *commuting diagram*. A commuting diagram (Figure 9) says nothing about how modules are implemented – such details are abstracted away so that implementation is a parameter to the theory, as it should be. It also says nothing about when composition takes

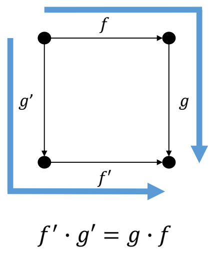


Figure 9. A Commuting Diagram

[3] In AOP-speak, the green module cannot be oblivious to the blue module in Figure 7a, and vice versa in Figure 7b. The same for Figure 8a-b. Compare with [17].

[4] It is possible to write two aspects that are commutative to produce Figure 7, but this *not* my point. I am interested in compositions of different modules that produce the same result.

[5] *Abstract Syntax Trees (ASTs).*

[6] To my aspect colleagues: I know you can define two aspects that are *commutative* whose composition yields the same as my examples. That is not the point that I want to make: namely, composing *different* modules yields the same result.

[7] These are examples of feature interaction, a phenomena well-known in the SPL literature [6, 10].

place: it could be at compile-time, load-time, or run-time; composition time is also a parameter to the theory, as it should be. But these diagrams say something important: all paths between two points/dots yield the same result. It is a statement of common sense: *there are many ways to modularly build an artifact.* Further, commuting diagrams define algebraic identities among compositions of different modules.

Incidentally, a "theory of modularity" which we are working toward requires us to separate concerns [13]. We should distinguish implementation techniques from their abstractions. It then becomes an interesting problem to decide when certain implementations are appropriate or not. We all know that no implementation technique is perfect for all situations; this too is common sense and our "theory" requires it.

4. ASD Modularity Diagrams: Part 2

Modularity is not just about code. Programs have many different representations, all of which should be consistent. Plato had the right idea: a program is abstract – a Platonic form [35] – and all that humans can perceive are its shadows (Figure 10). One shadow projects a Java representation; moving the light source, another shadow is cast to expose an HTML representation; yet another is a Java .class file representation; a fourth is an XML document (possibly containing configuration data); and a fifth is a Mathematica performance model of the program's run-time behavior.

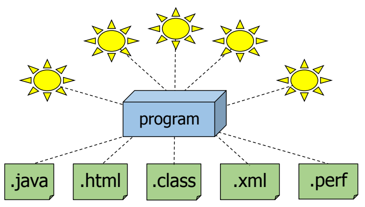


Figure 10. Multiple Representations of a Program

Further, program representations can be hierarchically decomposed. One such structure is shown in Figure 11a. An instance is shown in (b), where a Java program has a code representation (consisting of a pair of Java files), a makefile, a configuration file, and documentation (consisting of a pair of Word files). Another instance is (c), a client-server which has a client code base (a pair of C# files), UML and HTML artifacts, and a server code base (a C# and data file). In short, a module can contain any number of representations and decompositions, not just code. Equivalently, a module can contain any number of submodules. Our "theory" can express both.

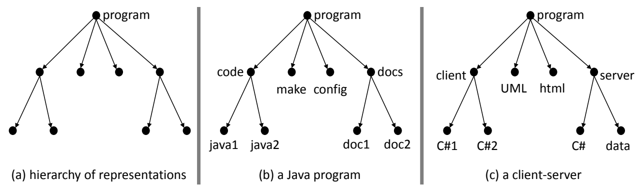


Figure 11. Hierarchy of Program Representations.

Let's return to the modularity diagrams of the previous section. We start with a program $\mathtt{P}_0$. Adding a module to $\mathtt{P}_0$ yields program $\mathtt{P}_1$, denoted earlier by $\mathtt{P}_0 \rightarrow \mathtt{P}_1$. But now, we know that $\mathtt{P}_0$ is a hierarchy of consistent representations. A module that encapsulates a semantic increment must update any or all of these representations lockstep, so that the corresponding representations of $\mathtt{P}_1$ are consistent too. Figure 12 shows this update as a set of red arrows – it is as if the hierarchy of $\mathtt{P}_0$ is "pulled forward" to the hierarchy of $\mathtt{P}_1$ through a modularized set of updates. Again, appreciate that an implementation of individual arrows is a parameter to our "theory".

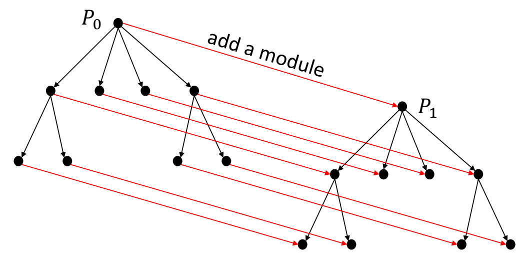


Figure 12. Modular Update of a Program's Representation.

Example 4. Remember RQO and its reliance on multiple representations of relational operations? Figure 13 illustrates the precise modularity relationships that RQO exploits.

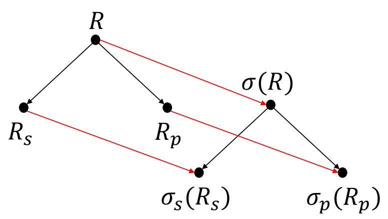


Figure 13. RQO Modules.

Example 5. In the early 1990s, Egon Börger (Humbolt Research Award 2007) developed *Abstract State Machines (ASMs)* as a methodology, formalism, and foundational theory for incrementally developing correct programs. He is a pioneer in modular incremental semantics.

We first met at a 1995 Dagstuhl seminar. And judging from each other's presentations, we knew that we were working on something similar. But we also knew that were were not ready to understand and appreciate each other's technical details or point of view.

We met next in 2006 at a Stanford workshop on the Verifying Compiler challenge. Egon would make a remark during the meeting, and I'm thinking: that's what I would say. And when I made a comment, Egon felt the same. That was the start of a very satisfying collaboration which continues to this day.

Egon, with Robert Stärk and Joachim Schmid, wrote the text *Java and the Java Virtual Machine*, referred to as the *JBook* [31]. They formally defined and proved correct an ASM version of the Java 1.0 compiler and (among other results) found errors in the Java 1.0 specification. The JBook presented a structured way to incrementally and modularly develop a grammar, interpreter, compiler, and bytecode JVM interpreter for Java 1.0. This was an enormous accomplishment.

They began with a sublanguage of Java expressions, the `Expr` program at the far left in Figure 14, with its grammar, ASM interpreter, ASM compiler, and ASM JVM interpreter as subrepresentations. One module was added at a time, incrementally extending the semantics of the language and its tools.[8] For example, module `Stm` added imperative statements, then module `ExpS` added static fields and expressions, then `StmM` added method calls and returns, and so on. Only *after* the definition of Java 1.0 had been constructed was a manual proof of tool consistency attempted.

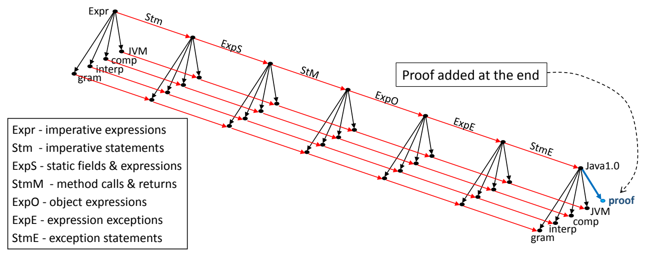

Expr
Stm
ExpS
StM
ExpO
ExpE
StmE
Java1.0
JVM
comp
interp
gram
Proof added at the end
proof
Expr - imperative expressions
Stm - imperative statements
ExpS - static fields & expressions
StmM - method calls & returns
ExpO - object expressions
ExpE - expression exceptions
StmE - exception statements

Figure 14. Development of the JBook and its Proof.

The "theory of modularity" spoke to us: the construction of the proof could be modular too [4]. Reason: a proof is just another representation (albeit complex) of the target program. And indeed this was the case (see Figure 15). A proof-of-correctness for the sublanguages could indeed be modularized and built as other representations. This was subsequently confirmed by Delaware [12] using the Coq theorem prover and by others (e.g., [32]).

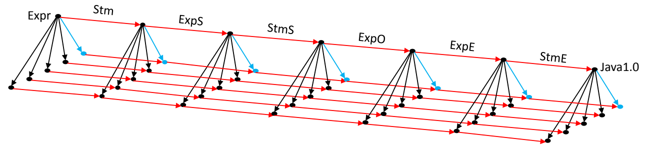


Figure 15. Modularizing and Composing Proofs.

5. My Path to Here

I approached ASD modularity starting from practice and working toward theory. I began with a simple idea, built it, reflected on what went right and wrong, was prepared to abandon hard-fought territory, and looped on this process.

At each step, I generalized what I knew and how I thought about ASD – the set of tenets that governed my world often collapsed into a smaller and more general core. Initial steps took ∼8 years, because (a) none of the ideas or implementations were obvious, (b) it involved one or more Ph.D.s, and (c) I had to re-learn what I previously understood from a broader context. Later steps took less time.

My work on ASD modules began with Genesis '82-'90. Inspired by Star Trek episodes and movies, the idea of composing plug-compatible "modules" to save the Universe seemed to be a good idea. Using standardized interfaces and modules that exported and imported these interfaces, I created a set of "legos" from which to construct customized *database management systems (DBMSs).*

Figure 16 illustrates how Genesis worked. From domain analysis, I defined standardized interfaces for fundamental programming abstractions in DBMSs. Figure 16a lists these interfaces as "types" `R` and `G`. Modules exported a single interface and imported zero or more interfaces. The signature `R` β(`G`) meant module β exports interface `R` and imports interface `G`. Signature `R` η meant module η exports interface `R` and imports nothing (other than making OS calls); η is a base module.

(a)

```
R α( G )
R β( G )
R κ( R )
R η

G γ( G )
G λ
```

(b) $P_b = \alpha(\gamma(\lambda))$

(c) $P_c = \beta(\gamma(\lambda))$

(d) $P_d = \kappa(\eta)$

Figure 16. Genesis ASD Modules.

Compositions of these modules implement programs that export a standardized interface. Expression Figure 16b defines program P_b that exports interface `R` and is a composition of three modules (α, γ, λ). Program P_c also implements `R` in Figure 16c and is another composition of three modules (β, γ, λ). Lastly P_d in Figure 16d implements `R` by two modules (κ, η).[9]

Genesis interfaces generalized Dijkstra's 1965 idea of *virtual machines (VMs)*. A VM was a collection of functions that expressed a particular abstraction; a VM at level $i + 1$ calls a VM at level i. I refreshed these ideas to define *Object-Oriented VMs (OOVMs)* as a set of Java classes and interfaces. Figure 17 shows two OOVMs: `R` has five distinct and related classes; `G` has two unrelated classes.

A Genesis module or *layer* was a package that translates objects and calls of the exported OOVM to objects and calls of the imported OOVM. Module α (or β) had the graphical depiction of Figure 18a – it exported `R` and imported `G`. This layer could have been a composition of two smaller layers $\alpha = x(y)$ in (b). The corresponding ASD diagram of

[8] The form of modularity Börger used matched that of AHEAD: introductions and wrappings of existing introductions.

[9] The connection of Figure 16a to a grammar, where sentences of this grammar are legal compositions of modules, did not go unnoticed. This is the origin of "GenVoca grammars" [8].

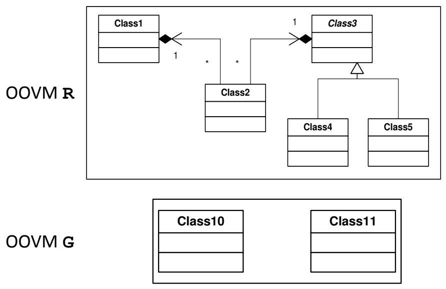


Figure 17. Object Oriented Virtual Machines.

this composition is (c) where, as before, each arrow adds a module. Figure 18 should look familiar to *Context Oriented Programming (COP)* researchers.

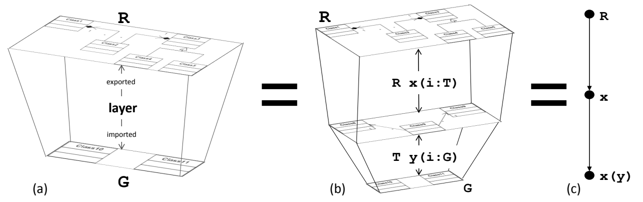


Figure 18. Layers and Layer Composition.

Modules that exported and imported standardized interfaces worked really well. Layers were increments in program/system semantics – and eventually were called *features*.[10] This was the first time I saw a tree of derivations that underlie software product lines: each node in Figure 19 is a particular Genesis DBMS, each arrow adds a module, and a path from ∅, the null or empty DBMS, to a particular node D_i was a composition of layers that defined the customized DBMS D_i.[11]

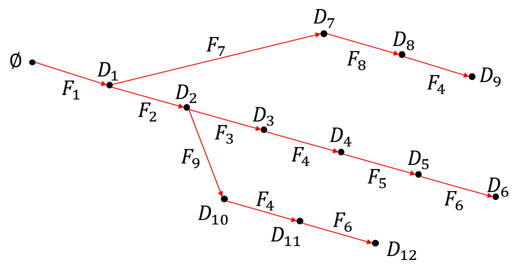


Figure 19. An ASD Diagram of a Software Product Line.

As I said, modules with standardized interfaces worked well, I needed more. I wanted to create customized classes from modules. This reminded me of the 1988 *"Programming by (Subclass) Differences" (PSD)* paper by Johnson and Foote [19].

Consider Figure 20. (a) shows a base class A. In (b), a feature's modifications to A are encapsulated in a subclass and – here's the trick – when the feature is composed, the name A is assumed by the subclass and the original class is no longer referenced. The addition of subsequently applied features in (c) and (d) show a repeat of these ideas (which is much easier illustrated visually). (e) shows this progression of changes in as an ASD diagram.

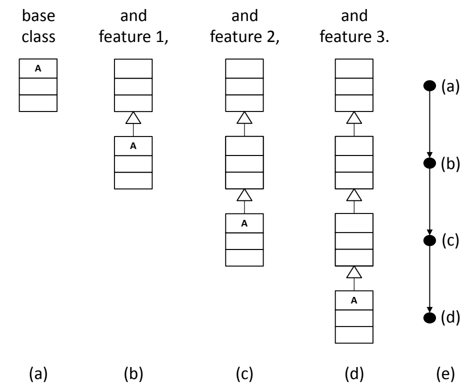


Figure 20. Programming by Subclass Differences (PSD).

This idea – which can be emulated by mixins [9] – must be coupled with the following trick, which was independently discovered by Smaragdakis [28] and Flatt, Krishnamurthi, and Felleisen [18]. I call the idea *mixin layers*. Consider Figure 21a. A base feature is a collection of classes; this particular base has classes A and B. A feature can introduce new classes and modify existing classes. Feature 1 introduces class C and modifies class B; feature 2 extends classes A and C and introduces class D; and feature 3 extends classes A, B, and D. The ASD modularity diagram of this composition is shown in Figure 21b. You should recognize this diagram as another example of Figure 12, where a hierarchy of changes/introductions of multiple files are modularized and composed.

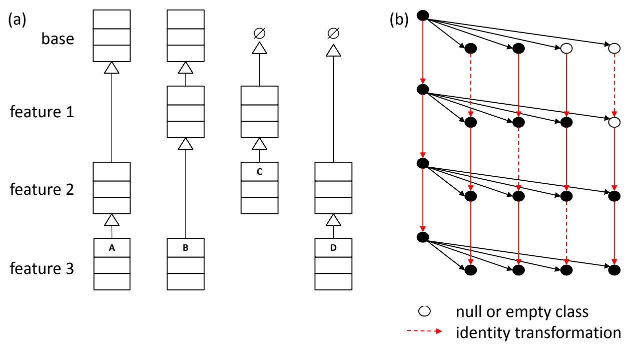


Figure 21. Mixin Layers Scale PSD.

In honestly, I did not recognize Figure 21b at the time mixin layers were discovered; this occurred years later with the advent of AHEAD, where my tools and ideas leapt from limited depth hierarchies of code files to arbitrarily-deep hierarchies and extensions of arbitrary files. It was the first

[10] It was not until 2001, almost 15 years later, that I first used "features" and "product lines" even though Kang used Genesis as one of the four exemplars to motivate his 1990 pioneering work on feature models [20].

[11] The feature model of an SPL encodes this graph as a set of rules.

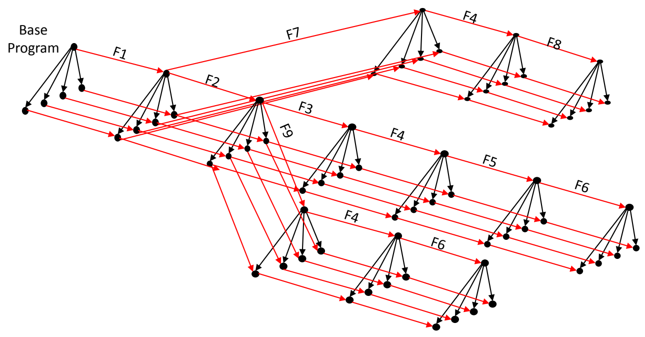


Figure 22. AHEAD Program Derivations.

time I visualized product lines as Figure 22, but I still had no "theory".[12]

The next advance broadened my view to include MDE. MDE is about creating models, integrating different models, and deriving models from other models. Figure 23a depicts a classical example that converts a state chart diagram into a set of relational tables, Java source code that implements the state chart is generated from these tables, and the source is compiled into bytecode. Figure 23b abstracts this process into an ASD diagram.

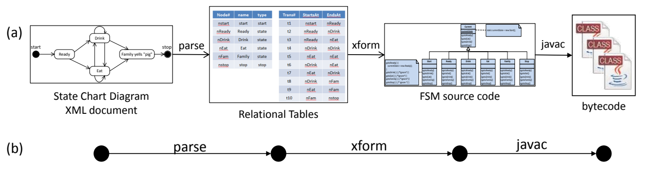


Figure 23. A Model Driven Application.

Look what happens when MDE is combined with SPLs: Figure 24a shows program $\mathtt{P}_0$ has four different representations – a state chart $\mathtt{SC}_0$, a database $\mathtt{DB}_0$, Java code $\mathtt{JV}_0$, and bytecode $\mathtt{BC}_0$. The green (vertical) arrows show how these representations are derived from each other.

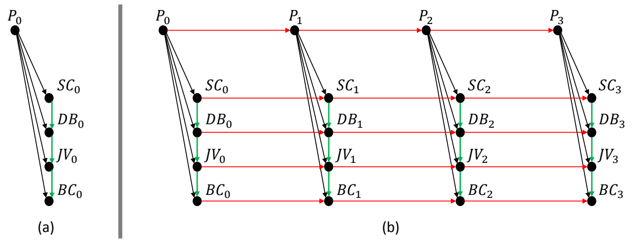


Figure 24. Model Driven Software Product Lines.

Figure 24b shows modular elaborations of $\mathtt{P}_0 \to \mathtt{P}_1 \to \mathtt{P}_2 \to \mathtt{P}_3$. If we treated $\mathtt{P}_0$ as the module $\emptyset \to \mathtt{P}_0$ (which is exactly what AHEAD did), modules are horizontal arrows and tools (which map one program representation to another) are vertical arrows.

Figure 25a shows this more clearly, where we see commuting diagrams galore. All paths from the upper-left dot ($\mathtt{SC}_0$) to the bottom-left dot ($\mathtt{BC}_3$) produce the same result. But from an engineering perspective, not all paths are equally efficient. When arrows are stretched to match their run-times, the rectangular grid of Figure 25a warps into Figure 25b, where the shortest path (a.k.a., *geodesic*) tells us the most efficient way to compute $\mathtt{BC}_3$ given $\mathtt{SC}_0$.

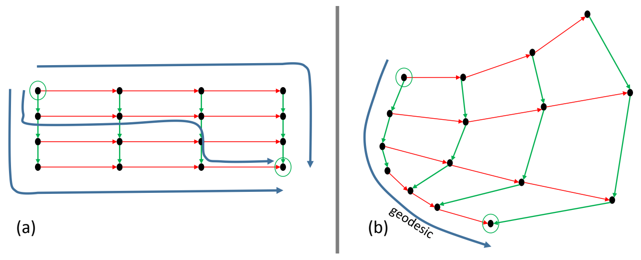


Figure 25. Commuting Diagrams and Geodesics.

Our "theory" predicts such situations would arise. Sure enough, we looked and discovered geodesics that improved program build times by 2-3× [33] and by 30-50× [34].

Armed with these ideas, I next explored correct-by-construction derivations of programs [5]. In particular, I looked at derivations of programs, where modules of of program construction were graph identities (i.e., replace graph $\mathtt{X}$ with an equivalent graph $\mathtt{Y}$). Such rules $\mathtt{X} \to \mathtt{Y}$ could always be expressed in one of two forms that are analogous to Java `implements` and `extends`: $\mathtt{X}$ is an interface and $\mathtt{Y}$ is a legal implementation of $\mathtt{X}$ *or* $\mathtt{X}$ is extended by $\mathtt{Y}$. The key to this approach is: if the starting specification graph is correct and each graph transformation that is applied is correct, the final (implementing) graph is also correct. Note that although modules in this universe are very different from modules that I discussed before, they are special implementations of arrows in ASD diagrams.

Let's look at the history of this line of work. Classical approaches (pre-1992) to formal software development start with a specification $\mathtt{A}_3$. A series of rewrites (identities) is applied to map it to code (an implementing program) $\mathtt{D}_3$ [23]. If the specification is correct and the transformations are correct, the resulting program is *correct by construction*. Figure 26a shows a derivation (path) from $\mathtt{A}_3$ to $\mathtt{D}_3$.

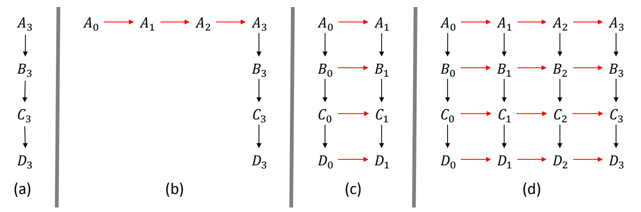


Figure 26. Formal Models of Program Development.

If programs could be developed incrementally – read "modularly" – wouldn't this also apply to specifications? Of course it should. Formal approaches, like Z [30], follow the design process of Figure 26b. To produce a complex specification $\mathtt{A}_3$, one starts with a simple specification $\mathtt{A}_0$. By (modularly) extending $\mathtt{A}_0 \to \mathtt{A}_3$ by adding one requirement at a

[12] Someone at ICSE'03 pointed out to me that AHEAD was related to categories. I didn't make the connection for another 4 years.

time, one can more easily understand $\mathtt{A}_3$'s design. Given $\mathtt{A}_3$, one then applies a series of implementing modules/rewrites, as before, to produce program $\mathtt{D}_3$.

It is well-known that deriving implementations of a simple spec is a lot easier than deriving implementations for a complicated spec. We observed that if a specification is too complicated, it is virtually impossible to derive an implementation: the vertical transformations (modules) are too complicated to explain and understand. This leads to an obvious question: Why can't *derivations* be modularly extended? Of course they can.[13] Figure 26c illustrates the mapping of derivation $\mathtt{A}_0 \rightarrow \mathtt{D}_0$ to its extended $\mathtt{A}_1 \rightarrow \mathtt{D}_1$, a graph that should now look familiar. In effect, we take the original derivation $\mathtt{A}_0 \rightarrow \mathtt{D}_0$ and incrementally (modularly) "pull it forward" to produce the desired derivation $\mathtt{A}_3 \rightarrow \mathtt{D}_3$. Doing so enables every step that we take to be small enough to be understandable and demonstrating correctness becomes easier [26]. We see this as a foundation for future interactive domain-specific program design tools.

We have found this approach to be indispensable for generating high performance Dense Linear Algebra libraries [24] and re-engineering complex legacy dataflow applications. We could not have done this without an intimate understanding of the relationships that our "theory" exposed.

6. A Theory of Modularity for ASD

Using the procedure of Figure 3 and the phenomena that I observed over decades to develop programs modularly, I recognized that a "Theory of Modularity for ASD" already exists: It is *Category Theory (CT)* [25]. The modularity diagrams that I've shown previously are diagrams of categories.

A *category* is a directed multi-graph[14] where a node, called an *object*[15], is a single artifact or a domain of artifacts. An *arrow* $\mathtt{A} : \mathtt{X} \rightarrow \mathtt{Y}$ is a total function from object $\mathtt{X}$ to object $\mathtt{Y}$. *CT says nothing about the implementation of* $\mathtt{A}$. An arrow is a statement that every $\mathtt{x} \in \mathtt{X}$ is paired with a $\mathtt{y} \in \mathtt{Y}$ (or in the case of individual artifacts, $\mathtt{X}$ is paired with $\mathtt{Y}$).

CT is governed by three rules or laws:

- *Arrows compose*: Given arrows $\mathtt{A} : \mathtt{X} \rightarrow \mathtt{Y}$ and $\mathtt{B} : \mathtt{Y} \rightarrow \mathtt{Z}$ there is an arrow $\mathtt{B} \cdot \mathtt{A} : \mathtt{X} \rightarrow \mathtt{Z}$;
- *Composition is associative*: $\mathtt{A} \cdot (\mathtt{B} \cdot \mathtt{C}) = (\mathtt{A} \cdot \mathtt{B}) \cdot \mathtt{C}$; and
- *Identity arrows*: Every object $\mathtt{X}$ has an identity arrow $\mathtt{Id}_\mathtt{X} : \mathtt{X} \rightarrow \mathtt{X}$, such that for every arrow $\mathtt{A} : \mathtt{X} \rightarrow \mathtt{Y}$, $\mathtt{Id}_\mathtt{Y} \cdot \mathtt{A} = \mathtt{A}$ and $\mathtt{A} \cdot \mathtt{Id}_\mathtt{X} = \mathtt{A}$.

We never talked about identity arrows, but that's OK: they are necessary for the mathematics behind CT and are typically no-brainers to implement (i.e., they do nothing).

[13] These derivations are substantially different than that of Börger [31] and Delaware [12]. In retrospect, they are instances of the same idea.

[14] A multi-graph allows multiple edges between the same pair of nodes.

[15] Not to be confused with "objects" in OO programming! CT was developed in the late 1940s, long before OO programming.

The theorems of CT are commuting diagrams. These are the fundamental relationships that exist among objects of a category. Here is its pragmatic meaning: if your implementation does not preserve these relationships, your implementation is wrong [33].

We used one other categorical concept in this paper. A *functor* $\mathtt{F} : \mathtt{C} \rightarrow \mathtt{D}$ is a mapping (basically an embedding) of category $\mathtt{C}$ into category $\mathtt{D}$ such that all objects in $\mathtt{C}$ are mapped to objects in $\mathtt{D}$ and all primitive and composed arrows in $\mathtt{C}$ are present in $\mathtt{D}$. That is: each object $\mathtt{c} \in \mathtt{C}$ maps to $\mathtt{F(c)} \in \mathtt{D}$ and each arrow $\mathtt{x} \rightarrow \mathtt{y} \in \mathtt{C}$ maps to arrow $\mathtt{F(x)} \rightarrow \mathtt{F(y)} \in \mathtt{D}$. Readers may have already recognized that functors were illustrated in 9 of the last 15 figures of this paper.

That's it. CT is the epitome of elegance, order, and power. Of course, CT is a *lot* more. But this is enough for a first lesson about categories.

7. Final Thoughts

There are many different ways in which an artifact (itself a module) can be decomposed into modules and recomposing them reconstructs the original artifact. Compositions of different modules can yield the same result; consequently these different compositions must be equivalences.

This paper presents logical conclusions that follow from this premise. Modularity is about science — a body of facts or truths systematically arranged showing the operation of general laws. Categories gave me a big picture perspective, not an in-the-trenches perspective, of what Modularity is about and how it and historical results fit together. It answers a fundamental question that interested me: What are modules? Answer: they are mappings that can assume any number of forms and implementations. Categories define properties that mappings and their compositions must satisfy.

It has been over 40 years since Codd proposed his relational theory of databases [11]. His Relational Model was based on set theory, which was panned in Computing Reviews [16]. To this day, it is not deep set theory, but something like the first few pages of a set theory text. The lesson here is that simple mathematics go a very long way. I use CT as a language, much like UML, to explain and define declarative relationships in modular development, not as a mathematical formalism. It provides a solid foundation about how to think about the nouns (objects) and verbs (arrows) of design. It gives me a framework to relate disparate phenomena with simple ideas. It tells me what is correct and what is not [22]. It can do the same for you, too, as it has for pioneers before me (e.g., [14, 15, 29]).

Acknowledgments. I thank Gary Leavens, Mark Grechanik, Dewayne Perry, Rui Gonçalves, and my graduate class (CS392F Automated Software Design) for their helpful comments. I gratefully acknowledge support for this work by NSF grants CCF-1212683 and CCF-1421211.

References

[1] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *GPCE*, 2008.

[2] D. Batory. Using Modern Mathematics as an FOSD Modeling Langauge. In *GPCE*, Oct. 2008.

[3] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An extensible database management system. *IEEE Trans. Softw. Eng.*, Nov. 1988.

[4] D. Batory and E. Börger. Modularizing theorems for software product lines: The jbook case study. *Journal of Universal Computer Science*, jun 2008.

[5] D. Batory, R. Goncalves, B. Marker, and J. Siegmund. Dark knowledge and graph grammars in automated software design. In *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2013.

[6] D. Batory, P. Höfner, and J. Kim. Feature Interactions, Products, and Composition. In *GPCE*, 2011.

[7] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *Proc. 5th Int. Conf. on Software Reuse*, ICSR, 1998.

[8] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1992.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP*, 1990.

[10] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 2003.

[11] E. F. Codd. A relational model of data for large shared data banks. *CACM*, June 1970.

[12] B. Delaware, W. Cook, and D. Batory. Theorem proving for product lines. In *OOPSLA/SPLASH*, 2011.

[13] E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[14] Z. Diskin and T. S. E. Maibaum. Category theory and model-driven engineering: From formal semantics to design patterns and beyond. In *ACCAT*, 2012.

[15] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2007.

[16] R. Elliott. Review: A relational model of data for large shared data banks. *ACM Computing Reviews*, Mar. 1971.

[17] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-oriented Software Development*. Addison-Wesley, 2004.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, 1998.

[19] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI-90-TR-021, 1990.

[21] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[22] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *PEPM*, 2006.

[23] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE TSE*, 18, 1992.

[24] B. Marker, D. Batory, and R. van de Geijn. Understanding performance stairs: Elucidating heuristics. In *Automated Software Engineering*, 2014.

[25] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[26] T. Riché, R. Goncalves, B. Marker, and D. Batory. Pushouts in Software Architecture Design. In *GPCE*, 2012.

[27] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD*, 1979.

[28] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *ECOOP*, 1998.

[29] D. R. Smith. Mechanizing the development of software. In *Client Resources on the Internet, IEEE Multimedia Systems 99*, 1999.

[30] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[31] R. Stärk, J. Schmid, and E. Börger. Java and the java virtual machine - definition, verification, validation, 2001.

[32] T. Thüm. *Product-Line Specification and Verification with Feature-Oriented Contracts*. PhD thesis, University of Magdeburg, 2015.

[33] S. Trujillo, D. Batory, and O. Diaz. Feature Oriented Model Driven Develop.: A Case Study for Portlets. In *ICSE*, 2007.

[34] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *IEEE TSE*, May 2010.

[35] WikiQuotes. Plato's theory of forms. `http://en.wikipedia.org/wiki/Theory_of_Forms`.