# Experience Report: Using tools and domain expertise to remediate architectural violations in the LogicBlox software base*

R. E. K. Stirewalt*, Spencer Rugaber†, Hwa-You Hsu†, and David Zook*

*LogicBlox, Inc.
Two Midtown Plaza, Suite 1880
1349 West Peachtree Street
Atlanta, Georgia, USA 30309
{kurt.stirewalt,david.zook}@logicblox.com

‡College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA 30332
{spencer,hhsu3}@gatech.edu

## Abstract

*When modeling the architecture of an existing software system, developers often find inconsistencies between the conceptual and the as-built architecture. To impose the conceptual view on the code often involves large refactoring to remediate architectural violations. This paper reports our experience applying large refactoring to remediate an architectural violation in LogicBlox, a large, multi-language multi-platform system. We used DSM-based analysis in conjunction with a suite of code analysis tools to identify and effect large refactorings. A key insight of this experience is the value of automatically generated proto-interfaces, which may help experts identify standard abstractions around which to structure the refactoring effort. We contribute a process for refactoring that includes the generation of proto-interfaces and the explicit inclusion of expert review.*

**Keywords:** Refactoring, Architecture, Experience Report.

## 1 Introduction

Many software systems are developed by teams who share design knowledge informally up to some point and then later decide to document this information. A common approach is to construct a model of the system's high-level architecture, usually in terms of "box and line" diagrams [16]. Unfortunately, attempts to construct these models often reveal a schism between the *conceptual architecture*, which emerges from conversations with developers, and the *as-built* architecture, which is inherently less well structured. Confronted with this schism, an organization may employ refactoring to remediate *architectural violations* that prevent the code from conforming to the conceptual architecture [11, 15, 1]. This paper reports our experience applying refactoring to remediate an architectural violation in LogicBlox, a large, multi-language multi-platform system.

Generally speaking, *refactoring* aims to improve code quality and stem the erosion of design in an aging software system [5]. A *large refactoring* is one that "requires more than one day to effect, impacts the development team and not just a single programmer, and which may require intermediate unsafe states" [14]. Bourquin and Keller argue that large refactorings have significant impact on the quality of a system and suggest that opportunities to apply them may be identified by looking for *architectural violations* [1]. Tools, such as Lattix [8] (the tool we used), apply dependence analysis to identify architectural violations using a representation called the *dependency structure matrix* (DSM) [15]. Starting with a diagram of the LogicBlox conceptual architecture, which we elicited through interviews with key developers, we used Lattix to create an explicit model of this architecture and then identified architectural violations to focus our refactoring effort.

In a DSM, an architectural violation manifests as a violation of a *design rule*, which is a programmer-specified assertion about the legality of dependence between two modules. To remedy such a violation, DSM theory provides *modular operations*, one of which, called *splitting*, aims to decouple two dependent design *parameters* (e.g., modules) by introducing a new parameter that constrains how the original parameters may interact [17]. Splitting typically involves the invention of a new interface and the subsequent refactoring of the two dependent modules to communicate with one another only through this interface. To support splitting, we developed a tool that computes a representation, called a

*This material is based on work supported by LogicBlox Inc, with additional support provided by the National Science Foundation under Grant Number CCF-0702667. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of LogicBlox or the National Science Foundation.

*proto-interface*, that codifies the essential dependency between two modules. This representation serves as a starting point in the design of the interface that is to be invented to effect the split, and we developed a tool, called `Slick`, to generate proto-interfaces automatically by analyzing compiled code.

To our surprise and delight, we found the `Slick`-generated proto-interfaces to be recognizably similar to well-known domain abstractions and roles from well-known design patterns [7]. Thus, by generating proto-interfaces, we uncovered standard and well-accepted abstractions around which to develop robust design parameters and rules. It was as if these good abstractions were there all the time, fighting to get out; the trick was uncovering and properly packaging them. Subscribing to the belief that good ideas tend to be reinvented, often poorly, it seems logical that legacy software could be rife with opportunities for uncovering and elevating good abstractions without requiring a massive rewriting of code. Moreover, once identified and elevated into first-class design parameters and rules, these abstractions could inform the refinement of the conceptual architecture into one that is more stable over time by virtue of the rich concepts upon which it is based.

Our experience led us to refine the prescriptions of existing processes for identifying and applying large refactorings (e.g., [10, 15, 14, 1]). Our refined process (Section 4) differs from those in the literature in two significant ways. First, we assist the developer in conducting split operations by means of a tool (`Slick`) that automatically infers proto-interfaces. Second, our process employs domain experts who review the generated proto-interfaces to look for salient patterns and/or domain abstractions with which to craft a design rule of high value to the enterprise. Here, we mean *high value* in the sense described by Sullivan and colleagues, who view modularization decisions in the framework of option investing [17]. While difficult to quantify, the assessment of the value of a refactoring, which Sullivan and colleagues refer to as the *net option value* (NOV) of a design decision, is useful in gaining developer acceptance and input prior to committing to the effort.

In the remainder of this paper, we describe how we applied this process to introduce design parameters that removed unwanted dependencies between two modules—`BloxLib` and `BloxLang`—in the LogicBlox code base. We present the process, the tools we used, and the lessons we learned from having conducted the work. While this experience represents but one data point, it serves to illustrate the value of automatically generated proto-interfaces and of expert review to uncover abstractions within existing code. In addition, we discuss a decision to decompose a large refactoring into multiple, smaller steps and the pros and cons of this decision. We feel similar issues and benefits might accrue for applying large refactorings to software in domains where a body of well-understood domain abstractions exists.

## 2 Background and related work

### 2.1 Refactoring

The term *refactoring* generally refers the application of fine-grained, semantics preserving alterations to source code in order to improve its design. The practice was popularized by Fowler's book [5] and has since become part of common practice as evidenced by the inclusion of refactoring capabilities into the Eclipse and Visual Studio software development environments. A survey of research into fine-grained refactoring is given by Mens and Tourwe [10].

Of more relevance to our work is the idea of coarse-grained refactoring, particularly at the architectural level. In their book, Roock and Lippert describe such refactorings as those requiring more than one day to effect, impacting the development team and not just a single programmer, and which may require intermediate unsafe states [14]. They introduce the concept of *architectural smells* to correspond to Fowler's *code smells* as indicators of potential refactorings. These include metrics-based indicators (Lines of code (LOC) per method, methods per class, etc.) as well as an extensive catalog of object-oriented (OO) design principles, the violations of which are signals of potential problems. Most relevant to our work is their recognition that dependency cycles are key indicators of candidate improvements and that they attempt to improve programs by introducing standard design patterns [7, 6].

Bourquin and Keller have applied many of the Roock and Lippert ideas in the case study of 140KLOC Java enterprise telecommunications application [1]. They point out the limitations of Roock and Lippert's suggestion to use architectural smells and code metrics to detect refactoring opportunities. Instead they suggest detecting architecture violations. The process is as follows: 1) model the architecture, having in mind a particular style, such as a layered architecture; 2) assign code units such as packages to components in the architecture; 3) compute intercomponent dependencies and compare them with the modeled architecture; and 4) concentrate the refactorings on differences between the modeled architecture and the computed dependencies. Like Roock and Lippert, the actual refactorings introduce design patterns. They validate their work by noting changes in code metrics such as reductions in class size and intercomponent cycles. Our approach differs in that we employ tools to assist in split operations and experts to identify domain abstractions during refactoring.

Fahmy and Holt have modeled software architectures as nodes (components) and arcs (containment or use de-

pendencies) [3]. Based on this representation, they delineate three categories of graph transformations: those intended for understanding, for analysis, and for modification. Transformations include lifting (raising dependencies into more abstract components), hiding, diagnosis (the inverse of lifting), sifting (for example, cycle detection), component movement and component splitting. The intent of this analysis is to better understand the space of architectural transformations.

Riva *et al.* present a process and a case study of architectural refactoring in the context of a software product line [13]. The process makes use of UML profiles. A UML profile is a set of extensions, typically stereotypes and Object Constraint Language (OCL) constraints that provide a coherent definition of a particular knowledge domain. In the case of [13], the knowledge domain is an architectural style. A profile defines the conceptual architecture, and the code determines the actual architecture. Problems arise when they do not match. The approach that the paper promotes adds an intermediate, graph-based representation. The graph facilitates the identification of violations, such as cyclical dependencies and supports the measurement of progress. Their case study identified the following deficiencies: incompleteness of profiles, inappropriate stereotype labelings, code sharing by team members leading to excessive dependencies, dependencies introduced to support non-functional requirements such as performance, and inappropriate allocation of functionality to components. Because our approach does not require the construction of a UML model of the architecture, our approach does not suffer from the incompleteness and inappropriateness of profiles and stereotypes. Moreover, we do not introduce dependencies to represent non-functional requirements.

Trifu and Reupke are concerned with tool support [19]. Two opportunities for automation arise: detection of flaws and application of refactorings. They primarily address the former by combining the metric and the code smells approaches to detection. In particular, they devise a set of heuristic metrics for detecting various smells and demonstrate their efficacy in a series of correlational case studies. They intend an Eclipse implementation of the approach.

With regard to detection, Pollet *et al.* have provided an extensive survey of tools and techniques [12]. The present the survey in terms of a taxonomy that includes the tools' goals, processes, inputs, techniques, and outputs. In all, on the order of 35 tools and over 140 papers are considered.

An important part of a successful architectural refactoring effort is how the effort is integrated into the normal development process. Lippert addresses the issue in the context of an agile software development process [9]. He suggests the use of *refactoring routes* and *refactoring plans* as support vehicles for the process. A route is a series of steps, and each step can be accomplished in a day. A plan is a depiction of a set of routes. He goes on to describe the mapping between routes and source code and how the process affects project planning.

Overall, the following abstract process can be described for architectural refactoring.

- Obtain or devise an ideal conceptual architecture
- Analyze the source code to determine its actual structure
- Compare the actual structure to the conceptual architecture to determine violations suggesting refactoring candidates
- Determine an appropriate design pattern to use that will address a high-priority violation.
- Refactor, test and iterate

The approach taken in this paper follows the ordering of tasks in this process. Specifically, the conceptual architecture was obtained by interviewing developers. We used the Lattix tool in conjunction with a tool we developed called `Slick` to perform the analysis. With the concurrence of the relevant developers a candidate violation/refactoring was selected. Several design patterns and domain abstractions were ultimately chosen as part of a target design. The code was refactored, tested and committed to the source code repository. Moreover, we refine the steps of this general process as described in Section 4.

### 2.2 Lattix

Lattix [8] provides a tool and methodology for defining, analyzing, and managing software architectures using *dependency structure matrices* (DSMs) [15]. The DSM representation allows for clustering a group of related (typically interdependent) design parameters into a single, composite parameter. When applied to C++-based systems, elementary design parameters typically correspond to individual C++ classes, and composite parameters typically correspond to groups of classes that are defined in the same subdirectory, C++ namespace, or both. Lattix automatically clusters classes into modules based on containment when these features exist in a code base under analysis. Here, and in the sequel, we limit our discussion to the C++ portion of the LogicBlox sources, which was the focus of our study.[1]

### 2.3 Slick

`Slick` is a suite of tools we developed for analyzing compiled C and C++ programs (i.e., ".o" files) to infer information about the structure of a software system and to

[1] Of the 1.5MLOC in the LogicBlox software base, over 840KLOC are written in C++.

$i := 0$
**while** $M_i \neq \varnothing$ **do**
$A := link(objects(M_i))$
$M_{i+1} = proto(A)$
$i := i + 1$
**od**

**Figure 1. Computing a hierarchy of candidate modules, rooted by some initial module $M_0$**

compute a variety of synthetic information that is useful in eliciting architectural models and performing large refactorings. `Slick` takes as input a model of the module structure of a given system, where each module comprises a collection of classes and/or functions, and a mapping that resolves a module–class or module–function pair to the location where the compiled C++ file associated with that class or function is stored. Using standard compiler tools, such as `g++` and `demangle`, `Slick` then builds a dependency database that maps entities in the model to the set of symbols directly used by that entity, as evident through an analysis of the compiled file(s) associated with the entity. By *symbol*, we mean the actual symbol used by the linker to refer to a element in an object file or link archive.[2] Data in this database may be queried using all the standard relational operators, e.g., selects and joins.

The idea of using compiled programs to infer the structure of a software system is not new (see, for example, [18]). `Slick`'s novelty owes to its use of these techniques to compute information that is useful in planning and conducting DSM-based modular operations (specifically the split operation) and in eliciting developer input on the feasibility and potential value of an operation. To support split operations, `Slick` computes an artifact, which we call a *proto-interface*, that captures the set of symbols provided by a given entity (module or class) and directly used by another. For example, if class A depends on class B, and the interface of class B provides three operations, $\{m_1, m_2, m_3\}$ but the methods of class $A$ refer only to operation $m_2$, then the *B proto-interface required by A* is $\{B :: m_2\}$. Proto-interfaces capture the fine-grain structure of a dependency and are thus useful in carrying out a splitting operation on the module structure of a system. Moreover, as we discuss later, these interfaces may trigger, in the mind of a domain expert, candidate abstractions that could be applied to promote the proto-interface into a robust and ultimately valuable design parameter.

In addition to proto-interfaces, given a C++ program that defines a function `main`, `Slick` can deduce a *candidate layering hierarchy* that minimizes the size (as in number of symbols) in each candidate layer. Fig. 1 lists our procedure for computing this hierarchy. Starting with module $M_0$, the procedure repeatedly creates a link archive containing only the object files associated with the elements of a given module $M_i$ and then computes the proto-interface to formulate a candidate module $M_{i+1}$. This procedure assumes that such a layering is possible and ignores any "back dependencies" such as might exist if a module at level $k$ contains an entity that depends on the module at level $k - 1$.

We use the candidate layering hierarchy as a first cut when decomposing a composite module into constituents in Lattix, which we can then analyze for back dependencies. In addition, these hierarchies are useful artifacts to bring to meetings with developers, whose surprise at the features that are needed (or not needed) in a candidate layer often raises issues that are salient to the refactoring effort but that are otherwise difficult to elicit in group discussions. Such discussion might dispose of a potentially complex constraint, e.g., when a dependence emanates from a top-level feature that clients no longer use. It might also inform the analyst with useful information regarding when to effect certain refactorings, for instance, when a candidate layer contains features that one or more of the lead developers expects to be modifying (or avoiding) soon. Such discussion also helps gain buy-in from developers, who can quickly see how code-level entities map to the nodes and arcs of high-level diagrams.

## 3 Subject of study

### 3.1 General introduction to LogicBlox

*LogicBlox* is a commercial platform for building enterprise-scale corporate planning and pricing applications, which feature analyses that require aggregation across very large (i.e., terabyte-scale) data sets, combined with simulation and modeling techniques. LogicBlox uses a Datalog-based language [2] to specify the structure and behavior of these models and simulations. The use of Datalog is salient to this paper because the architectural violation under study involves a backward dependency between the module that contains the Datalog compiler and type checker and the module that contains the engine that evaluates the Datalog rules. LogicBlox is implemented by a large collection of C++, Java, and Python code. As with many systems of comparable size, this software base has grown to incorporate new features over many years with teams of developers working concurrently.

Fig. 2 depicts the overall structure of the LogicBlox system. Conceptual layers are denoted by dashed boxes. At the top, various tools and applications are provided to the user to access LogicBlox functionality. The user interfaces

[2] When presented to human readers, these symbols are demangled via the `demangle` tool.

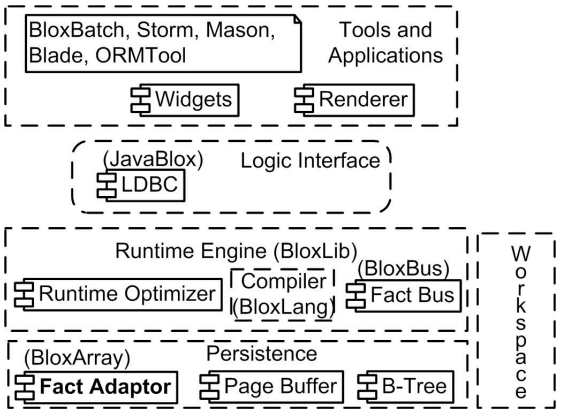


**Figure 2. Model of the LogicBlox architecture**

in these applications are implemented using a set of custom widgets and a rendering engine. Interface between the user-facing applications and the underlying computational and data resources is provided by the LDBC protocol. It is denoted with rounded corners because it was a work in progress at the time of this paper. Also, the notation "(JavaBlox)" denotes that the corresponding functionality comprises an object library (i.e., an as-built module in the LogicBlox software base) called `JavaBlox`.

The most complex and largest piece of LogicBlox is the runtime engine (RTE). Before the refactoring effort described in this paper, the RTE contained the Datalog compiler, an optimizer and the actual computational mechanism, the Fact Bus. The Compiler is enclosed in dashed lines to indicate that the refactoring described in this paper extracts it from the Logic Engine to become a layer on its own. In so doing, its functionality is enclosed in its own module, referred to here as BloxLang. A quick word on notational conventions: Here and in the sequel, we refer to as-built modules using a typewriter font (e.g., `BloxLib`) and conceptual modules using a Sans Serif font (e.g., BloxRTE, BloxLang). Thus, prior to refactoring, the Compiler is a conceptual module whose implementation is buried within the existing, as-built module `BloxLib`.

The lowest layer in LogicBlox provides physical persistence, making use of a variety of mechanisms. The Workspace component is depicted adjacent to both the RTE and the Persistence layer, because it is accessed by both. The component manifests as a large class with a complex API. One of the conclusions reached during the course of the refactoring described in this paper is that Workspace itself should be refactored.

As a subject of study, the LogicBlox code base is interesting because it is large (1.5 MLOC total) and realistically complex (e.g., built using multiple programming languages for deployment on multiple platforms). Moreover, multiple groups of developers (including researchers from academia) have been working for years to maintain and extend the software without the benefit of an explicit model of its architecture. We began to elicit an architectural model using a *hybrid approach* [20], whereby the model evolves based on a combination of facts extracted from the code and knowledge gained through discussions with developers.

### 3.2 Architectural violation

Having elicited a conceptual architectural model, we used Lattix to construct the as-built model, based on an analysis of the LogicBlox source code. Because the LogicBlox source tree contains an extensive collection of subdirectories, Lattix was able to automatically produce an as-built architecture with rich modular structure. That said, the as-built model was much more coarse than that of the conceptual model.

One of these as-built modules, called `BloxLib`, was notoriously large and unwieldy. Our refactoring effort focused on the decomposition of this module.

Before we could use the DSM to identify architectural violations, we had to manually pivot it to decompose the as-built modules into parts that correspond to the more fine-grain conceptual modules. This step required a significant amount of interaction with the developers and judgment by the analyst. Because `BloxLib` is so large and unwieldy, we focused our initial effort on the extraction of a conceptual module named BloxLang, which contains all of the classes and supporting artifacts needed to statically compile, analyze, and optimize Datalog programs. Thus, our refactoring effort can be viewed as extricating from `BloxLib` two new modules, BloxLang and BloxRTE where:

$$\text{BloxRTE} = \texttt{BloxLib} - \text{BloxLang}$$

We manually pivoted the DSM to represent this structure, as depicted in Fig. 3.

Fig. 3 illustrates the decomposition of `BloxLib` into BloxLang and BloxRTE and indicates the involvement of these two modules in a cycle of dependencies. Module BloxRTE appears as a large box that spans the cell range bounded on the top left by cell (9,9) and on the bottom right by cell (17,17). Module BloxLang is the small box in cell

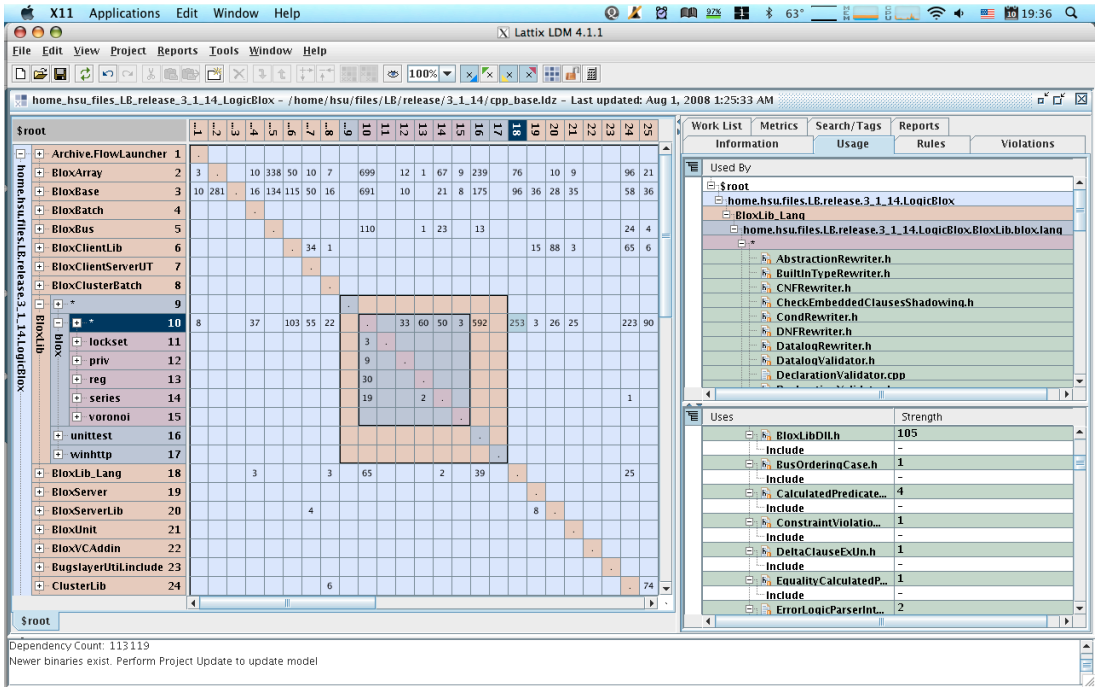


**Figure 3. Model of the LogicBlox architecture**

(18,18). The numbers in cells (18,10), (18, 14), and (18, 16) indicate BloxRTE dependencies on BloxLang. The number 253 in cell (10, 18) indicates the number of BloxLang dependencies on BloxRTE. These latter dependencies violate a design rule in the conceptual architecture. The DSM representation is useful for pivoting to form conceptual modules and for showing the existence of dependencies; however it does not suggest how to remediate these dependencies. To get this information, we used `Slick`.

Using `Slick`, we determined that the unwanted dependencies were as follows: Module BloxRTE contains a class called LogicEngine, whose instances compile and evaluate Datalog rules. LogicEngine creates an instance of a class called LogicProgram that creates and manages the data structures that are actually used to evaluate a rule. These data structures are instances of a hierarchy of classes rooted by an abstract class called LogicClause. This class hierarchy implements the guts of rule evaluation and heavily depends on a large number of other BloxRTE classes, which have nothing to do with Datalog compilation or semantic analysis. Thus, this hierarchy cannot live in module BloxLang.

LogicProgram instantiates objects in the LogicClause hierarchy by means of a Datalog compiler, which analyzes and translates the input program, performing a variety of semantic checks and optimizations in the process. The entry point to the compiler is a class called LogicCompiler, which is contained in module BloxLang. LogicCompiler works

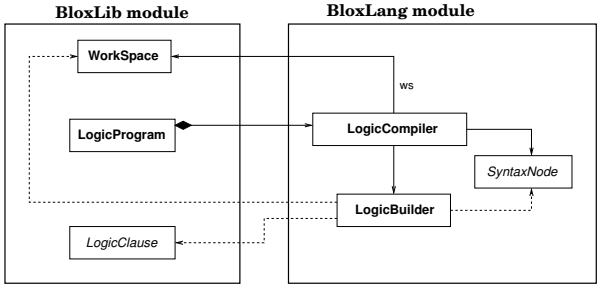


**Figure 4. Elided view of the cyclic dependency.**

by first analyzing and translating the input program into a Datalog abstract syntax tree (AST) representation, which is then traversed, using an instance of class LogicBuilder to construct LogicClauses. Unfortunately, to properly compile the input program and generate the execution structures, the compiler requires information maintained in the BloxRTE class WorkSpace, which we can think of as the in-memory representation of the database upon which a rule is evaluated. Clearly, class WorkSpace does not belong in module BloxLang.

The circular dependency indicated in the DSM can be more precisely articulated using the elided UML diagram in Fig. 4. Such diagrams model dependencies that cross the *seam* between two modules. These models must list classes

that are the source or target of a cross-module dependence; they may also depict *downstream* classes, i.e., classes that do not directly participate in a cross-module dependence. In this diagram, SyntaxNode is the only downstream class that is pictured. Class LogicProgram refers to class LogicCompiler, which lives in module BloxLang. At the same time, class LogicCompiler refers to class WorkSpace, and class LogicBuilder refers to classes Workspace and those classes that derive from LogicClause. Hence, a cycle of dependency, and a clear architectural violation.

## 4 Approach

Our process for identifying and effecting large refactoring is as follows: First, based on discussions with developers, formulate an initial model of the conceptual architecture. Second, use Lattix to identify architectural violations, which will take the form of unwanted dependencies between conceptual modules. Third, for each unwanted dependence, automatically deduce a proto-interface that represents the dependence minimally and at a fine grain based on the code's current structure. Fourth, and perhaps most important, consult experts to review each proto-interface to suggest appropriate design patterns or domain abstractions with which to refine and improve the design of the proto-interface. Fifth, analyze the impact of introducing a new design parameter based on the refined proto-interface(s). Finally, informed by well-reviewed interfaces and abstractions and with an eye toward the long-term value of the investment, conduct the low-level code refactorings required to adopt the new design rule, test, and commit.

This process differs from those in the literature in our use of tools to automatically infer proto-interfaces (Step 3) and in our use of domain experts to review and improve the generated proto-interfaces so as to formulate a well-conceived target design prior to refactoring. Experts need not be outsiders. In our study, the refactoring team consisted of two academic researchers and a graduate student and was embedded with the development group responsible for defining/extending the LB version of Datalog and writing its compiler. Thus, in this case, the developers were all experts in compiler design, as were some on the refactoring team. Moreover, the developers and the refactoring team collectively had significant practical expertise in the application of standard design patterns. The goal of expert review is to uncover any salient patterns and/or domain abstractions with which to craft a design rule of high value to the enterprise.

## 5 Performing the refactorings

Applying our process, we used `Slick` to generate the proto-interfaces required by BloxLang classes to satisfy

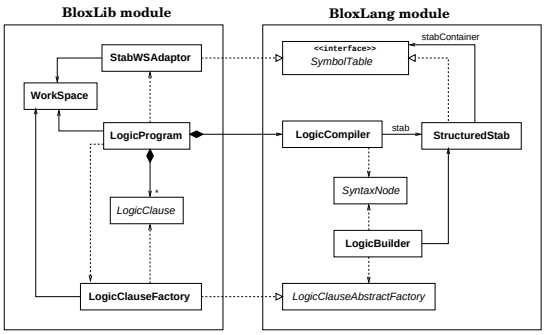


**Figure 5. Proposed refactoring**

backward dependencies (architectural violations) on classes in BloxRTE. Here, the first backward dependency was to class WorkSpace, and the second was to classes that derive from class LogicClause. Using the process we described in Section 4, we developed a plan to refactor a cluster of classes along the BloxRTE–BloxLang seam to remediate these architectural violations (Section 5.1). The design we proposed exploited two well-known abstractions—the notion of a *symbol table* (e.g., as in [4]) to aid in compilation and the use of an *abstract factory* to decouple the software responsible for constructing objects from the details of the classes of those objects [7]. Finding these abstractions helped to gain support from the developers and also helped in the planning of the refactoring itself. The proposal was discussed at length with key developers who suggested performing the task in two individual steps rather than all at once. Per this suggestion, we developed an alternative design, which removes the backward dependencies but does not realize the full power of the newly found abstractions (Section 5.2).

### 5.1 Initial refactoring proposal

Fig. 5 depicts our initial proposal for the interface between modules BloxRTE and BloxLang. The new interface class *SymbolTable* encapsulates the signatures in the proto-interface reported by `Slick`, with some minor changes in names to better conform to the symbol table abstraction, as it is developed in popular compiler texts (e.g., [4]). In this design, an instance of LogicCompiler receives and operates on an object that implements the SymbolTable interface rather than referring to an instance of class WorkSpace. By virtue of this new interface, one instance of the dependency from BloxLang back to BloxRTE will be removed, and the refactored code in class LogicCompiler and

its helpers will better reflect the use of standard compiler abstractions than was previously the case. To get the information needed to implement the SymbolTable operations, an instance of class LogicProgram constructs an instance of a new class StabWSAdaptor with the relevant WorkSpace and then passes this adaptor object to LogicCompiler, which uses it as a SymbolTable. As its name suggests, class StabWSAdaptor is a simple adaptor that implements its operations by delegation to the adaptee (in this case an instance of WorkSpace) [7].

The proposed design also calls for a concrete class, called StructuredStab, which implements the SymbolTable interface. This class implements a SymbolTable that can support *nested scopes* by delegating requests to lookup a symbol to another table representing a containing scope when the symbol cannot be resolved using the table that manages symbols for the scope at hand. A key insight, reflected in the new design, is that the old version of LogicCompiler (in `BloxLib`) was using WorkSpace to look up and check types against predicates that were defined in the environment in which a given Datalog program is being compiled. Thus, a Datalog program, which also introduces and uses predicates, constitutes a *block scope*, whose clauses are analyzed in the context of an *environment scope* that declares predicates that may be referred to in the program.

To address the second backward dependency from BloxLang to BloxRTE, we proposed to refactor the class LogicBuilder to create instances of the LogicClause hierarchy indirectly, by means of an *abstract factory* [7]. This design calls for a new concrete factory class called LogicClauseFactory, which implements operations in the LogicClauseAbstractFactory interface by creating instances of LogicClause classes. An instance of this concrete factory should be passed into LogicCompiler by LogicProgram.

## 5.2 Revised proposal

The developers liked the redesign proposed in Fig. 5 but were concerned that the task might be too large to perform in one step. Specific issues concerned:

1. the invasiveness of the dependency on WorkSpace, which might cause the seemingly simple refactoring to the SymbolTable interface to take longer than expected;

2. uncertainties regarding the impact of the refactoring on the extensive unit-testing framework used by the project;

3. potential difficulties performing the refactoring during an ongoing extension to some of the classes involved in the refactoring; and

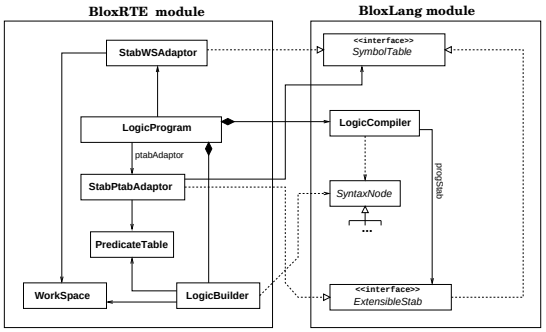


**Figure 6. Incremental proposal**

4. a preference for deferring the refactoring to the abstract-factory design by moving (perhaps only temporarily) LogicBuilder into BloxRTE, with the understanding that a subsequent refactoring might move it back into BloxLang

After much discussion, we adopted a plan that specifically addresses the first three concerns. For concern 1, we analyzed the code base using `Lattix`/`Slick` to find all uses of class WorkSpace within the collection of classes we wished to encapsulate in BloxLang. Of these classes, only two actually invoked WorkSpace operations; the others merely passed around pointers to usher an instance of WorkSpace to the objects that actually used it. In each of these latter cases, a simple substitution of declared type names—WorkSpace to SymbolTable—sufficed to effect the change. Of the two classes that actually invoked WorkSpace operations, one was LogicBuilder, which, thanks to the decision regarding item 4, was no longer a problem. The other class, called DeclarationValidator, contained much of the code that was to be moved into class StabWSAdaptor. However, this class also was the one being actively extended with new functionality (Concern 3). Concern 2 is well-known in the literature [21]. Fortunately for us, the LogicBlox tests are more akin to system than to unit tests and were therefore relatively neutral to the refactoring.

To address Concern 4, we decided to defer the refactoring to the abstract-factory design. This decision had a dramatic effect on the resulting design because moving LogicBuilder into BloxRTE required the inflation of the BloxLang interface with a host of classes that would have been hidden in our initial proposed design. Consequently, the number of dependencies across the seam increased as well.

Fig. 6 depicts the BloxRTE–BloxLang seam under the design that we finally adopted. Notice that class Log-

icBuilder resides in BloxRTE, and the BloxLang interface now exports the entire Datalog AST class hierarchy (depicted as an elided inheritance tree rooted by SyntaxNode) and a new interface class called ExtensibleStab. Despite the inflated interface, this design does remove the backward dependency.

## 6 Lessons learned

After assessing the size and extent of the refactoring proposed in Section 5.1, the developers strongly encouraged us to break the large refactoring into smaller pieces, each of which could be committed to the baseline, thereby minimizing opportunities for drift. Roock and Lippert observe that when a refactoring is decomposed into a series of smaller steps in this way, it may be necessary to introduce a seemingly redundant construct only to remove it in a later step [14]. They refer to this phenomenon as a *detour* and observe that, when viewed at the intermediate step of the process, the code may appear to have a worse design that it did at the start. Our detour resulted in the design depicted in Fig. 6. This design succeeds in isolating BloxLang from any dependence on BloxRTE classes. However, BloxRTE now contains two classes—StabWSAdaptor and StabPtabAdaptor—whose operations implement an abstraction that traditionally is thought of as "part of" the compiler. In the case of StabWSAdaptor, this concern is not major, as that class really just implements each operation by delegation to the WorkSpace. By contrast, the methods in class StabPtabAdaptor contain a significant amount of code specific to the implementation of a symbol table.

In addition to serving as a symbol table during compilation, class StabPtabAdaptor must construct a data structure called *PredicateTable*, which is consumed by the RTE. Class PredicateTable is, conceptually, a non-standard implementation of a symbol table, implemented at a very low level of abstraction. It is used by several downstream classes in BloxRTE; however, in theory, it should be completely replaceable by an object that implements the SymbolTable interface. That is, the RTE should, ideally, just use our symbol table directly, thereby obviating the need to construct a separate (redundant) data structure. However, this change was deemed to be beyond our purview: It was the responsibility of the run-time group, and we were working with the languages group.

Unfortunately, because the run-time group "owns" class PredicateTable, the language group was less experienced with its quirky API, and we had to invest lots of time learning how to recreate it properly from our symbol table. This work was clearly a detour in the Roock and Lippert sense. It could have been avoided, and the symbol-table functionality that now exists in BloxRTE via StabPtabAdaptor could have been located in BloxLang, if we had chosen to carry out our initial proposal rather than taking the detour. However, for this approach to have been effective, we would have needed to better understand how downstream classes in BloxRTE used class PredicateTable so that we could have properly refactored them to use a symbol table instead. Generally speaking, to attain such a level of understanding could require a significant investment of time if the abstraction is "owned" by another team, as was PredicateTable in this case. How to judge whether to opt for the larger refactoring or the series of detours is an open question.

Initially, much of the "compiler code" that touched classes WorkSpace and/or PredicateTable appeared in a class called DeclarationValidator, a *visitor class* [7] that traverses Datalog ASTs to do type checking and other semantic analyses. The refactoring we adopted can be thought of as introducing the SymbolTable abstraction to decouple the compiler from directly using WorkSpace and PredicateTable. Class DeclarationValidator was large and complex prior to the refactoring because it was consulting WorkSpace while visiting a Datalog AST and, simultaneously, creating the PredicateTable, which it also consulted during the visit. Following the refactoring, this class became much simpler because it used the SymbolTable, a well-known and robust abstraction that was designed to be applied for this purpose in just this way. However, the code that consulted the WorkSpace and constructed the PredicateTable did not disappear. Rather, it moved into the methods of the two adaptor classes, which now live in BloxRTE.

On some level, the simplification in BloxLang is being bought by moving complexity out to another module (and another development team). While difficult to quantify precisely, we view this exchange as a net gain because previously, the compiler was forced to build a non-standard, very low-level data structure that was needed by classes far downstream from it (i.e., in the RTE). Now, the compiler exports a strong and well-known abstraction (the symbol table) to the RTE. While this strong abstraction is currently implemented by translation to the weak abstraction, the BloxRTE developers now have an opportunity and an incentive to use the stronger abstraction in their downstream classes, perhaps eventually retiring class PredicateTable.

Another lesson concerns the obstacles to actually completing a refactoring when working concurrently with a team that is developing new features. In this scenario, there is constant pressure to wait until the new feature has been completely implemented and tested before committing the refactored solution. In our case, this led to a refactoring that was 90% complete but uncommitted and thus subject to its own measure of drift as features are added.

These experiences suggest that large refactorings are risky. Because they may involve changes to a large number of classes, they will take time to implement. The amount

of time is often difficult to estimate, and the longer an adaptive maintenance effort requires, the more it runs the risk of conflicting with a corrective or perfective maintenance effort. We experienced the latter while refactoring class DeclarationValidator. As we began to refactor DeclarationValidator, other programmers were actively extending it with new type-checking capability. Moreover, programmers working on other parts of the system were continually committing changes with new calls to the old API. Thus, before we could commit our refactoring to the baseline, we would need to synchronize our version with the current state of the baseline, and inevitably this would introduce compilation errors (due to the changes in the API) and/or bugs. This general lack of a stable base upon which to perform the refactoring extended our time to completion by several weeks.

Finally, our experience suggest the benefit of extending a DSM tool, such as Lattix, with the ability to generate proto-interfaces to support the application of modular operations. Proto-interfaces are also useful in calling to mind domain abstractions and patterns that lead to an improved design. For example, class WorkSpace declared operations such as hasPredicate and addPredicate, whose names suggest the analogous operations (e.g., hasSymbol or lookupSymbol and addSymbol) of a standard symbol table. Yet, because WorkSpace provides over a hundred operations and because the Datalog compiler was added to LogicBlox long after class WorkSpace was in existence, this seemingly obvious abstraction went unnoticed even when it was being used by programmers with expertise in compiler design. The virtue of the proto-interface is thus not what it contains but what it elides.

We continue to apply the process described in this paper to identify and effect refactorings to remediate architectural violations in LogicBlox.

## References


[1] F. Bourquin and R. K. Keller. High-impact refactoring based on architecture violations. In *Proc. of the* $11^{th}$ *European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.

[2] R. M. Colomb. *Deductive Databases and their Applications*. CRC Press, 1998.

[3] H. Fahmy and R. C. Holt. Software architecture transformation. In *Proc. of the IEEE International Conference on Software Maintenance*, pages 88–96, 2000.

[4] C. Fischer and R. J. LeBlanc. *Crafting a Compiler with C*. Benjamin Cummings, 1991.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.

[6] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison–Wesley Professional, 2002.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.

[8] The Lattix approach, Nov. 2004. http://www.lattix.com (accessed February, 2009).

[9] M. Lippert. Toward a proper integeration of large refactorings in agile software development. In J. Eckstein and M. Baumeister, editors, *Proc. of XP'2004*, number 3092 in Lecture Notes in Computer Science, pages 113–122. Springer Verlag, 2004.

[10] T. Mens and T. Tourwé. A survey on software refactoring. *IEEE Transactions on Software Engineering*, 30(2), Feb. 2004.

[11] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4), 2001.

[12] D. Pollet et al. Towards a process-oriented software arhitecture reconstruction taxonomy. In *Proc. of the* $11^{th}$ *European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.

[13] C. Riva, P. Selonen, T. Systa, and J. Xu. UML-based reverse engineering and model analysis approaches for software architecture maintenance. In *Proc of the* $20^{th}$ *IEEE International Conference on Software Maintenance (ICSM'04)*, 2004.

[14] S. Roock and M. Lippert. *Refactoring in Large Software Project: Performing Complex Restructurings Successfully*. John Wiley and Sons, 2006.

[15] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. of OOPSLA'2005*, 2005.

[16] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, New Jersey, 1996.

[17] K. J. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in software design. In *Proc. of ESEC/FSE 2001*, 2001.

[18] H. S. Teoh and D. B. Wortman. Tools for extracting software structure from compiled programs. In *Proc. of the* $20^{th}$ *IEEE International Conf. on Softw. Maint.*, 2004.

[19] A. Trifu and U. Reupke. Towards automated restructuring of object oriented systems. In *Proc. of the* $11^{th}$ *European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.

[20] V. Tzerpos and R. C. Holt. A hybrid process for recovering software architecture. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON'96)*, 1996.

[21] A. van Deursen and L. Moonen. The videostore revisited: thoughts on refactoring and testing. In *Proc. of* $3^{rd}$ *Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering*, 2002.