InfraRecord: Immediate Feedback on ORM Queries

Fabian Bornhofen, Lauritz Thamsen, Johan Uhle
Franziska Haeger
Hasso-Plattner Institute
March 15, 2013

***Abstract*—ORMs provide language-level abstractions for database queries, effectively relieving developers from expressing applications in multiple languages and from explicitly converting between objects and database entries. However, as ORMs do provide access to databases transparent in programming languages, queries are not as apparent as when expressed in SQL or other dedicated query languages. Especially which code queries the database and the mapping to actual database queries is not always obvious. Moreover, even if programmers recognize and understand ORM code, they still need database consoles to try their queries on actual databases when reasoning about correctness.**

We present *InfraRecord*, a development environment concept that integrates the functionality of an active database console with a code editor. It presents detailed information for given ORM queries and provides immediate feedback when such queries change. InfraRecord highlights ORM calls, presents the mapping to actual database queries, and previews result sets along with execution times. It, thus, helps programmers to interactively understand and develop ORM queries in a single integrated environment. ***Keywords*—InfraRecord, ORM, Database Queries, Tool Integration, Immediate Feedback**

I. Introduction

Business applications often rely on multi-tier architectures that separate presentation and application processing from data management [1]. With relational database management, developers often use SQL as dedicated query language and interface to the database while they express the logic of all other tiers often in general-purpose, object-oriented programming languages as, for example, Ruby [1]. In such application architectures object-relational mappers (ORMs) implement patterns that allow developers to program the database in the application language and convert data between the application layer and the database layer automatically [2]. This way, ORMs remove boilerplate code as well as repetitive programming tasks.

However, as ORMs do provide database abstractions on the level of the programming language, database queries are no longer easily distinguishable from code that resides on the application layer. Further, ORMs still interface the database internally with generated SQL queries, adding another layer of indirection. This mapping from ORM code to SQL queries happens transparently, though developers might have to understand it in certain situations, when database results appear ambiguous or manual query optimization is necessary. That is, while ORMs abstract database queries as well as automatically map domain objects to database entries, developers still need to know which code queries the data layer and maybe even which SQL is generated [3].

When retrieving and manipulating data, developers might require knowledge on which data is available and might want to try their code on actual databases even before running automated tests and certainly before subsequent deployment. Developers can copy their code from their editor to a database tool, but this approach results in long feedback cycles.

We present *InfraRecord* [2], an integration of a database console into a code editor. InfraRecord previews the results of database queries next to ORM code and during development. We implemented our idea as an extension of the Redcar editor [3] that assumes Ruby on Rails [4] projects, MySQL [5] database backends, and ActiveRecord [6] ORM queries. With InfraRecord, developers can see directly in Redcar which code accesses the database, which results are returned for the current data set, and how long query execution takes. Our extension provides immediate feedback on changes, enabling real-time debugging of ORM code in an integrated development environment.

The contributions of this paper are as follows:

- An approach to understand and develop ORM code interactively with information on database query results (Section IV).
- A prototypical implementation that integrates the functionality of an active database console into the Redcar editor (Section V-A).
- A parser component that indentifies potential database calls in Ruby on Rails code that uses the Active Record ORM (Section V-B).

The remainder of this paper is organized as follows. Section II gives a short overview of related work on immediate feedback in development environments. Section III highlights issues when using ORMs for database-backed applications and presents current development workflows. Section IV introduces our approach. Section V describes the implementation of our InfraRecord prototype, while Section VI discusses its limitations. Section VII evaluates both our approach and prototype. Section VIII outlines possible next steps and research areas, while Section IX concludes this paper.

[1]http://www.ruby-lang.org/

[2]http://github.com/lauritzthamsen/infrarecord

[3]http://redcareditor.com/

[4]http://rubyonrails.org/

[5]http://www.mysql.com/

[6]http://github.com/rails/rails/tree/master/activerecord, accessed 2013-03-10

II. Related Work

The idea of bridging the gap between a program's textual representation and it's state at runtime goes back to Lisp. Lisp lets programmers evaluate snippets of their code in an interactive shell called REPL (read-eval-print-loop). Meanwhile, various other programming languages such as Ruby or Python come with REPL shells [7].

Smalltalk-80 takes the concept further in that developers can evaluate text in any editor window (workspaces, source code browsers, debuggers etc.). This means that there is no need for compiling a text buffer or copying text into the REPL. The IDE would write the result directly back into the editor window. In particular it is possible to evaluate code in the context of an actual object, for example in a debugger session [4].

Bret Victor delivered a talk on "Inventing on Principle" [5] and later explained the ideas in detail in an blog post [6]. He wants programming environments to make code, control flow, and data more tangible. Not only should this enable novice programmers to pick up concepts more quickly, but also make experienced programmers more productive by reducing the need to mentally simulate programs. The examples in his talk are mainly graphical applications which can be visualized in a fairly straightforward way. One of them is code that draws a tree while the actual image of the tree is displayed next to the code. On each change in the code, the environment also highlights changes in the visualization.

Inspired by Bret Victor's ideas, Chris Granger proposed Light Table [7]. Light Table is an IDE for the Clojure programming language. Light Table is heavily inspired by Bret Victor's writings.

Microsoft Excel is a popular spreadsheet processor. Excel lets users model their calculations in a spreadsheet that automatically recalculates values once a cell has changed. This makes it possible to quickly evaluate a variety of values for different variables and immediately see the results.

Our vision is to work towards a similar experience for developers that write ORM-based source code.

III. Motivation

ORMs alleviate the conceptual and technical impedence mismatches that result from programming applications in an object-oriented language while using a relational database management system [8], [9]. However, understanding ORM code requires knowledge about the abstracted database queries. Further, developing ORM code might involve evaluating queries on actual data before publishing changes.

A. Understanding ORM Code

Distinguishing which code accesses the database and which code remains on the application layer is difficult with ORMs. This distinction is especially difficult with ORMs in dynamic programming languages such as Active Record in Ruby. As the following examples highlight, the distinction between Active Record code and ordinary application-layer Ruby calls is not possible without knowing the complete code base or investigating the behavior of particular code sections at runtime. For example, whether the following call triggers the database layer or remains on the application layer instead is not immediately obvious:

```
Customer.find(params[:id])
```

Even if developers know the interface of Active Record and the *Customer* class inherits from *ActiveRecord::Base*, the *find* method could also be overridden in the *Customer* class. In Ruby, this is even possible in just particular execution scopes depending on the call site [10]. That means reasoning about Ruby code that contains Active Record calls is difficult at design-time. Additionally, when developers are not familiar with the interface of a particular ORM, they might not immediately recognize which parts of their code query the database and which parts run only in the program runtime. Such confusions as, for example, emphasized by the following code snippet might impact performance characteristics:

```
Customer.all.count
```

This code fetches the complete *Customer* table and counts the resulting customer objects in the program runtime. The following alternative instead counts the number of entries in the database, without fetching any database rows and without creating customer objects in the program runtime:

```
Customer.count
```

Even though these two example snippets return the same result, the execution time of both snippets might differ significantly.

Both efforts, identifying which code snippets query the database at all and understanding how the code actually queries the database, could be reduced by showing the ORM queries and the mapping to generated SQL in an editor.

B. Developing ORM Code

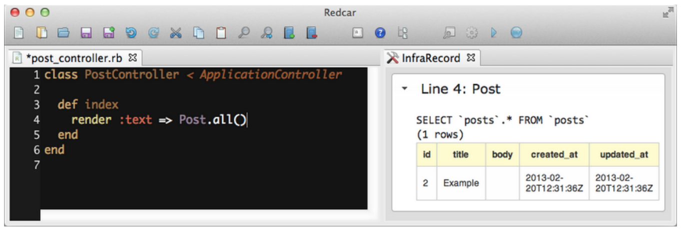
When developers design database queries, they have multiple options to reason about correctness and performance:

- evaluate their queries in active database consoles
- write and run automated tests for their specific queries
- build, deploy, and run the complete application to manually test through the application's user interface

Such manual or automated tests are best practices in software development [11], [12]. However, all three methods require developers to take additional steps to see the effects of their ORM code on the database and, thus, delay feedback. Copying their ORM code between database consoles and code editors as well as writing and triggering test cases during development impose not only manual and repetitive workflows on developers, but also involve context switches on the levels of different views or entire tools. Additionally, these approaches violate recommended practices of user interface design and information visualization, which advocate to show the actual data [13] interactively during development [6].

Applying these recommendations, a tool-based solution should present the actual query results along with additional information and with immediate updates on changes.

[7]http://www.paulgraham.com/diff.html, accessed 2013-03-13



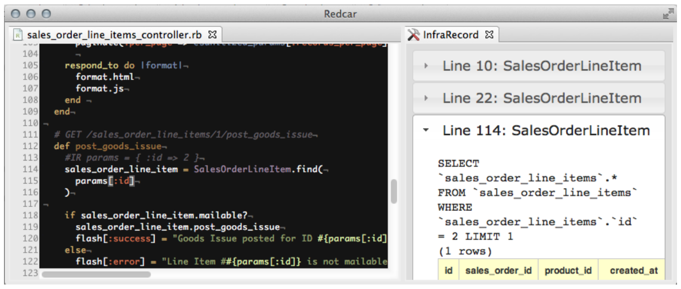
```
1 class PostController < ApplicationController
2
3   def index
4     render :text => Post.all()
5   end
6 end
7
```

Line 4: Post

```
SELECT `posts`.* FROM `posts`
(1 rows)
```

id	title	body	created_at	updated_at
2	Example		2013-02-20T12:31:36Z	2013-02-20T12:31:36Z

Figure 1. The InfraRecord prototype in the Redcar editor with Ruby on Rails code on the left and the InfraRecord Feedback Panel on the right



```
104         ¬
105     respond_to do |format|¬
106       format.html¬
107       format.js¬
108     end ¬
109   end¬
110 ¬
111   # GET /sales_order_line_items/1/post_goods_issue¬
112   def post_goods_issue¬
113     #IR params = { :id => 2 }¬
114     sales_order_line_item = SalesOrderLineItem.find(¬
115       params[:id]¬
116     )¬
117 ¬
118     if sales_order_line_item.mailable?¬
119       sales_order_line_item.post_goods_issue¬
120       flash[:success] = "Goods Issue posted for ID #{params[:id]
121     else¬
122       flash[:error] = "Line Item ##{params[:id]} is not mailable
123
```

Line 10: SalesOrderLineItem

Line 22: SalesOrderLineItem

Line 114: SalesOrderLineItem

```
SELECT
`sales_order_line_items`.*
FROM `sales_order_line_items`
WHERE
`sales_order_line_items`.`id`
= 2 LIMIT 1
(1 rows)
```

id	sales_order_id	product_id	created_at

Figure 2. The InfraRecord prototype in the Redcar editor with several queries. The InfraRecord Feedback Panel shows that three queries were found. The last query is currently displayed

IV. Approach

In Section III we identified two major challenges when working with ORM code: Understanding existing code and developing new code. These challenges originate from a disconnect between working with code in an editor and understanding it's behaviour at runtime. Common workflows, addressing this gap, include evaluating code in a REPL, executing automated tests or building, deploying and running the complete application for manual testing. All of these approaches force the developer to leave the editor and think in a different context.

ORM database queries tend to become complex and time consuming in execution. Furthermore, reasoning about about ORM code behavior is difficult since it interacts with a database and on a data set, that is not always visible to the developer in the context of the editor.

To enable a more integrated workflow for developers working with ORM code, we propose to provide immediate feedback inside a code editor next to the lines of code with ORM queries. Every time the developer changes the view or alters code, the information on that query is updated immediately. The developer does not need to switch to another tool anymore, since the feedback on the current line of code is always available in the editor.

To test this approach, we developed a prototype for Ruby on Rails and the Redcar editor. A screenshot is visible in Figure 1. In the left panel is the Ruby on Rails code. The developer works with the code as usual. On the right panel is the *InfraRecord Feedback Panel*. For each line that was identified to have an ORM query, there is a box that displays additional information to the query. In Figure 2 three queries are represented as boxes, with the two upper boxes collapsed and the third box expanded. For each query the following information is displayed:

- Line number
- Active Record model
- SQL query
- Result set
- Execution time to retrieve the result set
- Cardinality of the result set

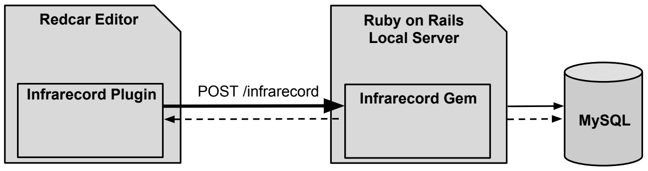


Figure 3. Overview of the architecture of the prototype

Following we describe the workflow in the prototype from a user's perspective: First, the user starts the Ruby on Rails server and the Redcar editor and opens a file of the Ruby on Rails project in the Redcar editor. The user can now edit the file as usual. The InfraRecord Feedback Panel can be opened by clicking on *Debug -> InfraRecord*. When the panel is open, it will display query information as mentioned before. The panel is updated each time the user changes code in the editor. Also, when the cursor in the editor enters a code line that includes an ORM query, the corresponding box in the InfraRecord Feedback Panel is highlighted.

A screencast shwoing the prototype can be seen online [8].

V. Implementation

In this section we give an overview of the architecture of the prototype, discuss how queries can be found in the source code, how variables in queries can be handled, and how eventually the queries are evaluated to gather results.

A. Overview

Figure 3 shows a diagram of the architecture. Our prototype consists of two major components, the editor and the server. The Redcar editor is where the programmer writes code. Inside the editor runs the InfraRecord plugin. The server runs an instance of the project that the programmer is currently developing. Ruby on Rails has a built-in code reloading mechanism, so we assume that all code recently saved by the programmer in the editor is dynamically reloaded by Ruby on Rails. The communication between editor and server happens via HTTP requests. To enable this we developed the Ruby on Rails InfraRecord gem [9]. It creates a route at *POST /infrarecord* that the editor's InfraRecord plugin sends query candidates to.

An InfraRecord roundtrip works as follows: When the editor plugin finds an ORM call candidate (as described in the following Section V-B), it sends it to the server via the HTTP route created from the InfraRecord gem. The gem extracts

[8]https://www.youtube.com/watch?v=dTJ9m_SdFIU, accessed 2013-03-13
[9]http://guides.rubygems.org/what-is-a-gem/, accessed 2013-03-13

the resulting SQL statement (if existing), collects profiling information etc. about it and sends it back to the editor. The editor displays relevant information in its Feedback Panel, which is rendered as an HTML/JavaScript view inside the editor.

We chose Redcar and Ruby on Rails because they are both written in the Ruby programming language, are open source and adapted well to our needs of rapid prototyping.

The current implementation was developed for Ruby on Rails 3.2.9 and Active Record 3.2.9 with the Active Record Mysql2 Adapter 0.0.3, which also limits the database to MySQL only. It is possible to extend the gem to support more versions of Ruby on Rails as well as other databases.

B. Client-Side Heuristic Query Detection

The Redcar editor plugin is responsible for reacting to changes in the source files that a user is editing. When a user changes a document by pressing a key or triggering a mouse event, the editor has to update the InfraRecord Feedback Panel. Since the editor plugin is ignorant of the internals of the Rails project that the edited source code belongs to, it has to get all feedback data from the server. Most importantly, the editor can only guess whether a certain piece of code contains an actual call to the database.

However, in order to limit the amount of communication between the editor and the server component, the editor preprocesses the source code before making queries to the server. We use a heuristic approach to decide whether or not a Ruby statement might or might not be an ORM call candidate. We send a statement to the server component only in case it is likely to be an ORM call.

In Active Record, the general pattern of database calls looks like this:

```
Model.method(param1, param2, ...)
```

In this case, Model would be the name of a model class and method is a class method that, for example, queries a number of instances of the model class from the database. For example, for a blog that has blog entries with titles, such a query might be typical:

```
BlogPost.find_by_title("my title")
```

In the current prototype, our focus is on `SELECT` queries rather than on `INSERT`, `UPDATE`, or `DELETE` operations. We discuss this decision further in Section VI-C.

In order to identify a call to an ORM model that causes a `SELECT` statement, the preprocessing step in the editor is to look for the above pattern in lines that changed during a mouse or keyboard event. In terms of the document's abstract syntax tree (AST), the editor identifies method calls on classes. On a change event, the editor would parse the document line by line and for each line (and its surrounding lines, in the case of a multiline statement) decide whether it is (1) a valid Ruby statement and (2) contains such a call. Only if a statement fulfills these conditions, the editor sends it to the server, collects feedback, and displays it.

There are two obvious limitations to this approach. First, the editor will detect a false positive whenever a statement contains a call to a class object that is not a model class. For instance,

```
Date.today
```

would be sent to the server and evaluated. Since it does not trigger a database query on the server, it would simply return an empty result.

The second case where the described approach fails is when there is a level of indirection between the model class and the actual call, like in the following example:

```
model = User
model.find_by_last_name("Doe")
```

The second line is a valid Ruby method call, but the editor does not know that the variable `model` in fact refers to a model class. As for our prototype, we neglect this case.

C. Server-Side Query Evaluation

To get feedback on a line of code, it has to be evaluated inside the runtime of the server. The InfraRecord server component is encapsulated by a Ruby gem. This simplifies integration into existing Ruby on Rails projects, since it merely means adding the gem in development mode.

When the Ruby on Rails server is started with the gem actived, a route is created that can be accessed from the editor with the following HTTP request (simplified):

```
POST /infrarecord
statement = "Page.find(params[:id])"
context = "params = {:id => 2}"
```

When the server receives such a POST call it runs through the following steps:

1) Extract potential ORM call from statement
2) Get SQL query for ORM call in context from patched Active Record
3) Execute SQL query against database to retrieve results
4) Return all gathered information as response to editor

In the last step, the server responds with a JSON-formated description of the results similar to this:

```
{
  "status" => "sql",
  "query" => "SELECT `posts`.* FROM `posts`
WHERE `posts`.`id` = 2 LIMIT 1",
  "column_names" =>
    ["id", "title", "created_at"]
  "rows"=>[2, "Post title", 1362996827],
  "possible_call"=>"Post.find(params[:id])",
  "model"=>"Post",
  "runtime"=>0.00071
}
```

Retrieving the SQL query is done by using the `eval()` method, which executes a string as a line of code in the context the `eval()` method was called in. If the evaluated statement includes an ORM call, during evaluation the `ActiveRecord::ConnectionAdapters ::Mysql2Adapter.execute` method will be executed. Due to the monkey patch in the InfraRecord gem, the SQL query is returned in an exception back to the controller method.

It has to be noted that using user-entered code from a (potentially public) route might be a security vulnerability. This is discussed further in Section VI-E.

D. Variable Bindings

For the vast majority of ORM calls in a code base, parameters to a query will be variables, not constants. In a typical controller in a project with a Page model, the show operation for a single page will look something like this:

```
def show
  @page = Page.find(params[:id])
end
```

Params contains the current request's parameters. One of them is the id of the page to be shown. Evaluating only the call statement

```
Page.find(params[:id])
```

in the context of the server without providing any value for params would result in an error.
We considered two possible solutions to this problem:

The first one is to parse the call statement and find out what variable and method names are unbound and prompt the user for the values that are to be used as test data for the query. The editor would then send the call statement along with a dictionary of names and values to the server. The server substitutes these names for their corresponding values before evaluating the statement.

The second one is a simpler mechanism that we implemented in the prototype. On detecting a possible call, the editor collects all preceding lines that start with "#IR". To the Ruby interpreter, these lines are comments.
The above example might be annotated with a params dictionary:

```
def show
  #IR params = {
  #IR   :id => 23
  #IR }
  @page = Page.find(params[:id])
end
```

Again, the editor sends the call statement to the server, but also a context string. The context string contains the Ruby statement that is extracted from the annotated lines, in this case

```
params = {:id => 23}
```

With this approach, the server component does not need to substitute any variable values. Instead, it concatenates the context string and the ORM statement and calls eval. In the previous example, the following code would be run:

```
eval(
'params = {:id => 23};
    Page.find(params[:id])'
)
```

The main disadvantage of this approach is that for InfraRecord to detect all ORM statements, a codebase must be thoroughly annotated.

Future versions of InfraRecord may combine value harvesting VIII-A with the variable substitution approach we described above. This would allow us to detect a large number of queries in codebases that are not annotated.

E. Laziness

For reasons of efficiency, Active Record does not materialize the result of each API call immediately. Consider the following statement:

```
cars = Car.where(:year => 2006).limit(5)
```

If Active Record loaded data eagerly, it would have to load all cars from 2006 from the database into memory and then return only five of them. What really happens is that it only issues a single SQL statement that contains a LIMIT clause.

This has an important implication on the query detection logic in InfraRecord. It means that when looking for message sends to models (see V-B), we need to detect the longest chain of calls.

For the editor this means that for the above example:

```
cars = Car.where(:year => 2006).
  limit(5)
```

it has to collect both lines and consider them as the complete statement. In the editor we only do some basic heuristic matching, so it might be save to send some more lines, if in doubt.

On the server, we extract the actual call from the statement using the RubyParser library [10]. In order to extract the complete ORM statement, we have to correctly detect the longest chain.

Our strategy is to first look for a CONST node N (a capitalized identifier) in the AST that receives calls. Starting from N, we include all CALL nodes that recursively wrap around it.

This is a simplified example using the Car model, leaving out all method arguments for brevity:

```
a = Car.where.limit
```

The AST looks somewhat like this:

```
(ASSIGN
  (VARIABLE "a")
  (CALL
    (CALL
      (CONST "Car")
      "where" NIL)
    "limit" NIL))
```

[10]http://github.com/seattlerb/ruby_parser, accessed 2013-03-11

In this simple example, we would walk the AST starting at the (CONST "Car") node until we find an outer node that is not a CALL node any more. Here, we would find

```
(CALL (CALL (CONST "Car") "where" NIL)
  "limit" NIL)
```

to be the longest chain of Active Record calls. The Ruby statement to be evaluated would thus be

```
Car.where.limit
```

VI. Limitations of the Prototype

In this section we will discuss some of the limitations of the prototype. We gauge their impact and propose potential solutions.

A. Limited Supported Technology Stack

Our prototype currently resides on a specific technology stack. It can be extended to support other technology stacks. Following is a list of currently used technologies and possible alternatives for each layer of the stack:

Technology	*In prototype*	*Potential Alternatives*
Web framework	Ruby on Rails	Sinatra, Django
ORM	Active Record	Datamapper, SQLAlchemy
Database	MySQL	Postgres, DB2
Editor	Redcar	Eclipse, Vim

It is important to note that our prototype was built using the reflective features of a dynamically typed language. We have not investigated possible implementations in other languages.

A web framework is not necessary. We used it because it is a common use-case to operate an ORM in and we utelize an HTTP route to enable the connection between editor and runtime environment (as described in Section V-C). Alternatives for enabling the connection between editor and runtime environment without a web framework are utelizing file sockets or loading the runtime environment directly in the editor.

B. Performance and Responsiveness

One of the goals of the prototype is to deliver immediate feedback to the user. A requirement of this is to have a user interface that responds consistently fast. In the current version, the performance is not fast enough to deliver compelling immediate feedback to the user. The request-response cycle from the user triggering a re-rendering of the InfraRecord Feedback Panel to the response being completely rendered might take over a second. Even worse, during that noticeable delay the Redcar editor is blocking, thus does not react to user input. This is due to the threading architecture of Redcar, in which every plugin runs in the main thread. Thus, while the InfraRecord plugin sends requests to the Ruby on Rails server and renders the response to the InfraRecord Feedback Panel, Redcar is blocking. This is especially visible when a user is typing fast as the InfraRecord plugin reloads and blocks on every keystroke.

Even though we implemented several optimizations like limiting the number of needed HTTP requests by frontend caching and trying to not re-render the InfraRecord Feedback Panel each time, we did not solve the performance problems comprehensively. Possible solutions are discussed in Section VIII-C

C. SELECT Queries Only

Our prototype only supports `SELECT` queries and ignores other query types like `INSERT`, `UPDATE`, `DELETE` or `ALTER TABLE`. We decided to focus on supporting `SELECT` queries in order to reduce the complexity of our prototype. In contrast to other query types, `SELECT` queries are safe, idempotent, and side-effect free. This is important because when delivering immediate feedback to the user, queries potentially have to be executed over and over again and should deliver consistent and predictable results. This can not be guaranteed if the queries may alter the underlying database or data set. To enable more queries in the future, un-safe queries could be wrapped in transactions and be rolled-back once all necessary information is collected.

D. Hidden Queries

As already discussed in Section V-B, our query parser only detects queries that are of following form:

```
Model.method(param1, param2, ...)
```

There are two major cases which InfraRecord's query parser misses:

1) Models assigned to variables: When a model is assigned to a variable, it will not be detected. This gets especially hard because in Ruby variables, functions, or constants may be defined outside of the scope of the current function as well as be altered dynamically at runtime. Thus, it is practically impossible to derive the correct assignment without being in that particular execution branch during runtime.

2) Relational queries on Active Record objects: Given that an Active Model object has been initiated and assigned to a variable, calling a relational query as, for example, `post.user` on it will execute a new SQL query (given that the relational model has not been loaded already via eager loading [11]).

Both cases are subject of future work as outlined in Section VIII-F.

E. Server-side Security

As noted before in Section V-C, evaluating code entered by users from a potentially public route in the `eval()` function might pose a severe security vulnerability. Therefore, the InfraRecord gem should not be deployed outside the local development environment, since it would allow attackers to

[11] http://guides.rubyonrails.org/v2.3.9/active_record_querying.html#eager-loading-associations, accessed 2013-03-13

execute arbitrary code on the server. To mitigate this risk, Ruby on Rails server used during development should not be accessible outside of the local environment. For the future it might be useful to implement more sophisticated access control mechanisms or to use a non-network method of inter-process communication.

VII. Evaluation

To evaluate our prototype, we interviewed three Ruby on Rails developers and observed them while using the prototype. These sessions took about half an hour each and were conducted in an unstructured way. All of the interviewees were students in the Master's program at the Hasso-Plattner-Institut Potsdam. They all described themselves as experienced Ruby on Rails developers.

All developers welcomed the functionality to get an overview over ORM queries in a file, especially when first exploring an existing project. When developing new ORM code, two of the three developers said they could imagine using the tool as a substitute to their current workflow of evaluating ORM code in a REPL. It would help them to spot syntactic errors as well as to verify results. As an example one developer mentioned that this would help him to choose the correct order key for a database query. Another developer mentioned how the tool could help programmers unfamiliar with the ORM to understand the ORM behaviour faster.

One developer raised concerns over the correctness of the ORM query detection. When we explained the limitations (as described in Sections VI-C & VI-D), the developer responded that he would not trust a tool with the mentioned blind spots and that this significantly decreases the usefulness of the tool to him.

One developer mentioned that revealing SQL queries is of low value to him, since he thinks less in terms of SQL and instead directly maps ORM code to the result. Thus, showing the returned result set and cardinality was of higher interest to him.

All developers were concerned about the low performance and inadequate responsiveness of the prototype. For all of them, this was a reason not to use the tool. We discussed this subject in Section VI-B.

One developer disliked the window layout of the prototype, calling it a "waste of screen real estate".

All developers said, they would be interested in an alert mechanism highlighting "bad" queries. Two developers raised concerns that a correct classification of "good" or "bad" queries might not be feasible. They stated that such a classifier would have to take into consideration the programmer's intent, which is impractical for a computer program.

VIII. Future Work

A. Parameter Harvesting for Variable Bindings

In Section V-D we described that in the current version, InfraRecord does not work well with existing codebases. The reason is that at this point we only give feedback on queries that are annotated with values for all variables that occur in the corresponding Ruby code. For example, we do not detect

```
@page = Page.find(params[:id])
```

because we do not know the value of params[:id].
One way of handling this would be to collect sample values for `params` from traces generated by test runs or by a production system. The approach would be analogous to the concept of type harvesting [14] by Michael Haupt et al.
InfraRecord could retain such typical values for all variables that occur in ORM calls. In an unannotated statement like the one above, InfraRecord could substitute params[:id] for an actual value before evaluating the statement and returning information about it to the editor. This could help users explore unknown codebases.
A feature like this would generally be desirable. InfraRecord might even be able to provide programmers with typical runtime values for all kinds of variables, not only those used in ORM statements.

B. Query Hotspot Detection

In the previous subsection VIII-A, we introduced the idea to use traces in order to populate values for variables. Bulding on that idea, traces could also be used to give developers better feedback on the performance implications of certain ORM queries. One indicator for the significance of queries could be how often a line of code was called, how often it failed, or how long the execution of the line's statements took. A scoring function could combine these three measures of significance and, subsequentely, help developers in spotting significant hotspots in their code.

This idea can only work if tracing data is available for queries of interest, which makes this approach more usefull for understanding existing code than for writing new code.

C. Asynchronous Execution

When we talked to other Rails developers (Section VII) about their expectations of an editor that provides immediate feedback, they emphasized that responsiveness is essential. Feedback might not have to be instant, but latency should be bearable and consistent. Also the workflow may newer be interrupted. In our prototype, when awaiting a server response, the Redcar editor has a blocked main thread and does not respond to user input.

To improve this, the editor plugin should run queries against the server by either using asynchronous I/O or by running the query in a background thread. Once the server responds to a request, the editor should dynamically populate the feedback panel without interrupting the user.

This becomes even more important when a database query takes more time to respond. In this case, it might be beneficial not to run a query on every single change in the source document, but to only run it after a cooldown period or number of file changes.

D. INSERT, UPDATE, DELETE

As pointed in Section VI-C, our prototype only detects `SELECT` statements on the database. The reason for this is that we expect these to have the heaviest impact on performance. `INSERT` and `UPDATE` operations on database rows could be detected in a similar way by looking for calls to Active Record's `save` or `update` methods.

For these operations it is important to find a consistent way of dealing with their side effects. When performing a `DELETE` operation on a row, it might be desirable from a user's perspective to only show the SQL string instead of running it against the database. By containing the side effect we would eliminate the need to run these statements in transactions that we might have to cancel so users do not lose their test data, either on purpose or accidentally.

While we think it is possible to conceive and implement such a model, we restricted our prototype to seeing data that is already in the database. For debugging purposes it might be worthwhile for developers to see what data they are storing, changing, or deleting as well.

E. Detecting ORM Code Smells

The motivation for InfraRecord focuses on the idea of helping developers to better understand ORM code. Opacity in the ORM layer makes novice users likely to write inefficient code since they might not know the exact semantics. In Active Record, a common code smell is this:

```
Model.all.select {|e| e ...}
```

This means that Active Record queries all objects of that model and the intended `SELECT` statement is executed in the Ruby process' memory. A more efficient way would be to only query those objects from the database that fulfill the relevant property and work with them.
This kind of mistake could happen either by accident or due to lack of knowledge. We assume that there is a number of such anti-patterns that are automatically detectable. Such an "ORM-lint" might give developers hints on how to improve their ORM code.

F. Finding Hidden Queries

As described in Sections V-B and VI-D, we are not able to detect all occurrences of database queries in the code. This is due to indirections between models and calls on them and due to abstractions by libraries. In Ruby on Rails, it is common to use external libraries that might query custom models as well. These are invisible to InfraRecord because of the library abstraction.

A solution addressing this issue might work similar to the harvesting VIII-A approach. Using traces from real program executions InfraRecord might find these hidden queries as well.

G. User Interface Improvements

In the evaluation section VII, we have shown that the usefulness of "immediate feedback" strongly depends on well-designed user interfaces. In our prototype, the focus was not on user interface aspects.

From our interviews we concluded that the most important improvements are to make the editor more responsive VIII-C and to make the feedback panel more configurable so it takes less space of the screen.

Drawing inspiration from Bret Victor's work [6], we would also like to highlight changes to the code over time in a more tangible way. To us, "Show comparisons" and "Make time tangible" would mean to not just re-render the feedback panel after each user interface event, but also to highlight the changes to the previous data. This could be done using animations or time lines with slider elements to navigate through different versions of code and feedback information.

IX. Conclusion

With InfraRecord, developers get feedback on their ORM code immediately. For each ORM query, our tool provides the generated SQL query, a preview and the cardinality of the retrieved result set, and basic profiling information. As this information is presented next to the Ruby code in the editor and updated immediately on changes, this approach reduces the effort of otherwise necessary context switches. Instead it integrates the application layer with database layer and, thereby, allows real-time debugging of ORM code.

In the future, we want InfraRecord's parser component to be able to detect all ORM queries, the query execution component to run queries asynchronously and in reversible transactions, and the user interface to be less obstrusive. Other features would be interesting to research. Those include detecting query hotspots in application code, providing scope for execution of parametrized queries through runtime tracing, and revealing ORM code smells during development using static or dynamic analysis.

Nevertheless, our InfraRecord implementation already helps developers to understand the effects of their code on actual databases without leaving their editing environment.

References

[1] A. O. Ramirez, "Three-Tier Architecture," *Linux Journal*, vol. 2000, no. 75es, July 2000.

[2] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[3] Martin Fowler, "OrmHate," May 2012, retrieved March 6th 2013. [Online]. Available: http://martinfowler.com/bliki/OrmHate.html

[4] Goldberg, Adele, "Smalltalk-80 The Interactive Programming Environment," 1984.

[5] VIctor, Bret, "Inventing on Principle," 2012, retrieved March 11th 2013. [Online]. Available: http://worrydream.com/#!/InventingOnPrinciple

[6] Victor, Bret, "Learnable Programming: Designing a Programming System for Understanding Programs," September 2012, retrieved March 6th 2013. [Online]. Available: http://worrydream.com/LearnableProgramming/

[7] Granger, Chris, "Light Table - a new IDE concept," April 2012, retrieved March 10th 2013. [Online]. Available: http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/

[8] Neward, Ted, "The Vietnam of Computer Science," June 2006, retrieved March 6th 2013. [Online]. Available: http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

[9] C. Date, *An Introduction to Database Systems*, 8th ed. Addison-Wesley, 2003.

[10] Akai, Shumpei and Chiba, Shigeru, "Method Shelters: Avoiding Conflicts Among Class Extensions Caused by Local Rebinding," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. ACM, 2012, pp. 131–142.

[11] Beck, Kent and Gamma, Erich, "More Java Gems." Cambridge University Press, 2000, ch. Test-infected: Programmers Love Writing Tests, pp. 357–376.

[12] Beck, *Test Driven Development: By Example*. Addison-Wesley, 2002.

[13] E. R. Tufte, *The Visual Display of Quantitative Information*. Graphics Press, 1986.

[14] M. Haupt, M. Perscheid, and R. Hirschfeld, "Type Harvesting: a Practical Approach to Obtaining Typing Information in Dynamic Programming Languages," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. ACM, 2011, pp. 1282–1289.