# Compreensão de Programas Apoiada por uma Linguagem de Consulta em Código Fonte

**Denis P. Pinheiro**[1], **Rodrigo G. Ribeiro**[1], **Ademir A. Oliveira**[3]
**Marcelo A. Maia** [2], **Roberto S. Bigonha**[1]

[1]Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG))
Belo Horizonte – MG – Brasil

[2]Faculdade de Computação (FACOM)
Universidade Federal de Uberlândia (UFU)
Uberlândia – MG – Brasil

[3]Google Inc.
Belo Horizonte – MG – Brasil

{denis,rribeiro,bigonha}@dcc.ufmg.br, marcmaia@facom.ufu.br

***Resumo.*** *A manutenção de sistemas é uma etapa do processo de desenvolvimento de software que consome grande esforço por parte dos desenvolvedores, elevando o custo total do projeto. Durante este processo, uma etapa de compreensão da implementação do sistema é exigida. Assim, ferramentas que suportam esta etapa podem facilitar o trabalho do desenvolvedor, diminuindo o esforço gasto. Este artigo apresenta uma linguagem de consulta em código fonte baseada no conceito de tabelas, que opera sobre uma representação sintática do programa acrescida de semântica estática (escopos, declarações e usos de símbolos). Como validação, é apresentada uma aplicação desta linguagem de consultas na atividade de compreensão de programas.*

***Abstract.*** *Software maintenance is an stage of software development process that consumes great developers effort, raising total cost of the project. During this process, a comprehension task of the system implementation is required. Therefore, tools supporting this proccess may facilitate the developer work. This paper shows a query language for source code based on relational concepts which works on a syntactic representation augmented with static semantics of the program (scopes, declarations and symbols use). It is presented, as validation, how this source code query language may be applied in the program comprehension activity.*

## 1. Introdução

No processo de desenvolvimento de sistemas, grande parte do esforço gasto por uma equipe de desenvolvimento é com atividades de manutenção e evolução. Durante o ciclo de vida de um sistema computacional, diversos desenvolvedores trabalham na manutenção deste sistema e, na maioria das vezes, são pessoas diferentes daquelas que o desenvolveram inicialmente. Assim, para realizar a manutenção de um sistema, às vezes desconhecido, um passo de compreensão do código fonte é inevitável.

Segundo von Mayrhauser e Vans [von Mayrhauser and Vans 1995], compreensão de programas é um fator determinante para se prover efetividade na manutenção de software e possibilitar a evolução de sistemas computacionais com sucesso. Porém, a tarefa de compreensão de um sistema torna-se cada vez mais complexa quanto maior for o sistema e menos documentação atualizada estiver disponível.

Tentando minimizar o problema, tem-se desenvolvido diversas pesquisas em torno da atividade de compreensão de programas e diversas teorias e ferramentas estão sendo desenvolvidas para suportar tal atividade. Em seu trabalho, Storey apresenta um estudo sobre as teorias que envolvem a atividade de compreensão de programas[Storey 2005]. São apresentadas também uma série de ferramentas de compreensão, onde são apontadas diversas características que tais ferramentas possuem e uma projeção de características que futuras ferramentas de compreensão deverão possuir. A capacidade de realizar consultas no código fonte foi apontada como uma das principais características que as ferramentas deverão possuir para possibilitar a compreensão da implementação de grandes sistemas computacionais.

Segundo Storey [Storey 2005], ferramentas de busca, tais como Google, apresentam-se como promessa para possibilitar buscas avançadas por componentes relevantes, partes de código e códigos relacionados em sistemas computacionais. Existem na literatura algumas abordagens de compreensão baseadas em consultas, podendo-se citar como exemplo REFINE [Kotik and Markosian 1989], ASTLOG [Crew 1997], PQL [Jarzabek 1998], GENOA [Devanbu 1999], JavaML[Badros 2000], GUPRO [Ebert et al. 2002], JQuery [McCormick and Volder 2004], etc. Porém, resultados definitivos ainda não foram alcançados.

A linguagem de consultas SCQL (*Source Code Query Language*), definida e implementa inicialmente por Oliveira [Oliveira 2004], é baseada no conceito de tabelas e opera sobre o modelo puramente sintático do ambiente de meta-programação MetaJ [Oliveira et al. 2004]. O principal componente de MetaJ é a *referência-p*[1], que encapsula a AST de um programa escrito em qualquer linguagem de programação, desde que exista um *plug-in* MetaJ para a referida linguagem. Assim, as operações disponíveis em SCQL possibilitam selecionar apenas trechos de código fonte de programas referentes a uma construção sintática definida pela gramática da linguagem base[2] e encapsulados pelas *referências-p*.

Um problema encontrado nesta abordagem foi a manipulação de referências cruzadas, que ocorre principalmente quando há repetição de nomes para uma mesma entidade em contextos diferentes. Por exemplo, quando dois métodos de classes diferentes possuem o mesmo nome e deseja-se verificar onde o método da primeira classe foi chamado no sistema. Utilizando a linguagem de consulta original, baseada somente em informações sintáticas, pode-se ter ocorrências de falsos positivos (i.e., seleção de chamadas de métodos ao método da segunda classe e não da primeira, como deseja-se). Alternativas para tratar este problema com as versões originais de MetaJ e SCQL resultam em soluções muito complexas.

Pensando neste proglema, foi desenvolvida uma extensão do ambiente MetaJ, ex-

[1]do inglês, *p-reference*

[2]Linguagem base é a linguagem de programação em que o programa manipulado foi escrito.

clusivamente para manipular programas escritos em Java, incluindo em seu modelo de código elementos de semântica estática dos programas, possibilitando o tratamento de referências cruzadas por meio da manipulação de escopos, declarações e usos de símbolos [Oliveira et al. 2005]. Este novo modelo representa todo o código fonte de um sistema Java por meio de um grafo, onde os nodos encapsulam escopos, declarações e usos e estão interligados com as respectivas construções sintáticas encontradas nas ASTs que encapsula cada unidade de compilação do sistema.

Este trabalho apresenta uma nova versão para a linguagem de consultas SCQL e sua aplicação na compreensão de programas, por meio da compreensão de um sistema real.

Esta versão de SCQL é definida como um super conjunto da linguagem original, cuja extensão opera sobre a representação de código fonte definida pelo ambiente estendido de MetaJ, possibilitando consultas sobre os elementos de semântica estática (escopos, declarações e usos de símbolos). Desta forma, o problema citado acima simplesmente não ocorre, devido à ligação entre usos e declarações e o tratamento de escopo que o modelo de código estendido realiza, possibilitando a declaração de consultas mais expressivas que as consultas na linguagem original.

O artigo está organizado da seguinte forma: na Seção 2 são apresentadas as definições da nova versão SCQL. Detalhes da implementação da linguagem é apresentada na Seção 3. Um estudo da aplicação de SCQL no processo de compreensão de um sistema real é apresentado na Seção 4. Trabalhos relacionados são apresentados na Seção 5. Finalmente, na Seção 6, é apresentada uma conclusão deste trabalho.

## 2. Linguagem de Consulta a Código Fonte

Uma consulta SCQL, assim como na versão original, é definida levando em consideração um *ambiente de consultas* configurado antes da consulta ser executada. *Referências-p*, que encapsulam o sistema ou partes do código fonte, são adicionadas ao ambiente de consulta vinculadas a um identificador, que poderá ser utilizado nas declarações das consultas SCQL para referenciar o conteúdo das *referências-p*.

### 2.1. Modelo de Código

O modelo básico de representação de código da versão original de SCQL é baseado nas *referências-p*, as quais encapsulam trechos de código fonte de programas e possuem tipo sintático (i.e. um tipo sintático equivale ao não terminal da gramática da linguagem objeto). No modelo estendido com semântica estática, utilizado por esta nova versão, outras entidades que compõem um sistema (i.e. declarações, usos e blocos de código), também são encapsuladas por *referências-p* especiais com tipos e operações diferentes. Na API de acesso ao modelo de código estas *referências-p* são: `PReference`, `DeclReference`, `UseReference` e `BlockReference`. Todas as *referências-p* do modelo que representam trechos de código fonte, possuem uma *referência-p* equivalente do modelo sintático. Na Tabela 1, são apresentados os significados e as principais operações das diferentes *referências-p*.

As diversas *referências-p* possuem tipos que caracterizam o trecho de programa que elas encapsulam. As *referências-p* que encapsulam construções sintáticas de programas possuem como tipo o não terminal da gramática da linguagem objeto

**Tabela 1. Operações das diferentes *referências-p* do modelo.**

| Referências-p | Significados | Operações |
|---|---|---|
| `PReference` | construção sintática | `getType, match, isComposedBy, toString` |
| `DeclReference` | declaração | `getName, getType, getUses, getUsesInThisScope, getPReference, isDeclOf, isComposedBy` |
| `UseReference` | uso de símbolo | `getName, getType, getDeclaration, getPReference, isUseOf` |
| `BlockReference` | bloco de código | `getName, getType, getDeclarations, getUses, getBlocks, getPReference, isComposedBy` |

**Tabela 2. Operadores especiais de seleção.**

| Operadores | Descrição |
|---|---|
| `USE OF id` | relação *é um uso de*. Usado na seleção de usos (`$USE`) e tipos sintáticos. |
| `DECL OF id` | relação *é uma declaração de*. Usado na seleção de declarações (`$DECL`). |
| `INSIDE id` | relação *está dentro de*. Usado na seleção de qualquer entidade localizada dentro de um determinado escopo. |
| `OUT OF id` | relação *está fora de*. Usado na seleção de qualquer entidade localizada fora de um determinado escopo. |
| `TYPE OF type` | relação *é um tipo de*. Usado na seleção de qualquer entidade cujo tipo é definido por `type`. |

(e.g. `#statement`, `#expression`, `#identifier`). As *referências-p* que encapsulam declarações de entidades no sistema possuem como tipo o nome da entidade no sistema (e.g. `@class`, `@field`, `@method`). O mesmo vale para os usos (e.g. `@method_invocation`, `@field_access`) e para os blocos de código (e.g. `@if`, `@for`, `@while`).

### 2.2. A nova versão de SCQL

SCQL (*Source Code Query Language*) é uma linguagem declarativa de consulta a código fonte inspirada em SQL. Nesta nova versão de SCQL, as consultas além de selecionar construções sintáticas, também possibilitam selecionar declarações, usos e blocos de código. O resultado das consultas são organizados de maneira tabular. Desta maneira, podem-se aplicar condições de seleção, realizar projeções e aninhar consultas, o que permite a construção de consultas complexas de maneira simples e expressiva. Para isto, SCQL oferece as operações `VIEW TREE` e `SELECT` apresentadas em sua nova versão nas seções a seguir.

#### 2.2.1. VIEW TREE

A operação `VIEW TREE` é utilizada para "visualizar" de maneira tabular as informações do modelo de semântica estática de um sistema previamente processado. Esta operação permite extrair informações relacionadas do modelo de forma declarativa. A operação `VIEW TREE` cria uma visualização tabular de um sistema por meio das seguintes informações fornecidas pelo usuário:

- o *sistema a ser visualizado*: um identificador que referencia o sistema no ambiente de consulta;
- *operadores de corte*: uma lista de tipos que especifica quais estruturas no modelo são exploradas (`FOCUSED ON`) ou evitadas (`EXCLUDING`) pelo mecanismo de busca. Estes operadores otimizam o processamento da consulta;

- *esquema da tabela*: uma lista de entidades que serão selecionadas. Informa-se o tipo de entidade a ser visualizada (tipo sintático ou `$DECL`, `$USE` e `$BLOCK`) e um identificador, usado para nomear a entidade para ser referenciada dentro da consulta. Operadores especiais podem ser declarados para tornar a consulta mais expressiva (Tabela 2). Opcionalmente, uma condição de seleção definida por uma expressão pode ser declarada (`FILTERED BY`).
- *condição de seleção de registros*: expressão booleana que, utilizando os nomes dos campos, realiza verificações sobre as entidades armazenadas por eles.

A operação `VIEW TREE` é definida como uma tupla $T = (s, c, e, w)$, onde $s$ é o sistema a ser visualizado, $c$ os operadores de corte a serem aplicados, $e$ o esquema da tabela a ser construída a partir de $s$ e $w$ a condição de seleção dos registros que serão inseridos na tabela após sua construção. O esquema da tabela $e$ é uma lista de especificações de colunas, i.e., $e = [e_1, e_2, \ldots, e_n]$. Cada especificação de coluna $e_i = [(n_i, t_i, o_i, f_i)]$ é uma tupla onde o primeiro elemento $n_i$ é o nome da coluna especificada, $t_i$ é o tipo da coluna, $o_i$ é um conjunto de operadores especiais e $f_i$ é um filtro. A seguir é apresentado os detalhes da execução de $T$:

1. aplicam-se os operadores de corte sobre o modelo do sistema $s$ a ser visualizado, resultando em $s_c$;
2. cria-se uma tabela $t$, cujo esquema é definido por $e$;
3. seja $n$ o número de colunas especificadas por $e$, para cada coluna $e_i = (c_i, t_i, o_i, f_i)$, um conjunto $v_i$ de entidades de tipo $t_i$, extraídas de $s_c$, é construído. Os operadores especiais $o_i$ e o filtro $f_i$ são aplicados sobre os valores de $v_i$ para reduzir a quantidade de valores a serem inseridos na tabela. Obtendo-se como resultado deste passo a lista de conjuntos $vs = [v_1, v_2, \ldots, v_n]$;
4. as tuplas (registros) resultantes do produto cartesiano dos conjuntos $v_i \in vs$, $1 < i \leq n$ são utilizadas para preencher a tabela $t$;
5. dentre os registros inseridos em $t$, alguns são selecionados pela expressão de seleção de registros $w$, para fazer parte do resultado da "visualização".

A sintaxe da operação `VIEW TREE` é apresentada na Figura 1, onde `MYTYPE` representa um tipo sintático e `MJXTYPE` um tipo de uma declaração, de um uso ou de um bloco de código.

Uma declaração de visualização tabular de um sistema inicia-se com a palavra reservada `VIEW` seguida de `TREE`. Em seguida deve ser declarado um identificador que referencia no ambiente de consulta o sistema a ser visualizado. Operadores de corte (`FOCUSED ON` ou `EXCLUDING`) podem ser declarados opcionalmente para definir construções do sistema a serem exploradas ou evitadas. Logo após, deve ser declarada a cláusula `AS TABLE` com uma lista de declarações dos campos da tabela, i.e. do esquema tabular da visualização. Cada campo é definido pela declaração do tipo da entidade do sistema a ser capturada (tipo sintático, `$USE`, `$DECL` ou `$BLOCK`), de um identificador, de propriedades da entidade declaradas por meio dos operadores `USE OF`, `DECL OF`, `INSIDE`, `OUT OF` e `TYPE OF`, e de um filtro, por meio da declaração `FILTERED BY` seguida de uma expressão booleana, para filtrar as entidades a serem capturadas. Finalmente, na seção `WHERE`, uma expressão indicando a condição de seleção de valores para a tabela pode ser especificada.

```
view → VIEW TREE ID focus_opt excluding_opt
       AS TABLE LPAREN typed_fields RPAREN where_opt;
focus_opt → FOCUSED ON exc_foc_list | λ;
excluding_opt → EXCLUDING exc_foc_list | λ;
exc_foc_list → exc_foc_list COMMA exc_foc_item | exc_foc_item;
exc_foc_item → MJTYPE | MJXTYPE | λ;
typed_fields → typed_fields COMMA typed_field | typed_field;
typed_field →
       MJTYPE ID decl_or_use_of_opt scope_opt type_of_opt filter_opt
   |   $DECL ID decl_of_opt scope_opt type_of_opt filter_opt
   |   $USE ID use_of_opt scope_opt type_of_opt filter_opt
   |   $BLOCK ID scope_opt type_of filter_opt;
decl_or_use_of_opt → decl_of_opt | use_of_opt;
decl_of_opt → DECL OF ID | λ;
use_of_opt → USE OF ID | λ;
decl_of_opt → DECL OF ID | λ;
scope_opt → INSIDE ID | OUT OF ID | λ;
type_of_opt → TYPE OF MJXTYPE | λ;
filter_opt → FILTERED BY expression | λ;
where_opt → WHERE expression | λ;
```

**Figura 1. Sintaxe do comando** `VIEW TREE`**.**

A seguir é apresentado um exemplo de consulta `VIEW TREE`:

```
VIEW TREE system          /* sistema a ser visualizado */
     AS TABLE ( $DECL classes TYPE OF @class )     /* esquema da tabela */
```

A consulta cria uma visualização tabular de todas as declarações de classes contidas no sistema. O sistema é referenciado pelo identificador `system` na consulta. Veja o formato da tabela construída como resultado da consulta:

| **classes:$DECL** |
|---|
| $\texttt{classe}_1$ |
| $\texttt{classe}_2$ |
| ... |
| $\texttt{classe}_n$ |

Esta tabela resultante contém uma listagem de classes, onde $\texttt{classe}_i$ ($1 \leq i \leq n$) é uma *referência-p* para a respectiva classe do modelo de código selecionada pela consulta.

Consultas mais especializadas podem ser construídas. Como exemplo, veja a seguinte consulta que visualiza todos os métodos e campos de uma determinada classe (adicionada previamente ao ambiente de consultas e vinculada ao identificador "`class`"), onde os campos são acessados dentro dos métodos.

```
VIEW TREE class EXCLUDING @anonymous_class
    AS TABLE (
      $USE fa TYPE OF @field_access,
      $DECL field TYPE OF @field,
      $DECL method TYPE OF @method
    ) WHERE fa.isUseOf(field) && method.isComposedBy(fa)
```

Esta consulta é mais complexa. Inicialmente, são criados um conjunto de usos de campos (i.e. variáveis de instância) e dois conjuntos de declarações, um de campos e outro de métodos. Por meio da declaração do operador de corte `EXCLUDING @anonymous_class`, o escopo definido pelas declarações de classe anônima são evitados durante o processo de busca. Assim, nenhuma entidade localizada dentro de uma classe anônima será selecionada. Um produto cartesiano é feito entre os três conjuntos tendo como resultado uma tabela cuja a cardinalidade é o produto das cardinalidades dos três conjuntos. Na cláusula `WHERE`, é declarada a condição de seleção dos registros, uma expressão contendo comandos que acessam as interfaces das *referências-p*. Neste exemplo, para cada registro da tabela, é verificado se o uso `fa` é um acesso ao campo `field` (i.e. `fa.isUseOf(field)`) e se está dentro do método `method` (i.e. `method.isComposedBy(fa)`). Finalmente, realizada a seleção dos registros, a tabela resultante terá somente os valores desejados. Veja abaixo uma ilustração hipotética da tabela resultante:

| **fa:$USE** | **field:$USE** | **method:$USE** |
|---|---|---|
| $\texttt{fa}_1$ | $\texttt{field}_1$ | $\texttt{method}_2$ |
| $\texttt{fa}_3$ | $\texttt{field}_1$ | $\texttt{method}_5$ |
| $\texttt{fa}_6$ | $\texttt{field}_2$ | $\texttt{method}_1$ |
| ... | ... | ... |
| $\texttt{fa}_l$ | $\texttt{field}_p$ | $\texttt{method}_q$ |

Neste exemplo, têm-se pelo menos dois acessos ao campo $\texttt{field}_1$ em dois métodos diferentes: $\texttt{method}_2$ e $\texttt{method}_5$. Um campo pode ser acessado por mais de um método, e um método, por sua vez, pode acessar mais de um campo. Assim, o número final de registros é totalmente dependente das ocorrências das entidades no código fonte do programa.

### 2.2.2. SELECT

A operação `SELECT` é utilizada para selecionar, a partir de tabelas ou combinações de tabelas, quais colunas farão parte do resultado, baseada nas informações passadas pelo usuário, que são:

- a listagem das colunas das tabelas que serão selecionados ($\pi$);
- a listagem das tabelas a serem combinadas ($ts$);
- condição de seleção de registros ($\rho$).

A semântica da operação `SELECT` de SCQL é bastante parecida com a de SQL, i.e., realiza-se um produto cartesiano das tabelas $ts$, aplica-se a condição de seleção de registros $\rho$ para filtrar os registros que estarão no resultado da consulta e, finalmente, seleciona-se os campos definidos em $\pi$. Veja a sintaxe do comando `SELECT` na Figura 2.

```
selection → SELECT fields FROM LPAREN resultset_list RPAREN where_opt
       |  SELECT * FROM LPAREN resultset_list RPAREN where_opt;
fields → fields COMMA field | field ;
field → ID AS ID | ID ;
resultset_list → resultset_list COMMA resultset | resultset ;
resultset → selection | view
```

**Figura 2. Sintaxe do comando `SELECT`.**

A operação de seleção começa pela palavra reservada `SELECT` seguida por "*" ou pela lista de colunas a serem selecionada. Então, na seção `FROM`, as tabelas a serem combinadas são listadas. Finalmente, a condição de seleção de registros é descrita na cláusula `WHERE`. A expressão definida nesta cláusula pode acessar os valores de todos os campos selecionados em $\pi$. De maneira equivalente ao comando `VIEW TREE`, estes valores podem ser acessados por meio das interfaces das *referências-p*.

Por exemplo, utilizando a consulta `VIEW TREE` da seção anterior, pode-se selecionar somente as colunas que contêm os campos e os métodos que os acessam, da seguinte forma:

```
SELECT field, method FROM (
  VIEW TREE class EXCLUDING @anonymous_class
    AS TABLE (
      $USE fa TYPE OF @field_access,
      $DECL field TYPE OF @field,
      $DECL method TYPE OF @method
    ) WHERE fa.isUseOf(field) && method.isComposedBy(fa)
 )
```

Executando esta consulta obtém-se como resultado a seguinte tabela somente com as colunas requisitadas:

| **field:$USE** | **method:$USE** |
|---|---|
| $field_1$ | $method_2$ |
| $field_1$ | $method_5$ |
| $field_2$ | $method_1$ |
| ... | ... |
| $field_p$ | $method_q$ |

Outros exemplos de consultas que utilizam o operador `SELECT` de SCQL podem ser encontrados na seção onde SCQL é aplicada no processo de compreensão de um sistema real (Seção 4).

## 3. Implementação

SCQL foi implementada como uma camada em cima do ambiente MetaJ, cujo modelo de código fonte puramente sintático foi enriquecido com informações de semântica estática. A API da linguagem disponibiliza, assim como a linguagem de consulta original , as seguintes interfaces para execução das consultas: `QueryFactory`, `Query` e `ResultSet`. Estas interfaces foram adaptadas para suportar consultas sobre o modelo

de código fonte estendido com semântica estática. A Figura 3 abaixo mostra o relacionamento entre os principais componentes de SCQL.

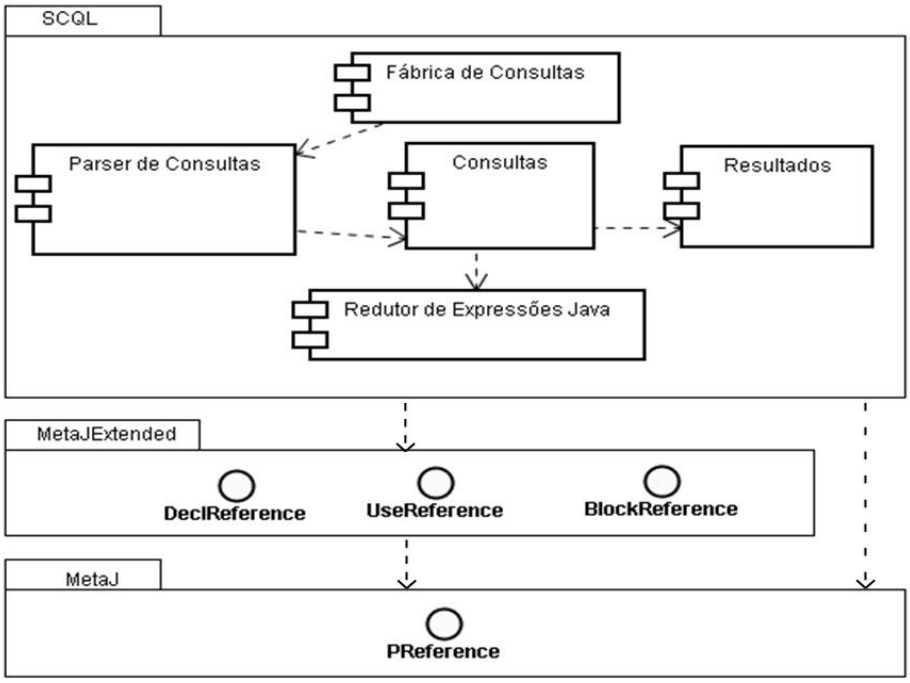


**Figura 3. Arquitetura Geral de SCQL.**

As fábricas de consultas (`QueryFactory`) encapsulam o ambiente de consulta, a qual objetos externos podem ser adicionados, vinculados a um identificador e serem referenciados dentro das declarações das consultas. As fábricas criam, utilizando-se do *parser* de consultas SCQL, consultas (`Query`) que geram como resultado objetos `ResultSets`. No processo de construção do resultado da consulta, objetos `Query` utilizam o redutor de expressões Java para avaliar as expressões especificadas nas cláusulas `WHERE` e em filtros (nas cláusulas `FILTERED BY`). Os objetos `ResultSets` possibilitam aos usuários da linguagem de consultas ter acesso aos registros contendo as *referências-p* que encapsulam as entidades do código fonte selecionadas.

## 4. Compreensão de Programas com SCQL

A tarefa de compreensão de programas torna-se um processo complicado devido ao grande volume de documentos que necessita ser analisado, que pode estar inconsistente ou com partes faltando. O código fonte freqüentemente torna-se a única fonte confiável de informações sobre o funcionamento do programa. Assim, torna-se imprescindível o desenvolvimento de ferramentas que auxiliem o desenvolvedor na tarefa de compreender um programa.

Uma das principais características de ferramentas de compreensão de programas apontadas por [Storey 2005] é a capacidade de realizar consultas avançadas nos códigos fontes dos programas. Durante o processo de compreensão, o desenvolvedor pode gerar uma série de hipóteses de fatos sobre a implementação e

usar uma ferramenta de busca para verificar se elas concretizam-se no código fonte do programa. Diferentes estratégias podem ser utilizadas durante o processo de compreensão [von Mayrhauser and Vans 1995, Brooks 1983, Soloway and Ehrlich 1989, Shneiderman and Mayer 1979, Pennington 1979]. Porém, em todas elas o suporte de consultas avançadas a código fonte mostra-se bastante útil.

Nas seções que se seguem, é apresentado um estudo de como SCQL pode auxiliar o proceso de compreensão da implementação de um sistema real, o Prevayler [3]. Uma breve descrição deste sistema é apresentada na próxima seção.

### 4.1. Um Sistema Real

O Prevayler é uma implementação do conceito de prevalência de objetos, em que todos os objetos armazenados em memória, de tempos em tempos ou no momento que a aplicação terminar, são serializados no disco (*snapshots*) e todas as manipulações de objetos são registrados num *log* de comandos, os quais são escritos imediatamente em disco.

Nos experimentos deste trabalho, foi utilizada a versão 2.3 do Prevayler escrita em Java. A implementação do sistema está localizada dentro do pacote `org.prevayler` e é composta por 8 pacotes, 28 classes, 9 interfaces, 179 métodos e 109 campos. Dentro deste pacote estão os principais objetos que funcionam como fachada de acesso: a classe `PrevaylerFactory` e as interfaces `Prevayler` e `Transaction`.

### 4.2. Compreendendo o Prevayler

Para compreender a implementação do Prevayler, um dos principais pontos é descobir como foi implementada a operação de persistência dos dados.

Inicialmente, para descobrir quais são as classes envolvidas na serialização dos objetos pelo Prevayler, uma hipótese é verificar quais classes são declaradas dentro de arquivos `.java` que importam o pacote `java.io`. A seguinte declaração expressa esta consulta:

```
SELECT classe, file FROM (
  VIEW TREE org.prevayler AS TABLE (
    $USE pack TYPE OF @package FILTERED BY pack.getName().contains("java.io"),
    $DECL file TYPE OF @file,
    $DECL classe TYPE OF @package)
   WHERE file.isComposedBy(pack) && file.isComposedBy(classe)
  )
```

Esta consulta gera três listas de entidades encontradas dentro do código do Prevayler: (i)uma lista de usos do pacote `java.io`, (ii) uma lista de arquivos Java, (iii) uma lista de declarações de classes. Um produto cartesiano é realizado com estas três listas, construindo-se uma tabela com três colunas. Por meio da cláusula `WHERE`, é declarado que somente os registros onde os usos do pacote `java.io` estão dentro do arquivo `file`, que é composto pela classe `classe` são selecionados. Finalmente, somente as colunas das classes e dos arquivos são selecionadas, obtendo como resultado uma tabela com as classes localizadas dentro de arquivos que importam o pacote `java.io`. Esta tabela contém 18 registros (Tabela 3).

[3] Disponível em http://www.prevayler.org. Acesso em 14/11/2006.

**Tabela 3. Classes e Arquivos do Prevayler que importam `java.io`.**

| classe:$DECL | file:DECL |
|---|---|
| PrevaylerFactory | org/prevayler/PrevaylerFactory.java |
| SimpleOutputStream | org/prevayler/foundation/SimpleOutputStream.java |
| DurableOutputStream | org/prevayler/foundation/DurableOutputStream.java |
| SimpleInputStream | org/prevayler/foundation/SimpleInputStream.java |
| FileManager | org/prevayler/foundation/FileMenager.java |
| TransactionTimestamp | org/prevayler/implementation/TransactionTimestamp.java |
| PrevaylerImpl | org/prevayler/implementation/PrevaylerImpl.java |
| PersistentLogger | org/prevayler/implementation/logging/PersistentLogger.java |
| CentralPublisher | org/prevayler/implementation/publishing/CentralPublisher.java |
| AbstractPublisher | org/prevayler/implementation/publishing/AbstractPublisher.java |
| StrictTransactionCensor | .../publishing/censorship/StrictTransactionCensor.java |
| ServerConnection | .../publishing/replication/ServerConnection.java |
| ServerListener | .../publishing/replication/ServerListener.java |
| ClientPublisher | .../publishing/replication/ClientPublisher.java |
| SnapshotManager | .../publishing/snapshot/SnapshotManager.java |
| NullSnapshotManager | .../publishing/snapshot/NullSnapshotManager.java |
| XStreamSnapshotManager | .../publishing/snapshot/XStreamSnapshotManager.java |
| XmlSnapshotManager | .../publishing/snapshot/XmlSnapshotManager.java |

Analisando o resultado da consulta, observa-se que, pelos nomes das classes listadas e por uma breve explorada no código fonte dos respectivos arquivos, algumas classes que não estão relacionadas com a persistência dos dados também foram selecionadas, ou seja, classes que também utilizam recursos do pacote `java.io`, mas que não estão relacionadas com a implementação a persistência. Como exemplo, podem-se citar as classes `ServerConnection`, `ServerListener` e `ClientPublisher` que estão relacionadas com a replicação de dados no Prevayler. Elas estão dentro do pacote `replication` e utilizam recursos tanto do pacote `java.io` quanto do pacote `java.net` (a classe `Socket`).

Portanto, a hipótese inicial de que as classes que implementam a persistência dos dados no Prevayler são aquelas que usam recursos do pacote `java.io` não se verifica. Assim, torna-se necessária levantar outra hipótese.

Pela documentação do Prevayler, a imagem dos dados armazenadas em disco é denominada *snapshot*. Analisando o resultado da consulta anterior (Tabela 3), observa-se a existência de uma classe gerenciadora de *snapshots*, a `SnapshotManager`. Explorando o código fonte desta classe por meio do arquivo relacionado a ela, observa-se que ela é possivelmente uma das principais classes que implementam a persistência no Prevayler. Assim, verificar quais classes possuem relacionamentos com esta classe parece ser uma boa hipótese para chegar-se às classes envolvidas com a implementação da operação de persistência.

A consulta a seguir seleciona as classes cujos métodos realizam alguma chamada de método aos métodos da classe `SnapshotManager`. São listados o método chamado da classe `SnapshotManager` , o método chamador e sua respectiva classe.

```
SELECT m_called, m_caller, class FROM (
    VIEW TREE org.prevayler AS TABLE (
        $DECL m_called TYPE OF @method,
        $DECL snapshot TYPE OF @class
                FILTERED BY snapshot.getName().equals("SnapshotManager")
     ) WHERE snapshot.isComposedBy(m_called) ,
    VIEW TREE org.prevayler AS TABLE (
        $USE call TYPE OF @method_invocation,
        $DECL m_caller TYPE OF @method,
        $DECL class TYPE OF @class
     ) WHERE m_caller.isComposedBy(call) &&
            class.isComposedBy(m_caller)
  ) WHERE call.isUseOf(m_called)
```

Esta consulta `SELECT` realiza uma combinação de mais duas consultas aninhadas para se obter o resultado final. Na primeira consulta `VIEW TREE`, são selecionados os métodos da classe `SnapshotManager`. Já na segunda, são selecionadas as chamadas de métodos, os métodos onde estas chamadas são realizadas e as respectivas classes dos métodos. Na cláusula `WHERE` mais externa, as chamadas de métodos são vinculadas aos métodos chamados. Portanto, são selecionados os métodos chamados da classe `SnapshotManager`, os métodos chamadores e suas respectivas classes. A tabela resultante desta consulta possui 11 registros apresentados na Tabela 4.

**Tabela 4. Classes relacionadas com a classe `SnapshotManager`**

| **called:$DECL** | **caller:$DECL** | **classes:DECL** |
|---|---|---|
| constructor | snapshotManager | PrevaylerFactory |
| constructor | constructor | XmlSnapshotManager |
| deepCopy | deepCopy | PrevaylerImpl |
| deepCopy | approve | StrictTransactionCensor |
| init | constructor | XStreamSnapshotManager |
| readSnapshot | produceNewFoodTaster | StrictTransactionCensor |
| recoveredPrevalentSystem | constructor | PrevaylerImpl |
| recoveredPrevalentSystem | constructor | StrictTransactionCensor |
| recoveredVersion | constructor | PrevaylerImpl |
| writeSnapshot | produceNewFoodTaster | StrictTransactionCensor |
| writeSnapshot | takeSnapshot | PrevaylerImpl |

Com esta listagem, pode-se ir direto ao código fonte dos métodos e verificar como eles operam em relação a persistência. Analisando o código fonte, verifica-se que `SnapshotManager` é a classe que realiza a persistência dos dados e as classes `XmlSnapshotManager` e `XStreamManager` são duas subclasses dela que implementam especializações de persistência de dados em arquivos XML usando respectivamente as bibliotecas *Skaringa*[4] e *XStream*[5]. A fábrica `PrevaylerFactory` instancia a classe `SnapshotManager` por padrão para ser utilizada pelo sistema prevayler. As classes `PrevaylerImpl` e `StrictTransactionCensor` utilizam a classe

[4]Disponível em http://skaringa.sourceforge.net. Acesso em 03/12/2006.

[5]Disponível em http://xstream.codehaus.org. Acesso em 03/12/2006.

`SnapshotManager` para requisitar a persistência dos dados ou recuperar dados já persistidos. Um cliente da interface `Prevayler` pode requisitar que os dados sejam persistidos usando o método `takeSnapshot`. O diagrama de seqüência para esta operação é apresentado na Figura 4. Observe que com duas consultas com SCQL, o desenvolve-

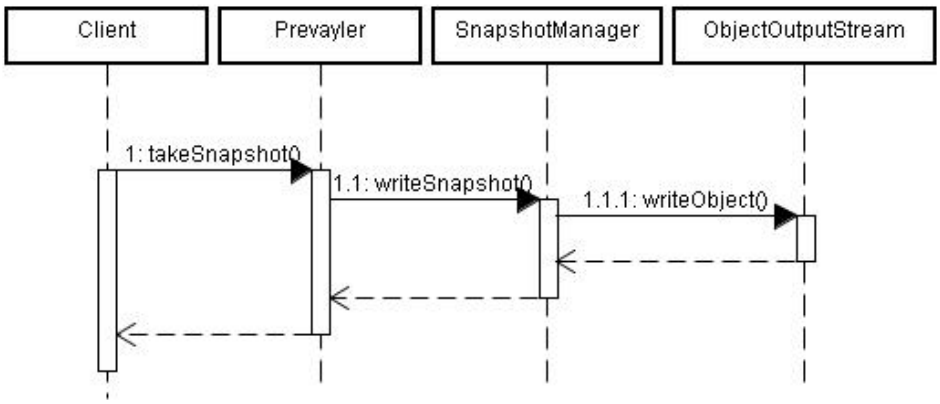


**Figura 4. Persistência requisitada por um cliente do Prevayler.**

dor evitou ter que ficar explorando exaustivamente o código fonte para chegar aos pontos onde é implementada a persistência. À medida que se foi extraindo informações da implementação do sistema com SCQL, pôde-se realizar consultas mais elaboradas que reduziram o número de classes que ter-se-ia que explorar, sem perder informações: de 28 classes para 18 na primeira consulta, e depois para 5 na segunda. Além disto, pôde-se ir direto aos métodos relacionados com a persistência, reduzindo-se ainda mais o escopo da análise do código fonte.

As informações extraídas utilizando SCQL são difíceis de serem obtidas com um ambiente de desenvolvimento comum. A maioria dos IDEs fazem tratamento de referências cruzadas, porém informações como, *quais métodos chamam um determinado método* ou *quais classes estendem uma determinada classe*, são exemplos de informações que exigem certo esforço extra do desenvolvedor, o que se transforma em tempo desperdiçado. Porém, outros IDEs, como o Eclipse [6], são mais poderosos e possibilitam obter tais informações. No entanto, estes IDEs recuperam somente informações de relações pré-definidas, enquanto SCQL possibilita o usuário definir suas próprias relações.

## 5. Trabalhos Relacionados

Ferramentas para consultas a código fonte foram amplamente estudadas na literatura e auxiliam várias tarefas da engenharia de software: entendimento de programas, engenharia reversa, análise estrutural de programas e análise de fluxo de programas [Paul and Prakash 1996]. Outro fator importante a ser destacado é que as operações de manipulação de código facilitam a reestruturação de programas e manutenção de software, o que pode ser complicado sem a ajuda destas ferramentas. A seguir são apresentadas algumas ferramentas encontradas na literatura e algumas comparações com SCQL.

[6] Disponível em http://eclipse.org. Acesso em 25/11/2006.

REFINE [Kotik and Markosian 1989] é uma linguagem construída como sistema de reescrita e não permite a utilização de padrões diretamente, assim como SCQL. Além disso, define linguagens completamente novas, o que gera uma dificuldade no aprendizado. SCQL possui a vantagem de ser parecida com SQL, uma linguagem bastante conhecida, o que facilita seu aprendizado.

ASTLOG [Crew 1997] é uma variação de PROLOG para examinar árvores de sintaxe abstrata de programas C/C++ e SCQL opera sobre programas Java. GENOA [Devanbu 1999] é uma linguagem que oferece mecanismos para consulta em código fonte, mas assim como em SCQL, não é possível fazer transformações nem geração de programas.

Em uma abordagem de analisadores estáticos de programas (SPA) encontra-se PQL [Jarzabek 1998], que é utilizada para construir SPA's flexíveis, facilitando a manutenção de programas. PQL é uma linguagem com uma notação independente de linguagem fonte para especificar consultas e visões de programas. Da mesma forma, SCQL também pode ser utilizada na construção SPAs para analisar código Java.

JavaML [Badros 2000] é uma linguagem baseada em XML utilizada para representar de maneira estruturada o código fonte de programas Java. JavaML consegue fazer ligações entre usos e declarações por meio de identificadores específicos de maneira restrita. Por exemplo, não são feitas ligações entre a declaração de um tipo (classe Java) e o uso deste tipo na declaração de uma variável. Consultas em código fonte podem ser feitas utilizando ferramentas de buscas em documentos XML, tais como XPath[Clark and DeRose 1999] e XQuery[Fernandez et al. 1999]. Por sua vez, o modelo de código estendido de SCQL trata de maneira robusta referências cruzadas, possibilitando consultas mais expressivas em SCQL.

GUPRO [Ebert et al. 2002] é um ambiente integrado para suportar compreensão de programas de sistemas de software heterogêneos num nível arbitrário de granularidade. GUPRO é fortemente baseado em grafos e os códigos fontes são armazenados em um repositório de grafos, cujas abstrações são feitas por meio de consultas a grafos e algoritmos sobre grafos. Trabalha somente com código fonte da linguagem C. As consultas a código fonte em GUPRO são realizadas por meio da linguagem GReQL que, como SCQL, organiza os resultados das consultas de forma tabular. Uma comparação interessante entre a utilização de ferramentas baseadas em grafos com ferramentas baseadas em banco de dados relacionais é encontrada em [Lange et al. 2001].

JQuery [McCormick and Volder 2004] é uma ferramenta para visualizar estruturas de programas Java e implementada como *plug-in* do Eclipse. Possui uma linguagem de consulta embutida que permite a seleção de estruturas relecionadas do programa por meio de consultas lógicas. Assim como SCQL, JQuery realiza tratamento de referências cruzadas por meio do modelo de código JDT do Eclipse e possibilita filtrar o resultado das consultas por meio de predicados. Por outro lado, JQuery é definido com um conjunto restrito de predicados, enquanto SCQL possibilita ao programador inserir no ambiente de consultas verificadores personalizados identificados, podendo ser referenciados no corpo da consulta. Estes verificadores devem ser escritos em Java e são executados durante a consulta por reflexão.

## 6. Conclusão

A manutenção de software invariavelmente requer um entendimento da implementação, realizada, em sua maioria, por meio do código fonte. Porém, entender o código de sistemas grandes é difícil e demanda muito esforço dos desenvolvedores. As técnicas normalmente utilizadas baseiam-se em navegar no código fonte ou em depurar a execução do software linha a linha. A compreensão pode ser auxiliada se apoiada por consultas especializadas no código fonte que ajudassem o desenvolvedor a localizar as informações mais rapidamente.

Este artigo propõe uma linguagem de consulta em código fonte e como esta linguagem pode ser útil na compreensao de programas, principalmente para localizar e focalizar em uma determinada parte do software que se queira analisar. A qualidade das consultas é fundamental para conseguir as informações desejadas no código fonte.

Neste trabalho mostramos ser possível focalizar determinados pontos do código fonte, por meio de consultas adequadas, para obter informação necessária para entender um determinado aspecto do software. Ainda é necessário desenhar uma metodologia mais abrangente para os desenvolvedores no projeto das consultas que eles precisam. Uma alternativa seria uma biblioteca de consultas customizáveis ou então o projeto de uma linguagem de consulta por meio de exemplos, assim como QBE para SQL, desenvolver uma QSCBE (Querying Source Code By Example) como uma camada em cima de SCQL. Operações relevantes, tais como operações entre conjuntos e de fecho, que são relativamente simples de serem implementadas, e operações de análise dinâmica (e.g. *slicing*), mais complexas, poderiam ser embutidas na linguagem para possibilitar consultas mais expressivas.

## Referências


Badros, G. J. (2000). Javaml: a markup language for java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 159–177, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554.

Clark, J. and DeRose, S. (1999). Xml path language (xpath). http://www.w3.org/TR/xpath.

Crew, R. F. (1997). Astlog: A language for examining abstract syntax trees. *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242.

Devanbu, P. T. (1999). Genoa: a customizable, front-end-retargetable source code analysis framework. *ACM Trans. Softw. Eng. Methodol.*, 8(2):177–212.

Ebert, J., Kullbach, B. V. R., and Winter, A. (2002). Gupro - generic uunderstanding of programs: an overview. *Electronic Notes in Theoretical Computer Science*, 72(2).

Fernandez, M., Siméon, J., and Wadler, P. (1999). Xml query language: Experiences and exemplars. http://www-db.research.bell-labs.com/user/simeon/xquery.html.

Jarzabek, S. (1998). Design of flexible static program analyzers with pql. *IEEE Transactions on Software Engineering*, 24(3):197–215.

Kotik, G. and Markosian, L. (1989). Automating software analysis and testing using a program transformation system. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 75–84, New York, NY, USA. ACM Press.

Lange, C., Sneed, H., and Winter, A. (2001). Comparing graph-based program comprehension tools to relational database-based tools. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, pages 209–218, Washington, DC, USA. IEEE Computer Society.

McCormick, E. and Volder, K. D. (2004). Jquery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 9–10, New York, NY, USA. ACM Press.

Oliveira, A. A. (2004). Metaj: Um ambiente para meta-porgramação em java. Dissertação de mestrado, Universidade Federal de Minas Gerais.

Oliveira, A. A., Braga, T. H., Maia, M., and da Silva Bigonha, R. (2004). Metaj: An extensible environment for metaprogramming in java. *Journal of Universal Computer Science*, 10(7):872–891.

Oliveira, A. A., Ribeiro, R. G., Pinheiro, D. P., Braga, T. H., Maia, M. A., and Bigonha, R. S. (2005). Iteradores, templates e consultas na análise e manipulação de programas. In *Anais do II Workshop de Manutenção de Software Moderna*, pages 50–65, Manaus, AM. IEEE Computer Society.

Paul, S. and Prakash, A. (1996). A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3):202–217.

Pennington, N. (1979). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3):295–341.

Shneiderman, B. and Mayer, R. (1979). Syntactic/semantic interactions of programming behaviour: A model. *International Journal of Computer and Information Sciences*, 8(3):219–238.

Soloway, E. and Ehrlich, K. (1989). *Empirical studies of programming knowledge*. ACM Press, New York, NY, USA.

Storey, M.-A. (2005). Theories, methods and tools in program comprehension: Past, present and future. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, Washington, DC, USA. IEEE Computer Society.

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55.