Towards a Repository of Common Programming Technologies Knowledge

Daniel Ratiu, Martin Feilkas, Florian Deissenböck, Jan Jürjens, Radu Marinescu

http://www4.in.tum.de/~ratiu/knowledge_repository.html

STSM
10 June 2008
Reverse engineering is a knowledge intensive activity

To analyze programs, engineers need and use of a lot of knowledge about

- Application domain
- Software technologies
- ... and how are they combined at the code level

but ...

Today's automatic tool support does not use this knowledge

because ...

There are currently no knowledge bases in a machine processable form, big enough and at the abstraction level of the source code
Ontologies are the de-facto technology for knowledge sharing

A lot of research efforts in the semantic web

However, a search for suitable ontologies for analyzing programs (e.g. Swoogle) is disappointing because:

- The existing ontologies do not cover the programming technologies domain
  - Or cover only small parts thereof
- The concepts are not at the proper abstraction level that would enable automatic code analyses

Repository of ontologies about programming technologies knowledge

http://www4.in.tum.de/~ratiu/knowledge_repository.html
The Technologies Knowledge Repository

http://www4.in.tum.de/~ratiu/knowledge_repository.html

- Contains common sense, basic knowledge that is known by every programmer, e.g.
  - Dialogs are graphical widgets,
  - ... have titles, layout information,
  - ... can contain other graphical components
  - ... can be opened, closed, moved
  - ...
Usage Scenarios

➢ Concept location
  ➢ We need semantics for the concepts
  ➢ We need a conceptual (as opposed to structural) decomposition of the program

➢ Assessing the quality of APIs
  ➢ In what measure does an API cover its domain?
  ➢ How extensible is an API with respect to its domain?
  ➢ How faithful does the API implement the domain?
  ➢ How domain appropriate is an API?
Usage Scenarios (2)

➢ **Enriching program analyses with semantic information**
  ➢ Clone detection vs. detecting logical redundancies
    ➢ Multiple implementation of concepts in the code
  ➢ Design quality assessment vs. appropriateness of the decomposition
    ➢ A program abstraction implements different and unrelated domain concepts
      ➢ e.g. logical GodClass
      ➢ e.g. the persistency layer contains GUI concepts

➢ **Indexing reusable components**
  ➢ Map the components (APIs) to the domain concepts that they implement
  ➢ Search for suitable components in an automatic manner
Usage Scenarios (3)

➢ **Teaching and technology transfer**
  ▶ Which concepts represent the core of the domain?
    ▶ e.g. what are the most important GUI concepts?
  ▶ Defining a common technological vocabulary in projects

➢ **Semantically rich IDE support**
  ▶ Smart IDEs can look over the shoulder of programmers and give hints about the logical mistakes
    ▶ Conceptual type checking
    ▶ Enable active reuse – warn if a concept is already implemented
The **possible applications** of the ontologies repository are **NOT limited** to the ones enumerated above.

Once a **critical mass of knowledge** is available in machine processable form, it **will enable the development of semantic-aware program analysis tools**.
Our approach for building the ontologies

- Existing **APIs contain** an important amount of **knowledge about the programming technologies** at the abstraction level of programs
  - Every programming language provide its own implementation for GUIs, XML processing, data bases, networking, ...

- We mine the existing APIs in order to extract domain ontologies that
  - are at the abstraction level of the code
  - represent a consensus in a community
    - the APIs represent a de-facto vocabulary among programmers
  - are already used and shared by millions of programmers
Different APIs offer different Views over the Domain.

Idea:
- different domain specific APIs
- offer different perspectives over the domain
- different implementation and design decisions
- different programming languages
... but the domain is the same
Extracting Domain Knowledge from APIs

(CSMR'08 -- Ratiu, Feilkas, Jürjens)

... by exploiting the similarities between the APIs
Example

```
package java.awt;

class Component extends Object {
  int getSize() { ... }
  int getLocation() { ... }
}

class Button extends Component { ... }

class Label extends Component {
  String getText() { ... }
}
```

```
namespace Windows.Forms;

class Control  :  ... {
  public Point Location { get; set; }
  public Size Size { get; set; }
  public string Text { get; set; }
}

class Label  :  Control  { ... }

class ButtonBase  :  Control  { ... }
```
Describing Domain Knowledge with Ontologies

- Describe the domain as light-weighted ontologies represented as graphs (set of triples: subject – verb - object):
  - Nodes – domain concepts
  - Edges – relations between the concepts
    - **isA** - between subordinate and superordinate
      - e.g. Window *isA* Component
    - **hasProperty** – between a concept and its properties
      - e.g. Component *hasProperty* Size
    - **actsOn** – between an action and the entities on which it is performed
      - e.g. Resize *actsOn* Window
    - **isDoer** – between an object and the actions to which it is subject
      - e.g. Window *isDoer* Paint
Extraction Steps

- **Step 1:** abstract the APIs in order to facilitate their comparison
- **Step 2:** inspect how are the abstract relations reflected in APIs
- **Step 3:** apply the ontology extraction algorithm
- **Step 4:** eliminate the noise
Step 1: Abstracting APIs

- We abstract APIs as graphs
  - Nodes = (lexically normalized) names of public program elements
  - Edges = program relations between the program elements
    - hasSupCls, hasAcc (Java), hasAtt, hasProp (C#), hasMeth, hasPar, hasType, hasConstr
Step 2: Reflecting abstract relations in APIs

- The taxonomic relation (isA) is typically implemented as:
  - sub-classing (hasSupCls)
  - sequences of sub-classes (<hasSupCls, hasSupCls>)
  - the type of a variable (hasType)

- **Example:**

  Dialog is a Window

  ![Diagram](Diagram.png)

  ```java
  public class Window { ... }
  public class Dialog extends Window {...}
  ```

  ```java
  public class Container extends Window {...}
  public class Dialog extends Container {...}
  ```

  ```java
  public class SomeClass {
      public Window aDialog; ...
  }
  ```
The relation "hasProperty" between a concept and its properties is typically implemented as:

- the attribute of a class (hasAtt)
- the accessor of a class (hasAcc)
- the parameter of a constructor (<hasConstr, hasPar>)

**Example:**

```java
public class Window {
    public String title; ...
}
```

```java
public class Window {
    public String getTitle() { ... }
}
```

```java
public class Window {
    public Window(String aTitle) { ... }
}
```
Step 3: Ontology extraction algorithm

- Extract the ontology as the intersection of the API graphs:
  - Matching nodes – based on the similarity of the names
  - Matching edges – according to the defined mapping strategies

At every step we discover a triple from our ontology
Example: GUI Concepts

- Accelerator
- Accessible
- Accessible Context
- Accessible Description
- Action
- Action Command
- Action Listener
- Activate
- Active
- Align
- Alignment
- Alignment X
- Alignment Y
- Button Border
- Alpha
- Angle
- Append
- Applet
- Arc
- Arc Angle
- Arc Height
- Arc Width
- Area
- Ascent
- Attribute
- Back
- Back Color
- Background
- Background Image
- Bar
- Bar Menu
- Base
- Baseline
- Bgcolor
- Block
- Blue
- Border
- Border Style
- Bottom
- Bound
- Box
- Browser
- Button
GUI Triples (1)

➢ Button | hasProperty | Active
➢ Button | hasProperty | Alignment
➢ Button | hasProperty | Background
➢ Button | hasProperty | Container
➢ Button | hasProperty | Content
➢ Button | hasProperty | Enable
➢ Button | hasProperty | Focus
➢ Button | hasProperty | Image
➢ Button | hasProperty | Label
➢ Button | hasProperty | Margin
➢ Button | hasProperty | Minimum Size
➢ Button | hasProperty | Mnemonic
➢ Button | hasProperty | Name
➢ Button | hasProperty | Parent
➢ Button | hasProperty | Style
➢ Button | hasProperty | Text
➢ Button | isA | Component
➢ Button | isA | Container
➢ Button | isA | Control
➢ Button | isA | Item
➢ Button | isA | Widget
➢ Button | isDoer | Check
➢ Button | isDoer | Click
➢ Button | isDoer | Focus
➢ Button | isDoer | Lost Focus
➢ Button | isDoer | Mouse Drag
➢ Button | isDoer | Mouse Move
➢ Button | isDoer | Notify
GUI Triples (2)

- Font | hasProperty | Bound
- Font | hasProperty | Family
- Font | hasProperty | Handle
- Font | hasProperty | Name
- Font | hasProperty | Size
- Font | hasProperty | Style
- Font | isA | Attribute
- Font | isA | Resource
- Font | isA | Text
The Knowledge Repository

➢ Contributors:
  ➢ Martin Feilkas (.Net)
  ➢ Adrian Linhard (Smalltalk)
  ➢ Petru Mihancea (C++)
  ➢ Yongming Li
  ➢ Daniel Ratiu (Java)

➢ Available ontologies
  ➢ GUI -- > 450 Concepts, > 1400 Relations
  ➢ XML -- > 150 Concepts, > 300 Relations
  ➢ Collections -- 46 Concepts, 62 Relations
  ➢ Calendar -- 46 Concepts, 46 Relations

➢ License: LGPL

http://www4.in.tum.de/~ratiu/knowledge_repository.html
Extracting (parts of) an ontology from analyzing more domain specific APIs is only the first step, in the practice there are several **pressing issues** like:

- Obtaining richer ontologies
  - More kinds of relations, more semantic (constraints)
- Validating and manually completing the ontology
  - Correctness and completeness
- Evolving the extracted ontology
  - Analyze new APIs, consistently merge new concepts
- Manipulating big ontologies
We build a community for the technologies repository

➢ Users
  ➢ use the ontologies in your research / tools

➢ Contributors
  ➢ enhance the repository

http://www4.in.tum.de/~ratiu/knowledge_repository.html