Solving the TTC Java Refactoring Case with FunnyQT

Tassilo Horn
Institute for Software Technology, University Koblenz-Landau, Germany
horn@uni-koblenz.de

This paper describes the FunnyQT solution to the TTC 2015 Java Refactoring transformation case. The solution solves all core tasks and also the extension tasks 1 and 2.

FunnyQT is a model querying and model transformation library for the functional Lisp-dialect Clojure providing a comprehensive and efficient querying and transformation API, many parts of which are provided as task-oriented embedded DSLs.

1 Introduction

This paper describes the FunnyQT[1] [2, 3] solution of the TTC 2015 Java Refactoring Case [4]. It solves all core and exception tasks with the exception of *Extension 3: Detecting Refactoring Conflicts*. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3].

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. Queries and transformations are plain Clojure programs using the features provided by the FunnyQT API.

As a Lisp, Clojure provides strong metaprogramming capabilities that are exploited by FunnyQT in order to define several *embedded domain-specific languages* (DSL, [1]) for different querying and transformation tasks.

FunnyQT is designed with extensibility in mind. By default, it supports EMF [5] models and JGraLab[5] TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into the following namespaces, each namespace providing constructs supporting concrete querying and transformation use-cases:

funnyqt.emf EMF-specific model management API
funnyqt.tg JGraLab/TGraph-specific model management API
funnyqt.generic Protocol-based, generic model management API
funnyqt.query Generic querying constructs such as quantified expressions or regular path expressions
funnyqt.polyfns Constructs for defining polymorphic functions dispatching on metamodel types
funnyqt.pmatch Pattern matching constructs
funnyqt.relational Constructs for logic-based, relational model querying inspired by Prolog
funnyqt.in-place In-place transformation rule definition constructs
funnyqt.model2model Out-place transformation definition constructs similar to ATL or QVT Operational Mappings

[1] http://funnyqt.org
[2] https://github.com/tsdh/ttc15-java-refactoring-funnyqt
[3] The SHARE image name is ArchLinux64_TTC15-FunnyQT_2
[4] http://clojure.org
[5] http://jgralab.github.io

Submitted to:
TTC 2015

funnyqt.extensional Transformation API similar to GReTL
funnyqt.bidi Constructs for defining bidirectional transformations similar to QVT Relations
funnyqt.coevo Constructs for transformations that evolve a metamodel and a conforming model simultaneously at runtime
funnyqt.visualization Model visualization
funnyqt.xmltg Constructs for querying and modifying XML files as models conforming to a DOM-like metamodel

For solving the java refactoring case, mainly the features of the namespaces *funnyqt.emf*, *funnyqt.query*, and *funnyqt.in-place* have been used.

2 Solution Description

2.1 Step 1: Java Code to Program Graph

The first step in the transformation chain is to create an instance model conforming to the program graph metamodel predefined in the case description from the Java source code that should be subject to refactoring. The FunnyQT solution does that in two substeps.

(a) Parse the Java source code into a model conforming to the EMFText JaMoPP[6] metamodel.
(b) Transform the JaMoPP model to a program graph metamodel using a FunnyQT out-place transformation.

Step (a) is implemented in the solution namespace *ttc15-java-refactoring-funnyqt.jamopp*. It simply sets up JaMoPP and defines two functions `parse-directory` and `save-java-rs`. The former parses all Java files contained in the given directory and returns a resource set representing the sources' abstract syntax graph conforming to the JaMoPP metamodel. The second function receives such a resource set and saves it back as Java code. Both just access JaMoPP built-in functionality.

Step (b) is implemented as a FunnyQT out-place transformation in the solution namespace *ttc15-java-refactoring-funnyqt.jamopp2pg*. It creates a program graph model from the JaMoPP model.

The transformation also minimizes the target program graph. The source JaMoPP model contains the complete syntax graph of the parsed Java sources including all dependencies of those. I.e., if the parsed Java program uses the java.util.List interface, then the JaMoPP model also contains this interface's ASG, i.e., all its declared methods, its super-interfaces, etc. The program graph created by the transformation only contains TClass elements for the Java classes parsed from source code and direct dependencies used as field type or method parameter or method return type. TMember elements are only created for the methods of directly parsed Java classes, and then only for those members that are not static because the case description explicitly excluded statics. As a result, the program graph contains only the information relevant to the refactorings and is reasonably small so that it can be visualized which is nice especially for debugging.

The FunnyQT out-place transformation API used for implementing this task is quite similar to ATL or QVT Operational Mappings. There are mapping rules which receive one or many JaMoPP source elements and create one or many target program graph elements.

A cutout of the transformation depicting the rules responsible for transforming fields is given in the following listing. The transformation receives one single source model `jamopp` and one single target model `pg`.

```
1 (deftransformation jamopp2pg [[jamopp] [pg]]
2   ...
```

[6] http://www.jamopp.org/index.php/JaMoPP

```
(field2tfielddef
 :from [f 'Field]
 :when (not (static? f))
 :to   [tfd 'TFieldDefinition {:signature (get-tfieldsig f)}])
(get-tfieldsig
 :from [f 'Field]
 :id   [sig (str (type-name (get-type f)) " " (j/name f))]
 :to   [tfs 'TFieldSignature {:field (get-tfield f)
                              :type  (type2tclass (get-type f))}])
(get-tfield
 :from [f 'Field]
 :id   [n (j/name f)]
 :to   [tf 'TField {:tName n}]
 (pg/->add-fields! *tg* tf))
(type2tclass
 :from [t 'Type]
 :disjuncts [class2tclass primitive2tclass])
...)
```

For each non-static field (declared by a user-defined class) in the JaMoPP model, the `field2tfielddef` rule creates a TFieldDefinition element in the program graph. The signature of this TFieldDefinition is set to the result of calling the `get-tfieldsig` rule.

This rule uses the `:id` feature to implement a n:1 semantics. Only for each unique string `sig` created by concatenating the field's type and name, a new TFieldSignature is created. If the rule is called thereafter for some other field with the same type and name, the existing field signature created at the first call is returned. The field signature's field and type references pointing to a TField and a TClass respectively are set by calling the `get-tfield` and `type2tclass` rules.

The `get-tfield` is again a n:1 rule creating a TField element for every unique field name. The `type2tclass` rule is a disjunctive rule that delegates to either `class2tclass` or `primitive2tclass` to create a (or retrieve an existing) TClass for a given JaMoPP class or primitive type.

Note that in the rules above, the name of a field is retrieved using `(j/name f)`. This is because the solution lets FunnyQT generate metamodel-specific APIs for both the JaMoPP and program graph metamodels into two namespaces which are required with the aliases `j` and `pg` respectively. These generated APIs contain attribute accessor functions (e.g. `(j/name x)` and `(j/set-name! x val)`), reference accessors (e.g., `(pg/->access tdef)`, `(pg/->set-access! tdef accs)`, `(pg/->add-access! tdef acc)`, and `(pg/->remove-access! tdef acc)`), element constructors (e.g., `(pg/create-TClass! model)`), and element sequence functions (e.g., `(pg/all-TClasses model)`). The functions of the generated APIs allow for a bit more conciseness and a better readability than the generic accessor functions. With the latter, the name of a JaMoPP field would be retrieved using `(eget f :name)`.

In total, the transformation consists of 10 rules summing up to 71 lines of code. In addition, there are five simple helper functions like `static?`, `get-type`, and `type-name` that have been used in the above rules already.

A FunnyQT transformation like the one briefly discussed above returns a map of traceability information. This map's keys are the transformation rules, and the values are maps from the respective rule's input elements to its output elements. For the third step of the overall transformation, i.e., the backpropagation of the changes performed in the program graph to the Java source code, we only need to be able to get from a program graph TClass to the corresponding JaMoPP Class, from a program graph TFieldDefinition to the corresponding JaMoPP Field, and from a program graph TMethodDefinition to the corresponding JaMoPP ClassMethod. I.e., we need an inverse lookup from target to source elements, and we are not interested in which rule created what element. Thus, the following helper function `prepare-pg2jamopp-ma` creates such an inverse lookup map from the given transformation trace.

```
(defn prepare-pg2jamopp-map [trace]
  (atom (into {} (comp (map #(% trace))
```

```
                    (map set/map-invert))
              [:class2tclass :field2tfielddef :method2tmethoddef])))
```

The function also wraps the inverse lookup map in a Clojure *atom*. All Clojure data structures such as lists, vectors, or maps are immutable. However, during (multi-step) refactoring we need to be able to update the inverse lookup map. Atoms are atomically mutable references to immutable data structures. Thus, during refactoring, the reference to the inverse lookup map can be changed atomically to a new map reflecting an updated inverse lookup map.

2.2 Step 2: Refactoring of the Program Graph

The refactorings are implemented in the solution namespace *ttc15-java-refactoring-funnyqt.refactor* using FunnyQT in-place transformation rules which combine patterns to be matched in the model with actions to be applied to the matched elements.

Before discussing the rules, the following two helper functions need to be discussed.

```
(defn find-tclass [pg qn]
  (first (filter #(= qn (pg/tName %))
                 (pg/all-TClasses pg))))

(defn find-tmethodsig [pg method-name param-qns]
  (let [pclasses (mapv (partial find-tclass pg) param-qns)]
    (first (filter #(and (-> % pg/->method pg/tName (= method-name))
                         (= pclasses (pg/->paramList %)))
                   (pg/all-TMethodSignatures pg)))))
```

The function `find-tclass` receives the program graph `pg` and a qualified name `qn` and returns the TClass with this qualified name. `find-tmethodsig` receives the program graph `pg`, a `method-name`, and a sequence of the method's parameter qualified names `param-qns`. It returns the `TMethodSignature` specified by this combination of method name and parameter types.

These two functions are called by the solution's TestInterface implementation class in order to have the actual refactoring rules parametrized with program graph elements instead of the ARTE classes `Pull_Up_Refactoring` and the like.

Pull Up Member. The case description requests *pull-up method* as first refactoring core task. However, with respect to the program graph metamodel, there is actually no difference in pulling up a method (TMethodDefinition) or a field (TFieldDefinition), i.e., it is possible to define the refactoring more general as *pull-up member* (TMember) and have it work for both fields and methods. This is what the FunnyQT solution does.

The corresponding `pull-up-member` rule is shown in the next listing. The rule is overloaded on arity. There is the version (1) of arity three which receives the program graph `pg`, the inverse lookup map atom `pg2jamopp-map-atom`, and the JaMoPP resource set `jamopp`, and there is the version (2) of arity four which receives the program graph `pg`, the inverse lookup map atom `pg2jamopp-map-atom`, a TClass `super`, and a TSignature `sig`.

```
(defrule pull-up-member
  ([pg pg2jamopp-map-atom jamopp]                                          ;; (1)
   [:extends [(pull-up-member 1)]]                                         ;; pattern
   ((do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)  ;; action
    jamopp))
  ([pg pg2jamopp-map-atom super sig]                                       ;; (2)
   [super<TClass> -<:childClasses>-> sub -<:signature>-> sig               ;; pattern
    sub -<:defines>-> member<TMember> -<:signature>-> sig
    :nested [others [super -<:childClasses>-> osub
                     :when (not= sub osub)
                     osub -<:signature>-> sig
                     osub -<:defines>-> omember<TMember> -<:signature>-> sig]]
```

```
21    :when (seq others)                                                    ;; (a)
22    super -!<:signature>-> sig                                            ;; (b)
23    :when (= (count (pg/->childClasses super)) (inc (count others)))      ;; (c)
24    :when (forall? (partial accessible-from? super)                       ;; (d)
25                   (mapcat pg/->access (conj (map :omember others) member)))]
26   (do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)))   ;; action
```

The version (2) is the one which is called by the TestInterface implementation when being called from ARTE. We'll discuss this one first.

The pattern of the version (2) matches a subclass `sup` of class `super` where `sub` defines a `member` of the given signature `sig`. A nested pattern is used to match all other subclasses of `super` which also define a member with that signature. The constraint (a) ensures that there are in fact other subclasses declaring a member with signature `sig`. Then the negative application condition (b) defines that the superclass `super` must not define a member of the given `sig` already. The constraint (c) ensures that all subclasses define a member of the given `sig`, i.e., not only a subset of all subclasses do so. Lastly, the constraint (d) makes sure that all field and method definitions accessed by the member to be pulled up is already accessible from the superclass.

The helper function `accessible-from?` is defined as follows.

```
27 (defn accessible-from? [cls m-or-f]
28   (let [defining-cls (econtainer m-or-f)]
29     (or (= defining-cls cls)                    ;; (i)
30         (superclass? defining-cls cls)          ;; (ii)
31         (not (superclass? cls defining-cls))    ;; (iii)
32         (and (pg/isa-TMethodDefinition? m-or-f) ;; (iv)
33              (member? (pg/->signature m-or-f)
34                       (pg/->signature cls))
35              (superclass? cls defining-cls)))))

36 (defn superclass? [super sub]
37   (loop [sub-super (pg/->parentClass sub)]
38     (when sub-super
39       (or (= sub-super super)
40           (recur (pg/->parentClass sub-super))))))
```

It receives a TClass `cls` and a TMember m-or-f. It returns true only if the given member is accessible from the given class. To decide that, it first computes the TClass defining the member. Then, there are four cases in which `m-or-f` is accessible from `cls`: (i) the member is defined in `cls`, (ii) `cls` inherits the member from a superclass, (iii) the defining class and `cls` are in no inheritance relationship at all, or (iv) the member is a method defined in a subclass of `cls`, and this method overrides a method defined by `cls`.

The pattern of the arity three variant (1) of the `pull-up-member` rule contains just an `:extends` clause specifying that its pattern equals the pattern defined for the arity four variant. The variant (1) is used by the extension task 2 where possible refactorings are to be proposed to the user. The difference between the overloaded versions of the `pull-up-member` rule is that version (1) matches `super` and `sig` itself whereas these two elements are parameters provided by the caller in version (2).

When a match is found, both versions of the rule call the function `do-pull-up-member!` which is defined as follows.

```
41 (defn do-pull-up-member! [pg pg2jamopp-map-atom super sub member sig others]
42   (doseq [o others]                                          ;; PG modification
43     (doseq [acc (find-accessors pg (:omember o))]
44       (pg/->remove-access! acc (:omember o))
45       (pg/->add-access! acc member))
46     (edelete! (:omember o))
47     (pg/->remove-signature! (:osub o) sig))
48   (pg/->remove-signature! sub sig)
49   (pg/->add-defines! super member)
50   (pg/->add-signature! super sig)
51   (fn [_]                                                    ;; JaMoPP modification
52     (doseq [o others]
```

```
      (edelete! (@pg2jamopp-map-atom (:omember o)))
      (swap! pg2jamopp-map-atom dissoc (:omember o)))
    (j/->add-members! (@pg2jamopp-map-atom super) (@pg2jamopp-map-atom member))))

(defn find-accessors [pg tmember]
  (filter #(member? tmember (pg/->access %))
          (pg/all-TMembers pg)))
```

It first applies the changes to the program graph by deleting all duplicate member definitions from all other subclasses of `super` and pulling up the selected member into `super`. It also updates all accessors of the old members in order to have them access the single pulled up member. Lastly, it returns a closure which performs the equivalent changes in the JaMoPP model and updates the reference to the inverse lookup map when being called.

We return a function encapsulating the changes here instead of simply applying the changes because the ARTE TestInterface defines that the back-propagation of changes happens at a different point in time than the rule application. Thus, the solution's TestInterface implementation simply collects the closures returned by appling a rule in a Java collection and invokes them in its `synchronizeChanges()` implementation.

Note that the rule's variant (1) immediately invokes the function returned by `do-pull-up-member!`. This is because this variant is not called by ARTE but is intended for extension task 2, and with that there is no need to defer back-propagation.

Create Superclass. The `create-superclass` rule uses the same mechanics as the `pull-up-member` rule. Again, it is overloaded on arity where the first version is intended for proposing refactorings to a user and the second version is for being called by ARTE.

```
(defrule create-superclass
  ([pg pg2jamopp-map-atom jamopp]
   [sig<TSignature>
    :let [classes (filter #(member? sig (pg/->signature %))
                          (remove pg/->parentClass (pg/all-TClasses pg)))
          new-superclass-qn (str (gensym "ext.NewParent"))]
    :when (> (count classes) 1)
    :extends [(create-superclass 1)]]
   ((do-create-superclass! pg pg2jamopp-map-atom classes scs new-superclass-qn)
    jamopp))
  ([pg pg2jamopp-map-atom classes new-superclass-qn]
   [:let [scs (into #{} (map pg/->parentClass) classes)]
    :when (and (= 1 (count scs))
               (not (find-tclass pg new-superclass-qn)))]
   (do-create-superclass! pg pg2jamopp-map-atom classes scs new-superclass-qn)))
```

The second version is called by ARTE with a set of `classes` for which a new superclass with qualified name `new-superclass-qn` should be created. That version's pattern first computes the set of superclasses `scs` of the given classes. The constraint then ensures that this set contains exactly one element, and that no class with qualified name `new-superclass-qn` already exists. The set of superclasses `scs` has exactly one element when either all the given `classes` have no superclass[7] or all the given `classes` have the same superclass.

The first version of the rule matches a TSignature `sig` and all classes with no superclass which define this signature. The constraint defines that there must be at least two such classes. This ensures that a create superclass refactoring is only suggested to the user if there are at least two classes with duplicate features which could probably be pulled up in further refactoring steps. Lastly, the pattern extends the pattern of the second version of `create-superclass` and thus inherits the additional `:let` binding and `:when` constraint defined in there.

The rule's action in both versions is to call `do-create-superclass!` which is defined as given below.

[7] Then the set `scs` contains just `nil`.

```
(defn do-create-superclass! [pg pg2jamopp-map-atom classes scs new-superclass-qn]
  (let [new-tclass (pg/create-TClass! pg {:tName new-superclass-qn
                                          :childClasses classes
                                          :parentClass (first scs)})]
    (fn [^ResourceSet rs]
      (let [[pkgs class-name] (let [parts (str/split new-superclass-qn #"\.")]
                                [(butlast parts) (last parts)])
            ^Resource other-r (.get (.getResources rs) 0)
            r (new-resource rs (str (->> other-r .getURI .toFileString
                                         (re-matches #"(.*[/-]src/).*")
                                         second)
                                    (str/join "/" pkgs) "/" class-name ".java"))
            nc (j/create-Class! nil {:name class-name
                                     :annotationsAndModifiers [(j/create-Public! nil)]})
            cu (j/create-CompilationUnit! r {:name (str new-superclass-qn ".java")
                                             :namespaces  pkgs
                                             :classifiers [nc]})]
        (doseq [c classes]
          (j/->set-extends! (@pg2jamopp-map-atom c) (make-type-reference nc)))
        (when-let [parent (first scs)]
          (j/->set-extends! nc (make-type-reference (@pg2jamopp-map-atom parent))))
        (swap! pg2jamopp-map-atom assoc new-tclass nc)))))

(defn make-type-reference [target-class]
  (j/create-NamespaceClassifierReference!
   nil {:namespaces (j/namespaces (econtainer target-class))
        :classifierReferences [(j/create-ClassifierReference!
                                nil {:target target-class})]}))
```

It creates a new TClass of the given `new-superclass-qn` in the program graph, and makes all `classes` a subclass of it. If those had a common superclass before the refactoring, then this common superclass becomes the superclass of the newly created class.

The function again returns a closure which performs the same change to the JaMoPP model when being invoked to the JaMoPP resource set. Here, the changes are a bit longish because actually a new resource containing a compilation unit defining the new class need to be created, and a bit string-matching needs to be performed to figure out with which file the newly created class has to be associated in order to make JaMaPP's serialization back to Java work.

Extract Superclass (Extension 1). With the `pull-up-member` and `create-superclass` rules in place, defining a `extract-superclass` rule is simply a matter of composing the former two as given below.

```
(defrule extract-superclass [pg pg2jamopp-map-atom jamopp]
  [:extends [(create-superclass 0)]]
  ((create-superclass pg pg2jamopp-map-atom classes new-superclass-qn) jamopp)
  (let [super (find-tclass pg new-superclass-qn)]
    (doseq [sig (filter (fn [sig]
                          (forall? #(member? sig (pg/->signature %)) classes))
                        (pg/all-TSignatures pg))]
      ((pull-up-member pg pg2jamopp-map-atom super sig) jamopp))))
```

The `extract-superclass` rule is not overloaded on arity because it is never called by ARTE. Instead, it is only called by extension task 2 in order to propose a refactoring.

Its pattern extends the pattern of the first variant of `create-superclass`. Remember that this pattern has been defined in such a way that it matches only a set of at least two classes which have common features, i.e., classes for which `pull-up-member` will probably be applicable after creating a superclass. Note that this is slightly incorrect: `extract-superclass` matches a set of classes with common features where none of them turns out as actually pullable, too. However, since the `accessible-from?` predicate requires an existing TClass element in its current form, the specification of the constraint *there are common features and at least one of them would be pullable into a new superclass if that were created* would require more code. Thus we have sacrificed a bit correctness in some corner-cases for conciseness of the solution.

The rule's actions then simply call `create-superclass`, and then call `pull-up-member` for each member whose signature is defined by all subclasses of the newly created superclass.

Propose Refactoring (Extension 2). The rules `pull-up-member`, `create-superclass`, and `extract-superclass` discussed above already have overloaded versions (or just a single version) which match all elements relevant for the corresponding refactoring themselves.

FunnyQT provides a rule combinator `interactive-rule` which receives one or many rules and returns a new rule which shows to the user all applicable rules and their matches. The user can then select the rule and the match to be applied interactively.

The definition for interactive refactoring is given below.

```
109 (defn refactor-interactively [pg pg2jamopp-map-atom jamopp]
110   ((interactive-rule create-superclass pull-up-member extract-superclass)
111    pg pg2jamopp-map-atom jamopp))
```

An interactive rule for `create-superclass`, `pull-up-member`, and `extract-superclass` is created and immediately applied to the arguments which the three rules have in common.

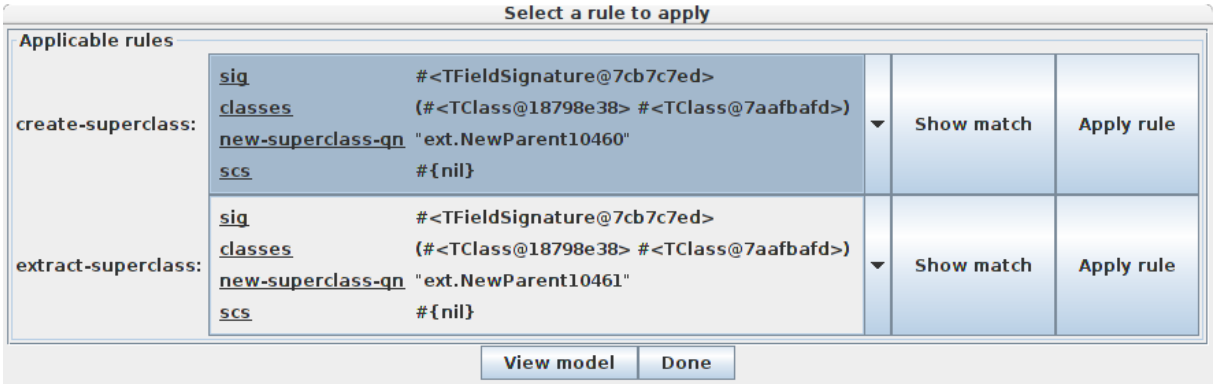

Figure 1: Interactive rule application

Figure 1 shows a screenshot of the interactive rule application GUI. All applicable rules are listed, and the match to which a rule should be applied can be selected from comboboxes. Using the *View model* and *Show match* buttons, visualizations of the complete model or only the parts around the currently selected match can be shown.

This interactive rule application is automatically started when running `lein test` in the solution project. The sources subject to the refactoring are given in the following listing.

```
// file foo/C1.java
package foo;
class C1 {
    String f1;
    String method1() {return f1;}
    String method2() {}
}
// file foo/C2.java
package foo;
class C2 {
    String f1;
    String method1() {return f1;}
    String method2() {}
}
```

This will bring up the exact rule selection dialog shown in figure 1 on the preceding page. If one chooses the `extract-superclass` rule, it is applied once and then no further refactorings are possible. If one chooses the `create-superclass` rule instead, afterwards the `pull-up-member` rule can be applied three times before no refactorings are applicable anymore.

In any case, the final refactored Java program equals the one in the following listing.

```
// file foo/C1.java
package foo;
class C1 extends ext.NewParent10460 {}
// file foo/C2.java
package foo;
class C2 extends ext.NewParent10460 {}
// file ext/NewParent10460.java
package ext;
public class NewParent10460 {
    String f1;
    String method1() {return f1;}
    String method2() {}
}
```

A new parent class NewParent10460[8] has been created in the package ext and set as superclass of C1 and C2. All members that have been duplicated in C1 and C2 previously are now defined in NewParent10460.

Detecting Refactoring Conflicts (Extension 3). With FunnyQT, the actions of an in-place transformation rule are defined using plain FunnyQT/Clojure code. Therefore, it is not possible to detect critical pairs using a static analysis in the general case.

Instead, FunnyQT provides facilities for state space generation and exploration. The state space graph with respect to a given set of refactoring rules could be computed and analyzed to determine which rule or sequence of rules makes some other rule inapplicable.

However, the state space approach doesn't work in this concrete case, too. The reason is that the state space generation internally copies the program graph model for each rule execution and then the inverse lookup map from program graph TClass and TMember elements to their JaMoPP counterparts is invalidated, i.e., it contains mappings only for the original elements but not for their copies.

2.3 Step 3: Program Graph to Java Code

The core `pull-up-member` and `create-superclass` rules both return closures which perform the refactoring's actions in the JaMoPP model when ARTE calls the TestInterface's `synchronizeChanges()` method. Thereafter, the JaMoPP model needs to be saved to reflect those changes also in the Java source code files. This is what `synchronizeChanges()` method of the solution's TestInterface implementation class does.

```
public boolean synchronizeChanges() {
    try {
        for (IFn synchronizer : synchronizeFns) {
            synchronizer.invoke(jamoppRS);
        }
        SAVE_JAVA_RESOURCE_SET.invoke(jamoppRS);
        return true;
    } catch (Exception e) {
        return false;
    } finally {
        synchronizeFns.clear();
    }
}
```

[8] The number varies.

In there, `synchronizedFns` is the list of functions returned by the rules. This list is filled by the two `apply*` methods which apply either the pull-up method refactoring or the create superclass refactoring.

After the synchronizing functions have been invoked to the JaMoPP resource set, the resource set is saved. `SAVE_JAVA_RESOURCE_SET` is a reference to the `save-java-rs` FunnyQT function discussed in subsection 2.1 on page 2.

3 Evaluation

In this section, the FunnyQT solution is evaluated according to the criteria suggested in the case description.

Correctness and completeness (max. 60 points). The FunnyQT solution passes all test cases provided by the ARTE testing framework thus it seems to be correct and complete with respect to the restricted subset of Java detailed in the case description. Thus, there is no obvious reason why it shouldn't earn the maximum of 60 points here.

Performance (max. 10 points). According to ARTE, the FunnyQT solution runs in less than a tenth of a second for all test cases on an off-the-shelf laptop except for *pub_pum3_1* where the execution takes 0.67 seconds. However, when running only that test (`execute --test pub_pum3_1`) its execution time is measured with 0.02 seconds which matches the execution time of the other cases. So this single outlier with `execute --all` seems to be a GC hiccup or something alike. However, the execution times for the toy examples tested by ARTE are not very significant anyhow. It would be very interesting to have tests covering larger code bases on which multiple refactorings depending on each other are performed.

In any case, the execution time of the actual refactorings on the program graph and the back-propagation into the JaMoPP model are completely negligible when being compared to the time JaMoPP needs to parse the Java sources, resolve references in the created model, and serialize the model back to Java again. As a reference, on a medium-sized project with 1358 files amounting to 257267 LOC, JaMoPP takes about two minutes for parsing and reference resolution.

Since the performance score will be assessed in comparison to other solutions, no value can be suggested here.

Reviewer opinion (max. 2 x 15 points). The strongest point of the solution is its completeness in that it solves all core tasks and two out of three extension tasks. FunnyQT's rule overloading and pattern inheritance features helped here a lot in order to avoid duplication of large parts of patterns. However, pattern inheritance trades comprehensibility for conciseness. The extended pattern is not visible in the extending pattern, thus the latter isn't understandable without the former. This is actually the same in OOP where the members inherited from superclasses aren't obviously visible in the subclasses.

A strong point of the solution is its conciseness. It weights only 271 NCLOC of FunnyQT/Clojure code for all core and extension tasks and 145 NCLOC of Java code for the `TestInterface` implementation class required by ARTE.

The performance is also good for the provided test cases although the EMFText JaMoPP which is used by the solution might the bottleneck when applying the refactorings to larger code bases.

With respect to debuggability, debugging FunnyQT rules quite doable. The interactive rule application combinator `interactive-rule` used for solving extension 2 is actually a high-level debugging

tool which lets users steer rule application manually, inspect matches, and visualize (parts of) the model under transformation.

A weak point of the solution and FunnyQT (Clojure) in general can be seen in that it is dynamically typed, and thus type errors are runtime errors. Of course, the type world implied by a metamodel is different than the type world of Java (at least unless classes are generated from the metamodel). But for example, Henshin requires the metamodel to be known when specifying patterns and rules, and then the visual Henshin editor makes it impossible to define patterns which use types, references, or attributes which aren't defined by the metamodel.

Extensions (max. 15 points). The FunnyQT solution provides runnable implementations for the extensions 1 (*extract superclass*) and 2 (*propose refactoring*). Extension 3 (*detect refactoring conflicts*) hasn't been solved practically but an idea for its solution has been sketched. However, that requires some further additions to FunnyQT's state space generation facility. So the FunnyQT solution should score at least 10 out of 15 points for the extension task score.

4 Conclusion

This paper discussed the FunnyQT solution to the TTC 2015 Java Refactoring case. The solution solves all core tasks and also two out of three extension tasks, namely extensions 1 (*extract superclass*) and 2 (*propose refactoring*).

The solution is correct and complete. All tests performed by the ARTE testing framework pass.

The solution is also quite concise summing up to 271 lines of FunnyQT/Clojure code for the actual realization and 145 lines of Java code for the `TestInterface` implementation required by ARTE.

The performance of the solution is also good. All test cases are executed in small fractions of a second on an off-the-shelf laptop. However, all tests performed by ARTE are executed on very small toy programs so the significance of the measured execution times with respect to real-world code bases is questionable.

References

[1] Martin Fowler (2010): *Domain-Specific Languages*. Addison-Wesley Professional.

[2] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT*, *Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[3] Tassilo Horn (2015): *Graph Pattern Matching as an Embedded Clojure DSL*. In: *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015*. To appear.

[4] Géza Kulcsár, Sven Peldszus & Malte Lochau (2015): *Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation*. In: *Transformation Tool Contest 2015*.

[5] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2008): *EMF: Eclipse Modeling Framework*, 2 edition. Addison-Wesley Professional.