Experiments and Results

Mehmet Kaya
Department of computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
mkaya@syr.edu

James W. Fawcett
Department of computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

This document demonstrates three of the experiments that we performed for our paper entitled "Identifying Extract Method Opportunities Based on Variable References". Results of these experiments are supported with a source code analysis tool we developed. Here we demonstrate, by giving the original and modified code for the methods that the proposed technique in our paper works effectively with the analysis and visualization tools we developed.

The tools that we developed for our experiments use a rule based ad-hoc parser. This parser analyzes source code extracting only the information we need using rules that work on specific collections of tokens we call "SemiExpressions". We chose this approach over using a traditional parser generator because the results we seek depend on only a small part of the (C++) language grammar.

Experiment 1

First method used for our experiments is shown in Figure 2. Figure 2 shows the placement tree for the method given in Figure 2.

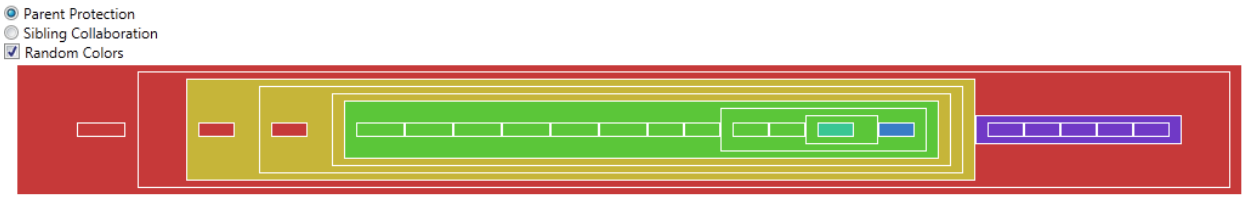


Figure 1 Placement Tree of Original Method

```
void doAction(ITokCollection *& pTc){
    ITokCollection & tc = * pTc ;
    int _for = tc.find("for");
    int _if = tc.find("if");
    int _while = tc.find("while");
    if(_for < tc.length()|| _if < tc.length()|| _while < tc.length()){
        DataAnalWithoutEq(pTc);
        return ;
    }
    int eq = tc.find("=");
    if(eq < tc.length()&& tc[tc.length()-1]!= "{"){
        int p,sp,tp ;
        p = sp = tp = 0 ;
        bool varFound = false ;
        if(eq == 1){
            if(! isSpecialChar(tc[0])){
                AddAReference(tc[0]);
            }
        }
        else {
            if(eq == 2){
                if(! isSpecialChar(tc[1])){
                    AddAReference(tc[1]);
                }
            }
            else {
                if(eq > 2){
                    for(int i = eq-1 ; i >= 0 ; i--){
                        if(tc[i]=="]"){
                            sp++;
                            continue ;
                        }
                        if(tc[i]==")"){
                            p++;
                            continue ;
                        }
                        if(tc[i]== ">"){
                            tp++;
                            continue ;
                        }
                        if(tc[i]=="["){
                            sp--;
                            continue ;
                        }
                        if(tc[i]=="("){
                            p--;
                            continue ;
                        }
                        if(tc[i]== "<"){
                            tp--;
                            continue ;
                        }
                        if(i >= 1 && tc[i-1]=="->"){
                            continue ;
                        }
                        if(i >= 1 && tc[i-1]=="."){
                            continue ;
                        }
                        if(i < tc.length()-1 && tc[i+1]!="("){
                            if(i > 0 && tc[i-1]=="::"){
                                continue ;
                            }
                            if(i < tc.length()-1 && tc[i+1]=="::"){
                                continue ;
                            }
                            if(! isSpecialChar(tc[i])){
                                if(p == 0 && sp == 0 && tp == 0){
                                    varFound = true ;
                                }
                                AddAReference(tc[i]);
                            }
                            if(varFound){
                                break ;
                            }
                        }
                    }
                }
            }
        }
        for(int i = eq ; i < tc.length(); i++){
            if(i >= 1 && tc[i-1]=="->"){
                continue ;
            }
            if(i >= 1 && tc[i-1]=="."){
                continue ;
            }
            if(i < tc.length()-1 && tc[i+1]=="("){
                continue ;
            }
            if(i < tc.length()-1 && tc[i+1]=="::"){
                continue ;
            }
            if(! isSpecialChar(tc[i])){
                AddAReference(tc[i]);
            }
        }
    }
    else {
        DataAnalWithoutEq(pTc);
    }
}
```

Figure 2 Original Method

Restructured version of the method given in Figure 2 is shown in Figure 3.

```
void BeforeEqualSignProcessing(int eq, ITokCollection& tc, int sp,
                               int p, int tp, bool varFound)
  {
    for(int i = eq-1 ; i >= 0 ; i--){
                        if(tc[i]=="]"){
                            sp++;
                            continue ;
                        }
                        if(tc[i]==")"){
                            p++;
                            continue ;
                        }
                        if(tc[i]== ">"){
                            tp++;
                            continue ;
                        }
                        if(tc[i]=="["){
                            sp--;
                            continue ;
                        }
                        if(tc[i]=="("){
                            p--;
                            continue ;
                        }
                        if(tc[i]== "<"){
                            tp--;
                            continue ;
                        }
                        if(i >= 1 && tc[i-1]=="->"){
                            continue ;
                        }
                        if(i >= 1 && tc[i-1]="."){
                            continue ;
                        }
                        if(i < tc.length()-1 && tc[i+1]!="("){
                            if(i > 0 && tc[i-1]=="::"){
                                continue ;
                            }
                            if(i < tc.length()-1 && tc[i+1]=="::"){
                                continue ;
                            }
                            if(! isSpecialChar(tc[i])){
                                if(p == 0 && sp == 0 && tp == 0){
                                    varFound = true ;
                                }
                                AddAReference(tc[i]);
                            }
                            if(varFound){
                                break ;
                            }
                        }
                    }
}
void  AfterEqualSignProcessing(int eq, ITokCollection& tc)
{
    for(int i = eq ; i < tc.length(); i++){
            if(i >= 1 && tc[i-1]=="->"){
                continue ;
            }
            if(i >= 1 && tc[i-1]="."){
                continue ;
            }
            if(i < tc.length()-1 && tc[i+1]=="("){
                continue ;
            }
            if(i < tc.length()-1 && tc[i+1]=="::"){
                continue ; |
            }
            if(! isSpecialChar(tc[i])){
                AddAReference(tc[i]);
            }
        }
}
void EqualSignProcesing(int eq, ITokCollection& tc, int sp,
                        int p, int tp, bool varFound)
{
        if(eq == 1){
            if(! isSpecialChar(tc[0])){
                AddAReference(tc[0]);
            }
        }
        else {
            if(eq == 2){
                if(! isSpecialChar(tc[1])){
                    AddAReference(tc[1]);
                }
            }
            else {
                if(eq > 2){
                    BeforeEqualSignProcessing(eq, tc, sp, p, tp, varFound);
                }
            }
        }
}
void doAction(ITokCollection *& pTc){
    ITokCollection & tc = * pTc ;
    int _for = tc.find("for");
    int _if = tc.find("if");
    int _while = tc.find("while");
    if(_for < tc.length()|| _if < tc.length()|| _while < tc.length()){
        DataAnalWithoutEq(pTc);
        return ;
    }
    int eq = tc.find("=");
    if(eq < tc.length()&& tc[tc.length()-1]!= "{"){
        int p,sp,tp ;
        p = sp = tp = 0 ;
        bool varFound = false ;
        EqualSignProcesing(eq, tc, sp, p, tp, varFound);
        AfterEqualSignProcessing(eq, tc);
    }
    else {
        DataAnalWithoutEq(pTc);
    }
}
```

Figure 1 Restructured Version For Experiment 1

After restructuring we run our tool and get the following placement tree for the new version of the code.

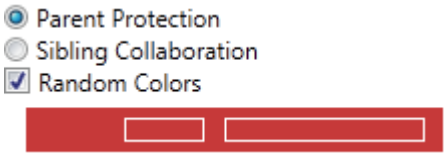
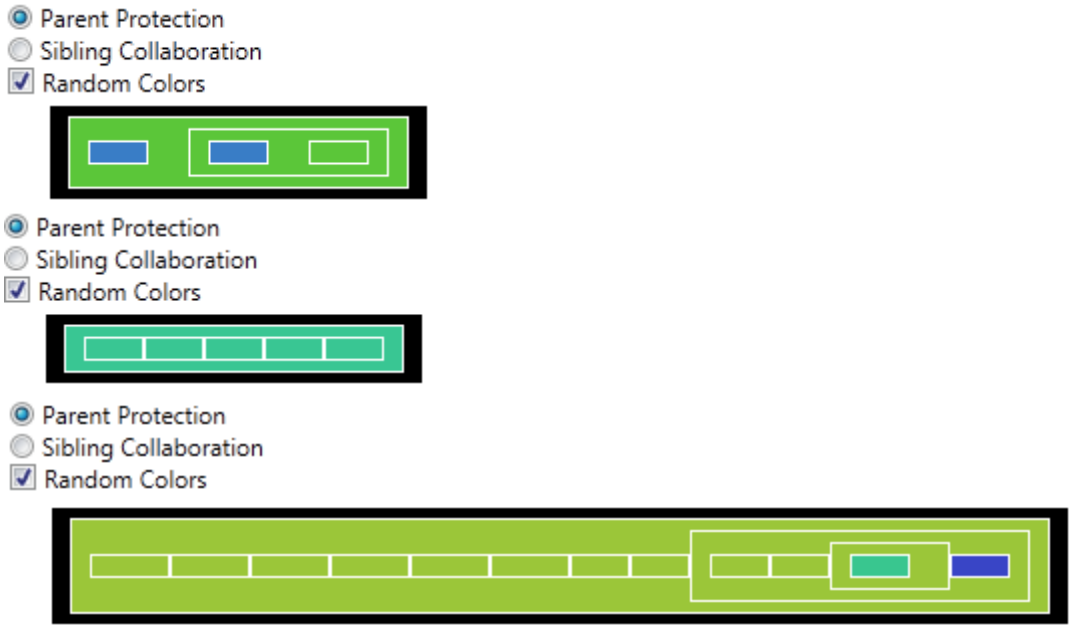


Figure 2 Original Method After Refactoring

Following figures show the placement trees for the new extracted methods.



Experiment 2

Second method used for our experiments is shown in Figure 5.

```
void doAction(ITokCollection *& pTc){
    if(hasOnlyOneWord(pTc)){
        return ;
    }
    ITokCollection & tc = * pTc ;
    if(tc.length()> 0 && tc[0]== "#"){
        return ;
    }
    int eq = tc.find("=");
    if(eq > tc.length()-1 && tc.length()>= 2 && tc[tc.length()-1]== ";" && tc[tc.length()-2]==")"){
        return ;
    }
    if(tc.find("for")< tc.length()){
        int deletel = 0 ;
        int _for = tc.find("for");
        int _eq = tc.find("=");
        int _sc = tc.find(";");
        int StartIndex ;
        if(_eq < _sc && _eq < tc.length()){
            StartIndex = _eq ;
        }
        else {
            if(_sc < _eq && _sc < tc.length()){
                StartIndex = _sc ;
            }
            else {
                return ;
            }
        }
        if(StartIndex == _for+3){
            return ;
        }
        for(int i = StartIndex-1 ; i >= 0 ; i--){
            if(! isSpecialChar(tc[i])){
                std::string type ;
                for(int j = _for+2 ; j < i ; j++){
                    type.append(tc[j]);
                    type.append(" ");
                }
                FoundLocal(type,tc[i]);
                return ;
            }
        }
    }
    if((tc[tc.length()-1]== ";" && eq < tc.length())||(tc[tc.length()-1]== "{" && eq == tc.length()-2)){
        int p = 0 ;
        int s = 0 ;
        for(int i = eq ; i >= 0 ; i--){
            if(tc[i]==")"){
                p--;
            }
            else {
                if(tc[i]=="("){
                    p++;
                }
                else {
                    if(tc[i]=="["){
                        s++;
                    }
                    else {
                        if(tc[i]=="]"){
                            s--;
                        }
                    }
                }
            }
            if(p == 0 && s == 0){
                if(i == 1){
                    return ;
                }
                if(i == 2 && ! isSpecialChar(tc[i-1])&&(tc[i-2]== "*" || tc[i-2]== ">" || tc[i-2]== "&")){
                    return ;
                }
                if(i > 1 && ! isSpecialChar(tc[i-1])&&(tc[i-2]== "*" || tc[i-2]== ">" || tc[i-2]== "&")){
                    int t = tc.find("<");
                    if(t < tc.length()&& t > 0){
                        FoundLocal(GetFullType(pTc,t-1),tc[i-1]);
                        break ;
                    }
                    else {
                        int star = tc.find("*");
                        int amp = tc.find("&");
                        int typeIndex ;
                        if(star < amp){
                            typeIndex = star ;
                        }
                        else {
                            typeIndex = amp ;
                        }
                        if(typeIndex < tc.length()&& ! isKey(tc[typeIndex-1])){
                            FoundLocal(GetFullType(pTc,typeIndex-1),tc[i-1]);
                            break ;
                        }
                    }
                }
                if(i > 1 && ! isSpecialChar(tc[i-1])&& ! isKey(tc[i-2])){
                    FoundLocal(GetFullType(pTc,i-2),tc[i-1]);
                    break ;
                }
            }
        }
    }
    else {
        if(eq > tc.length()-1 && tc[tc.length()-1]== ";"){
            std::vector < std::string > temp ;
            std::string type = "unkown" ;
            for(int i = tc.length()-1 ; i >= 0 ; i--){
                if(! isSpecialChar(tc[i])){
                    temp.push_back(tc[i]);
                }
                if(tc[i]== "*" || tc[i]== "&" || tc[i]== ">"){
                    int star = tc.find("*");
                    int amp = tc.find("&");
                    int typeIndex ;
                    if(star < amp){
                        typeIndex = star ;
                    }
                    else {
                        typeIndex = amp ;
                    }
                    if(typeIndex < tc.length()&& ! isKey(tc[typeIndex-1])){
                        type = GetFullType(pTc,typeIndex-1);
                    }
                    break ;
                }
                if(i == 1){
                    if(! isKey(tc[0])){
                        type = GetFullType(pTc,0);
                    }
                    break ;
                }
            }
            for(int i = 0 ; i < temp.size(); i++){
                FoundLocal(type,temp[i]);
            }
        }
    }
}
```

Figure 3 Original Method

Figure 6 shows the placement tree for the method given in Figure 5.

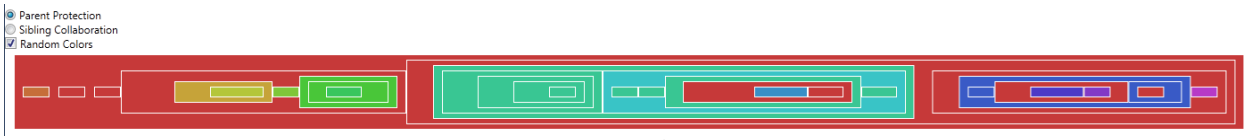


Figure 4 Placement Tree for Experiment 2

Restructured version of the method given in Figure 5 is shown in Figure 7.

```
bool FindStartIndex(int &StartIndex, int& _eq,
                    int& _sc, ITokCollection & tc)
{
  if(_eq < _sc && _eq < tc.length()){
        StartIndex = _eq ;
      }
      else {
          if(_sc < _eq && _sc < tc.length()){
              StartIndex = _sc ;
          }
          else {
              return true;
          }
      }
  return false;
}
bool FindDataBeforeStartIndex(int &StartIndex,
                              int& _for, ITokCollection & tc)
{
  for(int i = StartIndex-1 ; i >= 0 ; i--){
          if(! isSpecialChar(tc[i])){
              std::string type ;
              for(int j = _for+2 ; j < i ; j++){
                  type.append(tc[j]);
                  type.append(" ");
              }
              FoundLocal(type,tc[i]);
              return true;
          }
      }
  return false;
}
bool ProcessBeforeEqual(int& eq,ITokCollection & tc, int& p,
                        int& s, ITokCollection *& pTc)
{
  for(int i = eq ; i >= 0 ; i--){
          if(tc[i]==")"){
              p--;
          }
          else {
              if(tc[i]=="("){
                  p++;
              }
              else {
                  if(tc[i]=="["){
                      s++;
                  }
                  else {
                      if(tc[i]=="]"){
                          s--;
                      }
                  }
              }
          }
          if(p == 0 && s == 0){
              if(i == 1){
                  return true;
              }
              if(i == 2 && ! isSpecialChar(tc[i-1])&&(tc[i-2]== "*" || tc[i-2]== ">" || tc[i-2]== "&")){
                  return true;
              }
              if(i > 1 && ! isSpecialChar(tc[i-1])&&(tc[i-2]== "*" || tc[i-2]== ">" || tc[i-2]== "&")){
                  int t = tc.find("<");
                  if(t < tc.length()&& t > 0){
                      FoundLocal(GetFullType(pTc,t-1),tc[i-1]);
                      break ;
                  }
                  else {
                      int star = tc.find("*");
                      int amp = tc.find("&");
                      int typeIndex ;
                      if(star < amp){
                          typeIndex = star ;
                      }
                      else {
                          typeIndex = amp ;
                      }
                      if(typeIndex < tc.length()&& ! isKey(tc[typeIndex-1])){
                          FoundLocal(GetFullType(pTc,typeIndex-1),tc[i-1]);
                          break ;
                      }
                  }
              }
              if(i > 1 && ! isSpecialChar(tc[i-1])&& ! isKey(tc[i-2])){
                  FoundLocal(GetFullType(pTc,i-2),tc[i-1]);
                  break ;
              }
          }
      }
      return false;
}
```

```
void ProcessAStatement(ITokCollection & tc,
          std::vector < std::string >& temp,std::string& type,ITokCollection *& pTc)
{
  for(int i = tc.length()-1 ; i >= 0 ; i--){
              if(! isSpecialChar(tc[i])){
                  temp.push_back(tc[i]);
              }
              if(tc[i]== "*" || tc[i]== "&" || tc[i]== ">"){
                  int star = tc.find("*");
                  int amp = tc.find("&");
                  int typeIndex ;
                  if(star < amp){
                      typeIndex = star ;
                  }
                  else {
                      typeIndex = amp ;
                  }
                  if(typeIndex < tc.length()&& ! isKey(tc[typeIndex-1])){
                      type = GetFullType(pTc,typeIndex-1);
                  }
                  break ;
              }
              if(i == 1){
                  if(! isKey(tc[0])){
                      type = GetFullType(pTc,0);
                  }
                  break ;
              }
          }
}
void doAction(ITokCollection *& pTc){
  if(hasOnlyOneWord(pTc)){
      return ;
  }
  ITokCollection & tc = * pTc ;
  if(tc.length()> 0 && tc[0]== "#"){
      return ;
  }
  int eq = tc.find("=");
  if(eq > tc.length()-1 && tc.length()>= 2 && tc[tc.length()-1]== ";" && tc[tc.length()-2]==")"){
      return ;
  }
  if(tc.find("for")< tc.length()){
      int deletel = 0 ;
      int _for = tc.find("for");
      int _eq = tc.find("=");
      int _sc = tc.find(";");
      int StartIndex ;
      if(FindStartIndex(StartIndex,_eq,_sc,tc)){
          return;
      }
      if(StartIndex == _for+3){
          return ;
      }
      if(FindDataBeforeStartIndex(StartIndex,_for,tc)){
          return;
      }
  }
  if((tc[tc.length()-1]== ";" && eq < tc.length())||(tc[tc.length()-1]== "{" && eq == tc.length()-2)){
      int p = 0 ;
      int s = 0 ;
      if(ProcessBeforeEqual(eq,tc,p,s,pTc)){
              return;
      }
  }
  else {
      if(eq > tc.length()-1 && tc[tc.length()-1]== ";"){
          std::vector < std::string > temp ;
          std::string type = "unkown" ;
          ProcessAStatement(tc,temp,type,pTc);
          for(int i = 0 ; i < temp.size(); i++){
              FoundLocal(type,temp[i]);
          }
      }
  }
}
```

Figure 5 Restructured Version for Experiment 2

After restructuring we run our tool and get the following placement tree for the new version of the code.

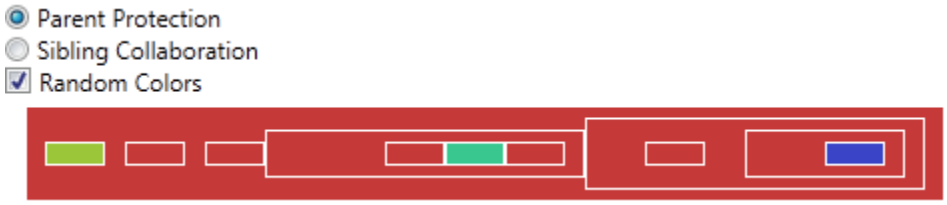
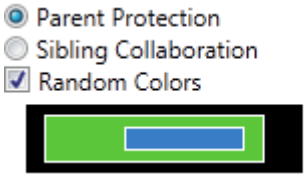
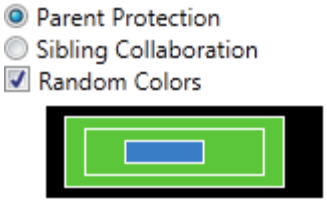
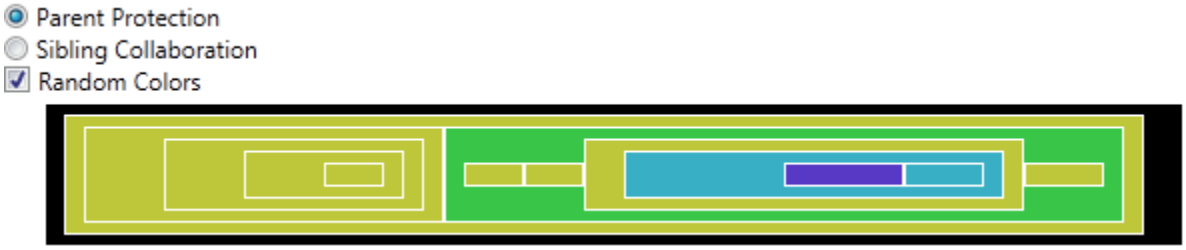


Figure 6 Original Method after Restructuring

Following figures show the placement trees for the new extracted methods.







Experiment 3

Third method used for our experiments is shown in Figure 9.

```
void doAction(ITokCollection *& pTc){
    if(p_Repos->scopeStack().size()== 0){
        return ;
    }
    if(BeginningOfScope::GetNonScopePar()){
        BeginningOfScope::BackInScope();
        return ;
    }
    element * elem = p_Repos->scopeStack().pop();
    elem->setELine(p_Repos->lineCount());
    if(elem->getType()== "function"){
        printElement(elem);
        std::string fn = p_Repos->Toker()->getCurrFileName();
        std::string temp = fn.substr(fn.find_last_of('\\')+1,fn.find_last_of('.')-fn.find_last_of('\\'));
        temp.append("xml");
        XMLCreator * xc = new XMLCreator(elem,temp);
        delete xc ;
    }
    else {
        if(p_Repos->scopeStack().size()== 0){
            return ;
        }
        if(! elem->IsInTree()){
            element * parentelem = p_Repos->scopeStack().pop();
            parentelem->addChild(elem);
            elem->setInTree(true);
            p_Repos->scopeStack().push(parentelem);
        }
        if(elem->getName()== "if"){
            p_Repos->setLatestIf(elem);
        }
    }
}
```

Figure 7 Original Method for Experiment 3

Figure 10 shows the placement tree for the method given in Figure 9.

Figure 8 Placement Tree for the Original Method

Restructured version of the method given in Figure 9 is shown in Figure 11.

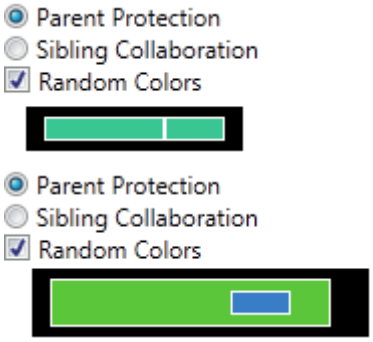
```
void ProcessElementInTree(element * elem)
  {
    if(! elem->IsInTree()){
            element * parentelem = p_Repos->scopeStack().pop();
            parentelem->addChild(elem);
            elem->setInTree(true);
            p_Repos->scopeStack().push(parentelem);
        }
        if(elem->getName()== "if"){
            p_Repos->setLatestIf(elem);
        }

  }
  bool ProcessFunctionelement(element * elem)
  {
    if(elem->getType()== "function"){
        printElement(elem);
        std::string fn = p_Repos->Toker()->getCurrFileName();
        std::string temp = fn.substr(fn.find_last_of('\\')+1,fn.find_last_of('.')-fn.find_last_of('\\'));
        temp.append("xml");
        XMLCreator * xc = new XMLCreator(elem,temp);
        delete xc ;
    }
    else {
        if(p_Repos->scopeStack().size()== 0){
            return true;
        }
         ProcessElementInTree(elem);
    }
    return false;
  }
  void doAction(ITokCollection *& pTc){
    if(p_Repos->scopeStack().size()== 0){
        return ;
    }
    if(BeginningOfScope::GetNonScopePar()){
        BeginningOfScope::BackInScope();
        return ;
    }
    element * elem = p_Repos->scopeStack().pop();
    elem->setELine(p_Repos->lineCount());
    if(ProcessFunctionelement(elem)){
            return;
    }
  }
```

Figure 9Restructured Version for Experiment 3

After restructuring we run our tool and get the following placement tree for the new version of the code.

Figure 10 Original Method After Restructuring

Following figures show the placement trees for the new extracted methods.


Parent Protection
Sibling Collaboration
Random Colors
Parent Protection
Sibling Collaboration
Random Colors