FEAT

Feature Exploration & Analysis Tool

CPRE 556- Lecture 14

Overview

- Developed University of British Columbia (UBC). Robillard Ph.D. thesis, November 2003.
- Objective: locate, analyze, and describe scattered concerns in source code.
- A plug-in for the Eclipse Platform
- http://www.cs.ubc.ca/labs/spl/projects/feat/

Basic Approach

- Understanding program = Extracting (gathering) information about a concern.
- Concern = relationships between program elements (*binary relations* as the formalism)
- Program elements = artifacts that constitute a program
- Example: artifact: *functions*, relationship: *calls*, extracted information: *call tree*.

Concern Graph

- Describes concerns in source code in terms of relations between program elements.
- Notion of consistency between a concern graph and the corresponding source code.
- FEAT: build concern graphs, view the code related to a concern, and perform analyses on a concern representation.

What is a Concern

- Any consideration a developer might have about the implementation of a program.
- For example, in a file server application based on the File Transfer Protocol, one possible concern is the requirement to log every file transfer command issued by the client programs.
- Corresponding source code = calls to functions such as log(String), and their implementations.
- This code is scattered throughout the all of the modules implementing file transfer commands.

Problems To Be Solved

- Concern Location and Understanding Problem.
- Concern Documentation Problem – consistency and cost of documentation.

Difficulties

- Scattering and tangling
  - code scattered in many different modules.
  - presence of code within a module, possibly overlapping, corresponding to different concerns.
- Causes: inadequate design, limitations of programming languages, emergence during program evolution, and code decay.

Limitations of Simple Tools

- Lexical searching tools, such as *grep* identify points in the code relevant to a concern.
- Follow a discover-and-discard model that provides little or no help for managing, understanding, and preserving the information discovered.

CG Requirements

- Language independent – applicable to concerns for code in any language.
- Flexible – to represent all kinds of program artifacts with various levels of granularity.
- Precise - a non-ambiguous mapping between CG and the corresponding source code.
- Robust - not be dependent on non-essential and brittle aspects of the source code, such as line numbers or indentation.

Formal Representation

- A mathematical model based on relational algebra.
- A named relation R_n = (n,R) consists of a name n associated with a binary relation R.
- A program model *(E,N)* consists of a set of program elements $E = \{e_1, e_2, ..., e_m\}$ and a set of named relations over E, $N = \{R_{n1}, R_{n2}, ..., R_{nk}\}$.
- Let $N = \{R_{n1}, R_{n2}, ..., R_{nk}\}$ be a set of named relations. *names(N)* = {n | there exists R : (n,R) belongs to N}.

Program Model

- Equivalent to the definition of a labeled directed graph.
- The nodes are program elements.
- A named relation represents a subset of the edges with same label.
- The name “concern graph” is thus intended to capture the idea of a graph of elements (nodes) and named relations (labeled edges) representing the subset of a program model addressing a concern.

Mapping Function

- *Let* $P_M = (E,N)$ *be a program model. The mapping function* M *consists of:*
  - A criterion defining which elements declared in program P should be listed in E.
  - A set of relation names supported by the model.
  - The definition of an analysis function a(n, P) taking as parameters a relation name n and a program P, and returning a named relation R_n contained in $E \times E$ representing the relationships between elements of P (meeting the mapping criterion), according to the semantics of n.
  - Analysis functions are defined using first-order logic.

Example 1 of Mapping Function

Mapping Function C1
E = {x | IsAFunction(x)} names(N) = {Calls, CalledBy} a(Calls, P) = {(x, y)|Calls(x, y)} a(CalledBy,P) = T(a(Calls, P))

T = Transpose

Example 2 of Mapping Function

Mapping Function J1
E = {x | IsAClass(x)} names(N) = {Declares, Extends, SuperclassOf , SubclassOf } a(Declares, P) = {(x, y) | Declares(x, y)} a(Extends, P) = {(x, y) | Extends(x, y)} a(SuperclassOf , P) = TC(a(Extends, P)) a(SubclassOf , P) = T(a(SuperclassOf , P))

TC = Transitive Closure

Illustration of Mapping J1

```
public class A
{
        int aField;
        class B {}
}
class C extends A
{
        void aMethod(){};
        class D extends A {}
        class E extends D {}
}
```

$P1_{J1}$

E_{P1} = {A, B, C, D, E}

$Declares_{P1}$ = {(A, B), (C, D), (C, E)}

$Extends_{P1}$ = {(C, A), (D, A), (E, D)}

$SuperclassOf_{P1}$ = {(C, A), (D, A), (E, D), (E, A)}

$SubclassOf_{P1}$ = {(A, C), (A, D), (D, E), (A, E)}

Motivation: Beyond Relations

- Next we will discuss fragments. Why fragments?
- Review the concept of equivalence relations:
  - A special type of relation that is reflexive, symmetric, and transitive.
  - Backbone of abstractions – gives rise to interesting partitions of a given set.
    - The set of students partitioned into teammates.
    - The set of variables partitioned into typed subsets.

Subsetting Mechanism

- Relations provide a mechanism to form interesting subsets.
- Given a set *S*, an element *x*, and a relation *R*, the subset R_X is defined as:
  - $\{Y \mid X\ R\ Y$, i.e. all elements related to X$\}$.
- If R is an equivalence relation then the subsets actually form a partition of the given set, i.e. distinct subsets are actually disjoint.

Why Fragments

- Fragments is nothing but a subsetting mechanism. It divides program artifacts into *interesting subsets*.
- We will quickly go through the rest of the slides and revisit the concept of fragments in the next lecture with a real-world program comprehension problem.

Fragments

- A fragment f_P is always defined on a program model P_M. It consists of an *intent* part and a *program subset* part.
- Example:
  - Intent = "all the subclasses of class C"
  - Program = "classes A and B"
- Formally, *Intent* consists of a (a) *domain* set, (b) a relation name, and a (c) *range* set.
- Example of Intent: For a fragment representing a function call from function *a* to function *b*, we would specify {a} as the domain, Calls as the relation name, and {b} as the range.

Projection Operator

- Introduced to define the program subset part of a fragment, it produces the relation corresponding to the intent.
- $P_M = (E_P , N_P)$ *be a program model,* Dom_P *and* Ran_P *be two subsets of* E_P *, and* n_P be *the name element of a named relation* $R_{n,P}$ *in* N_P*.*

  proj(Dom_P*,* n_P*,* Ran_P*,* P_M*)* =

  Subset of $R_{n,P}$ restricted to the given domain and range.

Formal Definition of Fragment

- $P_M = (E_P , N_P)$ *be a program model,* Dom_P *and* Ran_P *be two subsets of* E_P *, and* n_P be *the name element of a named relation* $R_{n,P}$. *We define a fragment* $f_P = (Dom_P, n_P, Ran_P, Proj_P)$ where $Proj_P =$ proj(Dom_P, n_P, Ran_P, P_M). We say f_P is defined on P_M.

Types of Domain/Range Specs

- A non-empty set of elements (e.g., Dom = {A}, Ran = {A, C, D}).
- The universal domain (or range), represented by the set E_P . Specifying E_P as the domain or range of a fragment will result in the projection including all elements in the domain of the specified relation.
- A subset specified as the range of a fragment projection. For example, to specify a domain as all of the members of class A in a program model P_M, we would specify Dom_P =
- Ran(proj({A}, Declares, E_P , P_M)).

Examples of Fragments -1

Mapping Function J2
E = {x | IsAType(x) OR IsAMethod(x)} names(N) = {I, Declares, Calls, CalledBy} I = Identity relation a(Declares, P) = {(x, y) | Declares(x, y)} a(Calls, P) = {(x, y) | CallsStatic(x, y)} a(CalledBy, P) = T(a(Calls, P))

Examples of Fragments -2

```
public class A
{
        public static void b(){};
        public static void c(){ c();b();
                D.f();}
}
class D
{
        public static void e() { f(); }
        public static void f() {}
}
```

P2$_{J2}$

E_{P2} = {A, b, c, D, e, f}
I_{P2} = {(A, A), (b, b), (c, c), (D, D), (e, e), (f, f)}
$Declares_{P2}$ = {(A, b), (A, c), (D, e), (D, f)}
$Calls_{P2}$ = {(c, b), (c, c), (c, f), (e, f)}
$CalledBy_{P2}$ = {(b, c), (c, c), (f, c), (f, e)}

Examples of Fragments - 3

Members of A

({A}, Declares,E_{P2}, {(A, b), (A, c)})

Apply the range operator to the projection of the fragment. Result?

Callers of f

({f}, CalledBy,E_{P2}, {(f, c), (f, e)})

Calls by Methods of A

(Ran(proj(({A}, Declares,E_{P2}, $P2_{J2}$))), Calls,E_{P2}, {(c, b), (c, c), (c, f)})

Mapping Functions for Java

- E = {x | IsAClass(x) OR IsAnInterface OR IsAField(x) OR IsAMethod(x)}
- names(N) = {Accesses, AccessedBy, Calls, CalledBy, Checks,
- Creates, Declares, ExtendsClass, ClassExtendedBy,
- ExtendsInterface, InterfaceExtendedBy,
- HasParameterType, HasReturnType, I, Implements,
- ImplementedBy, OfType, Overrides, OverridenBy,
- TransitivelyExtends,TransitivelyExtendedBy,
- TransitivelyImplements,TransitivelyImplementedBy}
- a(Accesses, P) = {(x, y) | Accesses(x, y)}
- a(AccessedBy, P) = T(a(Accesses, P))
- a(Calls, P) = {(x, y) | Calls(x, y)}
- a(CalledBy, P) = T(a(Calls, P))
- a(Checks, P) = {(x, y) | Checks(x, y)}
- a(Creates, P) = {(x, y) | Creates(x, y)}
- a(Declares, P) = {(x, y) | Declares(x, y)}
- a(ExtendsClass, P) = {(x, y) | ExtendsClass(x, y)}
- a(ClassExtendedBy, P) = T(a(ExtendsClass, P))
- a(ExtendsInterface, P) = {(x, y) | ExtendsInterface(x, y)}

Mapping Functions for Java-Contd.

- a(InterfaceExtendedBy, P) = T(a(ExtendsInterfaces, P))
- a(HasParameterType, P) = {(x, y) | HasParamterType(x, y)}
- a(HasReturnType, P) = {(x, y) | HasReturnType(x, y)}
- a(I, P) = {(x, y) | x = y}
- a(Implements, P) = {(x, y) | Implements(x, y)}
- a(ImplementedBy, P) = T(a(Implements, P))
- a(OfType, P) = {(x, y) | OfType(x, y)}
- a(Overrides, P) = {(x, y) | Overrides(x, y)}
- a(OverridenBy, P) = T(a(Overrides, P))
- a(TransitivelyExtends, P) = TC(a(ExtendsClass, P))
- a(TransitivelyExtendedBy, P) = T(a(TransitivelyExtends, P))
- a(TransitivelyImplements, P) = *What should it be??*

Comments on the Model

- 22 Relations with two categories - structural or behavioral.
- Structural represent static declarative relations between elements – typically, the type of relations that would be documented in a UML static structure diagram
- Behavioral relations represent code within a method. For example, the *Accesses* relation and its transpose represent code reading or writing to a field, or the *Calls* relation and its transpose represent method calls.

Model Caveats

- Intra-method elements not considered. Their hypothesis is that in most cases they are not needed to model scattered concerns.
- It does not support the distinction between different contexts in source code corresponding to a behavioral relation. For example, no distinction between calls to the same method. Their hypothesis is that context-insensitivity is a reasonable choice because when a call to a non-library method contributes to the implementation of a concern, most of the calls to that method are usually part of the concern as well.
- Does not support exception handling.

Recursive Definition of Concern

- **Definition:** *Let* P_M *be a program model. A concern* $C_P = (F_P , S_P)$ *defined on* P_M *is a tuple comprising a set of fragments* $F_P = \{f_1, f_2, ..., f_n\}$ *and a set of concerns defined on* P_M,$S_P = \{s_1, s_2, ..., s_m\}$. F_P or S_P can be empty sets.
- A fragment in F_P can also be in any sub-concern s in S_P. Fragments and concerns are composed into other concerns based on the requirements of a user. A root concern, not included in any parent concern, represents the broadest abstraction for a particular concern. It is called a *concern graph*.
- *Useful example ??*

Participants of Concern

- **Definition:** *Let* f_P = (Dom,n,Ran,Proj) *be a fragment. Then,*

  participants(f_P) = Dom(Proj) U Ran(Proj).
- Participants of a concern are obtained by union of participants of the fragments, done recursively over subconcerns.

An Example of Defining A Concern

```
public class A
{
        public static void b(){};
        public static void c(){ c();b();
                D.f();}
}
class D
{
        public static void e() { f(); }
        public static void f() {}
}
```

$P2_{J2}$

E_{P2} = {A, b, c, D, e, f}
I_{P2} = {(A, A), (b, b), (c, c), (D, D), (e, e), (f, f)}
$Declares_{P2}$ = {(A, b), (A, c), (D, e), (D, f)}
$Calls_{P2}$ = {(c, b), (c, c), (c, f), (e, f)}
$CalledBy_{P2}$ = {(b, c), (c, c), (f, c), (f, e)}

- Suppose we are interested in investigating the uses of classes A and D.

An Example of Defining A Concern – Contd.

- Suppose we are interested in investigating the uses of classes A and D.
- Based on the model $P2_{J2,}$ we first define a concern graph
  - G = (Ø, {Uses of A, Uses of D}), where both sub-concerns are currently empty.
- Next, to complete the concern graph we add fragments describing all calls to methods of class A to Uses of A, and all calls to methods of class D to Uses of D, respectively. We now have:

Uses of A = (ran(proj(({A}, Declares,E_{P2}, $P2_{J2}$))), CalledBy, E_{P2}, {(b, c), (c, c)}), Ø)

Uses of A = (ran(proj(({D}, Declares,E_{P2}, $P2_{J2}$))), CalledBy, E_{P2}, {(f, c), (f, e)}), Ø)

Concern Interaction

- **Definition:** *Let* C_P *and* D_P *be two concerns defined on a program model* $P_M = (E_P, N_P)$. *The interaction between* C_P *and* D_P *is defined as:*
  - interaction(C_P,D_P) = {(x, n, y, {(x, y)}) | x in participants(C_P) *and* y in participants(D_P) and there is a relation R_n s.t. (x, y) is in R_n.

The interaction between two concerns is a set of primitive fragments representing the relations between the participants of one concern and the participants of the other concern.

Element Set Inconsistency

- **Definition:** *Let* $P1_M = (E_{P1}, N_{P1})$ *and* $P2_M = (E_{P2}, N_{P2})$ *be the models corresponding to two versions of a program produced with the same mapping function* M. *Let* x be a subset of E_{P1}. *Then,*
  - IsInconsistent(x,$P2_M$) = x is *not* a subset of E_{P2}

Fragment Inconsistency

- **Definition:** *Let* $P1_M = (E_{P1}, N_{P1})$ *and* $P2_M = (E_{P2}, N_{P2})$ *be the models corresponding to two versions of a program produced with the same mapping function* M. *Let* $f_{P1} = (Dom_{P1}, n_{P1}, Ran_{P1}, Proj_{P1})$ be a fragment defined on $P1_M$.*Then,*
  - IsInconsistent(f_{P1}, $P2_M$) = IsInconsistent(Dom_{P1},$P2_M$) OR IsInconsistent(Ran_{P1},$P2_M$) OR

    $Proj_{P1} \neq$ proj(Dom_{P1}, n_{P1},Ran_{P1}, P_{2M}).

Concern Inconsistency

- **Definition:** *Let* $P1_M = (E_{P1}, N_{P1})$ *and* $P2_M = (E_{P2}, N_{P2})$ *be the models corresponding to two versions of a program produced with the same mapping function* M*. Let* $C_{P1} = (F_{P1}, S_{P1})$ *be a concern* defined on P1.*Then,*
  - IsInconsistent(C_{P1}, $P2_M$) if there is an inconsistent fragment in F_{P1} or there is an inconsistent sub-concern in S_{P1}.

Repair Operation

- *Let* $P1_M = (E_{P1}, N_{P1})$ *and* $P2_M = (E_{P2}, N_{P2})$ *be the models corresponding to two versions of a program produced with the same mapping function* M. *Let* $f_{P1} = (Dom_{P1}, n_{P1}, Ran_{P1}, Proj_{P1})$ be an inconsistent fragment defined on $P1_M$.*Then,*

$Repair(f_{P1,} P2_M) = (Dom_{P1}, n_{P1}, Ran_{P1}, proj(Dom_{P1}, n_{P1}, Ran_{P1}, P_{2M}))$

In informal terms, the repair function simply replaces the inconsistent projection of a fragment with a new projection consistent with the second program model

Feature Analysis and Exploration Tool

- **Model Extraction** It extracts a model of a program based on the java mapping function.
- **Concern Construction** It allows a user to build and modify concern representations by specifying fragments on the model extracted from a program. It supports the saving of a concern representation to permanent storage, and the loading of a concern representation in the tool.
- **Analysis** It supports the analysis of the interactions between different concerns. It also supports the detection and repair of inconsistencies between a concern graph and a program.

Empirical Studies

- They have evaluated the cost and usefulness of concern graphs in a series of case studies involving the evolution of five different systems of different size and style (AVID, Jex, Redback, jEdit, ArgoUML) .
- Their conclusion is that concern graphs are inexpensive to create during program investigation, can help developers perform program evolution tasks more systematically, and are robust enough to be used with different versions of a system.

Conclusions

- A formal model for describing concerns in source code. Model can be largely reused to support concern graphs in different languages.
- A specific instantiation of the model for the Java language.
- A usable tool capable of supporting the concern graph approach for Java programs.

What We Did not Cover

- An algorithm that can automatically infer concerns from a transcript of the program investigation of a developer. This algorithm serves as a proof of concept that such a technique is possible, and that it can produce documentation for scattered concerns.
- Five empirical studies of program evolution.
- A mechanism for the management of inconsistencies between a description of source code and an actual code base that can provide support for reasoning about the indirect cause of an inconsistency, in addition to the simple detection and repair of inconsistencies.

