Lecture 3: Program comprehension

- Program comprehension in software engineering
- Search Based Software Engineering for Program Comprehension
- Case studies – clustering based

Lecture 2 Machine learning in Software Engineering

- Existing approaches
- Reformulate software engineering problems
- Evaluation criteria for search based software engineering
Program comprehension

Maintenance and evolution represent important stages in the lifecycle of any software system. the two stages represent about 66% from the total cost of the software systems development.

Two subelds of software engineering which deal with activities related to software systems understanding and their structural changes are program comprehension and software reengineering.

Both subelds study activities from the maintenance and evolution phases of the software systems development process.

Reverse engineering is the process of analyzing a subject software system in order to create representations of the system at a higher level of abstraction. It can also be seen as going backwards through the development cycle.

Before modifying software systems in order to meet new requirements, the system has to be reengineered and its design has to be recovered, which can be a hard and time consuming task.

Software reengineering: inspection and modification of a software system in order to rebuild it and reimplement it in a new form.

Software reengineering consists of modifying a software system after it has been reversed engineered to understand it, in order to add new functionalities or to correct existing errors.
Program comprehension

Program comprehension ("program understanding", "source code comprehension")

A domain of computer science concerned with the ways software engineers maintain existing source code.

Software Engineering discipline which aims at understanding computer code written in a high-level programming language.

Study of cognitive and other processes involved in program understanding and maintenance.

It is necessary to facilitate reuse, inspection, maintenance, reverse engineering, reengineering, migration, and extension of existing software systems.

Program comprehension tools: aims at making the understanding of a software application easier, through the presentation of different perspectives (views) of the overall system or its components.
Recent researches in program comprehension

22th International Conference on Program comprehension (http://icpc2014.usask.ca/)

Some papers:

**How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK**
Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk (College of William and Mary, USA; University of Sannio, Italy; University of Molise, Italy)

**CODES: mining sourCe cOde Descriptions from developErs diScussions**
Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora (University of Sannio, Italy)

**Condensing Class Diagrams by Analyzing Design and Network Metrics using Optimistic Classification**
Ferdian Thung, David Lo, Mohd Hafeez Osman, and Michel R. V. Chaudron (Singapore Management University, Singapore; Leiden University, Netherlands; Chalmers, Sweden)

**Mining Unit Tests for Code Recommendation**
Mohammad Ghafari, Carlo Ghezzi, Andrea Mocci, and Giordano Tamburrelli (Politecnico di Milano, Italy; University of Lugano, Switzerland)

**Recommending Automated Extract Method Refactorings**
Danilo Silva, Ricardo Terra, and Marco Tulio Valente (Federal University of Minas Gerais, Brazil; Federal University of Lavras, Brazil)

**U Can Touch This: Touchifying an IDE**
Benjamin Biege, Julien Hoffmann, Artur Lipinski, Stephan Diehl (University of Trier, Germany)

**An Approach for Evaluating and Suggesting Method Names using N-gram Models**
Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden (University of Tokyo, Japan; National Institute of Informatics, Japan)

**Cross-Language Bug Localization**
Xin Xia, David Lo, Xingen Wang, Chenyi Zhang, and Xinyu Wang (Zhejiang University, China; Singapore Management University, Singapore)
Search-based software engineering in program comprehension

Concept location
  • identify sections of code that correspond to high-level domain concepts.

Design pattern identification
Identify hidden dependencies

Optimising Source Code for Comprehension
  • transformations to source code to improve comprehension
  • Pretty printing - produces code that is more readable
  • software visualization

Optimising Designs for Comprehension
  • modularization
  • decompose in components
  • package structure
  • transform procedural to object oriented
  • Introduce design patterns
  • refactoring identification
  • Aspect mining (Aspect oriented programming)
Program comprehension research / tools

Program comprehension tool general

- Extract information from the software system (source code, memory/stacktrace, UML)
- Store, handle, transform the extracted information
- Visualize/generate results

Search-based software engineering for program comprehension

- Extract information from the software system (source code, memory/stacktrace, UML)
- Store and handle the extracted information
- Apply machine learning techniques on the data
- Visualize/generate results
Clustering

Unsupervised classification, or clustering is a data mining activity that aims to differentiate groups (classes or clusters) inside a given set of objects.

The inferring process is, usually, carried out with respect to a set of relevant characteristics or attributes of the analyzed objects.

The resulting subsets or groups, distinct and non-empty, are to be built so that the objects within each cluster are more closely related to one another than objects assigned to different clusters.

Central to the clustering process is the notion of degree of similarity (or dissimilarity) between the objects (distance between objects).

The measure used for discriminating objects can be any metric or semi-metric function (Minkowski distance, Euclidian distance, Manhattan distance, Hamming distance, etc).

If we use vector-space model then each object is measured with respect to a set of k initial attributes, so every object is described using an k-dimensional vector.

The distance between two objects expresses the dissimilarity between them. The similarity between two objects Oi and Oj is defined as \( \frac{1}{\text{distance}(O_i, O_j)} \)
Clustering methods

A good clustering method will produce high quality clusters in which:
– the intra-class (that is, intra-cluster) similarity is high.
– the inter-class similarity is low.

Clustering methods

• partitioning algorithms
  ○ Construct various partitions and then evaluate them by some criterion

• hierarchical algorithms
  ○ Create a hierarchical decomposition of the set of data (or objects) using some criterion (agglomerative or divisive).

• density-based method
  ○ based on connectivity and density functions

• grid-based method
  ○ based on a multiple-level granularity structure

• model-based method
  ○ A model is hypothesized for each of the clusters and the idea is to find the best fit of that model to each other
**Partitioning clustering method: Kmeans, Kmedoids**

Construct a partition of a database $D$ of $n$ objects into a set of $k$ clusters

Given a $k$, find a partition of $k$ clusters that optimizes the chosen partitioning criterion.
- Global optimal: exhaustively enumerate all partitions.
- Heuristic methods: $k$-means and $k$-medoids algorithms.

Given a set of $n$ objects and a number $k$; $k \leq n$, such a method divides the object set into $k$ distinct and non-empty clusters.

The partitioning process is iterative and heuristic; it stops when a “good” partitioning is achieved.

Finding a “good” partitioning coincides with optimizing a criterion function defined either locally (on a subset of the objects) or globally (defined over all of the objects, as in $k$-means).

These algorithms try to minimize certain criteria (a squared error function); the squared error criterion tends to work well with isolated and compact clusters

- $k$-means (MacQueen’67):
  - Each cluster is represented by the center of the cluster.
- $k$-medoids or PAM (Partition around medoids) (Kaufman & Rousseeuw’87):
  - Each cluster is represented by one of the objects in the cluster.
**KMeans clustering algorithm**

The algorithm starts with \( k \) initial centroids, then iteratively recalculates the clusters (each object is assigned to the closest cluster - centroid), and their centroids until convergence is achieved.

**Algorithm**

Data: Set of objects (vector space model), \( k \) – nr of desired clusters
Results: Partition

Step 1: Pick \( k \) object from the list of objects (or generate random), the initial centroids

Step 2: Compute partition:
   - Each object will be assigned to the closest cluster
     (minimum distance between the object and the centroid)

Step 3: Compute the new list of centroids
   - for each cluster a new centroid is computed
     (average value for each attribute)

Step 5: If there is change in the centroids (or the change is greater than an epsilon) jump to Step 2

k-means algorithm minimizes the intra-cluster distance.

The main disadvantages of k-means are:
- The performance of the algorithm depends on the initial centroids. So, the algorithm gives no guarantee for an optimal solution.
- The user needs to specify the number of clusters in advance.
public Partition<ObjType> partition(List<ObjType> objects, int nrClusters, ClusteringListener<ObjType> list) {
    double epsilon = 0.001;
    List<? extends ObjectWithFeature> centroiziO = KMedoids.pickRandomSeeds(nrClusters, objects);
    Partition<ObjType> part;
    double centroidDist;
    do {
        part = KMedoids.computePartition(centroiziO, objects, list != null) {
            list.intermediatePartition(part);
        } List<Centroid> centroiziN = computeCentroizi(part);
        centroidDist = computeDistance(centroiziO, centroiziN);
        centroiziO = centroiziN;
    } while (centroidDist > epsilon);
    return part;
}
**KMedoid or PAM (Partitioning around medoids)**

Each cluster is represented by one of the objects in the cluster. It finds representative objects, called medoids, in clusters. To achieve this goal, only the definition of distance from any two objects is needed.

The algorithm starts with k initial representative objects for the clusters (medoids), then iteratively recalculates the clusters (each object is assigned to the closest cluster – medoid), and their medoids until convergence is achieved. At a given step, a medoid of a cluster is replaced with a non-medoid if it improves the total distance of the resulting clustering.

**Algorithm**

**Data:** Set of objects, k – nr of desired clusters

**Results:** Partition

**Step 1:** Pick k object from the list of objects (random), the initial medoids

**Step 2:** Compute partition:

- Each object will be assigned to the closest cluster
- (minimum distance between the object and the centroid)

**Step 3:** For each cluster

- Change the medoid and compute the cost
- Retain the partition with the smallest cost.
public Partition<T> partition(List<T> objects, int nrClusters, ClusteringListener<T> list) {
    // select initial medoids randomly
    List<T> medoids = KMedoids.<T> pickRandomSeeds(nrClusters, objects);
    // assign each object to the closest medoid
    Partition<T> part = computePartition(medoids, objects, dist);
    double cost = computeCost(part, medoids);
    boolean changed = false;
    do {
        changed = false;
        List<T> medoidCopy = new ArrayList<T>(medoids);
        for (int i = 0; i < medoids.size(); i++) {
            Cluster<T> cl = part.get(i);
            for (int j = 0; j < cl.getNRObjs(); j++) {
                // change medoid
                medoidCopy.set(i, cl.get(j));
                // compute a new partition for the new medoid list
                Partition<T> partAux = computePartition(medoids, objects, dist);
                double costAux = computeCost(partAux, medoidCopy);
                if (costAux < cost) {
                    // if the new partition is better
                    changed = true;
                    part = partAux;
                    cost = costAux;
                    medoids = new ArrayList<T>(medoidCopy);
                    notifyNewPartition(list, part);
                }
            }
        }
    } while (changed);
    return part;
}
Hierarchical Clustering Algorithms

**Agglomerative** (bottom-up): merge clusters iteratively.
- start by placing each object in its own cluster.
- merge these atomic clusters into larger and larger clusters.
- until all objects are in a single cluster.
- Most hierarchical methods belong to this category. They differ only in their definition of between-cluster similarity.

**Divisive** (top-down): split a cluster iteratively.
- It does the reverse by starting with all objects in one cluster and subdividing them into smaller pieces.
- Divisive methods are not generally available, and rarely have been applied.

Major weakness of agglomerative clustering methods:
- do not scale well: time complexity of at least $O(n^2)$, where $n$ is the number of total objects
- can never undo what was done previously.
Hierarchical agglomerative clustering sample implementation

```java
public Partition<T> partition(List<T> objects, ClusteringListener<T> list) {
    Partition<T> part = new Partition<T>(objects);
    boolean change = true;
    while (change) {
        hierarhicStep(part, list);
        if (stopC.isStopConditionReached(part)) {
            change = false;
        }
        if (list != null) {
            list.intermediatePartition(part);
        }
    }
    return part;
}
```

```java
/**
 * Join the closest two clusters
 * @param part
 */
private void hierarhicStep(Partition<T> part) {
    int nrClusters = part.getNRClusters();
    Cluster<T> minCl1 = part.get(0);
    Cluster<T> minCl2 = part.get(1);
    double dmin = dist(minCl1, minCl2, null, Double.MAX_VALUE);

    // search for the pair of clusters with minimum distance
    for (int i = 0; i < nrClusters - 1; i++) {
        Cluster<T> cl1 = part.get(i);
        for (int j = i + 1; j < nrClusters; j++) {
            double auxDist = linkMetric.metric(cl1, part.get(j));
            if (auxDist < dmin) {
                dmin = auxDist;
                minCl1 = cl1;
                minCl2 = part.get(j);
            }
        }
    }

    // join the closest clusters
    Cluster<T> c = new Cluster<T>(minCl1, minCl2);
    part.delete(minCl1);
    part.delete(minCl2);
    part.add(c);
}
```
Clustering algorithms – comparative run
Search-based software engineering in program comprehension case study: design pattern identification, refactoring identification
Design patterns identification

From a program understanding, extracting information from a design or source code is very important - localizing instances of design patterns in existing software can improve the maintainability of software.

The approach is a constraint satisfaction based approach that uses a clustering algorithm for partitioning the classes from the software system.

two clustering algorithms, a hierarchical agglomerative one and a divisive one, considering two case studies for identifying instances of Proxy and Adapter design patterns
Design patterns identification using clustering

An object oriented software system $S$ is viewed as a set of classes.

A given design pattern $p$ is described using a set of binary relations - a pair $p = (C_p, R_p)$, where

- $C_p, C_p \subseteq S$, represents the set of classes that are components of the design pattern $p$.
- $R_p$ is a set of binary constraints (relations) existing among the classes from $C_p$, constraints that characterize the design pattern $p$.

The problem of identifying all instances of the design pattern $p$ in the software system $S$ is a constraint satisfaction problem

- the problem of searching for all possible combinations of $|C_p|$ classes from $S$ such that all the constraints from $R_p$ to be satisfied.

The main goal of our clustering based approach is to reduce the time complexity ($O(n^{C_p})$) of the process of solving the analyzed problem.

Idea: to obtain a set of classes which are possible pattern participants (by applying a preprocessing step on the set $S$) and then to apply a clustering algorithm in order to obtain all instances of the design pattern $p$. 
**Design patterns identification - steps**

**Data collection**: The existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them.

**Preprocessing**: From the set of all classes from S we eliminate all the classes that cannot be part of an instance of pattern p. This preprocessing step will be explained later.

**Grouping**: The set of classes obtained after the Preprocessing step are grouped in clusters using a hierarchical clustering algorithm. The aim is to obtain clusters with the instances of p (each cluster containing an instance) and clusters containing classes that do not represent instances of p.

**Design pattern instances recovery**: The clusters obtained at the previous step are filtered in order to obtain only the clusters that represent instances of the design pattern p.
**Design patterns identification – distance**

In order to express the dissimilarity degree between any two classes relating to the considered design pattern $p$, we will consider the distance $d(C_i; C_j)$ between two classes $C_i$ and $C_j$ from $S$ given by the number of binary constraints from $R_p$ that are not satisfied by classes $C_i$ and $C_j$.

It is obvious that as smaller the distance $d$ between two classes is, as it is more likely that the two classes are in an instance of the design pattern $p$.

\[
d(C_i, C_j) = \begin{cases} 
1 + |\{k \mid 1 \leq k \leq nr_p \text{ s.t. } \neg(C_i \ r^p_k \ C_j \lor C_j \ r^p_k \ C_i)\}| \\
0
\end{cases}
\]
Design patterns identification – example

Proxy design pattern: use of proxy objects is prevalent in remote object interaction protocols (Remote proxy)

- a local object needs to communicate with a remote process hiding the details about the remote process location or the communication protocol.

proxy = (Cproxy,Rproxy), where: Cproxy = {Sbj, Prx,RSbj}; Rproxy = {r1, r2, r3}.

- r1(Sbj, Prx) - “Prx extends Sbj”.
- r2(Sbj,RSbj) - “RSbj extends Sbj”.
- r3(Prx,RSbj) - “Prx delegates any method inherited from a class C to RSbj, where both Prx and RSbj extend C”.

![Diagram of proxy pattern]
Design patterns identification - conclusion

The overall worst time complexity is $O(n^3)$ is reduced in comparison with the worst time complexity of a brute force approach ($O(n^{Cp})$).

not dependent on a particular design pattern. It may be used to identify instances of various design patterns, as any design pattern can be described as a pair (set of classes, set of relations).

may be used to identify both structural and behavioral design patterns, as the constraints can express both structural and behavioral aspects of the application classes from the analyzed software system.
Refactorings identification

The structure of a software system has a major impact on the maintainability of the system.

In order to keep the software structure clean and easy to maintain, most modern software development methodologies (extreme programming and other agile methodologies) use refactoring to continuously improve the system structure.

Refactoring is viewed as a way to improve the design of the code after it has been written. Software developers have to identify parts of code having a negative impact on the system’s maintainability, and apply appropriate refactorings in order to remove the so called “bad-smells”.

Clustering is used in order to recondition the class structure of a software system

The algorithm suggests the refactorings needed in order to improve the structure of the software system. The main idea is that clustering is used in order to obtain a better design, suggesting the needed refactorings.
Refactorings identification using clustering

A software system $S$ is viewed as a set $S = \{s_1, s_2, \ldots, s_n\}$, where $s_i$, $1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class.

Steps: **Data collection, Grouping, Refactorings extraction.**

The goal of the Grouping step is to obtain an improved structure of the existing software system.

A partitional clustering algorithm that uses an heuristic for determining the initial number of medoids.

The objects to be clustered are the entities from the software system $S$, i.e., $O = \{s_1, s_2, \ldots, s_n\}$. 
Refactorings identification using clustering – distance

The dissimilarity degree between the entities from the software system $S$ is expressed by a semi-metric function $d$, based on some relevant properties of the entities.

$$d(s_i, s_j) = \begin{cases} 
1 - \frac{|p(s_i) \cap p(s_j)|}{|p(s_i) \cup p(s_j)|} & \text{if } p(s_i) \cap p(s_j) \neq \emptyset \\
\infty & \text{otherwise}
\end{cases}$$

$S_i$ can be a class, a method or an attribute. $d$ highlights the concept of cohesion, i.e., entities with low distances are cohesive, whereas entities with higher distances are less cohesive.

Identified refactorings: Move Method, Move Attribute, Inline Class, Extract Class.
Refactorings identification – case study

Case study: the open source software JHotDraw, version 5.1 - it is well-known as a good example for the use of design patterns and as a good design.

Results: the algorithm determines an identical structure with the current structure of JhotDraw.

The second case study: a real software system: is DICOM (Digital Imaging and Communications in Medicine) and HL7 (Health Level 7) compliant PACS (Picture Archiving and Communications System) system, facilitating the medical images management, offering quick access to radiological images, and making the diagnosing process easier.

The algorithm applied on one of the subsystems from this application: 1015 classes, 8639 methods and 4457 attributes.

84 refactorings were suggested: 6 Move Attribute, 76 Move Method, and 1 Inline Class 25 accepted, 18 acceptable, 41 rejected.
Refactorings identification using clustering- overview
How to extract information from source code

- Textual search, regular expressions
- Source code parser
- Introspection
- Instrumentation
- Byte code analysis