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

Both subfields 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, 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 activities software engineers perform in order to 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

IEEE International Conference on Program Comprehension 2017

Fabio Palomba, Andy Zaidman, Rocco Oliveto and Andrea De Lucia
An Exploratory Study on the Relationship between Changes and Refactoring

Daniel Almeida, Gail Murphy, Greg Wilson and Mike Hoye
Do Software Developers Understand Open Source Licenses?

Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu and Jian Lu
RepDroid: An Automated Tool for Android Application Repackaging Detection

Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach and Dror Feitelson
Meaningful Identifier Names: The Case of Single-Letter Variables

Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci and Andy Zaidman
Developer-Related Factors in Change Prediction: An Empirical Assessment

Markus Borg, Emil Alégroth and Per Runeson
Software Engineers' Information Seeking Behavior in Change Impact Analysis – An Interview Study

Yutian Tang and Hareton Leung
Constructing Feature Model by Identifying Variability-aware Module

Yikun Hu, Yuanyuan Zhang, Juanru Li and Dawu Gu
Binary Code Clone Detection across Architectures and Compiling Configurations

Héctor Adrián Valdecantos, Katy Tarrit, Mehdi Mirakhorli and James O. Coplien
An Empirical Study on Code Comprehension: Data Context Interaction compared to classical Object Oriented

An Lam, Anh Nguyen, Hoan Nguyen and Tien Nguyen
Bug Localization with Combination of Deep Learning and Information Retrieval

Rubén Saborido Infantes, Foutse Khomh, Giuliano Antoniol and Yann-Gaël Guéhéneuc
Comprehension of Ads-supported and Paid Android Applications: Are They Different?

Elizabeth Poché, Nishant Jha, Grant Williams, Jazmine Staten, Miles Visper and Anas Mahmoud
Analyzing User Comments on YouTube Coding Tutorial Videos

Eran Avidan and Dror Feitelson
Effects of Variable Names on Comprehension: An Empirical Study

Tao Zhang, Jiachi Chen, He Jiang, Xiapu Luo and Xin Xia
Bug Report Enrichment with Application of Automated Fixer Recommendation

Mariano Ceccato, Paolo Tonella, Aldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano
How Professional Hackers Understand Protected Code while Performing Attack Tasks

Tung Dao, Lingming Zhang and Na Meng
How Does Execution Information Help with Information-Retrieval Based Bug Localization?

Fiorella Zampetti, Luca Ponzanelli, Andrea Mocci, Gabriele Bavota, Massimiliano Di Penta and Michele Lanza
How Developers Document Pull Requests with External References

Manishankar Mondal, Chanchal K. Roy and Kevin Schneider
Identifying Code Clones having High Possibilities of Containing Bugs

Mario Hozano, Alessandro Garcia, Nuno Antunes, Baldoino Fonseca and Evandro Costa
Smells are sensitive to developers! On the efficiency of (un)guided customized detection

Nevena Milojković, Mohammad Ghafari and Oscar Nierstrasz
Exploiting Type Hints in Method Argument Names to Improve Lightweight Type Inference

Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand and Andrzej Wasowski
Variability through the Eyes of the Programmer

Mivian Ferreira, Kecia Ferreira and Marco Tulio Valente
A Comparison of Three Algorithms for Computing Truck Factors

Bin Lin, Luca Ponzanelli, Andrea Mocci, Gabriele Bavota and Michele Lanza
On the Uniqueness of Code Redundancies

Boyang Li, Denys Poshyvanyk and Mark Grechanik
Automatically Detecting Integrity Violations In Database-Centric Applications

Shulamyt Ajami, Yonatan Woodbridge and Dror Feitelson
Syntax, Predicates, Idioms -- What Really Affects Code Complexity?

Bas Jansen and Felienne Hermans
The Effect of Delocalized Plans on Spreadsheet Comprehension - A Controlled Experiment

Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia and Rohit Gheyi
The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter

Shaikh Mostafa, Rodney Rodriguez and Xiaoyin Wang
NetDroid: Summarizing Network Behavior of Android Apps for Network Code Maintenance

IEEE International Conference on Program Comprehension **2015**

(https://dibt.unimol.it/ICPC15/Home.html)

Some papers:

I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time
Roberto Minelli, Andrea Mocci, Michele Lanza

Generating Refactoring Proposals to Remove Clones from Automated System Tests
Benedikt Hauptmann, Sebastian Eder, Maximilian Junker, Elmar Juergens, Volkmar Woinke

Fault Localization during System testing
Pavan Kumar Chittimalli, Vipul Shah

RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information
Tien-Duy B. Le, Mario Linares Vasquez, David Lo, Denys Poshyvanyk

Generating Reproducible and Replayable Bug Reports from Android Application Crashes
Martin White, Mario Linares Vasquez, Peter Johnson, Carlos Bernal-Cardenas, Denys Poshyvanyk

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

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

Optimizing 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/stack trace, 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/stack trace, UML)
- Store and handle the extracted information
- Apply machine learning techniques on the data
- Visualize/generate results

Clustering

Unsupervised clasification, 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 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 1/distance(Oi ,Oj)

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 valuate 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 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.

KMeans sample implementation

```
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,dist);
                  if (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)
Step3: For each cluster
            Change the medoid and compute the cost
            Retain the partition with the smallest cost.
```

KMedoid sample implementation

```java
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 $O(n2)$, where n is the number of total objects
- can never undo what was done previously.

Hierarchical agglomerative clustering sample implementation

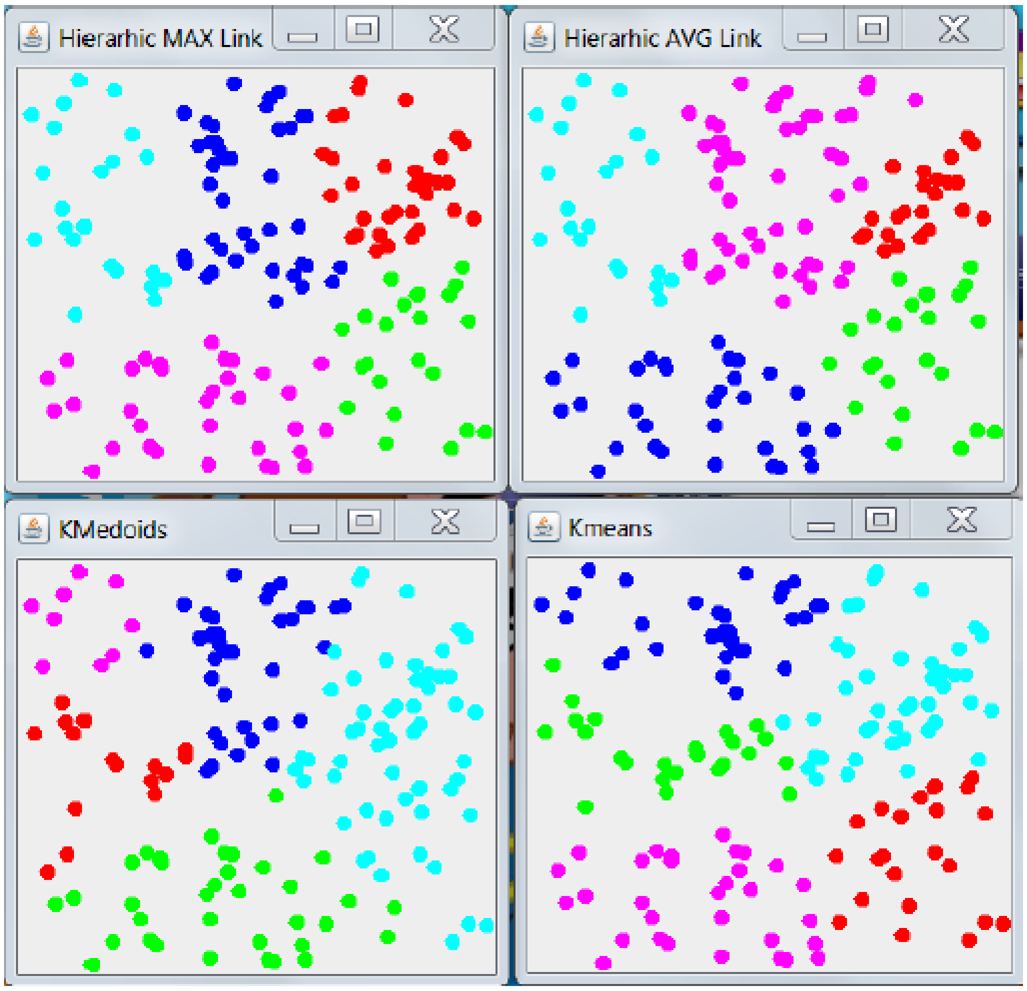
```java
public Partition<T> partition(List<T> objects, ClusteringListener<T> list) {
		Partition<T> part = new Partition<T>(objects);

		boolean change = true;
		while (change) {
			hierarhicStep(part, list);
			if (stopC.isStopConditionReached(part)) {
				change = false;
			}
			if (list != null) {
				list.intermediatePartition(part);
			}
		}
		return part;
	}

	/**
	 * 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

Software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Design patterns gained popularity in computer science after the book Design Patterns: **Elements of Reusable Object-Oriented Software** was published in 1994 by the so-called "Gang of Four" (Gamma et al.), which is frequently abbreviated as "GoF".

GoF: Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John

Design patterns

Design patterns were originally grouped into the categories:

- creational patterns
  - **Abstract factory**: provides an interface for creating related or dependent objects without specifying the objects' concrete classes.
  - **Builder pattern**: separates the construction of a complex object from its representation so that the same construction process can create different representations.
  - **Factory method**: allows a class to defer instantiation to subclasses.
  - **Prototype**: specifies the kind of object to create using a prototypical instance and creates new objects by cloning this prototype.
  - **Singleton**: ensures that a class only has one instance and provides a global point of access to it.
- structural patterns:
  - **Adapter**: 'adapts' one interface for a class into one that a client expects
  - **Aggregate**: a version of the Composite pattern with methods for aggregation of children
  - **Bridge**: decouple an abstraction from its implementation so that the two can vary independently
  - **Composite**: a tree structure of objects where every object has the same interface
  - **Decorator**: add additional functionality to a class at runtime where sub classing would result in an exponential rise of new classes
  - **Proxy**: a class functioning as an interface to another thing
- behavioral patterns:
  - **Strategy**: Algorithms can be selected on the fly, using composition
  - **Template method**: Describes the program skeleton of a program; algorithms can be selected on the fly, using inheritance
  - **Visitor**: A way to separate an algorithm from an object
  - **Observer**: a.k.a. Publish/Subscribe or Event Listener. Objects register to observe an event that may be raised by another object

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 = (Cp,Rp), where

- Cp, Cp $\subset$ S, represents the set of classes that are components of the design pattern p.
- Rp is a set of binary constraints (relations) existing among the classes from Cp, 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 |Cp| classes from S such that all the constraints from Rp to be satisfied.

The main goal of our clustering-based approach is to reduce the time complexity ($O(n^{|Cp|})$) 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 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 can not 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 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(Ci;Cj) between two classes Ci and Cj from S given by the number of binary constraints from Rp that are not satisfied by classes Ci and Cj .

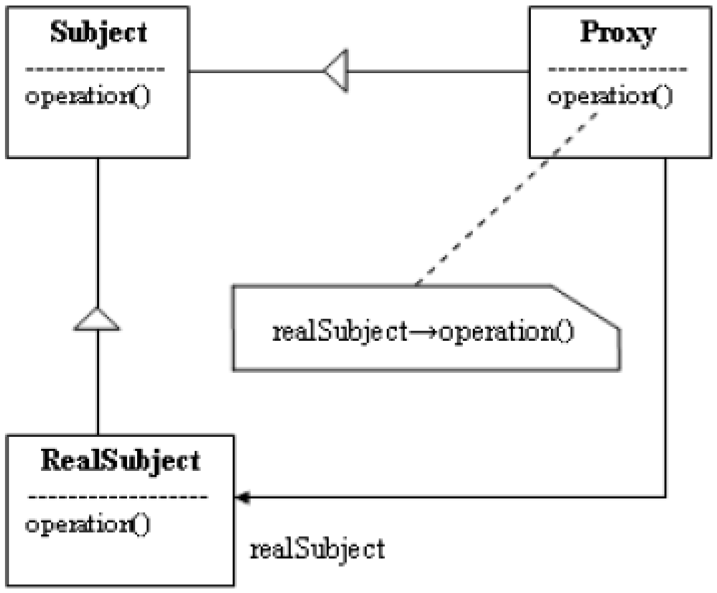
 It is obvious that as smaller the distance d between two classes is, as it is more likely that the two classes are in an instance of the design pattern p.

$$d(C_i, C_j) = \begin{cases} 1 + |\{k \mid 1 \leq k \leq nr_p \ s.t. \ \neg(C_i \ r_k^p \ C_j \vee C_j \ r_k^p \ C_i)\}| \\ 0 \end{cases}$$

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”.

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.

Refactoring

Refactoring: Improving the Design of Existing Code by **Martin Fowler**, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, don Roberts

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

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

It is a disciplined way to clean up code that minimizes the chances of introducing bugs.

When you refactor, you are improving the design of the code after it has been written.

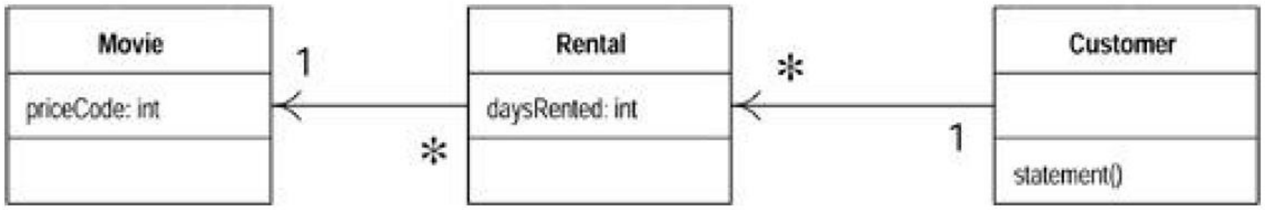
Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design.

It is the exact reverse of the normal notion of software decay.

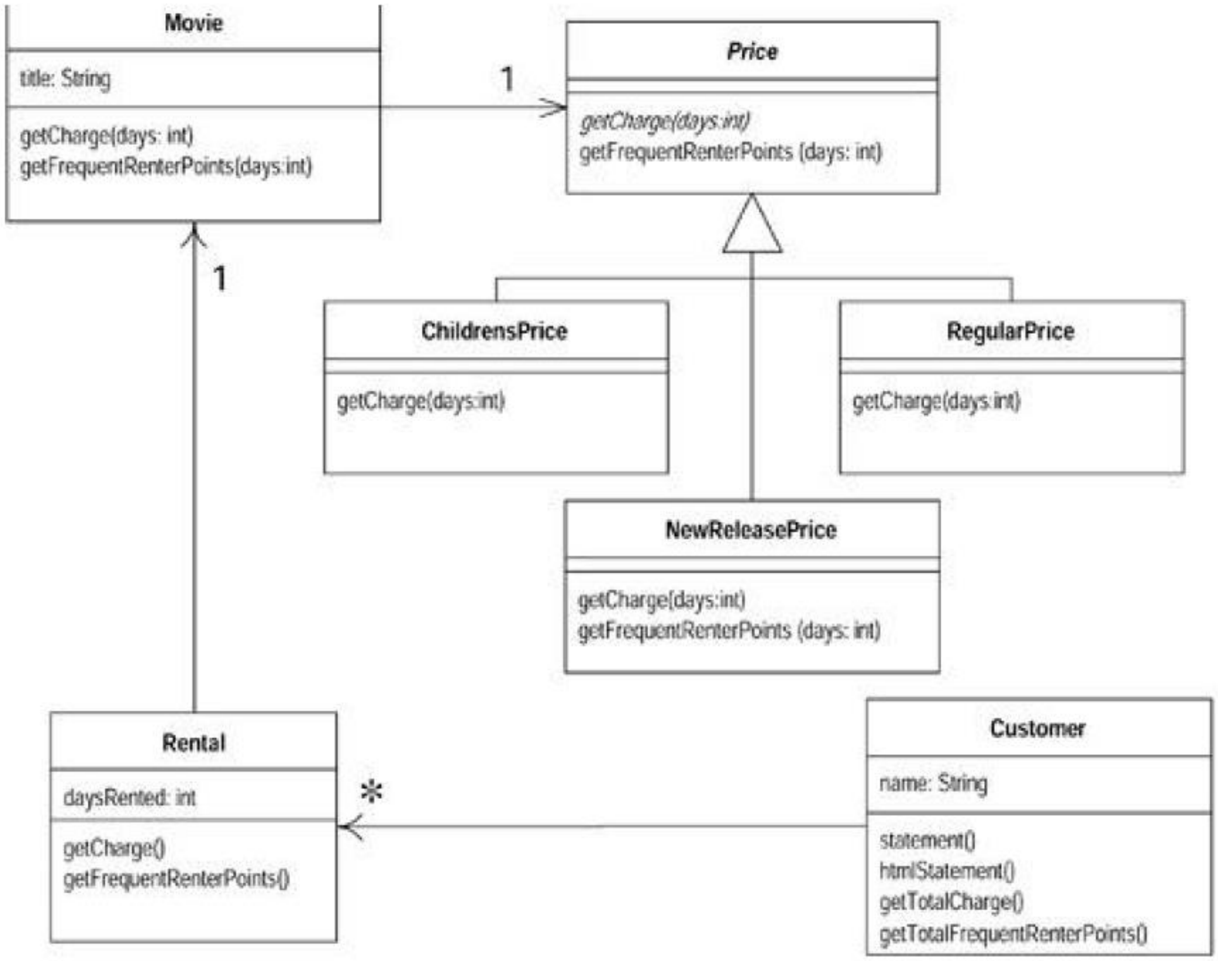
software decay - slow deterioration of software performance over time or its diminishing responsiveness that will eventually lead to software becoming faulty, unusable.
Software decays over time due to changes in the code and the technology in which it operated. Example: fixing a single issue can produce several other defects.

Technical dept – metaphor by Ward Cunningham, reflects the implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer
Technical dept – the refactoring effort needed to add a feature non-invasively

Sample Refactoring Workflow
Changes: Add HTML Rendering – Statements printed in HTML
Change the movie classification and charging rules.



1 **Extract method**: Customer.amountFor(Rental)
2 **Move method**: Customer.amountFor -> Rental.getCharge
3 **Replace temp with query**: In Cosutmer inline temp amount
4 **Extract method**, simplify, move: Customer.getFrequentRenterPoints -> Rental
5 **Replace temp with query**: Customer.statement
6 Add htmlStatement method to Customer
7 **Replace conditional with polymorphism**: Move getCharge to Movie, move getFrequentRenterPoints to Movie, Add Price, ChildrenPrice, RegulaPrice, etc…



Why Refactor

- Refactoring improves the design
- Refactoring make software easier to understand
- Refactoring helps find bugs
- Refactoring helps program faster

Bad smells in code

Duplicated Code
Long Method
Large Class
Long Parameter List
Divergent Change
Shotgun Surgery
Feature Envy
Data Clumps
Primitive Obsession
Switch Statements
…..

Refactorings identification

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 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 to obtain a better design, suggesting the
needed refactorings.

Refactorings identification using clustering

A software system S is viewed as a set S = {s1, s2, ..., sn}, where si, $1 \leq i \leq n$ 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 = {s1, s2, . . . , sn}.

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.

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

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.

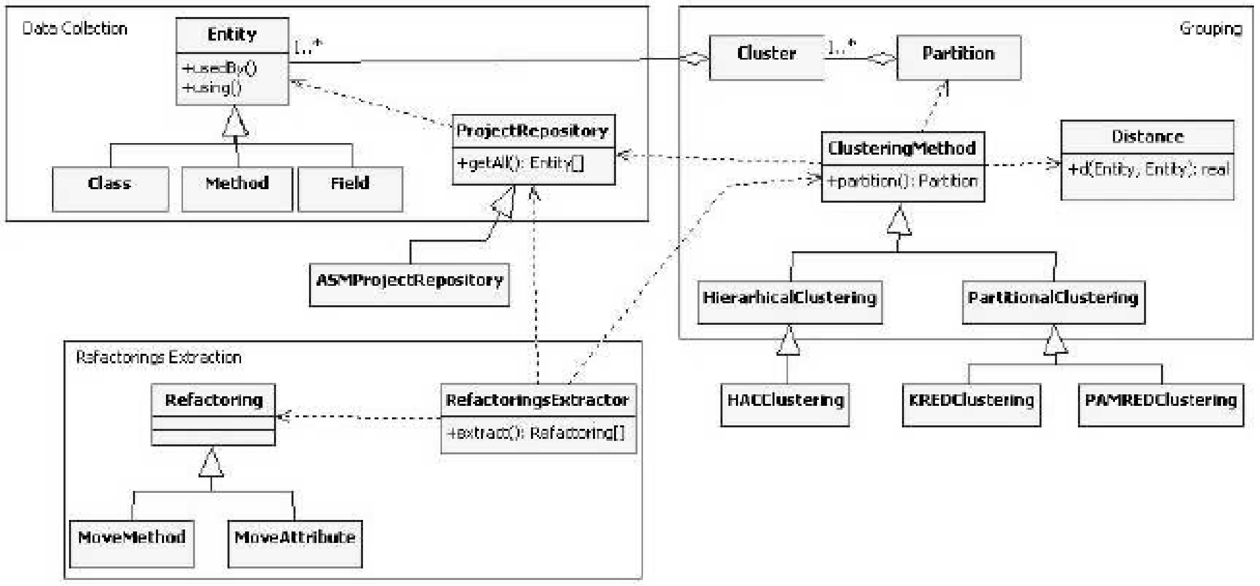
The second case study: a real software system: is DICOM (Digital Imaging and Communications in Medicine) and HL7 (Health Level 7) compliant PACS (Picture Archiving and Communications System) system, facilitating the medical images management, offering quick access to radiological images, and making the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