Computational Intelligence in Software Engineering

Computational Intelligence in Software Engineering

Will investigate the use of search techniques for solving several software engineering problems - *search based software engineering*.

Search based software engineering is a research and application domain that has many unresearched directions, and many already studied fields

Ideea: solving software engineering problems using articial intelligence techniques, particularly machine learning techniques.

Search Based Software Engineering

Reformulating software engineering problems as search problems.

Metaheuristic search techniques are used in solving different software engineering problems: testing, module clustering, cost estimation, requirements analysis, systems integration, software maintenance and evolution of legacy systems.

Harman and Jones (2001): Software engineering is ideal for the application of metaheuristic search techniques, such as genetic algorithms, simulated annealing and tabu search
Mark Harman and Bryan F. Jones. Search-based software engineering. Information & Software Technology, 43(14):833{839, 2001.

Approaching software engineering problems as search problems allows solving these problems using computational inteligence and consequently benefits from the advantages offered by AI.

search-based techniques provide solution to the dicult problem of balancing competing (and sometimes inconsistent) constraints and may suggest ways of finding acceptable solutions in situations where perfect solutions are either theoretically impossible or practically infeasible.

Machine Learning

Machine learning is an important research direction in Articial Intelligence.

Machine learning explores the construction and study of algorithms that can learn from data

Machine learning is the study of systems models that, based on a set of data (training data), improve their performance by experiences and by learning some specic domain knowledge.

there are three basic machine learning strategies:

- supervised learning (the learning strategy that is supervised by a teacher)
- unsupervised learning (learning without supervision)
- reinforcement learning (learning by interaction with the environment).

Machine learning techniques

Decision tree learning

Decision tree learning uses a decision tree as a predictive model, which maps observations about an item to conclusions about the item's target value.

Association rule learning

Association rule learning is a method for discovering interesting relations between variables in large databases.

Artificial neural networks

An artificial neural network (ANN) learning algorithm, usually called "neural network" (NN), is a learning algorithm that is inspired by the structure and functional aspects of biological neural networks. Computations are structured in terms of an interconnected group of artificial neurons, processing information using a connectionist approach to computation. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs, to find patterns in data, or to capture the statistical structure in an unknown joint probability distribution between observed variables.

Support vector machines

Support vector machines (SVMs) are a set of related supervised learning methods used for classification and regression. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that predicts whether a new example falls into one category or the other.

Machine learning techniques

Clustering (Unsupervised classication**)**
Cluster analysis is the assignment of a set of observations into subsets (called clusters) so that observations within the same cluster are similar according to some predesignated criterion or criteria, while observations drawn from different clusters are dissimilar. Different clustering techniques make different assumptions on the structure of the data, often defined by some similarity metric and evaluated for example by internal compactness (similarity between members of the same cluster) and separation between different clusters. Other methods are based on estimated density and graph connectivity. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis.

Reinforcement learning

Reinforcement learning is concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward. Reinforcement learning algorithms attempt to find a policy that maps states of the world to the actions the agent ought to take in those states. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

Software Engineering

Design, development, and maintenance of software

Application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software

Software development life cycle:

1. Requirement gathering and analysis
2. Design
3. Implementation or coding
4. Testing
5. Deployment
6. Maintenance

Common software development metodologies:

- waterfall
- prototyping
- iterative and incremental development
- feature driven development
- spiral development
- rapid application development
- extreme programming
- agile methodology
- ...

Software Engineering subdisciplines

Requirements engineering: The elicitation, analysis, specification, and validation of requirements for software.

Software design: The process of defining the architecture, components, interfaces, and other characteristics of a system or component. It is also defined as the result of that process.

Software construction: The detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

Software testing: An empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.

Software maintenance: The totality of activities required to provide cost-effective support to software.

Software configuration management: The identification of the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the system life cycle.

Software engineering management: The application of management activities—planning, coordinating, measuring, monitoring, controlling, and reporting—to ensure that the development and maintenance of software is systematic, disciplined, and quantified.

Software engineering process: The definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle process itself.

Software engineering tools and methods: The computer-based tools that are intended to assist the software life cycle processes (see Computer-aided software engineering) and the methods which impose structure on the software engineering activity with the goal of making the activity systematic and ultimately more likely to be successful.

Software quality management: The degree to which a set of inherent characteristics fulfills requirements.

Software engineering activities

Requirements analysis
- cost/time estimation
- treacability

Design
- decompose system into subsistems
- components, classes, state machines

Coding
- Implement new functionalities
- Repair bugs
- Refactoring
- Source code repository
- Continuous integration

Testing
- create test cases/unit tests
- integration testing
- system testing

Deployment
- configuration management
- application monitoring

Maintanance
- program comprehension
- refactoring/improooving overall design
- fix bugs/create testcases
- concept location
- software visualisation

Data available to use

Any artifact produced during the development of a software system.

Design documents

UML Diagrams- class diagrams, sequence diagrams, use case diagrams etc.

Functional/arhitectural/user interface design documents

Source code

instructions, methods, classes

comments

modules, package structure

test functions, test cases

source code files (size, creation date/time, modification date/time)

Project management data

tasks, dependencies, allocation

task estimated/actual time

Source code repository

comments entered for checkins

commit/update date/time, persin, location, team member

modified/moved/removed files

Configuration/deployment

config files, build files (ant, maven,

Data we can produce/collect

Software metrics

- method/class/package level metrics

Runtime data

- stack traces
- code coverage
- memory allocation/usage
- error conditions

Project management data

- project velocity

User data

- survay
- user behavior (clicks, selects, navigation)

Computational Intelligence in Software Engineering

We present some existing work in the search-based software engineering literature.

Existing approaches we are focusing:

- Refactoring
- Defect Detection and Prediction
- Object-oriented transformation
- Behavioral adaptation of software systems
- Software Testing
- Software Vizualization
- Effort prediction and Cost estimation
- Software Reuse
- Design Patterns identification

Machine learning in Software Engineering

ML algorithms can be used in tackling software engineering problems.

ML algorithms not only can be used to build tools for software development and maintenance tasks, but also can be incorporated into software products to make them adaptive and self-configuring.

A maturing software engineering discipline will definitely be able to benefit from the utility of ML techniques.

Search Based Software Engineering

Reformulating software engineering problems as search problems

Machine learning approaches for software engineering

- Requirement engineering - AL, BBN, LL, DT, ILP
- Rapid prototyping - GP
- Component reuse - IBL (CBR4)
- Cost/effort prediction - IBL (CBR), DT, BBN, ANN
- Defect prediction – BBN, AR
- Design defect detection - AR
- Test oracle generation - AL (EBL5)
- Test data adequacy - CL
- Validation – AL
- Restructuring/Refactoring CU
- Design patterns – CU
- Data structures – ANN, SVM
- Reverse engineering – CL

Major types of learning:

- concept learning (CL)
- decision trees (DT)
- artificia neural networks (ANN)
- Bayesian belief networks (BBN)
- reinforcement learning (RL)
- genetic algorithms (GA) and genetic programming (GP)
- instance-based learning (IBL)
- inductive logic programming (ILP)
- analytical learning (AL)
- support vector machines (SVM)
- association rules (AR)
- clustering (CU)

Component reuse

Component retrieval from a software repository is an important issue in supporting software reuse.

Formulated into an instance-based learning problem:
1. Components in a software repository are represented as points in the n-dimensional Euclidean space (or cases in a case base).

2. Information in a component can be divided into *indexed* and *unindexed* information (attributes). Indexed information is used for retrieval purpose and unindexed information is used for contextual purpose.

3. Queries to the repository for desirable components can be represented as constraints on indexable attributes.

4. Similarity measures for the nearest neighbors of the desirable component can be based on the standard Euclidean distance, distance-weighted measure, or symbolic measure.

5. The possible retrieval methods include: *K-Nearest Neighbor*, *inductive retrieval*, *Locally Weighted Regression*.

6. The adaptation of the retrieved component for the task at hand can be *structural* (applying adaptation rules directly to the retrieved component), or *derivational* (reusing adaptation rules that generated the original solution to produce a new solution).

Rapid prototyping/Generating programs

Rapid prototyping - tool for understanding and validating software requirements.

In genetic programming (GP), a computer program is often represented as a tree:

- nodes are functions
- leaf nodes are input to functions

Start with a random generated tree (program), GP generates the final computer program.

The framework of a GP-based rapid prototyping process can be described as follows:

1. Define sets of functions and terminals to be used in the developed (prototype) systems.

2. Define a *fitness* function to be used in evaluating the worthiness of a generated program. Test data (input values and expected output) may be needed in assisting the evaluation.

3. Generate the initial program population.

4. Determine selection strategies for programs in the current generation to be included in the next generation population.

5. Decide how the genetic operations (*crossover* and *mutation*) are carried out during each generation and how often these operations are performed.

6. Specify the terminating criteria for the evolution process and the way of checking for termination.

7. Translate the returned program into a desired programming language format.

Requirement engineering

Requirement engineering refers to the process of establishing the services a system should provide and the constraints under which it must operate.

A requirement may be functional or non-functional. A functional requirement describes a system service or function, whereas a non-functional requirement represents a constraint imposed on the system.

Functional requirements can be “learned” from the data if there are empirical data from the problem domain that describe how the system should react to certain inputs .

1. Let X and C be the domain and the co-domain of a system function f to be learned. The data set D is defined as: D = {<xi, ck>| xi in X and ck in C}.

2. The target functions f to be learned is such that any xi in X and any ck in C, f(xi) = ck .

3. The learning methods applicable here have to be of supervised type. Depending on the nature of the data set D, different learning algorithms (in AL, BBN, CL, DT, ILP) can be utilized to capture (learn) a system’s functional requirements.

Reverse engineering (program comprehension and understanding)

Recover the design or specification of a legacy system from its source or executable code

Legacy systems are old systems that are critical to the operation of an organization which uses them and that must still be maintained. They may be poorly structured and their documentation may be either out-of-date or non-existent.

Deriving functional specification of a legacy software system from its executable code.

1. Given the executable code p and its input data set X, and output set C, the training data set D is defined as: D = {< xi, p(xi)>| xi in X and p(xi) in C}.

2. The process of deriving the functional specification f for p can be described as a learning problem in which f is learned through some ML algorithm such that any xi in X [f(xi) = p(xi)].

3. Many supervised learning methods can be used here (e.g., CL).

Validation

Check a software implementation against its specification

If the specification and the executable code is available validationc can be performed as an analytic learning task as follows:

1. Let X and C be the domain and co-domain of the implementation (executable code) p, which is defined as: p: X -> C.

2. The training set D is defined as: D = {<xi, p(xi)>| xi in X }.

3. The specification for p is denoted as B, which corresponds to the domain theory in the analytic learning.

4. The validation checking is defined to be: p is valid if any <xi, p(xi)> in D [B and xi imply p(xi)].

5. Explanation-based learning algorithms can be utilized to carry out the checking process.

Software defect prediction

predict the likely delivered quality and maintenance effort before software system is deployed

size, complexity, testing metics, proces quality must be taken into consideration in order for the defect prediction to be successful.

compute the probability distribution for any subset of variables given the values or distributions for any subset of the remaining variables - Bayesian Belief Networks (BBN)

specifies the probability that the variable will take on each of its possible values (e.g., “very low”, “low”, “medium”, “high”, or “very high” for the variable “Defects Detected”) given the observed values of the other variables (complexity, metrics, etc)

When using a BBN for a decision support system such as software defect prediction, the steps below should be followed.

1. Identify variables in the BBN. Variables can be:
    - *hypothesis* variables for which the user would like to find out their probability distributions (hypothesis variable are either unobservable or too costly to observe),
    - *information* variables that can be observed
    - *mediating* variables that are introduced for certain purpose (help reflect independence properties, facilitate acquisition of conditional probabilities, and so forth).

2. Define the proper causal relationships among variables. These relationships also should capture and reflect the causality exhibited in the software life-cycle processes.

3. Acquire a probability distribution for each variable in the BBN.

Software defect prediction

use any supervised learning technique

- Collect data (measurement, metric, resource allocation) about defective systems, components, classes, modules. The measurements will be used as attribute values

- train the model using well know defective/un-defective system/component/ class

- The model can be used to predict defect rate for new systems (other than the ones used in the training phase)

Project effort (cost) prediction

approach to the project effort prediction using instance-based learning.

1. Introduce a set of features or attributes (e.g., number of interfaces, size of functional requirements, development tools and methods, and so forth) to characterize projects.

2. Collect data on completed projects and store them as instances in the case base.

3. Define similarity or distance between instances in the case base according to the symbolic representations of instances (e.g., Euclidean distance in an n-dimensional space where n is the number of features used). To overcome the potential curse of dimensionality problem, features may be weighed differently when calculating the distance (or similarity) between two instances.

4. Given a query for predicting the effort of a new project, use an algorithm such as Knearest Neighbor, or, Locally Weighted Regression to retrieve similar projects and use them as the basis for returning the prediction result.

Software restructuring/refactoring/transformation

Model the system as a set of objects.

Object can be a class, a module, a function/method, a package.

Use a clustering algorithm to group the objects.

Using the optained partition one can:

- identify the new/better structure of the system
- derive transformations for the analyzed system
- identify components, concepts or other higher level abstraction in the source code (i.e design patterns)

Object in the software system can be described by a set of features (i.e. software metric) -> we can use Euclidian or other distances

We can define custom distances between the objects. We can encapsulate domain knowledge in the definition of the distance used in the clustering algorithm.

Some Existing Work

Scenario-based requirement engineering

Formal method for supporting the process of inferring specifications of system goals and requirements inductively from interaction scenarios provided by stakeholders.

The method is based on a learning algorithm that takes scenarios as examples and counter-examples (positive and negative scenarios) and generates goal specifications as temporal rules.

Software project effort estimation

Instance-based learning techniques are used in for predicting the software project effort for new projects.
The empirical results obtained (from nine different industrial data sets totaling 275 projects) indicate that cased-based reasoning offers a viable complement to the existing prediction and estimations techniques.

Decision trees (DT) and artificial neural networks (ANN) are used to help predict software development effort. The results were competitive with conventional methods such as COCOMO and function points. The main advantage of DT and ANN based estimation systems is that they are adaptable and nonparametric.

Software defect prediction

Bayesian belief networks are used to predict software defects.
incorporating multiple perspectives on defect prediction into a single, unified model.

Variables are chosen to represent the life-cycle processes of

- specification
- design
- implementation
- testing

(Problem-Complexity, Design-Effort, Design-Size, Defects-Introduced, Testing-Effort, Defects-Detected, Defects-Density-At-Testing, Residual-Defect-Count, and Residual-Defect-Density).

Evaluation criteria for search based software engineering

Criterias that capture essential aspects/goals for any search based approach

Base line validity

The solution should be able to find better solutions or find the in less computational effort than random search

In some case maybe we just have to change the choise of search technique

Discovery of known solutions

There are situations where there is no known general algoritm for solving a problem.

We can validate the search-based approach by showing that the metaheuristic search is able to find solutions that compare well with known individual solutions.

Discovery of desirable solutions

Gather empirical data to provide evidence of the kind of solution which may be obtained.

Efficiency

In many cases a search-based approach is slower than existing analytical approaches (search involve repeated trails)

If the search-based approach produce better solutions then it can be a valuable tool where quality overrides speed.

Validation with respect to existing analytical techniques (no search-based)

Even if there are non search-based approches the search-based variant may still be usefoul if existent approaches:

- only work on subset of the problem space
- solution is not consistent, we can use the search-based variant as a "second guess"
- only produce sub-optimal solutions, we can use search-based variant to improve on the known solution or to produce better solutions

Psychological consideration

aesthetic application of metaheuristic search

(semi) automatization of common software engineering task

We can use search-based to better understand solutions

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 modication 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 improove 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 indentification
- 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 classication, or clustering is a data mining activity that aims to dierentiate 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 dierent 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 describied 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 methosd

- 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 dened 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 denition 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

```
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.

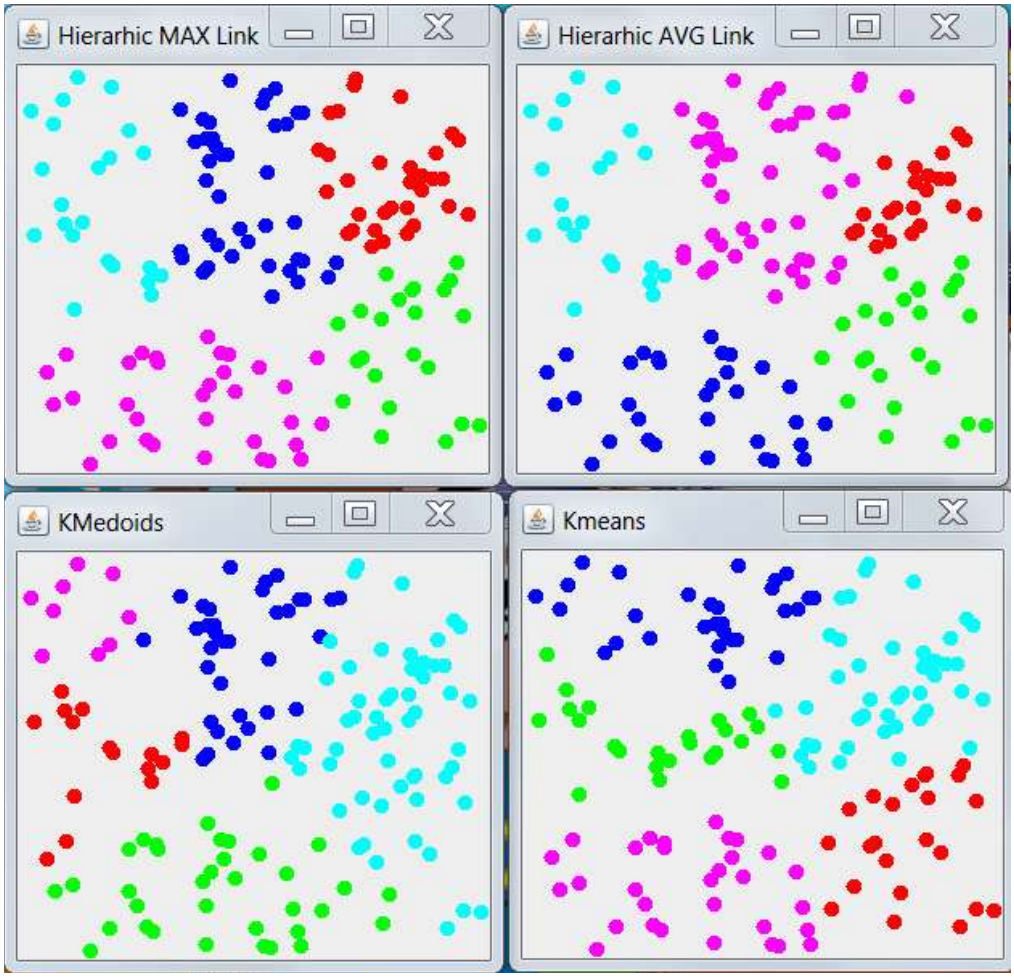
Major weakness of agglomerative clustering methods:

- 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

```
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 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 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 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 ltered 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(Ci;Cj) between two classes Ci and Cj from S given by the number of binary constraints from Rp that are not satised by classes Ci and Cj .

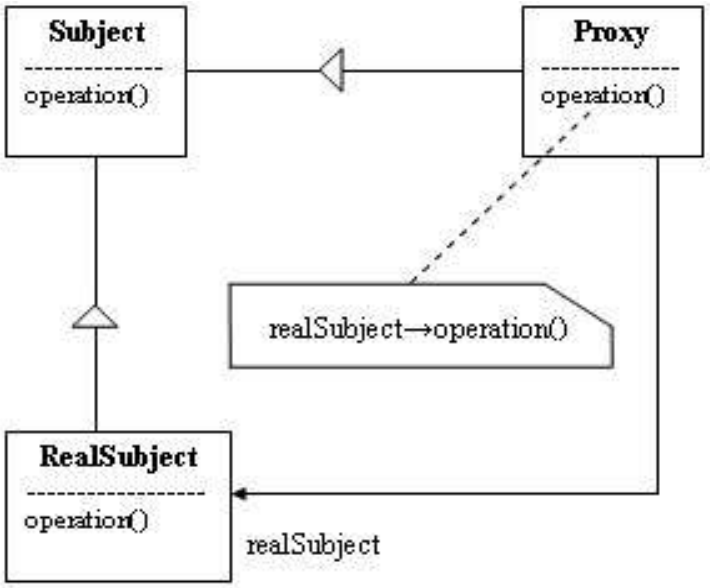
It is obvious that as smaller the distance d between two classes is, as it is more likely that the two classes are in an instance of the design pattern p.

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

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.

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 = {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.

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

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.

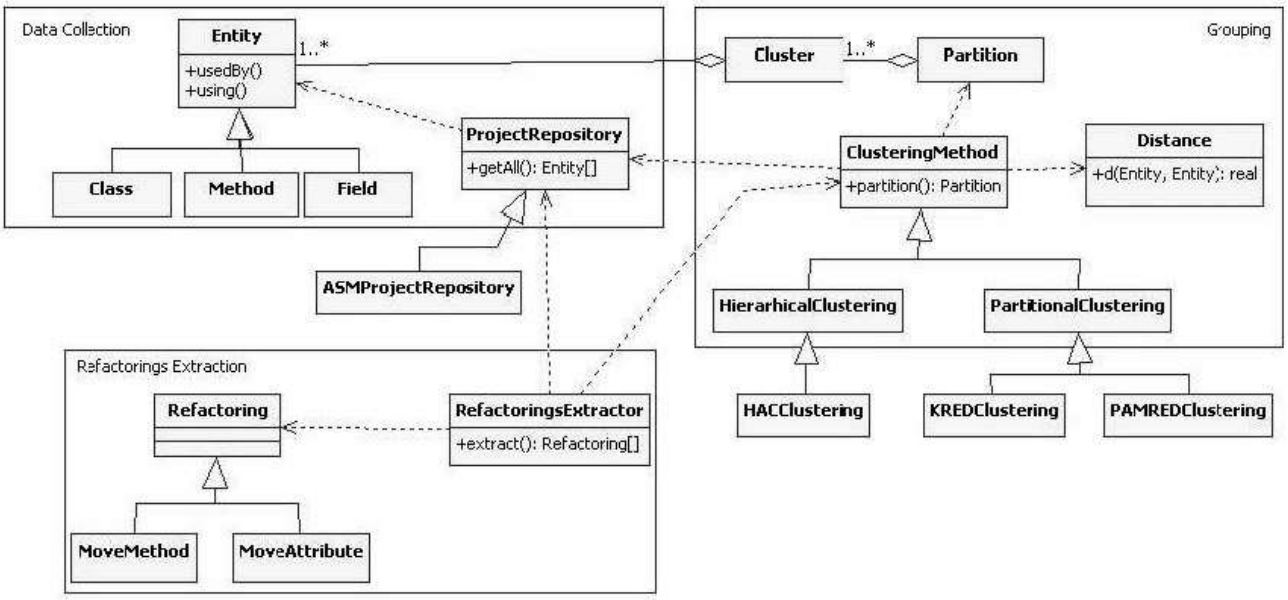
The second case study: a real software system: is DICOM (Digital Imaging and Communications in Medicine) and HL7 (Health Level 7) compliant PACS (Picture Archiving and Communications System) system, facilitating the medical images management, offering quick access to radiological images, and making the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

The algorithm applyed on one of the subsystems from this application: 1015 classes, 8639 methods a¸nd 4457 attributes.

84 refactorings were suggested: 6 Move Attribute, 76 Move Method, and 1 Inline Class

25 accepted, 18 acceptable, 41 rejected.

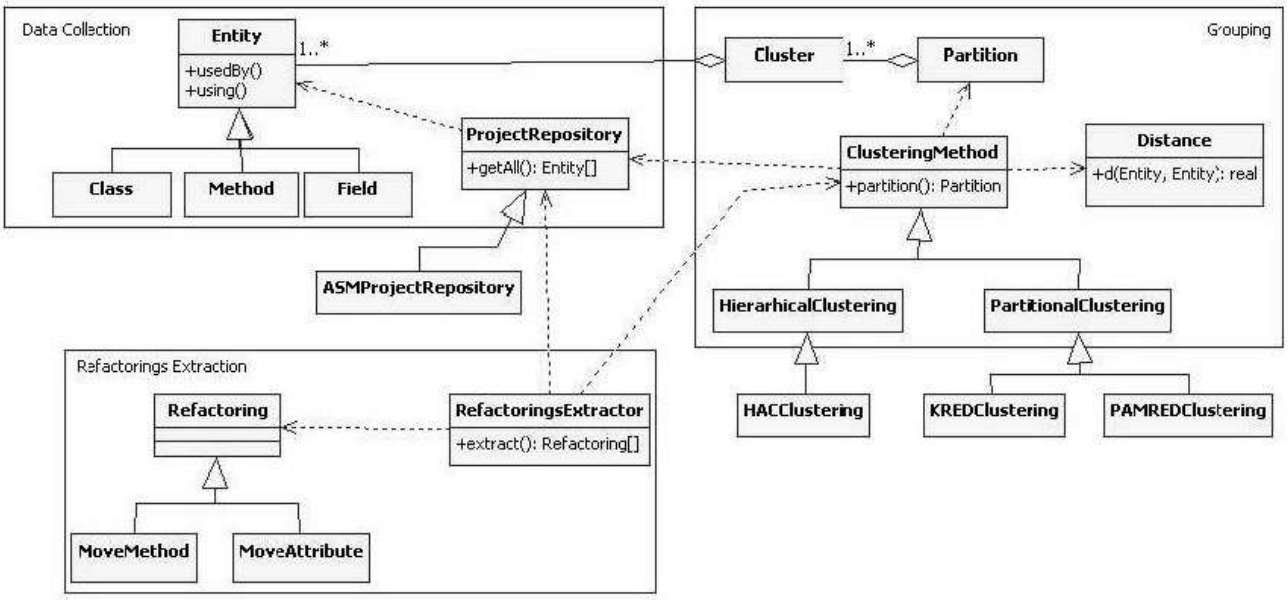
Refactorings identification using clustering- owerview



How to extract information from source code

- Textual search, regular expressions
- Source code parser
- introspection
- instrumentation
- byte code analysis

Refactorings identification using clustering- overview



How to extract information from source code

- Textual search, regular expressions
  - hard to extract semantic information (usage, inheritance, etc)
- Source code parser
  - use parser generators
  - Ex. ANTLR (http://www.antlr.org/) can generate lexers, parsers, tree parsers, and combined lexer-parsers.
    - Parsers can automatically generate abstract syntax trees which can be further processed with tree parsers.
    - ANTLR provides a notation for specifying lexers, parsers, and tree parsers.
    - Grammar exist for: c/c++,java, python, vb, etc (https://github.com/antlr/grammars-v4)
- Introspection
  - obtain information about a class during run-time (methods, fields, parameters,etc)
  - not all languages support this.
  - Usually a limited set of information (ex. Can not go inside a method)
- instrumentation
  - adding extra code (usually automatically, macros/instrumenting byte-code/instrumenting code at the compiler level, etc) to record certain aspects of the (running) code
- byte code analysis
  - specialized frameworks to analyze the byte code
  - useful for languages like java/.net (languages using “Virtual Machines”)
  - ex. ASM framework - Java byte-code manipulation and analysis framework
  - get detailed information (method,instruction level)
  - use the byte code representation (no need for the source code)

Inspecting java byte-code using ASM Framework

- http://asm.ow2.org/
- in many cases reflection is not sufficient we may need more detailed data
- ASM can be used to inspect java byte code
- offer access at the instruction level
- using ASM we can perform static analysis on the application, we can extract dependencies, annotations, etc. using just the byte code (no source code needed)
- the framework use the Visitor design pattern, class byte code is parsed and using custom visitor classes (class visitor, method visitor, annotation visitor,...) information can be gathered

Visitor design pattern

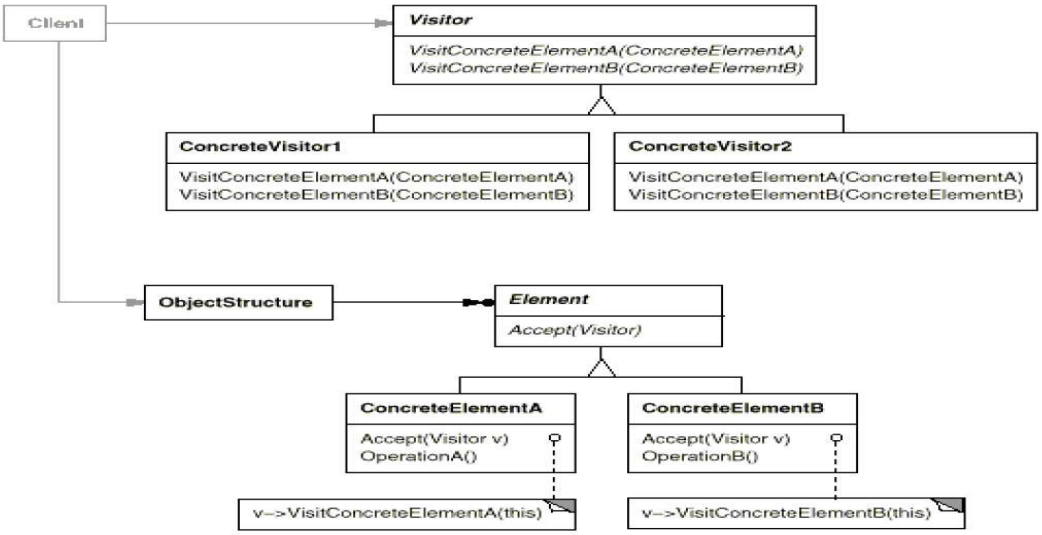
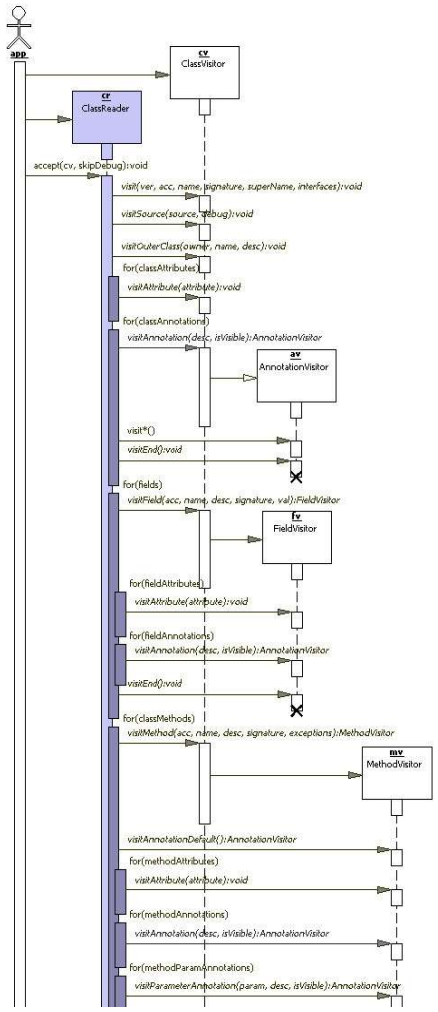


Figure 2 shows the sequence diagram for the common producer-consumer interaction.



ASM sample analyzer

```
public class ClassInfoCollector implements ClassVisitor {
	public static int allClasses = 0;
	public static int nrMethVizited = 0;
	private MClass currentClass;

	public void visit(int version, int access, String name, String signature,
			String superName, String[] interfaces) {

		if (name.contains("$")) {
			currentClass = null;// e un inner class
			return;
		}
		allClasses++;
		currentClass = infC.getCreateMClass(name, superName);
	}

	public FieldVisitor visitField(int access, String name, String desc,
			String signature, Object value) {
		if (currentClass == null) {
			return null;
		}
		currentClass.addCreateField(name, desc, access);
		return null;
	}

	public MethodVisitor visitMethod(int access, String name, String desc,
			String signature, String[] exceptions) {
		nrMethVizited++;
		if (currentClass == null) {
			return null;
		}
		if (name.equals("<init>")) {
			// e constructor implicit
			return null;
		}
		if (name.equals("main") && desc.equals("([Ljava/lang/String;)V")) {
			currentClass = null;// e un main
			return null;
		}
		MMethod meth = currentClass.addCreateMethod(name, desc, access);
		return new MethodInfoCollector(infC, meth);
	}
```

Self organizing maps

A self-organizing map (SOM) - self-organizing feature map (SOFM) or Kohonen map - is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples, called a map.

Self-organizing maps are different from other artificial neural networks in the sense that they use a neighbourhood function to preserve the topological properties of the input space.

The SOM algorithm is based on unsupervised, competitive learning. It provides a topology preserving mapping from the high-dimensional space to map units.
Map units, or neurons, usually form a two-dimensional grid and thus the mapping is a mapping from a high-dimensional space onto a plane.

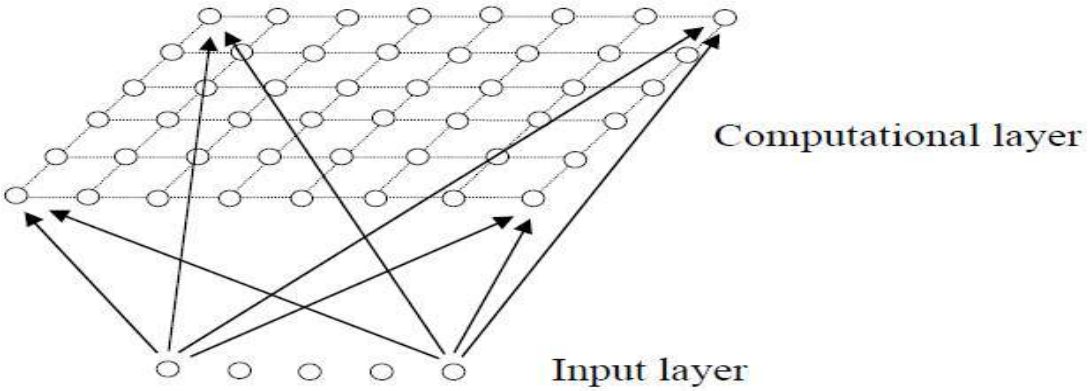
topology preserving - groups similar input data vectors on neurons: points that are near each other in the input space are mapped to nearby map units in the SOM.

The SOM can serve as a clustering tool as well as a tool for visualizing high-dimensional data.

Given a set of input object represented using a vector space model (each object is represented as set of features) after training the SOM will map each input object to a lower dimensional space. The obtained mapping can be use for visualization, clustering or to discover hidden patterns in the input data.

SOM Network architecture

A self-organizing map consists of components called nodes or neurons. Associated with each node are a weight vector of the same dimension as the input data vectors, and a position in the map space



Each neuron on the computational layer is fully connected to all the nodes in the input layer

Computational layer may arranged:

- hexagonal or rectangular grid
- toroidal grid (opposite edges are connected)

Self organizing maps – training

Unsupervised learning - does not need a target output to be specified

usually performed using the Kohonen algorithm.
There are two phases of operation:

- the similarity matching phase
  - the distances between the inputs and the weights are computed. Next, the output node having the minimum distance is chosen and is declared as the “winner” node (best matching unit)
- the weight adaptation phase.
  - the weights from the inputs to the “winner” node are adapted. In addition, a topological neighbourhood of the winning node is defined and the weights connecting the inputs to the nodes contained in this topological neighbourhood are also adapted

Training occurs in several steps and over many iterations:

- Each node's weights are initialized.
- A vector is chosen at random from the set of training data and presented to the lattice.
- Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).
- The radius of the neighbourhood of the BMU is now calculated. This is a value that starts large, typically set to the 'radius' of the lattice, but diminishes each time-step. Any nodes found within this radius are deemed to be inside the BMU's neighbourhood.
- Each neighbouring node's (the nodes found in step 4) weights are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered.
- Repeat step 2 for N iterations.

Self organizing maps – sample implementation

```
	public void train(int nrIteration, SOMTrainData trData,
			double startLearnigRate, double startNeighborRadius,
			SOMTrainingListener l, SOMTrainInputChooser inputChooser) {
		double lambdaTimeConstant = nrIteration / Math.log(startNeighborRadius);

		for (int iteration = 0; iteration < nrIteration; iteration++) {
			// calculez learningRate
			double learningRate = startLearnigRate
					* Math.exp(-(double) iteration / lambdaTimeConstant);
			double neighborRadius = startNeighborRadius
					* Math.exp(-(double) iteration / lambdaTimeConstant);

			// alege un numar(corespunzator unui vector de intrare)
			int[] inputIndexes = inputChooser.getNextInputIndex();

			for (int i = 0; i < inputIndexes.length; i++) {
				int inputIndex = inputIndexes[i];
				trainingStep2(iteration, trData.get(inputIndex), learningRate,
						neighborRadius, l, trData.getLabel(inputIndex));
			}
		}
	}
	public void trainingStep2(int curIter, double input[], double learningRate,
			double neighborRadius, SOMTrainingListener l, Object lbl) {
		// caut Best maching unit
		BMU bmu = computeBestMatchingUnit(input, lbl);
		// actualizez weights de la bmu
		bmu.getNeuron().adjustWeights(input, learningRate, 1);
		// obtin lista vecinilor ce pica in radiusul cerut
		List<NeighborSOMNeuron> neighbors = comptationL.getNeighbors(
				bmu.getNeuron(), neighborRadius);
		// actualizez valorile in functie de cat de apropiat e de BMU
		if (neighbors != null && neighbors.size() > 0) {
			updateNeighbors(input, learningRate, neighborRadius, neighbors);
		}

	}
	public BMU computeBestMatchingUnit(double[] input) {
		int nrNeurons = comptationL.getNrNeurons();
		SOMNeuron bmuN = comptationL.getNeuron(0); // neuronul castigator
		double minDist = dist.distance(input, bmuN.getWeights());
		for (int i = 1; i < nrNeurons; i++) {
			SOMNeuron neuron = comptationL.getNeuron(i);
			double d = dist.distance(input, neuron.getWeights());
			if (d < minDist) {
			// cu cat distanta e mai mica, neuronul e mai aproape de input
				minDist = d;
				bmuN = neuron;
			}
		}
		return new BMU(bmuN, minDist, input);
	}
```

How to use self-organizing maps

Describe input instances as a list of feature values and train the SOM.
We can visualize the SOM:

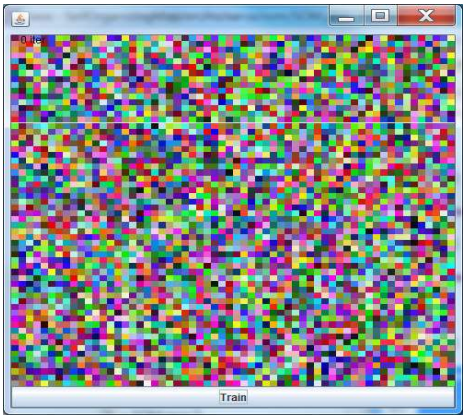
- Visualize the underlining neurons (topology) and outline the BMU (best matching unit) for every instance
- use the UMatrix to get more insight into the learned weights
- represent the neurons in 2D/3D according to their associated weights

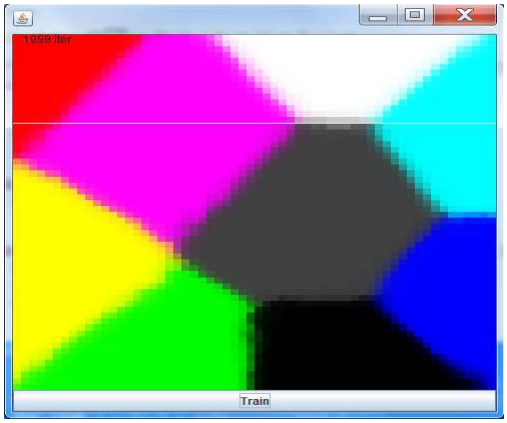
The U-matrix value of a particular node is the average distance between the node and its closest neighbors. In a rectangular grid for instance, we might consider the closest 4 or 8 nodes. The U-Matrix method uses the distances between the units in a SOM as a boundary defining criteria. These distances can be then be displayed as heights giving a U-matrix landscape

Why use SOM:

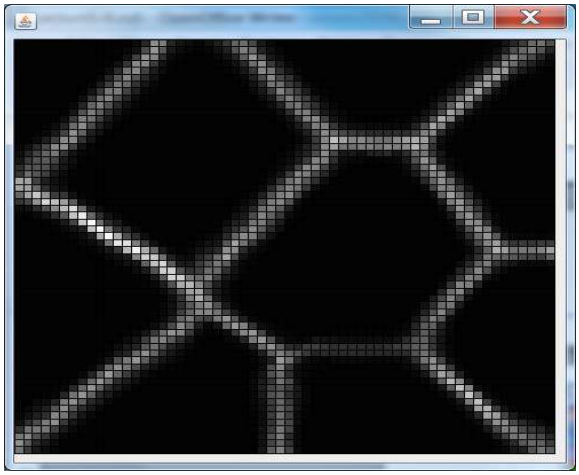
- SOM is able to discover hidden patterns in the data
  - we analyze the final SOM (after training) and similar instances will have there associated BMU close to each other in the network
- we can group (partition) the instances. On the map we can identify group of related instances (there BMU) by visual inspection (maybe using the UMatrix)
  - Interpretation of the U-matrix: altitudes or the high places on the U-matrix will encode data that are dissimilar while the data falling in the same valleys will represent input vectors that are similar.
  - heuristic (find instances in a close range)
  - using clustering algorithm on the neurons
- we can use the SOM as a classifier.
  - If we know the class of every instances used in training we can classify new instances based on the BMU. Compute the BMU for the new instance and select the closest instance from the training data set.

Self organizing maps – sample apps

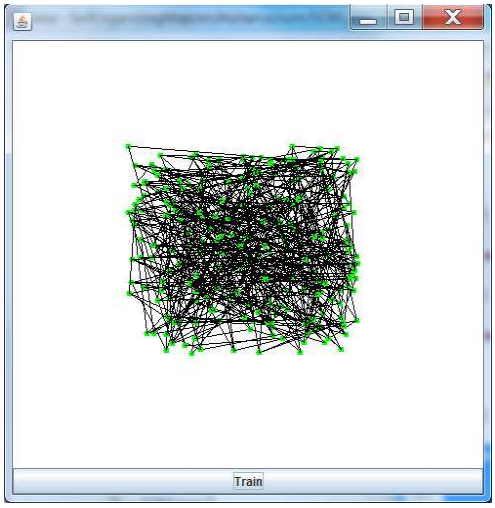


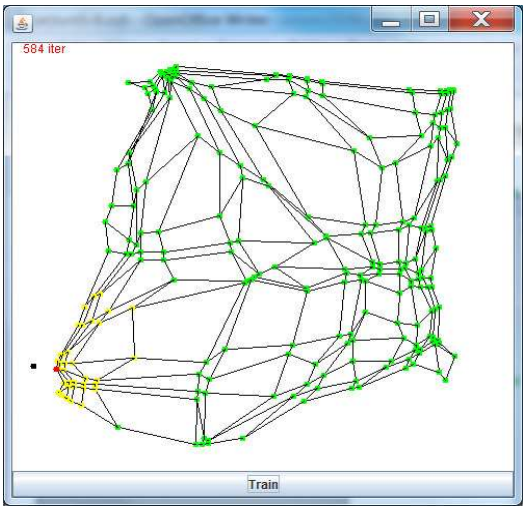


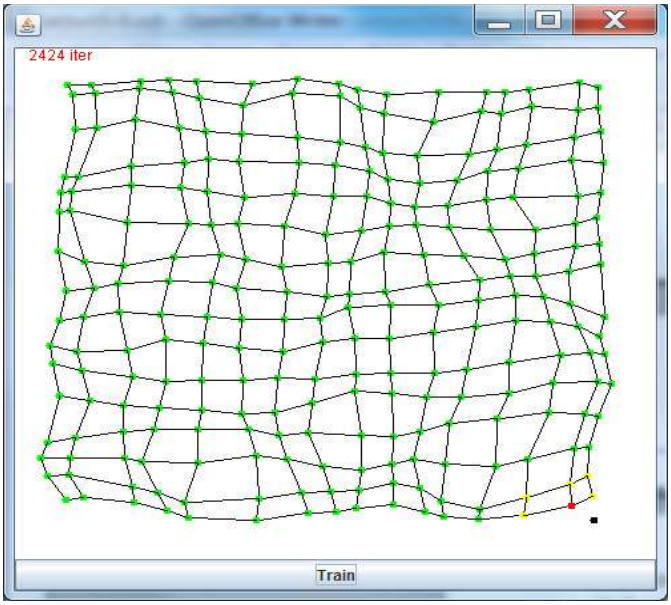
UMatrix



Self organizing maps – sample apps







Software restructuring using self organizing maps

The software system is analyzed and we obtain: list of classes, methods, attributes and relationships between them (inheritance, composition, dependency)
Each entity (class, method or class) s_i (1 <= i <= n) from the software system S is characterized by a l - dimensional vector: (s_i1, s_i2, . . . , s_il), where sij (any j, 1 <= j <= l) is computed as follows:

$$s_{ij} = \begin{cases} \frac{|p(s_i) \cap p(C_j)|}{|p(s_i) \cup p(C_j)|} & if \quad p(s_i) \cap p(C_j) \neq \emptyset \\ 0 & otherwise \end{cases},$$

for a given entity e in S, p(e) is a set of entities from S that are related to e:

- If e in Attr(S) (e is an attribute) then p(e) consists of: the attribute itself, the application class where the attribute is defined, and all methods from Meth(S) that access the attribute.
- If e in Meth(S) (e is a method) then p(e) consists of: the method itself, the application class where the method is defined, all attributes from Attr(S) accessed by the method, all the methods from S used by method e, and all methods from S that overwrite method e.
- If e in Class(S) (e is an application class) then p(e) consists of: the application class itself, all attributes and methods defined in the class, all interfaces implemented by class e and all classes extended by class e.

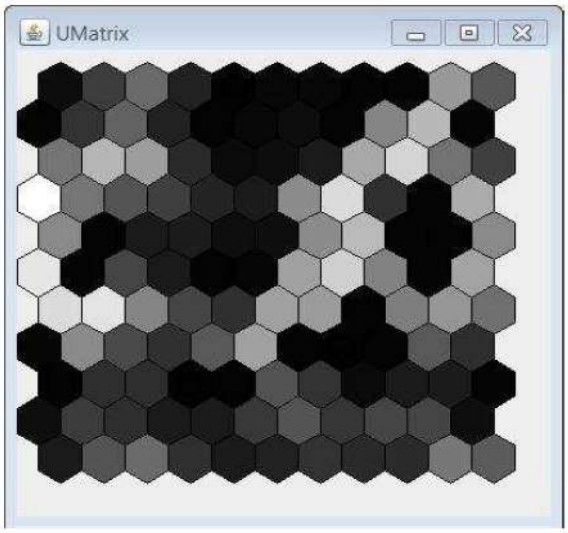
Train the SOM using the above vector space model for the entities from the software system.

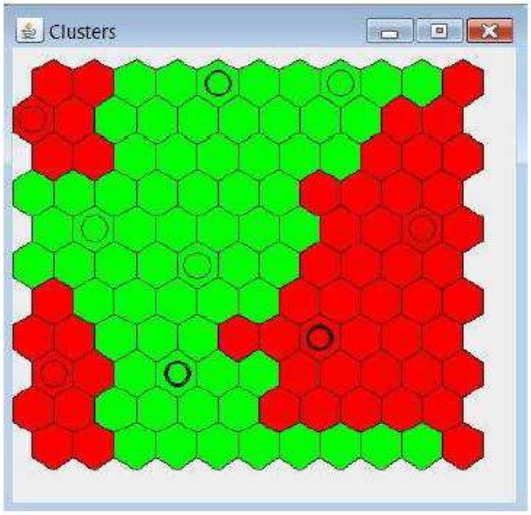
Use the obtained mapping to group entities from the software system.

Software restructuring using self organizing maps – example

```
public class Class_A {

	public static int attributeA2;
	public static int attributeA1;

	public static void methodA1() {
		methodA2();
	}

	public static void methodA2() {
		attributeA2 = 0;
		Class_A.attributeA1 = 12;
	}

	public static void methodA3() {
		attributeA2 = 0;
		methodA1();
		methodA2();
	}
}

public class Class_B {
	private static int attributeB1;
	private static int attributeB2;

	public static void methodB1() {
		attributeB1 = 0;
		Class_A.methodA1();
	}

	public static void methodB2() {
		attributeB1 = 0;
		attributeB2 = 0;
		Class_A.attributeA1 = 12;
	}

	public static void methodB3() {
		attributeB1 = 0;
		methodB1();
		methodB2();

		Class_A.attributeA1 = 12;
	}

	public static void methodB4() {
		attributeB1 = 0;
		methodB2();
		Class_A.attributeA1 = 12;
	}
}
```

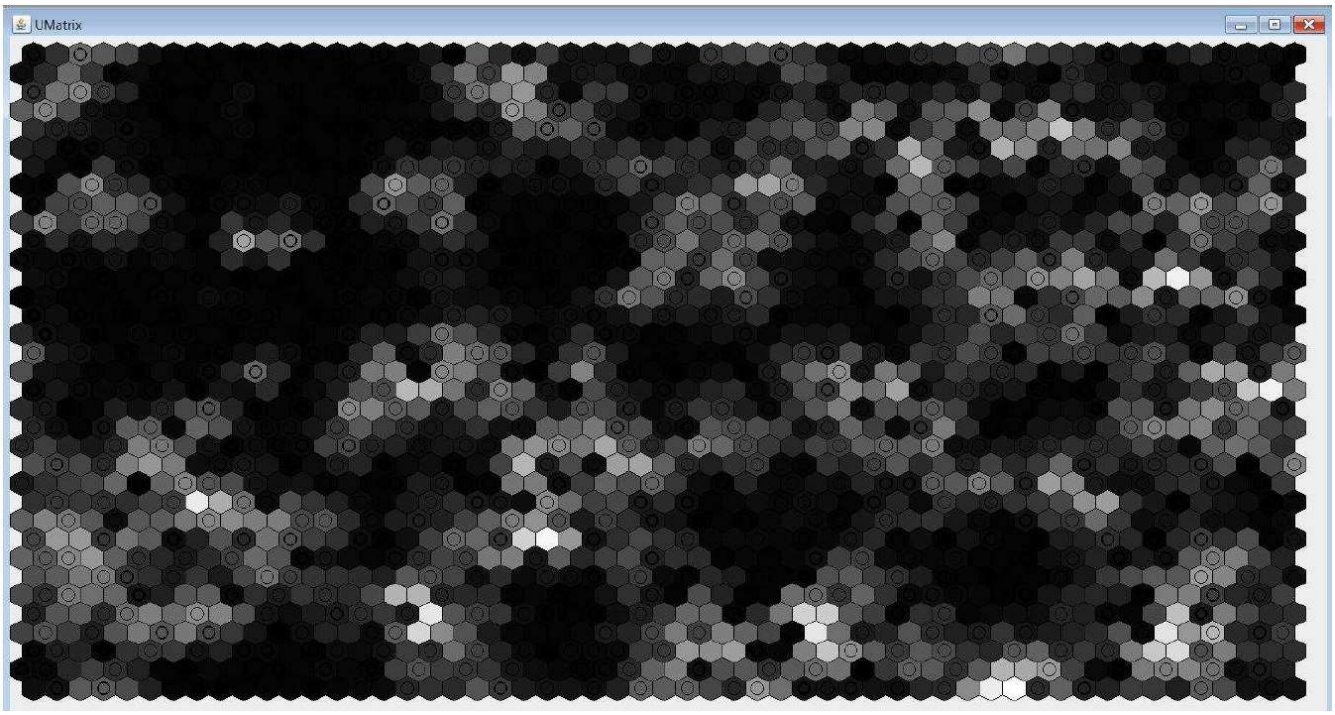
Restructuring using SOM – classA; classB





Software restructuring using self organizing maps – example

UMatrix for JHotDraw



Package restructuring using self-organizing maps – discussion

Given a software system establish a proper package structure for the project

Software Defect Prediction

Software testing is one of the most critical and costly phases in software development.

A **software defect** represents any error or deficiency in a software artifact or a software process and a major focus is on predicting those defects that influence project or product performance.

Software defect prediction is the task of classifying software modules into fault-prone (fp) and non-fault-prone (nfp) ones by means of metric-based classification

Software defect prediction helps in detecting, tracking and resolving software anomalies that might have an effect on human safety and lives, particularly in safety critical systems.

Defect prediction also allows changes to be made earlier in the software life-cycle, assuring this way a lower software cost and improving customer satisfaction

Use of software metrics to predict software faults was initiated by Porter and Selby in 1990. Since then, there has been an extensive interest on metric based fault prediction

Automatic defect predictors based on design and code attributes are an active research area

- over 70% prediction accuracy - defect predictors which employ machine learning/data mining
- 60% detection rate of manual software review (based on a panel in IEEE Metrics 2002)
- an expert reviewer can merely inspect 8 to 20 LOC/minute

Software Defect Prediction – software metrics

Software metrics have become an integral part of the software development process

Most defect prediction approaches are based on software metrics (Code level and Design level metrics)

Example metrics: Fan in, Fan Out, Line count of code, cyclomatic complexity, Depth in Inheritance Tree,...

NASA MDP(Metrics Data Program) repository – Each module is described by a set of code-level and design-level attributes. All discovered faults of the system are also registered in each dataset, together with the number of modules containing the fault

The association between the error data and metrics data in the repository provides the opportunity for users to investigate the relationship of metrics or combinations of metrics to the software

NASA Dataset

Attributes Within the MDP Datasets

	Metric	Comment
1	Loc	McCabe's line count of code
2	v(g)	McCabe "cyclomatic complexity"
3	ev(g)	McCabe "essential complexity"
4	iv(g)	McCabe "design complexity"
5	N	Halstead total operators + operands
6	V	Halstead "volume"
7	L	Halstead "program length"
8	D	Halstead "difficulty"
9	I	Halstead "intelligence"
10	E	Halstead "effort"
11	B	Halstead "error"
12	T	Halstead's time estimator
13	loCode	Halstead's line count
14	loComment	Halstead's count of lines of comments
15	loBlank	Halstead's count of blank lines
16	loCodeAndComments	Count of code and comment lines
17	uniq_Op	unique operators
18	uniq_Opnd	unique operands
19	total_Op	total operators
20	total_Opnd	total operands
21	BranchCount	Count of the flow graph
22	Problems	Module has/has not one or more reported effects

Machine Learning based Software Defect Prediction

Statistical, machine learning, and mixed techniques are widely used in the literature to predict software defects

Problem: given a set of software entities, characterized by a set of software metrics, predict if the software entity is foult-prone (or predict the number of errors)

Extensive interest in metric-based defect prediction, many machine learning techniques are presented in the literature:

- Decision Trees, Nad'ye-Bayes, Logistic Regression, Nearest Neighbor, 1-Rule ,Regression, Neural Networks, Association rules

Most defect prediction techniques used in planning are based on historical data, hence rely on supervised classification:

- use metric data to train the classifier
- apply on a different project

Machine Learning based Software Defect Prediction – Performance

Investigating Classifiers' Performances on NASA Datasets

Dataset		KC1		KC2		CM1		PC1		JM1	
Classifier		ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC
1	BayesNet	69.89	0.791	78.35	0.824	64.65	0.689	74.39	0.703	68.05	0.701
2	NaiveBayes	82.36	0.790	83.52	0.834	85.34	0.658	89.17	0.650	80.42	0.679
3	NaiveBayes Updateable	82.36	0.790	83.52	0.834	85.34	0.658	89.17	0.650	80.42	0.679
4	Logistic	85.68	0.796	82.95	0.808	88.15	0.730	92.42	0.809	81.35	0.713
5	Multilaye Perceptron	85.91	0.771	84.67	0.828	87.55	0.734	93.59	0.723	80.95	0.690
6	SGD	85.20	0.539	84.48	0.663	89.55	0.497	93.05	0.512	80.77	0.504
7	SimpleLogistic	85.72	0.798	84.29	0.838	89.15	0.544	92.60	0.651	81.12	0.711
8	SMO	84.77	0.516	82.75	0.597	89.55	0.497	92.96	0.500	80.72	0.502
9	Voted Perceptron	83.73	0.548	30.26	0.575	90.16	0.500	92.60	0.499	52.21	0.559
10	IBK	84.40	0.735	80.45	0.643	84.73	0.589	90.06	0.740	76.97	0.640
11	Kstar	83.97	0.832	79.11	0.612	87.14	0.644	91.79	0.655	78.56	0.638
12	LWL	84.44	0.765	79.50	0.779	89.75	0.682	93.23	0.713	80.65	.667
13	AdaBoostM1	84.96	0.783	81.41	0.784	90.16	0.700	93.05	0.803	80.79	0.710
14	Attribute Selected Classifier	84.30	0.699	82.56	0.739	89.35	0.542	93.41	0.740	80.86	0.666
15	**Bagging**	**85.44**	**0.809**	**82.95**	**0.823**	**89.75**	**0.720**	**93.50**	**0.915**	**81.42**	**0.742**
16	Classification Via Regression	85.58	0.793	81.80	0.820	89.35	0.752	93.14	0.868	81.24	0.720
17	CV Parameter Selection	84.54	0.496	79.50	0.487	90.16	0.490	93.05	0.486	80.65	0.499
18	Filtered Classifier	84.87	0.752	82.37	0.762	90.16	0.490	93.50	0.589	81.12	0.696
19	Logit Boost	85.39	0.784	83.52	0.824	88.95	0.724	93.14	0.843	80.89	0.713
20	Multi Class Classifier	85.68	0.796	82.95	0.808	88.15	0.730	92.42	0.809	81.35	0.713
21	Multi Scheme	84.54	0.496	79.50	0.487	90.16	0.490	93.05	0.486	80.65	0.499
22	Random Committee	85.49	0.804	81.22	0.786	87.75	0.731	93.59	0.762	81.06	0.723
23	Random SubSpace	85.44	0.789	83.90	0.816	90.16	0.627	93.23	0.835	81.41	0.733
24	Stacking	84.54	0.496	79.50	0.487	90.16	0.490	93.05	0.486	80.65	0.499
25	Vote	84.54	0.496	79.50	0.487	90.16	0.490	93.05	0.486	80.65	0.499

Area under the ROC curve measure (**AUC**)
The classification accuracy (**ACC**) measures the percentage of instances that are classified correctly (or wrongly) by a classifier

Association Rules

method for discovering interesting relations between variables in large databases

Relational association rules

attributes may be in an ordinal relationship, if the domains of the attributes are similar or comparable
In relational association rule mining, the objective is to find several relationships between the attributes that tend to hold over a large percentage of records

Let R = r1; r2; . . . ; rn be a set of instances (entities or records in the relational model), where each instance is characterized by a list of m attributes, a1; . . . ; am
Between two domains (domain of the attributes) we have relations such as less then (<), greather than (>) etc.
relational association rule is an expression: a_{i1} rel_1 a_{i2} rel_2 .... a_{ik-1} relk a_{ik}
where:
a_{i1} , a_{i2} , a_{ik-1} , a_{ik} occur together (non empty) - support
a_{i1} rel_1 a_{i2} rel_2 .... a_{ik-1} relk a_{ik} are true for a sub-set of instances – confidence

Relational association rules are "interesting" if the support and confidence are above a certain (user provided) value.
Association rules has a length - number of relations in the rule. We may be interested in the longest rule for a given support and confidence

Association rules – implementation

Many possible implementations

Start with association rules with length 2

Try to extend the rules

- by adding a new relation and an attribute to an existing rule
- by joining 2 rules (of arbitrary length)

```
public AssociationRuleSet genBinCandidates(double minSup, double minConf)
            throws Exception {
      AssociationRuleSet ars = new AssociationRuleSet();
      int n = es.get(0).getNoAttr();
      for (int i = 0; i < n - 1; i++)
            for (int j = i + 1; j < n; j++) {
                  for (int k = 0; k < RelationSet.numberOfRelations(); k++) {
                        Relation rnew = RelationSet.getRelation(k);
                        AssociationRule anew = new AssociationRule();
                        anew.add(es.get(0).getAttr(i));
                        anew.addRel(rnew, 0);
                        anew.add(es.get(0).getAttr(j));
                        if (anew.valid()) {
                         if (anew.computeIfInteresting(minSup, minConf, es)) {
                                    ars.addRuleLast(anew);
                         }
                        }
                  }
            }
      return ars;
}
```

Association rules – implementation

```
private void updateSupportConfidence(Entity ec) {
		boolean inRelation = true;
		Attribute ai = ec.searchAttr(getAttribute(0));
		if (ai.isNull()) {
			return; // not in support
		}
		for (int i = 0; i < length(); i++) {
			Attribute aiPlus1 = ec.searchAttr(getAttribute(i + 1));
			if (aiPlus1.isNull()) {
				return; // not in support
			}
			if (inRelation
				&& !(getRelation(i).areInRelationaNonNull(ai, aiPlus1))) {
			// is clear that is not in confidence but we need to continue
			// to check if is in support
				inRelation = false;
			}
			ai = aiPlus1;
		}
		// is in support, all the values are different from null
		support++;
		if (inRelation) {
			// is a valid relation (all relations are satisfied
			confidence++;
		}
	}
public boolean computeIfInteresting(double minSup, double minConf,
			EntitySet es) {
		if (rels.isEmpty()) {
			return false;
		}
		confidence = 0;support = 0;
		int nrEntities = es.size();
		int requiredConf = (int) Math.round(minConf * nrEntities);
		int requiredSupport = (int) Math.round(minSup * nrEntities);
		int remainedEntites = nrEntities;
		// calculeaza suportul in setul de entitati ale regulilor din setul cs
		for (int j = 0; j < nrEntities; j++) {
			Entity ec = es.get(j);
			// verific regula pentru entitatea curenta
			updateSupportConfidence(ec);
			remainedEntites--;
			if (support + remainedEntites < requiredSupport) {
				return false;// can not reach the requested support
			}
			if (confidence + remainedEntites < requiredConf) {
				return false;// can not reach the requested confidence
			}
			if (confidence >= requiredConf && support >= requiredSupport) {
			return true;// we reached the required support and confidence
			}
		}

		return false;
	}
```

Association Rules – example

```java
public class Class_A {

	public static int attributeA2;
	public static int attributeA1;

	public static void methodA1() {
		methodA2();
	}

	public static void methodA2() {
		attributeA2 = 0;
		Class_A.attributeA1 = 12;
	}

	public static void methodA3() {
		attributeA2 = 0;
		methodA1();
		methodA2();
	}
}
```

```java
public class Class_B {
	private static int attributeB1;
	private static int attributeB2;
	public static void methodB1() {
		attributeB1 = 0;
		Class_A.methodA1();
	}
	public static void methodB2() {
		attributeB1 = 0;
		attributeB2 = 0;
		Class_A.attributeA1 = 12;
	}
	public static void methodB3() {
		attributeB1 = 0;
		methodB1();
		methodB2();

		Class_A.attributeA1 = 12;
	}
	public static void methodB4() {
		attributeB1 = 0;
		methodB2();
		Class_A.attributeA1 = 12;
	}
```

Dataset

Entity	*DIT*	*NOC*	*FI*	*FO*
Class_A	1	0	3	1
Class_B	1	0	0	2
mA1	1	0	2	1
mA2	1	0	2	0
mA3	1	0	0	2
mB1	1	0	1	1
mB2	1	0	1	0
mB3	1	0	0	2

Dit – Depth in inheritance; NOC – Number of children; Fi – Fan In; Fo – Fan Out

Association rules

Interesting relational association rules.

Length	Rule	Confidence
2	DIT > NOC	1
2	NOC < FI	0.625
2	NOC < FO	0.75
2	FI > FO	0.5
3	DIT > NOC < FI	0.625
3	DIT > NOC < FO	0.75
3	NOC < FI > FO	0.5
4	DIT > NOC < FI > FO	0.5

Maximal interesting relational association rules.

Length	Rule	Confidence
3	DIT > NOC < FO	0.75
4	DIT > NOC < FI > FO	0.5

Association rule based Software Defect Prediction

represent the entities (classes, modules, methods, functions) of a software system as a multidimensional vector, whose elements are the values of different software metrics applied to the given entity

we consider a set of software metrics (the feature set in a vector space model based approach) relevant for deciding if a software entity is or not defective

supervised learning scenario for predicting defective software entities, two sets containing positive and negative instances are given. By "+" we denote the class corresponding to software entities having defects - positive instances or defects.

We detect in the training datasets all the interesting relational rules, with respect to the user-provided support and confidence thresholds.

After the training was completed, when a new instance (software entity) has to be classified (as "+" or ""), we reason as follows.

Considering the rules discovered during training in the set of positive and negative instances, two scores, score (indicating the similarity degree of the instance to the positive class) and score (indicating the similarity degree of the instance to the negative class), are computed.
If score is greater than score, then the query instance will be classified as a positive instance, otherwise it will be classified as a negative instance.

For classifying a a software entity as being or not defective, the following steps will be performed:

1. Data pre-processing.
2. Training/building the classifier.
3. Testing/classification.

Data pre-processing

During this step, the training data are scaled to [0,1] and a statistical analysis is carried out on the training datasets Ds- and DS in order to find a subset of features that are correlated with the target output.

The statistical analysis on the features is performed in order to reduce the dimensionality of the input data, by eliminating features which do not significantly influence the output value.

A Spearman correlation of 0 between two variables X and Y indicates that there is no tendency for Y to either increase or decrease when X increases. A Spearman correlation of 1 or 1 results when the two variables being compared are monotonically related, even if their relationship is not linear.

At the statistical analysis step we remove from the feature set those features that have no significant influence on the target output, i.e. are slightly correlated with it.

Example: correlation for attributes in the NASA Dataset – CM1

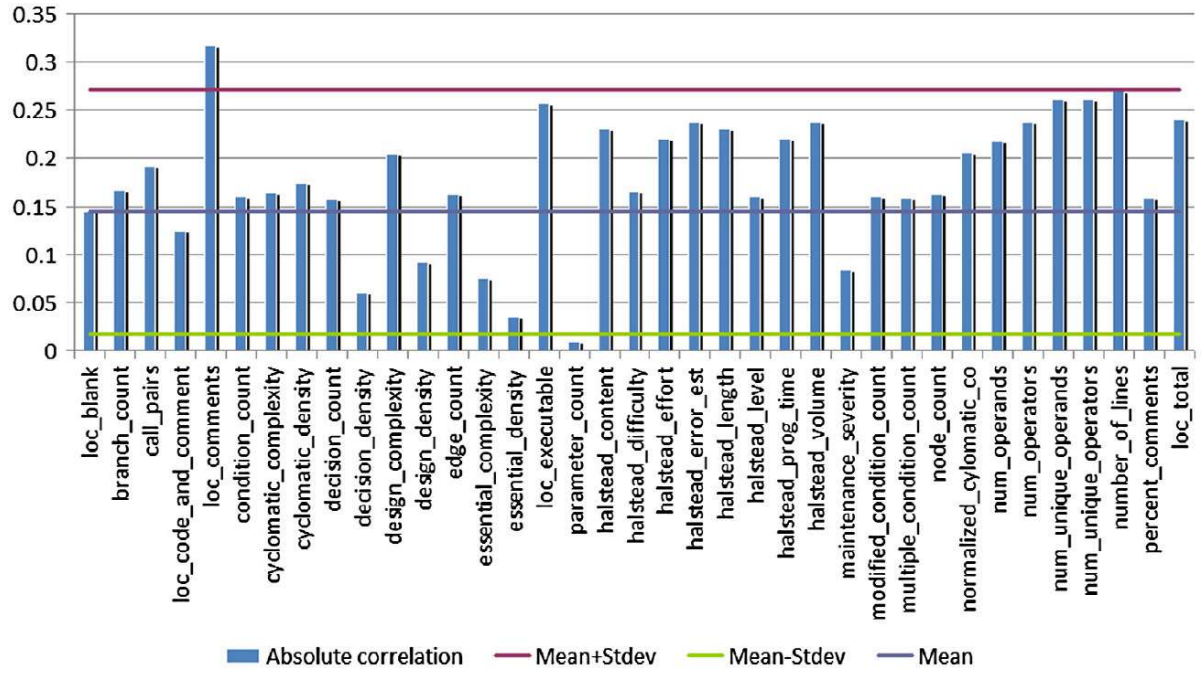


Fig. 2. Correlations for the **CM1** dataset.

Training

Determine from DS+, using the DRAR algorithm, the set RAR+ of relational association rules having a minimum support and confidence.

Determine from DS- using the DRAR algorithm, the set RAR- of relational association rules having a minimum support and confidence

For each rule r from the sets RAR+ and RAR- determined as indicated above, the support and the confidence
of the rule are computed. We denote by ratio the value obtained by dividing the confidence of the rule to its support

Classification

A new software entity e has to be classified, we calculate the scores score+ (the similarity of e to the positive class) and score- (the similarity of e to the negative class).

The score computation method takes into consideration the strength of the verified and unverified rules (by using the value of ratio, which increases as the confidence of the rule increases)

there are other possibilities for score computation as well: using only the number of these rules, or computing one single score, which can be transformed into a class label with the use of a threshold.

At the classification stage of a new instance e if score+ > score- then instance e will be classified as a positive instance (defect), otherwise it will classified as a negative instance (non-defect).

Results

Comparative results.

Data	Acc		Pd		Spec		Prec		AUC	
CM1	CBA2	0.8036	CBA2	0.2	CBA	0.885	CBA2	n/a	CBA2	0.598
	1R	0.8816	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	**0.8995**	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	0.720
	EDER-SD	0.88	EDER-SD	n/a	**EDER-SD**	**0.947**	EDER-SD	0.357	EDER-SD	n/a
	DPRAR	0.8716	**DPRAR**	**0.929**	DPRAR	0.8632	**DPRAR**	**0.5**	**DPRAR**	**0.896**
KC1	CBA2	0.8371	CBA2	0.461	CBA2	0.910	CBA2	n/a	**CBA2**	**0.836**
	1R	0.831	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	0.8568	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	0.809
	EDER-SD	**0.859**	EDER-SD	n/a	**EDER-SD**	**0.980**	EDER-SD	0.596	EDER-SD	n/a
	DPRAR	0.823	**DPRAR**	**0.818**	DPRAR	0.825	**DPRAR**	**0.628**	DPRAR	0.822
KC3	CBA2	0.9091	CBA2	0.333	CBA2	0.962	CBA2	n/a	CBA2	0.696
	1R	n/a	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a
	EDER-SD	**0.935**	EDER-SD	n/a	**EDER-SD**	**1**	**EDER-SD**	**0.643**	EDER-SD	n/a
	DPRAR	0.83	**DPRAR**	**0.889**	DPRAR	0.8165	DPRAR	0.5246	**DPRAR**	**0.85225**
PC1	CBA2	0.9178	CBA2	0.44	CBA2	1	CBA2	n/a	CBA2	0.827
	1R	0.9369	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	0.9332	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	0.915
	EDER-SD	0.943	EDER-SD	n/a	**EDER-SD**	**1**	EDER-SD	0.577	EDER-SD	n/a
	DPRAR	**0.956**	**DPRAR**	**0.885**	DPRAR	0.963	**DPRAR**	**0.692**	**DPRAR**	**0.924**
JM1	CBA2	0.7352	CBA2	0.461	CBA2	n/a	CBA2	n/a	CBA2	0.688
	1R	0.7987	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	0.8104	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	0.742
	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a
	DPRAR	**0.96**	**DPRAR**	**0.842**	**DPRAR**	**0.992**	**DPRAR**	**0.967**	**DPRAR**	**0.917**
MC2	CBA2	0.6981	CBA2	0.333	CBA2	0.886	CBA2	n/a	CBA2	0.671
	1R	n/a	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a
	EDER-SD	0.759	EDER-SD	n/a	**EDER-SD**	**1**	EDER-SD	0.529	EDER-SD	n/a
	DPRAR	**0.896**	**DPRAR**	**0.773**	DPRAR	0.9632	**DPRAR**	**0.919**	**DPRAR**	**0.868**
MW1	CBA2	0.9104	CBA2	0.5	CBA2	0.919	CBA2	n/a	CBA2	0.86
	1R	n/a	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a
	EDER-SD	**0.941**	EDER-SD	n/a	**EDER-SD**	**1**	EDER-SD	0.456	EDER-SD	n/a
	DPRAR	**0.941**	**DPRAR**	**0.889**	DPRAR	0.947	**DPRAR**	**0.667**	**DPRAR**	**0.918**
PC2	**CBA2**	**0.992**	CBA2	0.455	**CBA2**	**0.994**	CBA2	n/a	CBA2	0.809
	1R	n/a	1R	n/a	1R	n/a	1R	n/a	1R	n/a
	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a	Bagging	n/a
	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a
	DPRAR	0.984	**DPRAR**	**0.938**	DPRAR	0.985	**DPRAR**	**0.577**	**DPRAR**	**0.962**
PC3	CBA2	0.8648	CBA2	0.255	CBA2	0.934	CBA2	n/a	CBA2	0.821
	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a
	DPRAR	**0.967**	**DPRAR**	**0.85**	**DPRAR**	**0.983**	**DPRAR**	**0.877**	**DPRAR**	**0.917**
PC4	CBA2	0.8396	CBA2	0.648	CBA2	0.866	CBA2	n/a	CBA2	0.885
	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a	EDER-SD	n/a
	DPRAR	**0.961**	**DPRAR**	**0.814**	**DPRAR**	**0.985**	**DPRAR**	**0.894**	**DPRAR**	**0.899**

Software testing

Testing is observing the behavior of a program in many executions.
Execute the program for some input data and observe if the results are correct for these inputs.

Testing does not prove program correctness (only give us some confidence). On the contrary, it may prove its incorrectness if one execution give wrong results.

Testing can never completely identify all the defects within software.

Testing methods

Exhaustive testing

Check the program for all possible inputs.
Impractical so we need to choose a finite number of test cases.

Black box testing

The selection of input data for testing is decided by analyzing the specification. Distinct cases of the problem are decided and we use a test input data for each case

White box testing

Select the test data by analyzing the text of the program. We select test data such that all the execution paths are covered.
We test a function such that each statement is executed.

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level.
Testing unit of code in isolation (functions). Test small parts of the program independently

Integration testing

Considers the way program works as a whole. After all modules have been tested and corrected we need to verify the overall behavior of the program.

Software testing

Automated testing

Test automation is the process of writing a computer program to do testing that would otherwise need to be done manually.
use of software to control the execution of test, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions

A **test case** has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.
A **test suite** is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors

Code coverage

Is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage

Test coverage is a useful tool for finding untested parts of a code-base

Coverage criteria:

- Function coverage- Has each function (or subroutine) in the program been called?
- Statement coverage- Has each statement in the program been executed?
- Branch coverage - Has each branch of each control structure (such as in if and case statements) been executed?
- Condition coverage (or predicate coverage) - Has each Boolean sub-expression evaluated both to true and false?

White box vs Black Box testing

```
def isPrime(nr):
    """
      Verify if a number is prime
      return True if nr is prime False if not
      raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True
```

Black Box

- test case for a prime/not prime
- test case for 0
- test case for negative number

White Box (cover all the paths)

- test case for 0
- test case for negative
- test case for 1
- test case 3
- test case for prime (no divider)
- test case for not prime

```
def blackBoxPrimeTest():
    assert (isPrime(5)==True)
    assert (isPrime(9)==False)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

```
def whiteBoxPrimeTest():
    assert (isPrime(1)==False)
    assert (isPrime(3)==True)
    assert (isPrime(11)==True)
    assert (isPrime(9)==True)
    try:
        isPrime(-2)
        assert False
    except ValueError:
        assert True
    try:
        isPrime(0)
        assert False
    except ValueError:
        assert True
```

Role of machine learning in software testing

Various types of data can be collected while testing software:

- Execution traces and coverage information for test cases can be captured at different levels of detail.
- Failure data that captures where and why a failure occurs is also of interest

Testing problems:

- completeness of test suites,
- the automation of test oracles,
- the distribution of testing resources according to levels of risks (risk-driven testing),
- localization of faults causing failures

Recurrent issues in applying machine learning techniques to software testing:

- whether, for a given application, we can possibly obtain adequate data, both in terms of form and content, to learn interesting, actionable patterns.
- patterns that are an inaccurate approximation of reality or cannot be easily used to drive beneficial actions are of no practical interest.
- the benefits must be significantly higher than the cost of additional data collection and analysis. The complexity of patterns to be learned may be such that the data needs complex preprocessing before a machine learning algorithm becomes capable of learning useful patterns.

Examples of machine learning for software testing

Test specification and test suite refinement: understand the limitations of test suites and their possible redundancies

MELBA (MachinE Learning based refinement of Black-box test specification) methodology requires as input both a test suite and a test specification in the form of Category-Partition (CP) categories and choices

- Categories are properties of parameters that can have an influence on the behavior of the software under test (e.g., size of an array in a sorting algorithm).
- Choices (e.g., whether an array is empty) are the potential values of a category
- abstract test cases are generated : shows an output equivalence class and pairs (category, choice) that characterize its inputs and environment parameters (instead of raw inputs).
- MELBA can identify a number of potential problems (by analyzing decision tree):
  - Case 1—Instances (test cases) can be misclassified: the wrong output equivalence class is associated to a test case.
  - Case 2—Certain categories or choices are not used in the tree
  - Case 3—Certain combinations of choices, across categories, are not present on any path, from the root node to any leaf of the tree.
  - Case 4—A leaf of a tree contains a large number of instances (test cases).

Debugging / Fault localization: to identify suspicious statements, which are likely to contain a fault related to a failure observed during testing. This can be used to help fault localization during debugging.

use of decision trees to learn various failure conditions based on information regarding the test cases' inputs and outputs. Failing test cases executing under similar conditions are then assumed to fail due to the same fault(s). Statements are then considered suspicious if they are covered by a large proportion of failing test cases that execute under similar conditions.

Uses categories and choices. Ex.Triangle: input values characterize the length of triangle sides. We can use CP to define categories on the relationships among the lengths of the triangle sides, e.g., whether they are equal or
otherwise

Then the test cases can be expressed as tuples in terms of these properties and we refer to these tuples as abstract test cases, e.g., input data `(1, 1, 2)` becomes `(side1=side2, side2<side3,side3>side1)`

Examples of machine learning for software testing

Risk-driven testing : prioritize testing efforts.

Usually this is done by analyzing the “risk” associated with a functionality or system components, depending on the testing level. Risk is usually defined as a combination of probability of failures and the damages they can potentially cause

focused on the construction of models predicting the location of faults across files or classes.

Test oracles: A test oracle might specify correct output for all possible input or only for specific input. It might not specify actual output values but only constraints on them.

Using machine learning we can learn for a given function/module the relations between inputs and outputs.
This can be used as a test oracle for further testing as usually modules/functions are changing during the lifetime of a project.

Especially useful in the context of iterative development and testing, when algorithms are constantly refined and re-tested

Examples of machine learning for software testing

Test Data Generation: automatic generation of test data (input / expected output)

Test data generation can have different objectives:

- the coverage of specif program structures, as part of a structural, or white-box testing strategy;
- the exercising of some specif program feature, as described by a specification;
- attempting to automatically disprove certain gray-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- to verify non-functional properties, for example the worst-case execution time of a segment of code.

Machine learning used in software test data generation:Hill Climbing, Simulated Annealing, Evolutionary Algorithms

Hill Climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighborhood of this solution is investigated. If a better solution is found, then this replaces the current solution. Hill climbing is simple and gives fast results. However it is easy for the search to yield sub-optimal results when the hill climbed leads to a solution that is locally optimal, but not globally optimal.

Simulated Annealing is similar in principle to Hill Climbing. However, by probabilistically accepting poorer solutions, Simulated Annealing allows for less restricted movement around the search space.

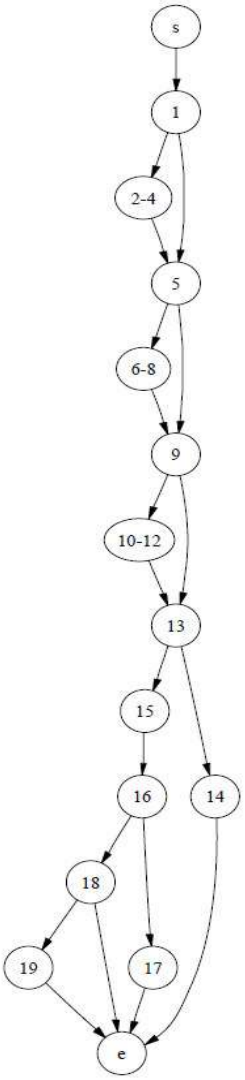
Evolutionary Algorithms use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection. **Genetic Algorithms** are probably the most well known form of Evolutionary Algorithm. For Genetic Algorithms, the search is primarily driven by the use of **recombination** - a mechanism of exchange of information between solutions to "breed" new ones - whereas Evolution Strategies principally use **mutation** - a process of randomly modifying solutions

The name "Genetic Algorithm" comes from the analogy between the encoding of candidate solutions as a sequence of simple components, and the genetic structure of a chromosome. Continuing with this analogy, solutions are often referred to as individuals or chromosomes. The components of the solution are sometimes referred to as genes, with the possible values for each component called alleles, and their position in the sequence the locus.

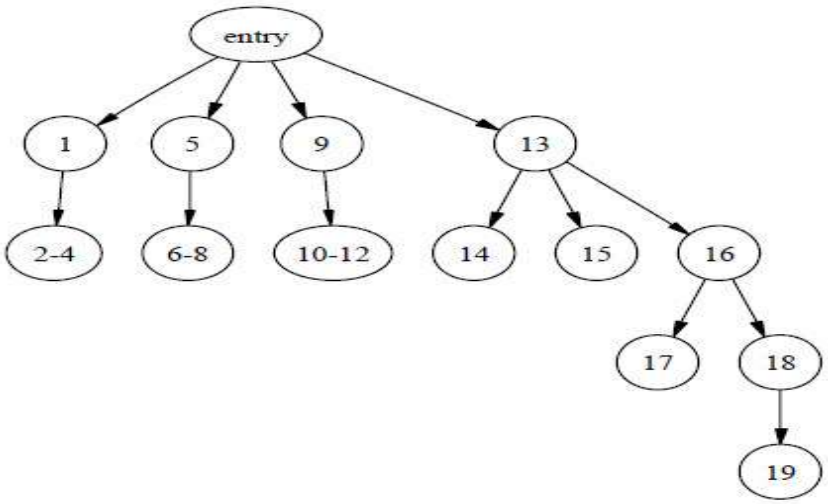
Test Data generation – White box testing – static approaches

Many approaches use the control flow graph(CFG) of a program.

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`int type;`
1	`if (a > b)`
2-4	`{ int t = a; a = b; b = t; }`
5	`if (a > c)`
6-8	`{ int t = a; a = c; c = t; }`
9	`if (b > c)`
10-12	`{ int t = b; b = c; c = t; }`
13	`if (a + b <= c)`
	`{`
14	`type = NOT_A_TRIANGLE;`
	`}`
	`else`
	`{`
15	`type = SCALENE;`
16	`if (a == b && b == c)`
	`{`
17	`type = EQUILATERAL;`
	`}`
18	`else if (a == b || b == c)`
	`{`
19	`type = ISOSCELES;`
	`}`
	`}`
e	`return type;`
	`}`



Control dependency graph



Test Data generation – White box testing – static approaches

Symbolic Execution

Symbolic Execution is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure.

The technique can be used to derive a constraint system in terms of the input variables which describes the conditions necessary for the traversal of a given path.

Solutions to the constraint system are input data which will execute the path. Constraint satisfaction problems are in general NP-complete. Heuristic methods can be used in to attempt the nding of a solution

Domain Reduction

Symbolic execution is used to develop the constraints in terms of the input variables. Domain reduction is then used to attempt a solution to the constraints.

Dynamic Structural Test Data Generation

Dynamic methods execute the program in question with some input, and then simply observe the results via some form of program instrumentation

Goal is to generate test data to cover all the possible execution paths (white box testing)

Dynamic method overcome some limitation of the static approaches:

- presence of pointer arithmetic (*p = *q)
- arrays (ex a[i]!=a[j])

Random Testing: generate random input observe the output and the path covered

Applying Local Search: The tester selects a path through the program, and then produces a straight-line version of it, containing only that path. The inputs are modified to obtain other paths (alter the input based on the constraint predicates)

Applying Simulated Annealing:a neighborhood structure has to be defined for the various different input variable types. For integer and real variables, the neighborhood is simply a defined range of values around each individual value. Since the ordering of values is not significant for boolean and enumerated types, all values for these variables are considered as neighbors.
The objective function is simply the branch distance of the required branch when control ow diverges away from the intended path, or away from the target structure down a critical branch

Applying Evolutionary algorithms: often referred to in the literature as Evolutionary Testing

- Coverage-Oriented Approaches: reward individuals on the basis of covered program structures.
- Structure-Oriented Approaches: A separate search is undertaken for each uncovered structure required by the coverage criterion

Artificial neural network (ANN)

The feed-forward ANN can be used for *classification* or *prediction* scenarios

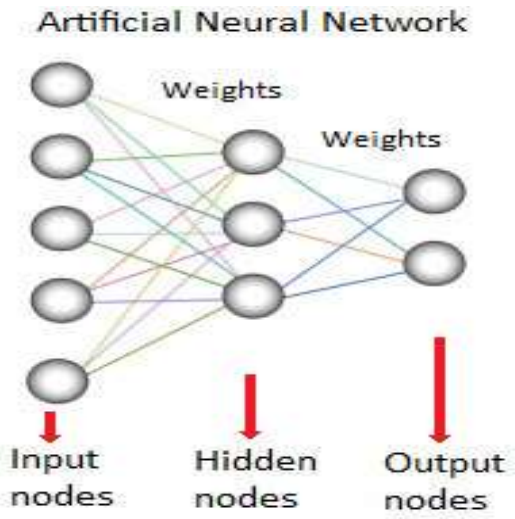
The network is trained to classify certain patterns into certain groups, and then is used to classify novel patterns which were never presented to the net before.

Artificial neural networks are generally presented as systems of interconnected "neurons" which can compute values from inputs, and are capable of machine learning as well as pattern recognition thanks to their adaptive nature.

An artificial neuron is a computational model inspired in the natural neurons. Natural neurons receive signals through synapses located on the dendrites or membrane of the neuron. When the signals received are strong enough (surpass a certain threshold), the neuron is activated and emits a signal though the axon. This signal might be sent to another synapse, and might activate other neurons.

These basically consist of inputs (like synapses), which are multiplied by weights (strength of the respective signals), and then computed by a mathematical function which determines the activation of the neuron.

Artificial neural network (ANN)



Artificial neurons are **organized in layers**, and send their signals “forward”, and then the errors are propagated backwards.

The network receives inputs by neurons in the **input layer**, and the output of the network is given by the neurons on an **output layer**. There may be one or more intermediate **hidden layers**.

The back-propagation **algorithm** uses **supervised learning**, which means that we provide the algorithm with examples of the inputs and outputs we want the network to compute, and then the error (difference between actual and expected results) is calculated.

The idea of the back-propagation algorithm is to reduce this error, until the ANN learns the training data. The training begins with random weights, and the goal is to adjust them so that the error will be minimal.

Artificial neural network (ANN) – sample implementation

Implementation propagate forward

```
	public PropagationResult forwardPropagation(float[] inValues) {
		if (inValues.length != nrInputNeurons) {
		   throw new IllegalArgumentException("invalid number of input values");
		}
		PropagationResult result = new PropagationResult();
		// calculam iesirile din stratul hidden
		float[] hiddenOuts = propagateForwardOnLayer(inValues, weigthInHi);
		result.setHiddenResults(hiddenOuts);

		// calculam iesirile din stratul out
		float outputs[] = propagateForwardOnLayer(hiddenOuts, weigthHiOut);
		result.setOutputs(outputs);
		return result;
	}

	private float[] propagateForwardOnLayer(float[] inValues, float[][] weigths) {
		float[] out = new float[weigths[0].length];
		for (int i = 0; i < out.length; i++) {
			// initializam cu bias0ul
			float sum = weigths[weigths.length - 1][i];
			for (int j = 0; j < weigths.length - 1; j++) {
				sum = sum + inValues[j] * weigths[j][i];
			}
			out[i] = activationF.f(sum);
		}
		return out;
	}
```

Artificial neural network (ANN) – sample implementation - training

```
public float train(float[] input, float[] expectedOut) {
		PropagationResult result = netw.forwardPropagation(input);
		backPropagation(input, result, expectedOut);
		return computeError(result.getOutputs(), expectedOut);
	}

	private float computeError(float[] networkOut, float[] expectedOut) {
		float error = 0;
		for (int i = 0; i < netw.getNumberOfOutputNeurons(); i++) {
			error += (networkOut[i] - expectedOut[i])
					* (networkOut[i] - expectedOut[i]);
		}
		return error;
	}
	private void backPropagation(float[] input, PropagationResult result,
			float[] expectedOut) {
		float[] networkOut = result.getOutputs();

		ActivationFunction activationF = netw.getActivationF();
		// calculam eroarea pe stratul de iesire
		float[] outputDelta = new float[networkOut.length];
		for (int i = 0; i < networkOut.length; i++) {
			outputDelta[i] = (expectedOut[i] - networkOut[i])
					* activationF.fDeriv(networkOut[i]);
		}
		// calculam eroarea pe stratul ascuns
		float[] networkHiddenOut = result.getHiddenResults();

		float[] hiddenDelta = new float[networkHiddenOut.length];
		for (int i = 0; i < netw.getNumberOfHiddenNeurons(); i++) {
			float sum = 0;
			for (int j = 0; j < networkOut.length; j++) {
				sum += netw.getWeigthHiddenOut(i, j) * outputDelta[j];
			}
			hiddenDelta[i] = activationF.fDeriv(networkHiddenOut[i]) * sum;
		}
		// actualizam ponderile hidenOutput
		for (int i = 0; i < networkOut.length; i++) {
			for (int j = 0; j < netw.getNumberOfHiddenNeurons(); j++) {
			   momentumHiOut[j][i] = learningRate * outputDelta[i]
					* networkHiddenOut[j] + alfa * momentumHiOut[j][i];
			float aux = netw.getWeigthHiddenOut(j, i) + momentumHiOut[j][i];
				netw.setWeigthHiddenOut(j, i, aux);
			}
			// actualizez si pentru bias
			momentumBiasHiOut[i] = learningRate * outputDelta[i] + alfa
					* momentumBiasHiOut[i];
			float aux = netw.getBiasInHidden(i) + momentumBiasHiOut[i];
			netw.setBiasInHidden(i, aux);
		}
```

Use of Artificial neural network (ANN) in software testing

ANN Test oracle

Artificial neural networks can be used to compute a function using example inputs and outputs

For an existing function one can use ANN to "learn" a function(method) and use the trained ANN as a test oracle.

In an iterative development scenario this can be useful to test new versions of the method

Triangle function example: given the length of each side the function decides if the triangle is isosceles, scalene, equilateral, or invalid

triangleType(int a, int b, int c)

ANN structure:

Input Layer: 3 input neurons (length side 1, length side 2,length side 3)

Output Layer: 4 output neurons (isosceles, scalene, equilateral, or invalid)

Hidden Layer (Optional)

Training data generation

We invoke the triangleType function for random values and obtain the result. So the train data will consist of several test data:

input value of a, value of b, value of c

expected output: [0/1, 0/1, 0/1, 0/1] we put 1 to the corresponding triangle type and 0 for the rest

Training

We scale the input values (range 0-1) and train the network.

Prediction:

for any input data (a,b,c) we can propagate forward the scaled values.

On the output layer we get real values (probabilities intuitively) for the triangle being isosceles, scalene, equilateral, or invalid.

Software visualization

a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration

Investigates approaches and techniques for static and dynamic graphical representations of algorithms, programs (code), and processed data.

The goal is to improve our understanding of inherently invisible and intangible software, particularly when dealing with large information spaces that characterize domains like software maintenance, reverse engineering, and collaborative development.

The main challenge is to find effective mappings from different software aspects to graphical representations using visual metaphors.

Software visualization

has been applied in various areas like algorithm animation , software engineering, concurrent program execution, static and dynamic visualizations of object-oriented code, fault diagnostics , debugging, requirements analysis

properties of a software visualization system:

- Scope and content:What is the aspect of the program being visualized?
- Abstraction: What kind of information is conveyed by the visualization?
- Form and technique: How is the graphical information being conveyed?
- Method: How is the visualization specified?
- Interaction: How can the user interact with the visualization?
- Level of automation

Task-oriented view:

- Tasks: Why is the visualization needed?
- Audience: Who will use the visualization?
- Target: What is the data source to represent?
- Representation: How should it be represented?
- Medium: Where should the visualization be represented?

Software visualization sub fields:

- PROGRAM VISUALIZATION:actual program code or data structures in either static or dynamic form
- ALGORITHM VISUALIZATION:showing an abstract representation of an algorithm. Visualizations usually show the data and the effect on that data as the algorithm ms.
- ALGORITHM ANIMATION: presents the running of an algorithm as a movie where the visual representation of objects of the program smoothly change their location and appearance
- TEST DATA VISUALISATION: present test function,test data, coverage
- PROGRAM AURALIZATION: the use of sound to assist in the formation of mental images of the behavior, structure and function of a program or algorithm
- VISUAL PROGRAMMING:is a type of programming, which uses graphical objects to build software

Software Visualization Evolution

2D visualization

Two-dimensional SV techniques typically involve graph or treelike representations consisting of a large number of nodes and arcs

present pieces of the graph in different views or different windows so that the user can focus on the level of detail he desires. The software system is therefore represented in multiple windows that present to the observer different characteristics of the system under consideration

3D visualization
visualizing software in 2D does introduce a cognitive overload by presenting too much information

software objects are mapped to visual 3D objects

experiments analyzed perception in 2D and 3D and conclude that there is encouraging empirical evidence that error rates in perception are less in 3D visualization. One major advantage of 3D visualization is that it allows a user to perceive the depth of a presented structure

3D visualization for: metrics-based visualization of object-oriented programs and visualization to track software errors, isolate problems, monitor progress of development, Three-dimensional UML (Unified Modeling Language) representations

Virtual environments

Virtual environments (VEs) open possibilities of “immersion” and “navigation” that may help to better explore software structure.

VEs enable the user to interact with a representation of something familiar, namely a world with familiar objects that he/she can interact with

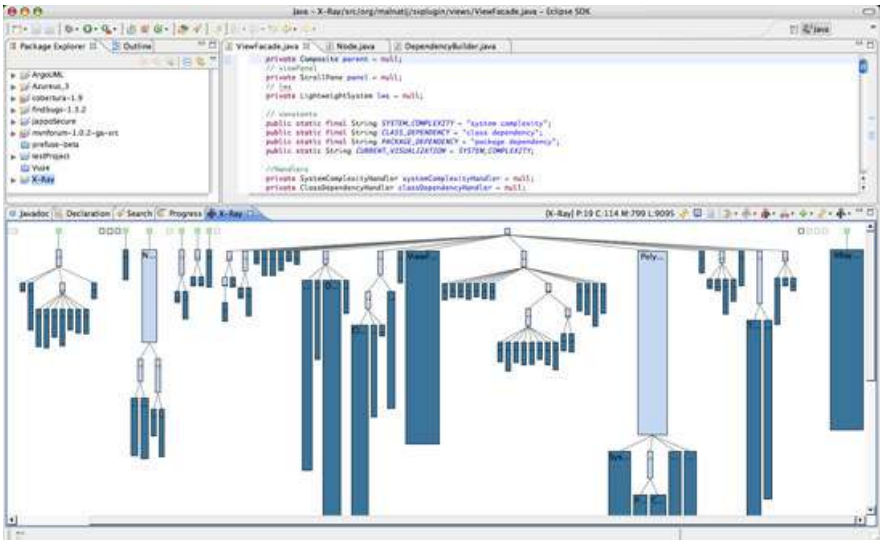
allow a user to concentrate on one aspect of the world in detail while providing a distant view of other aspects that are situated farther away. As the user moves close to each entity or visual component, it comes to “life” or presents a higher level of detail. (elision - abstracts distant objects and details closer objects)

Distributed Virtual environments

multiple users interact with each other in real time, even though those users may be physically located around the world

can be used for collaborative SV-based applications dealing with large and distributed software projects including coding, maintenance, and interactive visualization

Examples of software visualization



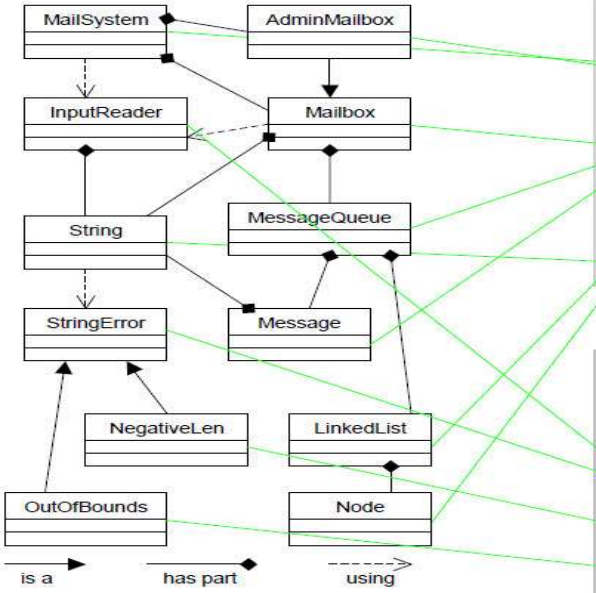


Figure 3. UML Class Diagram of MailSystem.

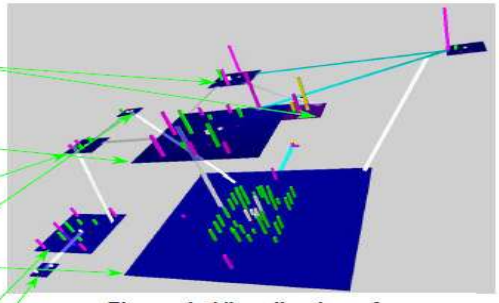


Figure 4. Visualization of a MailSystem in Imsovision.

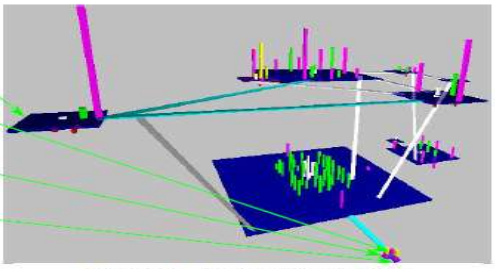
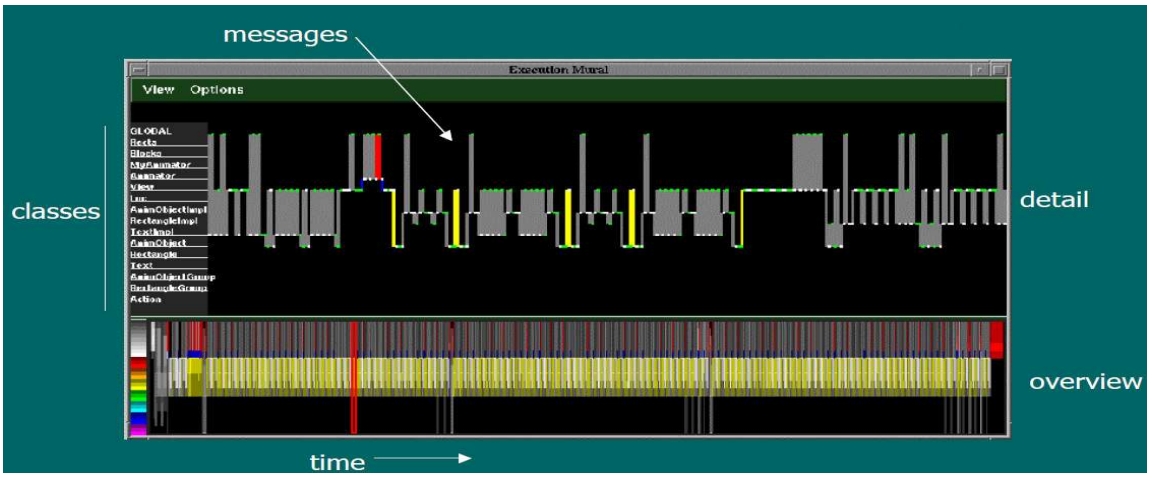


Figure 5. Another view of the MailSystem looking from the opposite direction as figure 4.



Examples of software visualization

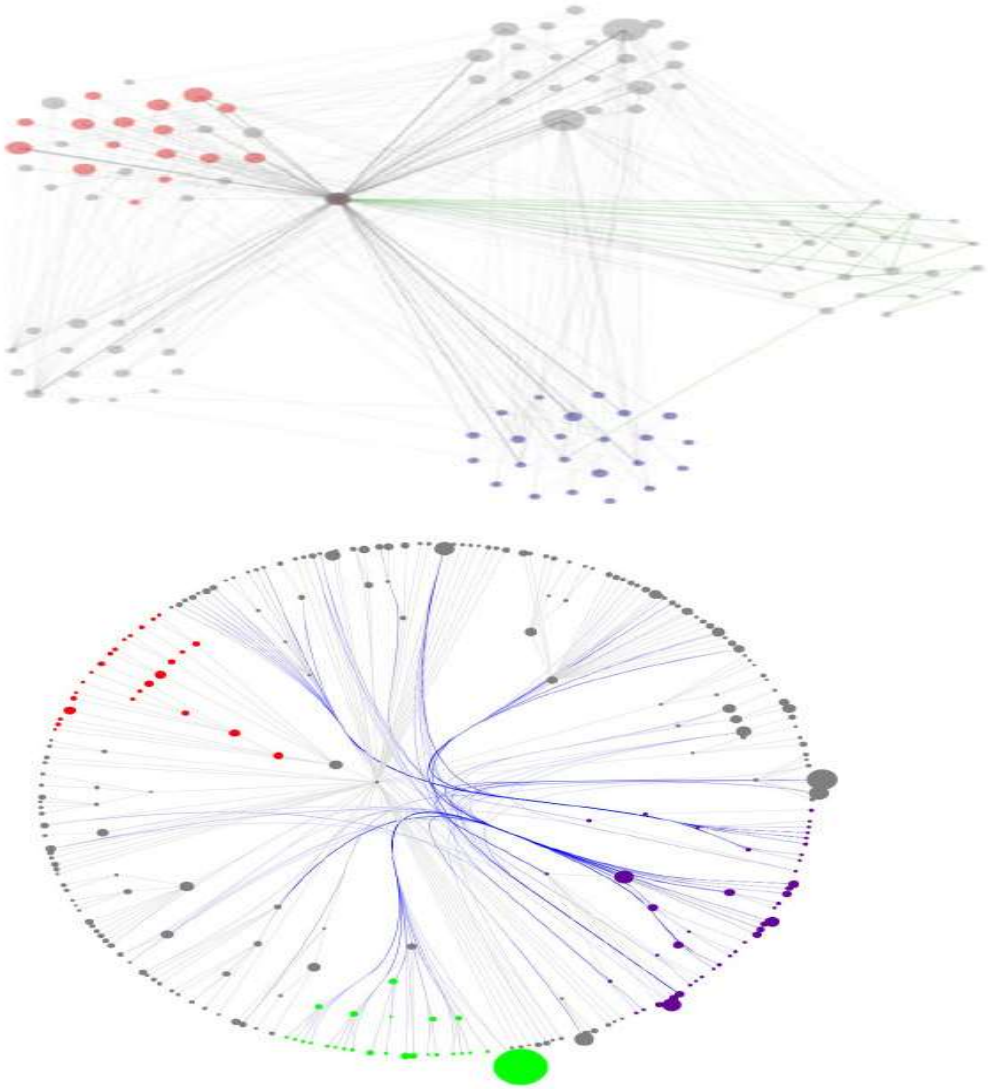


Fig. 1. Visualization of a software system

Figure 3: TARANTULA's "Continuous" view using both hue and brightness changes to encode more details of the test cases executions throughout the system.

Software visualization - Recent papers

IEEE Working Conference on Software Visualization

Session 1: Visualization Techniques

Combining Tiled and Textual Views of Code by Michael Homer, James Noble
Integrating Anomaly Diagnosis Techniques into Spreadsheet Environments by Daniel Kulesz, Jonas Scheurich, Fabian Beck

Action-Based Visualization by Antti Jääskeläinen, Hannu-Matti Järvinen, Heikki Virtanen

Slicing-Based Techniques for Visualizing Large Metamodels by Arnaud Blouin, Naouel Moha, Benoit Baudry, Houari Sahraoui

Session 2: Visualization Techniques, Paradigms, and Languages

Search Space Pruning Constraints Visualization by Blake Haugen, Jakub Kurzak

Livecoding the SynthKit: Little Bits as an Embodied Programming Language by James Noble

A Domain-Specific Language for Visualizing Software Dependencies as a Graph by Alexandre Bergel, Sergio Maass, Stéphane Ducasse, Tudor Girba

Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines
by Jabier Martinez, Tewfik Ziadi, Raul Mazo, Tegawendé F. Bissyandé, Jacques Klein, Yves Le_Traon

Validation of Software Visualization Tools: A Systematic Mapping Study
by Abderrahmane Seriai, Omar Benomar, Benjamin Cerat, Houari Sahraoui

Using a Task-Oriented Framework to Characterize Visualization Approaches
by Marcelo Schots, Claudia Werner

Session 3: Formal Tool Demos

Mr. Clean: A Tool for Tracking and Comparing the Lineage of Scientific Visualization Code
by Giacomo Tartari, Lars Tiede, Einar Holsbø, Kenneth Knudsen, Inge Alexander Raknes, Bjørn Fjukstad, Nicolle Mode, John Markus Bjørndalen, Eiliv Lund, Lars Ailo Bongo

Visual Clone Analysis with SolidSDD
by Lucian Voinea, Alexandru C. Telea

Polyptychon: A Hierarchically-Constrained Classified Dependencies Visualization
by Donny T. Daniel, Egon Wuchner, Konstantin Sokolov, Michael Stal, Peter Liggesmeyer

Session 4: Compilers, Control Flow, and Debugging

How Developers Visualize Compiler Messages: A Foundational Approach to Notification Construction
by Titus Barik, Kevin Lubick, Samuel Christie, Emerson Murphy-Hill

Lightweight Structured Visualization of Assembler Control Flow Based on Regular Expressions
by Sibel Toprak, Arne Wichmann, Sibylle Schupp

Templated Visualization of Object State with Vebugger
by Daniel Rozenberg, Ivan Beschastnikh

The Challenge of Helping the Programmer during Debugging
by Steven P. Reiss

Session 5: Evolution

ChronoTwigger: A Visual Analytics Tool for Understanding Source and Test Co-evolution
by Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E. Young, Pourang Irani

Visualizing the Evolution of Systems and Their Library Dependencies
by Raula Gaikovina Kula, Coen De_Roover, Daniel German, Takashi Ishio, Katsuro Inoue

AniMatrix: A Matrix-Based Visualization of Software Evolution
by Sébastien Rufiange, Guy Melançon

Session 6: Developers and Teams

Visualizing Developer Interactions
by Roberto Minelli, Andrea Mocci, Michele Lanza, Lorenzo Baracchi

Information Visualization for Agile Software Development
by Julia Paredes, Craig Anslow, Frank Maurer

FAVe: Visualizing User Feedback for Software Evolution
by Emitza Guzman, Padma Bhuvanagiri, Bernd Bruegge

Validation of software visualization tools

Validation of Software Visualization Tools: A Systematic Mapping Study
by Abderrahmane Seriai, Omar Benomar, Benjamin Cerat, Houari Sahraoui

consider 752 articles from multiple sources, published between 2000 and 2012, and study the validation techniques of the software visualization articles

Research questions:

- How mature is software visualization research with respect to validation?
- What validation techniques have been used for the assessment of visualization research?

cover the period spaning from 2000 to 2012 and query two publication databases:

- IEEE Xplore(ieeexplre.ieee.org)
- Scirus (www.scirus.com)

Software Visualization articles

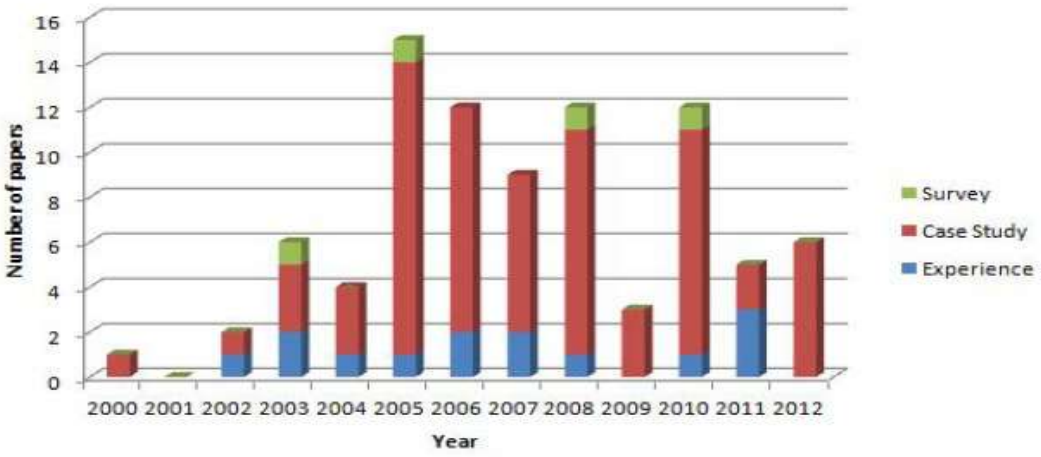


Fig. 6: Investigation type distribution per year.

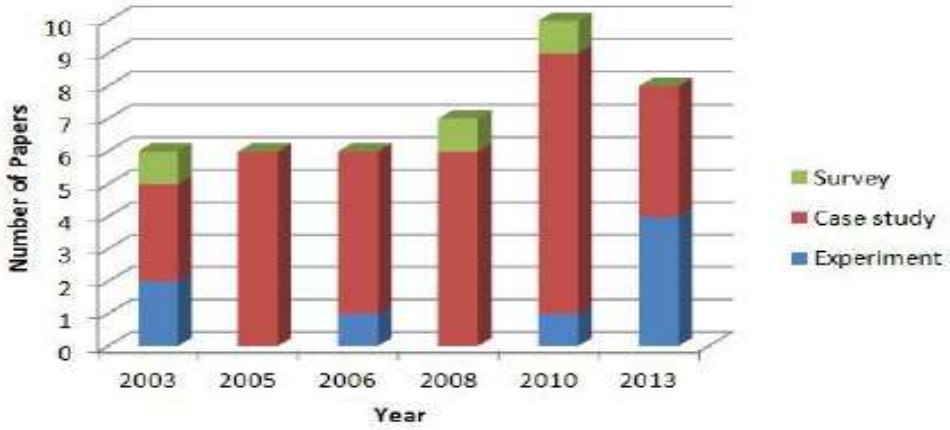


Fig. 7: Softvis investigation type distribution per year.

Investigation Types

- Experiment: it is possible to control the studied phenomenon, to manipulate the factors which can influence it, to validate or invalidate a hypothesis. The studied hypotheses consist of testing particular cause and effect relationships (e.g., the proposed visualization tool helps decrease the comprehension effort). As usual, the studies try to reject a null hypothesis (e.g., there is no difference in comprehension effort using the proposed visualization tool).
- Case study: studying a given case (product, project, organization) to collect detailed information or to make an exploration. Its outcome often serves as preliminary information of an experiment.
- Survey: It is a retrospective study of a situation

Validation of software visualization tools

Property	Class	Number of articles
Investigation Type	Experiment	14
	Case Study	68
	Survey	5
Task	Specific	63
	Exploratory	24
Data Source	Open Source	67
	Industrial	20
	Home Data	7
Subjects	None	61
	Students	12
	Practitioners	6
	Both	8
Objective Measures	Yes	53
	No	34
Comparison	None	67
	Visualization	14
	Domain	2
	Both	4

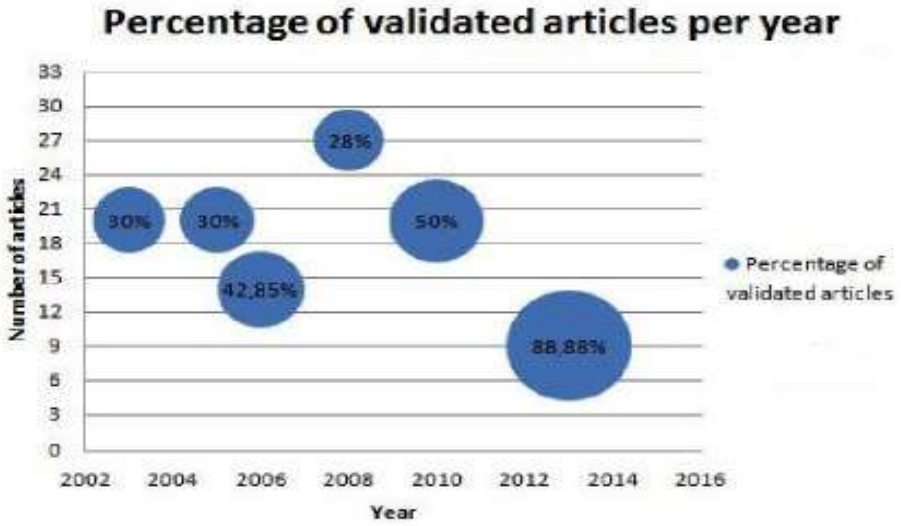


Fig. 8: Ratio of validated articles per year.