Modeling XP Refactoring Using Random Graphs

Michele Marchesi
DIEE
Università di Cagliari
Piazza d'Armi
I-09123 Cagliari, Italy
+39(070)675 5899
michele@diee.unica.it

Giancarlo Succi
Center for Applied Software Engineering
Free University of Bolzano-Bozen
Piazza Domenicani 3
I-39100 Bolzano-Bozen, Italy
+39(0471)315-640
Giancarlo.Succi@unibz.it

Nicola Serra
DIEE
Università di Cagliari
Piazza d'Armi
I-09123 Cagliari, Italy
+39(070)675 5899
nicola.serra@diee.unica.it

ABSTRACT
This paper describes a random graph based model to represent an object oriented software system. The system is represented as a graph made of a set of nodes and a set of arcs with given crossing probabilities. The model provides also the representation of the refactoring process as a random propagation through the graph.

Empirical data taken by real software projects are used for a first validation of the approach.

Keywords
Refactoring, Random Graphs, Software Engineering.

1 INTRODUCTION

Extreme Programming (XP) [1] succeeds where requirements are constantly changing. In fact, response to change is a kernel concept of XP approach. This means that during XP process a software system is constantly evolving and it is subject to continuous modification.

There are two kinds of system changes. The first is the incremental change, when functionality is gradually added, producing temporary and intermediate releases. The second is the evolutionary change, occurring when design and code are constantly revisited and simplified, in the refactoring phase.

Understanding how a single modification affects the whole system may help in reducing the cost of the XP development process and to make better usage of the team efforts. The model we introduce in this paper provides a representation of software systems based on random graphs theory. The system is represented as a graph made of nodes and arcs. A node is the representation of a specific software entity – for instance, a class – and an arc is the representation of a specific relationship between two software entities – for instance, inheritance. Given a specific system, there is no randomness in its structure, since the graph representing the system is perfectly determined. On the contrary, the evolution of a specific software system due to maintenance or refactoring shows a random behaviour, and it may be modelled as the growth process of a random graph.

The way a change influences the whole system is computed as a fluid random propagation through the graph representing the system itself. The main hypothesis is that the modification does not change the graph representation of the system, or modifies it in a negligible way. Under this hypothesis, the impact of a modification can be computed on the system representation before the changing takes place.

In the followings, we present our model, introduce significant metrics built on it, and provide some results on the application of the model to four large C++ projects performed at a North American telecommunication company.

2 RANDOM GRAPHS

Random graphs theory was devised around '60 by two mathematicians, Paul Erdos and Alfred Rèmyi [3], and it was developed during the last decade by researchers such as Palmer [9] and Bollobas [2]. Random graphs theory has been successfully applied to model systems characterized by high level of complexity in different fields such as biology, sociology, computer science.

According to Erdos and Rèmyi, a random graph can be built starting from a set of n nodes without connections. Then, considering each possible couple of nodes, a connection is drown with probability p or nothing is done with probability 1-p.

Starting from this simple definition Erdos and Rèmyi demonstrated a number of proprieties. All proprieties are given on *almost all random graphs,* meaning that the probability the propriety is true approaches 1 as the number of nodes becomes large ($n \to \infty$).

Two different formal definitions are possible [7]. In the first, a Complete Graph is defined as follows:

$K_n \equiv \{E_n , [n]\}$, where $[n] \equiv \{1,2,..,3\}$ is the Set of Nodes and $E_n \equiv \{(i,j) \mid i \in [n], j \in [n], i \neq j\}$ is the Complete Set of Arcs. Given a Complete Graph $K_n \equiv \{E_n , [n]\}$, a Random Graph is defined as follows:

$$G_{n,m} \equiv \{E_{n,m} , [n]\} \quad (1)$$

where $E_{n,m}$ is a random subset of m arcs extracted from E_n.

The second definition, called Independent Model, doesn't need the definition of Complete Graph. Given $0 \leq p \leq 1$, the Independent Model is defined as follows:

$$G_{n,p} \equiv \{E_{n,p} , [n]\} \quad (2)$$

where every possible arc $\varepsilon \in E_n$ appears independently on $E_{n,p}$ with probability p.

As the number of nodes become large, model $G_{n,m}$ and the independent model $G_{n,p}$ become equivalent, and a

specific propriety true on $G_{n,m}$ is also true on $G_{n,p}$.

The previous definitions may be modified to adapt them for representing a given real situation or focusing on certain aspects. In our context, we are interested in representing software systems and maintenance process. The model we propose may be seen as a modification of $G_{n,m}$. According to definition (1), we define $G_{n,m}$ starting from the complete graph and then randomly selecting a m-subset of arcs. Let's call our model $G_{n,m,p}$. To build $G_{n,m,p}$ we start from a colored graph with nodes and arcs of different types. Then, subgraphs of $G_{n,m}$ itself are randomly selected.

A more practical way to look at the model, is to consider it as a graph (not random) traversed by a random propagation process. According to this view, every arc is characterized by a propagation probability p. Starting from a specific node, the propagation process may cross each arc with probability p or stop with probability 1-p. Thus, the focus is on the process rather than on the static structure of the graph. This approach isn't new. A similar model has been successful adopted for representing an epidemiological virus propagation within a population or to describe how computer viruses may infect computer nets [8].

3 APPLYING THE MODEL TO THE REPRESENTATION OF SOFTWARE SYSTEMS: THE RG MODEL

Object oriented software systems are made of entities such as classes, methods, attributes, variables and relationship among such entities. For instance, a specific class is the super class of its subclasses, that is an inheritance relationship exists between the superclass and its subclass, or a specific method is related with the class it belongs to. When a module within the system needs to be changed or refactored, this could affect related modules and entities. The way a change will affect the whole system depends upon the kind of entity involved in refactoring or change and upon the kind of relationship this entity has with other software modules. If the change to an entity affects another entity, this is in turn changed, and other entities related to it may be affected. Thus, refactoring may be seen as a random process, which propagates across the software system.

We can represent a software system as a graph where a specific kind of software module is represented by a specific type of node and a specific relationship between software entities is represented by a specific type of edge. Let's call our model *RG Model.*

An instance of RG Model is made of a set of nodes of 5 different types, and a set of relationships between pairs of nodes of 9 different types (Table 1).

Table 1 – Types of Nodes and Relationship.

Node Types	**Relationship Types**
Global	Contains
Class	InnerClass
Method	Function
Attribute	Extends
Interface	Calls
	Uses
	Instance
	Parameter
	Implements

Each relationship is made of a *Server* node and a *Client* node. Moreover, for each type of relationships it is defined a crossing probability from server to client (*ProbToCLient*) and a crossing probability from client to server (*ProbToServer*).

For example, suppose n_i to be the representation of a software module needing a refactoring action. The Change action impact will propagate along an edge to adjacent nodes with probability *ProbToClient* if n_i is the server node or with probability *ProbToServer* if n_i is the client node of the relationship.

Given a specific software system, its RG Model is made of nodes and relationships connected according to the rules described by RG Metamodel shown in Table 2.

Table 2 – RG Relationships.

RG Relationships	**Client ⇒ Server**
Contains	Class ⇒ Attribute
	Interface ⇒ Attribute
	Method ⇒ Attribute
InnerClass	Class ⇒ Class
	Global ⇒ Class
Function	Class ⇒ Method
	Interface ⇒ Method
Extends	Class (subClass) ⇒ Class (superClass)
	Interface ⇒ Interface
Calls	Method (calling) ⇒ Method (called)
Uses	Method ⇒ Attribute
Parameter	Method ⇒ Attribute
Instance	Attribute ⇒ Class
Implements	Class ⇒ Interface

Let's take a look at all possible configuration admitted by an instance of RG Model.

Class nodes may be connected with *Class, Method, Attribute, Global* and *Interface* nodes. Two nodes of type Class can be connected by relationships of type *InnerClass* and/or relationships of type *Extends*. *Extends* relationship represents inheritance, with the subclass as server node and the superclass as client node. *InnerClass* implements inner class definition, with external class as client. A *Class* node may be connected to a *Method* node through the *Function* relationships, that is a method belongs to a class.

Connections between *Class* and *Attribute* nodes are provided by *Instance* and/or *Contains* relationships. *Instance*

implements the relationship between an object and its class, *Contains* implements the relationship between a class and its instance variables. *Class* nodes are related to *Global* nodes using *InnerClass* relationships and *Interface* nodes are related with *Class* nodes by *Implements* relationships.

Nodes of type *Method* may be connected to nodes of type *Method, Class, Attribute and Interface.*

The *Calls* relationship provides a representation of a call between two methods, where the called method is the server node. *Method* nodes may be connected to *Attribute* nodes through relationships of type *Uses, Parameter* and/or *Contains*. *Method* nodes may also be connected to *Interface* (or *Class*) nodes through *Function* relationships. All meanings should be clear.

Two *Interface* nodes may be connected by *Extends* relationships, providing representation of inheritance between interfaces. *Interface* nodes may connect *Attribute* nodes through *Contains* relationships.

The RG Model provides a static view of the system, that is the representation of all entities and relationships within an object oriented software system. Obviously, given a specific system, there is no randomness in its graphic representation, since all entities and relationships within the system are well determined. However, the set of *all* (existing and potential) software systems may be modeled as a random graph, characterized by defined distributions of the class, methods, attributes etc. [5].

4 THE DYNAMIC VIEW: RG VISITS AND RG METRICS.

Maintenance and refactoring are dynamic activities which start from a specific software entity and affect other parts of the system, accordingly to the type of refactored entity and to the type of relationship this entity has with other entities.

The way a change could affect the whole system has a random nature. Given the graph representation of a software system, maintenance and refactoring (which from now on we'll simply call "refactoring") may be thought as a random propagation process, which starts from the first node, typically a Class node, representing the software module needing a change, and propagates according to given probability values across the whole graph.

In our approach, refactoring is simulated with a *visit* to the graph. The visit starts from a specific Class node and propagate randomly to adjacent nodes according to certain probability values defined for each type of crossed relationship (Table 3). These values are based on programmers' empirical experience and have been devised interviewing a number of programmers and averaging their answers. To avoid unending propagation of the visits through the graph, a fading coefficient has been defined. It is a multiplicative factor smaller than one, which decreases the probabilities of propagation as long as the propagation itself is running.

As already pointed out, the underlying hypothesis of the model is that refactoring does not change, or changes in a negligible way, the structure of the graph. This may seem quite a strong assumption, since refactoring precisely aims to restructure and simplify the program. However, our hypothesis is that most part of refactoring is made working on the existing structure of the program, and that the work on the changed structure does not introduce a substantial bias. Moreover, many refactoring are made on methods, and are not likely to have impact on the graph structure.

Table 3 – Probability values of crossing the arcs of the graph.

Relationships	ProbToServer	ProbToClient
Contains	0.5	0.8
InnerClass	0.3	0.5
Function	0.8	0.0
Extends	0.2	0.9
Calls	0.2	0.5
Uses	0.2	0.0
Parameter	0.0	0.5
Instance	0.1	0.8
Implements	0.2	0.8

Three different types of metrics have been defined:

- Cost, related to the total number of lines of code (LOC) involved in propagation;
- Marks, related to the total number of visited nodes;
- Visits, integrating Marks metric over all existing Class nodes.

Each visit is independently run a large number of times, starting from every class node of the graph. Starting from a Class node and propagating across adjacent arcs, Cost metric accounts for the total number of lines of code (LOC) involved in propagation. Similarly, starting from a Class node and propagating across adjacent arcs, Marks metric accounts for of how many nodes are touched during propagation. Visits metric is quite different, and it provides a more global meaning. It records how many times a specific node is touched after a set of visits has been performed, starting from *all* the existing Class nodes.

5 RESULTS AND CONCLUSIONS

To validate the RG Model, we developed an application able to build the graph representation of a generic software system and to compute the three defined metrics. We used this application to build the RG Model and simulate all the visits on five large C++ projects developed by a North American telecommunication company. Table 4 shows the number of nodes and relationships of different types for each project. Note that no Interface nodes and no Implements relationships have been found by the application, according to the fact that all projects were developed using C++ language. For each project, the number of maintenance interventions per file, and

some classical metrics (LOC, CK) were known. Our intention was to find a correlation between defects and RG Metrics. Thus, all possible visits have been computed for each project, producing a large amount of data.

Since data about revisions and data about RG Metrics do not show normal distribution – as usually happens on software data [4] – we couldn't use the traditional Pearson coefficient.

Table 4- Nodes and arcs in the five considered projects.

	Proj. 1	Proj. 2	Proj. 3	Proj. 4	Proj. 5
Global	851	888	516	1065	544
Class	2093	2523	1217	2641	1721
Attribute	25507	28739	13761	22524	17864
Method	19395	15333	9961	19572	9724
Interface	0	0	0	0	0
Contains	38784	49154	20986	40398	30534
InnerClass	4278	5046	2434	5268	3442
Extends	2074	1274	536	1714	558
Function	42022	30798	19814	39462	19492
Calls	27901	20830	9588	10475	11032
Uses	70572	65106	37594	34406	29158
Instance	14186	10846	4382	8524	6692
Parameter	9668	6334	4996	5282	3456
Implements	0	0	0	0	0

1.1

For this reason, to correlate the number of revisions with RG Metrics, we adopted the Spearman robust correlation coefficient [6]:

$$r_s = 1 - 6 \cdot \frac{\sum_{i=1}^{n} d_i^{\,2}}{n \cdot (n^2 - 1)}$$

(3)

Given n samples, X_i and Y_i, in ascending order, Spearman correlation is computed as shown in (3), where $d_i = X_i - Y_i$.

The computation of the correlation coefficient is followed by a test of significance. Significance is the chance of correlation value not being true [10].

Table 5 shows that a correlation seems to exist. However, we have to be careful. As it usually happens in software engineering experiments, our data were incomplete and rough. The main problem is that the number of revisions for each project is given for files – not for classes. To compute defects per class we accounted for the number of lines of code per class within the file. This means that the numbers in table 5 accounts for the correlation not only between RG Metric and revisions, but also between RG Metrics and LOC.

Table 5 – Spearman correlation coefficient between three projects and the proposed metrics.

Rg metrics	Correlation	Project#1	Project#2	Project#3
Cost	Spearman	0.505	0.460	0.588
	Significance	0.000	0.000	0.000
Visits	Spearman	0.185	0.293	-0.052
	Significance	0.001	0.000	0.717
Marks	Spearman	0.048	0.369	0.383
	Significance	0.396	0.000	0.006

Table 5 shows that Cost metric is the best correlated with the number of revisions, having also perfect significance.

More experiments are obviously needed to be more definitive. The main problem remains the difficulty to find software with good documentation about refactoring, which should allow to perform significant statistical analysis.

However, the obtained results are encouraging about the usefulness of our model.

REFERENCES

1. Beck K., *Extreme Programming Explained*, Addison-Wesley, 1999.
2. Bollobas B., *Random Graphs*, Academic Press, London, 1985
3. Erdos E. e Rèmyi A., *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Kzl.5, 1960
4. Fenton N. E. e Pfleeger S. L., *Software Metrics, A Rigorous & Practical Approach,* Second Edition, PWS Publishing Company, Boston, MA, 1997
5. Focardi S.,Marchesi M., Succi G., *A Stocastic Madel of Software Maintenance and its Implication on Extreme Programming Proccess*, Extreme ProgrammingExamined, Addison Wesley, 2001.
6. Frund J.E. e Simon G.A, *Modern Elementary Statistics,* Prentice-Hall, Upper Saddle River, NJ, 1997
7. Janson S., *Random Graph*, http://www.math.uu.se/~svante/papers/sjb4.ps, 2000
8. Kephard J. O. e White S. R., *Directed-Graph Epidemiological Models of Computer Viruses*, online at: http://researchweb.watson.ibm.com/antivirus/SciPapers/Kephart/VIRIEEE/virieee.gopher.html
9. Palmer E.M., *Graphical evolution. An introduction to the theory of random graphs*, Wiley, 1985.
10. Wonnacott T.H. and Wonnacott R.J, *Introductory Statistic*, John Wiley & Sons, New York, 1997.