Estimating Relative Size When Alternative Designs Exist

D. Janaki Ram and S.V.G.K. Raju
Distributed & Object Systems Lab
Department of Computer Science & Engineering
Indian Institute of Technology Madras, Chennai
India
Phone: +91 44 4458343
E-mail: {janaki, raju}@lotus.iitm.ernet.in
http://lotus.iitm.ac.in

Abstract
Existence of alternative patterns gives rise to alternative designs. Size can be used as a criterion to select a suitable design from a set of alternative designs. Size cannot be estimated without including the application under consideration. Accuracy of the estimate depends on the amount of information known about the application. But, designing all the alternative designs with complete information and selecting one among them is not advisable. This is because the effort developers put for the alternative designs will be wasted except for the suitable one. In this paper, a method is proposed to select a suitable design based on size from a set of alternative designs that exist in an application. The method is based on the estimation of relative size of the alternative designs. Relative size estimation suppresses all common patterns and functionality among the designs. It considers only the alternative patterns that exist in the alternative designs. This method takes minimum effort to select a suitable design from the set of alternative designs based on size. Method Points have been proposed for estimating size. points. Also, this method brings out the concept of relative measure to estimate different characteristics of a software system.

Keywords
Design handbook, Alternative designs, Alternative Patterns, Pattern graph, Relative size, Null functionality, Method points.

1 Introduction

Design patterns provide solutions for recurring design problems occuring across different domains in a particular context [1]. They capture the knowledge of experienced designers in the form of a catalog in object oriented paradigm. The availability of such a catalog of design patterns helps both experienced and novice designers to recognize situations in which design reuse is possible. In the catalog proposed by Gamma et al. [1], patterns are classified based on the purpose and scope. Further, purpose is classified as creational,

structural or behavioural and scope as class or object. For a design problem, a developer may refer the pattern catalog to choose a suitable pattern for solving the problem. Even though, catalogs in their present form are very useful, an effective usage of design patterns as presented in the catalog depends very much on the experience and intuition of the designer.

It is very difficult for an inexperienced designer to compare two alternative designs that solve the same problem in different ways and select one of them. The problem of choosing an appropriate design becomes more significant with the arrival of new patterns. Specifying patterns along with quantified values of its key attributes can alleviate this problem. These quantified values help a designer in choosing the most suitable pattern for an application under consideration.

The design handbook based on design patterns [2] provides four key attributes viz. size, Static Adaptability (SA), Dynamic Adaptability (DA), and EXtendability (EX) to quantify a pattern. A designer can choose a suitable design based on the trade-off among these key attributes of the patterns in the alternative designs. The design handbook based on design patterns is briefly explained in the Section 2.

The key attribute, size in the design handbook [2] is estimated in terms of Lines of Code (LOC). But, size as a measure of LOC has disadvantages. The main disadvantages of using LOC as a unit of measure for size are the lack of universally accepted definition and language dependency [3, 4, 5]. Size can be estimated more accurately by using function point counting procedure [6] than LOC metric. In this paper, a method is proposed to estimate the size of a pattern when alternative patterns exist.

The rest of the paper is organized as follows. In Section 2, the design handbook based on design patterns is explained briefly. In Section 3, the relative size is explained. Section 4, presents the method to estimate the relative size of a design when alternative designs exist. Section 5, concludes the proposed method.

2 Design Handbook Based on Design Patterns

Selecting a suitable design pattern from a catalog depends on the experience and intuition of the designer. For example, in a printer-server application [7, 2], the problem of printer-server selecting an idle Printer can be solved in two ways. One way is to select one printer randomly among the idle printers to honour the request. In the other way, all the printers are connected sequentially one after the other and the request is submitted to the first printer. The request either accepted by the printer or forwarded to the next printer in the chain. A designer can use Iterator pattern [1] for the former case and Chain of responsibility [1] for the later. Here a designer has to choose a suitable pattern. It depends on the application under consideration and trade-off between these patterns. The patterns Iterator and Chain of responsibility are known as alternative patterns. Patterns are said to be alternative if they solve the same problem in different ways. The existence of alternative patterns for a problem provides alternative designs.

The design handbook based on design patterns provides four key attributes viz., size, SA, DA, and EX [2]. A designer can select a suitable pattern from a set of alternative patterns based on the trade-off among these key attributes. The attribute size, measures the size of a pattern in terms of LOC. SA is the measure of ease with which a pattern can be adapted to a particular context at the time of coding [7]. DA reflects the ease with which the behaviour of a pattern can be modified or adapted at runtime [7]. EX is the measure of ease with which a pattern can be extended at the maintenance phase. These four key attributes are indirect measures which are estimated from a set of direct measures. The direct measures of a pattern are obtained from its *pattern graph*. The pattern graph represents the communication among the classes of a design pattern. The pattern graph notation is shown in the Figure 1. An OO methodology based on this design handbook was proposed in [7, 2]. The methodology facilitates the developers to identify alternative patterns thereby alternative designs. A suitable design can be chosen based on the key attributes proposed in the design handbook.

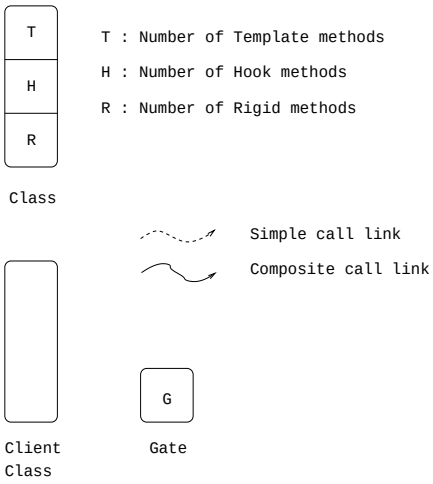

Figure 1: Pattern graph notation

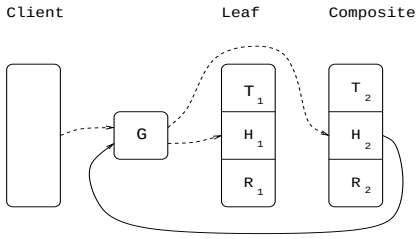

Figure 2: Pattern graph for Composite Pattern

In the pattern graph notation, classes are represented using rounded rectangles. Every class has three partitions. These partitions correspond to three types of methods in the class viz., *template methods, hook methods* and *rigid methods*. Hook methods are the methods that are declared in a class and defined in its subclass. Template methods invoke at least one hook method that may belong to any class. Rigid methods are declared and defined in the same class. Rigid methods do not call any hook or template methods. The partitions are named as template partition, hook partition and rigid partition as shown in the Figure 1. Every partition is associated with a number that represents the number of methods corresponding to that partition. The set of classes that differ only in the definition of hook methods is treated as a single class and is represented only once in the pattern graph. Any class that uses the services of a pattern is called a client class. Communication between classes is shown using arrows. The dashed-line arrow indicates one-to-one association and solid-line arrow indicates multi-valued association. Dashed-line arrow is named as simple call and solid-line arrow is named as composite call. Gate captures inheritance and polymorphism in a less formal way. Pattern graph for the Composite pattern [1] is shown in the Figure 2.

The key attributes SA, DA, and EX are out of the scope of this paper. Size estimation is based on the weighted sums of template, hook and rigid methods. This way of estimating the size is not correct as explained at the end of this section. So in this paper, we have focussed only on size and proposed a method to estimate the size of a design when alternative designs exist.

Pattern graph provides ten direct measures for a pattern to estimate the four key attributes of a pattern. The direct measures for estimating the size are given below. The other six direct measures are used to estimate DA and EX. Since only size is considered in this paper those six direct measures are not presented here. The procedures and the equations to estimate SA, DA, and EX were presented in [2].

1. **NC** : Number of Classes in the pattern graph
2. **NT** : Number of Template methods in the pattern
3. **NH** : Number of Hook methods in the pattern
4. **NR** : Number of Rigid methods in the pattern

In [2], size is estimated by using the following equation.

$$Size = w_{SZ,NT} * NT + w_{SZ,NH} * NH + w_{SZ,NR} * NR + w_{SZ,NC} * NC \quad (1)$$

All the code corresponding to an implementation of a design pattern can be broadly put under four heads namely those corresponding to template methods, hook methods, rigid methods and other aspects of a class. These are captured by NH, NT, NR, and NC. So, size can be expressed as given in the equation 1. Here, $w_{SZ,NT}$, $w_{SZ,NH}$, $w_{SZ,NR}$, and $w_{SZ,NC}$ are weights. For any pattern, the values of these weights are equal to the average number of LOC for a template method, hook method, rigid method and other aspects of a class in the pattern. The weights are obtained by studying the code of the existing systems.

This way of estimating the size may not give a correct value because, NH, NR, and NT are fixed irrespective of the application under consideration. As the patterns can be applied across different domains and weights in the equation 1 are calculated based on the average LOC of the existing systems, weights may not converge. By making these four direct measures application specific, and calculating the weights based on the communication among classes and/or objects, it is possible to estimate the size of a pattern more accurately. The method is explained in the following sections.

3 Estimating the Relative Size of a Design

At any phase of the software development, accuracy of the estimate depends on the amount of information known about the final product. So, to estimate the size of a pattern it is necessary to include the details of the application under consideration. But, designing all the alternative designs with complete information and selecting only one among them is not advisable. It is because, the effort used for alternative designs except for the suitable one will be wasted. Hence, it is required to select a suitable pattern with as much less information as possible. Consider the printer-server example that was explained in the section 2. The paper [7] has given two alternative designs for this example. These alternative designs are shown in the Figures 3 and 4.

In the Figures 3 and 4 User (U) is requesting the Printer-Server (PS) to print a File (F). PS selects an ideal Printer (P) among the existing printers. Alternative design one (Figure 3) has Prototype, Singleton, and Iterator patterns and alternative design two (Figure 4) has Prototype and Chain of responsibility patterns. In both the designs Prototype pattern is common. So to select a suitable design based on size, Prototype pattern can be omitted. Also, the functionality of a system should be the same from the end users' perspective regardless of the chosen design. The difference is only in terms of how a designer has designed the problem under consideration. So, to select a suitable design from a set of alternative designs it is sufficient to

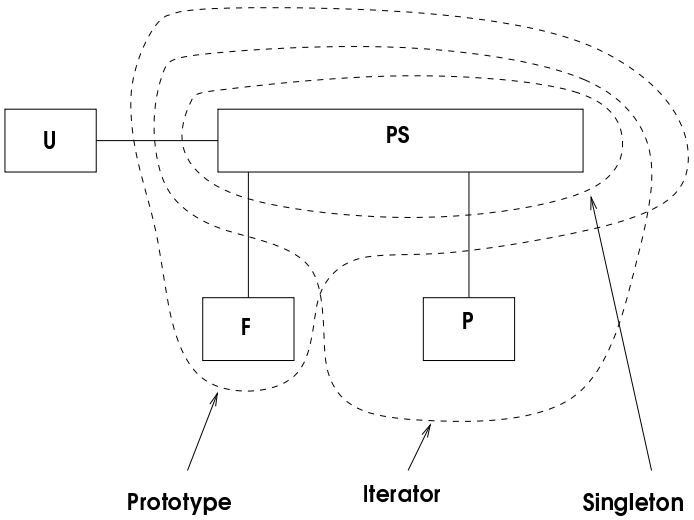

Figure 3: Alternative Design One

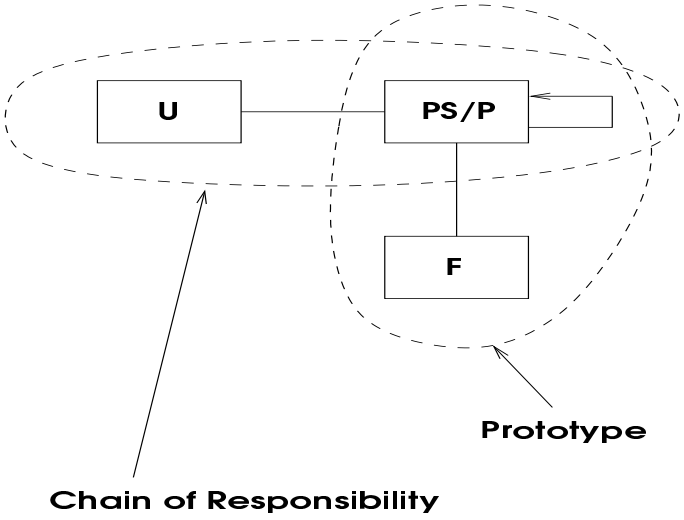

Figure 4: Alternative Design Two

estimate the size of the structure of each alternative pattern in each alternative design. This gives a relative size measurement among the alternative designs. To estimate the size of the structure of a pattern it is adequate to consider the communication among the classes and/or objects within that pattern. In other words the size of the structre of a pattern estimates the number of LOC that is required to implement the *Null Functionality* of the pattern. Huang [8] has developed automatic design pattern code generator which gives null functionality of the Gamma et al. patterns [1].

Object Oriented Function Points (OOFP) [9] and Object Oriented Design Function Points (OODFP) [10] are based on the logical design and assume complete information that is supposed to be required at the corresponding phases. Hence, OOFP or OODFP cannot be used to quantify the null functionality. So, a new procedure called *Method Points* (MP) is proposed to estimate the null functionality of an alternative pattern.

4 Method Points

In this section, it is explained how to estimate the relative size of an alternative design. Relative size of a design is the sum of the number of LOC that are required to implement the null functionality of the alternative patterns that exist. Relative size is estimated with the help of pattern graphs of the corresponding patterns.

4.1. Pattern Partition

A pattern is partitioned into two parts.

1. **Application Specific Part** tells about the number of similar classes, hook methods, template methods and rigid methods that are required to complete the structure of a pattern based on the application under consideration. Similar classes are explained in the Section 4.1.1.
2. **Fixed Part** gives the abstract structure of a pattern i.e. pattern graph.

The methods in a class are broadly categorized into hook methods, template methods and rigid methods. Hook methods are declared in a class and defined in the derived classes. As the number of children for a class increases the size corresponding to that hook method also increases. Template method invokes a hook method and rigid method does not invoke either hook or template method. So, these methods should be considered independently for estimating the size of the structure of a pattern. Size of the structure of a pattern also depends on the number of classes in that pattern. The client is not a part of the structure of a pattern. Even then it tells with how much effort somebody can use the pattern. Therefore, the client should also be considered while calculating the size of the structure of a pattern. For a pattern, the communication among the classes and/or objects is fixed. So, the fixed part can be captured as weights for the corresponding pattern.

The following equation estimates the size S_i of the structure of a pattern i.

$$S_i = \begin{cases} \sum_{j=1}^{NC} (w_{NH_j} * NH_j * NC_j + \\ w_{NT_j} * NT_j + w_{NR_j} * NR_j + \\ w_{NC_j} * NC_j) + w_{Client} \end{cases} \quad (2)$$

4.1.1. Application Specific Part

Application specific part captures the values of NH_j, NT_j, NR_j and NC_j. NH_j, NT_j and NR_j are the number of hook, template and rigid methods required for a class j in the pattern graph specific to an

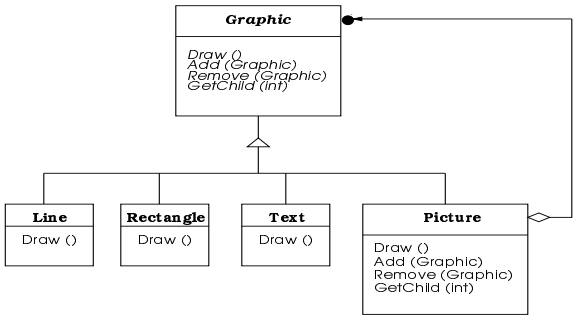

Figure 5: Composite Pattern: Drawing-Editor

application. NC_j is the number of similar classes that are required for a class j in the pattern graph. It depends on the application under consideration. If there are no similar classes then NC_j equals to one. Two classes are said to be similar if they belong to the same category. For example, consider the Composite pattern [1] for the Drawing-Editor application [1]. Its structure and pattern graph are shown in the Figures 5 and 2 respectively. The classes Line, Rectangle and Text belong to the category of Leaf class. The class Picture belongs to the category of Composite class. So, NC_1 is three for the Leaf class and NC_2 is one for the Composite class. NH_1, NT_1, NR_1, and NC_1 for the Leaf class are 1, 0, 0, and 3 respectively. NH_2, NT_2, NR_2, and NC_2 for the Composite class are 4, 0, 0, and 1 respectively.

4.1.2. Fixed Part

Pattern graph is derived from class interaction diagram and object interaction diagram of a pattern, which is fixed irrespective of the application. So, pattern graph captures the fixed part of a pattern. Hence, weights can be obtained from a pattern graph that are fixed for a pattern independent of the application. w_{NH_j}, w_{NT_j}, w_{NR_j}, w_{NC_j} and w_{Client} are the weights corresponding to NH_j, NT_j, NR_j, NC_j and $Client$.

4.1.3. Calculating Weights

In a pattern graph the communication among classes and/or objects are shown by simple and composite links. A simple call link from a method or client is considered as **ST** correspondingly [9, 10]. Also composite call link from a method or Client is considered as **CT** correspondingly [9, 10]. Objects in a pattern structure are not included in its pattern graph. But, they also contribute to the size of a structure of a pattern. Hence, weight for a method or client that is responsible for creating an object should be given 1 CT. For example, in an Abstract Factory pattern [1], hook method in the *AbstractFactory* class is responsible for creating a concrete products. A class reference is considered as a complex type [9, 10]. So, weight for each NC_j is considered as 1 CT. Hook methods will be overridden in the subclasses so in the Equation 2, w_{NH_j} is multiplied by both NH_j and NC_j. Based on this method the weights for the design patterns referred in [1] are given in the Table 1. The weights for Composite pattern are given in the Table 1. The size of the Composite pattern for Drawing-Editor application (Figure 5) by using the Equation 2 is 8 CT and 2 ST. But to compare the size of different patterns, ST and CT should be mapped to a single value.

4.1.4. MP Estimation

Application specific data is not considered because common functionality of the alternative designs is suppressed. In this case, OOFP and Function Point (FP) [6] tables cannot be used to map ST and CT to a single value. They will give high value even for a simple problem.

[11, 12] have given threshold values for average number of children per class as six, number of methods overridden by a subclass as three, and average number of instance methods per call as twelve. Also it is observed that a pattern consists of maximum number of three pattern classes other than client. According to the authors observation across different patterns [1, 13] not more than two key classes consists of maximum threshold values. From the Table 1, the following points are inferred.

- The value of w_{NH} is 1 CT or 2 ST for not more than one pattern class.
- The value of w_{NT} or w_{NR} is 1 CT or 1 ST for not more than one pattern class.
- Maximum value of NC_j is 6
- Maximum value of NH_j is 3
- Maximum value of $NR_j + NT_j$ is 12
- Maximum value of w_{Client} is 1 CT or 2 ST

From these observations and Equation 2 maximum value of CT for any pattern is 43 {(6*3)+12+(2*12)+1}. Similarly maximum value of ST is 50 {((6*3)*2)+12+0+2}.

To convert ST and CT to a single value, different patterns are implemented with null functionality in C++ with varying values of NH_j, NT_j, NR_j, and NC_j. Also, tools [14, 8] are available to generate code automatically for a given design pattern with application specific information. Null functionality was considered because data was not included for estimating the size of the structure of a pattern. Jones [15] has observed different projects and has given that 1 FP = 30 LOC of C++. The same value of LOC was considered for 1 MP. For each implemented pattern, ST and CT were calculated by using the corresponding weights and Equation 2. The values of MP for each implemented pattern were tabulated with CTs as rows and STs as columns. Right now method points were obtained only from a fixed set of patterns [1]. As this backfiring is applied on different patterns across different domains it is possible to get more accurate values of method points. The method points are presented in the Table 2.

Size of an alternative pattern can be estimated in the following manner:

1. Calculate w_{NH_j}, w_{NT_j}, w_{NR_j}, w_{NC_j} and w_{Client} from the pattern graph for each alternative pattern in a design.
2. Identify NH_j, NT_j, NR_j and NC_j based on the application.
3. Equation 2 calculates the total number of Simple Types (ST) and Composite Types (CT) for the corresponding pattern.
4. MP can be obtained by using the Tables 2.
5. An organization can use their own conversion value from MP to LOC or Jones [15] values.

To see the trade-off, consider the printer-server problem that was discussed in Section 2. The values of SA, DA and EX are taken from [2]. Assume that five printers are connected to the print-server. Iterator pattern has two pattern classes in the pattern graph viz., aggregate and iterator. For aggregate class NH_1, NT_1, NR_1 and NC_1 are 1, 0, 0 and 5 respectively. The hook methods considered for the iterator class are *First (), Next (), IsDone () and CurrentItem ()*. NH_2, NT_2, NR_2 and NC_2 for iterator class are 4, 0, 0 and 1

respectively. Chain of responsibility has one class in the pattern graph viz., Handler. For Handler NH_1, NT_1, NR_1 and NC_1 are 1, 0, 0 and 5. From the Table 3 it is observed that alternative design one is better from the view point of DA, whereas alternative design two has less LOC and more EX and SA. Therefore, the effort required to implement the alternative design two will be less. A designer can select a suitable pattern according to the application and the trade-off shown in the Table 3.

4.3. Relative Measure

The method proposed for estimating the size of the structure of a pattern brings out the concept of estimating the relative measure. Assume that there are different solutions to solve a problem for a software system. Then it is adequate to estimate relative measures for different characteristics for that system. Hence, relative measures are placed well to choose a suitable solution when alternative solutions exit to solve a problem.

5 Conclusions

It is not always possible to use tools for code automation, could be because of its cost. [8] tool generates automating code only for Gamma et al. patterns [1]. So, it would be useful if simple manual procedure is available to estimate the relative size of alternative designs. A simple and elegant method is proposed and illustrated to estimate the relative size of a design when alternative designs exist. While calculating size, the method considers the application under consideration. The proposed method minimizes the effort involved in choosing a suitable design specific to an application based on size. After selecting a suitable design OODFP can be used to estimatte the actual size. To estimate the relative size method points is proposed which estimates the null functionality of alternative designs. The reliability and validity of method points needs to be tested across different projects. Finally, the method provides a new concept of relative measure that does not exist in the literature. It is possible to estimate different characteristics of a software system in terms of relative measures when a developer has alternative solutions.

REFERENCES

[1] **Gamma, E., R. Helm, R. Johnson, and J. Vlissides**, *Design Patterns: Elements of Reusable of Object-Oriented Software*. California, New York: Addison Wesley, 1995.

[2] **Janaki Ram, D., K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S. V. G. K. Raju, and A. A. Rao**, (2000), An Approach for Pattern Oriented Software Development Based on a Design Handbook, *Annals of Software Engineering*, vol. 10, pp. 329–358.

[3] **Levitin, A. V.**, (1986), How To Measure Software Size, And How Not To, 10^{th} *International Computer Software and Applications Conference*, Chicago, pp. 314–318.

[4] **Low, G. C. and D. R. Jeffery**, (1990), Function Points in the Estimation and Evaluation of the Software Process, *IEEE Transactions on Software Engineering*, vol. 16, pp. 64–71.

[5] **Matson, J. E., B. E. Barrett, and J. M. Mellichamp**, (1994), Software Development Cost Estimation Using Function Points, *IEEE Transactions on Software Engineering*, vol. 20, pp. 275–287.

[6] **IFPUG**, *Function Point Counting Practices Manual Release 4.1.* Westerville, Ohio: International Function Point Users Group, 1999.

[7] **Janaki Ram, D., K. N. Anantharaman, and K. N. Guruprasad**, (1997), A Pattern Oriented Technique for Software Design, *ACM Software Engineering Notes*, vol. 22, pp. 70–73.

[8] **Huang, J. Q.**, The Designer's Assistant Interface. Available at http://csg.uwaterloo.ca/dptool/interfac.htm.

[9] **Caldiera, G., G. Antoniol, R. Fiutem, and C. Lokan**, (1998), Definition and Experimental Evaluation of Function Points for Object-Oriented Systems, *Proc. of the* 5^{th} *International Symposium on Software Metrics*, November, pp. 167–178.

[10] **Janaki Ram, D. and S. V. G. K. Raju**, (2000), Object Oriented Design Function Points, *Proceedings of the Frist Asia-Pacific Conference on Quality Software*, Kowloon, Hong Kong, October, pp. 121–126.

[11] **Linda, H. R.**, (1998), Applying and Interpreting Object Oriented Metrics, *Software Technology Conference*, Utah. Available at http://www.hq.nasa.gov/office/hqlibrary/books/nasadoc.htm.

[12] **Lorenz, M. and J. Kidd**, *Object-Oriented Software Metrics*. New Jersey: PTR Prentice Hall, 1994.

[13] **Lavender, R. G. and D. C. Schmidt**, (1995), Active Object : An Object Behavioral Pattern for Concurrent Programming, *Proc. of the* 2^{nd} *Annual Conference on the Pattern Languagues of Programs*, Monticello, Illinois, September, pp. 1–7.

[14] **Budinsky, F. J., M. A. Finnie, J. M. Vlissides, and P. S. Yu**, (1996), Automatic Code Generation from Design Patterns, *IBM Systems Journal*, vol. 35, no. 2, pp. 151–171.

[15] **Jones, C.**, (1995), Backfiring: Converting lines of code to function points, *IEEE Computer*, vol. 29, pp. 86–87.

Table 1: **Weights for Gamma Patterns**

Pattern	**j**	**Class Name**	w_{NH}	w_{NT}	w_{NR}	w_{NC_j}
Abstract Factory	1	Abstract Factory	1 CT	0	0	1 CT
		Client	1 ST			
Builder	1	Director	0	1 ST	0	1 CT
	2	Builder	1 CT	0	0	1 CT
		Client	2 ST			
Factory Method	1	Creator	1 CT	1 ST	0	1 CT
		Client	1 ST			
Prototype	1	Prototype	0	0	0	1 CT
		Client	1 ST			
Singleton	1	Singletor	0	0	0	1 CT
		Client	1 ST			
Class Adapter	1	Adapter	1 ST	0	1 CT	1 CT
		Client	1 ST			
Object Adapter	1	Adapter	1 ST	0	0	1 CT
	2	Adaptee	0	0	0	1 CT
		Client	1 ST			
Bridge	1	Abstraction	0	1 ST	0	1 CT
	2	Implementor	0	0	0	1 CT
		Client	1 ST			
Composite	1	Leaf	0	0	0	1 CT
	2	Composite	1 CT	0	0	1 CT
		Client	2 ST			
Decorator	1	Component	0	0	0	1 CT
	2	Decorator	2 ST	0	0	1 CT
		Client	2 ST			
Flyweight	1	Factory	0	0	0	1 CT
	2	Shared	0	0	0	1 CT
	3	Unshared	0	0	0	1 CT
		Client	2 ST, 1CT			
Proxy	1	Proxy	1 ST	0	0	1 CT
	2	Real Subject	0	0	0	1 CT
		Client	1 ST			
Chain of Responsibility	1	Handler	1 ST	0	0	1 CT
		Client	1 ST			
Command	1	Invoker	0	1 ST	0	1 CT
	2	Command	1 ST	0	0	1 CT
	3	Receiver	0	0	0	1 CT
Interpreter	1	Terminal Expression	0	0	0	1 CT
	2	Nonterminal Expression	1 CT	0	0	1 CT
		Client	1 CT, 2 ST			

Table 1: Weights for Gamma Patterns (contd.)

Pattern	j	Class Name	w_{NH}	w_{NT}	w_{NR}	w_{NC_j}
Iterator	1	Aggregate	1 CT	0	0	1 CT
	2	Iterator	0	0	0	1 CT
		Client	2 ST			
Mediator	1	Mediator	1 CT	0	0	1 CT
	2	Colleague	1 ST	0	0	1 CT
		Client	0			
Observer	1	Subject	0	1 CT	0	1 CT
	2	Observer	1 ST	0	0	1 CT
		Client	2 ST			
State	1	Context	0	1 ST	0	1 CT
	2	State	0	0	0	1 CT
		Client	1 ST			
Strategy	1	Context	0	1 ST	0	1 CT
	2	Strategy	0	0	0	1 CT
		Client	1 ST			
Template Method	1	Class	0	1 ST	0	1 CT
		Client	1 ST			
Visitor	1	Object Structure	0	1 CT	0	1 CT
	2	Element	1 ST	0	0	1 CT
	3	Visitoir	0	0	0	1 CT
		Client	1 ST			

Table 2: CT & ST Table

	1-10 ST	**11-20 ST**	**21-30 ST**	**31-40 ST**	**41-50 ST**
1-5 CT	MP	2 MP	2 MP	3 MP	4 MP
6-10 CT	2 MP	3 MP	4 MP	4 MP	5 MP
11-15 CT	3 MP	4 MP	4 MP	5 MP	6 MP
16-20 CT	4 MP	5 MP	6 MP	7 MP	8 MP
21-25 CT	6 MP	7 MP	8 MP	9 MP	10 MP
26-30 CT	8 MP	9 MP	10 MP	11 MP	12 MP
31-35 CT	10 MP	11 MP	12 MP	13 MP	14 MP
35-43 CT	12 MP	13 MP	14 MP	15 MP	16 MP

Table 3: Trade-off for the Printer-Server Problem

Pattern	**Estimated Relative Size (LOC)**	**Actual Relative Size (LOC)**	**Actual Size (LOC)**	**SA**	**DA**	**EX**
Design 1	60	77	202	0.43	5	0
Design 2	30	45	140	1	2	4