Objected Oriented Design Defect Analysis and Refactoring-Overview.

Nagaraj MS[1] Dr. R.Selvarani[2]
[1]Honeywell Technology Solutions, Bangalore, India
[2]Director –Doctoral Program – ACED, Alliance University, India

Abstract- Software design defects often lead to bugs, runtime errors and software maintenance difficulties. They should be systematically prevented, found, removed or fixed all along the software lifecycle (Design, development and maintenance stages). However, detecting and fixing these defects is still to the greater extent a difficult, time-consuming and manual process. Identifying and fixing the defects at earlier part of software life cycle will reduce the significant maintenance cost. In this paper, we propose detecting the design defects at design phase and software defects at implementation phase of the software life cycle. Detecting defects in early stage of design cycle is useful from the perspective of cost quality and schedule reduction.

Index Terms- Software design defects, software bugs, run time errors, object oriented defects, anti-patterns, code smells

I. Introduction

Object-oriented programming (OOP) is industry adapted a programming paradigm which consists of “objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, are instances of classes, are used to interact with one another to design applications and computer programs. Object oriented design, today, is becoming more popular in software development environment and Object Oriented design metrics is an essential part of software environment. The main objective of analyzing these metrics is to improve the quality of the software. Design defects, anti-patterns, code smells defects at the architectural level and software coding errors must be detected and corrected to improve software quality, automatic detection and correction of these software architectural defects, which suffer of a lack of tools. The contribution of this paper is to present issues related to the detection and correction of design defects at design level and software coding errors at implementation phase.

II. DIFFERENCES BETWEEN DESIGN PATTERNS AND DESIGN FLAWS.

Design patterns is a reusable solution to commonly occurring design problems in software. Each class in design pattern plays specific role and interactions between the classes are well defined. Anti-pattern represents undesirable design structure that is difficult to maintain and understand the software. The complexity of the classes that constitute design patterns are comparatively less than compared to classes in anti-patterns. Roles and complexity and interaction with other classes can be used to identify the design pattern because their important features can be easily defined and recognized, which is not possible in case of some anti-patterns [1] as the classes that constitute the anti-pattern higher in complexity and consists of multiple roles. Anti-patterns have very large variety of characteristics (e.g., number of methods, naming of methods/classes, method parameters, class functionality etc.), therefore it is harder to apply general detection rules to all of them.

III. DESIGN-DEFECTS OR ANTI-PATTERNS

Design defects are design structures that are complex, difficult to understand and maintain. They are bad practices in software design. Design solution that is initially appears to be a good solution for the problem to solve results in the creation of conflicts because of its implementation. Having knowledge of design defects, the developer is equipping with the knowledge needed to avoid or fix errors before writing any code or designing the software. A design defect or anti-pattern is a literary form that describes a commonly occurring solution to a design problem, solution which generates negative consequences in maintaining the software. Design defects are bad solutions to recurring design problems. The idea of design defect is to show what not to do. The Blob, the Spaghetti, the Poltergeist, the Lava Flow are among well-known design defects. For example, the Blob represents single complex controller class that monopolizes the processing and is surrounded by simple data classes. The Spaghetti code, which is one of the most famous design defect, describes a program or system with a software structure that lacks clarity and hard to maintain.

IV. RELATION BETWEEN OBJECT ORIENTED MATRICES AND DESIGN DEFECT

Design defects can be identified by measuring the objected oriented metrics coupling, cohesion, complexity and inheritance. Below table lists the relationship between the different object oriented metric categories and most commonly occurring design defects [6].

Anti-patterns	Metrics Category			
	Coupling	Cohesion	Complexity	Inheritance
Blob	High	low	High	Low
Lava Flow	Low		High	
Functional Decomposition		high	Low	Very Low
Poltergeists	High		Low	
Swiss Army Knife	High		High	

V. BLOB DESIGN DEFECT

The Blob class also known as God class of the design. The Blob class violates the "Single-responsibility principle" [14], Single-responsibility principle states there should one only one reason to change the class, The Blob class is responsible for all (or most of the) behavior of an application while the rest of the classes (the data classes) are only responsible for Encapsulating data, hence it monopolizes the processing and acts as controller class that performs majority of system responsibilities. The basic form of a god class is defined in Figure 1.

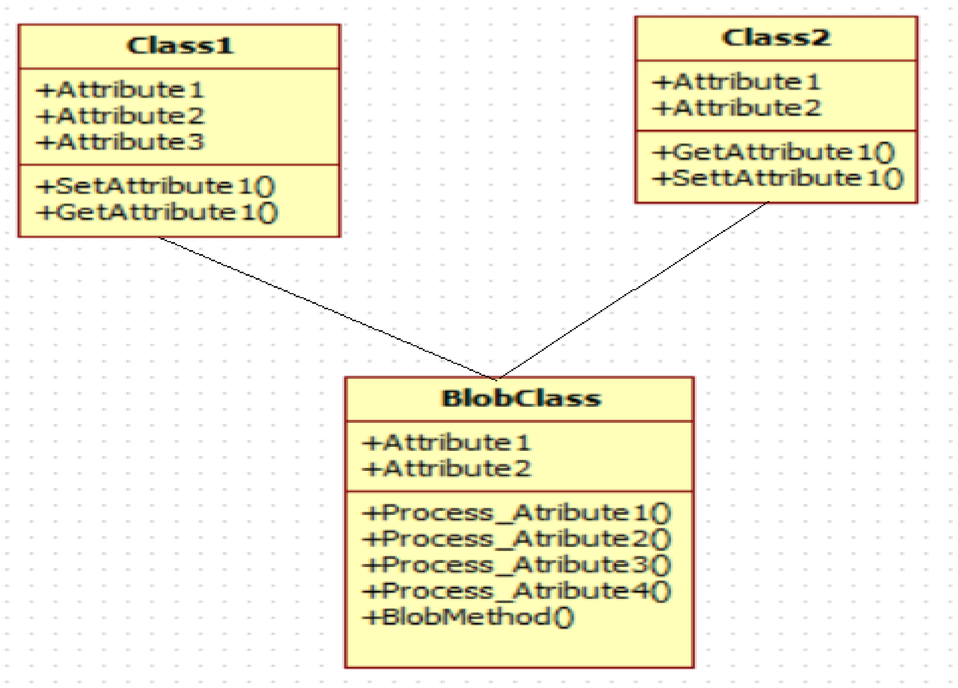


Figure 1 Blob or God Class Architecture

Blob class will consist of one or more Blob method that obtains their data from classes different from the class they belong to [2] and perform the complex computations or operations. The blob method can also be visualized as complex method that performs more than one functionality i.e. there will be more than one reason to change the method. The basic form of a god method is defined in Figure 2 by the diagram modelled with continuous lines. This shows that the god method accesses attributes from the other classes through its method which expose the attributes of the class.

Note: Arrow marks $\longrightarrow$ in all the diagrams indicate the function call. Where the pointing to Called from caller function.

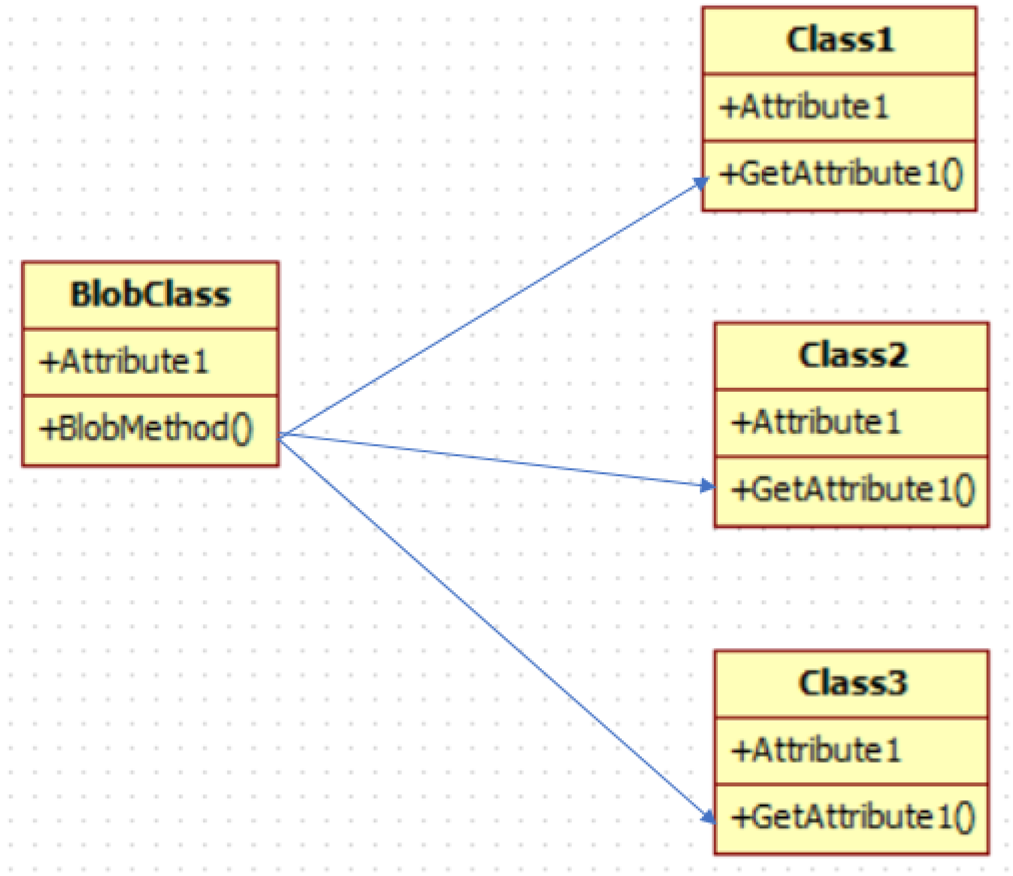


Figure 2 Blob Method Architecture

Detecting Blob pattern

As Blob method performs the complex computations, the complexity of the function is high and, they access the data from other classes resulting high coupling and low Cohesion. The Blob defect can be identified by measuring the object-oriented metrics complexity, coupling and Cohesion. Object oriented metrics are captured through software metrics and properties are expressed in terms of valid values for these metrics [11]. The most widely used metrics are the ones defined by Chidamber and Kemerer [3]. These include: Weighted methods per class, WMC, Coupling between objects, CBO.

Refactoring a Blob Class

Solution: Refactoring of Blog class exposes the operation rather than the attributes as shown in Figure 3.

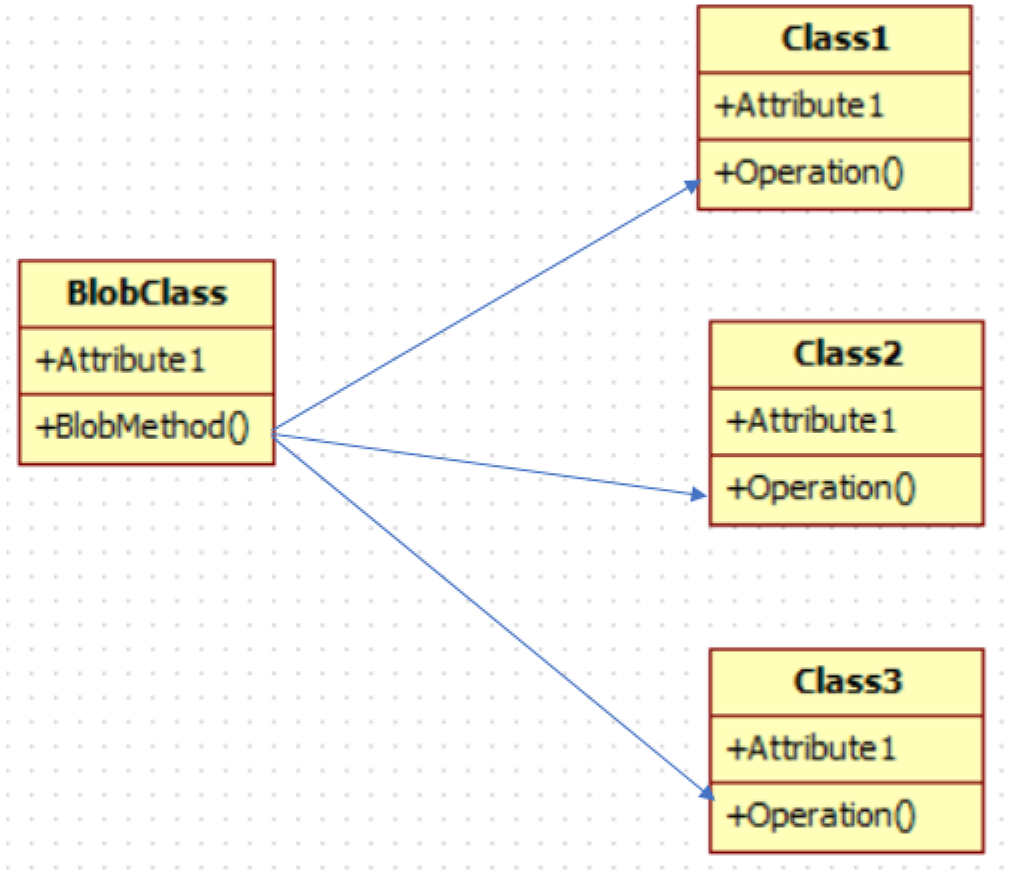


Figure 3 decomposing the responsibilities

Refactoring a Blob Method.
The Blog method consists of multiple responsibilities; this method can be decomposing until complexity reduces and becomes a function with single responsibility with different classes as shown in the Figure 3.

VI. POLTERGEISTS DESIGN DEFECT

Poltergeist is a class with minimal or limited responsibilities and roles to play in the software system; therefore, their effective life cycle is quite brief; they clutter software designs, creating unnecessary abstractions [6];
Poltergeists can occur in four different forms as follows

- Irrelevant classes:
- Agent classes
- Operation classes
- Object classes

Irrelevant classes:
An irrelevant class does not have any meaningful behavior in the software design. They composed of only of get (class data member accessor), set (sets the class data member) methods. Figure 5 shows the UML notation of design defect Irrelevant classes. The concreate class will access the irrelevant class attribute through the get method and sets attributes using set method.

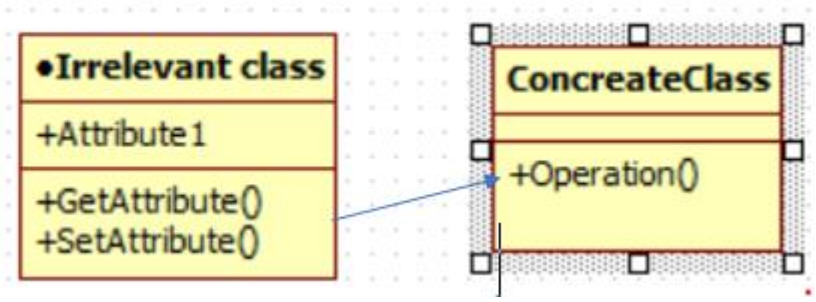


Figure 4 Irrelevant Classes

Refactoring Irrelevant Class.
Although the behavior of irrelevant classes is meaningless, the data that it may contain is not. The correction of irrelevant classes consists in both eliminating them from the design and placing the data they contain with the respective accessor class.

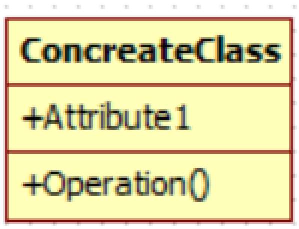


Figure 5 Refactoring irrelevant class

Agent classes:
Agent design defect are classes that are responsible for only passing the messages from one class to another, i.e., methods that offer redundant paths to access operations of other classes in the design.
The UML specification of this design defect is shown in Figure 6.

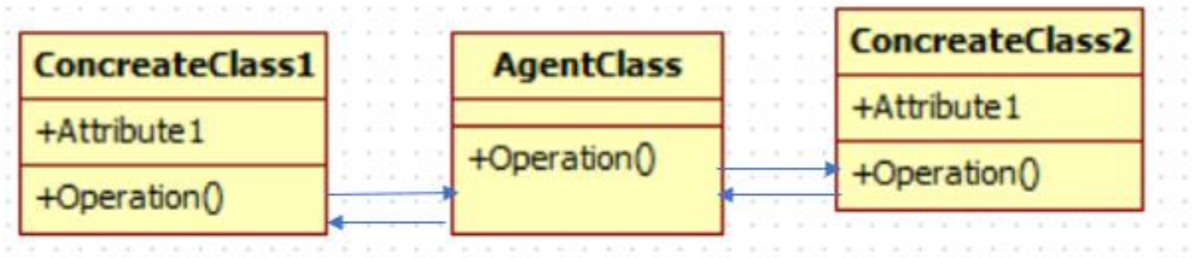


Figure 6 Agent classes

Refactoring Agent Class
Refactoring agent class involves removing the agent method from the design and replacing the communication it performs to be done directly between the other two classes involved in the anti-pattern [4].

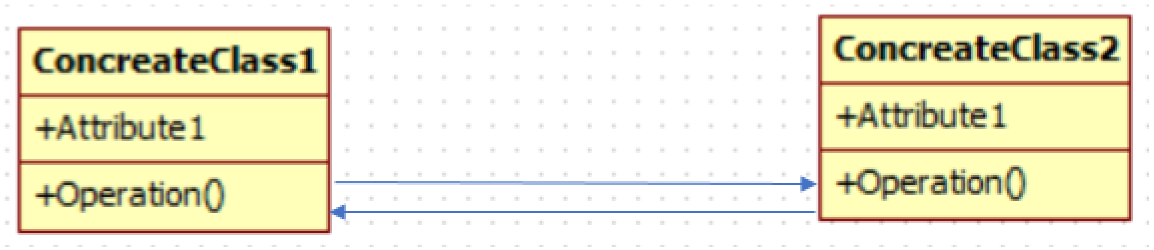


Figure 7 Refactoring Agent class

Operation classes:
Operation design defect are classes with only one meaningful behavior and for having a short life cycle. The main idea of an operation class is that an operation that should have been a method within a class has been turned into a class itself. UML notation of Operation classes is shown in Figure 8.

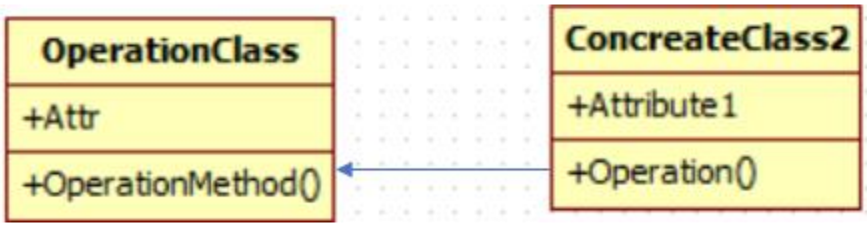


Figure 8 Operation classes

Refactoring operation class.
Refactoring of operation class design defect involves moving the attributes and functionality to suitable class.

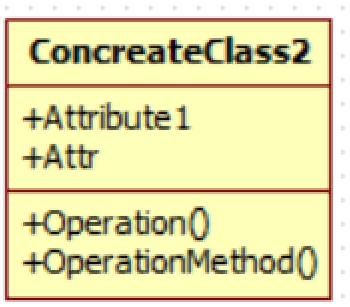


Figure 9 Refactoring operation class

Object classes:
Object classes are subclasses representing exactly the parent classes with no additional functionalities or attributes. In figure 11 classes SbClass1 and SubClass2 are Object classes as they are exactly same as their parent class ConcreateClass2.

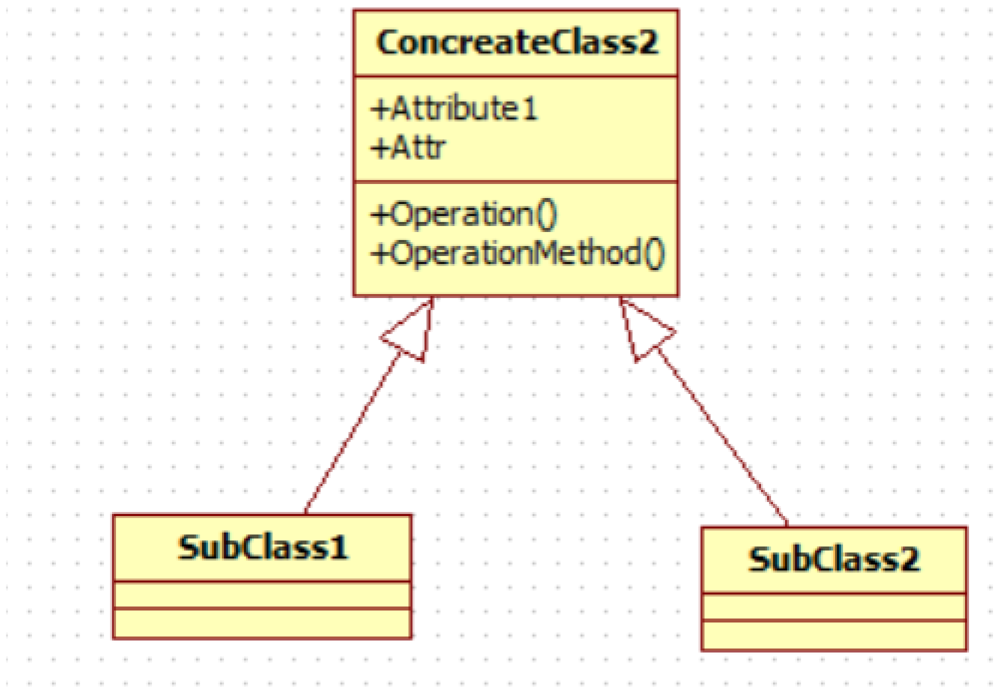


Figure 10 Object classes

Refactoring Object class.
Object class does not override any behavior or functionality of their parent class and they do not have additional behavior, they are unnecessary and therefore, must be remove them from the class hierarchy altogether.

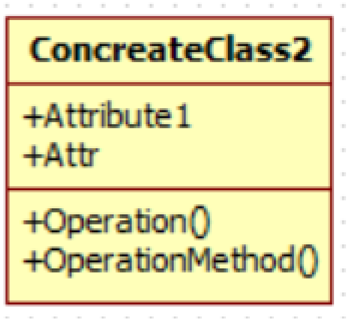


Figure 11 Refactoring Object class

VII. LAVA FLOW DESIGN DEFECT OR FUNCTIONAL DECOMPOSITION

Lava flow design defect is a class with single action such as function which makes it simple [4].
Lava flow design defect is created by designer when he creates each class for function Figure 13, resulting multiple classes in the design where the functionality not logically grouped.

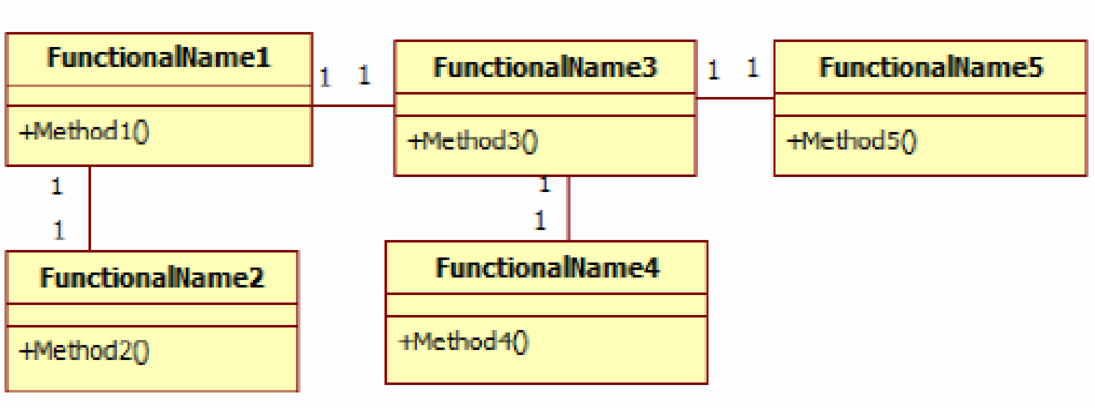


Figure 12 Lava Flow

Refactoring Function Decomposition design defect

If the class has a single method are helper classes with single functionality, remove this class by moving the method to part of an existing class base class. The goal is to consolidate the functionality of several types into a single class that captures a broader domain concept than the previous finer-grained classes. For example, rather than have classes to manage device access, to filter information to and from the devices, and to control the device, combine them into a single device controller object with methods that perform the activities previously spread out among several classes. If the class does not contain state information of any kind, consider rewriting it as a function. Potentially, some parts of the system may be best modeled as functions that can be accessed throughout various parts of the system without restriction.

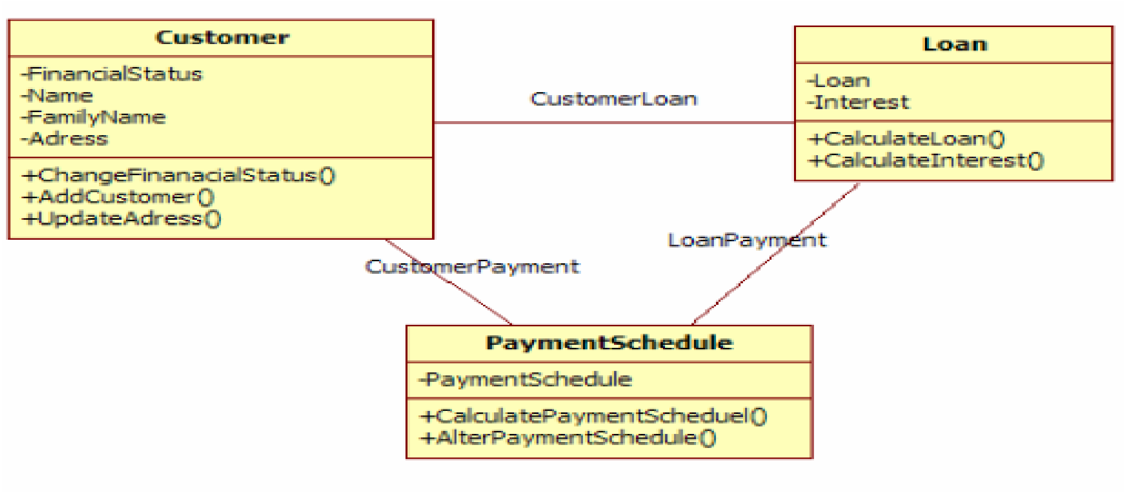


Figure 13 Refactoring Function Decomposition Design Defect

VIII. SWISS ARMY KNIFE DESIGN DEFECT

Swiss army knife class implements many interfaces to expose the maximum possible functionalities. As it implements may interface it becomes complex class exposing many functionalities. The difference between Swiss army knife and the Blob is that the Swiss army knife exposes a high complexity to address all foreseeable needs of the class, whereas the Blob is a single large multifunctional object that monopolizes all the treatment and the system data. The symptoms of the presence of Swiss Army Knife anti-pattern is : Complex interfaces with no clear abstraction or purpose for the class, which is represented by the lack of focus in the interface.

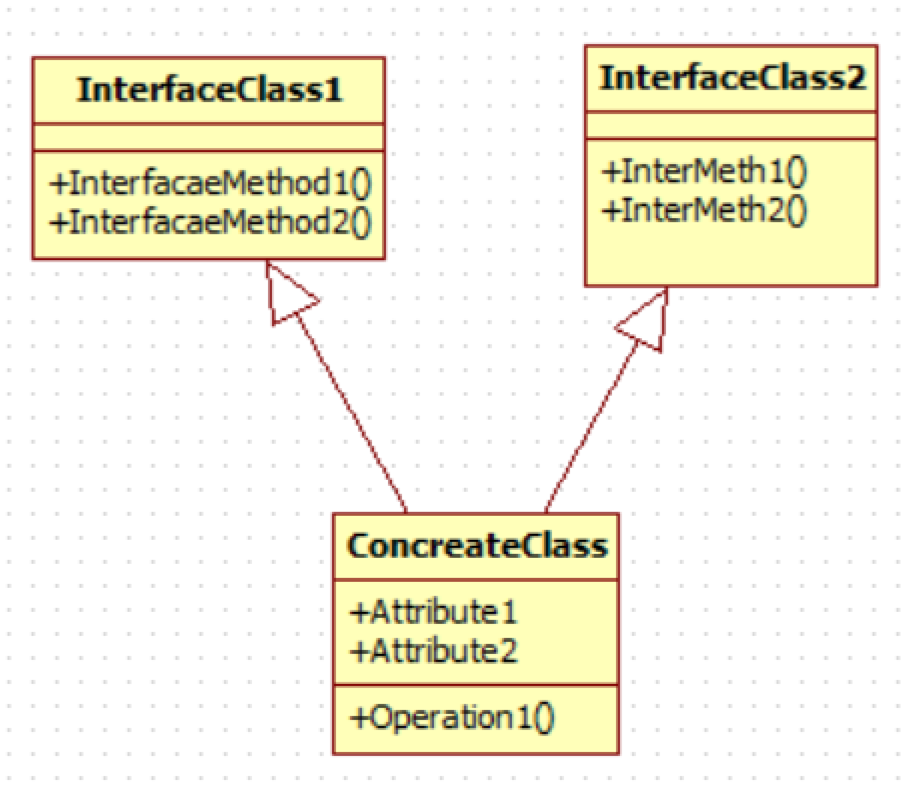


Figure 14 Swiss Army Knife

Refactoring Swiss Army Knife

Refactoring the swiss army knife involves reducing the complexity of the interfaces.

Apply Extension Interface Patterns:

- Introduce a common protocol for all provided interfaces (incl. Interface navigation)
- Integrate additional functionality so that clients can discover existing component interfaces and navigate between them.

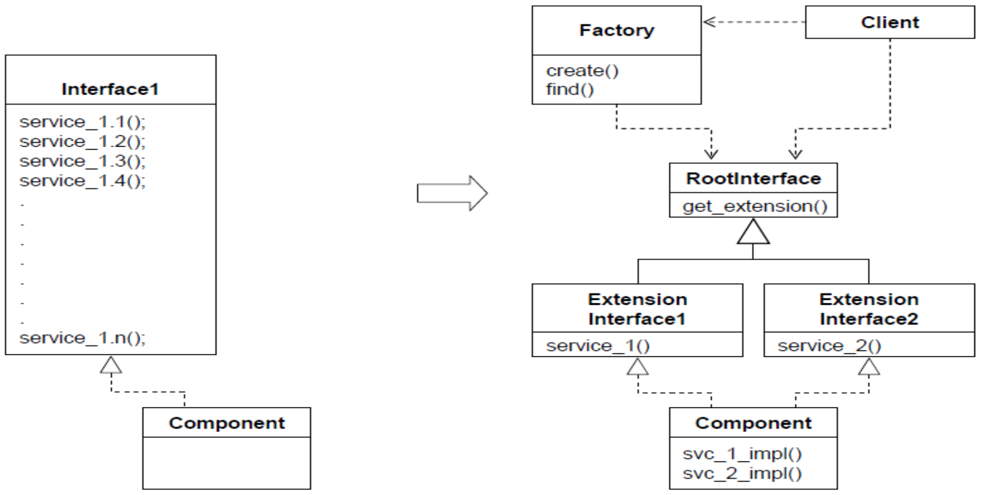


Figure 15 Refactoring of Swiss Army Knife anti-patter

IX. PROGRAMMING ERRORS OR DEFECTS.

Defects for programming coding errors, assignment versus equality operators, type mismatch, wrap around, string arrays. They occur due to the bad programming below table lists defects and their symptom and correction. Defects are under Numerical Defects, Programming Defects and Programming Defects.

Numerical Defects [8]		
Defect Type - Defect Description	**Defective Code Sample**	**Corrective Code Sample**

Float overflow - Overflow from operation between floating points	float square(void) {
	float val = FLT_MAX;
	return val * val;
	}
		double square(void) {
		float val = FLT_MAX;
		return val * val;
		}

Invalid use of standard library floating point routine - Wrong arguments to standard library function	double arccosine(void) {
	double degree = 5.0;
	return acos(degree);
	}
		double arccosine(void) {
		double degree = 5.0;
		double radian = degree*180/(3.14159);
		return acos(radian);
		}

Float division by zero - Dividing floating point number by zero	float fraction(float num){
	float denom = 0.0;
	float result = 0.0;
	result = num/denom;
	return result;}
		float fraction(float num){
		float denom = 0.0;
		float result = 0.0;
		if(((int)denom) != 0)
		result = num/denom;
		return result;}

Integer conversion overflow - Overflow when converting between integer types	char convert(void) {
	int num = 1000000;
	return (char)num;
	}
		long convert(void) {
		int num = 1000000;
		return (long)num;
		}

Integer overflow - Overflow from operation between integers	int plusplus(void) {
	int var = INT_MAX;
	var++;
	return var;
	}
		long plusplus(void) {
		long lvar = INT_MAX;
		lvar++;
		return lvar;
		}

Invalid use of standard library integer routine - Wrong arguments to standard library function	int absoluteValue(void) {
	int neg = INT_MIN;
	return abs(neg);
		int absoluteValue(void) {
		int neg = INT_MIN+1;
		return abs(neg);

	}	}

Integer division by zero - Dividing integer number by zero	int fraction(int num){
	    int denom = 0;
	    int result = 0;
	    result = num/denom;
	    return result;
	}
		int fraction(int num){
		    int denom = 0;
		    int result = 0;
		    if (denom != 0)
		        result = num/denom;
		    return result;
		}

Shift of a negative value - Shift operator on negative value	int shifting(int val){
	    int res = -1;
	    return res << val;
	}
		int shifting(int val){
		    unsigned int res = -1;
		    return res << val
		}

Shift operation overflow- Overflow from shifting operation	int left_shift(void) {
	    int foo = 33;
	    return 1 << foo;
	}
		long left_shift(void) {
		    int foo = 33;
		    return 1 << foo;
		}

Sign change integer conversion overflow - Overflow when converting between signed and unsigned integers	char sign_change(void) {
	    unsigned char count = 255;
	    return (char)count;
	}
		int sign_change(void) {
		    unsigned char count = 255;
		    return (int)count;
		}

Unsigned integer conversion overflow - Overflow when converting between unsigned integer types	unsigned char convert(void) {
	    unsigned int unum = 1000000U;
	    return (unsigned char)unum;
	}
		unsigned long convert(void) {
		    unsigned int unum = 1000000U;
		    return (unsigned long)unum;
		}

Unsigned integer overflow - Overflow from operation between unsigned integers	unsigned int plusplus(void) {
	    unsigned uvar = UINT_MAX;
	    uvar++;
	    return uvar;
	}
		unsigned long plusplus(void) {
		    unsigned uvar = UINT_MAX;
		    unsigned long ulvar = uvar++;
		    return ulvar;
		}

Programming Defects [8]		

Invalid use of == (equality) operator - Equality operation in assignment statement	for (j == 5; j < 9; j++) {
	    array[i] = j;
	    i++;
	}
		for (j = 5; j < 9; j++) {
		    array[i] = j;
		    i++;
		}

Invalid use of = (assignment) operator - Assignment in control statement	if(alpha = beta){
	    printf("Equal\n");
	}
		if(alpha == beta){
		    printf("Equal\n");
		}

Invalid use of floating point operation - Imprecise comparison of floating point variables	float flt = 1.0;
	if (flt == 1.1)
	    return flt;
	return 0;
		float flt = 1.0;
		if (fabs(flt-1.1) < Epilson)
		    return flt;
		return 0;

Dead code - Code cannot be reached along any execution path	int table[5];/* Create a table */
	for(int i=0;i<=4;i++)
	  table[i]=i^2+i+1;
	if(table[ch]>100) return 0;
	/*Defect: Condition always false */
	return table[ch];}
		int table[5];
		/* Create a table */
		for(int i=0;i<=4;i++)
		  table[i]=i^2+i+1;
		/* Fix: Remove dead code */
		return table[ch];
		}

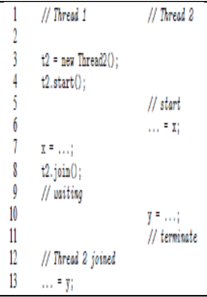
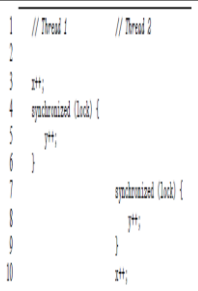
Non-initialized variable - Variable not initialized before use	int command;
	int val;
	command = getsensor();
	if (command == 2)
	  {
	    val = getsensor();
	  }
		int command;
		/* Fix: Initialize val */
		int val=0;
		command = getsensor();
		if (command == 2)
		  {
		    val = getsensor();
		  }

	return val;	return val;

Uncalled function - Function with static scope never called in file	static int Initialize(void) {…
	}
	void main()
	{
	int num;
	num=0;
	printf("The value of num is %d",num);
	}
		void main()
		{
		int num;
		/* Fix: Call static function Initialize */
		num=Initialize();
		printf("The value of num is %d",num);
		}

Variable shadowing - Variable hides another variable of same name with nested scope	int fact[5]={1,2,6,24,120};
	int factorial(int n){
	int fact=1;
	/*Defect: Local variable hides global array with same name */
	return(fact);
	}
		int fact[5]={1,2,6,24,120};
		int factorial(int n){
		/* Fix: Change name of local variable */
		int f=1;
		for(int i=1;i<=n;i++)
		f*=i;
		return(f);
		}

Defects in Multi-Threaded [8]		

Data race - A data race is a situation where events from different threads execute without ordering and read and write the same data. Data races can lead to data inconsistency and unintended nondeterminism.	
	// Thread 1 // Thread 2
	
	t2 = new Thread2();
	t2.start();
	// start
	… = x;
	x = …;
	t2.join();
	// waiting
	y = …;
	// terminate
	// Thread 2 joined
	… = y;
		
		// Thread 1 // Thread 2
		
		x++;
		synchronized (lock) {
		y++;
		}
		synchronized (lock) {
		y++;
		}
		x++;

violation of atomicity - A violation of atomicity occurs if a sequence of shared data access of one thread is interleaved with access to the same data from other threads.	class Stack {
	int top;
	Object[] arr;
	int size() {
	return top;
	}
	void push(Object o) {
	// assert (top < arr.length);
	arr[top++] = o;
	}
	Object pop() {
	// assert(top > 0);
	return arr[--top];
	}
	}
		class Stack {
		int top;
		Object[] arr;
		int size() {
		return top;
		}
		synchronized void push(Object o) {
		// assert (top < arr.length);
		arr[top++] = o;
		}
		synchronized Object pop() {
		// assert(top > 0);
		return arr[--top];
		}
		}

Deadlock - A deadlock situation occurs at runtime if threads use synchronization so that a cyclicwait condition arises.	deadlock situation occurs at runtime if threads use synchronization so that a mutual wait condition arises	Detected should be detected and corrected

ACKNOWLEDGMENT

In preparation of this paper, I had to take help and guidance of respected persons, who deserve my deepest gratitude. I would like show my gratitude to Mr. Rabindranath Raul from Honeywell who provided the technical support also helped in IP review. In addition, I thank Dr. R. Selvarani, who guided me throughout paper preparation.

X. CONCLUSION

Currently system engineers use different design tools like UML, SYSML, MATLAB, SCADE etc to convert system requirements to architectural design diagrams. Design verification to detect the Design Defects and rectify them at design level and software errors at implementation phase is very impartment to control the flow of defects to the subsequent process steps of SLDC, making design and software more robust. Fixing design defects will make design more maintainable and reduces significant maintenance cost and software errors will reduce the unexpected behavior of the software and those reduce the defects identified at the testing phase, those reducing the cycle time, which intern reduces the manual hours.

REFERENCES

[1]. IVAN POLÁŠEK : Anti-pattern Detection as a Knowledge Utilization

[2] W. Brown, R. Malveau, H. McCormick, and T. Mowbray, AntiPatterns:Refactoring software, architectures, and projects in crisis. John Wiley & Sons, 1998.

[3] Naouel Moha1, Amine Mohamed Rouane Hacene2, Petko Valtchev3, and Yann-Ga¨el Gu´eh´eneuc1 Refactorings of Design Defects using Relational Concept Analysis

[4] Roger Lee Studies in Computational Intelligence,Volume 364

[5] R. Selvarani. S. Aishwarya. “A Formal Model indicating Inter-relationship of CK Metrics in view of Defect Proneness". Accepted for publication in the International Journal of computers and Technologies (IJCT). August 2013

[6] Maria Teresa Llano† and Rob Pooley UML Specification and Correction of Object-Oriented Anti-patterns.

[7]. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: Antipatterns: Refactorin Software, Architectures, and Projects in Crisis, 1st edn. John Wily and Sons,West Sussex (1998)

[8]. Matlab defects web page http://www.mathworks.in/help/bugfinder/ref/floatconversionoverflow.html

[9] Christoph von Praun, Dipl.-Inf. TU-M¨unchen
Detecting Synchronization Defects inMulti-Threaded Object-Oriented Programs

[10] R. Selvarani.Areej, R.Bharathi “Software Reliability Estimation at Design Stage based on multifunctional estimation Technique” submitted to ACM TOSEM, August 2013

[11] R. Selvarani. S. Aishwarya , “A frame work for codifying the association of CBO and NOC in the observation of Defect proneness” submitted to Empirical software engineering journal, August 2013

[12] Smith, C.U. and L.G. Williams, Software performance antipatterns, in Proceedings of the
2nd international workshop on Software and performance. 2000, ACM: Ottawa, Ontario,
Canada. p. 127-136.

[13] Lanza, M. and R. Marinescu, Object-Oriented Metrics in Practice. 2005: Springer-Verlag
New York, Inc.

[14] Gaffney, J. E.: Metrics in software quality assurance, in *Proc. of the ACM '81 Conference*, ACM, 126-130, 1981.

[15]. A. J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[16] Roger S. Pressman. *Software Engineering { A Prac- titioner's Approach*. McGraw-Hill Higher Education, 5*th* edition, November 2001. isbn: 0-07-249668-1.

[17] Robert C. Martin, first five object-oriented design(OOD) principles, https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

[18] R. Selvarani. T.R.G. Nair. "Defect Proneness Estimation and Feedback Approach for Software Design Quality Improvement" ELSEVIER, Information and Software Technology, volume 54 issue 3, 274 – 285 (2012)

[19] OMG, *OMG Unified Modeling Language (OMG UML), Infrastructure. V.2.2*. Object Management Group, 2009.

[20] N. Moha and Y. Gu´eh´eneuc, "On the automatic detection and correction of software architectural defects in object oriented designs," *ECOOP Workshop on Object-Oriented OUTReengineering*, 2005.