AN OVERVIEW OF OBJECT ORIENTED DESIGN HEURISTICS

Master Thesis

Department of Computer Science, Umeå University, Sweden

January 27, 2005

Author: Muhammad K. Bashar Molla

Email: ens03mbr@cs.umu.se

Supervisor: Jürgen Börstler

Email: jubo@cs.umu.se

ACKNOWLEDGMENT

I would like to thank my supervisor Dr. Jürgen Börstler for his help, support, valuable idea and suggestion, which helped me to make this study. Without him, this study would never exist. I would like to thank Dr. Colin A Higgins to give me TOAD heuristics documents. Thanks my uncle Dr. M. K. Islam to encourage me to study in Sweden. Thanks Per Lindstöm to give me admission, and special thanks to my parents for financial and mental support**.**

ABSTRACT

This study focuses on heuristics for object oriented design. We compare design heuristics to related concepts like for example design methods and design patterns. Design heuristics are rules of thumb to support the object oriented design process.

At the beginning level of design, novice designers need guidelines. Expert object oriented designers can guide the novice designers by means of their experience. However, very few designers have published their experience formally as design guidelines and heuristics.

We describe, compare and categorise different heuristics proposed in the literature to make them more accessible to novice designers. We conclude that heuristics are easier to understand and remember as for example design patterns. Designers can use their judgment and evaluate the justification of heuristics to support their design.

Table of Contents

1 Introduction 8
2 Object Oriented Concepts 9
2.1 Introduction 9
2.2 Object 9
2.3 Class 10
2.4 Inheritance 10
2.5 Polymorphism 11
2.6 Data Abstraction and Information Hiding 12
2.7 Encapsulation 13
2.8 Aggregation 13
3 Support for OOD 15
3.1 Introduction 15
3.2 OOD Methods 16
3.2.1 BOOCH Method 17
3.2.2 OMT Method 18
3.2.3 Coad-Yourdon Method 21
3.3 OO Principles 23
3.3.1 Martin's Collection of OO Principles 23
3.3.2 The Law of Demeter 25
3.4 Design Patterns 26
3.5 OO Heuristics 27
3.6 Summary 27
4 Existing Heuristics Design Models 29
4.1 Introduction 29
4.2 Riel's Heuristics 29
4.3 TOAD Heuristics 34
4.4 MeTHOOD Heuristics 36
4.5 Meyers Heuristics 38
4.6 Summary 39
5 Evaluation of heuristics 40
5.1 Relationship between Patterns and Heuristics 40
5.2 Comparison of heuristics 43
5.3 Conflicting Heuristics 49
5.4 Summary 50
6 Discussion 51
7 References 52
8 Appendix 54

1 Introduction

The concept of object oriented design is gaining more popularity and there is a constant increase in the shift towards it, when compared to traditional design. Object Oriented Design (OOD) is a method for decomposing software architectures. It has the advantages of faster development, easier maintenance for the designers and increased quality for the users. A designer runs a piece of design against a list of design heuristics that were build up through many years of design experience [2].

Heuristics are simple rules of thumb and characteristically without supporting observed verification, how can designers know that they represent good guidance?

Many designers have their own design heuristics. They use their own experiences. These experiences are their heuristics, and they apply them as needed. They rarely collect those heuristics in an organized way. The main goal of this study is to give an overview over published design heuristics and categorize them in a way that makes them accessible to others.

Most of the heuristics found in Riel, MeTHOOD, TOAD, and Meyers literature. The object of this study is to provide fundamental survey of those heuristics in the context of object orientation and provide a guideline for the novice designers. To understand design heuristics, this study is organized in the following way.

- Chapter 2 discusses the basic concepts of object oriented design which consists of objects, classes, and their relationships, and other object oriented principles.
- Chapter 3 describes design methods; design principles; design patterns and design heuristic that assist development of object oriented design.
- Chapter 4 shows the heuristics from different views from different authors, how they proposed their heuristics, what are the areas covers their heuristics.
- Chapter 5 discusses relationship between heuristics and patterns, similar heuristics, and conflicting heuristics.
- Chapter 6 discusses the summary of this study.

2 Object Oriented Concepts

This section introduced the concept of object oriented terminology which is used in every object oriented design.

2.1 Introduction

The basic element in an object oriented system is an object. Object oriented development is a method of implementation in which, programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. In such programs, classes are viewed as static whereas objects typically have a much more dynamic nature.

Object orientation is a technique for software system. These techniques lead to design architectures based on objects that are manipulated in every system. OO design systems are better modeled domain systems than similar systems created by structure systems. It offers a number of concepts, which are well suited for this purpose. By understanding these object oriented concepts; designers will learn how to apply those concepts to all stages of the software development life cycle. In the following subsections, We will introduce the basic concepts within object oriented environment.

2.2 Object

The term object was first formally applied in the Simula language, and objects typically existed in Simula programs to simulate some aspect of reality [5]. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.

> **Definition: "***An object is characterized by a number of operations and state which remembers the effect of these operations"* [21].
> **Definition:** "*An object is a concept, abstraction or thing with crisp boundaries and meaning for the problem at hand"* [33].

All information in object oriented system is stored within its objects. An object can appear as a text, an icon or both and those information can be manipulated when the objects are ordered to perform operation.

2.3 Class

In object oriented designing or programming, an actual understanding of a class that consists of data and the operations related with that data are important. It is an item that a user can manipulate as a single unit to perform a task.

> **Definition:** "*A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their information structure"* [21].

A class represents a set of objects that share a common structure and a common behavior. In an OO environment, a class is a specification of instance variables, methods, and inheritance for objects. Once a class is defined, any number of objects can be created which belong to that class i.e. class is everything about objects where objects are individual instance of a class.

2.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of the objects of another class. In OO design, the concept of inheritance supports the idea of reusability.

> **Definition:** "*Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship"* [33].

By inheritance, it is possible to add new features to an existing class without modifying the previous class, so this is the way to derive a new class from an existing one. The new class is called a subclass or a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations [13].

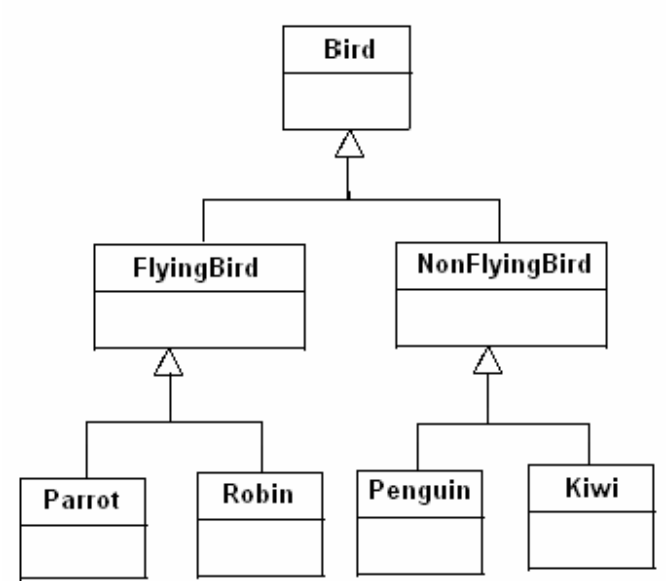

Figure 1: Inheritance: relationship between subclass and superclass.

Consider the figure 1, according to object oriented concepts, FlyingBird and NonFlyingBird classes are all subclass of Bird class. Similarly, the Bird class is the superclass of FlyingBird and NonFlyingBird class. On the other hand, the bird Parrot is a part of the class FlyingBird that is again the part of the class Bird. It means that, each derived class share common characteristics with the class from which it is derived. Therefore, inheritance is a usual way to model the system of discourse, and provides an accepted model for object oriented analysis and design. Through inheritance, designers can eliminate redundant code and extend the use of existing class that means inheritance also provides code and structural reuse.

2.5 Polymorphism

Polymorphism means the ability to take more than one form. Through polymorphism, it is possible to hide many implementations behind the same interface. Polymorphism is a concept in type theory, according to which a name may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways [5].

Definition: *"Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realizations in different classes"* [25].

Definition: *"Polymorphism means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class"* [21].

Polymorphism plays an important role in allowing objects to have different internal structures but share the same external interface. This means that a general class of operations can be accessed in the same manner, even though specific actions associated with each operation may differ. For example, consider the following figure 2, a draw operation on a circle object does not result in the same behavior as a box object or triangle object.

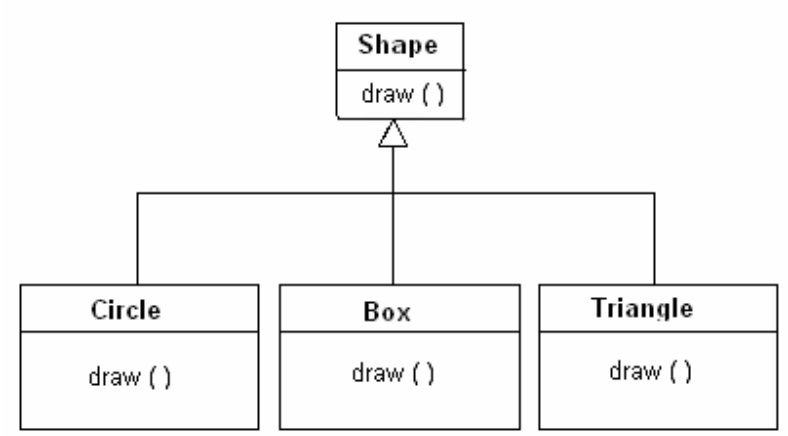

Figure 2: Polymorphism shows same function but different behavior.

By using polymorphism, a designer can build up a library of reusable classes and contributes their skill to develop reusable object oriented design.

2.6 Data Abstraction and Information Hiding

Data abstraction means, to represent the essential feature without including the background explanation. It is a form of abstraction where the details of the important algorithms and necessary data are hidden. Data abstraction allows objects to be treated as black boxes.

Information hiding is a way that, an object can communicates with other object in their public interface. The object can maintain private information and methods that can be changed at any time without affecting the other object. For example, the users of a microwave oven, they change the temperature whatever they want and it works. They don't need to understand the mechanism of the microwave oven.

2.7 Encapsulation

Encapsulation is the first principle in object oriented approach. The idea of encapsulation comes form abstract data types. Encapsulation is a mechanism to realize data abstraction and information hiding. It hides detailed internal specification of an object, and publishes only its external interfaces. Thus, users of an object only need to hold on to these interfaces. By encapsulation, the internal data and methods of an object can be changed without changing the way of using the object. By hiding a representation and implementation in an object, more reusable specialized classes can be created. The representation cannot be accessed and is not visible directly from the object. Operations are the only way to access and in modify an object's representation [13].

> **Definition:** "*Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics"* [5].

Encapsulation is the most remarkable feature of a class. The data is not accessible to the outside world, only those functions, which are wrapped in the class, can access it. Coad et al. mention encapsulation is a principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision [9]. In object oriented approach, by using encapsulation a designer makes design easier, less annoying, more sustainable, and more efficiently workable.

2.8 Aggregation

Aggregation is a type of relationship between objects. Objects are organized into an aggregation structure that shows how one object is composed of a number of other objects. In aggregation, host object acts as a relationship between the outside world and an inner object. It could be done in several ways. One is by having both objects

implemented on the same interface. Another technique allows the method of the inner component to be directly exposed to the outside world. The most significant property of aggregation is transitivity, which means, if X is a part of Y and Y is a part of Z, then X is a part of Z. Aggregation is also antisymmetric, which means, if X is a part of Y, then Y is a part of X [33].

Figure 3: Aggregation

> **Definition:** *"An aggregate is a union of several objects, and the union as such is often represented by an object its own"* [21].

Aggregation is a kind of association. This association used to model whole part relationship between things, which is called composite. Composite aggregation states, the multiplicity at the composite end may be at most one, and it is identified by small diamond[1]. For example, figure 3 shows a portion of an object oriented design for a human program. A body consists of hands, each of which consists of fingers. In object oriented design, association establishes relationship among object and class. It also describes a group of relationships with common structure and common semantics. Multiplicity specifies how many instances of one class may relate to each instance of another class. Aggregation is commonly encountered in bill-of-material or parts explosion problem [33].

[1] Larman, C.: Applying UML and Pattern, An Introduction to Object Oriented Analysis and Design, Prentic Hall, Inc., 1998

3 Support for OOD

This section briefly describes the important characteristics of different object oriented design methods, design principles; design patterns in the context of object oriented design heuristics.

3.1 Introduction

OO technology is a way of thinking abstractly about a problem using real world concepts. OOD promotes better understanding technique, cleaner design, and more maintainable systems.

> **Definition:** *"Object oriented design is the construction of software system as structured collection of abstract data type implementations"* [25].

The goal of object oriented design is developing an object oriented model and implementing the requirements identified. To support OOD, there are many design methods, design principles, design patterns and design heuristics presented to fulfil their requirements. This section mainly focuses on these supporting tools of OOD.

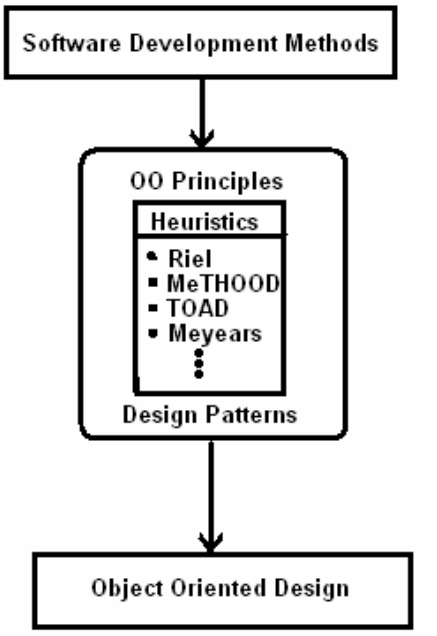

Fig 4: Representation of design heuristics

The design methods provide a set of techniques for analyzing, decomposing, and modularizing software system architectures. Design principles are applied for decomposing software into components. Design patterns provide high-level reusability of design where heuristics provide design guidelines.

Figure 4 represents the overview of support for applying in object oriented design. According to the figure, final design is obtained by using heuristics mentioned above, those are based on object oriented principles and design patterns.

3.2 OOD Methods

Many methodologies have been proposed for OO software design. For example Berard [12], Business Object Notation (BON) [36], Booch [5], Hierarchical Object-Oriented Design (HOOD) [31], Object Modeling Technique (OMT) [33], Coad-Yourdon Method [9], Class-Responsibility Collaborator (CRC) [7], The Fusion Method [0], Jacobson Method [21], Moses [18], Object Behaviour Analysis (OBA) [32], Object-oriented Process, Environment and Notation (OPEN) [19], Shlaer and Mellor Method [34].

These methods represent a collection of techniques, formalised ideas and heuristics that can be used when developing object oriented design. It contains good design guidelines and it provides specific guidelines for the construction of reusable components. For example, Hierarchical Object-Oriented Design (HOOD) has been developed by the European Space Agency as a design/notation method for ADA. A basic design step has its goal, the identification of child objects of a given parent object, and of their individual relationships to other existing objects, or the refinement of a terminal object to the level of the code. This process is based on the identification of objects by means of object oriented design techniques and the purpose of HOOD is to develop the design as a set of objects, which together provide functionality to the program [31]. HOOD focuses on the hierarchical level, OMT model focuses on the database level, and Booch focuses on the design and structure level. The following subsection is a brief discussion of BOOCH, OMT and Coad-Yourdon methods.

3.2.1 BOOCH Method

The Booch method covers the analysis and design phases of an object oriented system implementation. He defines many symbols to document almost every design decision. In his method, he shows designers what they can do in terms of system design. This method emphasises the distinction between a logical view of a system, in terms of classes and objects, and a physical view of a system in terms of modules and processes. This method also makes a distinction between static and dynamic model of a system. From those view of design, he comes up with a notation that includes six diagrams. These diagrams are discussed below.

Class Diagram: A class diagram is used to show the existence of class and their relationship in the logical design of a system. In the design phase, designers use class diagrams to capture the structure of the classes that form systems architecture. A single class diagram represents a view of the class structure of a system.

Object Diagrams: An object diagram is used to show the existence and their relationships in the logical design of a system. Object diagram shows only a snapshot of the object at a given period. A single object diagram is typically used to represent a scenario.

Module Diagram: Module diagrams show the allocation of classes, class utilities, object and other decelerations of the physical module. A single module diagram represents a view of the module architecture of a system. Through these diagrams, it is possible to understand the general physical architecture of a system.

Process Diagram: A process diagram is used to show the allocation of processes to processors in the physical design of a system. A single process diagram represents a view of the process architecture of a system.

State Transition Diagram: A state transition diagram is used to show the state space of an instance of a given class, the events that causes a transition from one state to another, and the actions that result from the change the change of state.

Interaction Diagram: An interaction diagram is used to trace the execution of a scenario in the same context as an object diagram.

The Booch method is one of the well known object oriented methods. It provides a sound understanding of the fundamental concepts of the object model. This method offer a path from requirements to implementation, using object oriented analysis and design. As a guideline, the method suggests three steps can be used for analyzing a system in preparation for designing a solution in an object-oriented manner. At first, define the problem; second, develop an informal strategy for the software realization of the real world problem domain; and last, formalized the strategy. To formalize the strategy, Booch suggests the following order of actions:

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Implement these classes and objects.

Booch defines a lot of notations to document almost every design decision. These notations can be used in the early stage of design.

3.2.2 OMT Method

Rumbaugh et al. describe Object Modeling Technique (OMT) [33]. In his method, he shows object orientation as a way to organize software as a collection of discrete objects that include both data structure and behaviour. This method leads to three different model of the system corresponding to three different views. Those models are: Object model, dynamic model and functional model.

- Object model describes the object to which something happens. The technique of object model is to describe the static structure of objects in a system and their relationship.

- The dynamic model describes when control flow will happened. The technique of dynamic model is to describe the control flow of a system and it also describe the aspects of the system that change over time.

- Functional model describe which process computation will be happen. The technique of functional model is to describe the computation within a system.

OMT represent a methodology for object oriented development and a graphical notation for representing object oriented concepts. This methodology is divided into the following four phases which shows analysis and design process.

1. **Analysis:** This phase shows the building of a model of real world situation starting with a problem statement. The purpose of analysis is to state and understand the problem and the application domain so that a correct design can be constructed.

2. **System design:** This phase mention the designs of the overall architecture of the system. After analyzed a problem, designer must decide how to move towards the design. System design is the high level strategy for solving the problem and building a solution.

3. **Object design:** In this phase, object design construction of a design, based on analysis model with implementation details which are added to the object. The focus of object design is to determine the full definition of the classes and associations used in the implementation.

4. **Implementation:** In implementation phase, the object class and relationship developed during object design are finally translated into a particular programming language. That language influences design decisions to some point i.e. traceability to the design is straight-forward and that reason the implemented system remains flexible and extensible.

In this method OMT notations are used for describing the structure of a system, which gives a model of the class structure specified in the object oriented design. Object oriented concepts can be applied throughout the system development life cycle, from analysis through design to implementation [33].

This model describes object oriented design guidelines (see Appendix E), that can be used to analyze problem requirements, design a solution to the problem, and then implement the solution in a programming language or database. These guidelines focus on reusability, extensibility, robustness categories. The following discussion shows those categories.

Reusability

Reusability is the process to which a software module or component can be used in more than one computing program or software system. The goal of reusability guidelines is to reduce designing, coding, and testing, so that the total system will be understandable and ensures faster development of software applications. According to OMT model, reusability can be divided into two groups: Style Rules for reusability, which is the way of sharing the newly written code within a design and using inheritance, which is the way of reusing previously written code on new design. In reliability group, OMT describes 12 guidelines (see Appendix E). These guidelines are focus on design methods and using inheritance.

Extensibility

Extensibility mentions the ability to add or change a function or data without changing the existing functions or data, so that the design will be free from unwanted side effects. OMT model claims that most of the software is extended in a way such that its original developers may not expect. In extensibility category, OMT describes 5 guidelines (see Appendix E). These guidelines focus on encapsulation, information hiding and operation.

Robustness

Robustness is the stability of software applications in extreme situations that is, maximum load conditions. OMT model mentions that a method is robust if it does not fail even, if it receives improper parameters. In extensibility category, OMT describes 5 guidelines (see Appendix E). These guidelines focus on errors, arguments, debugging and monitoring.

As a design guidelines OMT method attempts to show how to use object oriented concepts throughout the entire object oriented design lifecycle. Although this method is complex in different phases and different views, the concept of using those phases and views are very powerful.

3.2.3 Coad-Yourdon Method

Coad-Yourdon proposed Object Oriented Analysis (OOA) method, this method is a step by step method for developing object oriented models. OOA uses basic principles and merge them with object oriented point of view. Those principles are:

1. Abstraction
2. Encapsulation
3. Inheritance
4. Association
5. Communication with message
6. Pervading method of organization
7. Scale
8. Categories of behaviour

This method consists of five steps. These are:

Finding Class and Objects: This step specifies how classes and objects should be found.

Identifying Structure: This step is done in two different ways, Generalisation-specialisation structure and Whole-Part structure. Generalisation-specialisation captures the hierarchy among the identified classes. Whole-Part structure is used to model how an object is part of another object.

Identifying Subject: This step is done by partitioning the class and object model into larger units. Group of class and object is known as subject. The size of each subject is selected to help a reader to understand the system through the model.

Defining Attributes: This step is done by identifying information and the associations that should be related with each instance.

Defining Services: This step is done by identifying the object states and defining services, for example, create, access, connect etc.

According to these steps, an OOA model that is built during Analysis consists of five layers:

- Subject layer
- Class & Object layer
- Structure layer (i.e. inheritance, relationships)
- Attribute layer
- Service layer

Identifying the elements of each layer constitutes an activity within the method. An OOD model consists of the following four components:

- **Problem Domain Component:** The results of the object oriented analysis are put directly into this layer. Classes that deal with the problem domain; for example, Customer classes and Order classes
- **Human Interaction Component:** This component involves such activities as: classifying human users, describing task scenarios, designing the command hierarchy, designing the detailed interaction, prototyping the Human Computer Interface, user-interface classes such as window classes.
- **Task Management Component**: This component consists of identifying tasks, the services they provide, the task priority, whether the process is event or clock driven, and how it communicates.
- **Data Management Component**: This component depends greatly on the storage technology available, and the persistence of data required.

In this method, OOA is an analysis model is developed to describe the functionality of the system and OOA is essentially an object oriented approach and concepts, for example class, instance, inheritance, and encapsulation are here essential elements. The effect of the OOA method is documented in a special graphical notation and special guidelines for the textual documentation of classes and objects [9].

According to design guideline, this method is one of the easiest object oriented method to learn and implement, although it focuses on analysis of business problems. By following above mention five steps, designers can implements their design. This method is designed for a good introduction to object orientation, even though now days it is being used even for large systems.

3.3 OO Principles

Object oriented principles are derived from the combination of theory based knowledge, experience, common sense and tend to be written in a prescriptive manner. These principles suggest the designers what to support and what to avoid. When design principles are used in practice, they are commonly referred to as heuristics [28]. These heuristics are based on object oriented principles as shown in Fig 1.

3.3.1 Martin's Collection of OO Principles

Martin states object oriented design is replaced with principles and techniques for building dependency firewalls and for managing module dependencies [30]. He discusses the following popular principles in software design.

- The Open/Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Dependency Inversion Principle (DIP)
- The Common Closure Principle (CCP)
- The Interface Segregation Principle (ISP)
- The Reuse/Release Equivalency Principle (REP)
- The Common Reuse Principle (CRP)
- The Acyclic Dependencies Principle (ADP)

The Open/Closed Principle (OCP): Meyer [25] proposes this principle. The main idea of this principle is software entities for example; class module, function, etc. should be open for extension but close for modification. Open for extension means, designer can make their design such a way that, requirements of application change or needs new feature to fulfill the design. On the other hand, close for modification means, designer can allowed to make changes to it, for example, add new feature by extending existing design. By using data abstraction, OCP principle is used in object oriented design.

The Liskov Substitution Principle (LSP): Barbara Liskov [3] proposes this principle. LSP mention, programs that use pointers or references to base classes must be able to use

objects of derived classes without knowing it [3]. This principle mentions checkable design heuristics, which is similar to the heuristics in the MeTHOOD heuristics.

The Common Closure Principle (CCP): CCP define the classes in a package should be close together against the same kind of changes. A change that affects a closed package affects all the classes in that package and no other package.

The Dependency Inversion Principle (DIP): This principle deals with the overall structure of a well designed object oriented application. “Depend on abstraction” is the main theme of this principle and it describes not to depend on a concrete class – that, all relationships in a program should terminate on an abstract class or an interface [30]. High-level classes should not depend on low level classes. The principle of dependency inversion is the fundamental low-level mechanism behind many benefits that are claimed for object oriented technology. So, proper application is necessary for the creation of reusable framework [30].

The Interface Segregation Principle (ISP): The Interface Segregation Principle state “The dependency of one class to another one should depend on the smallest possible interface” [25]. A class with several different functions can create separate interfaces for each function. The function of the class can be organized as a group of related methods. “Clients should only have to depend on methods that they actually call. This can be achieved by breaking the interface of the fat class into many client-specific interfaces” [30]. Fat interface lead to inadvertent coupling and accidental dependencies between classes.

The Reuse/Release Equivalency Principle (REP): Martin [30] states the granule of reuse is the granule of release, only components that are released through a tracking system can be effectively reused, this granule is the package. This principle also describes that the granule of reuse can be no smaller than the granule of release.

The Common Reuse Principle (CRP): This principle discussed about those classes that change together, belong together. It is a consequence of the Reuse Release Equivalence Principle which states that the clients of a class have to compile, link, and/or test dependencies on an entire package instead of just the classes it directly uses [29].
The Acyclic Dependencies Principle (ADP): This principle mentions that “The dependency structure between packages must not contain cyclic dependencies” [23] i.e.

the dependency structure between packages must be a directed acyclic graph that is not containing cyclic dependencies. A single cyclic dependency that gets out of control can make the dependency list very long.

3.3.2 The Law of Demeter

The Law of Demeter (LoD) is a simple rule for designing object-oriented systems as well as conventional structure programming language. One useful guideline in choosing the relationship among objects is called the Law of Demeter [6]. LoD attempts to minimize the coupling between modules of a given design. For example, if module M2, M3, M4 and M5 are written is a way in which they all rely on the structure of module M1. Then, if the structure of module M1 changes, it is possible that code in module M2, M3, M4 and M5 may have to be modified in order for the design to continue to work properly.

> **LoD: "***The methods of a class should not depend in a way on the structure of any classes, except the immediate (top level) structure of their own class. Further, each method should send message to objects belonging to a very limited set of classes only*[2]*"*.

The Law of Demeter focuses on the control access information. All class relationship must be explicit from the design. Since memory only can keep a limited set of items; it is easier to keep them in memory if they are closely related.

Benefits of LoD: Coupling control, i.e. reduce data coupling; Information hiding, i.e. prevents a method from directly retrieving a subpart of an object; Information restriction, i.e. restricts the use of methods that provides information; Few interface, i.e. restricts the class that can be used in a method; Small interface, i.e. restricts the amount of information passed in a method; and Explicit interface, i.e. explicitly states which classes can be used in a method are the benefits of Law of Demeter[3].

[2] As discuss in [6], page 140, Ref. 56, Sakkinen, M. 1998,Comments on "the Law of Demeter" and C++. SIGPLAN Notice, vol. 23(12), p 12.
[3] Advanced OO, Law of Demeter:
http://www.eli.sdsu.edu/courses/spring97/cs696/notes/demeter/demeter.html

3.4 Design Patterns

The architect Christopher Alexander introduced the term design pattern in the 1960s. Design patterns are “standard” solutions to common problems in object oriented design. Design patterns solve specific design problems and make object oriented systems more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. Each pattern has the following essential elements [13]:

Pattern Name

Each pattern has a unique name that helps to identifying and referring to it. Naming a pattern immediately increase the design vocabulary. The pattern classified according to a classification such as the one described earlier. This classification helps in identifying the use of the pattern.

Problem

The problem explains when to apply the pattern. It also describes the problem and its context that is a description of the relevant forces and constraints, and how they interact. In many cases, entries focus almost entirely on problem constraints that a reader has probably never thought. Sometime the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

Solution

The Solution describes the elements that make up the design, their relationship, responsibility, and collaboration. The pattern provides an abstract description of a design problem and how a general arrangement of elements solves it. Solutions also reference and relate other higher and lower level patterns.

Consequences

The consequences are the trade-offs side effect and result of applying the pattern. Since reuse is often a factor in object oriented design, the consequences of a pattern include its impact on a system’s flexibility, extensibility. Consequences may address language and implementation issues as well.

Pattern oriented design activities resist accommodation within a linear development process, and raise challenges in the construction of suitable process models that still meet costing, predictability, and control criteria [1]. Design patterns help to choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design pattern can even improve the documentation and maintains of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent [13]. The aim of design patterns is to solve a specific design problem regarding a particular context.

3.5 OO Heuristics

A heuristic is a rule of thumb and it promotes better quality of design. Heuristics represent good practices that are learned through practice. A design heuristics is a sort of guidance for making design decision.

In literature, different authors proposed different heuristics. Every heuristics has a name, which shows the essential part of the heuristics. A Heuristic has an intent, i.e. reasoning as to why the heuristics should be used; a rational, i.e. property of heuristics; consequence, i.e. what are the perceived advantages and disadvantages. Those heuristics can be found in literature [1], [15], [16], [26].

There are some common problems made by developers. Heuristics attempt to solve these problems in the form of rules and annotate potential pitfalls. They point inexperienced designers in the right direction [15].

3.6 Summary

OO design methods are mainly notations and some high-level process activities. Very few methods support the designers in getting more experience. In this sense, current methods provide poor support for quality design. This situation led to the development of design principles, design patterns, and design heuristics. Consider the following figure 5; methods in general are used at process level, whereas OOD patterns are used in specific situation for a particular design problem.

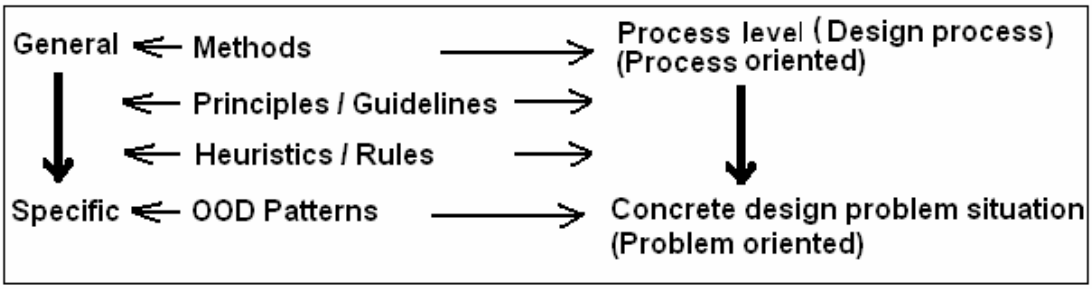

Figure 5: OO design supports from general to specific problems[4].

The goal of the design heuristics is to build a body of literature to support design and development in general. Most of the heuristics are fundamental aspects of object orientation and provide effective support for design refinement and improvement. For novice designers, it is quite hard to apply design patterns in their design, because to understand a single pattern they have to read five to eight of pages. On the other hand, heuristics are relatively easy to understand and remember, so novice designers can apply heuristics in their design. Design patterns also address people and development issues like maintainability, reusability, communicating commonality, and encapsulation variation [13]. Riel mentions that heuristics can be used to identify new design patterns in a given design [2].

[4] Discuss with Börstler J. regarding support of OO design.

4 Existing Heuristics Design Models

This section investigates the different heuristics model. Different mode provides different design guidelines.

4.1 Introduction

Several works, related to the OO Design heuristics and design patterns detection, have been presented in the last few years. Object oriented design heuristics proposed in a number of ways. According to applicable or violated features, heuristics might be differing from one to another.

The following discussion shows some popular heuristics design model by [2], [15], [16] and [26]. In this discussion, I will discuss a general overview of each heuristics model; what level of abstraction the heuristics cover and how the heuristics are applied.

4.2 Riel's Heuristics

Riel [2] catalogued 68 design heuristics. He categorized heuristics into eight different ways, they are- Class and Objects, Topologies of action-oriented versus object-oriented application, the relationship between Class and Objects, Inheritance relationship, Multiple inheritance, Association relationship, Class-specific and data behavior, and Physical Object-Oriented Design. For each and every heuristic he mentioned has a name, an outline and example of the problem, as well as he mention a suggested approach to solve the problem. The following is a summery of his heuristics.

In Class and Object chapter, Riel discusses several heuristics related with the encapsulation of data and its related behavior in a bidirectional relationship. For example, "Users of a class must be dependent on its public interface, but a class should not be dependent on its users" (see # 2.2 in Appendixes A). This heuristic focus on reusability and it states, public interface is belongs to a users of a class it means that users of a class needs public interface. On the other hand, since users of a class are not a part of public interface, so public interface cannot depend on users of a class. Consider a user as a person and public interface as an alarm clock as shown in figure 6. This heuristic states

public interface is essential for users of a class. According to this heuristic, for example, a person has to depend on an alarm clock; the alarm clock should not depend on a person. If the alarm clock depends on the person, then there will be no use to build a time clock safe without attaching a person to the safe.

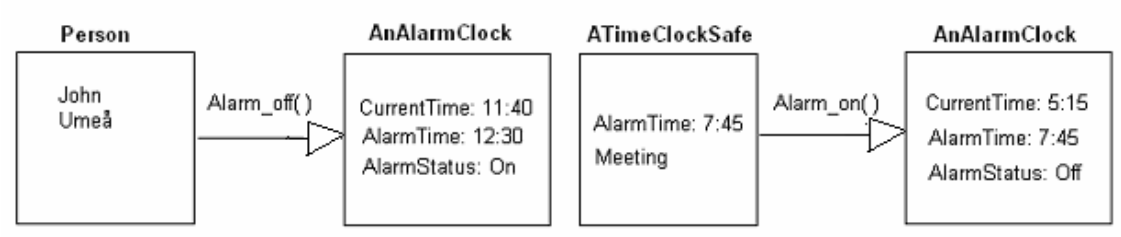

Figure 6: A user is depend on his public interface (adapted from [2]).

Topologies of action-oriented versus object-oriented application discussed a number of heuristics, which describe different topologies of these methodologies contain the kernel of truth behind object oriented development which focus on a decentralized collection of cooperating entities. For example, "Do not create god class/object in your system, Agent classes are often placed in the analysis model of an application" (see # 3.2 in Appendix A). In this category, heuristics are focus on god class problem. A god class is a class that holds everything that is not always good; such a god class problem is discussed in the following example as shown in figure 7, 8, and 9. Consider a house, which have three rooms. Each room consisting of a desired temperature input devices, an actual temperature sensor and an occupancy sensor. Consider a heat flow regulator whose main job is to sense when each room needs heat. HeatFlowRegulator act like a god class because only the HeatFlowRegulator sends message to other three classes as shown in the figure 7. Since three room objects contain an object of each sensor, the room can encapsulate the three objects as shown in figure 8. In this design, HeatFlowRegulator is performing most of the work for this system and still there is god problem. According to the heuristic (see # 3.1 in Appendix A), there will be a solution, if HeatFlowRegulator ask each room whether they need heat or room objects have to inform to HeatFlowRegulator as shown in figure 9. These heuristics are for developing optimal application topology.

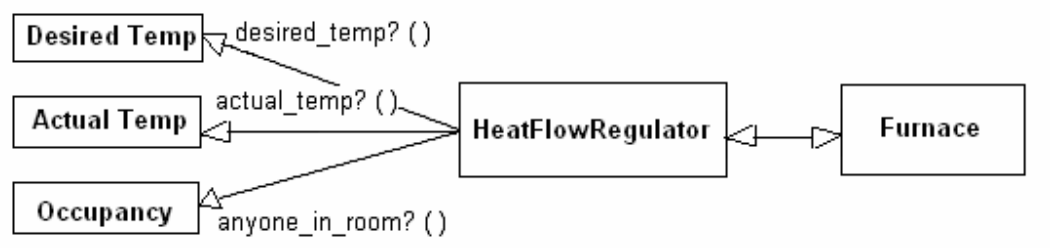

Figure 7: God class problem without encapsulation (taken from [2]).

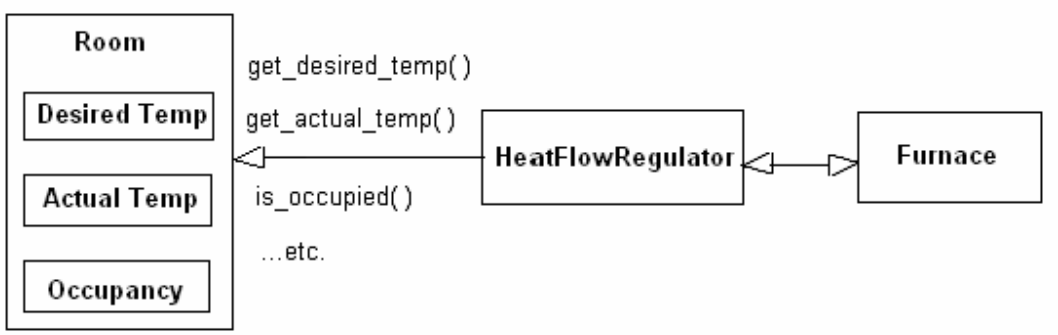

Figure 8: God class problem with encapsulation (taken from [2]).

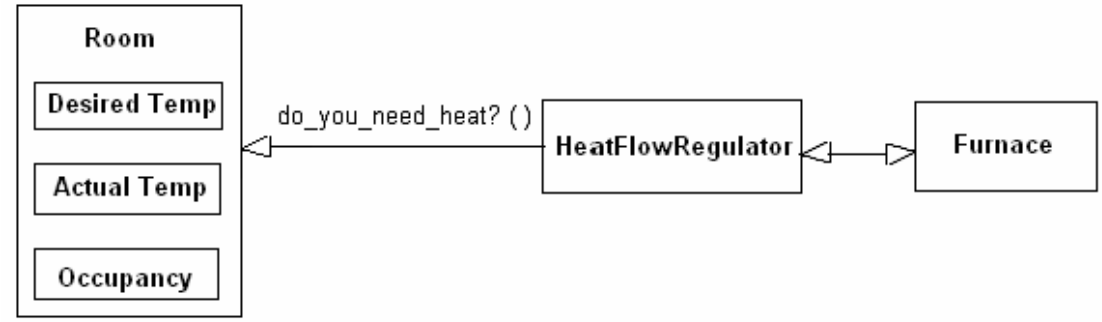

Figure 9: Removed God class problem (taken from [2]).

According to the relationship between Class and Objects, inheritance relationship, multiple inheritances, and association relationship concept, Riel mentions quite a lot of heuristics [2]. Sometimes a single violation makes major effect on the design and that effect makes difficulty in the design that is called proliferation of class problem. Several heuristics describe such information, which is vital in reducing the proliferation of class problem. For example, "Minimize the number of classes with which another class collaborates" (see # 4.1 in Appendix A). The rational behind this heuristic is, the greater the number of collaboration among various class the lesser the chances of reusing those

class. Collaboration exists between the two classes when an object of a class sends a message to an object of another class. Consider the following example, as shown in figure 10. The first figure of the example shows RestaurantPatron is collaborating with Melon, Steak and Pie objects. According to reduce the number of collaboration as states in this heuristics, meal class is indicate those foods that are not collaborating with John as shown in next figure.

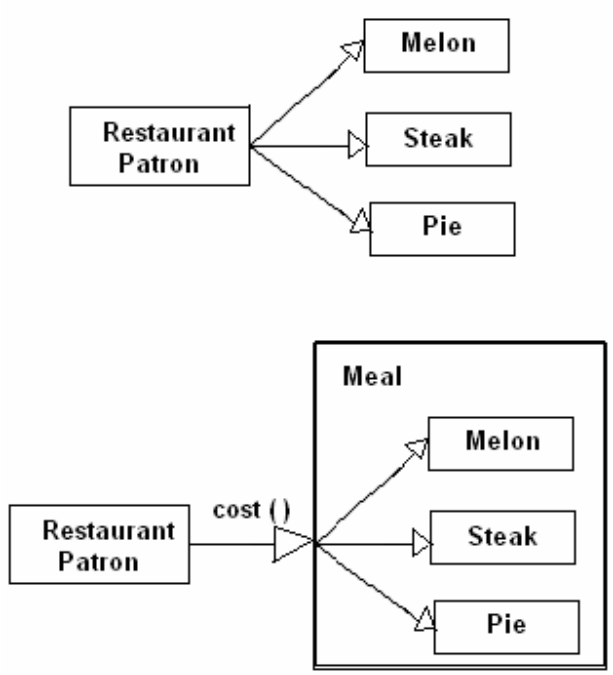

Fig 10: Finding containing classes to reduce collaboration (taken from [2]).

Inheritance relationship is one of the most important relationships in the object oriented environment. By the misuse of inheritance relationship, design can be complex. Optional components are another issue in design. Inheritance and containment by reference can be a solution for optional components. For example, "Do not confuse optional containment with the need for inheritance, modeling optional containment with inheritance will lead to a proliferation of classes" (see # 5.18 in Appendixes A). The rational behind this heuristic describe, optional component of a class are sometime incorrectly modeled with inheritance. For example, consider a house that may have different kind of systems like heating system, cooling system, electrical system, heating cooling system, etc. as shown in Fig 11. The above mentioned systems are the optional components of the building and it shows by the arrow. These optional components are poorly represented. According to this heuristics, containment could be a better solution as shown in Fig 12 and hold

optional references to objects that provides those systems (heating system, cooling system, electrical system and plumbing system).

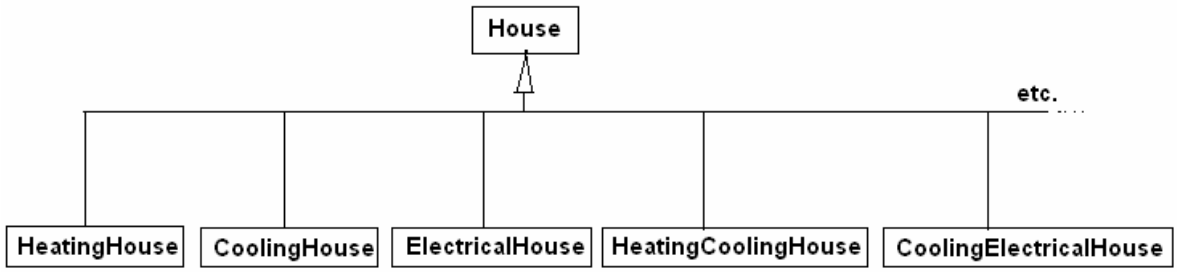

Fig 11: Class proliferation and optional components (taken from [8]).

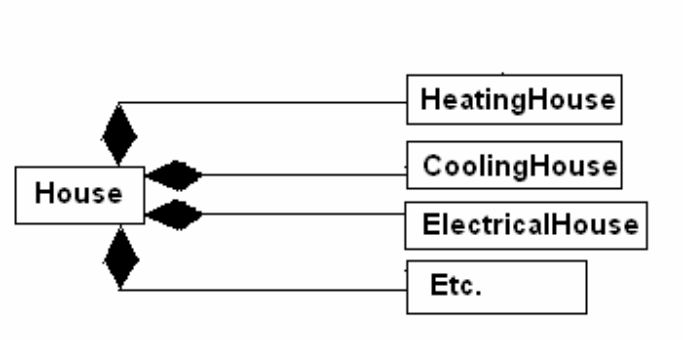

Fig 12: Containment by reference (taken from [8]).

4.3 TOAD Heuristics

TOAD (Teaching Object-Oriented Analysis and Design) [15], a prototype to evaluate object oriented designs. For each heuristic, Gibbon states a name, an intent i.e. why the heuristics should be used, property of the heuristic, consequences i.e. what are the advantages and disadvantages of the heuristic. He also mentions the motivation of using the heuristic, achievable effect when applying the heuristic, context specific information related to the application of the heuristic, description of other heuristics that may be related to the application of the current heuristic.

TOAD proposed to address the area of the capturing, categorizing, and cataloging of object oriented heuristics from multiple sources to create a single, comprehensive heuristics repository [35]. TOAD's heuristics states as a type of informal design evaluation to categorize design flows. By highlighting those flaws and showing the designers which heuristics violated, the designers gain insight on how to improve the design.

For example, "Inheritance hierarchy root should be abstract" (see # I3 in Appendix C). The rational behind of this heuristic is; since an inheritance hierarchy describes a family of classes, root class is the most important class because it provides an interface to its all dependence object. Designers cannot make instance of an abstract class. For that reason, they need subclass in order to access any functionality provided by the abstract class. Consider the following example as shown in figure 13; a student needs a class to store his information as shown in first figure. That student will be a full student after complete orientation course as shown in the next figure.

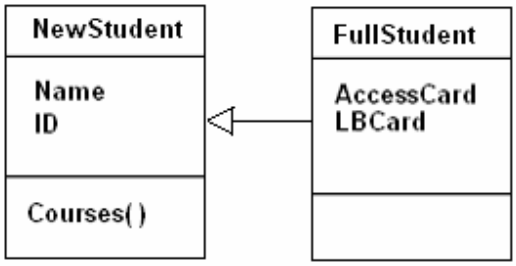

Figure 13: Using inheritance; NewStudent class and FullStudent class.

Since all students should go to an orientation course, designers want to add an attribute to track whether the student has attended the orientation course. Designers cannot add this attribute without adding it to the FullStudent class because FullStudent is a subclass of new student. According to above mention heuristics, the solution is: both classes inherit from an abstract base class that captures the common feature of both classes as shown in figure 14.

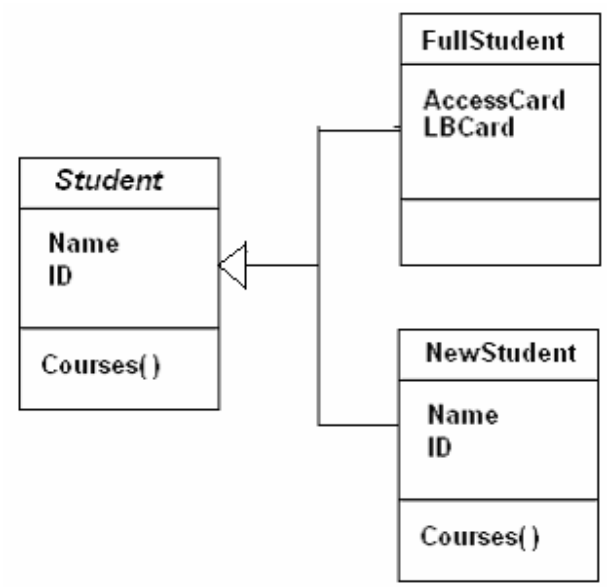

Figure 14: NewStudent class and Fullstudent class are inheriting from an abstract base class.

4.4 MeTHOOD Heuristics

MeTHOOD (Measures, Transformation Rules, and Heuristics for Object-Oriented Design) [16], is an approach where automated tools is used for object-oriented design evaluation and improvement. In this model, Grotehen and Dittrich et al. documented 20 design heuristics. For each and every heuristics MeTHOOD's describe, a name of the heuristics , a short description, a definition, rational, position in life cycle, granularity, an example, subsuming/subsumed heuristics, checking rules, transformation rules, violated heuristics, justification for violating the heuristics, and effect on measures.

Grotehen and Dittrich et al. categorized heuristics into three different ways; they are checkable heuristics, uncheckable heuristics and implied heuristics [16]. In checkable heuristics, a checking algorithm or function can be applied to the design fragment that evaluates to either true or false and checked for violation independently. An example of this type of heuristic is "A class should capture one and only one key abstraction with all its information and all its behavior "(see # 5 in Appendix B). The rational behind of the heuristics is that, if a single key abstraction is distributed among a set of classes, it is harder to find locations where changes have to be applied. Often some of the classes are very small and can be eliminated [17]. Consider the following example as shown in the following figure 15, in StudentProfile class, three key abstraction (Student, Profile and, Result) are modeled. The attribute value of student address for a given student will occur redundantly if more then one object contains the same customer. Therefore, the data consistency can easily be violated. This heuristic suggests that knowledge should not be distributed about an important abstraction among several classes.

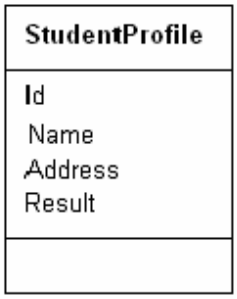

Figure 15: Student Profile

Which heuristics cannot be checked for a given design fragment automatically is known as Uncheckable heuristics mention by [24]. For example, "Avoid additional relationships

from base classes to their derived classes" (see # 8 in Appendix B). The rational behind this heuristics is that the derived class inherits the additional relationship, so besides the relationship between super and derived class, there is also a recursive relationship from the derived class to itself. Base and derived classes have already a relationship that implies a strong dependency of the derived to the base class. Avoid associations and using relationships that lead to a dependency from a base class to its derived class [17].

Implied heuristics mention a recommendation to optimize the value of a measure and its do not have a hard checking formula. These heuristics offer automated testing and testing algorithm returns a value, a particular design quality by evaluating the return value produced by an evaluation formula, such as a matrix.

4.5 Meyers Heuristics

In his C++ book, Meyers mention 35 ways to build more effective C++ applications [26]. Meyers rules are more like “idioms” (low level language based pattern) and too low level, that is, cannot be compared to Riel’s, TOAD’s, or MeTHOOD’s heuristics[5]. The following discussion is a summary of Meyer’s rules.

Meyer discusses pointers, references, casts, array and constructors in basis chapter. This chapter focuses on the difference between pointers and references and offer guideline on when to use each. This section also examines the C notion of arrays and C++ notion of polymorphism and it describes why mixing the two is an idea whose time will never come. For example, “never treat arrays polymorphically” (see # ITEM 3 in Appendix D). The rational behind this rule is, sometime designers might innocently manipulate arrays of derived class objects through base class pointers and references. There is no feature at all by manipulating arrays and it will never work.

In operator chapter, Meyers discuss when and how overloaded operator are called, how they behave, how they should relate to one another, and how they can seize control of these aspects overloaded operators. For example, “beware of user-defined conversation functions” (see # ITEM 5 in Appendix D). The main reason for this rule is that, C++ allows compilers to perform implicit conversation between types. For example, *char* to *int* and from *short* to *double*. This is why; designer can pass a *short* to a function that expects a *double*. Designers can not do anything about such conversation, because they are hard-coded into the language [26].

Some time designers have to face different problems that commonly arise when working with exceptions, Meyers discuss this issue in exception section in detail. There are several principles that can improve the performance of any program, are discuss in efficiency chapter. This chapter also focuses on how to prevent unnecessary objects from creeping into software design. In technique chapter Meyers discuss: how can a designer limit the number of instance of a class; how can a designer prevent objects from being created on the heap; how can a designer cerate objects that automatically perform some actions anytime. All the solutions to problems commonly encountered by C++ programmers. For example, “limit the number of objects of a class” (see # ITEM 26 in

[5] Discussed with Börstler J

Appendix D). This rule is similar to the heuristic of Riel “Minimize the number of messages in the protocol of a class” (see # 2.3 in Appendix A). The rational behind this heuristics is that, for example, from a large public interface, it is hard to find out something. By keeping the interface minimal, the system will be easier to understand and the objects easier to reuse.

4.6 Summary

All the authors mention their heuristics almost in the same way. Everyone define a name for a heuristic, the motivation of a heuristic, the advantages and disadvantages of a heuristic, an example of a heuristic and so on. Riel heuristics cover almost all areas of design. TOAD heuristic serves to assist novice learners and emphasis on aggregation, inheritance and single class design heuristics. MeTHOOD heuristics highlights on checking rules and the transformation rules. In Meyers heuristics, most of those heuristics focused on the programming language, rest of them deal with design. Sometimes designers want to make change to reduce complexity, flexibility from their system. They have to decide which heuristic makes the most sense for the existing design.

5 Evaluation of heuristics

This section will discuss about relationship between heuristics and design patterns. There are some similar heuristics found in literature those are frequently use by designers will be discussed.

5.1 Relationship between Patterns and Heuristics

A design pattern captures the concentrate of a successful solution for a particular design problem. Design pattern have a variety of uses, and an evaluative power. On the other hand, heuristics are simply general rules of thumb for solving a general design problem (see figure 5).

There are also exceptions to these general rules where particular problem have to be solved. That means design patterns can violate heuristics in certain cases. Lang C. et al. for example, discuss the relationship between design pattern and Riel's heuristics in detail [22]. The following is a summary of this discussion.

Singleton Pattern

The singleton pattern is one of the well known patterns in software design. This pattern can generate only a single instance of the class type. This pattern does not allow any parameters to be specified when creating the instance.

Violates: "Distribute system intelligence horizontally as uniformly as possible, i.e. the top level classes in a design should share the work uniformly" "(see # 3.1 in Appendix A) and "Do not create god classes/objects in your system. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem" (see # 3.2 in Appendix A) heuristics.

Reason for violation: The motivation for this violation is that certain objects in a computer system might be unique, i.e. there must be one and only one instance of them, that can be shared by a number of other objects.

Comments: A class, which has only a single instance with global visibility [8]. Singleton pattern only applies to classes that are guaranteed

to have a unique instance. All objects that use an instance of that class use the same interface. The design prohibits the creation of multiple instances (there is no public constructor).

Observer pattern

Observer pattern describes how to establish relationships. In this pattern, the change in the state of an object can cause automatic updates in a list of dependents objects, that means each observer will query the subject to synchronize its state with all dependents state.

Violate: "Keep related data and behavior in one place" (see # 2.9 in Appendix A) and "Minimize the number of classes with which another class collaborates" (see # 4.1 in Appendixes A).

Reason for violation: The motivation for this violation is to minimize the strength of the coupling between the subject and the observers.

Comments: When one object changes in the system, all its dependent objects are notified and updated automatically. For example, consider a spreadsheet program, when data insert or modify in that program, all the corresponding charts are change immediately, because document and chart class must know each other. The main idea of observer pattern is changing data in a window should be immediately reflect in all. This pattern also teaches how an object can tell other objects about events.

Command pattern

A command pattern is an object behavioral pattern that allows us to achieve complete decoupling between the sender and the receiver. Here, a sender is an object that invokes an operation, and a receiver is an object that receives the request to execute a certain operation. The term request here refers to the command that is to be executed[6].

Violations: "Do not turn an operation into a class" (see # 3.9 in Appendix A).

Reason for violation: The motivation for this violation is that the commands are subject to an undo operation and therefore, designers must implement them as class so that their corresponding object can be operate.

[6] Learn how to implement the Command pattern in Java.
http://www.javaworld.com/javaworld/javatips/jw-javatip68.html

Comments: The command pattern is popular for supporting undo. It states, how to implements command as objects whenever a command has both behavior and state[7].

Bridge pattern

Bridge pattern is used for decoupling an abstraction from its implementation. In this pattern, designers separate the interface and the implementation of a class and they link the implementation dynamically to the interface.

Violates: "Keep related data and behavior in one place" (see # 2.9 in Appendix A).

Reason for violation: The motivation for this violation is to allow there to be different versions of the class.

Comments: Bridge pattern implements the abstractions and implementations as independent classes that can be combined dynamically [13]. This pattern is sometime called driver because it act like a software interface which is used between software and hardware.

Decorator Pattern

The decorator pattern is used for adding additional functionality to a particular object as opposed to a class of objects. In this pattern, designers extended class by creating wrapper classes to encapsulate the class to be extended.

Violates: "Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better (see # 5.4 in Appendix A).

Reason for violation: The motivation for this violation is that the extensions are decorations added to the outside of the base class rather then internal modifications to the base class.

Comments: The decorate pattern provides more flexibility than inheritance, so by using the decorator pattern generally results in fewer classes then using inheritance. Few classes make design simple. The drawback of decorate pattern is that, it provides more objects; more object make debugging more difficult.[8]

[7] Horstmann, C.: Object oriented Design and Patterns, John Wiley & Sons, Inc 2004
[8] Grand, M.: Patterns in Java, Wiley Computer Publishing, 1998

As discussed in [22], Meyers states that design pattern can help designers understand the range of applicability of object oriented rules by showing how a design pattern produce a superior solution to that obtained by following the rules blindly [24].

Grotehen T. claims in his thesis [17] about MeTHOOD heuristics and design patterns. The following is a summary of this discussion.

MeTHOOD provides a high level language for systematic improvement and repair. It also supports the “refinement” part of a design. On the other hand, patters are high-level language for building and support the “creation” part of a particular design. Heuristics used for high-level communicating about the reasons for design trade-off (the why) whereas patterns used for high-level communicating about the structure of a design (the what), documenting the structure of a design. Heuristics focus on more general quality problem, patterns focus on very specific structural problem.

5.2 Comparison of heuristics

Riel’s [2] heuristics contained the most important collection of design heuristics. His heuristics are mostly general heuristics for class and objects, which prevent action oriented application. He discussed heuristics in relationship between class and objects, collaboration, containment, inheritance, association, and multiple inheritance categories. Very few heuristics cover in physical object oriented design. MeTHOOD [16] heuristics provide measure, heuristics and transformation rules as a design knowledge base for a design support system. TOAD [15] heuristics are able to examine static class properties captured in C++ header files based on a set of heuristics.

Heuristics can be used to automatically check the design for potential design flaws. In MeTHOOD [16], explicit transformation rules are use to generate proposals for alternative designs. As discussed in [17], Riel [2] suggests that a transformation pattern can be considered as a pattern pair consisting of a “good” and a “bad” pattern, which are “linked” by a heuristic. He suggests that the designer has to resolve these conflicts and proposes only limited support for this kind of decisions. MeTHOOD, heuristics are weighted according to their importance during different phases of the development cycle.

If the weighting is different for conflicting heuristics in a given development phase, this provides further selection criteria for the designer [17].

Although different writers have described different heuristics, according to object oriented design, there are some similarities that can be found in Riel [2], TOAD [15], MeTHOOD [16] and Meyers [26] heuristics. As indicated in section 2, some object oriented terminologies have been presented in this literature on object oriented design. Heuristics have been proposed in different categories in the context of object oriented terminology.

According to heuristics category, which is base on that design concepts, table 1 show among Riel, MeTHOOD, TOAD and Meyers heuristics. In this table, we have categorized the heuristics proposed by Riel, MeTHOOD, TOAD and Meyers in the following categories.

- The aggregation category denotes which instance variables have type that are user define classes in the design.
- The single class category could be implied to any number of objects which have the same general structure and behavior.
- In class attribute category discuss, where two or more objects to share same attributes.
- God class category indicates that a single class drives the application whereas all other classes are data holders.
- In class inheritance category, inheritance can be used to define a class in term of one or more other class.
- Method category indicates that methods are implemented by procedures that can access and design to instance variables of the target object.
- Multiple inheritance categories discuss when a class inherits from more than one superclass.
- Sometimes even a single violation cause major effect on the entire application. The proliferation of class category discuss on a particularly nasty pitfall in the design in the context of that violation.

Table 2 groups the most common heuristics, which was found by analyzing table 1. For example, one or more heuristics from Riel and one or more heuristics from MeTHOOD

or others heuristics use the same idea in their heuristics. Table 2 shows the most common heuristics and we mention their name respectively in the left column.

The goal of these tables is to identify the particular heuristics from different categories. First, designers have to find out which problem they have in their design. For example, assume your design contains a class with a very large interface. This is a large problem with reusability because it is quite hard to find out something particular, for example message. The solution can be found in single class design category. After analyze this category, “Heuristic 2” (see table 2) can be found as a “right” heuristic for that problem.

By doing so, designers can restrict number of applicable heuristics. This makes it easier to find the “right” heuristic for a design problem.

Heuristics Category	Riel [2]	MeTHOOD [16]	TOAD [15]	Meyer [26]
Aggregation	4.1, 4.2, 4.3		A1, A2, A3, A5, A6, A7, A8	
Single Class Design	2.1, 2.3, 2.8, 2.9, 2.10, 3.1, 3.3	1, 2, 3, 4, 5, 6, 8, 11, 12, 14, 15	C1, C2, C3, C4	25, 26, 28, 29, 30, 31, 33
Class Attribute	2.1, 2.3, 2.9, 3.4, 4.6, 5.9	9, 10, 12, 18, 19	C1, C2	
God Class	2.3, 2.8, 2.10, 3.1, 3.2, 3.3, 3.4, 4.6		C1, C3	
Class Inheritance	5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19	8, 9, 13, 14, 15, 16, 17	I1, I2, I3, I4, I5, I6, I7,I8, I9	
Methods	3.3, 8.1, 9.1	3,7,20,10, 17, 18, 20	C4	
Multiple Inheritance	6.1, 6.2, 6.3			24
Object-Class	2.9, 2.11, 9.2		C1, C2, C3	19, 27, 31
Proliferation of Classes	2.11, 3.7, 3.8, 3.9, 5.15			
Class Relationship (Coupling)	4.4, 4.5, 4.6, 4.7, 7.1	8,11	U1, U2, U3	

Table 1: Comparison of heuristics

Heuristic	Riel [2]	MeTHOOD [16]	TOAD [15]	Meyers [26]
Heuristic 1	2.1, 2.2	2		
Heuristic 2	2.3		C1, C2	25, 26, 28-31, 33
Heuristic 3	2.8, 2.9	5	A1, C1, C2	
Heuristic 4	3.2, 3.3		C1, C3	
Heuristic 5	4.1, 4.2, 4.3, 4.4	8, 9, 10		
Heuristic 6	5.9, 5.10, 5.11	14		
Heuristic 7	5.8		I3	

Table 2: Most similar heuristics by analyzing Table 1.

The following discussion shows those similar heuristics:

Heuristic 1: These heuristics describe that class properties should be hidden behind accessor method. It means that class properties should not be used in the public interface of a class. The public interface of a class is a list of methods and attributes that could be used from outside the class. Maintenance is the main motive of these heuristics [2]. The consistent enforcement of information hiding at the design and implementation level is responsible for a large part of the benefits of the object oriented paradigm. These heuristics are used in analysis, design and coding phases [16]. The main goal of these heuristics is maintenance and these heuristics aid designers to modify the behavior and implementations more freely.

Heuristic 2: The main motivate of these heuristics is to try to keep the number of member function to manageable number as well as to limit the number of attribute within a class. “By keeping the interface minimal, the system will be easier to understand and the components easier to reuse” (see # 2.4 in Appendix A). There will be too many responsibilities for those classes which have too many methods and for this reason the system will be more complex and harder to maintain.

Heuristic 3: These heuristics suggest that a class should capture one and only one key abstraction with all its information and all its behavior. A main entity within a domain

model is known as a key abstraction [1]. If the model have more than one abstraction in one class then the design will be centralized and becoming more complex. “If a key abstraction is distributed among a set of classes, an operation of the abstraction might involve more then one class. Some time it is hard to determine all affected classes. Often some of these class are very small and can be eliminated” (see # 5 in Appendix B).Riel states that there is a possibility that the designer has capture each function is to a class. Combining methods and attributes into a single class might result is a very huge class, even though it might correctly model the abstraction.

Heuristic 4: These heuristics mention, having too many message means either class doing too much or the methods have wrong granularity and for this reason the system will be complex to maintain. With large public interface, it is also difficult for designers to find a required method easily, thus making the class harder to understand and more complex [2]. A class that captures more then one set of abstraction suggests that there might be missing class. In this issue Riel mention that the majority of class methods use the majority of the class attributes.

Heuristic 5: These heuristics mention that once the number of collaboration is controlled, then it makes sense to minimize the amount of collaboration between classes. If these heuristics considered equal to the collaboration minimization heuristic, then it means that there is some break-even point between adding more messages sends between class and adding a new collaboration [2]. The amount of coupling is a small problem compared with the complexity of a new collaboration.

Heuristic 6: These heuristics suggest that common property could be a single abstraction, a class, and an instance. This makes the property more centralized and reusable. Dittrich R. mentions that if properties are common, defining them in different location results in undesired redundancies. These redundancies lead to unnecessary effort if the properties have to be changed [16]. As discussed in [2], Riel states as example that if all valves turn on and turn off, but each valve does it differently, then the base class “valve” is not useful unless there exists some place in the application where a generic valve will need to decide which it is at runtime.

Heuristic 7: These heuristics discuss, all the roots of an inheritance tree should be abstract. Since root class is possibly the most important class as it provides an interface. Abstract root classes that distribute implementation become problematic as the size of the

inheritance hierarchy increases. Hence, the root of inheritance hierarchy should be abstract type definition [15].

5.3 Conflicting Heuristics

Heuristics are nothing but a set of rules. These rules sometime may give conflicting advice. Börstler, J. for example discusses the conflicting heuristics in brief [8]. The following describes a summary of this discussion.

Heuristics: “All base class should be abstract” (see # 5.7 in Appendix A), “Eliminate irrelevant classes from your design” (see # 3.7 in Appendix A), and “Do not exclude behavior from subclasses” (this heuristics violate the LSP (see section 3.3).

The following example shows bird and penguin both are in Bird class and can move. Penguin can swim but bird cannot. Here, swim is not a meaningful behavior in Bird class so swim should be eliminating from the design.

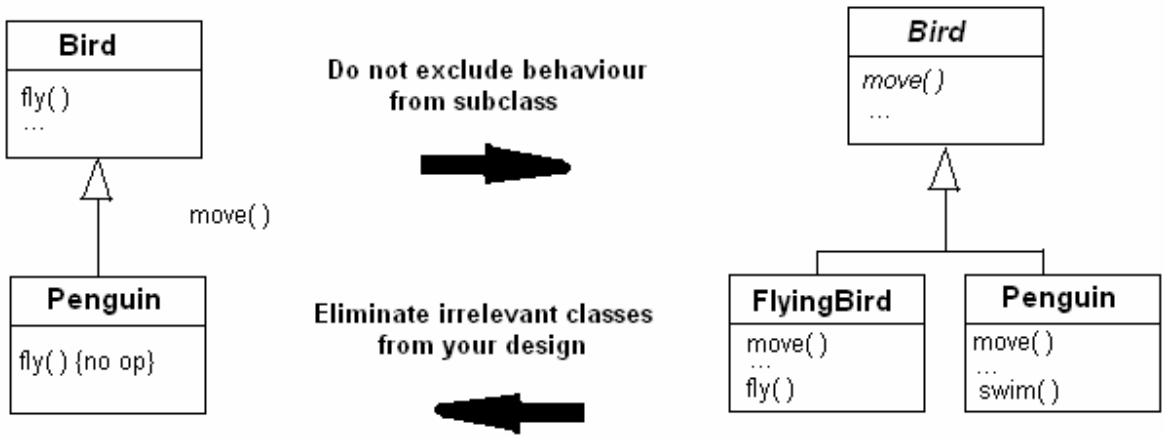

Figure 16: Conflicting heuristics (see [8]).

5.4 Summary

The aim of this section was to evaluate the design heuristics from different views. The first subsection mainly focuses on design patterns related to the heuristics. I discussed some similar heuristics; the purpose of the evaluation was to find out some general heuristics by comparing Riel, MeTHOOD, TOAD and Meyer heuristics. The way heuristics are expressed in the literature leaves a huge opportunity for designer's creativity. Designers can use their judgment and evaluate the rationale of heuristics to support their design. By underrating conflicting heuristics, designers can remove unnecessary behaviors from their design. Table 1 and table 2 are important to find the particular heuristics for a design problem.

6 Discussion

In this paper, I have discussed object oriented design heuristics. Heuristics can play an important role in the evaluation of object oriented designs. Many heuristics are proposed in the literature. Design heuristics are straightforward set of laws for software design, they are easier to remember, and use them concerned with developing an object oriented model of a software system to implement the identified requirements.

To support OO design, there are several object oriented methodologies being advocated at this time for designers. On the other hand, OO principles deal with class, cohesion, and coupling which are determines, how packages should be consistent. Design principles address the global design space while design patterns focus on solving a specific type of design challenges. Design patterns provide a design for refining the mechanism of a software system. Design patterns also describe commonly recurring structure of communicating mechanism that solves a general design problem within a particular situation. The violation of a rule in a design pattern clearly indicates its importance for ordinary object oriented design.

There is always tradeoff in design so that all heuristics will not work together; some heuristics are conflicting [8]. These conflicting heuristic may give wrong advice. For that reason, designers have to use measure to compare the effects of different heuristics on the design. Expert designers have to express their experience as heuristics to support object oriented design.

In this study, it could be found that Riel's heuristics [2] is more comprehensive and cover most of the object oriented design. Even though Meyer's heuristics [26] is not as comprehensive as Riel's heuristics, one must consider the fact that Meyers heuristics more for the object oriented programmers then for the designers. TOAD heuristics is responsible for analysis on a given design, and can give feedback on the latter's relatives metrics, anomalies and potential pitfalls. TOAD uses the design heuristics within an educational framework so that the learner appreciates why design heuristics improve quality as well as how to do it [15]. MeTHOOD heuristics makes available concrete design knowledge that help to improve conceptual object oriented schemas [15]. It is also clear that object oriented heuristics are an inevitable part of object oriented design and following these heuristics it is possible to produce better object oriented design.

7 References

[1]	Alexander, C.: An Introduction for Object-Oriented Designers, Oxford University Press, 1987.
[2]	Arthur J. Riel:Object-Oriented Design Heuristics.Addison-Wesley May 1996.
[3]	Barbara, H. Liskov and Stephen, N. Zilles: Programming with Abstract Data Types, Computation Structures Group, Memo No 99, MIT, Project MAC, Cambridge Mass, 1974.
[4]	Beck, K. and Cunningham, W.: A laboratory for teaching object-oriented thinking. In Proc. OOPSLA'89, ACM Sigplan Notices 17(4), pp. 1-6
[5]	Booch, G.: Object-Oriented Analysis And Design With Applications, 1991.
[6]	Booch, G: Object-Oriented Analysis and Design with Applications, 2nd ed., Benjamin Cummings, 1994
[7]	Budd, T.:Introduction to Object-Oriented Programming, Addison-Wesley, 1991
[8]	Börstler, J: Object Oriented Software Development, Class Lecture, http://www.cs.umu.se/kurser/TDBC31
[9]	Coad, P., Yourdon, E.; 'Object-Oriented Design'; Yourdon Press, Prentice Hall, New Jersey; 1991
[10]	Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P.: Object-Oriented Development: the Fusion Method, Prentice-Hall, 1994
[11]	Edward, V. Berard "A Comparison of Object-Oriented Development Methodologies," The Object Agency, Inc., 1995
[12]	Edward,.V. Berard: Essays on Object-Oriented Software Engineering, Volume 1, Prentice Hall, Englewood Cliffs, New Jersey, 1993
[13]	Gamma, E., Helm, R., Johnson, R., and Vlissides J.: *Design Patterns*, Elements of Reusable Object-Oriented Software Addison-Wesley, 1995
[14]	Gibbon, C., Higgins, C.: Teaching object-oriented design with heuristics, ACM SIGPLAN Notices, Volume 31 Issue 7, July 1996
[15]	Gibbon, C.: TOAD(Teaching Object-Oriented Analysis and Design), PhD Thesis, 1997
[16]	Grotehen, T., Klaus, R. Dittrich : The MeTHOOD Approach, Transformation Rules, and Heuristics for Object Oriented Design, Technical report
[17]	Grotehen, T.: “Objectbase Design: A heuristic Approach.”, Dissertation, Der Wirtschaftswissenschaftlichen Akultätder Universität Zürich, 2001

[18]	Henderson, B. Sellers, and Edwards, J.: Book Two of Object-Oriented Knowledge: the Working Object, Prentice-Hall, 1994
[19]	Henderson, B. Sellers, Simons, A., and Younessi, H.: The OPEN Toolbox of Techniques, Addison Wesley Longman, 1998
[20]	Ince, D.; 'Object-Oriented Software Engineering with C++'; McGraw-Hill, London; 1991
[21]	Jacobson, I., Christerson, M., Jonsson, P., and Overgaard G.: Object-Oriented Software Engineering: A Use-Case Driven Approach, Addison-Wesley, 1992
[22]	Joseph, E. Lang, Brian, R. Bogovich: “Object-Oriented Programming and Design Patterns”, SIGCSE Bulletin, Vol 33.No. 4. 2001
[23]	Lakos, J.: Large Scale C++ Software Design, Addison and Wesley, 1996.
[24]	Meyer, B.: “Object-Oriented Software Construction”, Prentice Hall 1997
[25]	Meyer, B.: Object Oriented Software Construction, Prentice Hall, 1988.
[26]	Meyers, S.: More Effictive C++,35 New Ways to Improve your Programs and Design, Addison Wesley, 1996
[27]	Paul, H.: Objects In Action: Commercial Applications Of Object- Oriented Technologies. Jan, 1993. .
[28]	Preece, J., Sharp, H.., and Rogers, H.: Interaction Design: Beyond Human-Computer Interaction. John Wiley & Sons. 2002
[29]	Robert C. Martin, Engineering Notebook, C++ Report, Nov-Dec, 1996
[30]	Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices ,2002
[31]	Robinson, P.J. 'Introduction and overview'; In: Object-Oriented Design; Robinson P.J. ed.; Chapman & Hall, London; 1992; P1-10
[32]	Rubin, K. and Goldberg A.: "Object behaviour analysis", Comm. ACM, 35 (9), (1992).
[33]	Rumbaugh, J., Blaha, M., Premerlani, W., Eddy F. and Lorensen, W: Object-Oriented Modeling and Design, Prentice-Hall, 1991.
[34]	Shlaer, S. and Mellor, S.: Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, 1988
[35]	Thomas, J. Graser: “The Reference Architecture Representation Environment (RARE) Systematic Derivation of Object-Oriented Systems”, Dissertation Proposal, 1998
[36]	Walden, K. and Nerson, J. M.: Seamless Object-Oriented Software Architecture, Prentice-Hall, 1994,

8 Appendix

A. Riel Heuristics

Heuristic #2.1 : All data should be hidden within its class.

Heuristic #2.2 : Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

Heuristic #2.3 : Minimize the number of messages in the protocol of a class.

Heuristic #2.4 : Implement a minimal public interface which all classes understand (e.g. operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from a ASCII description, etc.).

Heuristic #2.5 : Do not put implementation details such as common-code private functions into the public interface of a class.

Heuristic #2.6 : Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.

Heuristic #2.7 : Classes should only exhibit nil or export coupling with other classes, i.e. a class should only use operations in the public interface of another class or have nothing to do with that class.

Heuristic #2.8 : A class should capture one and only one key abstraction.

Heuristic #2.9 : Keep related data and behavior in one place.

Heuristic #2.10 : Spin off non-related information into another class (i.e. non-communicating behavior).

Heuristic #2.11 : Be sure the abstraction that you model are classes and not simply the roles objects play.

Heuristic #3.1: Distribute system intelligence horizontally as uniformly as possible, i.e. the top level classes in a design should share the work uniformly.

Heuristic #3.2 : Do not create god classes/objects in your system. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem.

Heuristic #3.3 : Beware of classes that have many accessor methods defined in their public interface, many of them imply that related data and behavior are not being kept in one place.

Heuristic #3.4 : Beware of classes which have too much non-communicating behavior, i.e. methods which operate on a proper subset of the data members of a class. God classes often exhibit lots of non-communicating behavior.

Heuristic #3.5 : In applications which consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

Heuristic #3.6 : Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behavior in one place).

Heuristic #3.7 : Eliminate irrelevant classes from your design.

Heuristic #3.8 : Eliminate classes that are outside the system.

Heuristic #3.9 : Do not turn an operation into a class. Be suspicious of any class whose name is a verb or derived from a verb. Especially those which have only one piece of meaningful behavior (i.e. do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class.

Heuristic #3.10 : Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Heuristic #4.1 : Minimize the number of classes with which another class collaborates.

Heuristic #4.2 : Minimize the number of message sends between a class and its collaborator.

Heuristic #4.3 : Minimize the amount of collaboration between a class and its collaborator, i.e. the number of different messages sent.

Heuristic #4.4 : Minimize fanout in a class, i.e. the product of the number of messages defined by the class and the messages they send.

Heuristic #4.5 : If a class contains objects of another class then the containing class should be sending messages to the contained objects, i.e. the containment relationship should always imply a uses relationship.

Heuristic #4.6 : Most of the methods defined on a class should be using most of the data members most of the time.

Heuristic #4.7 : Classes should not contain more objects than a developer can fit in his or her short term memory. A favorite value for this number is six.

Heuristic #4.8 : Distribute system intelligence vertically down narrow and deep containment hierarchies.

Heuristic #4.9 : When implementing semantic constraints, it is best to implement them in terms of the class definition. Often this will lead to a proliferation of classes in which case the constraint must be implemented in the behavior of the class, usually, but not necessarily, in the constructor.

Heuristic #4.10 : When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down a containment hierarchy as the domain allows.

Heuristic #4.11 : The semantic information on which a constraint is based is best placed in a central third-party object when that information is volatile.

Heuristic #4.12 : The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.

Heuristic #4.13 : A class must know what it contains, but it should never know who contains it.

Heuristic #4.14 : Objects which share lexical scope, i.e. those contained in the same containing class, should not have uses relationships between them.

Heuristic #5.1 : Inheritance should only be used to model a specialization hierarchy.

Heuristic #5.2 : Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

Heuristic #5.3 : All data in a base class should be private, i.e. do not use protected data.

Heuristic #5.4 : Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better.

Heuristic #5.5 : Pragmatically, inheritance hierarchies should be no deeper than an average person can keep in their short term memory. A popular value for this depth is six.

Heuristic #5.6 : All abstract classes must be base classes.

Heuristic #5.7 : All base classes should be abstract classes.

Heuristic #5.8 : Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.

Heuristic #5.9 : If two or more classes only share common data (no common behavior) then that common data should be placed in a class which will be contained by each sharing class.

Heuristic #5.10 : If two or more classes have common data and behavior (i.e. methods) then those classes should each inherit from a common base class which captures those data and methods.

Heuristic #5.11 : If two or more classes only share common interface (i.e. messages, not methods) then they should inherit from a common base class only if they will be used polymorphically.

Heuristic #5.12 : Explicit case analysis on the type of an object is usually an error, the designer should use polymorphism in most of these cases.

Heuristic #5.13 : Explicit case analysis on the value of an attribute is often an error. The class should be decomposed into an inheritance hierarchy where each value of the attribute is transformed into a derived class.

Heuristic #5.14 : Do not model the dynamic semantics of a class through the use of the inheritance relationship. An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at runtime.

Heuristic #5.15 : Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.

Heuristic #5.16 : If you think you need to create new classes at runtime, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a class.

Heuristic #5.17 : It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing.

Heuristic #5.18 : Do not confuse optional containment with the need for inheritance, modeling optional containment with inheritance will lead to a proliferation of classes.

Heuristic #5.19 : When building an inheritance hierarchy try to construct reusable frameworks rather than reusable components.

Heuristic #6.1 : If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise.

Heuristic #6.2 : Whenever there is inheritance in an object-oriented design ask yourself two questions: 1) Am I a special type of the thing I'm inheriting from? and 2) Is the thing I'm inheriting from part of me?

Heuristic #6.3 : Whenever you have found a multiple inheritance relationship in a object-oriented design be sure that no base class is actually a derived class of another base class, i.e. accidental multiple inheritance.

Heuristic #7.1 : When given a choice in an object-oriented design between a containment relationship and an association relationship, choose the containment relationship.

Heuristic #8.1 : Do not use global data or functions to perform bookkeeping information on the objects of a class, class variables or methods should be used instead.

Heuristic #9.1 : Object-oriented designers should never allow physical design criteria to corrupt their logical designs. However, very often physical design criteria is used in the decision making process at logical design time.

Heuristic #9.2 : Do not change the state of an object without going through its public interface.

B. MeTHOOD Heuristics

Heuristic #1 : A class in a containment hierarchy should only depend from its child classes.

Heuristic #2 : Every attribute should be hidden within its class.

Heuristic #3 : A client-server dependency between two classes should not lead to dependencies from the server to the client.

Heuristic #4 : Avoid dependencies from database classes to their clients.

Heuristic #5 : A class should capture one and only one key abstraction with all its information and all its behaviour.

Heuristic #6 : Do not create unnecessary classes to model roles.

Heuristic #7 : Avoid pure accessor methods.

Heuristic #8 : Avoid additional relationships from base classes to their derived classes.

Heuristic #9 : Avoid classes with properties implying redundancies .

Heuristic #10 : Avoid multivalued dependencies .

Heuristic #11 : Convert associations, and uses relationships in the strongest containment relationship wherever possible.

Heuristic #12 : Avoid contained instances that have to be modified concurrently.

Heuristic #13 : All properties of the base class interface must be usable in instances of its derived classes in every location where a base class instance is expected.

Heuristic #14 : Common properties of instances should be defined in a single location.

Heuristic #15 : Instable classes should not be base classes.

Heuristic #16 : Do not misuse inheritance for sharing attributes.

Heuristic #17 : The overloading should define only differences to the overloaded method.

Heuristic #18 : Avoid case analysis on properties of instances.

Heuristic #19 : Prefer typing by attribute before typing by inheritance.

Heuristic #20 : A method should use only classes of attributes of its class, classes of its parameters, or classes of instances locally created.

C. TOAD Heuristics

Aggregation Design Heuristics:

A1. The aggregate should limit the number of aggregated.
A2. Limit the number of user-defined aggregated.
A3. Restrict direct access to aggregated by clients.
A4. Limit enabling mechanisms that breach encapsulation.
A5. The aggregation leaf nodes are small, reusable & simple.
A6. Create aggregation hierarchies that are narrow and deep.
A7. Aggregation hierarchies should not be too deep.
A8. Stability should descend the hierarchy

Inheritance Design Heuristics:

I1. Limit the use of multiple inheritance.
I2. Inheritance hierarchy should be no more than six deep.
I3. Inheritance hierarchies roots should be abstract.
I4. The inheritance hierarchy is a specialisation hierarchy.
I5. Minimise breaks in the type/class hierarchy.
I6. Abstract root classes should be type definitions.
I7. Prevent the over-generalisation of a superclass.
I8. Make abstract as many intermediary nodes as possible.
I9. Stability should ascend the hierarchy.

Using Design Heuristics:

U1. Limit the number of collaborating classes.
U2. Minimise visibility of interface collaborators.
U3. Identify and stabilise common place collaborators.

Single Class Design heuristics:

C1. Limit an object's absolute bandwidth.
C2. Limit the internal attributes of an object.
C3. Limit an object's true responsibilities.
C4. Minimise complex methods.

D. Meyers Heuristics

Basics.

ITEM 1. Distinguish Between Pointers and References.
ITEM 2. Prefer C++-Style Casts.
ITEM 3. Never Treat Arrays Polymorphically.
ITEM 4. Avoid Gratuitous Default Constructors.

Operators.
ITEM 5. Be Wary of User-Defined Conversion Functions.
ITEM 6. Distinguish Between Prefix and Postfix Forms of Increment and decrement operators.
ITEM 7. Never Overload &&, ||, or,.
ITEM 8. Understand the Different Meanings of New and Delete.

Exceptions.
ITEM 9. Use Destructors to Prevent Resource Leaks.
ITEM 10. Prevent Resource Leaks in Constructors.
ITEM 11. Prevent Exceptions from Leaving Destructors.
ITEM 12. Understand How Throwing an Exception Differs from Passing a Parameter or Calling a Virtual Function.
ITEM 13. Catch Exceptions by Reference.
ITEM 14. Use Exception Specifications Judiciously.
ITEM 15. Understand the Costs of Exception Handling.

Efficiency.
ITEM 16. Remember the 80-20 Rule.
ITEM 17. Consider Using Lazy Evaluation.
ITEM 18. Amortize the Cost of Expected Computations.
ITEM 19. Understand the Origin of Temporary Objects.
ITEM 20. Facilitate the Return Value Optimization.
ITEM 21. Overload to Avoid Implicit Type Conversions.
ITEM 22. Consider Using Op= Instead of Stand-Alone Op.
ITEM 23. Consider Alternative Libraries.

ITEM 24. Understand the Costs of Virtual Functions, Multiple Inheritance, Virtual Base Classes, and RTTI.

Techniques.

ITEM 25. Virtualizing Constructors and Non-Member Functions.
ITEM 26. Limiting the Number of Objects of a Class.
ITEM 27. Requiring or Prohibiting Heap-Based Objects.
ITEM 28. Smart Pointers.
ITEM 29. Reference Counting.
ITEM 30. Proxy Classes.
ITEM 31. Making Functions Virtual With Respect to More Than One Object.

Miscellany.

ITEM 32. Program in the Future Tense.
ITEM 33. Make Non-Leaf Classes Abstract.
ITEM 34. Understand How to Combine C++ and C in the Same Program.
ITEM 35. Familiarize Yourself With the Language Standard.

E. Guidelines (OMT Model)

REUSABILITY

1. Keep methods coherent
2. Keep methods small
3. Keep methods consistent
4. Separate policy and implementation
5. Provide uniform coverage
6. Broaden the method as much as possible
7. Avoid global information
8. Avoid modes
9. Subroutines
10. Factoring
11. Delegation
12. Encapsulate external code

EXTENSIBILITY

1. Encapsulate classes
2. Hide data structures
3. Avoid traversing multiple links or methods
4. Avoid case statements on object type
5. Distinguish public and private operations

ROBUSTNESS

1. Protect against errors
2. Optimize after the program runs
3. Validate arguments
4. Avoid predefined limits
5. Instrument the program for debugging and performance monitoring

PROGRAMMING-IN-THE-LARGE

1. Do not prematurely begin programming
2. Keep methods understandable
3. Make methods readable
4. Use exactly the same names as in the object model
5. Choose names carefully
6. Use programming guidelines
7. Package into modules
8. Document classes and methods
9. Publish the specification