Heuristics are powerful tools in designing software and they provide a more subjective view of software quality. Application of heuristics is a difficult task and this potentially places a greater burden on the developers who must interpret this view since it consists of potentially conflicting indicators with varying degrees of precision and relevance. Heuristics may occur as individual pieces of developers’ or as a suite covering multiple aspects of software development.
CHAPTER 5
DESIGN HEURISTICS

Software development is a complex process. It is very difficult to define or measure software qualities and quantities and to determine a valid measurement metric.

5.1 Object Oriented Design

Object-oriented design is a method of design encompassing the process of object-oriented decomposing and a notation for depicting logical and physical as well as static and dynamic models of the system under design (Booch, 1994). Object-oriented paradigm is more closely related to the real world situations and has significantly replaced the structured paradigm in industry in past two decades. Procedural systems are based on functions and data does not depend upon operations in these systems. But in case of real world problems, data is closely related to operations as it defines the state of a real world object whereas operations define the behavior of that object. This concept is used by Object-Oriented systems to solve a problem. The main emphasis in object oriented design is focused towards objects and classes. Object-oriented design is related to develop an object-oriented module of a software system to serve the identified and analyzed requirements for the system. The main features of an object-oriented design include improved design quality, faster development process, high reusability features and modularity. Booch introduced a methodology, given in Figure 5.1, which shows the necessary features of an object-oriented design (Booch, 1994). It comprises of a four-step process to design an object-oriented system. These are reproduced as under:

- Identify Classes and Objects: Identify key abstractions in the problem space and label them as potential classes and objects.
- Identify semantics of Classes and Objects: The meanings of the previously identified classes and objects are established, including defining the life cycle of each object from creation to destruction.
• Identification of Class-Object Relationship: Class and object interaction is identified, for example patterns of inheritance and patterns of visibility among objects and classes

• Specify Class and Object Interfaces and Implement Classes and Objects: Detailed internal view is constructed which includes defining methods and their behaviors.

![Diagram of Object-Oriented Design](image)

**Figure 5.1: Object-Oriented Methodology (Booch, 1994)**

There are a number of internal attributes of Object-Oriented Design which are described as follows:

• **Coupling**: Coupling is defined as the measure of the relative interdependence among modules. For example, object A is coupled to object B only if A sends a message to B. Thus coupling refers to the number of messages passed between objects.

• **Cohesion**: “Cohesion measures the degree of connectivity among the elements of a single class or object” (Booch, 1994). Cohesion relates to a measure of the relative functional strength of a module. It measures encapsulation within an object and deals with method’s data interaction inside an object that makes it internally bonded. A class is said to be cohesive when the methods and variables contained are highly correlated. Cohesion can be used to identify the badly designed classes.
• Inheritance: Inheritance is a mechanism in which one object inherits characteristics from one or more than one objects. It occurs at all levels of a class hierarchy. It is used to construct relationship between super classes and subclasses in various ways as inheritance enables the attributes and operations of a class to be inherited by all subclasses and the objects that are instantiated.

• Encapsulation: Encapsulation encapsulates data and the operations into a single named object. It is an indirect measure of data abstraction and information hiding. Encapsulation hides internal specification of an object and shows only external interface. The process of compartmentalizing the elements of an abstraction that constitutes its structure and behaviour is encapsulation. Encapsulation serves to separate the contractual interface of an abstraction and its implementation (Booch, 1994). Encapsulation influences software metrics by changing the focus of measurement from a single module to a package of data.

• Information Hiding: Information hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics (Booch, 1994). Public interface and a private implementation of an object are kept distinct. All information about a module should be private to the module unless it is specifically declared public. Information hiding plays a strong role in such metrics as object coupling and the degree of information hiding.

• Localization: Localization is a characteristic of software that indicates the manner in which information is concentrated within a program (Pressman, 1997). In the object oriented context, information is concentrated by encapsulating both data and functions within the bounds of a class or object. According to Booch, localization is the process of gathering and placing things in close physical proximity to each other (Booch, 1994). It is based on objects in case of object-oriented design. A design plan is totally dependent upon the localization approach, because one function may involve several objects and one object may provide many functions. Metrics should apply to the class as a complete entity. Even the relationship between functions and classes is not necessarily one-to-one. For that reason, metrics that reflect the manner in which classes collaborate must be capable of accommodating one-to-many and many-to-one relationship.
5.2 Heuristics for Object Oriented Design

Design evaluation is effective and beneficial to both expert and novice designers during software development process. Design heuristics are proposed as a more accessible and informal means by which developers can evaluate OO design. Software heuristics are small, simple, legible, self-contained nuggets of design expertise. They target specific design problems and provide guidance to affect a solution. Unlike metrics, heuristics are outwardly defined in terms of the observable problems that occur during OOD. Moreover, the lessons learned from applying the low level design concepts such as coupling, cohesion and size throughout metric research provide a solid theoretical foundation that heuristics build upon to document recurring and observable problems within OO systems.

Design heuristics are available to all developers performing OOD and are applicable within a number of software domains. They permit small, incremental enhancements to maintainability and provide a common vocabulary for expressing design problems (Gibbon, 1997). Design heuristics are shown to be concentrated pieces of design expertise that deliver knowledge and experience from the expert to the novice. These need to be simple and understandable in order for them to be useful to the majority of developers performing design evaluation. Heuristics offer insightful information based upon past experience that is known to work in practice. They provide a means by which knowledge and experience can be delivered from the expert to the novice. To achieve this goal, heuristics must identify and encapsulate experience so that their subsequent application ultimately results in the reuse of this experience (Kumar et al., 2011a).

In the context of OOD, a design heuristic focuses upon a single, self-contained design problem. The design heuristic not only imparts knowledge of the documented problem but also suggests the ways to solve the problem. By representing small yet focused design problems, heuristics are applicable within a number of diverse software domains and at different phases of the life-cycle. Typically, the problems documented by design heuristics are visible during all phases starting from analysis till up to implementation. There are two main types of design heuristics namely, Class and Relationship (weblink 5). Class heuristics are the simplest type of heuristic that acts upon individual classes. They evaluate class properties pertaining to method complexity, size, behavioral content
and encapsulation, mechanisms that breach information hiding, centralized control and so on. They are purposefully simple and typically used in conjunction with relationship heuristics.

Relationship heuristics are defined for the inheritance, aggregation and using class relationships. A relationship heuristic acts upon a group of classes, or class group, that participates in a specific class relationship. A head class is representative of a class group such that all relationship heuristics are directed at the head classes. For example, the head class in an inheritance relationship is the root of a class hierarchy and the members of the class group are its descendant classes.

Measuring the structural design properties of software artifacts with design metrics, is a promising approach at an early stage. Estimation model provides an assessment of the defect proneness of the system in an early stage by analyzing the inter relationship among the defect occurrence and design parameters of the software (Selvarani et al., 2009).

5.3 Conflicts among Heuristics

A publication by Riel (Riel, 1996) was possibly the first in-depth research effort that focused directly upon the need and use of OOD heuristics within the software development process. The heuristics were presented in an example-driven manner and were comprehensively described. However, there were no common mechanisms for documenting the design heuristics as self-contained, transferable pieces of design expertise. Also, the research neither presented the inter relationship among heuristics nor any plan was proposed to deploy these heuristics in support of an informal approach to design evaluation. Furthermore, a number of Riel's design heuristics were qualitative except the automated one.

A notable characteristic of design patterns is that they often break rules. For example, the Composite pattern advocates the use of methods that are overridden to do nothing, contrary to a common maxim, expressed by Riel’s heuristic as “It should be illegal for a derived class to override a base class method with an NOP method, i.e. a method which does nothing.” Many similar examples of conflicting forces can be found.

Some conflicts are so pervasive that they apply to nearly all design situations. Separation of concerns, for example, encourages decoupling portions of a design, while another
heuristics, “Keep related data and behavior in one place” often suggests the opposite. It has also been observed that conflicts occur even within an organized set of heuristics. One heuristic says “Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better”, while another adds the qualification that “In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six”. Reusability is enhanced by depth and not by breadth. But a popular heuristic contradicts this statement by suggesting “Greater is the value of NOC greater will be the reusability of the parent class”. Heuristics are a valuable tool for identifying design forces and evaluating design quality, but their application is not straightforward (Churcher, 2007) for many reasons, such as:

- **Lack of consensus on which heuristics should be adopted**: Some conflicting heuristics usually illuminate matters of concern to the designer. Other conflicts, however, reflect differing design philosophies, and a particular designer is likely to be interested only in one side of the debate. Many of the tenets arising from software reuse culture, for example, are in opposition to more recent refactoring and agile method approaches.

  The open/closed principle, for example, encourages anticipation of future needs by making the design open for extension (reusable), but without requiring modification of existing code where as refactoring culture discourages anticipation of future needs and prefers modifying existing code when necessary. This cultural difference might show up in unexpected ways, such as a stronger preference for small methods in the reuse culture, so that methods constitute small overrideable units.

- **Nebulous definitions**: Heuristics are generally defined in vague and indefinite ways. One heuristic, for example, says “A class should capture one and only one key abstraction”, but rigorously specifying the meaning of “key abstraction” is problematic. Similarly, another heuristic “Model the real world whenever possible”, is only as firm as our grip on reality.

- **Subjectivity and calibration**: Code smells require the designer to judge when some intangible threshold has been crossed. The “large class smell”, “lazy class smell” and “long method smell” are obvious examples where different standards might apply. The relative importance of conflicting heuristics is also dependent on the value
system of the designer. Such as if breaking up a large class produces a lazy class and then the resultant lazy class is evaluated to check whether better result has been achieved or not.

- **Interpretation in different contexts:** Many heuristics are expressed abstractly, in order to apply to any OO design. It may be necessary, however, to adapt a heuristic to local conditions. For example, when deciding if an inheritance hierarchy is too deep, should the root class be counted in programming languages that enforces a single root? Or, in an organization that has adopted a refactoring approach to software development, how much emphasis should be placed on a heuristic motivated by software reuse, such as heuristic “All base classes should be abstract classes”?

- **Diverse levels of abstraction.** Some heuristics can be interpreted at different levels. For example, “All data should be hidden within its class”, might be viewed as a syntactic restriction—make attributes private—or as a semantic one, which might also discourage the use of getters. A “long method smell” could be detected at a lexical level by counting lines of code, at a syntactic level by counting statements and expressions, at a language semantic level by counting method invocations, collaborators etc. or at a problem-domain semantic level by gauging the conceptual size of the method.

- **Information overload:** Heuristics are intended to help software engineers to manage the complexity of software. But at the same time injudicious application of heuristics could compound the problem.

- **Acquiring relevant data and relating it to heuristics:** Many heuristics require substantial data gathering. Heuristic “Minimize fan-out in a class” and another “Most of the methods defined on a class should be using most of the data members most of the time” are examples. Additionally, the correspondence between available information and heuristics is not always clear.

The automation of heuristics becomes difficult due to these issues and the inherent fuzziness of heuristics. In consequence, designers usually must gauge the quality of their products without assistance from tools. The designer builds a mental model of the software and evaluates according to a subjective and subconscious process that is likely to be informed by heuristics but may explicitly apply few.
5.4 Conflicts among Inferences

Heuristics give judgments based on past experience and are not meant to be exact. Due to this reason, there exist conflicts in inference of an individual heuristic. Some of the examples of conflicts among inferences are discussed in the following section:

- **Fan- In (FIN) and Fan-Out Metric:** Henry and Kafura introduced software structure metrics based on information flow in 1981 which measures complexity as a function of Fan-in and Fan-out. In procedure oriented code, FIN can be defined as the number of modules that use (make calls to) the module, plus the number of global variables read by the module where as fan-out is the number of modules that this module uses (makes calls to), plus the number of global variables written by the module (Henry et al., 1981). If FIN of a module is zero, it indicates that no other modules are using this one module and it is possibly an unused or dead code.

  Structural complexity of a module i, is defined as:

  \[
  S(i) = [\text{Fan}_{\text{in}}(i) \times \text{Fan}_{\text{out}}(i)]^2
  \]

  Eq. (5.1)

  Where \( \text{Fan}_{\text{in}} \) is Fan-in of module i and \( \text{Fan}_{\text{out}}(i) \) is Fan out of module i.

  If Fan-out of a module increases, it indicates that the responsibility of module increases and which in turn increases the complexity of a module. But if Fan-out is less than one, it is an indicator of less hierarchical design.

  In object oriented code, Fan-in and Fan-out metrics are structural metrics which measure inter-module complexities. Fan-in defines the number of modules that call a given module where as Fan-out represents the numbers of modules that are called by a given module. It is an indication of multiple inheritances. High Fan-in indicates that an object is extensively used by other objects and thus, promotes reusability. Therefore, FIN should be maximized.

  But increase in inheritance increases complexity of a class because if \( \text{FIN} > 1 \), it indicates that a class inherits its attributes and operations from more than one root class. And hence, should be avoided when possible.

  So, there exist conflicts in inference of an individual heuristic whether to maximize or to minimize its value.

- **DIT Metric:** DIT measures the maximum length from the node to the root of the tree. More is the depth of the inheritance tree, greater will be the reusability of class and
reduces coding, testing and documentation time. So, value of DIT should be high. But as DIT increases, complexity, debugging and maintenance effort also increases. Therefore, DIT should be minimized as testing becomes more expensive. Hence, it is advisable to regulate the depth of a tree and the value of DIT metric should be kept neither too high nor too low.

- **NOC Metric**: Number of children represents the number of immediate sub-classes subordinated to a class in the class hierarchy. As value of NOC increases, reusability also increases. Classes with large number of children are harder to maintain and modify. Thus, such classes require more testing due to the effects of changes on all the children. They are also considered more complex and fault-prone. But at the same time, high NOC has been found to indicate fewer faults due to high reuse, which is desirable. So, there exist conflicts in inference of NOC whether to maximize or to minimize its value.

- **Method/Attribute Hiding Factor Metric**: Method/Attribute hiding factor measures the process with which variables and methods are encapsulated in a class. The value of MHF/AHF is related to the average amount of hiding among all classes in the system. A private method/attribute fully exhibits data hiding which increases usability. Encapsulation promotes maintenance and code changes can be made independently with more flexibility. It also reduces coupling of modules and increases cohesion inside a module. For a good design, coupling should be kept low and cohesion should be high. The number of visible methods is a measure of the class functionality. MHF will reduce if overall functionality will increase. However, for implementing the functionality, hidden methods are used which offers information hiding benefits and favors increase in MHF. A low MHF indicates that the probability of errors is high. A high MHF indicates very less functionality and that the design includes a high proportion of specialized methods that are not available for reuse. Increased value of MHF decreases bug-density, defect density and rework effort to find and correct defects and hence, increases quality. Hence, a conflict exist in inference of this metric that whether to maximize its value or to minimize it. So, MHF should be kept in a reasonable range.
• **Method/Attribute Inheritance Factor (MIF) Metric:** MIF and AIF should be kept in a reasonable range, i.e. neither too low, nor too high. Very high value indicates either superfluous inheritance or too wide member scopes. A low value of MIF and AIF indicates lack of inheritance and heavy use of Overrides. The value of MIF/AIF provides an indication of inheritance. Inheritance increases reusability, productivity, reduces coding, testing and documentation time. Thereby, decreases complexity by reducing the number of operations and operators but inheritance increases testing effort, complexity and fault-proneness.

• **Polymorphism Factor (PF) Metric:** PF can be defined as the number of methods that redefine inherited methods divided by the maximum number of possible distinct polymorphic situations. In case of run time polymorphism, Polymorphism factor increases as overriding increases which in turn increases the reusability of a class and reduces lines of code. When value of PF is 100%, it means that all methods are overridden in all derived classes. A PF value of 0% may indicate that there is no use of inheritance and polymorphism. A PF value well above 10% is considered very high and reduces quality benefits. Polymorphism can be used to a reasonable extent to keep the code clear, but excessive use of polymorphic code may decrease understandability. So, there are conflicting recommendations for the acceptable values of PF and its value should lie in a reasonable range with a lower and an upper bound.

**5.5 Summary**

Heuristics are an important part of software design and are becoming more widely used. Effective visualization of heuristics includes quantitative, qualitative and ambient aspects. Visualization of heuristics provides many challenges. Heuristics are likely to be studied both individually and in comparison with others. There exist conflicts among heuristics like Lack of consensus on which heuristics should be adopted, nebulous definitions, subjectivity, calibration and interpretation in different contexts. Also, conflicts among inferences exist in case of metrics like DIT, NOC, MHF/AHF, PF, Fan-in and Fan-out etc.