Fundamental Object Oriented Design Heuristics

- Do not repeat yourself! Do not repeat yourself! Do not repeat yourself! [I just did that :( ]
- All data should be hidden within its class:
- Users of a class must be dependent on its public interface, but a class should not be dependent on its clients.
- Minimize the number of messages in the specification of a class
- Implement a minimal public interface that all clients understand
- Do not put implementation details, such as supporting functionality, in the public interface of a class
- Do not clutter the public interface of a class with features that users are not able to use or are not interested in using.
- A class should capture one and only one key abstraction
- Keep related data and behavior in one place.
- Spin-off unrelated information into another class
- Ask not that an object give you its state, or set its state explicitely, ask an object to do something for you.
- See objects as bundles of services not as bundles of data.
- design service oriented objects that use their state to decide how to behave.
- Use immutable objects, also refered as Messengers
- **During design, Distribute system intelligence horizontally and uniformly as possible**
- Do not create “god” classes in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem.
- Beware of classes that have many accessor(set and get) methods defined in the public interface. Having many implies that related data and behavior are not being kept in one place, that is the class that contains all these accessors.
- In applications that consist of an OO model interacting via a user interface, the model should never be dependent on the interface. The interface should depend on the model. Note: there is a need of accessor methods (gets and sets) to follow this guideline. This need for accessor methods is unfortunate but no less necessary. Although the classes of the model will have accessor methods, the other classes in the model should NOT use them!!(Ha, fat chance). That portion of class’s public interface should be reserved solely for the classes in the user interface portion of the design. Could we use the private interface pattern here???
- Model the real world whenever possible.
- Eliminate irrelevant classes from your design.
- Eliminate classes that are outside the system.
- In general, do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those that have only one piece of meaningful behavior. One exception is when you are objectifying an operation to abstract choices of implementations of such operation (strategy pattern.)
- Minimize the number of classes with which another class collaborates.
- Minimize the number of message sends between a class and its collaborator.(This is a form of coupling called dynamic coupling)
- Minimize the amount of collaboration between a class and its collaborator, i.e. the number of different messages sent.
- 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.
- Most of the methods defined on a class should be using most of the data members most of the time.A sign of poor cohesion when this heuristic is broken.
- Classes should not contain more class instance variables (attributes) than a developer can fit in his or her short term memory. A favorite value for this number is six.
- A class must know what it contains, but it should never know who contains it.
- Inheritance should only be used to model a specialization hierarchy.
- Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.
- All data in a base class should be private, i.e. do not use protected data. Specify protected queries which return the desired data while hiding that data’s representation.
- Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better.
- 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.
- For implementation hierarchies, All abstract classes must be base classes.

Fundamental Object Oriented Design Heuristics

- In implementation hierarchies, All base classes should be abstract classes.
- 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.
- 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.
- If two or more classes only share common interface (i.e. not implementation) they they should inherit from a Java interface or common base class only if they will be used polymorphically.
- Explicit case analysis on the type of an object is usually an error, the designer should use polymorphism in most of these cases.
- **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. Consider the possible use of the State pattern.**
- **Favor composition over inheritance.**
- 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.
- If you think you need to create new classes at run-time, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a class.
- 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. Breaks the Contract as established in the base class.
- Do not confuse optional containment with the need for inheritance, modeling optional containment with inheritance will lead to a proliferation of classes.
- When using inheritance in an object-oriented design ask yourself two questions: 1) Is this a special type of the thing I'm inheriting from? and 2) Is the thing I'm inheriting from part of it?
- Do not change the state of an object without going through its public interface.
- Program to an interface not an implementation.
- To close a class you must have in mind a list of future changes. In general there will be changes for which a class is not closed.
- Overwritten methods in subclasses should weaken superclass' pre-conditions and strengthen superclass' post-conditions for such methods.
- The implementation of a class should not depend on implementation details of other classes. All classes should depend upon abstractions.
- Beware of fat classes. Different clients will use different subsets of a fat class interface. Redesign the fat class so that a client should not be forced to depend upon interfaces they do not use.