OOAD
Object-Oriented-Analysis and Design

OOAD

- OOS
- OOA
- OOD
- **Thumbnails** for the principles of the object oriented design .
- OO Design Quality Metrics

Object-Oriented System(OOS)

An **object-oriented system** is composed of objects.

Behavior of the system is achieved through collaboration between these objects

State of the system is the combined state of all the objects in it.

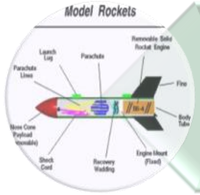
Collaboration between objects involves them sending messages to each other.

Exact semantics of message sending between objects varies depending on what kind of system is being modeled.

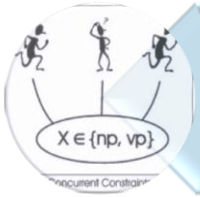
“Sending Message“ Types :

1. By “Invoking a method".
2. By Sending data via a socket

Object-Oriented Analysis (OOA)



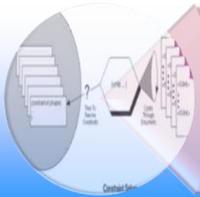
Aims to model *the problem domain*, the problem we want to solve by developing an object-oriented (OO) system.



An analysis model will not take into account implementation constraints, such as concurrency, distribution, persistence, or inheritance, nor how the system will be built.

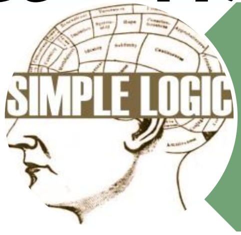
Model of a system can be divided into multiple domains each of which are separately analyzed, and represent separate business, technological, or conceptual areas of interest.

of **what** is to be built, using concepts and relationships between concepts, often expressed as a conceptual model. Any other documentation that is needed to describe what is to be built, is also included in the result of the analysis. That can include a detailed user



Implementation constraints are decided during the object-oriented design (OOD) process.

Object-Oriented Design (OOD)



Looks for logical solutions to solve a problem, using objects.

Takes the conceptual model from OOA result, and adds implementation constraints imposed by

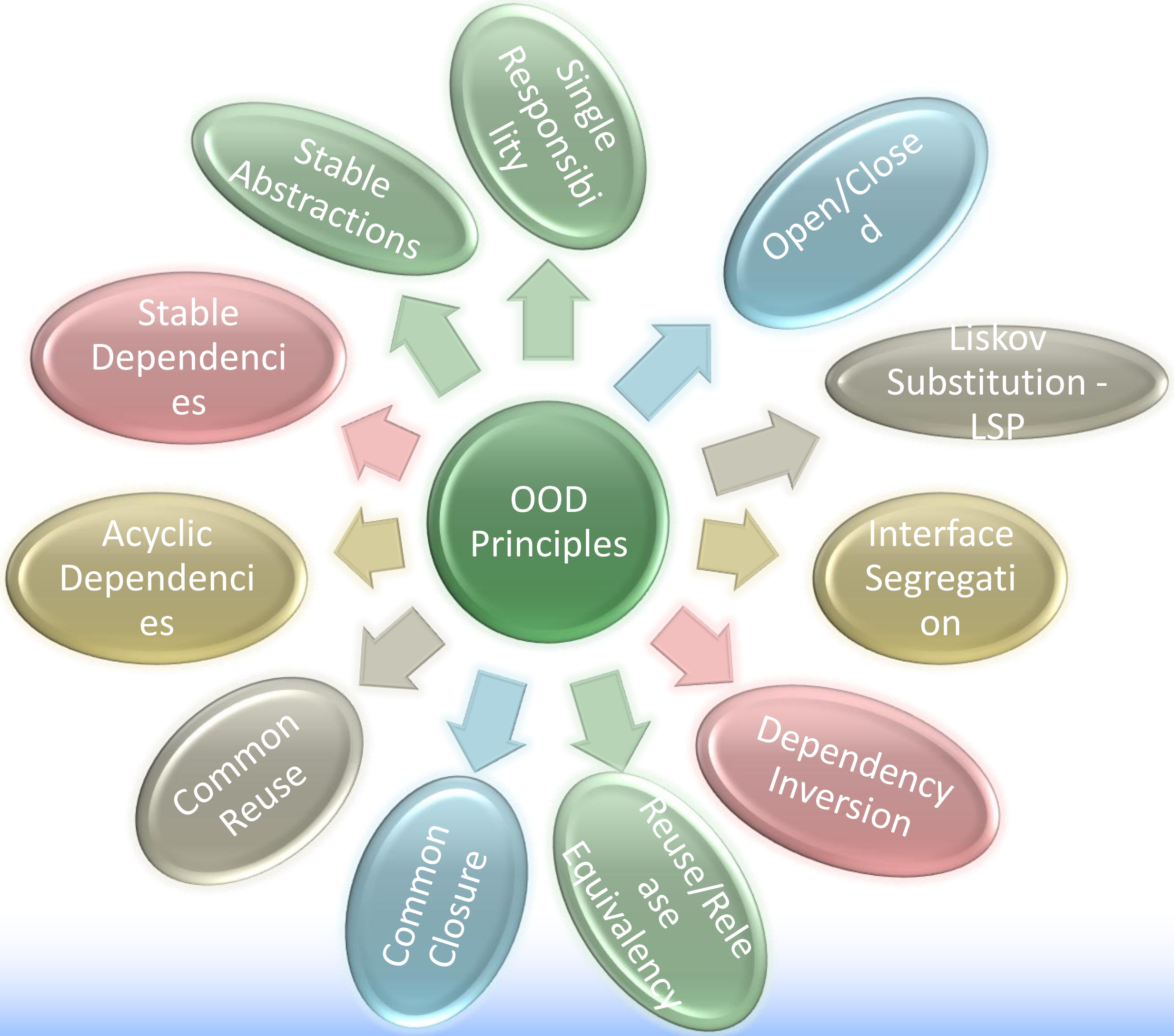
- the environment,
- the programming language and the chosen tools,
- as well as architectural assumptions chosen as basis of design.

Concepts in the conceptual model are mapped

- to concrete classes, to abstract interfaces in APIs and
- to roles that the objects take in various situations. The interfaces and their implementations for stable concepts can be made available as reusable services.

Unstable Concepts in OOA will form basis for policy classes that make decisions, implement environment-specific or situation specific logic or algorithms.

Principles of the object oriented design



Principles -SOLID

Single Responsibility Principle

- ✓ A class should have a single responsibility:
  - ✓ it does it all,
  - ✓ does it well, and
  - ✓ does it only.

- ✓ A class should have one, and only one, reason to change.
- ✓ Each responsibility should be a separate class, because each responsibility is an axis of change.
- ✓ If a change to the business rules causes a class to change, then a change to the database schema, GUI, report format, or any other segment of the system should not force that class to change

Rule of thumb:

- ✓ Unless description of a class is in 25 words or less, with no use of "and" or "or" may actually have more than one class.
- ✓ Classes, interfaces, functions, etc. all become large and bloated when they're trying to do too many things.
- ✓ To avoid bloat and confusion, and ensure that code is truly simple (not just quick to hack out) we have to practice *Code Normalization*, which seems to be a variation on OnceAndOnlyOnce and also *DoTheSimplestThingThatCouldPossiblyWork* . This is part of ResponsibilityDrivenDesign.

Open/Closed Principle:

- Software entities (classes, modules, etc) should be open for extension, but closed for modification.
- In other words, (in an ideal world...) you should never need to change existing code or classes: All new functionality can be added by adding new subclasses and overriding methods, or by reusing existing code through delegation.
- This prevents you from introducing new bugs in existing code. If you never change it, you can't break it.
- Using Abstract Class, Class Implemented with Interfaces

Liskov Substitution Principle- LSP :

- Derived classes must be usable through the base class interface without the need for the user to know the difference.
- "An instance of a derived should be able to replace any instance of its superclass"

In Detail

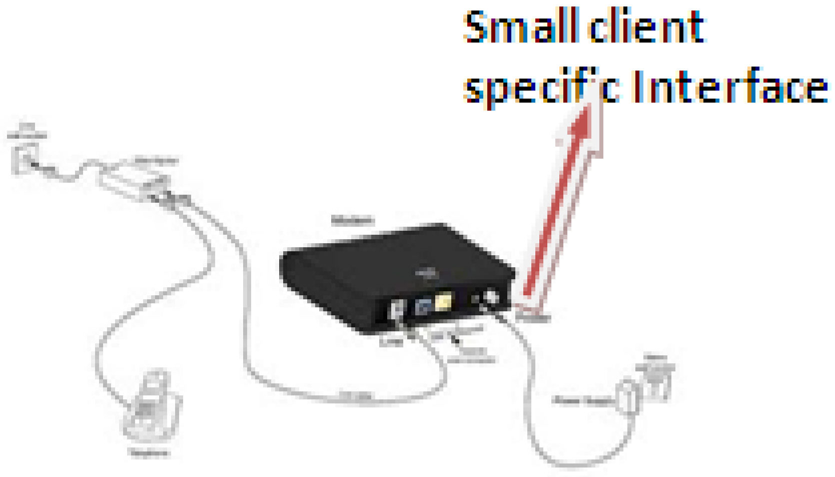
- The use of hierarchy is an important component in object-oriented design. Hierarchy allows the use of type families, in which higher level supertypes capture the behavior that all of their subtypes have in common. For this methodology to be effective, it is necessary to have a clear understanding of how subtypes and supertypes are related.
- "objects of the subtype ought to behave the same as those the supertype as far as anyone or any program using supertype objects can tell".

Important

- 1. Because if not, then class hierarchies would be a mess. Mess being that whenever a subclass instance was passed as parameter to any method, strange behavior would occur.
- 2. Because if not, unit tests for the superclass would never succeed for the subclass

Interface Segregation Principle:

- Many client specific interfaces are better than one general purpose interface
- The dependency of one class to another one should depend on the smallest possible interface



Dependency Inversion Principle:

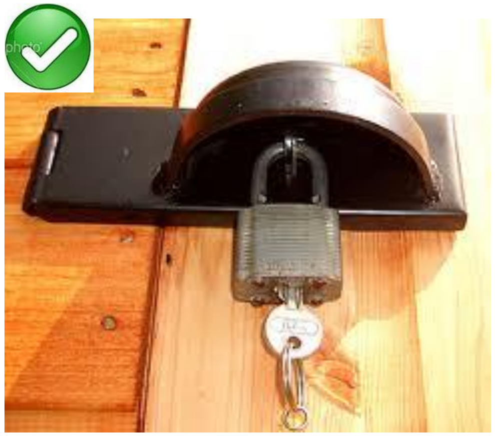
- Details should depend upon abstractions. Abstractions should not depend upon details.
- "The modules that implement a high level policy should not depend on the modules that implement the low level policies, but rather, they should depend on some well-defined interfaces. It stems from the realization that we must reverse the direction of these dependencies to avoid rigidity, fragility and immobility.
- avoid designs which are:
- Rigid (Hard to change due to dependencies. Especially since dependencies are transitive.)
- Fragile (Changes cause unexpected bugs.)
- Immobile (Difficult to reuse due to implicit dependence on current application code.)

Principles of package cohesion

- Reuse/Release Equivalency Principle: The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused.
- Common Closure Principle: Classes that change together, belong together.
- Common Reuse Principle: Classes that aren't reused together should not be grouped together.

Principles of Package Coupling

- The Acyclic Dependencies Principle: The dependency structure for released components must be a directed acyclic graph. There can be no cycles.
- The Stable Dependencies Principle: Dependencies between released categories must run in the direction of stability. The dependee must be more stable than the depender.

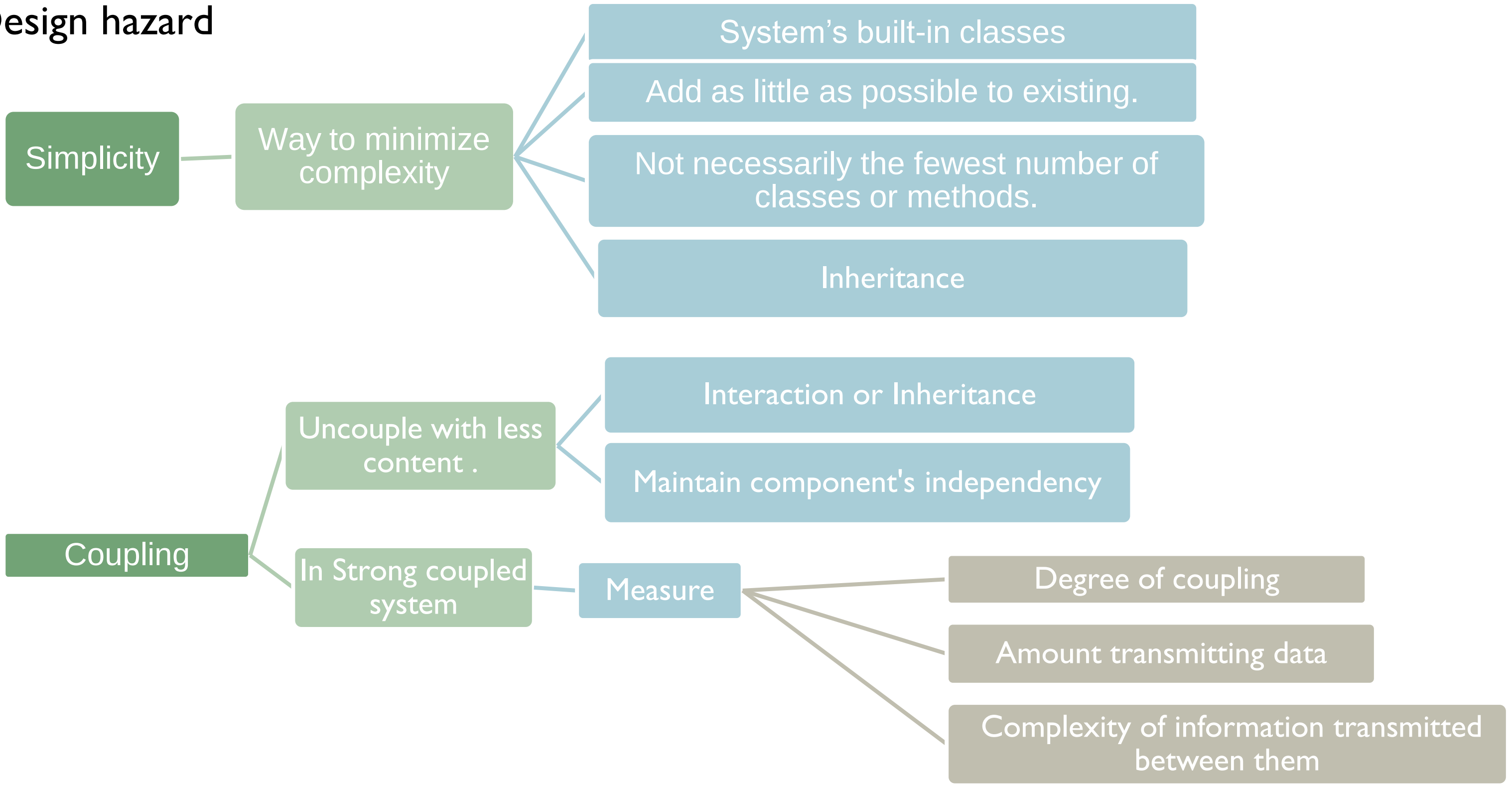


- The Stable Abstractions Principle: The more stable a class category is, the more it must consist of abstract classes. A completely stable category should consist of nothing but abstract classes.

Best designs

OOD Guidelines –(**Design rules for** *best designs* **)**

During the design process, as we go from requirement and use-case to a system component, each component must satisfy that requirement, without affecting other requirements. Following are steps to avoid Design hazard


Simplicity
Way to minimize complexity
System's built-in classes
Add as little as possible to existing.
Not necessarily the fewest number of classes or methods.
Inheritance
Coupling
Uncouple with less content .
Interaction or Inheritance
Maintain component's independency
In Strong coupled system
Measure
Degree of coupling
Amount transmitting data
Complexity of information transmitted between them

Inheritance

✓A subclass is coupled to its super-class in terms of attributes & methods

✓High inheritance coupling is desirable

✓Each specialization class should not inherit lots of unrelated & unneeded methods & attributes

Coupling Types and Degree

- Direct reference to attributes or methods of another object

Content Coupling Very High-%

- Two objects accessing a 'global data space', for both to read & write

Common Coupling High-%

- Explicit control of the processing logic of one object by another

Control Coupling Medium-%

- Passing an aggregate data structure to another object, which uses only a portion of the components of the data structure

Stamp coupling Low-%

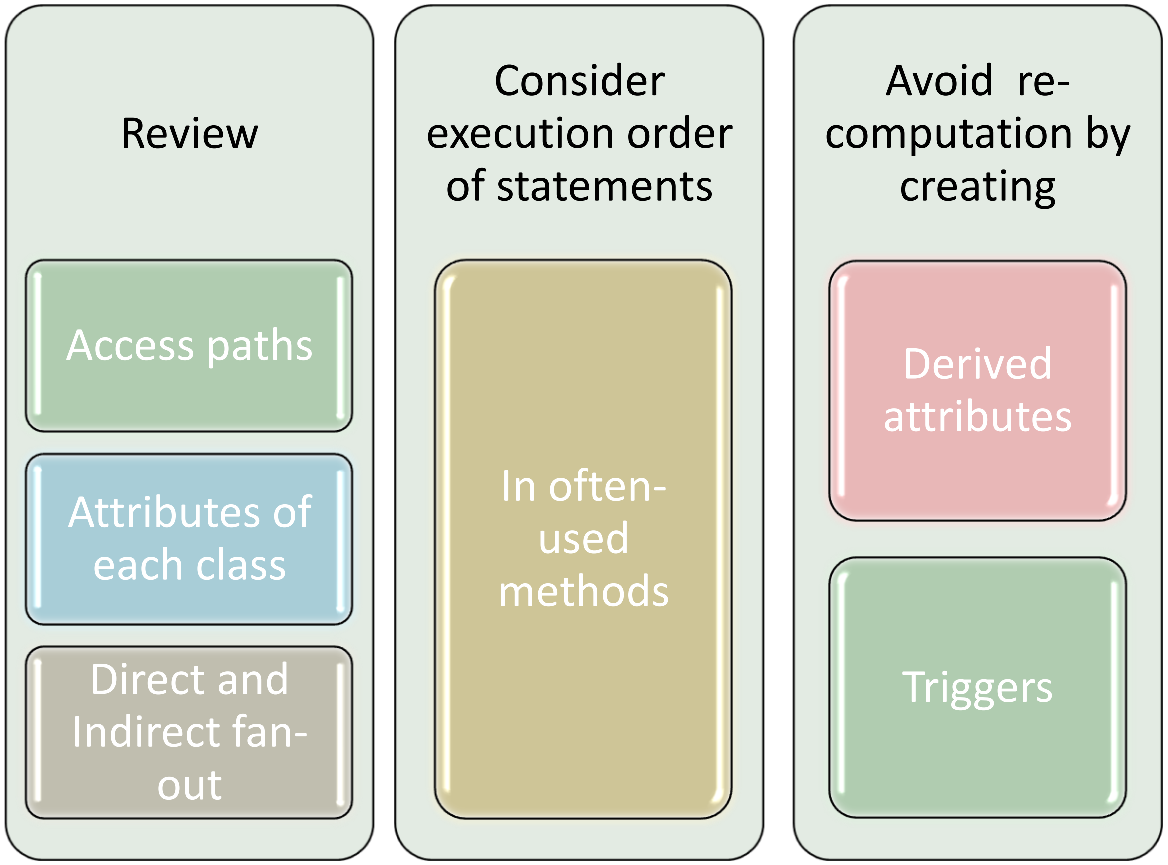
- Either simple data items or aggregate structures all of whose elements are used by the recevingobject. (this is the goal of an architectural design)

Data coupling VeryLow-%

Cohesion (interaction within a single object or swcomponent)

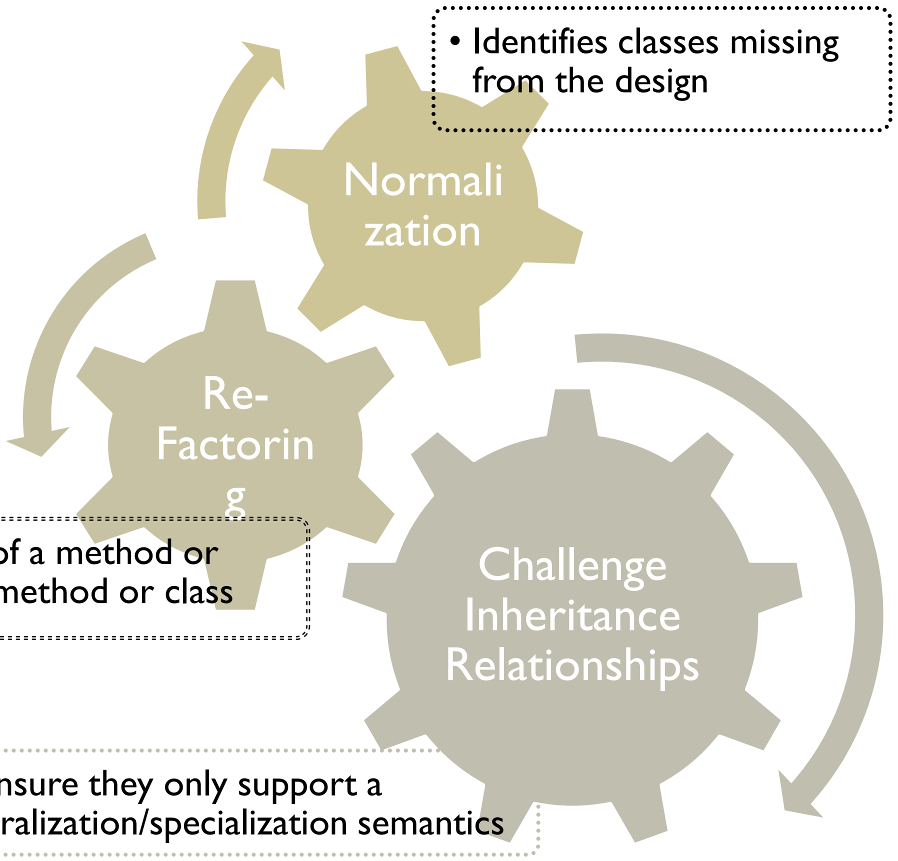
- Reflects the ’single-purposeness’ of an object
- Method cohesion
    - A method
        - should carry only one function.
        - carrying multiple functions is undesirable

Optimization Techniques



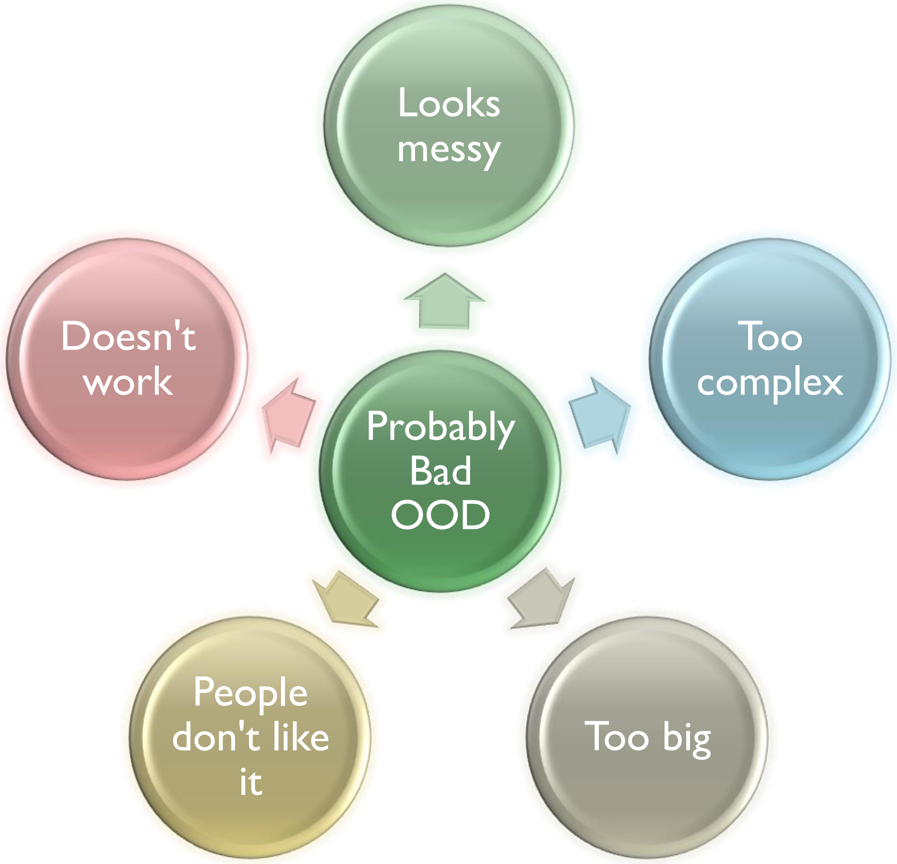
Review
Access paths
Attributes of each class
Direct and Indirect fan-out
Consider execution order of statements
In often-used methods
Avoid re-computation by creating
Derived attributes
Triggers

Restructure Consideration



- Identifies classes missing from the design
- Separate aspect of a method or class into a new method or class
- To ensure they only support a generalization/specialization semantics

Rules For Identifying Bad Design



OO Design Quality Metrics

Metrics (To measure the quality of an object-oriented design)

Interdependence (between the subsystems)

- High interdependent tend to be rigid, un-reusable , hard to maintain and cascade changes in dependent modules
- Collaboration of subsystems need interdependence
- Support communications within the design,
- Isolate reusable elements from non-reusable elements, and
- Block the propagation of change due to maintenance.
- Decide ability of design to survive change, or to be reused
- Fragile on single change.

Dependency

- “Good Dependency” is a dependency upon something that is very stable
- “Bad Dependency” is a dependency upon something that is instable

Stability

- Most stable classes of all, are classes that are both Independent and Responsible. Such classes have no reason to change, and lots of reasons not to change.

Class Categories: the granule of Reuse and Release

It is seldom that a class can be reused in isolation.

"Class Category" - A Class Category (hereinafter referred to as simply a category) is a group of highly cohesive classes that obey the following three rules:

- Classes within a category are closed together against any force of change. If one class must change, all of the classes within the category are likely to change.
- Classes within a category are reused together. Strongly interdependent and cannot be separated from each other. Thus if any attempt is made to reuse one class within the category, all the other classes must be reused with it.
- Classes within a category share some common function or achieve some common goal.
- Three rules are listed in order of their importance. Rule 3 can be sacrificed for rule 2 which can, in turn, be sacrificed for rule 1.

the authors must provide releases of their categories and identify them with release numbers so that reusers can be assured that they can have access to versions of the category that will not be changed.

Dependency Metrics

Responsibility, Independence and Stability of a category can be measured by counting the dependencies that interact with that category.

Ca : Afferent Couplings : The number of classes outside this category that depend upon classes within this category.

Ce : Efferent Couplings : The number of classes inside this category that depend upon classes outside this categories.

I : Instability : (Ce ÷ (Ca+Ce)) : This metric has the range [0,1].

I=0 indicates a maximally stable category.

I=1 indicates a maximally instable category.

Highly stable categories depend upon nothing

Instable categories should not be abstract, they should be concrete.

Desirably portion of the design to be flexible enough to withstand significant amount of change.