Object Oriented Design

# Reasons for Object Oriented Design

The promise of object oriented design (OOD) is that the code created will be easier to maintain and evolve. The ease of evolution is critical since modifying code in business cost money and is time critical.

Object-oriented applications are made up of parts (objects) that interact through messages to produce the behaviour of a whole (program), and meet customer requirements. Getting the right message to the target requires the sender to know things about the receiver, this creates dependencies between the two and these dependencies stand in the way of change. Object oriented design is about managing these dependencies such that an object can tolerate change without crashing the program.

# Object Oriented Design Principles

SOLID

* S Single Responsibility
* O Open-Closed
* L Liskov Substitution
* I Interface Segregation
* D Dependency Inversion

# Single Responsibility

It is good OOD to model applications using classes of single responsibility, with all their corresponding methods contained within them. The qualities of good code generally follow the TRUE acronym:

* T Transparent – consequences of any change should be obvious
* R Reasonable – cost/benefit of any change should be reasonable
* U Useable – code should remain usable in existing and new concepts
* E Exemplary – good code encourages any modifying code to be good

Single responsibility is when a defined class or method does the smallest possible useful thing, as this is very important when it comes to clearly defining what the behaviour the class or method is to have now, and allowing it to be easy to change/modify in the future. If a class was defined with many different roles and responsibilities, changing one part such as a method or data structure will likely require changing many others, costing time and money. Therefore, classes should act as single pluggable units, with all code relating to the function of the class, in the class. If the code in the class is then all acting for the classes central purpose, the code will be of high cohesion.

Code is easy to change when:

* There are no unexpected side effects
* Small changes of requirements mean small changes in code
* Easy to reuse

The best way to check whether a class or method is of single responsibility, is whether its purpose can be described in one sentence.

Hiding Instance Variables

One of the first step to making code easy to change is by wrapping class instance variables in accessor methods, such as attr\_reader, instead of referring to them in the class directly. This is in effect hiding the definition of the variable to the rest of the class. Calling a instance variables with attr\_reader will encapsulate the instance variable allowing it to be called like an object in the class.

Unhidden instance variables:

*class Gear*

*def initialize(chainring, cog)*

*@chainring = chainring*

*@cog = cog*

*end*

*def ratio*

*@chainring / @cog.to\_f* 🡨 instance variables are called directly

*end*

*end*

Hidden instance variables:

*class Gear*

*attr\_reader :chainring, :cog* 🡨 attr\_reader used to encapsulate instance variables

*def initialize(chainring, cog)*

*@chainring = chainring*

*@cog = cog*

*end*

*def ratio*

*chainring / cog.to\_f* 🡨 instance variables call via reader

*end*

*end*

What attr\_reader effectively does is put the instance variable equal to itself in a method called itself:

*def cog*

*@cog*

*end*

Therefore, if at any point in the future the definition of cog changes, it will be possible to simply explicitly change define and change the cog method definition without changing all callings of the instance variable throughout the class, saving considerable time.

The only issue with hiding instance variables using attr\_reader is that now the variables are publicly accessible via object.variable (gear\_instance.cog), if the variable contains sensitive data, this will be undesirable. To prevent this define sensitive variables using attr\_reader privately, by adding private before definition.

Hiding Data Structures

Like instance variables, data structures are also subject to frequent change during class/program development. Therefore, referring to them directly in methods, such as cell[1], will cause many issues if the data structure of the object ‘cell’ were to change. For this reason it is important to hide data structures from the class in a similar way to which instance variables are hidden.

*class RevealingReferences*

*attr\_reader :wheels*

*def initialize(data)*

*@wheels = wheelify(data)* 🡨 data structure automatically hidden as it enters class

*end*

*def diameters*

*wheels.collect {|wheel|*

*wheel.rim + (wheel.tire \* 2)}* 🡨 method calls data indirectly via wheel.name

*end*

*Wheel = Struct.new(:rim, :tire)* 🡨 create wheel structure with name accessible values

*def wheelify(data)*

*data.collect {|cell| Wheel.new(cell[0], cell[1])}* 🡨 define values for required class data

*end*

*end*

Using this method of hiding a data structure not only possible to call the required data by name (wheel.tire), if the data structure of the input data changes at a point in the future, only the method to define the data structure for the class is required to change, instead of every point at which the data structure is called.

Single Responsibility Methods

Method also require single responsibility, which although may mean creating considerably more methods, affecting performance, it will allow for several benefits in the design of the code. In any case, the code can always be refactored for performance in the future if required. As with classes, to get single responsibility methods, all behaviour must be isolated, so that each method only does one thing. The benefits this brings are:

* Shows the full abilities of a class clearly, making its purpose more obvious
* Behaviours can be easily modified when required without causing bugs
* Reduces the need for comments as all methods do what they say on the tin
* Encourages reuse as it easier to find and reuse methods simply named
* Easily move the methods to different classes due to their simplicity

# Managing Dependencies

For any desired behaviour, an object either knows it, inherits it, or knows another object who knows it. Since OOD objects have single responsibility, for a progam to perform an object must know something about another, creating a dependency.

Recognising Dependencies

A dependency is when an object knows (defined in object/class):

* Name of another class
* Name of message (method) to send to another object/class
* Arguments required for a message
* Order of arguments for a message

How highly the objects are dependant on other is how tightly coupled they are. Since when code is tightly coupled, if modifications are made to one, modifications must be made to another to allow the code to function. The tighter they objects are coupled, the higher the cost to modify code.

Another problem with objects which are coupled is chain dependency, this is where an object has dependencies with another, which directly has dependecies with another, and so on. Therefore, making any change very destructive since the whole chain will require modification to allow the program to function.

Luckily it is possible to eliminate or significantly reduce most of these dependencies by writing loosly coupled code using a series of methods.

Inject Dependencies

To remove the issue of refering directly to the name of a object/dependency in the class definition, inject any objects (instances/classes/modules/etc) required directly as arguments during initalization and save them as a variable.

Methods can then be applied directly to the variable, meaning if the injected object were to change, there would be no issue so long as it still responds to the required methods. This also gives the benefit that the class will then work with any object which responds to the ‘object.method’ message sent in the class definition.

See example below, where wheel is an object (instance of wheel class) injected into new class defintion of cog.

*class Gear*

*attr\_reader :chainring, :cog, :wheel* 🡨 Variables wrapped for future change

*def initialize(chainring, cog, wheel)* 🡨 wheel class passed as argument

*@chainring = chainring*

*@cog = cog*

*@wheel = wheel* 🡨 wheel class injected

*end*

*def gear\_inches*

*ratio \* wheel.diameter* 🡨 wheel object responds to .diameter method

*end*

*end*

Dependecy injection is only needed when there is a dependency on another class in the class definition.

Isolate Object Name Dependencies

For use when the dependency cannot be removed from the class definition, this is very similar to wrapping variables in class/method defintions. Using a basic method to define the dependency as itself allows them to be easier to find and limits any changes to one section of code, allowing for DRY.

It is also important to isolate the creation of any dependency from an immovable object name, this can be done by putting the creation in the class initialize section, limiting the dependency’s reach into the code and showing it clearly for any future modification. For example:

*def initialize(chainring, cog, rim, tire)*

*@chainring = chainring*

*@cog = cog*

*@wheel = Wheel.new(rim, tire)* 🡨 Dependency creation isolated

*end*

Isolate Vunerable External Messages

Isolate any outgoing external messages (methods) by wrapping them in their own methods, such as with wrapping variables. Therefore, if the name of the external messages are changed, there will only be one change required, instead of changing code throughout the class definition.

*def gear\_inches*

*foo = some\_intermediate\_result \* diameter*

*end*

*def diameter*

*wheel.diameter* 🡨 external message isolated by wrapping into its own variable

*end*

Remove Argument Order Dependency

Many methods not only require arguments, but they also require that they are entered in a specific order, such as: ‘object.method(arg1, arg2)’ is not the same as ‘object.method(arg2, arg1)’. Therefore, the trick to avoid this and the problems it may cause in the future if the arguments list or order is changed, is to use hashes as the argument input. Hashes provide various advantages:

* Remove dependency on order of arugments
* Hash keys give explict documentation on arguments contents

*class Gear*

*attr\_reader :chainring, :cog, :wheel*

*def initialize(args)* 🡨 arguments inputted as hash

*@chainring = args[:chainring]* 🡨 aguments called induvidudally by name

*@cog = args[:cog]*

*@wheel = args[:wheel]*

*end*

*end*

While using hashes for arguments does have advantages it does add verbose (additional lines of code) and add dependency on the names of the arguments in the hash. Therefore, this method should only be used in a situation where the code is likely to change often, such as framework which many people work on.

Defaults can be defned using ‘args.fetch(:key, <default-value>)’ so if no value for the key exists the default will be used instead. It is also possible to wrap the default values in a method and then merge the inputs so any changes are added:

*def initialize(args)*

*args = defaults.merge(args)* 🡨 merge inputs with defaults

*@chainring = args[:chainring]*

*... others*

*end*

*def defaults*

*{:chainring => 40, :cog => 18}* 🡨 defaults method

*end*

Isolate Multiparameter Initialization

When you do not own a method that still requires fixed order arguments, e.g. an external api, it will not be possible to change its inputs to a hash to give insurance on future changes. Therefore, to isolate the external method wrap it in a defined module called a factory, which isolates all external dependencies to one place.

The factory module’s only purpose is to create instances of another class, giving intermediary isolation between your code and external dependencies which may permate your code. Also being a module conveys the idea that it is not expected to have instances of it.

*module ExternalFramework* 🡨 External method you’re not able to change

*class Gear*

*attr\_reader :chainring, :cog, :wheel*

*def initialize(chainring, cog, wheel)*

*@chainring = chainring*

*@cog = cog*

*@wheel = wheel*

*end*

*end*

*end*

*module GearWrapper* 🡨 module used to isolate external method dependencies

*def self.gear(args)*

*ExternalFramework::Gear.new(args[:chainring],*

*args[:cog], args[:wheel])*

*end*

*end*

*GearWrapper.gear{* 🡨 Module used in own code making less dependent gear object

*:chainring => 52,*

*:cog => 11,*

*:wheel => Wheel.new(26, 1.5)).gear\_inches}*

Dependency Direction

It is always important to depend on code less likely to change than the one being written:

Most likely to change 🡪 Your code – Concrete (tightly coupled)

Your code – Abstract (loosly coupled)

Framework code

Least likely to change 🡪 Base Classes of Language

A screenshot of a cell phone

Description automatically generated

Abstract code – Disassociated from any specific instance

# Flexible Interfaces

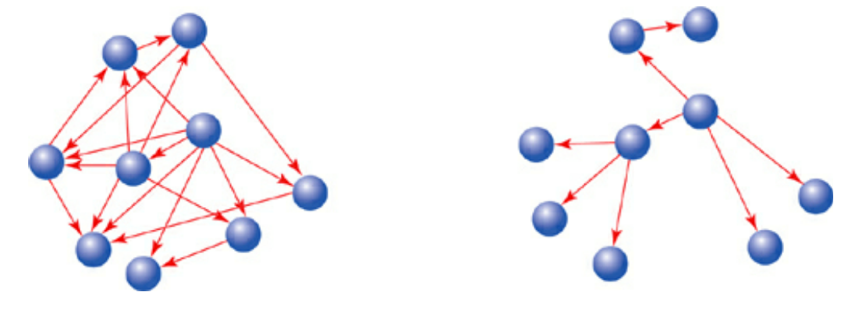
Messages are sent to and from objects via the objects’ public interfaces, the list of messages(methods) available to be sent to the object. The methods that comprise the public interface:

* Reval class responsibility
* Are expected to be invoked by other
* Should be stable / are safe for others to depend on
* Are thoroughly documented in tests (100% cover of public methods)

The remaining methods are private and they

* Handle implementation details
* No expected to be invoked by others
* Can be unstable / are unsafe for other objects to depend on
* Are not documented

If messages are designed to be sent randomly between objects whenever they are needed, modification to the interface of one object can cause throughout the program, making it very costly to implement. Well designed interfaces reduce dependencies and make message paths clearer.



The design goal is always to retain maximum future flexibility, while only writing enough code to meet todays requirements. Designing objects like pluggable components, allows for easy reuse and lower cost modifications.

To start defining an objects interface, first it is important to have an idea of what classes (objects) will be in the program. Generally the nouns from client requirements will be the *domain objects*, each having *data* and *behaviour*. List these out to give a basic outline of what objects the program will require, however more importantly is what messages will be sent between them. To form an intention of the messages needed to satisfy the client requirements it is good to create sequence diagrams.

Sequence Diagrams

Sequence diagrams are defined in the Unified Modelling Language (UML) *add information from UML*. Although agile design doesn’t require the creation an maintenance of UML artifacts, they processes defined by UML sequence diagrams provide a great way to communicate OOD.

Showing the messages which are likely to pass between objects helps define who should be responsible for each action within the program and respond to it.

What vs. How - Messages on the public interface of an object should be asking what they want done, instead of how. This reduces responsibility on the object asking, and as the implementation of behaviour is likely to change during development of programs and therefore is less stable, so should be kept off the public interface.

Context is what an object knows about others in order to implement it behaviour, ideally objects should be made completely independent of its context. The sequence diagram on the left contains context to prepare bicyles during preparing trip, however this context can be removed. Since the Trip object know the mechanic object is a preparer, it doesn’t need to know what it prepares. By sending the message to ‘prepare\_trip(self)’ the preparer (mechanic object) can then ask the sender (self/Trip) to return the objects it is desgined to prepare, the bicycles. As the context of prepare\_trip has been reduced, it could easily be reused for other preparers later on in the class definition.

