# Summary

<https://refactoring.guru/design-patterns/behavioral-patterns>

| **Design Pattern** | **Example** | **When to Use** | **Comments** |
| --- | --- | --- | --- |
| Strategy | Duck has a Quack Behaviour | Separate a changing behaviour (algorithm) from the object (class).  Define a family of algorithms and make them interchangeable.  Use composition to plug in the Strategies into one class. | The concrete class (Duck) has receive an interface for the behaviour.  Depending on the concrete class that is passed in real time, this behaviour can change. |
| Template | Every Packet type follow the same processing steps (verify, decrypt, etc). But every step is different depending on the packet type (e.g. different encryption types/length etc) | When all classes are doing the same steps but the steps themselves are different between the classes.  Use polymorphic inheritance to implement. | Very similar to Strategy.  Use Strategy when you want to re-use the same behaviour in different combinations.  Use Template when every sub-class implements a unique algorithm without code duplications between multiple classes. |
| Observer | newspaper subscription service | Defines a one-to-many dependency between objects so that when one object change state – all its dependents are notified and updated automatically. | The observers should receive the ‘Update’ call and then pull all the data they need from the subject.  The observed (subject) has a list of observers that implement the update function (From Observer interface). The Observers call the subject’s Register/Unregister functions to be added/removed from this list.  The subject (observed) doesn’t need to know the observer and vice versa (except for the interface functions register/unregister/update) |
| Decorator | Wrapper – coffee house: whip(mocha(dark roast)) | Attach additional responsibilities to specific objects dynamically and transparently to the client.  Allow supporting large combinations of responsibilities without class explosion.  Provide alternative to subclassing for extending functionality. | Caution: Decorators can lead to many small objects in our design and overuse can be complex.  Decorator classes are the same type as the components they decorate, either through inheritance or interface implementation.  Decorators change the behaviour of their components by adding new functionality before and/or after (or even in place of) method calls to the component. |

| Simple Factory | Pizza Store: add new Pizza(type)  Doors Factory (e.g carpenter) | | To encapsulate the decision on which concrete objects to create – a dedicated class decides which object to create and return the new objects.  Can be for decoupling or just because the code can’t predict in advance which object it will have to create. | Not really a formal design pattern but very useful | |
| --- | --- | --- | --- | --- | --- |
| Factory Method | Pizza Stores Franchise: createPizza (NYStyleCheese, ChicagoStylePepperoni etc.)  Decide which doors factory to use (carpenter, plastic doors, steel doors etc) | | When the class can’t know which objects to create and we need to move this decision to the subclasses.  Abstract interface for creating one product.  To Support DIP (Dependency Inversion Principle) | Relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects. | |
| Abstract Factory | | Pizza Stores – Ingredients factories (create dough, sauce etc.) | Abstract interface to create a family of products that should be defined together (related or dependent on each other). | Relies on objects composition: object creation is implemented in methods exposed in the factory interface. | |
| Composition | | Organization hierarchy | When you want to use recursion on collections of objects and/or hierarchy of objects (example: find out how many employees an organisation has by recursing over its organizational tree). |  | |
| Model View Controller | |  | To reduce dependencies between these parts. Very useful for web-dev  TODO: Calculator app with MVC in .Net. See <https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design>  <https://realpython.com/blog/python/the-model-view-controller-mvc-paradigm-summarized-with-legos/> | Model – Observer pattern – allows the view and controller to register for change event.  Controller – Strategy (of the view)– allows using different controllers (behaviours)  View – uses the controllers as his strategy. uses Composite internally (collection of smaller components e.g. text boxes, labels etc) | |
| Circuit Breaker | |  | To protect protect parts of a system from failures in other parts, without letting the failures snow ball.  An example is protecting your service from a dependency service failure  <https://martinfowler.com/bliki/CircuitBreaker.html>  If the circuit breaker triggers, you will return a failure status without calling the failed service (which will cause even worse latencies) | <https://github.com/Netflix/Hystrix/> |
| Orchestrator | | DeliverDestinationsOrchestrator | To start many threads working in parallel and check their results |  |
| Proxy | | Micro-services proxies | Implement the same interface as the proxied class but add operations before/after calling the original object.  Controls and manager access to the object they are protecting.  <https://www.geeksforgeeks.org/proxy-design-pattern/> | Allow isolating issues with the original object (e.g. throttling, exceptions etc) and allow just-in-time initiation that is transparent to the clients code. |
| Builder | | Sparse-objects builders | * Initiate complex objects with a lot of optional parameters, step by step, using only some of them. * Support creating different representations of some product (e.g. sport vs family cars), each requiring different parameters * Helpful for constructing Composite Trees and other complex objects.   <https://refactoring.guru/design-patterns/builder> | If you need to call the builder steps in particular order, consider creating a director class to control this.  Example: Car sportCar = carBuilder  .setCarType(type)  .setSeats(2)  .setGPSNavigator(gps)  .build(); |

<https://www.codeproject.com/Articles/29036/Patterns-in-Real-Life>

# Iterators

There are two rules for making container-based code general and efficient:

* Never pass containers into a function. Pass iterators instead.
* Never return containers. Return -- or pass -- iterators instead.

In order to define an iterator for a new class, you have to define:

* The **iterator types** for that container, e.g., iterator and const\_iterator, e.g., vector<int>::iterator
* the **begin/end methods** for that container, i.e., begin() and end()

## Iterators from Nested Containers

Fortunately, the most common way to define the iterator types and begin/end methods is also the simplest. If your container uses an STL container in an internal data member to hold its elements, all you need to do is delegate the types and methods to the underlying container.

For example, suppose you have a class CourseList that is a list of the students taking a course, and internally you use a vector of Student to store the course list. Then all you need to do is:

class CourseList {

private:

typedef vector<Student> StudentList;

StudentList students;

public:

typedef StudentList::iterator iterator;

typedef StudentList::const\_iterator const\_iterator;

iterator begin() { return students.begin(); }

iterator end() { return students.end(); }

...

};

In other words, all we do is define

CourseList::iterator and CourseList::const\_iterator to be whatever our container's iterator types are, and

begin() and end() to return whatever our containers's begin() and end() return

This is a common programming pattern called **delegation**, because we delegate the work to another class.

The StudentList typedef is optional, but it makes it easy to switch containers with just one line change.

## Iterators from Pointers

C pointers are legal iterators, so if your internal container is a simple C array, then all you need to do is return the pointers. Thus if we used an array for our StudentList, the code would look like this:

class CourseList {

private:

typedef Student StudentList[100];

StudentList students;

public:

typedef Student \* iterator;

typedef const Student \* const\_iterator;

iterator begin() { return &students[0]; }

iterator end() { return &students[100]; }

...

};

These two cases, iterators from internal STL containers, and iterators from pointers, cover 90% of the cases you need to worry about.

## Defining Iterators for New Containers: Overview

Our last case of when you want to define an iterator is when you have a new kind of container and it needs an iterator.

The STL does not use class hierarchies and inheritance. Therefore, when defining an iterator, there is no iterator superclass to start from. Something qualifies as an iterator as long as it defines some or all of the operators described on [the iterators page](https://www.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterators.html).

For this example, we're only going to describe and define the most basic operations.

* operator\*() -- the dereferencing operator
* operator++() -- the incrementing operator
* operator!=() -- the inequality operator, needed for all those "<TT>while ( begin != end ) ...</TT>" loops

Suppose we wanted to define product() to multiply together the numbers in a container.

We can immediately reject any definition like this:

double product( vector<double> v ) ...

because it's

* not general; it only works for vectors
* inefficient; it copies the container

But we could define it like this:

template <class Container>

double product( const Container & container )

{

Container::iterator i = container.begin();

double prod = 1;

while ( i != container.end() ) prod \*= \*i++;

return prod;

}

This definition seems general. It works for any STL container, e.g.,

vector<double> nums;

...

return product( nums );

Unfortunately, it won't work with regular arrays, e.g.,

double nums[] = { 1.2, 3.0, 3.5, 2.8 };

return product( nums );

because there are no begin() or end() methods for regular C-style arrays. Furthermore, it doesn't let us calculate the product of a subrange of the container.

The following definition is clearly more general:

template <class Iter>

double product( Iter start, Iter stop )

{

double prod = 1;

while ( start != stop ) prod \*= \*start++;

return prod;

}

This works fine with regular arrays:

double nums[] = { 1.2, 3.0, 3.5, 2.8 };

return product( nums, nums + 4 );

as well as with STL containers and subranges.

# Strategy

Defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that uses it.

public class Duck

{

public Duck(IStrategyCall call)

{

call\_ = call;

}

public string Call()

{

return call\_.Call();

}

//---------------------------------------

// Member variables:

//---------------------------------------

private IStrategyCall call\_;

}

public interface IStrategyCall

{

string Call();

}

public class IStrategyCall\_Quack : IStrategyCall

{

public string Call()

{

return "Quack";

}

}

public class IStrategyCall\_Qual : IStrategyCall

{

public string Call()

{

return "Qual";

}

}

# Observer

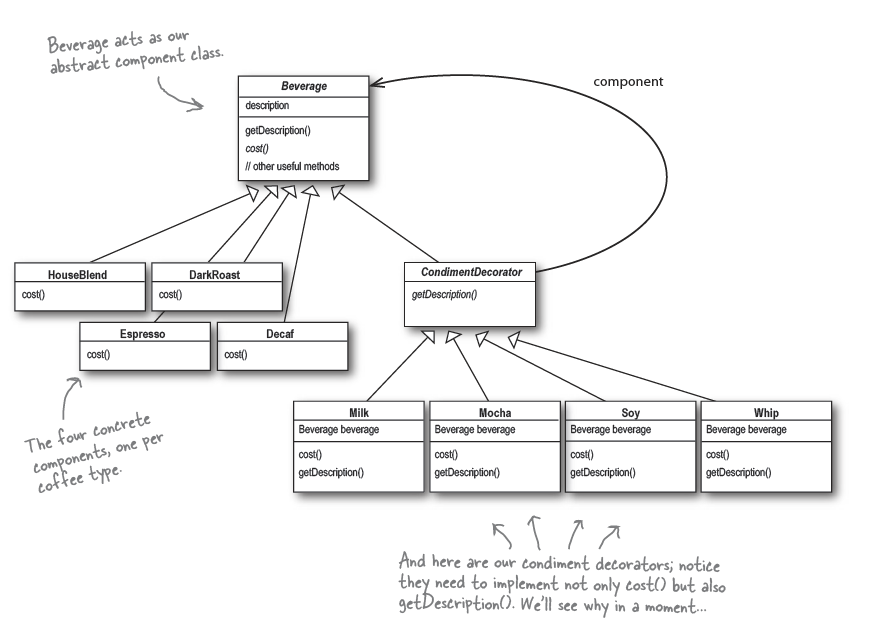
Defines a one-to-many dependency between objects so that when one object change state, all its dependents are notified and updated automatically.

# Decorator

The Decorator Pattern attaches additional responsibilities to an object dynamically.

Decorators provide a flexible alternative to subclassing for extending functionality.

This helps maintain the Open Close principle: extending classses’ functionality without changing them.

* Decorators have the same supertype as the objects they decorate ( to achieve class matching).
* You can use one or more decorators to wrap an object.
* Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
* The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
* Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

## In Practice

#### Decorate an Existing Class

Find the class’s interface/abstract class and inherit from it to create your decorator parent class.

## Notes

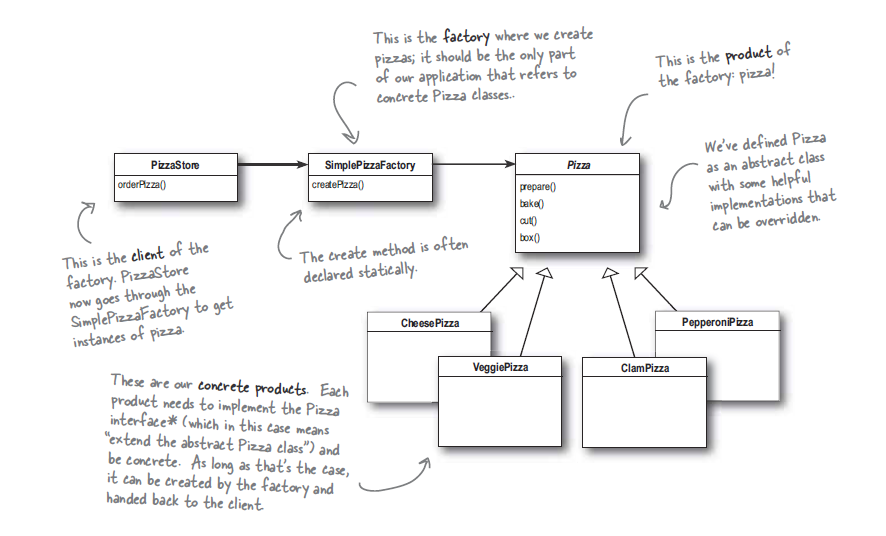
* If you have code that relies on the concrete component’s type, decorators will break that code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you’ll want to rethink your application design and your use of decorators.  
  For example: a discount on one type of coffee (e.g. Decaf) only.
* Decorators are meant to add behaviour to the object they wrap. When you need to peek at multiple layers into the decorator chain, you are starting to push the decorator beyond its true intent. Nevertheless, such things are possible. Imagine a CondimentPrettyPrint decorator that parses the final description and can print “Mocha, Whip, Mocha” as “Whip, Double Mocha.” Note that getDecription() could return an ArrayList of descriptions to make this easier.

# Factory

## Simple Factory

The factory creates the objects from the required type. All concrete types inherit from the same abstract class.

The client code which request the object doesn’t have to know about the concrete classes (e.g. VeggiePizza) , only about the abstract class (Pizza) and about the factory.



## Abstract Factory

The factory ‘Simple Pizza Factory’ is now abstract, together with its factory method ‘createPizza’.

1. The Pizza Store creates a factory for one type of pizza-factory:

PizzaStore nyPizzaStore = new NYPizzaStore();

1. The NYPizzaStore (concrete factory), calls the orderPizza function which calls the factory method to create the pizza, and then calls all the action-functions on the pizza:

nyPizzaStore.orderPizza(“cheese”);

PizzaStore::orderPizza(“pizzaType”)

{

Pizza pizza = createPizza(“cheese”);

pizza.prepare();

pizza.bake();

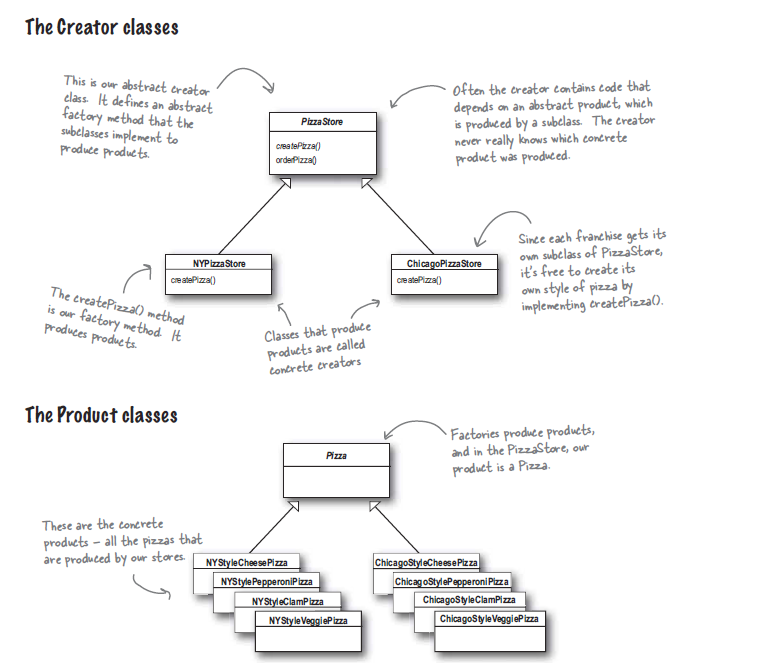
pizza.cut();

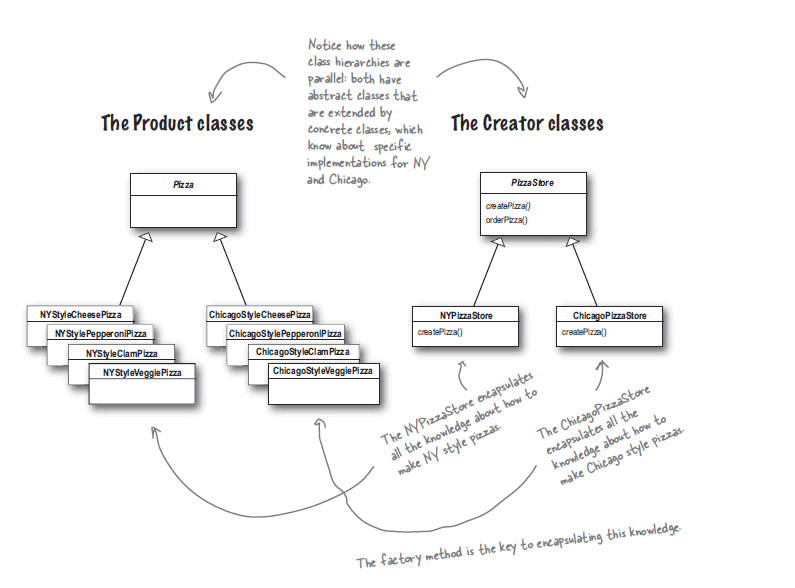
pizza.box();

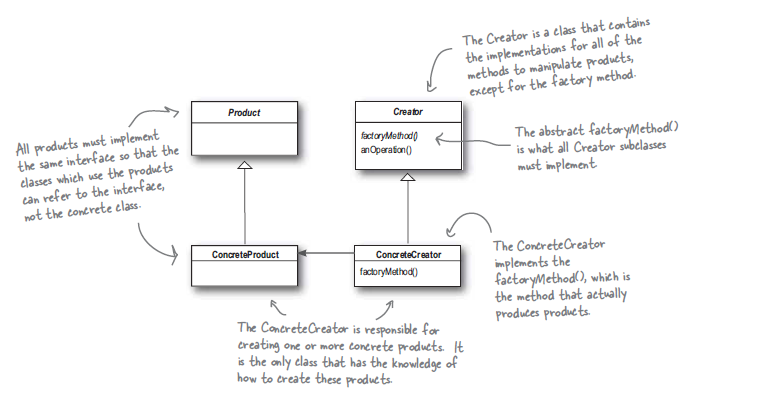
}

1. createPizza is the pizza factory method that is implemented in each PizzaStore subclass (e.g. NYPizzaStore, ChicagoPizzaStore) and it’s the factory that create the concrete pizza types (e.g. NYCheesePizza, ChicagoVegiePizza):

Pizza pizza = createPizza(“cheese”)







# Model-View-Controller (Box-Photo-Camera) – Awesome Explanation:

**“I explained MVC to my Dad like this:**

MVC (Model, View, Controller) is a pattern for organising code in an application to improve maintainability.

Imagine a photographer with his camera in a studio. A customer asks him to take a photo of a box.

The box is the *model*, the photographer is the *controller* and the camera is the *view*.

Because the box does not *know* about the camera or the photographer, it is completely independent. This separation allows the photographer to walk around the box and point the camera at any angle to get the shot/view that he wants.

Non-MVC architectures tend to be tightly integrated together. If the box, the controller and the camera were one-and-the-same-object then, we would have to pull apart and then re-build both the box *and* the camera each time we wanted to get a new view. Also, taking the photo would always be like trying to take a selfie - and that's not always very easy.

MVC is often seen in web applications, where **the view is the actual HTML page**, and **the controller is the code that gathers dynamic data and generates the content within the HTML**. Finally, the model is represented by the actual content, usually stored in a database or XML files, and the business rules that transform that content based on user actions.

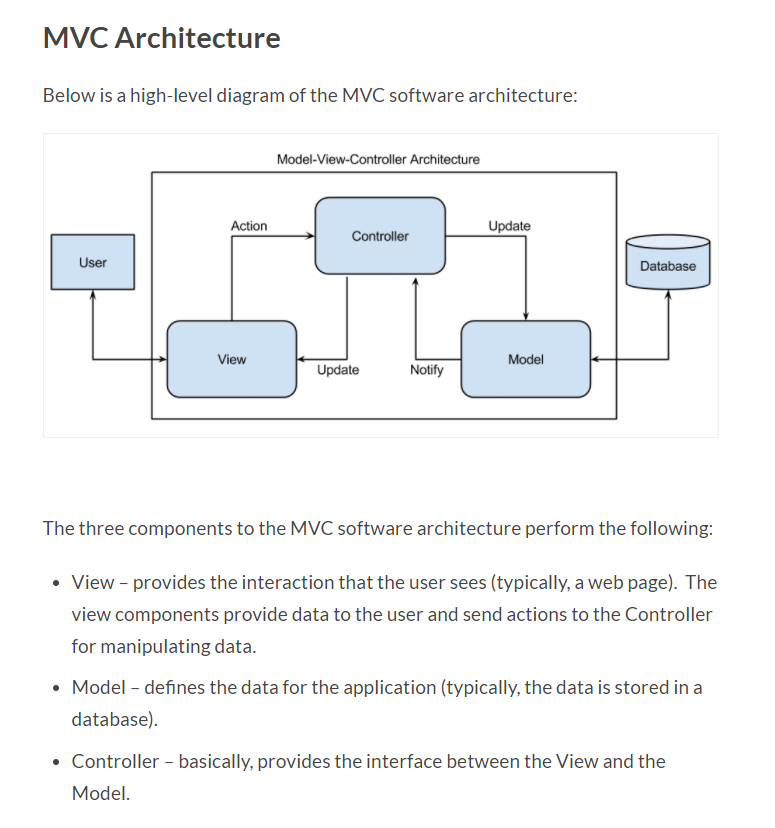
Though MVC comes in different flavors, control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g. presses a button).
2. A controller handles the input event from the user interface, often via a registered handler or callback.
3. The controller notifies the model of the user action, possibly resulting in a change in the model's state. (e.g. controller updates user's shopping cart).
4. A view uses the model (indirectly) to generate an appropriate user interface (e.g. the view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view.
5. The user interface waits for further user interactions, which begins the cycle anew.

By decoupling models and views, MVC helps to reduce the complexity in architectural design, and to increase flexibility and reuse.

How to know where to split the View from the Controller from the Model:

* The Model should contain only business model and should not call any function in the Controller or View. You should be able to write unit tests that test all the model without using the Controller or View.
* The View is the user interface. It should include everything the user sees.
* Controller is the part that communicate with both View and Model.



## Model Objects

Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. For example, a model object might represent a character in a game or a contact in an address book. A model object can have to-one and to-many relationships with other model objects, and so sometimes the model layer of an application effectively is one or more object graphs. Much of the data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects after the data is loaded into the application. Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains. Ideally, a model object should have no explicit connection to the view objects that present its data and allow users to edit that data—it should not be concerned with user-interface and presentation issues.

**Communication**: User actions in the view layer that create or modify data are communicated through a controller object and result in the creation or updating of a model object. When a model object changes (for example, new data is received over a network connection), it notifies a controller object, which updates the appropriate view objects.

## View Objects

A view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application’s model objects and to enable the editing of that data. Despite this, view objects are typically decoupled from model objects in an MVC application.

Because you typically reuse and reconfigure them, view objects provide consistency between applications. Both the UIKit and AppKit frameworks provide collections of view classes, and Interface Builder offers dozens of view objects in its Library.

**Communication**: View objects learn about changes in model data through the application’s controller objects and communicate user-initiated changes—for example, text entered in a text field—through controller objects to an application’s model objects.

## Controller Objects

A controller object acts as an intermediary between one or more of an application’s view objects and one or more of its model objects. Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa. Controller objects can also perform setup and coordinating tasks for an application and manage the life cycles of other objects.

**Communication**: A controller object interprets user actions made in view objects and communicates new or changed data to the model layer. When model objects change, a controller object communicates that new model data to the view objects so that they can display it.

## MVC Frameworks

### MVC and Flask

A structure as simple as this would suffice:

1. Flaskapp/
2. - app.py (controllers)
3. - templates/ (views)
4. - **base**.html (minimum view)
5. - modules.py (**module**/db handling)
6. - **static**/
7. - **base**.css
8. - **base**.js

Outside of something like this, you're really stretching what a microframework is great at, which is minimalism. If I needed an mvc-like approach, I'd grab django or pylons. One of the best things about the larger web frameworks is how well tuned they are to building crud apps, and building them quickly.

# Command Pattern

Client (waitress)– creating the command object

Command (cook) – the command object that contains all the commands to execute on the receiver wrapped in an execute() method. The Command knows the Receiver.

Invoker (orderUp) – gets the command object from the client (the client calls invoker.setCommand() to set it)

Receiver ( restaurant’s customer ) – the object that the commands refer to (receiver.command1(); receiver.command2() etc)

Null Object

Macro Command

Questions:

* + - 1. What is the mapping between the restaurant and the command pattern?
      2. Is the undo just the off of every on button?
      3. How to implement this in Java?
      4. How to implement this in JS?
      5. How can we use it in our service?
      6. Is this what’s used in databases to make them transactional?
      7. What is the macro command?

# Design Principle

## Object Oriented Design

The 4 Pillars of OOP: A Pie:

1. **Abstraction** - **We take out unnecessary details and only focus on aspects that are necessary to that context or system under consideration**. Example: a class Person might need height and weight if it’s a medical app but if it’s a university student, it won’t need them and will need ATAR score instead. To perceive an entity in a system or context from a particular perspective.
2. **Polymorphism - the ability (in programming) to present the same interface for differing object types and to use them interchangeably**. So the ability to have a function that can take any object from a whole family of objects (e.g. Animals in Zoo) and run the same functionality on them (e.g. PrintName). Built on **interfaces**.  
   When classes implement different functionality but share the same interface, you can call methods on the base class and during runtime, the correct derived class’s method will be called depending on the class of the actual object.
3. **Inheritance** – the ability to **re-use the implementation of one class in another class**. For example: a Student is also a Person and a lot of the properties and functions of a Person will be re-used in the Student (e.g. Name, GetAge() etc).
4. **Encapsulation** – **hiding the implementation details of classes and exposing only the public operation they support**. For example: if I have a class of Products, I will not expose the internal data-structure I use to hold them (e.g. List, Array, HashTable). Instead I’ll expose and iterator to iterate over the products, GetProduct(by name) etc. This way if I change the internal implementation, the user doesn’t need to know about it.

## SOLID

### Single Responsibility Principle

* + A class should have only one responsibility and only one reason to change.
  + Every responsibility is a potential reason to change so if a class have more than one responsibility it has more reasons to change.
  + Help **reuse, more readable, maintainable** etc.
  + Example: **Person with an email property**. The Email should be its own class with email validation etc and not part of the Person class.
  + My example: L2G File System – had all the functionality in one module. This cause repeated called to the same function (virtualToPysicalAddress) that caused issues. When I separated into different modules in different layers, the flow went from top (audio file) to low (->file system ->Virtual Memory -> Bad blocks Manager->Flash Driver)

### Open Closed Principle:

* + A class should be open for extension and closed for modification.
  + Methods to implement this:
    - Template Method Pattern
    - Strategy Pattern
    - Decorator Pattern
    - Hollywood Principle

**NOTE:** Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard *to understand code.*

* + Example: **Decorators for Coffee** (whip(mocha(dark roast)))
  + My example: Strategy + Factory pattern – **Modems in T6** - My modem had all the functionality implemented but I had a virtual table for all the changeable functionality so that I can plug the right behaviour (strategy) depending on the current modem. So I didn’t need to change the functionality (e.g. check registration) when supporting a new modem, just add a new behaviour (e.g. SendSMS, TurnOff, GetRegistrationStatus).

### Liskov Substitution Principle:

**~ Design By Contract**

**Functions that use pointers/references to the base class must be able to receive an object of the derived class without knowing it and without side effects**. (Robert C. Martine on Liskov Substitution Principle).

Every function defines explicitly or implicitly:

* + - 1. Pre-conditions: what the user must do before calling the function
      2. Post-conditions: what the function guarantee to do.

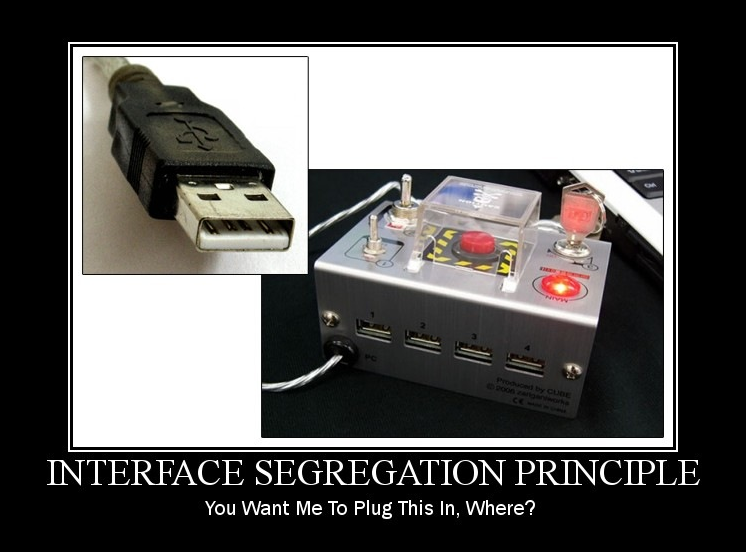
In Derived class I can:

* + - 1. Weaken the pre-condition of all the functions I inherit from the base class (add **or** to the pre-conditions), or leave unchanged.
      2. Strengthen the post-conditions of all the functions I inherit from the base class (add **and** to the base conditions), or leave unchanged.

Every derived class is a type of the base class as far as the contract is concerned.  
However, it does not imply that it should be a type of the base class in a natural language conceptual way.  
**A Square is NOT a type of Rectangular!!** setHeight and setLength in rectangular doesn’t make much sense in the square and cause side effects when sent to a square (set all sides instead).

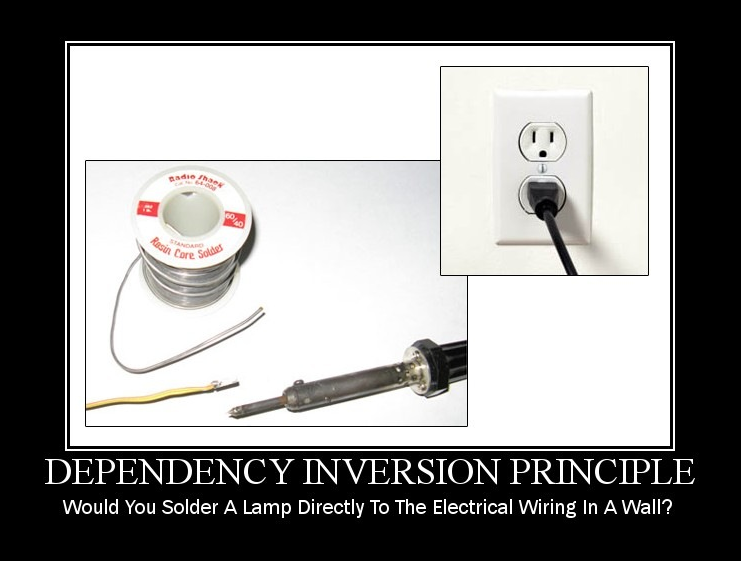
* Is-a-type-of is not true. Use can-substitute-a instead.

### Interface Segregation Principle:



* + **No client should be forced to depend on methods that they don’t need**.
  + If you implement an interface and throw new “NotImplementedException”, your code smells.
  + Example: instead of ISmartDevice with Print, Fax and Scan - define IPrinter, IFax and IScanner for each and then you can support both all-in-one-printers and simple-printer-only printers.

### Dependency Inversion Principle:



That’s a bad example – a better one would be – depend on a uniform electrical connector instead of specific country connector.

* Depend on abstractions. Not on concrete classes.
* To help you follow the dependency Inversion Principle, **where your code is likely to change**, aim to:
  + **Depend upon abstractions (either interfaces or abstract classes). Don’t depend on concrete classes.**
  + No variable should hold a reference to a concrete class (**use factory instead of ‘new’**)
  + No method should override an implemented method of any of its base classes (the base class should be abstract with the abstract methods defining the common interface for all concrete classes).
* Example: Use IEnumerable<T> instead of List<T> if you just need to enumerate over the collection. You should know or care if the developer of the collection decides to change its type (e.g. from List to Stack). For example IPrintPeople – should accept IEnumerable<Person> and not List<Person>
* My example: I’ve used it many times while writing web-code in Java (Amazon) and in C#. Not used often in embedded.

## Analyse and Encapsulate

* Analyse, identify and encapsulate the aspects of your application that vary and separate them from what stays the same.

## Decoupling

* Strive for loosely coupled designs between objects that interact.

## Cohesion

Measure how closely a class or a module support a single purpose or responsibility (see Single Responsibility Principle).  
We say that a class has high cohesion when it’s designed around a set of closely-related functions. This also means that it’s more likely to change only if these functions need to change.

We say that a class has low cohesion when it’s designed around a set of unrelated functions.

## Interfaces

* Program to an interface, not an implementation - dependency inversion principle.

## Composition vs Inheritance

* **Favour composition over inheritance**  
  *Prefer composition over inheritance as it is more malleable / easy to modify later, but do not use a compose-always approach.* With composition, it's easy to change behavior on the fly with Dependency Injection / Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type. So the goose is more or less cooked once you derive from TypeA.

My acid test for the above is:

* + Does TypeB want **to expose the complete interface** (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates **Inheritance**.

e.g. A Cessna biplane will expose the complete interface of an airplane, if not more. So that makes it fit to derive from Airplane.

* + Does TypeB want **only some/part** of the behavior exposed by TypeA? Indicates need for **Composition.**

e.g. A Bird may need only the fly behavior of an Airplane. In this case, it makes sense to extract it out as an interface / class / both and make it a member of both classes.

**Update:** Just came back to my answer and it seems now that it is incomplete without a specific mention of Barbara Liskov's [Liskov Substitution Principle](http://en.wikipedia.org/wiki/Liskov_substitution_principle) as a test for 'Should I be inheriting from this type?'

* Composition vs. Aggregation:
  + Composition: Building objects from other, smaller objects
  + Aggregation: bringing together different objects that have different lifetimes.

## Dependency Injection (DI):

* + **providing the objects that an object needs (its dependencies) instead of having it construct them itself. You make it somebody else's problem (context code) to construct the dependencies.**  
    This **reduced the coupling** of the code and **assist in unit testing**.  
    Common methods for DI include:
    - through the constructor:  
      instead of:

public SomeClass() {

myObject = Factory.getObject();

}

You can write:

public SomeClass (MyClass myObject) {

this.myObject = myObject;

}

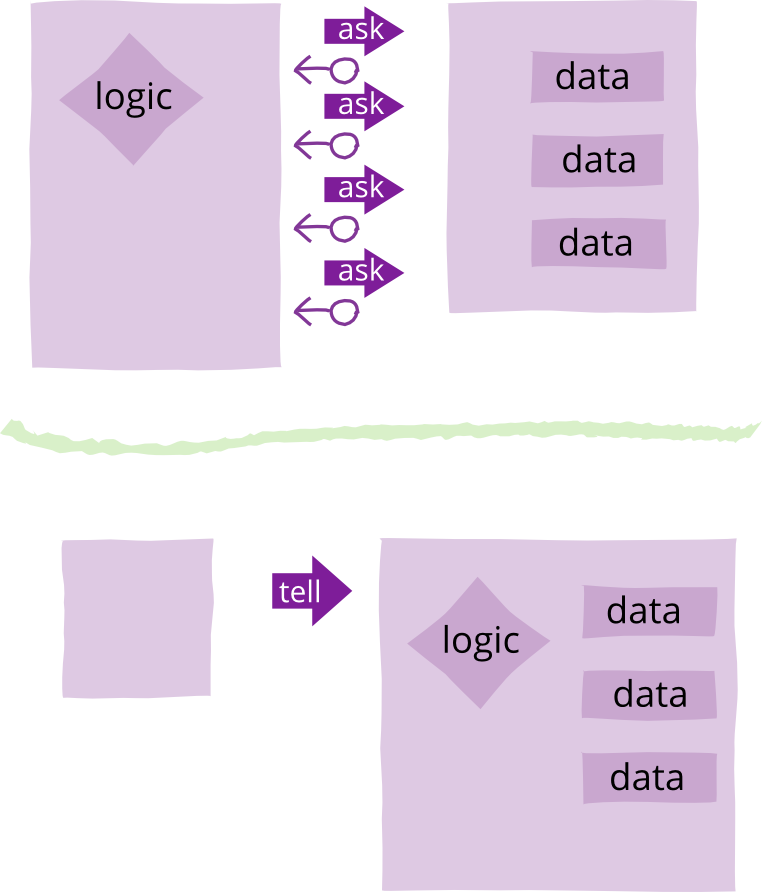
* + - Using setters (with an empty constructor)

## Tell Don’t Ask

**Law of Demeter** – Talk only to Friends. Don’t Talk to strangers:

* An object should avoid invoking methods of a member object returned by another methods.
* a method *m* of an object *O* may only invoke the methods of the following kinds of objects:[[2]](https://en.wikipedia.org/wiki/Law_of_Demeter#cite_note-2)
  + itself
  + *m*'s parameters
  + Any objects created/instantiated within *m*
  + *O*'s direct component objects
  + A global variable, accessible by *O*, in the scope of *m*
* The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.
* Note: the law of Demeter applies to objects. Not to data structures (see code quality document on the difference between the two)!

**Tell-Don't-Ask** is a principle that helps people remember that object-orientation is about bundling data with the functions that operate on that data. It reminds us **that rather than asking an object for data and acting on that data, we should instead tell an object what to do**. This encourages to move behavior into an object to go with the data.



## Code Complexity

A writer usually can’t see that their code is complex because they are deep in the details.

If others say that the code is complex, it is.

### Symptoms of complexity:

* Change amplification -  
  seemingly small code changes require a lot of changes in different places in the code.
* Cognitive load -  
  How much a developer needs to know about the system in order to complete a task? If the change is a few lines of code but it’s hard to know what they are – it might be better to have more lines of code that are more expressive.  
  Example: global vars, too many functions in the API, dependencies between modules etc.
* Unknown unknowns -  
  All the things we don’t know that we need to know. When there are unknown unknowns, we get surprised by things not working as we thought they would and by unexpected bugs.

### Causes of complexity

* **Dependencies** => leads to change amplification and cognitive load  
  When a code can’t be changed in isolation and in order to change it, you need to change or understand other code  
  Goal: **reduce** dependencies and make them **as simple and obvious as possible**Note: we don’t want to (and can’t) remove all dependencies (every function call creates a dependency). We just want to simplify and contain them.
* **Obscurity** => leads to Cognitive load and unknown unknows  
  When important information is not obvious (e.g. inexpressive variable names, inconsistent use of variables for different things etc).   
  Goal: **Make the implicit explicit, self-documenting code**
* Goal: **Zero tolerance** for complexity creep!

# Building Blocks

## Interfaces

Interfaces as guardrails: <https://www.codingblocks.net/programming/interfaces-as-guardrails/>

Interfaces includes:

* + - 1. Formal part - the public methods and variables it exposes
      2. Informal part - all the hidden dependencies that must be met before/after calling the methods, methods’ side effects etc.

### The Dark Side of Interfaces

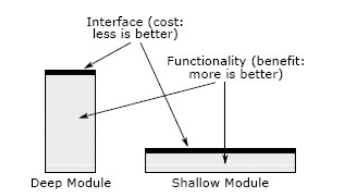
Closer look at a few of the pit falls and confusion around explicit interfaces.  
<https://www.codingblocks.net/programming/the-dark-side-of-interfaces/>

## Modules

Modules can be classes, subsystems, services etc.

Modules has two distinct parts:

* Interface - what people working on other modules need to know about this module in order to work with it.
* Implementation - what devs working on this module need to know in order to work within it. Devs working on other modules shouldn’t need to know the implementation details of other modules they call.

The best modules are deep modules:  


* They provide powerful **functionality**.   
  This is the **depth** of the module and represent the **benefit** of using this module.
* Their **interface** is simple. It contains all the parameters/methods that it needs but without any redundant details (that will over-complicate it).   
  This is the **width** of the module and represent of the **cost** (complexity) of using this module.

As a general rule - the smaller the module is, the less it contribute to simplify the system ( because it’s relatively shallow compared with its interface).  
Therefore, it is better to **aspire toward big modules** (as long as they are not overly complex in themselves) and not toward a lot of tiny modules.