oComputer Science **3307** – **Object Oriented Design** – Course Notes

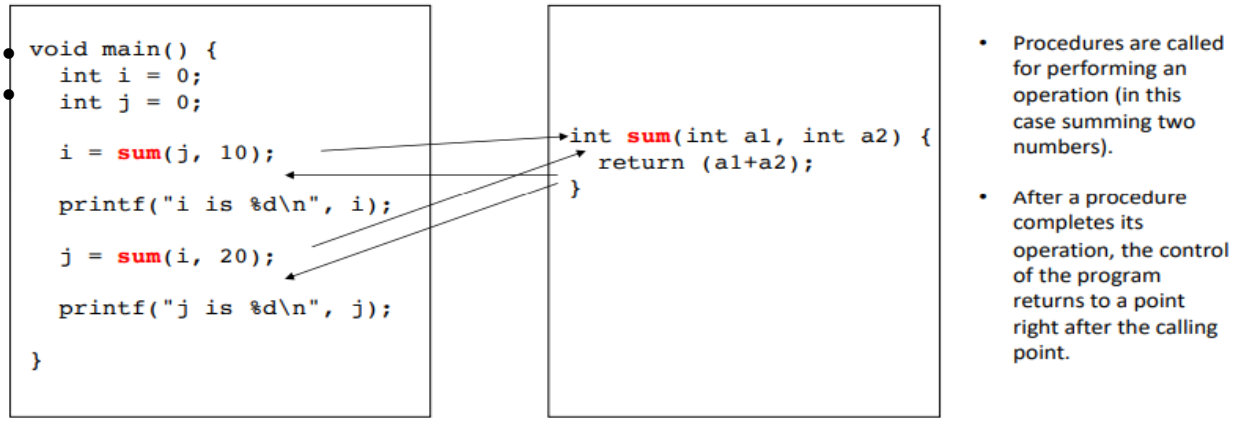
_____ (**Lecture #1** – Course Introduction) _____

- N/A

_____ (**Lecture #2** – Key Concepts) _____

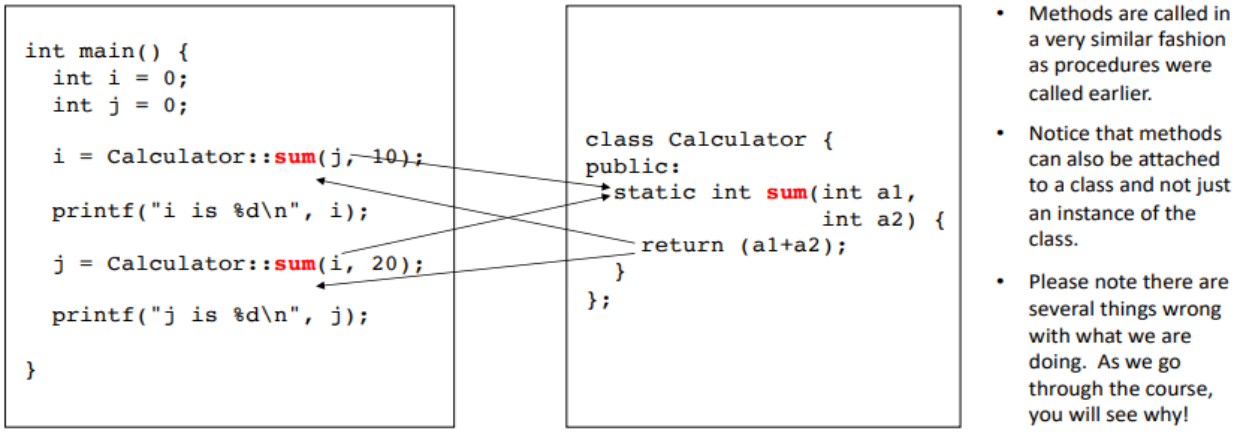
- Procedural/Imperative Programming = programming oriented around procedures (functions, routines and subroutines) which accept data, usually as parameters.
    - Uses blocks and scoping rules that non structured imperative languages don't.

- *Procedural Languages:*
    - C, Basic, Fortran, Pascal, Go, Python (can be).



- Object-Oriented Programming = structured around objects (which are instances of classes).
    - Objects accept data from other classes in order to perform their actions.
    - Objects encapsulate data and functionality (known as data members, and member functions). Key difference between Procedural and Object-Oriented is Object-Oriented encapsulates both data and functionality, Procedural has both, but separately.

- *Object-Oriented Languages:*
    - Java, C++, C#, Objective-C, Swift.



- *Advantages of Object-Oriented Programming:*
    - Better modeling of real world elements.
    - Better application of software engineering principles.
    - Better integration with other programs and libraries.
    - Information Shielding/Hiding = to keep the design and implementation of a class from the rest of the program.
    - Encapsulation.
    - Abstraction = modeling an entity (program, method, algorithm, data) so that only important characteristics are presented, and others are omitted.
    - Modularization = the design approach where a software application is implemented as a collection of independent subsystems communicating only the necessary data required for an operation.
        - This achieves low coupling,
        - Each subsystem performs a specific operation or contains a collection of closely related operations.
        - This achieves high cohesion.

- *Disadvantages of Object-Oriented Programming:*
    - Not the solution for all programming tasks.
        - Ex. Database applications work better with SQL or 4GLs.
        - Ex. Embedded Systems work better with Assembly.

_____ (**Lecture #3** – Intro to C++) _____

- *History of C:*
    - Became popular in the late 1970's.
    - While widely used, and available, C did not enforce users to use software engineering principles that were emerging, offering freedom.
    - 2 different versions of C emerged in the early 1980's. C with classes by Stourstrup. Objective-C by Love and Cox.

- History of Objective C:
    - While C++ became the most popular, Objective-C did fairly well. It was adopted by NeXT and then when Apple acquired NeXT, it used Objective-C primarily.
    - Objective-C was used to create Swift in 2014m although it wasn't very successful.

- *History of C++:*
    - In 1983, Mascitti referred to C with classes as C++ (due to the ++ increment) and the name stuck.
    - C++ has evolved in each version, expanding options, features and libraries.
    - Influenced the development of Python, Java, C#, Lua, Perl, PHP, and Rust.
    - Java specifically was created as an alternative to C++ with better portability and garbage collection.

- TIOBE Index in 2019 rated C++ as the 4th most popular language behind Java, C and Python.

(**Lecture #4** – C++ Basics - I)

- Note: most C code is valid for C++, and entire programs can be written in C with non class defined functions.

- One main function must exist, and only one.

C's Hello World is a C++ Program

```
#include <stdio.h>

/* Simple Hello World program. */

int main()
{
  printf("Hello World!\n");
}
```

A Slightly Better More C++-ish Hello World

```
#include <iostream>
using namespace std;

// Simple Hello World program.

int main()
{
  cout << "Hello World!" << endl;
}
```

- As C++ is a compiled language, its source code must be compiled and linked to produce an executable file.

- Building C++ Programs:
  - From a command line, building in one step would look like:

```
> g++ HelloWorld.cpp -o HelloWorld
> ./HelloWorld
```

  - Alternatively, you can build and keep the object files and do things in multiple steps like:

```
> g++ -c HelloWorld.cpp
> g++ HelloWorld.o -o HelloWorld
> ./HelloWorld
```

- #define constants = preprocessor definitions.

- Const constants = named constant declarations that cannot be changed once inputted.

- Constexpr constants = constant expressions evaluated at compile time.
  - Ex: a circle that will not be changed once made, but made with variables at run time.

- Block/Compound Statement = the term given to a collection of statements enclosed in { }.
  - Local variables within the block statement are only put onto the stack for its duration.

Accessing fields: Individual fields of a structure can be accessed using the "." syntax

```
Point p1;
p1.x = 3;
p1.y = 2;
```

Structures

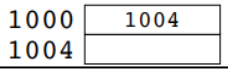
- Structures and classes are nearly equivalent, except structure members are default public, and class

Reference Operator &: When applied to a variable, & generates a pointer-to (or address-of) the variable

Example:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
```

```
int *p;  // suppose p is at address 1000 in memory
int c;   // and c is at address 1004 in memory
p = &c;  // stores the address of c in p as a pointer
```



You can make a pointer point at whatever another pointer is pointing at using the standard assignment operator (=)

Example:

```
int *p1; // p1 is a pointer to int (a declaration)
int *p2; // p2 is a pointer to int (a declaration)
int c;
p1 = &c; // causes p1 to point to c
p2 = p1; // causes p2 to point to c as well
```

Dereference Operator (*)

When we want to access or manipulate what a pointer is pointing at, we dereference the pointer first. Example:

```
int *p;  // p is a pointer to int (a declaration)
int c;
p = &c;  // causes p to point to c
*p = 10; // assigns the value of 10 to variable c through p
```

members are default publics.

- *Pointers:*

- *Passing by reference:*

```
void Two (int& x) {
   x = 2;
   cout << x << endl;
}

void One () {
  int y = 1;
  Two (y);
  cout << y << endl;
}
```

The output when function One() is called:

2
2

- Static = function is not visible outside of its file.

- Arrays:

Creating simple static arrays:

```
int foo1[5]; // array "foo1" with five int elements
             // that are not initialized

int foo2[5] = {1, 2, 3, 4, 5};  // array "foo2"
                                // initialized with five
                                // consecutive numbers
```

- You can allocate and deallocate array pointers using new and delete.

- *Processor Directives:*

To prevent multiple inclusion of the same header file (which can have bad consequences like duplicate definitions of various things) we can use preprocessor directives to create include guards:

```
#ifndef MYHEADER_H
#define MYHEADER_H
…
#endif
```

- Namespaces:
  - "using namespace std;" is important to not have to restate the type repeatedly.
  - Use of namespaces in header files is considered poor form.
  - Use of multiple namespaces with the given deceleration can be dangerous.

- Insertion Variable = <<
  - Ex. cout << "You are " << age << " years old" << endl;

- Extraction Variable = >>
  - Ex. cin >> storage_variable;

Insertion and Extraction

```
int x;  // variable declaration

cout << "Enter a number: "; // print a prompt
cin >> x; // read value from terminal into variable x
cout << "You entered: " << x << endl; // echo
```

```
string name;
cout << "Enter your name: ";
getline(cin, name);
cout << "You entered: " << name << endl;
```

If you were to enter "John Doe" at the prompt, the variable name

- How to read lines of a text file:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  string line;
  ifstream file;
  file.open("text.txt");
  while (getline(file, line)) {
     cout <<  line << endl;
  }
}
```

_____ (**Lecture #6** – C++ Classes) _____

- Public = accessible from anywhere where the class and its objects are visible.
- Protected = accessible from other members of the same class or derived classes.
- Private = accessible from only members of the same class.

Objects and Their Creation

```
class Triangle {
    private:
         float base, height, area;
    public:
       void setBase(float val) {
           base = val; }

       float getBase() {
            return base; }

       void setHeight(float val) {
            height = val; }

       float getHeight() {
             return height; }

       void calculateArea() {
             area = (base*height / 2);}
};
```

```
int main() {

   Triangle *tr1 = new Triangle;
   Triangle tr2;

    tr1->setBase(2.0);
    tr1->setHeight(3.0);
    tr1->calculateArea();

    tr2.setBase(1.0);
    tr2.setHeight(2.0);
    tr2.calculateArea();

}
```

- Objects created on the stack are destroyed when the function they were created in returns.
- Dynamically created objects must be destroyed manually with destroy.
- Constructors are not strictly required, but strongly recommended.
- Copy Constructors can be used to initialize an object from another.
    - A compiler will typically give a copy constructor implicitly if not declared or told not to.
- Deconstructors are not strictly required, but strongly recommended.

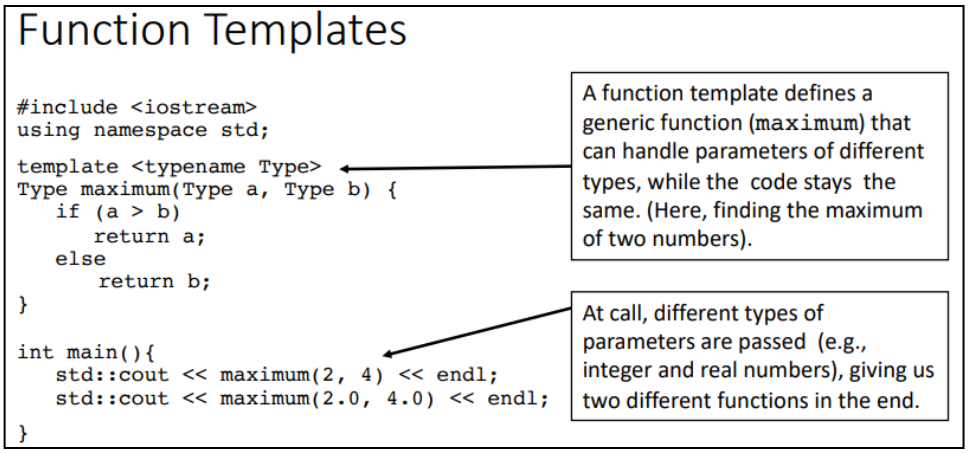
```
class Point {
   private:
        int x;
        int y;
   public:
        ...
        // copy constructor
        Point(const Point &p) {
            x = p.x;
            y = p.y;
        }
};
```

- Inheritance:
  - A subclass is made with the form:
    - class derived-class: access-specifier base-class
    - Where the access-specifier is – public, protected, or private.
    - Private is the default if unspecified.
    - Normally you want public for inheritance to be done.

- Private/Protected members of a class can be accessed by classes with the friend keyword from both classes mutually declaring friendship.

_____ (**Lecture #7** – C++ Advanced Concepts) _____

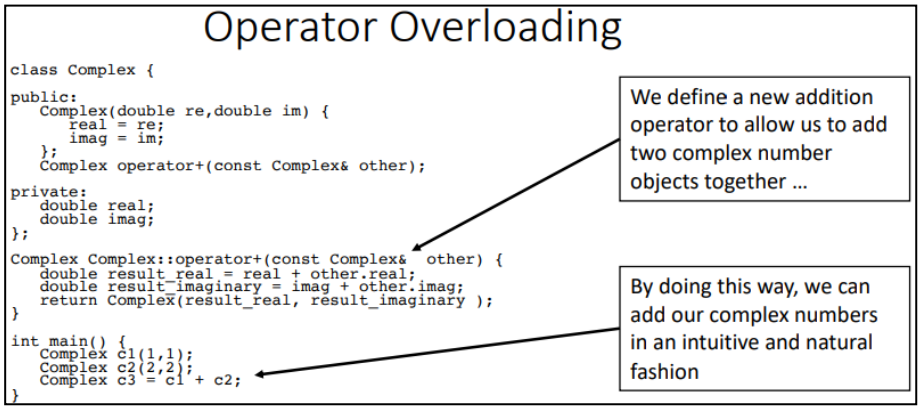
- Templates = a generic way of writing functions and classes capable of operating on multiple data types without duplicating the code.



- You can also create templates with multiple typenames.

- Standard Template Library (STL) = C++ library that provides a collection of general purpose template classes and functions, to be used whenever feasible instead of re-implementing them.

- You can overload operators, as in essence they are functions with parameters and returns.



The following operators can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

These operators, however, can't be overloaded:

::	.*	.	?:

- Polymorphism = "many forms", in C++ refers to sub-type polymorphism where we can use derived classes through base class pointer and references.

```
#include <iostream>
using namespace std;

class Person {
public:
   void sayHello() {
      cout << "I am a Person." << endl;
   }
};

class Student: public Person {
public:
   void sayHello() {
      cout << "I am a Student." << endl;
   }
};

int main() {
   Person *p;
   Student *s = new Student();

   s->sayHello();
   p = s;
   p->sayHello();
}
```

Executing this code gives us:

```
I am a Student.
I am a Person.
```

  - The "virtual" header keyword = is used to enable dynamic dispatching for determining which class to use at runtime, increasing flexibility but reducing performance.
    - For readability, continue adding virtual to all descendent classes of the virtual class, even though it's automatically assumed by the compiler.
    - You can also make pure virtual methods (abstract methods without implementation) that force derived classes to implement particular methods.

- "this" pointer = is an implicit parameter to all member functions of a specific function/object.
  - Ex. this -> width = width_param;

- Static members = can be accessed without requiring instantiation of an object in advance, and belong to the class itself, instead of just individual members or function of the class.
  - No storage is allocated for it from just a declaration, however the class knows it exists if referenced anywhere.

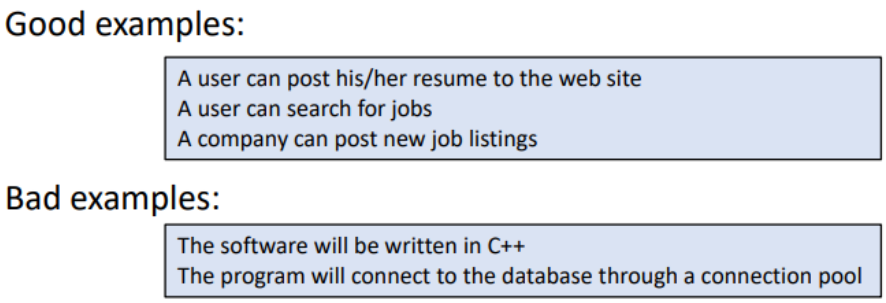
- You can put default values within a function's parameters, only used if no parameter is specified.
  - But note: if one value has a default value, all parameters must have one for that function.

(**Lecture #8** – User Stories)

- User Stories = describe the functionality that is valuable to a user/customer.
    - Composed of 3 components:
        - Card = a written description of the story used to plan and serve as a reminder.
        - Conversation = fleshing out the details of the story.
        - Confirmation = tests that convey and document details to confirm when the story is complete.

- Stories are typically written by a customer team, composed of developers, testers, users, customers and more. This is done in order to ensure the software meets the needs of the users.

- INVEST = followed by good stories.
    - Independent, Negotiable, Valuable, Estimatable, Small, Testable.



    - Not all details are given, as the specific come from a conversation when relevant for continuing the software.

- Front of the user story card – shows the user story and a point estimate.

- Back of the user story card – shows the acceptance tests of what they want allowed by the story.

- Story Cost = an estimate given by the developers of the time and complexity of the story relative to other stories. Each team defines story points differently.

- Traditional Waterfall Model (for development) = Write requirements, analyze them, design a solution, code, test code.

- Story-Driven Project = customers and users interact throughout the duration of the project, segmenting the project into iterations (kind of like blocks for gym training) where they reevaluate and adapt their approaches.

- Epic = story that is too large, and becomes an overarching theme, supported by smaller stories.

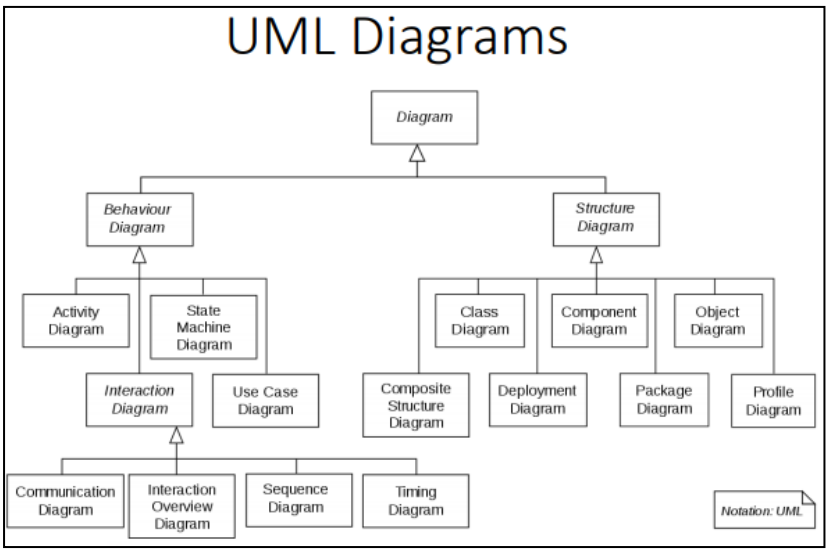
- User stories are not as useful for giant projects with geographically located teams, or many multiple teams.

_____ (**Lecture #9** – UML Diagrams) _____

- Unified Modeling Language (UML):
  - Unified = brings together several techniques and notations for design.
  - Modeling = describes a software system and its design at a high level of abstraction.
  - Language = provides the means to communicate this design in a consistent and understandable fashion.

- UML is used to model object-oriented designs, independent of programming languages.

- History of UML:
  - Object-oriented languages began to appear in the mid 1970's, and were abundant by the mid 1990s.
  - It became cluttered, requiring a necessary unified approach for design, giving UML.

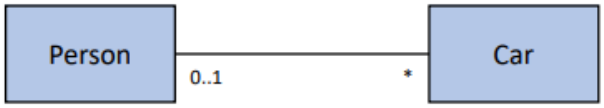


- Each class in a class diagram is representing in a rectangle with 3 sections:
  - Name (of the class), Attributes (the data members of the class), and Operations (functions of the class).
  - Visibility within the class may be shown with Public (+), Protected (#), Private (-).
  - Attributes and operations may be shown with types.

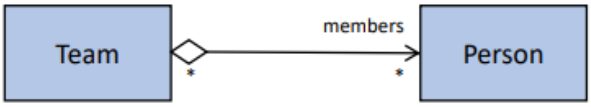
Person
firstName lastName
getFirstName() getLastName() setName( ) equals( ) toJSON()

Person
- firstName: string - lastName: string
+ getFirstName(): string + getLastName(): string + setName(first: string, last: string) + equals(otherPerson: Person): boolean + toJSON(): string

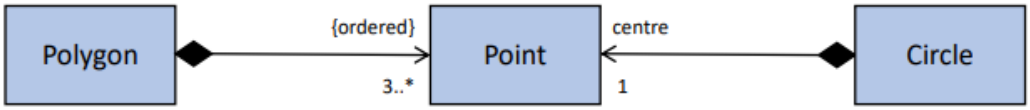
- *Class Diagrams:*



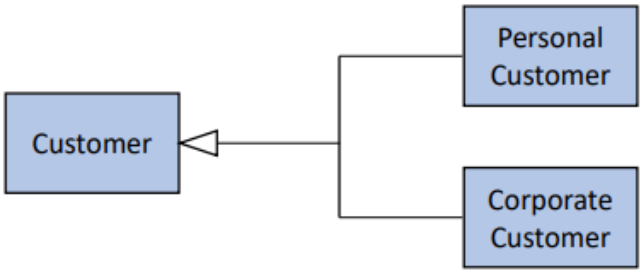
  - Aggregation = a part-of relationship, that can exist independent of the relationship.



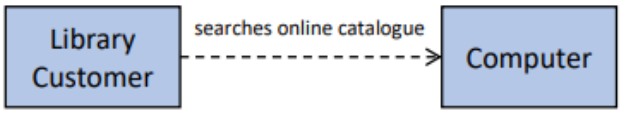
  - Composition = a dependent relationship that exists only with its owner.



  - Generalization = indicates one or more classes are derived from another class.



  - Dependency = exists between 2 classes if changes to one might impact another.



(**Lecture #10** – Design Principles)

- Coupling = describes the interdependence of entities (low is best).
  - High Coupling – cone object uses the internal data of another directly.
  - Medium Coupling – controlling the flow of another entity, passing entire data structures when only some data is needed.
  - Low Coupling – data is shared only through parameters or event messages.

- Cohesion = relatedness of functionality or focus of an entity (high is best).
  - High Cohesion – entity members and functions have a well defined purpose, acting on similar data or topics.
  - Low Cohesion – related only by grouping, not by topic. Functions are distinct.

- *Design Principles:*

    - Encapsulate What Varies = if a behavior of a class is not consistent to all members of it now and or in the future, it may be easier to create an interface for the behavior (all birds fly differently).
        - This eliminates code duplication, and allows more code duplication.
        - Behaviors can be independently modified, and can dynamically be changed at runtime.

    - Code to an Interface, Not an Implementation = if given a choice between interacting with a subclass or supertype, choose the supertype.

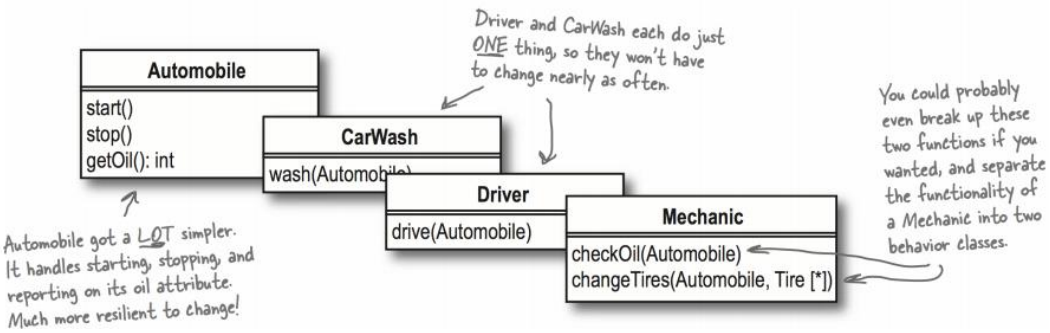
    - Favor Composition Over Inheritance = this is done in order to produce more flexible, maintainable and reusable code.
        - Inheritance establishes an IS-A relationship.
        - Composition/Aggregation establishes a HAS-A relationship.

    - Composition = an object composed of other objects OWNS those objects. When it is destroyed, so are they.

    - Aggregation = an object composed of other objects USES those objects. When it is destroyed, they remain.

- *SOLID Design Principles:*

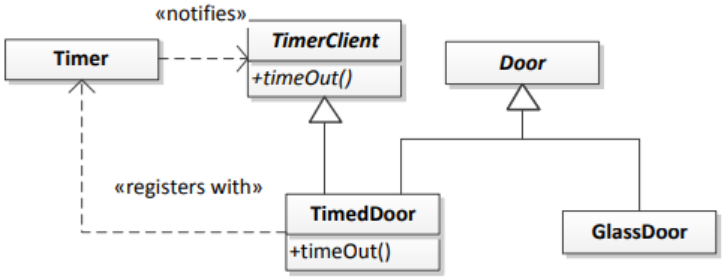
    - Single Responsibility Principle = every object in a system should have a single responsibility, with all its services focused on that single responsibility.



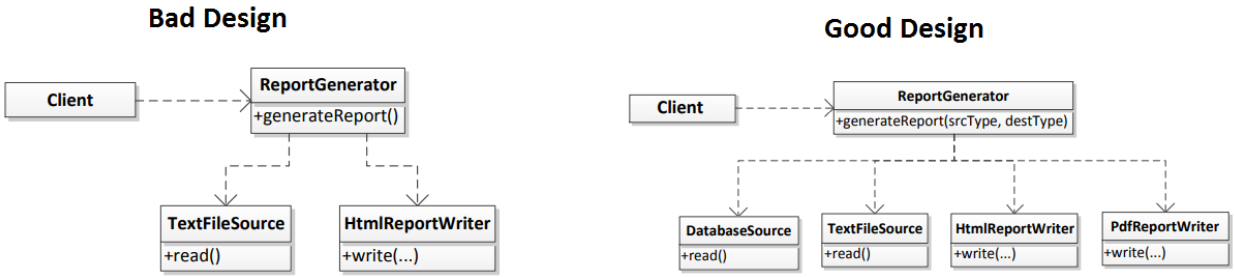
    - Open/Closed Principle = classes should be open for extension and closed for modification.
        - Treat good source code as untouchable, and only add onto it.

    - Liskov Substitution Principle = subtypes must be substitutable for their base types.
        - To put it differently, any method of the base class should work on the subclass.

- Interface Segregation Principle = many client interfaces are better than one general purpose interface. Clients should not be forced to depend upon interfaces that they don't use.



- Dependency Inversion Principle = high level modules should not depend on low level modules, but instead both should depend on abstractions.



_____ (**Lecture #11** – Intro to Design Patterns) _____

Discovering Design Patterns

1. Solve a software problem
2. Abstract and generalize the solution into patterns
3. Record the patterns for future use

Applying Design Patterns

1. Search for patterns applicable to the problem at hand
2. Understand the patterns and how to use them to solve the problem
3. Adapt and implement the patterns to solve the problem

- Creational = deals with object creation.
- Behavioral = deals with the ways classes or objects interact and distribute responsibility.
- Structural = deals with composition of classes or objects.

Gang of Four Design Patterns

Creational	Structural	Behavioral
Factory Method Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Flyweight Façade Proxy	Interpreter Template Method Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

- Model View Controller = a behavioral design pattern involving encapsulating components of interactive applications.
    - Model – the data classes.
    - View – the user interface components.
    - Controller – the application logic/functionality.

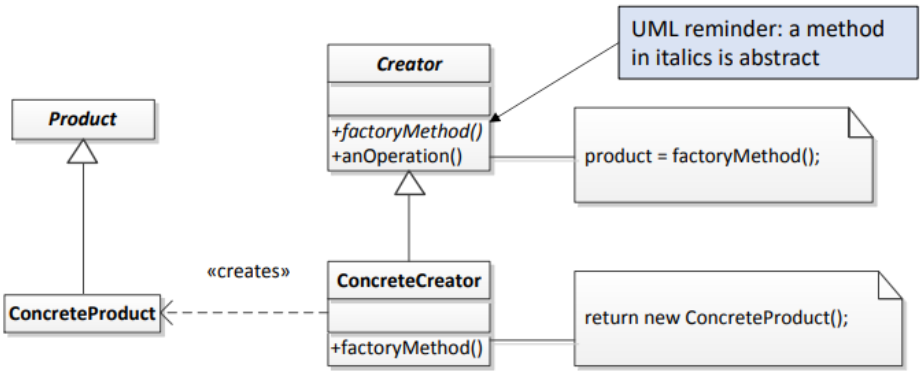
_____ (**Lecture #12-a** – Creational Design Patterns) _____

- Singleton = ensures a class only has one instance, and provides a global point of access to it.
    - Usually made with a constructor check, that keeps track of the number instances.

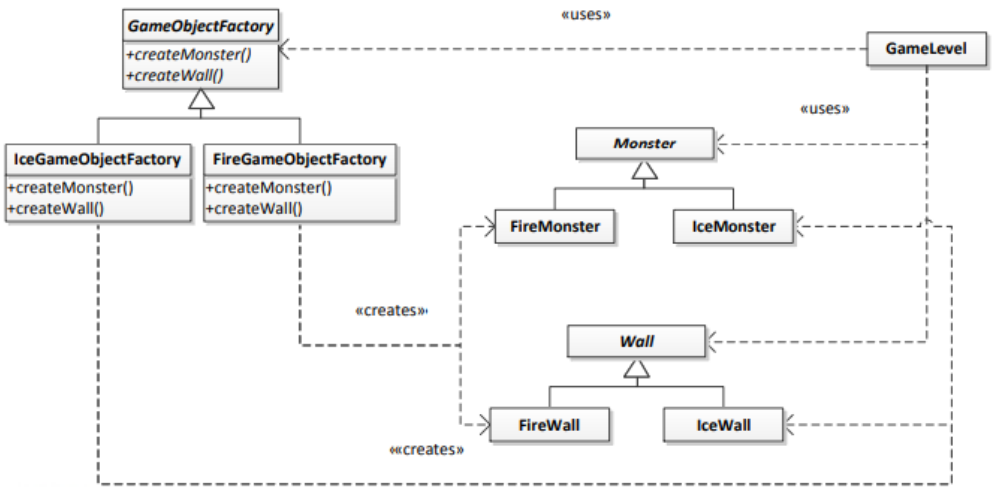
```
if (uniqueInstance == NULL)
  uniqueInstance = new Singleton();

return uniqueInstance;
```

    - Gives controlled access to instantiation, and reduces name space, but you have to worry about who deletes the instance.

- Factory Method = defines an interface for creating an object, but lets subclasses decide which class to instantiate.
    - A class can't anticipate the class of objects it must create, so it wants it's subclasses to specify them in order to best create them.



- Abstract Factory = provides an interface for creating families of related or dependent objects without specifying their concrete classes.

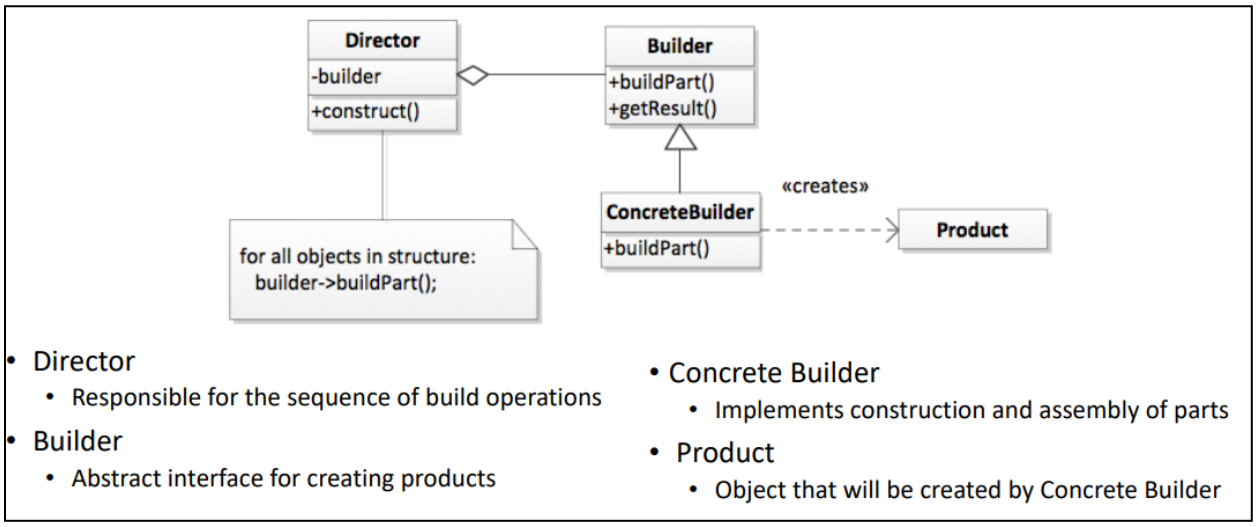


- Abstract Factory promotes a smooth exchange between product families, and consistency in products, but difficulty in supporting new products.

Factory Method:
- Creates a single product
- Uses inheritance
- Superclass methods remain generic and use the factory method as needed to create the product

Abstract Factory:
- Collects multiple factory methods into a class to create multiple related products
- Uses aggregation / composition
- Client remains generic and uses the factory as needed to create the products

- Builder = extracts the object construction code from its own class in order to create separate builder objects, that create a complex object made up of different parts



- Director
  - Responsible for the sequence of build operations
- Builder
  - Abstract interface for creating products
- Concrete Builder
  - Implements construction and assembly of parts
- Product
  - Object that will be created by Concrete Builder

- Prototype = specify objects to create with a prototype instance and create new objects by copying the prototype.
  - Can't be copied with a copy constructor as it uses an abstract class.
  - Useful when classes to instantiate are specified at runtime, or expensive to create but easy to copy.

(**Lecture #12-b** – C++ Const Correctness)

Both pointers and the data they point to can be `const`

```
char* p = "Hello";                      // non-const pointer
                                        // non-const data

const char* p = "Hello";                // non-const pointer
                                        // const data

char* const p = "Hello";                // const pointer
                                        // non-const data

const char* const p = "Hello";          // const pointer
                                        // const data
```

When the data pointed to is constant, some add `const` before the type name; others add it after

```
const Widget* const w

Widget const* const w
```

- You can't assign the address of a const object to a non-const pointer.
  - However you can do: char* c = "Hello";
  - As this specific case was allowed in C++ to make many C programs valid. You can't modify a string literal ("Hello") through a pointer like that

- Temporary objects are always const. Adding const to a param type allows for temporaries.
  - Ex. printName("Joe"); vs. printName(name_variable);
  - The first would still work because of const.

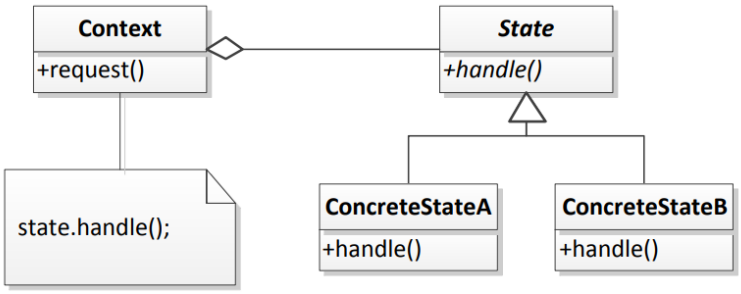
- A const member does not modify the object it is called on.

- If a function is const, you cannot change/reassign it's private data (this.name -> name;).

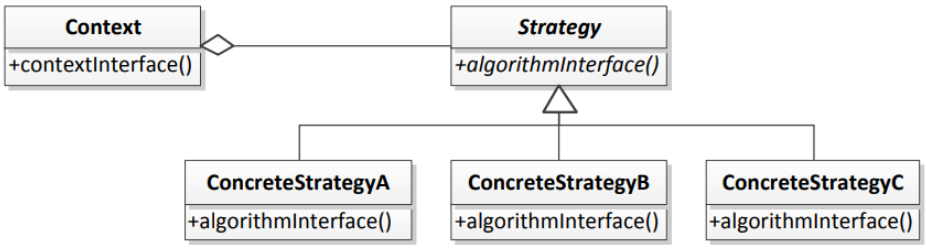
- A const function can be called on both const and non const objects. A non const function can only be called on non const objects.

(**Lecture #13** – Behavioral Design Patterns)

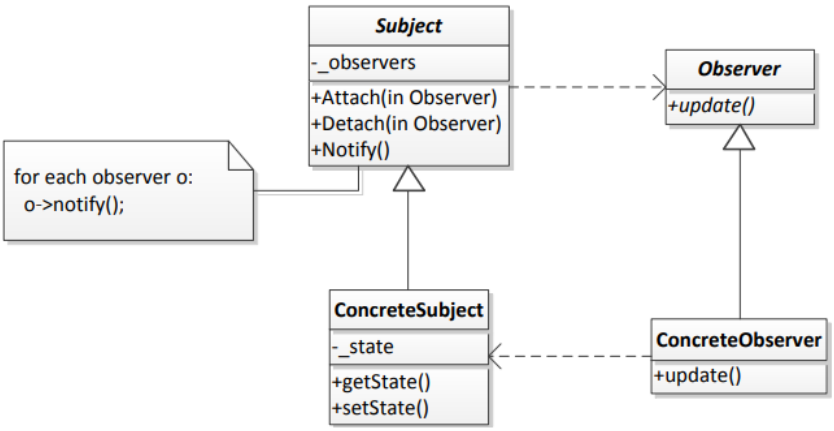
- State = allows an object to alter its behavior when it's internal state changes.



- Strategy = defines a family of algorithms, encapsulating each other to make them interchangeable.



- Observer = defines a one-to-many dependency between objects, so that when one object changes stat, all its dependents are notified and updated.



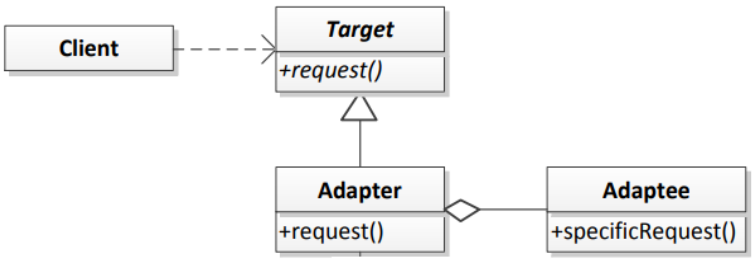
- Command = encapsulates a request as an object, in order to allow parameterization of clients with different requests.

```
menu.add(new MenuItem("Turn off lamp", cmd2));
menu.add(new MenuItem("Turn on porch light", cmd3));
menu.add(new MenuItem("Turn off porch light", cmd4));
```

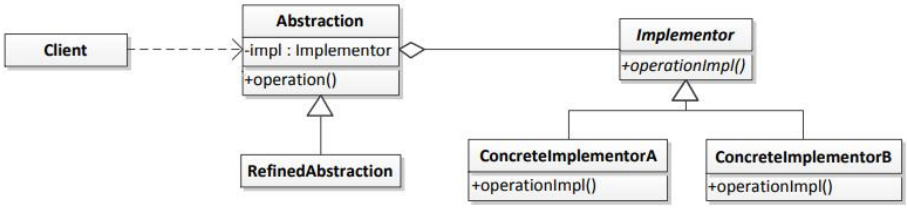
- Visitor = allows for defining a new operation without changing the classes of the elements on which it operates. Only if a class is designed to accept a visitor.

_____ (**Lecture #14** – Structural Design Patterns) _____

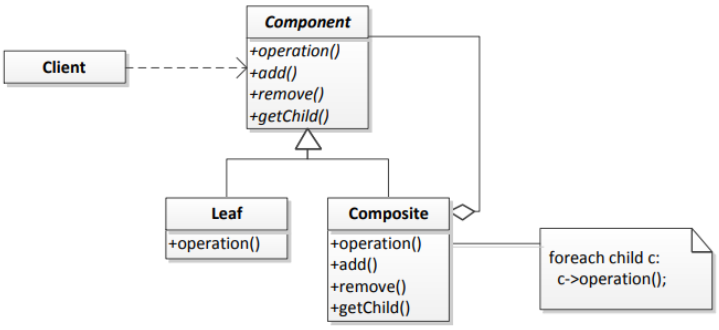
- Adapter = converts the interface of a class into another interface clients desire.



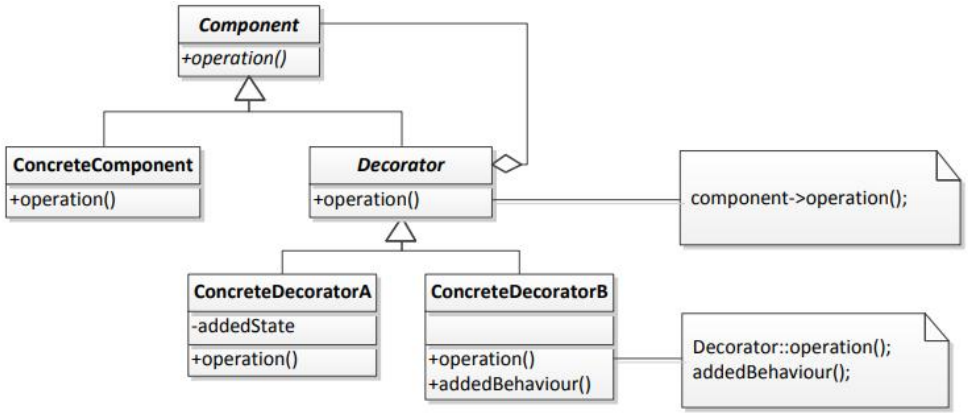
- Bridge = decouple an abstraction from its implementation.
  - Similar to strategy design, but strategy focus on providing different families. Bridge focuses on decoupling abstraction and implementation into different classes.



- Composite = composes objects into tree structures to represent part-whole hierarchies.
  - This let's clients treat individual objects and compositions of objects equally.



- Decorator = attaches additional responsibilities/functionality to an object dynamically.



- Flyweight = a shared object that can be used in multiple contexts to support large numbers of objects.
    - Done by sharing or being organized by a certain trait.