# W1

**Procedural Programming**

* Generally oriented around procedures (also known as functions, routines, or subroutines) which accept data, usually as parameters
* These process this data according to their internal programming logic, using basic statements or by calling other procedures
* In the end, they produce a result and return it to where they were called from
* Typically, a main procedure drives the program by executing statements and procedure calls in sequence until completion

Procedural programming is also sometimes referred to as imperative programming

* procedural programming uses blocks and scoping rules that non- structured imperative languages do not



**Object-Oriented Programming**

* Structured around objects
* Objects are instances of classes
* Objects accept messages from other objects; these messages essentially call methods in the receiving object in order to perform an action (that is the programming logic of the invoked method)
* You may think of methods as procedures that are encapsulated inside a class

Objects

* They combine (i.e. encapsulate) in one unit both data and functionality



* The data encapsulated in an object are called data members of the object
* The functionality is provided by functions called methods; the methods encapsulated in an object are also referred as member functions of the object
* Usually methods of an object provide the only way for an external piece of code (i.e. another object) to access its data members
* Objects are instances of corresponding types that are called classes
* There is a direct link between the concept of a class and that of a traditional Abstract Data Type

**Procedural vs. Object-Oriented Programming**

* The way it organizes things
  + procedural languages organize by procedures with data and functionality separate
  + object-oriented languages organize by classes and objects, with data and functionality effectively bundled together

**Benefits of Object-Oriented Programming**

* Better modeling of elements of the real world



* Better application and use of Software Engineering principles
  + Information/data hiding
  + Encapsulation
  + Abstraction
  + Modularization
* Better integration with other programs and libraries through agreed upon and standardized interfaces
* Information hiding



* + shielding and “hiding” the underlying design and implementation details of a subsystem (or a class) from the rest of the program
  + the rest of the program simply interacts with the subsystem through a published interface
* Encapsulation



* + Refers to the bundling of data and functionality together into a single package that is accessed through a well-defined interface,
  + effectively restricts direct access to at least some of the package’s data and functions
* Abstraction
  + Refers to modeling an entity (e.g. program, method, algorithm, data) in a way so that only the important characteristics are presented, while the non important ones are omitted
* Modularization
  + Refers to the design approach where a software application is implemented as a collection of independent units (subsystems) that communicate by exchanging only the absolutely necessary information (data) that is required to perform an operation
  + OOC got this as default
  + have low coupling
  + each unit (subsystem) performs a specific operation or a collection of very closely related operations
  + have high cohesion
  + a collection of related classes constitutes a subsystem

Some Caveats

* Object orientation is not the solution to every programming need
  + Data base applications 🡪 SQL, 4GLs
  + Embedded systems 🡪 Assembly
* can be difficult and developers often fall short of achieving these goals

**Why C++?**

* Its template library provides a decent collection of starting points, and add-on packages like Boost take things even further
* Mastering C++ will also make you a better programmer

**Why Not C++?**

* C++ lacks garbage collection and forces users to manage memory on their own
* C++ lacks some high level language features like reflection
  + Reflection: let the program ask questions about itself
* Templates are still harder to use and more error prone than they should be in practice
* Some recent developments in the language are perceived as feature creep and unnecessary

**C++ programming**

Structure of a C++ Program

* The basic elements of a C++ program are
  + The classes (i.e., a notion of Abstract Data Types),
  + The methods (i.e., functions encapsulated in classes), and
  + The data members (i.e., data fields encapsulated in classes)
* Most programs are made up of multiple classes and functions
* A main function is required for a program as an entry point to bootstrap the rest of its functionality
  + One main function must exist
  + No more than one can exist in the same program

**C++ basics**

Constant Expressions

* #define constants – preprocessor definitions that do a simple replacement during preprocessing and before compiling



* const constants – named constant declarations where the programmer commits to not changing a value, and this promise is enforced by the compiler



* + setup at runtime
* constexpr constants – constant expressions evaluated at compile time, allowing them to be placed in read-only memory to improve performance



* + when to use
    - everything has to b a literal
    - have to be something that is known at compile time

Statements and Blocks

* A block, or compound statement, is the term given to a collection of statements enclosed in {...}

Structures

* In C++, a user-defined record (aggregate type) is called a structure and is referred to as a struct in a program
* C++, structures and classes are more-or-less equivalent except for their default visibility:
  + In a structure, members default to public;
  + In a class, members default to private.

**Pointers and Memory**

Pointers

* In C++, a pointer variable can point to any sort of memory
* two main uses for pointers in C++:
  + As a way of referring to memory allocated dynamically off the heap (using the malloc() function or the operator new)
    - This allows for data that can be created on the fly or dynamically sized (such as linked lists, trees, etc.)
  + When passing large chunks of data to a function or method, passing a pointer can be more efficient, as it reduces copying and speeds up processing
* the programmer is responsible for freeing up dynamically allocated storage when they are done with it (using free() or delete, depending on how it was allocated in the first place)
* If a programmer fails to do this properly, their program will leak memory leading to performance and stability problems

## W2

**Functions**

* Functions in C++ are similar to functions in C, with a few additions and options that aren't present in C
* Functions must be declared before they can be called, otherwise the compiler won't know what to do with them
* Functions also have to be defined; that is, somewhere the body of the function must be given along side its declaration
* Note that a function can be declared in one spot (like a header file) and then defined elsewhere (in a source code file)

Function declarations

* Function declarations in C++ contain several things
  + The name of the function
  + The argument or parameter list for the function (which could be empty)
  + The return type of the function (which could be void if the function is not going to be returning anything)
  + Optionally, one or more modifiers designating special behaviour of the function or modifying how the compiler treats or compiles it (this is where things can get ugly and deviate from C ...)

**Function Modifiers**

Inline

* The compiler should try to embed the code for the function where it is called from, typically for performance reasons

Constexpr

* The compiler should evaluate the results of calling the function at compile time, making this usable with constexpr constants

Static

* The function is not visible outside of its file / translation unit
* Can be only used inside the code file

**Preprocessor directives**

* In C++ (as in C), before compilation, a preprocessor goes through your code, doing a variety of things
* In particular, the preprocessor looks for directives that give it instructions on things it should do with or to your code
* Preprocessor directives begin with a # and generally take an entire single line of code
* Unlike regular code, they do not end with a ;

#include

* Used to include the contents of another source file into the current file, like: #include <iostream> // C++ standard headers drop the .h
* Generally, this is used to include header files containing various declarations, type definitions, other preprocessor directives, and so on
* That said, you can include code files (and other things!) as well, but doing so is generally frowned upon

Include guards

* #ifndef MYHEADER\_H
* #define MYHEADER\_H
* #endif
* To prevent multiple inclusion of the same header file

#define

* Used to define constants, replaced during compilation
* Can also be used to define macros that are also processed before compilation, such as: A picture containing knife

  Description automatically generated

• There are other directives as well for various purposes

* Conditional compilation (#ifdef, #if, #elif, #endif, ...)
* Throwing errors (#error)
* Line control (#line)
* Various other things (generally under #pragma)

**Namespaces**

* Namespaces provide a method for explicitly defining scope to the identifiers within it (e.g. types, functions, variables, etc.)
* allows us to logically organize our code better and avoid name collisions that can occur in large projects with multiple programmers (or when code is used from multiple sources)



* Only one entity can exist with a particular name in a particular scope; otherwise we have a name conflict or collision
* Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes
* Useful in larger projects

A picture containing knife, table

Description automatically generated

* Recall that entities (variable, types, constants, and functions) of the standard C++ library are declared within the std namespace, and we can avoid explicitly using std:: each time we reference them by:

using namespace std;

* Don’t use it in header files

**Input & output**

Limitations of Extraction

* The extraction operator works fine as long as a program’s user provides the right kind of data – a number when asked for a number, for example
* If erroneous data is entered (a letter instead of a number, for example), the extraction operator isn’t equipped to handle it; instead of reading the data, it puts a premature end to the extraction
* If we wanted to parse a line and extract and type check data from it, there are various methods in the string class to assist us
* Another option would be to use something called a stringstream that creates a stream-like interface to a string, allowing us to use the extraction operator on it in a smarter way

**Class**

* Put private even you don’t have too.

A screenshot of a cell phone

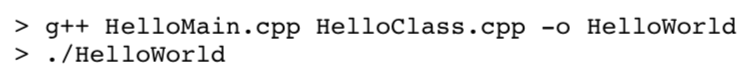
Description automatically generated

A screenshot of a cell phone

Description automatically generated



* Compile

1. 
2. A picture containing knife, bird

   Description automatically generated

**Access Specifiers**

* Classes have access specifiers to limit and restrict access to the various data members and member functions of the class
* An access specifier is one of the following keywords:
  + private: private members of a class are accessible only from within other members of the same class (this is the default for classes)
  + public: public members are accessible from anywhere where the class and its instantiated objects are visible
  + protected: protected members are accessible from other members of the same class and members of their derived classes

**Objects and Their Creation**

* Class definitions are a form of type definition
* when we define a variable of a class type, we say that this variable denotes an object (i.e. an instance) of this class
* If we define the variable as a pointer to a class type, we haven’t actually created an object yet; to create the object (i.e. allocate memory for it) we use the C++ new operator

A screenshot of a cell phone

Description automatically generated

Things to keep in mind with objects

* An object of the class Triangle corresponds, models, and denotes a specific Triangle that has a specific name (e.g. tr2)
* Each object has all the data members and member functions defined by the class of which it is an instance of (e.g. height, base, area)
* While each object from a class has the same set of data members and member functions, each object has its own values in its data members
* For instance, in this example, each object of the class Triangle has a data member called height, but each object may have a different value in its data member height

Ex

A screenshot of a cell phone

Description automatically generated

**Objects and Their Destruction**

* Objects created on the stack are destroyed when the function they were created in returns
* For dynamically created objects, as we are using the new operator to create them, we use the delete operator to destroy them
  + They are not automatically garbage collected as in some languages like Java
* When the program terminates, the heap is destroyed along with everything it contains; that said, it is still better to delete objects when you are done with them as destructors are not called when the heap is destroyed when the program ends (more on this in a bit ...)



**Constructors**

* A class can include a special function called a constructor
* A constructor is automatically called whenever a new object of this class is created (on the stack or on the heap via new)
* A constructor allows the class to initialize member variables, allocate storage, and so on
* Constructors are not strictly required in C++, but are strongly recommended as the proper way to initialize objects
* If you do not provide one, a default one that takes no parameters and does nothing is implicitly created behind the scenes for you

Copy Constructors

* The copy constructor is a special constructor in C++ that creates an object by initializing it with another, previous initialized object of the same class
* The copy constructor can be used to:



* + Initialize one object from another of the same type
  + Copy an object to pass it as an argument to a function
  + Copy an object to return it from a function
* If you do not declare a copy constructor, the compiler will typically give you one implicitly that does a simple member-wise copy of the source object
  + You can ask it not to do this, if you don’t want it
* If the simple implicit copy constructor does not suffice to initialize the object properly on its own, however, you should define one for yourself
  + For example, if your objects contain pointers to other things, you might need to make copies of those other things too, instead of just copying the pointers

**Destructors**

* A class can also include a special function called a destructor
* A destructor is automatically called whenever an object of this class is being destroyed (when delete is used, or when a function returns and it has objects in its stack frame)
* A destructor allows the class to deallocate storage and so on
* Destructors are also not strictly required in C++, but are strongly recommended as the proper way to tear down objects



* + If you do not provide one, a default one that does nothing is implicitly created behind the scenes for you

## W3 More Fun With C++ and Classes!

**Inheritance**

* define a class in terms of another class, which makes it easier to create, organize, and maintain an application
* provides an opportunity to reuse code functionality and accelerate implementation time
* the programmer can designate that the new class should inherit the members of an existing class
* This existing class is called the base class or superclass
* the new class is referred to as the derived class or a subclass of the base class
* A class can be derived from more than one class



* + it can inherit data and functions from multiple base classes
* To define a derived class, we use a class derivation list to specify the base class(es); a class derivation list has the form:
  + class derived‐class: access‐specifier base‐class
    - access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class
    - If the access specifier is not used, then it is private by default
* class B: public A { ... } 🡪 means: B is a subclass of A

Inheritance and Access

Table

Description automatically generated

Inheritance of Constructors and Destructors

* a base class’s constructors and destructor are also passed down to derived classes during inheritance
* They are automatically called by constructors and destructor of the derived class when its constructors or destructor are called
  + Base class constructors are called first, but their destructors are called last
* Graphical user interface, text, application

  Description automatically generated
* A picture containing graphical user interface, text

  Description automatically generated
  + If child constructors don’t specify which parent constructor to use, it will use the first one

Multiple Inheritance

* for one class to inherit from more than one class;
* listing multiple classes, separated by commas, in the base class section of the class definition
* A picture containing graphical user interface

  Description automatically generated

Friends of Classes

* Private and protected members of a class cannot be accessed from outside that class; however, this rule does not apply to “friends”
* Friends are functions or classes declared with the friend keyword
* A couple of things to keep in mind:
  + Friendship is not reciprocal; just because X is a friend of Y, that does not mean that Y is a friend of X, unless it is explicitly declared



* + Friendship is not transitive; a friend of a friend is not considered a friend, again unless explicitly declared



* A non-member function can access the private and protected members of a class if it is declared a friend of that class
* That is done by including a declaration of this external function within the class, and preceding it with the keyword friend
  + Because of the way they are declared, such friend functions can appear as though they are members of the the class but they are not; they simply have access to private and protected members that they ordinarily wouldn’t have access to

Diagram

Description automatically generated

Friend Classes

* a friend class is a class whose members have access to the private or protected members of another class
* Diagram

  Description automatically generated

**Templates**

* a generic way of writing functions and classes that are capable of operating on more than one data type without having to duplicate the code



* + the data type becomes a parameter in the definition of the templated function or class
* templates can match hand-written, less general code in terms of run-time and space efficiency, but are more general and only require a single implementation to be defined
* Function Templates

A picture containing diagram

Description automatically generated

* Class Templates

Diagram

Description automatically generated

Final notes on defining templates

* You can create templates that take more than one type using
  + template<typename T1, typename T2>
* template<typename ...> and template<class ...> syntaxes are terchangeable when defining basic templates
  + the syntax does matter when creating things like template templates

Standard Template Library

* provides a collection of general-purpose templated classes and functions that implement many commonly used algorithms and data structures
* The STL is standardized across C++ implementations;
  + for portability and maintainability of code, the STL should be used wherever feasible instead of re-implementing things unnecessarily
* At the core of the STL are:
  + Containers, algorithms, and iterators

Graphical user interface, text, application, table

Description automatically generated

* Guide: <https://en.cppreference.com/w/>

**Overloading and Polymorphism**

Overloading

* have more than one definition for a function name or operator in the same scope
* While they have the same name, overloaded functions and operators will have different argument types and, naturally, different implementations
* When you call an overloaded function or operator, the compiler will determine the most appropriate definition to use based on the types that you are using at the time

Function Overloading

* you can have multiple definitions for the same function name in the same scope
* this applies to methods or member functions of classes as well
* The definition of the function must differ from each other by the types and/or the number of arguments in the argument list



* You cannot overload function declarations that differ only by return type as the compiler might not be able to determine which version of the function you are attempting to call



* Ex

Diagram

Description automatically generated

Operator Overloading

* Most of the built-in operators in C++ can be redefined or overloaded
* a programmer can also add and use operators with user-defined types, including classes
* As an overloaded operator is, treated like a function behind the scenes, it is considered to have an argument list and a return type and follows the same general rules as an overloaded function
* if we would like to be able to add or combine objects from one of our classes together, we could overload operator+ and then bring them together using object1 + object2
* This is a powerful mechanism in C++, but can be confusing to other people if abused, overused, or used improperly
  + There is nothing stopping you, for example, for using operator+ for other things; syntactically, your program would still make sense, but it would be much more difficult to understand from a semantic perspective
* Ex

A close up of text on a white background

Description automatically generated

* Ex: we can overload stream operators << and >> if we wanted to stream our own user-defined types to and from files

A picture containing chart

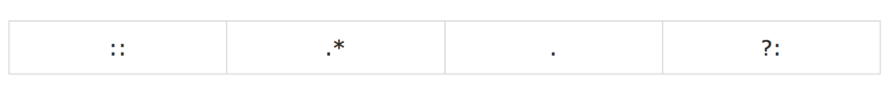
Description automatically generated

* The following operators can be overloaded:

Water next to the ocean

Description automatically generated

* These operators, however, can’t be overloaded:



Polymorphism

* Polymorphism means “many forms”
* In C++, we are usually referring to sub-type polymorphism, in which we can make use of derived classes through base class pointers and references
* Ex

A picture containing diagram

Description automatically generated

* the method invoked depends on the type of the pointer, not what the object actually is
* This is referred to as static dispatching, as it is determined by the C++ compiler when code is compiled
* This is the default for efficiency
* Ex

A picture containing diagram

Description automatically generated

* The virtual keyword enabled dynamic dispatching



* In this case, the compiler does not know which method to invoke at compile time and instead the program needs to determine this at run-time as a derived class might override the method
* This gives flexibility, but costs in performance
* So, use virtual if a method may potentially be overridden



* virtual is specified in the header file, not in the implementation
* A method declared as virtual stays virtual in all descendent classes
* For readability, convention is to continue adding virtual as a reminder, even though it is not strictly necessary
* It is also possible to make use of pure virtual methods (also referred to as abstract methods in other languages)
* These are methods without an implementation
  + These are used to force derived classes to implement particular methods
  + These are declared as in this example:
    - virtual void sayHello() = 0;
* The use of pure virtual methods gives rise to what are referred to as abstract classes
* These are classes that have one or more pure virtual methods
* These classes cannot be instantiated and are used to serve as base for other derived classes
* A derived class is concrete if and only if all inherited pure virtual methods are implemented

Diagram

Description automatically generated

Questions?

* Template is like python n
* Is it more convenient to just use template

## W4

**The this Pointer**

* Every object in C++ has access to its own address through the use of a pointer called this
* The this pointer is an implicit parameter to all member functions, and so inside a member function, the this pointer can be used to refer to the invoking object
* Ex
  + Diagram

    Description automatically generated

**Static Members**

* By declaring a class member as static, it is possible to have that member belong to the class itself, as opposed to individual objects of the class
* Such static members, in a way, are essentially shared among all objects of the class
* Static members can also be accessed without requiring the instantiation of an object in advance
  + Diagram

    Description automatically generated
* When we declare a member variable as static, we are simply telling the class that it exists
* As it is not attached to any object, no storage is allocated for it from only its class declaration
* static member variables need to also be declared separately to instantiate them and, optionally, initialize them
* We can also have static member functions as part of our classes
* Again, these can be used without having to instantiate objects
* As these functions are not attached to particular objects, they do not
* have a this pointer
  + Diagram

    Description automatically generated
* There are some caveats to creating static members that should be kept in mind whenever using them
  + Static member variables are more-or-less the same as having global variables, albeit potentially with some access restrictions



* + Because static member variables are essentially shared between all objects of a class, they represent a threat to thread-safeness and access to them needs to be regulated like other globally shared data



**Default Parameters**

* Default parameters allow functions to be called without providing one or more trailing parameters
* Instead of explicitly providing such a parameter, a default value is substituted in its place
* This default value is specified in the declaration of the function
* Both regular functions and member functions of classes can use default parameters
* Ex
  + Diagram

    Description automatically generated
* Once a default parameter is specified in a parameter list for a function, all subsequent parameters must also have default values
* As another important note, default parameters can cause some attempts to overload a function to fail because it creates ambiguity between the two functions …
* Ex
  + A picture containing diagram

    Description automatically generated

**Big Design Up Front**

Traditional requirements analysis:

* Talk to stakeholders (customer, end users, etc.)
* Think about the planned system; develop UML models
* Maybe prototype a bit
* Spend months developing a big document covering every requirement of the system up front
* Give said document to developers
* Receive word from developers that the project will actually take 24 months instead of the desired 6 months

Problems

* Very difficult to envision every possible feature needed up front
* Process of documenting every requirement is tedious and error-prone
* Customers often change their minds or come up with new ideas as they see the software being developed
* Requirements documents are long and boring
* Greater chance they will simply be skimmed or entire sections will be skipped
* Hard to grasp the big picture behind a 300-page requirements specification
* Time wasted writing 3/4 of the requirements that the team won’t be able to complete in the allotted 6 months
* More time wasted as the development team decides which requirements it can implement in time
* Levels of indirection between customers and developers
  + Lack of direct communication leads to misinterpretation of the intended functionality
  + Remember the telephone game?
* Focusing on a checklist of requirements rather than on the user’s goals
* does not necessarily lead to a good overall understanding of a product:
  + Text

    Description automatically generated
* Software is intangible
  + Very hard to estimate reliably
  + Pipe dream: glorious PERT charts enumerating every task that must be completed, the duration of each task, and the order in which the tasks must be completed
    - Table

      Description automatically generated
* As software is built, customers often change their minds about existing features and think up new features
* Requirements changes are called a change of scope
  + Implies that the scope of the project was previously fully defined
  + Implies that a project is complete when it fulfills its list of requirements rather than its intended users’ goals

**Software Requirements = Communication**

* Those who want the software must communicate with those who will build it
* A project relies on information from those who view the software from a business perspective, and those who view it from a development perspective
  + If the business side dominates too heavily, it can mandate functionality and deadlines typically with little concern that:
    - The development team can deliver the requested functionality on schedule
    - The development team understands exactly what is being requested
* On the other hand, if the development side dominates, it
  + Replaces the language of business (the language of the domain) with technical jargon
  + Loses the opportunity to learn what is needed by listening

**User Stories**

* A user story describes functionality valuable to a user/customer



* Three main components:
  + Card: A written description of the story used to plan and serve as a reminder



* + Conversation: Conversations about the story to flesh out its details
  + Confirmation: Tests that convey and document details; confirm to us when the story is complete
  + Often written on the front of an index card, with confirmation details written on the back

Customer Team

* Stories are usually written by a customer team:
  + Ensure the software will meet the needs of its users
  + Includes developers, testers, product managers, actual users, customers, etc.
* Stories must be written in the language of the user – not in technical jargon
  + Allows the customer team to be able to prioritize stories

Good examples:

* A picture containing text

  Description automatically generated

Bad examples:

* A picture containing text

  Description automatically generated

Common Templates

* “As a <role>, I want <goal/desire>.”
* “In order to <receive benefit> as a <role>, I want <goal/desire>.”
* “As <who> <when> <where>, I <what> because <why>.”
* “As a <role>, I can <action with system> so that <external benefit>.”
* “As <persona>, I want <what?> so that <why?>.”

Conversation

* Notice that many details are missing
  + What fields can the user search for jobs on?
  + Does the user have to be logged in?
* A user story is a reminder to have a conversation
  + We will discuss these details with the customer at the time they become important (just-in-time requirements analysis)
  + The conversation is the important part – not the story itself
* Cards represent customer requirements rather than document them
  + Card contains the story; details worked out in the conversation and recorded in the confirmation

Front of the card:

* User story
* Estimate in story points (more on this later)
  + 

Back of the card:

* Captures the expectations of the user in the form of acceptance tests
* These tests will help confirm that a story is complete
  + Text

    Description automatically generated

Estimation

* The cost of a story is the estimate given to it by the developers
* Each story is assigned an estimate in story points
  + This indicates the size and complexity of the story relative to other stories
  + For example, a story estimated at four points is expected to take twice as long as a story estimated at two points
* Each team defines the meaning of story points:
  + One team might treat a point as an ideal day of work
  + Another team might define a story point as an ideal week of work
  + Yet another team might treat a story point as a measure of the complexity of a story
* Recommendation for the project: treat one story point as an ideal evening of work for one developer (ideal = no interruptions, good focus/concentration)

Acceptance Tests

* The back of each card contains acceptance tests
  + They capture important aspects about stories
  + They verify that a story was developed to work exactly the way the customer team expected it to work
  + They run frequently – ideally automated
  + They are intentionally left short and incomplete
  + The team can add/remove tests at any time
* Goal is to convey additional information so developers will know when the story is done
* Story: A user can pay for the items in a shopping cart with a credit card
  + Graphical user interface, text, application

    Description automatically generated
* By providing tests to the developer early, the customer team has:
  + Stated their expectations
  + Reminded the developers of a situation they might have forgotten

**Story-Driven Development**

* The customer team and developers choose length of each iteration of development of the project
  + Might be anywhere from 1 to 4 weeks
  + Same iteration length used for entire project
  + At the end of each iteration, developers are responsible for delivering fully usable code for some subset of the application
  + The functionality captured by the stories in the iteration just completed should now be fully usable in the application

Release Planning

* Developers estimate how much work (in story points) they’ll be able to complete each iteration to set the project’s velocity
  + First estimate will be wrong, but this is useful to roughly determine the iteration schedule
* To plan releases, we sort stories in piles representing iterations, where the stories in each pile add up to no more than the estimated velocity
  + Highest-priority stories go in the first pile (iteration 1)
  + Next highest-priority stories go in the second pile (iteration 2)
* Table

  Description automatically generated

Prioritization

* To plan releases, customer team must prioritize stories:
  + Desirability of a feature to a broad base of users/customers
  + Desirability of a feature to a small number of important users/customers
  + Cohesiveness of a story in relation to others
    - For example, a zoom out story might not be high priority on its own, but might treated as such because it is complementary to zoom in, which is high priority
* Developers have different priorities for many stories:
  + They may suggest a story’s priority be changed based on technical risk or because it is complementary to another
* Customer team listens to their opinions, but ultimately sets priorities in a manner that maximizes value delivered to the organization

BDUF vs. Story-Driven Development

* Traditional waterfall model:
  + Write requirements, analyze them, design a solution, code said solution, test solution
  + Customers involved at beginning to write requirements
  + Customers involved at the end for acceptance testing
  + Between requirements and acceptance: customers often disappear
* Story-Driven Project:
  + Customers and users involved throughout duration of the project
  + Not allowed to disappear in the middle of the project

**Writing Good Stories**

* Good stories should follow the INVEST acronym: Independent, Negotiable, Valuable, Estimatable, Small, Testable

Independent

* Dependencies between stories should be avoided
  + Dependencies lead to prioritization and planning problems
  + Make estimation more difficult
  + Need to have the flexibility to easily move stories around in the release schedule, if needed
* Solutions:
  + Combine dependent stories into a larger, independent story
  + Find a different way of splitting the stories
* Dependent stories:
  + Text

    Description automatically generated
    - The first credit card will take 3 days to support, but each other card after that will only take 1 day. Which story should receive the 3 day estimate?
  + A better split:

Graphical user interface, text

Description automatically generated

Negotiable

* Stories should be negotiable – they are not written contracts or requirements
* Stories are reminders to have a conversation
  + Do not need to include all relevant details
  + We negotiate the details in a conversation between the customer team and the development team
  + If we do know some details when the story is written, we can annotate the card with those details
* Right amount of information to the developer and customer who will discuss the story:

Graphical user interface, text, application

Description automatically generated

* Too many missing details for developers to view the story as definitive
* Too much detail:

Text

Description automatically generated

* + Can lead to mistaken belief that the story card reflects all the details and that there’s no further need to discuss the story with the customer

Valuable

* User stories should be valuable to the users and customers
* Avoid stories valued only by developers
  + Should be written so that the benefits to the customers / users are apparent
  + This allows customers to intelligently prioritize the stories
* Poor examples:

Text

Description automatically generated

* Better

Graphical user interface, text, application

Description automatically generated

Estimatable

* Developers need to be able to estimate the size of a story
* Three reasons why a story might not be estimatable:
  + Developers lack domain knowledge
    - If developers do not understand a story, they should discuss it with the customer who wrote the story
    - Not necessary to understand all details but developers should have a general understanding of the story
  + Developers lack technical knowledge
    - Send one or more developers on a spike: a brief experiment to learn about an area of the application
    - Developers learn just enough that they can estimate the task
    - The spike itself is given a defined maximum amount of time (a timebox), allowing the developers to estimate the spike
    - Unestimatable stories turn into 2 stories: the spike and then the actual story
  + The story is too big
    - Example: A Job Seeker can find a job
    - Developers will need to disaggregate it into smaller, constituent stories
* Stories that are too large are called epics
* Despite being too large to estimate, epics are useful as placeholders – reminders about big parts of the system that must be discussed
  + This allows us to make a conscious decision to temporarily gloss over large parts of a system
  + Can be assigned a large, pulled-from-thin-air estimate

Small

* Size matters – stories should be small, but not too small
* Stories that are too large or small make planning difficult

Testable

* Stories must be written so as to be testable
* Successfully passing its tests proves a story has been successfully developed
* If the story cannot be tested, how will the developers know when they have finished implementing it?
* Untestable stories commonly appear with nonfunctional requirements:

Text

Description automatically generated

* Tests should be automated as much as possible. Strive for 99% automation – not 10%
* Things change quickly:
  + Code that worked yesterday may be broken today
  + Automated tests detect this quickly
* Some stories cannot be tested automatically, however:



* This can be tested, but not readily automated

**User Stories: Summary**

* Emphasize verbal rather than written communication
  + Remind us to have a conversation
* Comprehensible by both customers and developers
  + Written in the language of the domain
* The right size for planning
  + Not too large, not too small
* Work for iterative development
  + As long as we follow INVEST, can easily move stories between iterations
* Encourage deferring detail
  + We may not even need a story – don’t waste time detailing it up front

**User Stories: Some Caveats**

Scaling up

* How does this scale to very large projects with geographically distributed teams, especially if we are using small physical cards for our stories?

Vague, informal, incomplete

* As user stories are intended to start conversations, they are informal, open to interpretation, and short on details; they are definitely not appropriate for a formal agreement or contract

Lack of non-functional requirements

* Since performance or non-functional requirements are most often not included in user stories (testability), there is a risk of them being overlooked

## W5

**What is UML?**

UML stands for Unified Modeling Language

* Unified: It brings together several techniques and notations for design, and has been standardized (first by the Object Management Group in 1997 and then by the International Organization for Standardization in 2005)
* Modeling: It describes a software system and its design at a high level of abstraction
* Language: It provides the means to communicate this design in a logical, consistent, and comprehensible fashion

Goals of UML

* Enable the modeling of object-oriented designs
* Visually depict various aspects of the overall design of a solution
* Provide extensibility and specialization mechanisms to extend core concepts
* Be independent of particular programming languages and development processes
* Support higher-level development concepts such as collaborations, frameworks, patterns, and components

Goals of UML

* Enable the modeling of object-oriented designs



* Visually depict various aspects of the overall design of a solution



* Provide extensibility and specialization mechanisms to extend core concepts



* Be independent of particular programming languages and development processes



* Support higher-level development concepts such as collaborations, frameworks, patterns, and components



UML Diagrams

* Each UML diagram allows developers and customers to view a software system from different perspectives and from different levels of abstraction
* Diagram

  Description automatically generated

Class Diagrams

* widely used to describe the types of objects in a software system, their relationships, and constraints on their relationships
* In early stages of development, class diagrams might be incomplete, but are then refined and filled out as development proceeds
  + Some classes might be missing
  + Classes might be missing optional elements or details
* Each class in a class diagram is represented by a rectangle divided into three sections:
  + Name of the class
  + Attributes of the class (the data members of the class, including both variables and constants)
  + Operations of the class (equivalent to member functions/methods)
    - A picture containing table

      Description automatically generated
* Attributes and operations may include:
  + Visibility: public (+), protected (#), or private (-)
  + Type of attribute or operation
  + Multiplicity of attribute (how many of this are there)
  + Parameter list for operations
  + Other properties of attributes or operations
* Including this information is of the form:
  + visibility variable\_name: type
  + visibility variable\_name: type multiplicity = default\_value {property}
  + visibility method\_name(parameter\_list): return\_type {property}

A picture containing text

Description automatically generated

* Associations represent relationships between classes



* + Indicated with a solid line between the classes
  + Arrowheads are used to indicate the direction of the association; both unidirectional and bidirectional associations are possible
  + Can be annotated with a name for the association or roles that the classes play in the association
  + Can also include cardinalities to include the number of entities that can be involved with in the association, including one-to-one, one-to-many (1..\*), many-to-many (\*..\*), zero-to-many (0..\*), and so on

Diagram

Description automatically generatedTable

Description automatically generated

* Two special forms of association are aggregation and composition
  + Aggregation (indicated by a hollow diamond) indicates a part-of relationship, but such parts are non-essential and can exist independent of the aggregate



* + Composition (indicated by a filled diamond) indicates that something is an essential and integral component of something else



* + - Instances of a component may only have a single owner; sharing is not allowed
    - Components cannot exist independent of their owner
    - Components live or die with their owner

Diagram

Description automatically generated

* Generalization is another form of relationship supported by UML
  + This is used to capture the object-oriented concept of inheritance



* + In other words, it allows you to indicate that one class is derived from another class, creating parent-child relationships between classes

Diagram

Description automatically generated

* A dependency between two classes exists if changes to one class might induce changes to or otherwise impact the other class
  + There are many types of dependency, including use, calling, creation, and so on
  + The broken line used to indicate the dependency can be labelled with a message to provide details on the dependency

Shape

Description automatically generated

* UML class diagrams can also contain many other general programming or object-oriented concepts and constructs
  + Notes and comments
  + Enumerations
  + Interfaces and abstract classes
  + Templates
  + Qualified associations
  + Association classes

**Coupling**

* Describes the interdependence of entities
* Bound together to be functional
* High/strong coupling
  + One object uses the internal data of another directly
  + Returning a pointer to object data
  + Sharing global variables
  + C++ friends
  + High coupling leads to interdependent entities
    - Design is difficult to change and extend
    - Code is difficult to read
* Medium coupling
  + Controlling the flow of another entity
  + Passing entire data structures when only some data is needed
* Low coupling
  + Share only data, through parameters and returns
  + Events or messages passed
  + Low coupling leads to information hiding and encapsulation
    - Stable interfaces
    - Better maintainability

**Cohesion**

* Design is difficult to change and extend
* Code is difficult to read

Low cohesion

* Related only because they are grouped together
* Entity names often vague
* Functions provided perform various activities
* Groups of functions often act on distinct sets of data

High cohesion

* Entity’s members contribute to a well defined purpose



* Entity name often very specific



* Functions act on the same set of data
* Improved clarity of entities and their dependencies
* Easier to maintain and extend
* Leads to code reuse

**High Coupling / Low Cohesion**

* Hard to read and understand
* Boundaries between entities are weak and few
* High interdependence causes unstable code
  + Diagram

    Description automatically generated

**Low Coupling / High Cohesion**

* Easier to read, maintain and extend
* Entities are responsible for specific functionality
* Entities function more independently
  + Chart, box and whisker chart

    Description automatically generated

**Encapsulate What Varies**

* Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t.

Benefits:

* Eliminated code duplication



* Other types of objects can reuse the fly and quack behaviours
* Can easily add / modify existing behaviours without necessarily (or heavily) modifying our duck classes
* Can dynamically change behaviours at run-time

**Code to an Interface, Not an Implementation**

* When faced with the choice between interacting with subclasses or interacting with a supertype, choose the supertype. Your code will be easier to extend and will work with all of the interface's subclasses – even those not yet created.



* The interface we are talking about could be an interface, abstract class, or even a concrete superclass
* This design principle really says: Code to a Supertype, Not a Subtype

Benefits

* Adds flexibility
  + Code can now work with any type of Athlete – even those we haven’t created yet
* Simplified architecture
* Reduced duplication
  + Having a hierarchy of teams (BaseballTeam, FootballTeam, …) would result in extensive duplication of code
  + addPlayer would duplicate in each class

**Favour Composition Over Inheritance**

* By favouring delegation, composition, and aggregation over inheritance, we can produce software that is more flexible, and easier to maintain, extend, and reuse.
* Inheritance establishes an IS-A relationship
* Composition/aggregation establish a HAS-A relationship – this can often be preferable
* Instead of inheriting their behaviour, the ducks get their behaviour by being composed with the right behaviour object

Benefits:

* Creating systems using composition gives us more flexibility
* Encapsulate a family of algorithms into their own set of classes
* Can easily extend the code with new behaviours
* Can change behaviour at run-time
* Reduce code duplication

Rule of thumb

* If you need to use functionality in another class, but you don’t need to change that functionality, consider using delegation instead of inheritance.

**Composition**

* An object composed of other objects owns those objects
* When the object is destroyed, so are all the objects of which it is composed
* Note: we are not necessarily seeking to model the real world exactly here:
  + Yes, a weapon could exist on its own in the real world
  + The question you should be asking yourself is whether or not in your model, you need to track a weapon outside of a unit
  + If not, use composition (OWNS-A)
  + If so, use aggregation (HAS-A)

**Aggregation**

* An object comprised of other objects uses those objects
* Those objects exist outside of the object
* When the object is destroyed, the objects that comprise it remain

**Composition vs. Aggregation**

* When deciding which to use, simply ask: Do I need this object outside of the class?
  + If no, use composition (black diamond of death)
  + If yes, use aggregation (white diamond of life)

## W6

**SOLID DESIGN PRINCIPLES**

Summary

* Single Responsibility Principle
* Open/Closed Principle
* Liskov Substitution Principle
* Interface Segregation Principle
* Dependency Inversion Principle

Single Responsibility Principle

* Every object in a system should have a single responsibility, and all the object’s services should be focused on carrying out that single responsibility.
* Every object in a system should have a single responsibility



* Another way to think about a responsibility is as a reason to change
* When a class has more than one reason to change, it might be trying
* to do too much
* In such a case, we should try to break the class into multiple classes where each individual class has a single responsibility and thus has only one reason to change
* Benefits:
  + Doing this minimizes the chance that a class will need to be changed by reducing the number of things in the class that can change
  + Generally, this also results in high cohesion as the elements of the class belong together in a stronger way



* + Achieving higher cohesion generally increases reusability, robustness, understandability, and so on
* A picture containing graphical user interface, text

  Description automatically generated
* Diagram

  Description automatically generated

Open/Closed Principle

* Classes should be open for extension, and closed for modification.
* Closed for modification: The source code of our classes is to be treated as immutable … no one should be allowed to modify it



* Changing existing code can introduce new bugs
* If we need a different behaviour, we should extend the class



* Open for extension: Behavioural changes that may be required should be accomplished through inheritance or other means (e.g. the Observer pattern … more on this later)
* We should not touch our existing, well-tested code!
* Ex
  + Diagram

    Description automatically generated
    - This example violates the open/closed principle
    - What happens when we need to add a new Shape subclass?
    - We would need to change PaintApp to accommodate the new shape
  + Diagram

    Description automatically generated
    - A better approach is to refactor the code and take advantage of the inheritance hierarchy that is in place
    - We can now create new Shape subclasses without requiring changes to PaintApp
* Benefits:
  + This effectively allows the behaviour of a class to be modified without touching its source code
  + In doing so, we don’t risk breaking well-tested code
  + Instead, we only modify existing code to fix errors; new/modified features require extension

Liskov Substitution Principle

* Subtypes must be substitutable for their base types.
* The Liskov Substitution Principle is all about well-designed inheritance
  + When you inherit from a base class, you must be able to substitute your subclass for that base class without affecting program correctness
  + If not, you are misusing inheritance
* Ex
  + Diagram

    Description automatically generated
    - Bad example
  + Diagram

    Description automatically generated
    - The compiler will allow us to substitute 3DBoard for Board just fine:

Board\* board = new 3DBoard();

* + - But, 3DBoard cannot really stand in for Board without affecting program correctness

List units = board->getUnits(8, 4);

* + - What does this method mean on 3DBoard?
      * The Liskov Substitution Principle states that any method on Board should be usable on 3DBoard without affecting correctness
      * 3DBoard really is not substitutable for Board: none of the methods on
      * Board will work in a 3D environment
  + Diagram

    Description automatically generated
* Suppose we have a class Rectangle in our system
  + Text, letter

    Description automatically generated
  + A few months later we realize we need to add a Square
    - Does the following hierarchy violate the Liskov Substitution Principle?
    - After all a square IS-A rectangle
    - Diagram

      Description automatically generated
    - This one is more subtle and requires more consideration
    - The first clue that something is wrong:
      * A Square does not need both width and height members, but it inherits them anyways
      * Really, a Square only needs a single side length
      * It’s not a big deal though … memory is cheap these days, we have lots of RAM, and we won’t be making tons of Square objects anyways
    - The second clue:
      * The setWidth and setHeight methods inherited from Rectangle will
      * not be appropriate for Square
      * That’s okay; we can override them
      * Text, letter

        Description automatically generated
    - Third clue:
      * Our function f works for Rectangle but not for Square
      * Graphical user interface, text, application

        Description automatically generated
    - No worries … we can fix this too by making setWidth and setHeight virtual in the Rectangle base class
    - The fact that we have to violate our Open/Closed Principle to make this work is another hint that something is wrong

Text, letter

Description automatically generated

* + - Fourth clue:
      * Our function g works for Rectangle but not for Square

Text

Description automatically generated

* + - * The Liskov Substitution Principle says that anywhere we can use the base type, we should be able to use the subclass type without affecting program correctness … does this hold true here?
  + What gives? In real life, a square IS-A rectangle
  + A Square object, though, is not a Rectangle object
    - In the end, the behaviour of a Square is not consistent with the behaviour of a Rectangle
    - Behaviour is what software is all about
  + Conclusion: IS-A does not tell the whole story
  + We should use inheritance when one object behaves like another, rather than just when the IS-A relationship applies

Interface Segregation Principle

* Many client-specific interfaces are better than one general purpose interface. Clients should not be forced to depend upon interfaces that they do not use.
* A “fat interface” is supplied by a class whose interface is not cohesive
  + It has many responsibilities and is unfocused and hard to understand/modify
* The Interface Segregation Principle seeks to avoid fat interfaces
  + Some objects may require non-cohesive interfaces
  + Clients should not know about them as a single class
  + Clients should know about abstract base classes with cohesive interfaces
* Suppose we are implementing a security system
* We start with an abstract class Door:
  + Text

    Description automatically generated
* We wish to have a class TimedDoor that will sound an alarm if left open for too long
* First, we will create a class Timer which TimerClients can register with to receive notifications about timeouts
  + Table

    Description automatically generated
* We want TimedDoor to be able to register itself with Timer so that it can receive notifications when the door has been open for too long
* We choose to have Door extend TimerClient , so that a new derived class TimedDoor will be able to register itself with Timer
  + Diagram

    Description automatically generated
  + Problems:
    - The interface of Door has been polluted with an interface it does not require
    - Door is now dependent on TimerClient, but not all doors need timing
    - Those that don’t need timing will have to override the timeOut method to do nothing
    - When clients #include those timing-free doors, they will include the definition of the TimerClient class even though it won’t be used
    - Diagram

      Description automatically generated
  + If we continue this practice, then each time we need a new interface, we will have to add it to the base class, further polluting its interface
  + We will have to go back and implement the new interface methods in every subclass, violating the Open/Closed Principle
  + Door and TimerClient provide interfaces used by completely different clients:
    - Timer uses TimerClient
    - Classes that manipulate doors use Door
    - If the clients are separate, then so, too, should the interfaces be separate
  + Bottom line:
    - Don’t add new methods appropriate to only one or a few implementation classes
    - Instead, divide the bloated interface into multiple smaller, more cohesive interfaces
    - New classes can then implement only the ones they need
  + Solution using multiple inheritance:
    - Diagram

      Description automatically generated
    - The Adapter design pattern can also be used to solve this sort of problem – more on this pattern later

Dependency Inversion Principle

* High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions.
* Suppose we want to take data stored in text files and generate reports in HTML format
  + Diagram

    Description automatically generated
  + Graphical user interface, text, application

    Description automatically generated
  + TextFileSource and HtmlReportWriter are certainly reusable
  + But, we cannot reuse ReportGenerator unless we want to read from text files and write to HTML files
  + Suppose we write a new program that needs to read from a database and write to PDF files – it would be nice to reuse ReportGenerator
  + ReportGenerator is dependent on TextFileSource and HtmlReportWriter, so this is not possible
* We could modify generateReport to accept the type of source and destination to use
  + Diagram

    Description automatically generated
  + Text

    Description automatically generated
  + This drastically increases coupling in the system
    - Over time, more source and destination types will be added to generateReport
    - The ReportGenerator class will be littered with if-else statements and dependent upon many lower-level modules
  + This also results in a rigid and fragile system
    - Rigid: the system will become hard to change since every change will affect too many parts of the system
    - Fragility: when changes are made to the system, unexpected parts will break due to the changes
* Better solution:
  + Make ReportGenerator (the higher-level class) independent of the lower-level classes it controls
  + We can then reuse it freely
  + This is called dependency inversion
  + Diagram

    Description automatically generated
  + Text

    Description automatically generated

## W7

**Design Patterns**

* A description of communicating objects and classes that are customized to solve a general design problem in a particular context
* Applies to a specific problem we face
* Most design patterns help us to adhere to the well-known design principles

A solution to an abstract software problem within a particular context

* Abstract - Not specific to a language
* Context - Common situations in code design
* Problem - Goals vs. constraints in some context
* Solution - Design for cooperating classes/objects which resolve the goals and constraints

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

Components of Design Patterns

* Four main components of a design pattern:
  + Pattern name
    - Brief descriptive name that summarizes the pattern
  + Problem
    - Describes when to apply the pattern
    - Often includes conditions that should be met before applying the pattern
  + Solution
    - The classes/objects that make up the designs, along with their relationships, responsibilities, and collaborations
    - Doesn’t describe a particular concrete implementation → a pattern is a template
    - Provides an abstract description of a design problem and how a particular arrangement of classes and objects solves it
  + Consequences
    - Results and trade-offs of applying the pattern
    - Help us to evaluate the costs and benefits of applying the pattern
    - Often concern time/space trade-offs
    - May address language/implementation issues
    - The term consequences often has a negative connotation – this is not the case here

Design Patterns in Detail

* Pattern name and classification
* Intent (purpose, rationale, what it does, etc.)
* Also known as (alias name, if any)
* Motivation (scenario illustrating the problem solved by the pattern) • Applicability (situations where it is useful, etc.)
* Structure (graphical representation)
* Participants (elements, both classes and objects, involved, etc.)
* Collaborations (how the participants collaborate to do the required job)
* Consequences (tradeoffs and results of using the pattern)
* Implementation (pitfalls, hints, or techniques useful for implementation)
* Sample code (code fragments to aid in implementation)
* Known uses (examples in real systems)
* Related patterns (other patterns related to this, similarities/differences with other patterns, other patterns used in conjunction with this one)

Types of Design Patterns

* Three main types:
  + Creational: primarily deal with object creation
  + Behavioural: primarily deal with the ways in which classes or objects interact and distribute responsibility
  + Structural: deal with composition of classes or objects

Gang of Four Design Patterns

* classified a collection of design patterns into creational, behavioural, and structural categories
* Table

  Description automatically generated

Benefits of Design Patterns

* Patterns encapsulate a particular design to solve a specific problem
  + Helps create a knowledge base of patterns
* Reuse previously demonstrated solution without reinventing the wheel (possibly more costly and error-prone)
  + Faster development, recognized quality, lower cost, more reliable time/cost estimates, and so on
* Focused/improved communication amongst the pattern stakeholders
  + Common terminology
  + Common point of reference during design and analysis
* Allows one to think in “design” terms and not have to deal with code early on
* Increased use of patterns may move a system towards a more modular design in general
* Such a modular design may aid ease of change (maintenance) and evolvability of systems
* Encourages shared learning among developers and developing a mindset for abstract solutions to common problems

Drawbacks of Design Patterns

* Perhaps creates a tendency to use a “standard” solution instead of creating an innovative solution to the problem at hand
* Risk of forcing patterns into situations they don’t fit
* Risk of adjusting problem to make patterns work, instead of solving the actual problem
* Relying heavily on patterns may cause novices to think abstractly at the expense of not knowing the detailed workings

Model-View-Controller

* A behavioural design pattern (although some consider this to be an architectural pattern ... more on this later)
* Encapsulate components of interactive applications
* Text

  Description automatically generated
* Model
  + Class(es) which capture data, which can be:
    - Persistent - stored between program executions
    - Dynamic - data required during execution
  + Persistent data ↔ Dynamic data
  + Model classes handle computation on the data
* View
  + Class(es) which present the interface to the user
  + Typically GUI elements
* Controller
  + Coordinates the Model and View classes
  + May change the view
  + Can modify and retrieve data from the model

Diagram

Description automatically generatedDiagram

Description automatically generated

* Advantages
  + Reduces coupling
  + High cohesion within each element
  + Widespread approach to GUI software design

Model View Presenter

* Related pattern
* Further decouples view and model
* Often called MVP
  + Diagram

    Description automatically generated

**Creational Design Patterns**

Two main goals:

* Encapsulate knowledge about which concrete classes the system uses
* Hide how instances of these classes are created and built

System at large knows about objects through their interfaces defined by abstract classes

Give us flexibility in:

* what gets created
* who creates it
* how it gets created
* when it gets created

Types

* Singleton; Factory Method; Abstract Factory; Builder; Prototype

Singleton

* Consider a class called Logger
  + Logs information to a file
  + Needed by many different parts of an application
  + Graphical user interface, text, application

    Description automatically generated

**Const Correctness**

const and Pointers

* Both pointers and the data they point to can be const
* Use the right-to-left rule to read
  + Text

    Description automatically generated
  + Graphical user interface, text, application

    Description automatically generated
* const Data vs const Pointers
  + A picture containing text

    Description automatically generated
* const Data vs const Pointers
  + A picture containing text

    Description automatically generated
  + A picture containing application

    Description automatically generated
  + A picture containing chart

    Description automatically generated

const and Pointers

* When the data pointed to is constant, some add const before the type name; others add it after
* You will see both in the real world
  + A picture containing text

    Description automatically generated
* Address of non-const object can be assigned to a const pointer
  + A picture containing text

    Description automatically generated
    - In this case, we are promising not to change an object that was previously okay to change

Pointer Assignments Involving const

* You cannot assign the address of const object to a non-const pointer
* Text

  Description automatically generated
  + The second line causes a compilation error because we’re saying that we might change i through the pointer
* Exception: string literals
  + 
  + "Hello" is a const char \*, but we can assign it to a char \*

**const and Functions**

Return Values

* Prevents the caller from modifying the data returned
* Useful for efficient encapsulation
* Text, letter

  Description automatically generated
* Does not apply/make sense when returning by value
* const data returned by value can be assigned to non-const variables
  + Text, letter

    Description automatically generated
  + Works because we are not assigning the original const object – we are assigning a copy
* For data returned by pointer or reference, we almost always want to return it const
  + Text, letter

    Description automatically generated
    - Return by reference
  + Better
    - Text, letter

      Description automatically generated

Parameters

* Can pass by value, pointer, or reference
  + const makes a promise that the function will not modify the passed parameter
  + const with pass by value not very useful/meaningful
  + const with pointer passing also not very useful/meaningful
* Pass by const reference
  + Efficient – no copy needed, as with any pass by reference parameter
  + Passing const means function cannot change the referenced object
    - Exactly the same as pass by value semantically
    - Cannot modify or reassign the variable/object
    - Can only call const functions if it is an object (coming up in a few slides)
* we can accept temporaries
  + Temporary objects are always const
  + We cannot pass temporaries to a function that takes a pointer
  + Text

    Description automatically generated
  + Text, letter

    Description automatically generated

Functions

* A const member function does not modify the object it is called on
  + it does not modify the \*this object
* when we create a function in a class, the compiler quietly inserts a this parameter as the first parameter in the function signature
  + A picture containing text

    Description automatically generated
  + essentially compiles to
  + A picture containing text

    Description automatically generated
* When we call a function on an object, the compiler silently passes a pointer to the object as the first parameter
  + A picture containing circle

    Description automatically generated
  + essentially compiles to
  + A picture containing circle

    Description automatically generated
* We can declare a function to be const, in which case \*this will also be made to be const
  + A picture containing text

    Description automatically generated
  + essentially compiles to
  + Text

    Description automatically generated
* Cannot call non-const functions within the function on \*this
  + Even if those functions don’t actually change the receiver
* Reference and pointer return types must be declared const if a function is declared const
  + Text, letter

    Description automatically generated
* A const function offers a guarantee that \*this object won’t be changed by calling the function on it
  + You can always call a const function
  + You can only call a non-const function on non-const objects
* Generally, accessor functions should be declared const
  + Graphical user interface, text, application

    Description automatically generated
* What does it mean for a member function to be const?
  + Two camps:
    - 1. Bitwise constness: Does not modify any of the bits in the object (this is the C++ compiler’s definition)
    - 2. Conceptual constness: Can modify the object, but only in ways that are undetectable to clients
* Will it compile?
  + Graphical user interface, text, application

    Description automatically generated
* This leads to the notion of conceputal constness
* Adherents to this philosophy argue that a const function should be able to modify the bits of the receiving object, but only in ways undetectable by clients
* Example: the length() function of a String class
  + We want it to be const so that it can be called on both const and non- const String objects
  + We also want it to be const because, semantically, a length() function does not change the object on which it is invoked
  + However, we might want to cache the length to improve efficiency on later calls to length()
  + Text, letter

    Description automatically generated
* What can we do?
  + We want the function to be const , but we also want to be able to cache the length
  + That is, we want conceptual constness – we argue that the function should be able to change some bits of the object, but only in ways that are imperceptible to the client
* Fortunately, the mutable keyword was added to the C++ standard for just this purpose
* mutable members can be modified in const functions
  + Text, letter

    Description automatically generated
* const member functions specify which member functions may be invoked on const objects
* Member functions differing only in their constness can be overloaded
  + Text

    Description automatically generated
* By overloading operator[], we can have const and non-const Strings handled differently:
  + Text, letter

    Description automatically generated

**const Correctness**

* If
  + a pointer or reference parameter is not modified,
  + a pointer or reference member is returned,
  + a member function does not change member data,
* then it should be declared const
* Rule of thumb for const functions:
  + Start by making your functions const
  + Remove constness as needed
* Widely used in libraries
* Documents the properties of your code
  + Compiler enforces them
* Assists in system robustness
* Can allow the compiler to optimize in some instances

## W8

**Creational Design Patterns**

Two main goals:

* Encapsulate knowledge about which concrete classes the system uses
* Hide how instances of these classes are created and built

Main idea

* System at large knows about objects through their interfaces defined by abstract classes

Give us flexibility in:

* what gets created
* who creates it
* how it gets created
* when it gets created

Singleton

* Consider a class called Logger
  + Logs information to a file
  + Needed by many different parts of an application
* Ensure a class has only one instance, and provide a global point of access to it.

Diagram

Description automatically generated

* Consequences:
  + Controlled access to sole instance
  + Lazy initialization
  + Reduced name space
  + Permits refinement through subclassing
  + Permits a variable number of instances, if needed
  + Have to worry about who deletes the instance
  + std::shared\_ptr or boost::shared\_ptr can help with this

**Factory Method**

* Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
* Applicability:
  + A class can’t anticipate the class of objects it must create
  + A class wants its subclasses to specify the objects it creates

Diagram

Description automatically generated

* bad

Diagram

Description automatically generated

* better
  + This is called a Simple Factory – not a design pattern
  + Keep in mind that StudentFactory may have many clients
  + We might also have other classes that need to create students
  + This encapsulates Student creation in one class so we only have to make changes in one place when new Student types added
  + This also decouples Registrar from concrete implementations, making it much more reusable
  + Problems with this Simple Factory:
    - We’ve just offloaded the problem to a new class; instead of high coupling between Registrar and the various classes, we now have high coupling between StudentFactory and the Student classes
    - Still have to violate the Open-Closed Principle when we want to add new Student types to StudentFactory
    - The if-else block is unwieldy
    - Using strings as parameters is error-prone

Diagram

Description automatically generated

**Abstract Factory**

* Factory method allows us to create one product through inheritance
* Sometimes, we want to create families of related products
* Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
* Applicability:
  + A system should be independent of how its products are created
  + A system should be configured with one of multiple families of products
  + A family of related product objects are designed to be used together, and you need to enforce this constraint
  + You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Diagram

Description automatically generated

Consequences:

* Isolates concrete classes
  + Client controls when objects are created
  + Factory controls which objects are created and how
* Makes exchanging product families easy
  + Promotes consistency among products
  + Supporting new kinds of products is difficult

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

## W9

**Creational Patterns: Builder**

* Separate the construction of a complex object from its representation so that the same construction process can create different representations.
* encapsulate this creation process
* Diagram

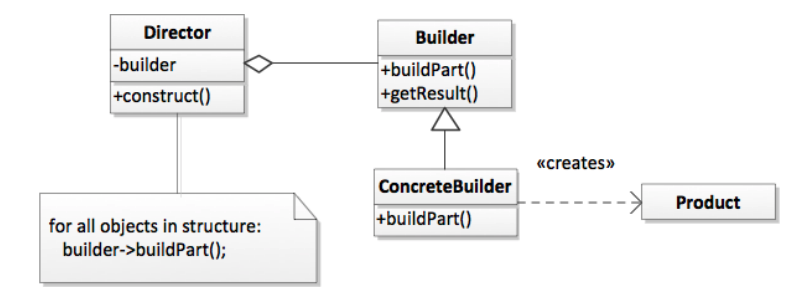
  Description automatically generated

Applicability:

* The algorithm for creating a complex object should be independent of the parts that make up the object and how they’re assembled
* The construction process must allow different representations for the object that’s constructed
* Diagram

  Description automatically generated

Classes:

* 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
* 
* Diagram

  Description automatically generated

Consequences:

* Lets you vary a product’s internal representation
* Isolates code for construction and representation
* Gives you finer control over the construction process

Builder vs. Abstract Factory

* Abstract Factory
  + Deals with families of related objects
  + Available immediately
* Builder
  + Creates one, complex product, usually made up of different parts
  + Available via getResult()

**Creational Patterns: Prototype**

* Specify the kinds of objects to create using a prototypical instance, and create new objects by copying the prototype.

Diagram

Description automatically generated

Applicability:

* When the classes to instantiate are specified at run-time, for example, by dynamic loading; or
* When instances are expensive to create, but easy to copy; or
* When instances of a class can have one of only a few different combinations of state; in such a case, it may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state

Consequences:

* Hides the concrete product classes from the client – we don’t have to know which concrete type we’re cloning
* Specify new objects by varying values
* Configuring an application with classes dynamically
* Add/remove varieties at run time from a pool of prototypes
* May reduce need for subclassing
  + Dragons, salamanders, etc. may not have to be subclasses – just generic FireMonsters cloned and then given different characteristics
* May even remove need for Factory subclasses
  + Fire object factory = generic ObjectFactory given several FireMonsters as prototypes
  + Ice object factory = generic ObjectFactory given several IceMonsters as prototypes

**Behavioural Design Patterns**

Concerned with:

* Algorithms
* The assignment of responsibilities between objects

Two types:

* Class Behavioural - Use inheritance to distribute behaviour between classes
* Object Behavioural - Use object composition rather than inheritance

Categories

* State
* Strategy
* Observer
* Command
* Visitor

**Behavioural Patterns: State**

* Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Applicability:

* An object’s behaviour depends on its state, and it must change its behaviour at run-time depending on that state
* Operations have large, multipart conditional statements that depend on the object’s state
* Usually represented by one or more enumerated constants
* Often, several operations will contain this same conditional structure

Consequences:

* Localizes state-specific behaviour and partitions behaviour for different states
* Makes state transitions explicit
* State objects can be shared

**Behavioural Patterns: Strategy**

* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Applicability:

* Many related classes differ only in their behaviour; strategies provide a way to configure a class with one of many behaviours



* You need different variants of an algorithm; for example, we might define algorithms reflecting different space/time tradeoffs
* An algorithm uses data that clients shouldn’t know about; use the Strategy pattern to avoid exposing complex, algorithm-specific data structures
* A class defines many behaviours, and these appear as multiple conditional statements in its operations; instead of many conditionals, move related conditional branches into their own Strategy classes

Diagram

Description automatically generated

Consequences:

* Families of related algorithms
* Inheritance can help factor out common functionality of the algorithms
* An alternative to subclassing
* Eliminate conditional statements
* A choice of implementations
* Clients must be aware of different strategies
* Increased number of objects

**Behavioural Patterns: Observer**

* Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
* We often have need to notify multiple subscribers about an event that occurs
  + We don’t necessarily know which subscribers may be interested in our events
  + We might want to modify the subscriber list at run time
* Example: spreadsheet application with multiple graphs
  + Need to update graphs when spreadsheet data changes
  + Graphs can be added/removed at any time

Applicability:

* When an abstraction has two aspects, one dependent on the other; encapsulating these aspects in separate objects lets you vary and reuse them independently
* When a change to one object requires changing others, and you don’t know how many objects need to be changed
* When an object should be able to notify other objects without making assumptions about who these objects are (i.e. we don’t want these objects tightly coupled)
* Diagram

  Description automatically generated

Some Observers may observe more than one subject

* We often pass in an event object containing details on which object generated the event as well as other pertinent information

Deleting a Subject should cause Observers to remove any references to the Subject

* Destructor of Subject can notify Observers of its deletion

Consequences:

* Abstract coupling between Subject and Observer; a Subject only knows that it has a list of Observers – it doesn’t know anything about them
* Support for broadcast communication (one-to-many)



* Unexpected updates
  + Observers have no knowledge of each other’s presence
  + A seemingly innocuous operation on a Subject may cause a cascade of updates to Observers and their dependent objects

**Behavioural Patterns: Command**

* Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Applicability:

* You want to parameterize objects by an action to perform
* You want to specify, queue, and execute requests at different times
* You want to support undo operations

Diagram

Description automatically generated

Classes Involved:

* Client
  + Creates the commands and sets the receiver object
  + Issues these commands to the invoker
* Invoker
  + Maintains a queue (or stack, log) of commands
  + Execution of commands is done through invoker
* Receiver
  + Implements any actions that may be executed by commands

Consequences:

* Decouples the object that invokes the operation from the one that knows how to perform it



* Commands are first-class objects; they can be manipulated and extended like any other object
* We can assemble commands into a composite command to allow for macro recording and playback
* It is easy to add new Receivers and/or Commands, because we don’t have to change existing classes



* We can have multiple receivers



* We can easily support rollback/undo operations by adding an unexecute or undo method
* Diagram

  Description automatically generated
* We can easily queue up commands to be executed as a batch
* We can easily provide progress on a set of commands being executed

**Behavioural Patterns: Visitor**

* Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Applicability:

* An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
* Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid polluting their classes with these operations
* The classes defining the object structure rarely change (or cannot change), but you want to define new operations over the structure
* For instance, we may be defining operations on third-party libraries classes to which we do not have the source code
* Diagram

  Description automatically generated

Consequences:

* Visitor makes adding new operations easy
* A visitor gathers related operations and separated unrelated ones
* Accumulating state
* Adding new ConcreteElement classes is hard

## W11

**Structural Design Patterns**

* Concerned with how classes/objects are composed to form larger structures
* Two types:
  + Class Structural
    - Use inheritance to compose interfaces and implementations
  + Object Structural
    - Describe ways to compose objects to realize new functionality
    - Added flexibility → can change composition at run-time

Categories

* Adapter
* Bridge
* Composite
* Decorator
* Flyweight

**Structural Patterns: Adapter**

* Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Applicability:

* You want to use an existing class, and its interface does not match the one you need



* You want to create a reusable class that cooperates with unrelated or unforeseen classes that don’t necessarily have compatible interfaces



* (Object Adapter only) You need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.



* Diagram

  Description automatically generated
* Diagram

  Description automatically generated

Consequences (class adapter):

* Adapts an Adaptee to Target by committing to a concrete Adapter class; a class adapter won’t work when we want to adapt a class and all of its subclasses
* Lets Adapter override some of Adaptee’s behaviour, since it’s a subclass of Adaptee
* Introduces only one object, and no additional pointer indirection is needed to get to the adaptee

Consequences (object adapter):

* Lets a single Adapter work with many Adaptees including both the Adaptee itself and all of its subclasses
* Makes it harder to override Adaptee behaviour as it will require subclassing Adaptee and making Adapter refer to the subclass instead of the Adapter itself

**Structural Patterns: Bridge**

* Decouple an abstraction from its implementation so that the two can vary independently.

Applicability:

* You want to avoid a permanent binding between an abstraction and its implementation
* Both the abstractions and their implementations should be extensible by subclassing; Bridge lets you combine the different abstractions and implementations and extend them independently
* Changes in the implementation of an abstraction should have no impact on clients
* You want to share an implementation among multiple objects and this fact should be hidden from the client
* Diagram

  Description automatically generated

Consequences:

* Decouples interface and implementation → the two are not bound permanently
* Can potentially change implementation at run-time
* Improved extensibility → can extend Abstraction and Implementor independently
* Hides implementation details from clients

Bridge:

* Structural pattern concerned with decoupling abstraction from implementation
  + Separation of interface and implementation
* Want run-time flexibility on both the lhs and rhs
* Diagram

  Description automatically generated

## W12

**Composite**

* Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Applicability:

* You want to represent part-whole hierarchies of objects
* You want clients to be able to ignore the difference between compositions of objects and individual objects; clients will treat all objects in the composite structure uniformly

Consequences:

* Defines class hierarchies of primitive and composite objects



* + Primitive objects can be composed into more complex objects, which, in turn, can be composed
  + Wherever client code expects a primitive object, it can also take a composite object
* Makes client code simple – doesn’t have to distinguish between primitive and composite objects
* Makes it easier to add new kinds of components



* + Newly-defined Leaf/Composite subclasses work automatically with existing structures and client code
* Can make the code overly general
  + Hard to restrict the components of a composite
* Note: Often used to model files and directories

**Decorator**

* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Applicability:

* To add responsibilities to individual objects dynamically without affecting other objects
* For responsibilities that can be withdrawn
* When extension by subclassing is impractical
  + We may have a large number of independent extensions possible
  + The use of subclassing would produce an explosion of subclasses to support every combination

Consequences:

* Provides more flexibility than static inheritance



* + Can add/remove responsibilities at run-time by attaching/detaching decorators
* Avoids feature-laden classes high up in the hierarchy
  + It would be inappropriate to put GZIP compression/decompression routines in the InputStream class
* A decorator and its component are not identical
  + Decorators act as transparent enclosures
  + However, we can no longer rely on object identity when using decorators
* Added complexity



* + Lots of little objects
  + Those new to Java often experience a WTF? moment when first discovering the many stream and reader classes available
  + “What happened to simple file I/O?”

**Flyweight**

* Use sharing to support large numbers of fine-grained objects efficiently.

Applicability:

* An application uses a large number of objects
* Storage costs are high because of the sheer quantity of objects



* Most object state can be made extrinsic
* Many groups of objects can be replaced by relatively few shared objects once extrinsic state is removed
* The application doesn’t depend on object identity



* Since flyweight objects are shared, identity tests will return true for conceptually distinct objects

Definitions:

* Flyweight
  + A shared object that can be used in multiple contexts simultaneously
  + Acts as an independent object in each context – indistinguishable from an instance of the object that’s not shared
* Intrinsic state
  + Stored in the flyweight
  + Information that is independent of the flyweight’s context, thus making it shareable
* Extrinsic state
  + Depends on / varies with the flyweight’s context
  + Can’t be shared

Consequences:

* The obvious: reduced memory requirements



* May introduce run-time costs (transferring, finding, computing extrinsic state)



* + Should be offset by space savings
  + Savings increase as more flyweights are shared