C++ Review

- variables are created on stack
- dynamic allocation is on heap

pointers - store address to a variable location (must assign ptr = &var)
reference - is an alternate name for a variable
arrays - are treated as pointers to beginning of memory group
→ adding integers to pointers moves pointer by int* pointer_type_size
↳ ptr-ptr = int

Dynamically allocated memory requires overloading
- copy constructor (deep copy of dynamic mem.)
- assignment operator
- destructor (must avoid memory leaks)

String Constants are all loaded into memory before a program begins - ask more

Can give funcs default values
→ must put to far right (?)

Default, everything is private

Code-level Design Features

Friend - share access to restricted members
can be: a global function
a member function of another class (must be visible)
an other class (must be visible)
→ helps with encapsulation
→ allows access to private + protected members
→ allows classes to access private variables without allowing public viewing
↳ useful for classes used to build other classes such as Node in LinkedList
→ only one-way access
→ increases coupling

Static - members belonging to a class, not an object
→ useful for info that must be shared

OO Design

Abstraction - hiding the details of implementation in functions
→ users should be able to just call a funct. and trust it
→ funct. must be named properly
→ a class must take care of all internal needs

Encapsulation - controlling access to variables + methods
private: only this class
protected: only this + child classes
public: anyone
→ use getters + setters to limit access → limit the how
→ allows inputs to be verified before applied

Modularity - functionality + communication of classes
coupling: how many other classes are linked
cohesion: how well does a class do it's job.
→ want loose connections for simple maintenance (coupling)
→ want one class to do 1-2 jobs very well (cohesion)

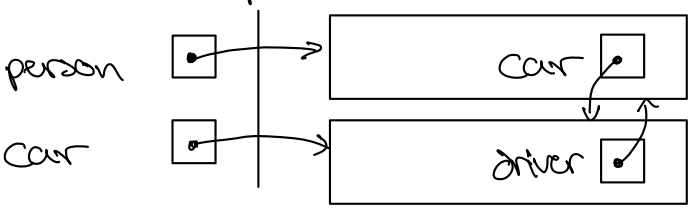
→ allows maintenance of one class to be simple
→ avoids needing to propagate changes throughout a program
→ changing body (implementation)
↳ requires recompiling module
↳ changing interface (inter-class communication) must recompile all modules involved

Heirarchy → ask more on this

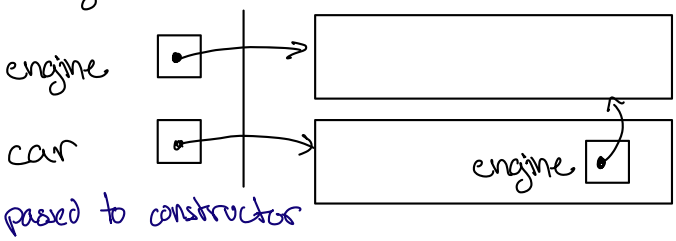
b/w objects/instances
→ initialize static members in .cpp
type Class::var_name = value;
→ no this pointer
→ can be accessed through existing class/class ptr or using Class::name
→ functions that only use static members can be static
→ in static memory

Relationships Among Classes

Association – objects containing other objects
→ can be one or two way
→ there is no heirarchy
ownership
whole-part relationship
→ all objects have own lifestyle
→ relationship must be labeled in UML

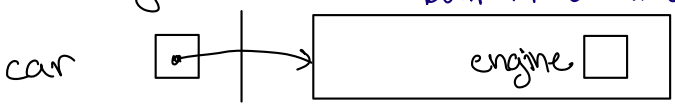


Aggregation – association where one class is more "important" than the other
→ easier to maintain than inheritance
→ object can still exist outside of class if higher class is deleted
→ memory is not solely tied to higher class



passed to constructor

Composition – strong aggregation
→ memory is solely contained within higher class
built in constructor



INHERITANCE – "is a" heirarchy
→ inherits behaviour + structure of of superclass
→ can be single or multiple

Overloading Operators → lab 2, Ex A
→ cannot define own operator
→ allows simple functionality specific to a class
→ must take at least one class arg.

return_type Class:: operator symbol (args)

Member Overloaded operators
→ member of class calls operator
string + string ⇒ string.operator+(string)
program equivalent

Non-Member Overloaded operators
→ some other class acts on existing class
ostream << string ⇒ ostream.operator<<(string)

Type Conversion
implicit: automatic
→ eg. operators which accept only 1 argument
String::String(int len);
String s = 100;

explicit: typecasting w (type)

Single responsibility – a class should have 1 job
Open-close principle – open for extension but closed for modification
Liskov substitution principle – every class should substitutably for their parent or base
Interface segregation principle – an interface should never force implementation of the unused
Dependency Inversion principle – high level modules should not depend on low level modules but abstractions

https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

---

Realization: one model implements the behaviour of a supplier
JAVA: interfaces
C++: abstract classes w pure virtual funcs.

Templates (Generics)
allow reuse of code w/o unnecessary duplication for various data types

→ all members to be inherited must be protected or public
→ base classes must be kept general for all children

class child : access_type Parent

↳ Public: inherited members maintain access level
↳ Protected: public changes to protected
↳ Private: all inherited components become private

exemptions can be done w "using"

public: using A::fun; // function from base

→ inherited members can be accessed directly or with scope resolution
Class::
resolving naming conflicts
→ can use baseclass constructor (non-default)
B(int x, int y) : A(x) { b=y; }

→ any parent can be assigned a child
↳ children can be assigned type cast parent

Virtual Functions – allow polymorphism in inheritance & solve multiple inheritance issues
→ pure virtual functions cause abstraction of a class
virtual void foo() = 0;
↳ must be defined in inherited classes
↳ cannot have an object of a parent class (abstraction)

Polymorphism put simple w virtual methods:

1. a pointer of a parent can point to a child.
2a. If a non-virtual function is called through that parent, it will check from base class down the inheritance structure to find the first instance of it
2b. If a virtual function is called through the pointer, the lowest instance of it in the inheritance tree will be called.

↳ maintenance of code is in one location → what is difference

template < class T, typename C >

- All parameters must be declared once
- Naming does not need to match b/w forward declaration & definition
- template <> comes before any other key words such as "static"

prototypes declared within a class use the class declared templates. Definitions must declare their own templates
↳ nested classes can use the templates of outer class

overloading requires different signatures

specialization – different implementation for a specific data type

Templates should not need to know what a class does
↳ class will need proper overloaded operators etc.

Instantiate template classes w <type> where type is the actual type

forward declarations tell compiler something will be defined later.

Compilation – only template classes which have been instantiated or specialized will be in compiled code

friend operators must be declared w <type_list> after the operator

Single Header Model: everything is in header
Inclusion Compilation Model: #include cpp file
Separate Compilation Model: (normal?) not supported by many compilers

→ classes containing templates must become templates
→ template parameters must be included in namespace operator before ::
↳ only for template classes & not nested classes

→ you can declare a null function in children to avoid initializing a virtual function.
void foo () {} → bad practice, use interfaces

→ dynamic memory allocation in inheritance should produce virtual destructors
↳ can operators and constructors be virtual?
this ensures that there will be no memory leaks by calling the right function for the class

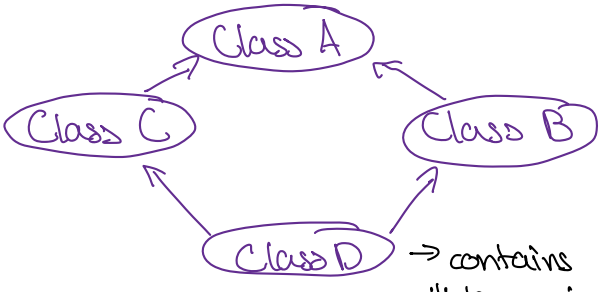
Multiple Inheritance

- inheriting from multiple parents
  ↳ in Java, we use interfaces to implement this

Class C: public B, public A {};

→ base class constructors are called in order of declaration (B then A)

ISSUE: the diamond problem
- a class only appears once in an inheritance tree but it's data members may appear multiple times



→ contains multiple copies of A

- ambiguity caused can result in improper calls

FIX: declare A as a virtual base class
↳ will be initialized by the most derived class → only D holds A

Class B: virtual public A {};
↳ most derived class will also have to call virtual base constructor
- parameters are still passed to each class but virtualization prevents initialization

- a child will inherit everything except const. destr. assgn.
↳ an explicitly defined copy constructor

STANDARD TEMPLATE LIBRARY (STL)

4 components
1. algorithms – functions specifically designed to be used on ranges of elements
2. containers – advanced data types (vector)
3. iterators – traversal objects for inside containers
4. functors – object that can be used as () like a function

ITERATORS

- object designed to travel linear containers
  ↳ must dereference iterator to access data
  ↳ will be in a nested namespace called iterator
  ↳ must use the template containers

→ can be used in for loop like we would use int i

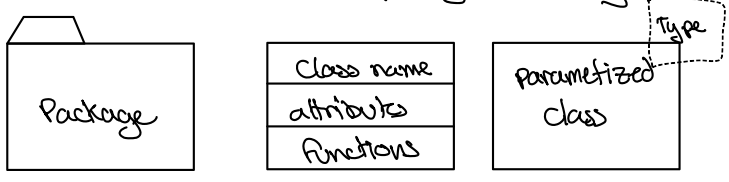
→ (++) + (--) can be used to move iterator back + forth

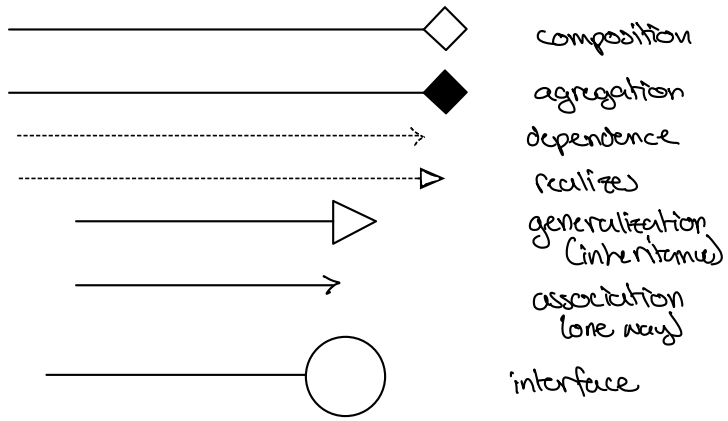
→ can be implemented to wrap around linear container etc.

UML

- – private
- # protected
- ~ package
- + public
- / derived

abstract (italisized) or {abstract}





cardinality is on the other side

static <<stereotypes>>

must copy it's base class components
↳ same as assignment operator
↳ destructors are never responsible for base class components

Design Patterns

- open close principle in action
- allow for requirements to change & grow
- minimal modification for modified functionality
- patterns developed by Gang of Four (GoF)
  ↳ why reinvent patterns

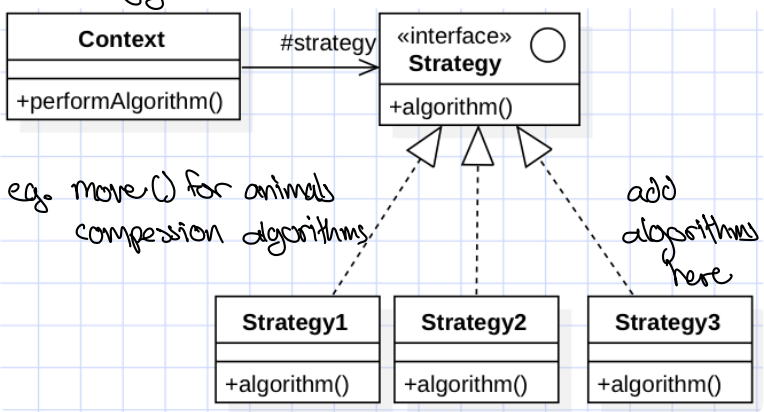
Creational Patterns
- providing a way to create objects while hiding creation logic
- Factory Method - Builder - Prototype
- Abstract Factory - Singleton

Structural Patterns
- class and object composition
- Adapter - Bridge - Composite
- Decorator - Façade

Behavioral Patterns
- communication b/w objects
- Command - Iterator - Mediator
- Observer - Strategy

1. Identify the aspects/behaviors of the application that change & those that don't
2. Create an interface for each changeable behavior.
3. For each interface, create a class that only implements that interface.

Strategy Pattern



→ separate algorithms from the class
→ main class can remain unchanged
↳ has a reference to the interface

Adapter

- adapt old code to new clients

{leaf} — final classes
→ self-related
reflexive relationships must be labeled for direction & cardinality

- Exceptions are modelled as dependencies.
- Show library methods that are inherited or implemented without needing to fill in attributes or methods.
- CONST is "read only" or "query."
- Do not represent global functions or variables (not part of classes)
- Look for inheritance first, the any ownership (composition, aggregation, association), then look for dependencies (returning the type, accepting the type in a function etc.).
- Stereotypes are like tags.
- For aggregation and composition, diamond points to the owner (class with object).

Models

- models simplify reality
- allow for low risk changes
- shows only a portion or view of the system
- together, models show the full system

Activity	Models (diagrams)
Requirements	Use case / Sequence
Development	Class / Package / Sequence
Deployment	Deployment

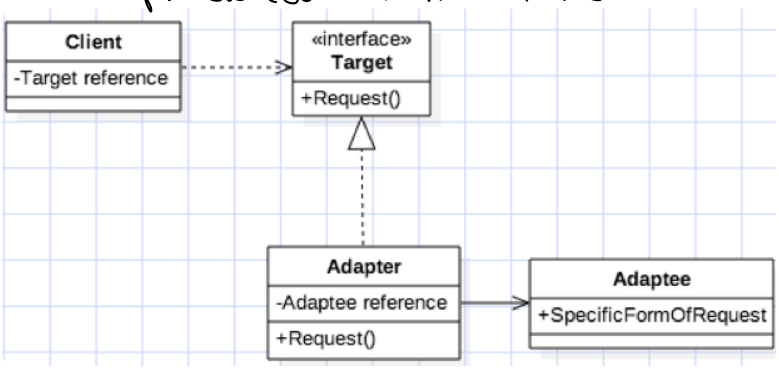
- development often deviates from models
- models can be expensive to develop
- too few models can result in poor design

Behavioural Models

- Use Case diagrams
- Interaction / sequence diagrams
- State transition diagrams
- Activity diagrams

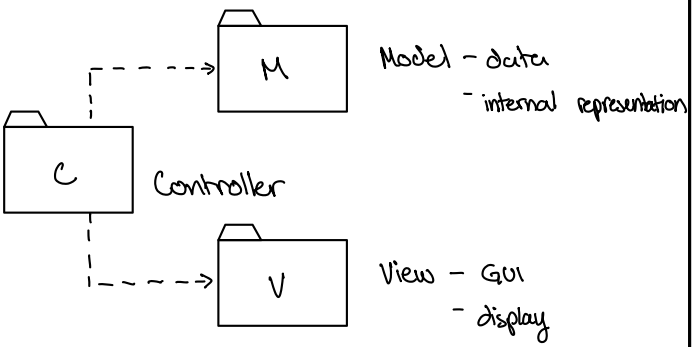
Static Models

- class diagrams
- object diagrams
- component diagrams
- deployment diagrams



→ use legacy code + modern interfaces

Observer – Model View Controller (MVC)



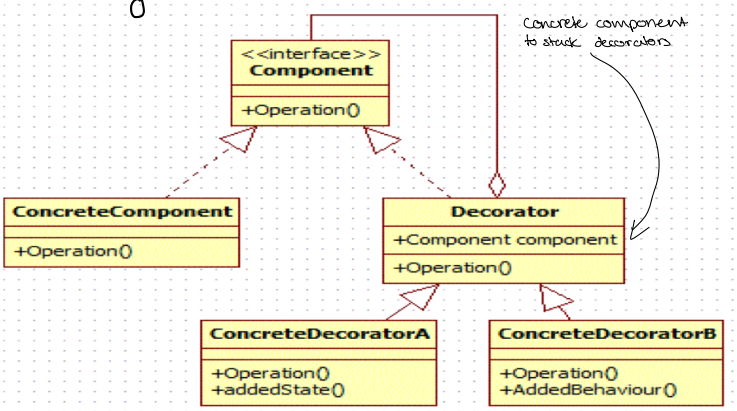
Listeners update when data modifies.

Observer Patterns
- watcher + watched

1. Create an Observer Interface with a n update method.
2. Create either an interface or abstract class for Subject that contains methods to add or remove an observer object.
3. Create a class that implements Subject
4. Create one or more class that that implements Observer:

Decorator Pattern

→ adding decorations to content



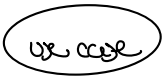
Singleton Pattern

- control access to a resource
- class calls own constructor

Use Case


Actor

Actors are external stimuli or factors
- human
- other systems
- hardware



Use cases are intended functions of a program
↳ general stories (epics)
↳ sequence of actions performed by a system that results in observable value to an actor

Activity diagrams

- flow of events within a use case
  ↳ what can happen within a use case

Classes + Class types

Boundary class
↳ interacting b/w surroundings + inner workings
↳ interfacing w actors (one for each actor–use case pair)

Control Class
↳ controls behaviour of use cases
↳ delegates work (only directing, never doing)
↳ dependent on use case – not sure what this would look like
↳ one per use case

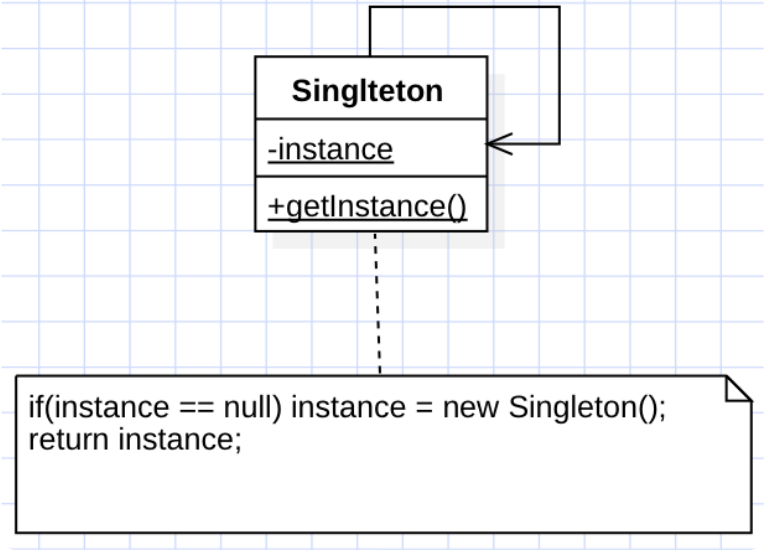
Entity Class
↳ models the system (key concepts + long lived information)
↳ can be used in multiple use cases
↳ used by controll classes but not directly tied to use cases

Associations

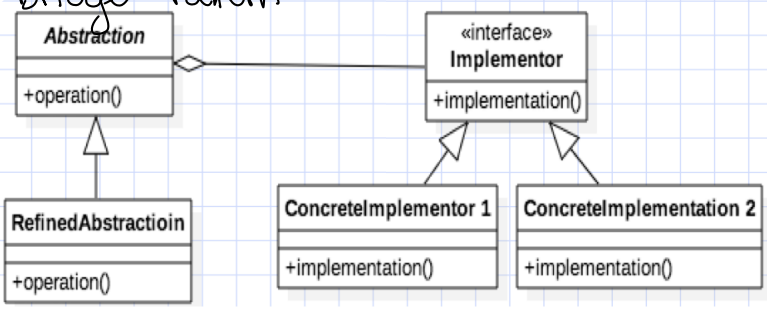
↳ multiple associations can be made to represent communication b/w classes then reduced to a single (appropriate) association.

Interaction Diagram

→ shows objects + their relationships including messages
→ sequence diagram emphasizes time ordering of messages

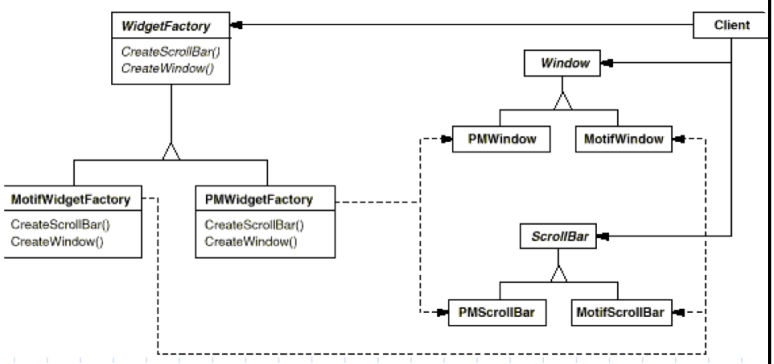


Bridge Pattern



↳ eg. textBox ↳ fill, colour, pattern

Abstract Factory Pattern
→ creational pattern
→ interface for families of related or dependent objects w/o specifying concrete class.



→ shows multiple associations & can be reduced to class diagram (w single associations)

State Transition Diagram
→ models state to state changes within an entity
→ not a flow diagram
→ arrow transitions must be labelled to trigger event.
→ initial states are mandatory, final states are not ● ◉
→ states can contain activities or be composite (contain states)

Architecture

Steps - not really sure how to apply most of this
1. Design Use Cases
2. Add new classes
3. Add classes to separate concepts
   3.1 Boundary classes
   3.2 Control Classes
   3.3 Entity Classes
4. Unify or decompose analysis classes
5. Discover class behaviour (interaction diagrams)
6. Add attributes & operations to classes
7. Consider Advanced Class Relationships
8. Multiplicity (cardinality)
9. Add navigability & Dependencies
10. Add association classes
11. Consider design patterns
12. Add tactical design decisions
    ↳ look & feel
    ↳ exception handling

→ well defined layers of abstraction
→ clear definition b/w interface & implementation
→ simplicity

→ architecture comes into package diagrams

Layering Considerations
- Visibility - dependencies should only be within a layer
- Volatility - higher layers are affected by requirements & lower layers by environment changes (hardware, OS etc.)
- Generality — more abstract ⇒ lower layers
- Number of layers - depend on system complexity

- Access - done through public classes only

Packages should
- not be cross coupled (coupled both ways)
- not depend on packages in upper layers
- not have dependencies skipping layers

Component Diagrams