CSCB07 Final Review

Version Control

- SSH: Secure Shell (Not a version control): operating network services securely over an unsecured network (remote)
- SVN: SubVersioN: uses local copy, can push to repository and download from that
- SCP: Secure CoPy (Not a version control): way to securely copy files from another PC. Transfer a copy of the file. Does not version the files

- svn status
  - ? not under version control
  - A scheduled for adding
  - C conflict
  - D scheduled for delete
  - M has been modified

- svn status -v

M	44	23	Harris	bar.c	mc accept mine
A	0	?	?	barr.c	tc accept theirs

svn cat : examine file contents
pwd : print work directory
svn log : show message

- Version Control < Centralized – "master copy" (need internet)
  Distributed – "git" (Clone)

- branching strategies ① No branching ⟶
  ② Production
  ③ main

Introduction to Java

Program: code being run on a computer to complete a task

- interpreted: translate and execute code line by line, every time when run — Python, Javascript
- compiled: translate all source one time — C

Java: compiled → interpreted
in Java every program must have a class and a main method

Decompose code: process of breaking down code into smaller task.

why? ① too big or complex to implement at once
② some code is reuseble
③ work on team, easy to distributed
④ easy to interpeted

Primitive vs Objects types

byte b = 8; // 1 byte memory int i=4000 // 4 bytes mem char c = 'x'; // 2 byte float f =2.56F; // 4 bytes boolean bool = true // 1 byte
short s = 50; // 2 bytes memory long l = 6000000L; // 8 bytes char c2 = '\u4465' double d = 3.14159; // 8 bytes

Abstraction and Decomposition

① Abstraction by Parameterization eg. def gcd(one x) variables
② Abstraction by Specification: specify what input and output will take and give without how these occurs

input → [Black box] → output

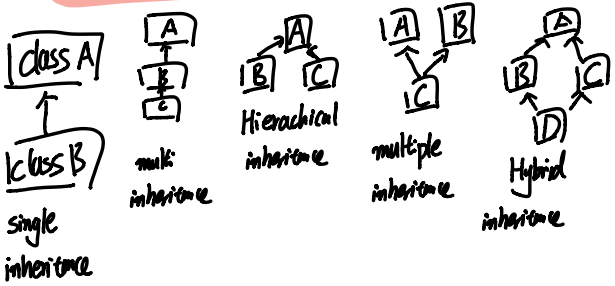
overloads: diff param ✓
diff return type X

int A[] = new int [5]
A[]- ...

int A[] = {1, 2, 3, 4}

Java modifier scope
Public ✓ ✓ ✓ ✓
Protected ✓ ✓
default ✓ ✓
private ✓

Inheritance, Generics and Casts



- Use final key word to prevent overriding

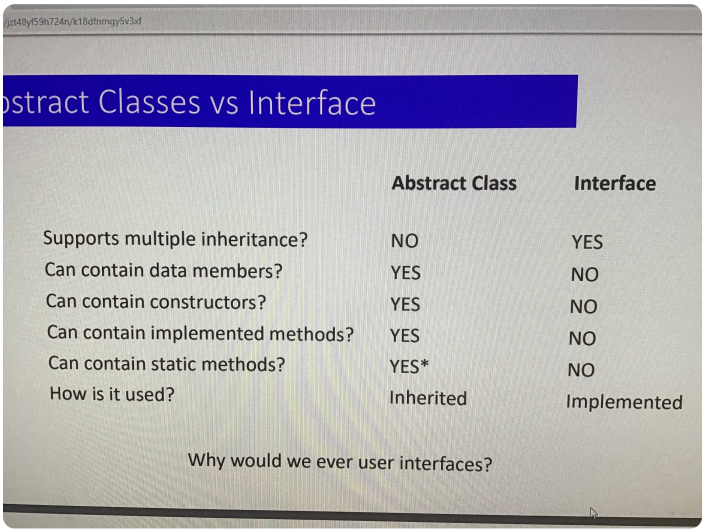
Polymorphism is the ability for an object to take on many form.

Animal x = new cat(); → object type

Reference variable

Reference variable type

Virtual method Invocation: is the invocation of the correct overriden method, which is based on the type of the object instead of the reference type



- Inheritance can override and only public method can be inheritated
- An abstract method can only be in abstract class without implementation

- polymorphism eg. food bucky = new tuna()
- override ≠ overload (same param different return x)
- Generics : public static <T> void printMe(T[] x){ }
- Casting when? ① ② move down the hierarchy
  errors < compile time error: not within the same hierarchy
  Runtime error: ClassCastException

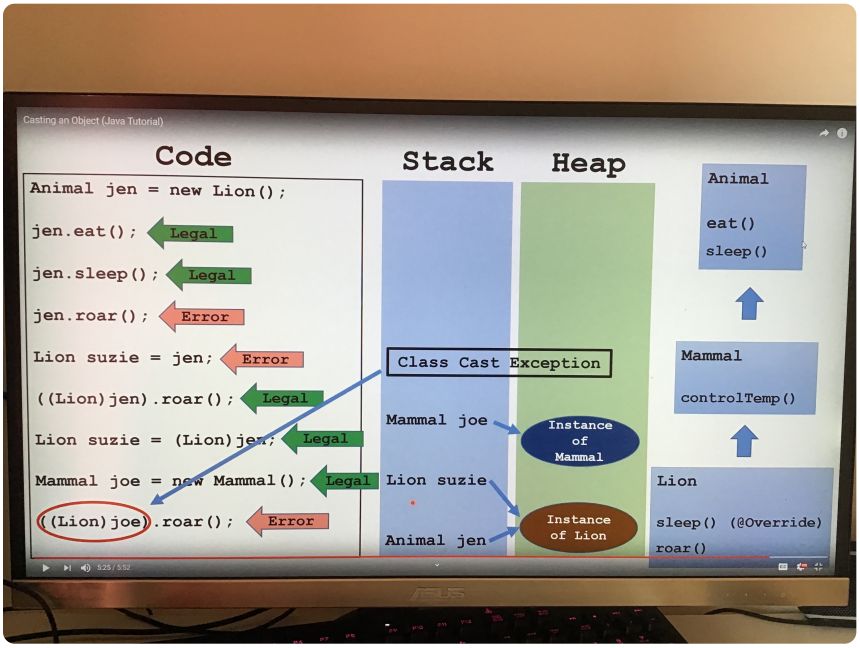
Java will upcast auto
we must downcast manually

Generic:

① why? reduced the risk of runtime error, and casting is risky

② what they do? allow a type or method operate on objects of various types

③ when? not care about the input type



Testing Practices

Two ways of testing < white box : validate highly specific path
black box : validate the "what" is correct

grey-box testing : tester is made aware of the underlying structure, but tests from "outside" the code.

1. Unit testing: validating small sections of code
2. Integration testing: validating components are working together
3. System Testing: Testing the system once fully integrated system
4. Acceptance Testing: Test that the final system is working right as was originally specified.

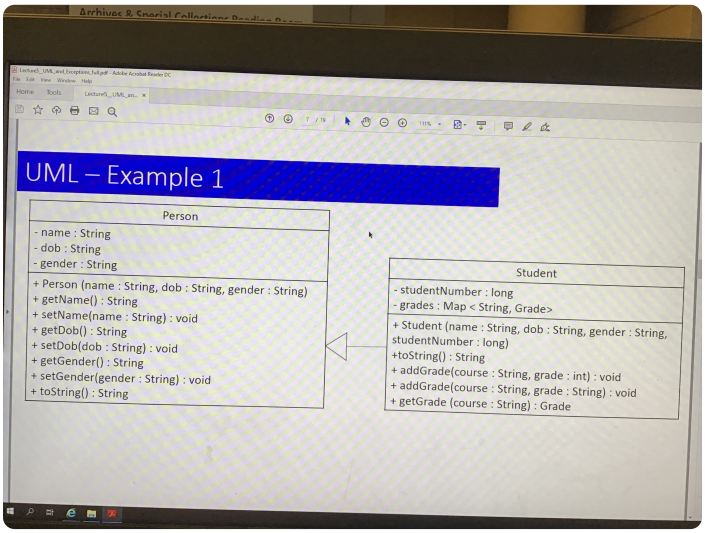
white box / black box

Test Driven Development (TDD): write what you would test to ensure that you will eventually meet the requirements

assert(expected, actual)

Object Oriented Design and Exceptions

UML: Unified Modelling language < behavioural (use case diagram), structural (class diagram)



——▷ inheritance

- - - - - ▷ interface Implementation

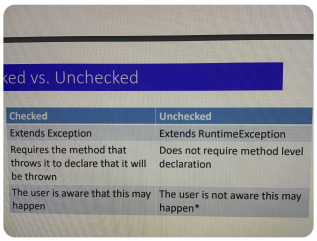
Exception: an event which occurs when a program or method behaves in a manner beyond it's normal flow

Throwable
↙ ↘
Error Exception

① Throw exception ③ upstream method
② try ... catch

checked Exception: must be handled or declared otherwise it causes compile time error

unchecked: No need to handle, won't cause compile time error



ked vs. Unchecked

Checked	Unchecked
Extends Exception	Extends RuntimeException
Requires the method that throws it to declare that it will be thrown	Does not require method level declaration
The user is aware that this may happen	The user is not aware this may happen*

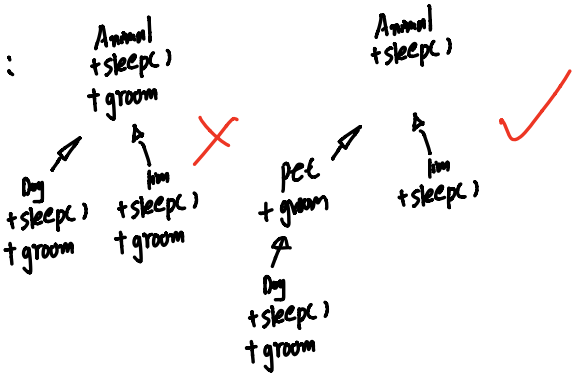
SOLID Design

Single responsibility Principle : A class should have one and only one reason to change

Open/closed Principle : open for extension but closed for modification (modify) → interface/Abstract classes should not be

Liskov Substitution Principle : a square should never be a sub-type of rectangle

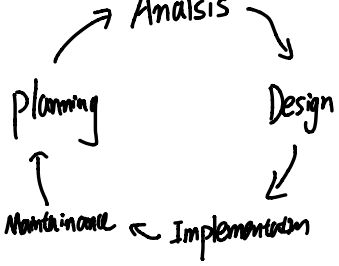
Interface Segregation Principle :



Dependency inversion Principle

High level module shouldn't depend on low level modules, but both need to depend on abstraction.

SDLC (System Development Lifecycle)



Planning: develop a plan for creating the concept

Analsis: Analyze the needs for the plan using the system. Creat detailed requirements

Design: Translate the detailed requirements into detailed design work

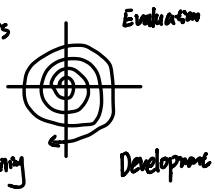
Implementation: Complete the work of developing and testing the system

Maintenance: Complete any required maintenance to keep the system running

3 ways to implement SDLS

- Spiral – risk adverse
- Waterfall – rigid timeline/budget
- Agile – Quality Deliverable/less management

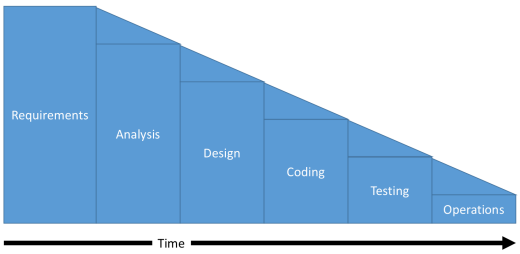
1. Spiral



– more time is spent on a give phase based on the amount of risk that phase poses for the project

2. waterfall (subclass of spiral)

The waterfall process involves a large amount of upfront work, in an attempt to reduce the amount of work done in later phases of the project. This makes it a sequential (non-iterative) model. Phases are followed in order.



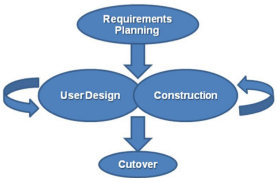
cons: – things change
– frequently time get squeezed the further process, huge pressure on development team testing team. (only Agile comes in to play)

3. Agile (subclass of spiral)

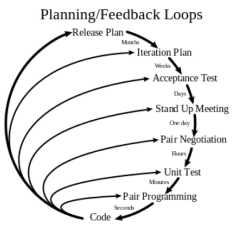
① Rapid Application Development (RAD)
– first attempt to breakaway from waterfall

The model as it was originally designed, splits work into four phases:

1. Requirements Planning – Done by a group of business owners, technical leads, and system users. Completed when all agree on what is being built
2. User Design – During this phase, users interact with analysts and dev teams to rapidly prototype out interfaces, and evolve what is being built
3. Construction – developers develop from what is found in User Design phase, and iterate
4. Cutover – Testing is done here, final handover of finished system



② eXtreme Programming (XP)
– most rigorous form of Agile
– building a series of feedback loops



The Agile Manifesto

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

Agile vs Waterfall

	Agile	Waterfall
Iterative?	Yes	No
Late Changes?	Yes	No / $$$
Fixed timeline?	No*	Yes
Fixed Cost?	No*	Yes*
Volume of meetings	Consistent	Heavy up front, reduced middle, heavy end
Release frequency	Every Sprint	Once per project
Business Involvement	Heavy throughout	Heavy early, and at very end
Cost to fix mistakes	Low	High

Design Patterns

def: a general description of the solution to a well defined problem using an arrangement of classes and objects

Split in three major groups:

1. Creational – Patterns that deal with the mechanics of object creation
2. Structural – Patterns that deal with creating simple ways of building relationships between objects
3. Behavioural – Patterns that deal with common communication between objects

4. Architectural

1) Creational Patterns

Pattern 1: Factory Method

why?

Problem: Creating new objects often requires complex process that are not really appropriate to expose to a client.

why?

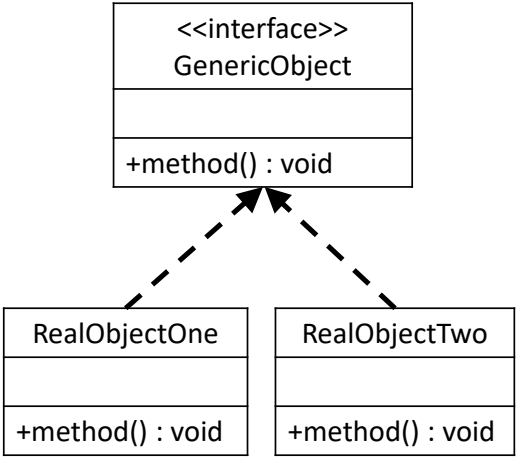
Motivation: We want to try and keep things as abstract as possible, don't expose instantiation logic to end user, use a common interface to refer to all similar objects

when?

Applicability: Any time we have multiple ways of realizing the same concept – and we want our users to decide which implementation they want to use, but we do not want to expose them to the nitty-gritty details

```
if (input.equals("SQUARE")){
  System.out.println("Make a square");
  Square shape = new Square();
}else if(input.equals("CIRCLE")){
  Circle shape = new Circle();
}else if(input.equalsIgnoreCase("RECTANGLE")){
  Rectangle shape = new Rectangle();
}else{
  Object shape = new Object();
}
```

This code is messy, and needs to use lots of copy paste!



ObjectFactory
+createObject(type:String) : GenericObject

Note: The Object factory createObject method will return the interface type, but using polymorphism, will actually use the specific object requested

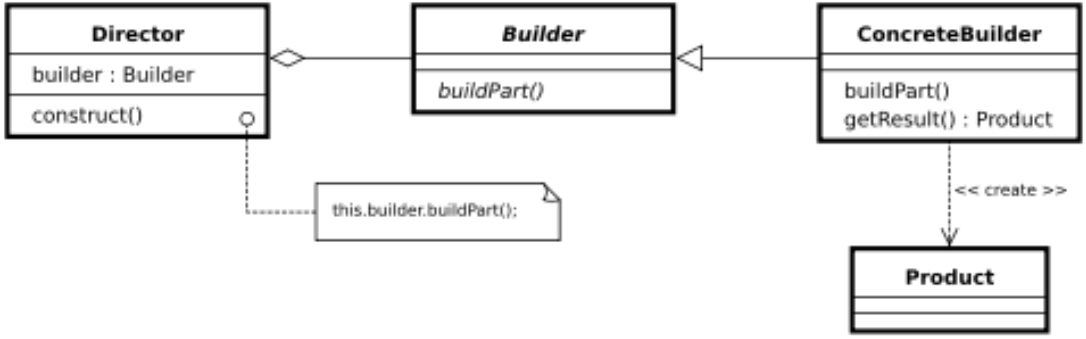
[DEMO – Shape Factory!]

Pattern 2: Builder

Problem: Telescoping constructors – we have too many options for a constructor, that need many different variables, we also need to remember the order of these

Motivation: We want to have a single way of initializing objects that is simple for the user to follow, and still allows our objects to be immutable to avoid inconsistency

Applicability: If there are many potential ways to construct the same object, and sometimes we will need many different constructors, this can be a good pattern to follow.



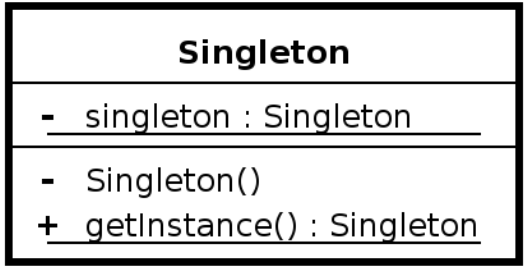
[NOTE: Demo this using a pizza!]

Pattern 3: Singleton

Problem: Sometimes we only want a single instance of an object

Motivation: either due to efficiency or due to real world behaviour, we only want a single instance to ever exist of a specific object.

Applicability: Probably the most misused pattern! Should only be used for one of the above two stated reasons. Often gets paired with Builder or Factory, in order to reduce memory footprint on the JVM



[DEMO – PrimeMinister]

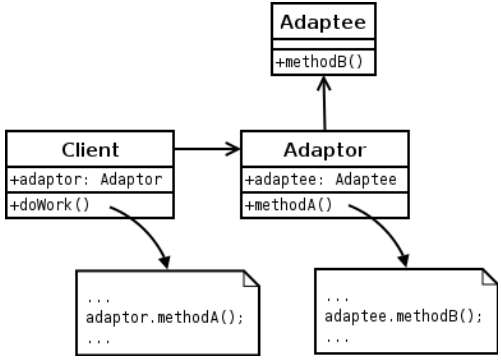
(2) Structural Patterns

Pattern 1: Adapter / Wrapper

Problem: Sometimes the objects we currently have do not match what our clients are expecting, and we want to make something that can convert them into what is desired
Motivation: We want to be able to use already existing code as frequently as possible, and sometimes we need to adapt it to plug into another persons code.
Applicability: When there is currently one or more interfaces that have things in a format other than what we want, and we desire our clients to be able to use them

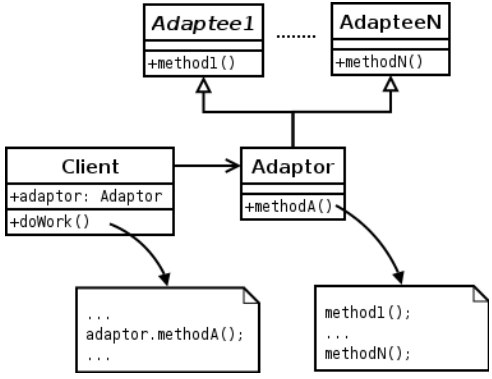
NOTE: There are two forms of this pattern – a simple one, and a complex one ☺

Object Adapter (aka simple Adapter)



Put simply, the client wants to call something called methodA(), but the interface we have calls it methodB(), so we make an adapter that let's the client do what they want.

Class Adapter (multiple Inheritance)



③Behavioral Patterns

Pattern 1: Iterator

why?

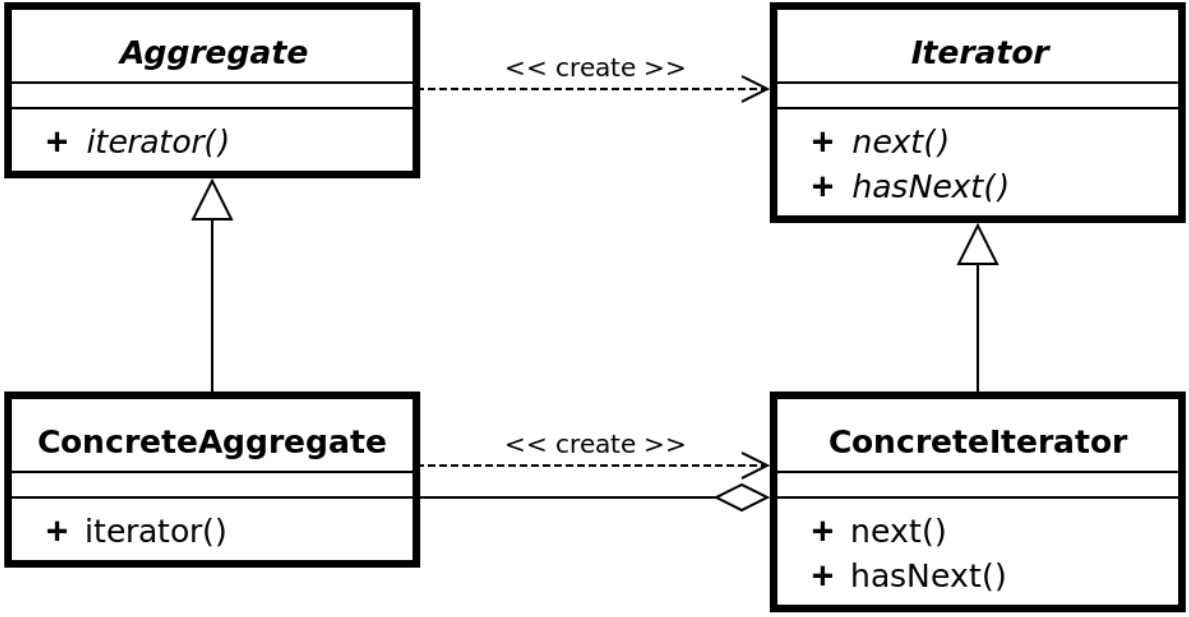
Problem: We want to be able to see the objects stored within an aggregator sequentially, but do not want to expose the underlying representation

why?

Motivation: Often there are algorithms for doing specific iterations on various different types of container objects, and we want to decouple those algorithms from the container itself.

when?

Applicability: When we have an aggregator and want to iterate through the objects in it, using an algorithm that is not container-specific.

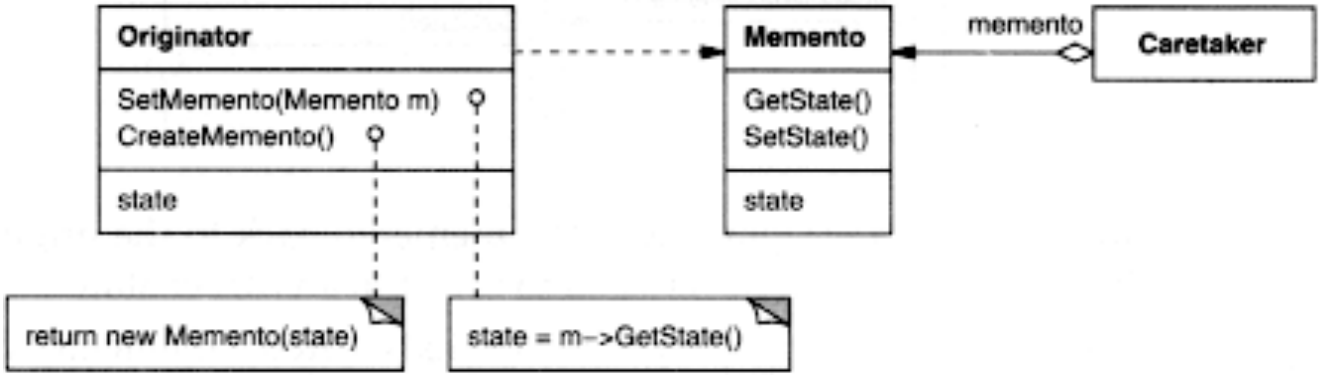

Aggregate
+ iterator()
<< create >>
Iterator
+ next()
+ hasNext()
ConcreteAggregate
+ iterator()
<< create >>
ConcreteIterator
+ next()
+ hasNext()

Pattern 2: Memento

Problem: We want to be able to revert to a previous version of an object if something goes wrong

Motivation: When a client changes things, there may be potential that something else fails because of that change, and they require to revert. Memento gives them this ability.

Applicability: When you have an object whose state may need reversion.

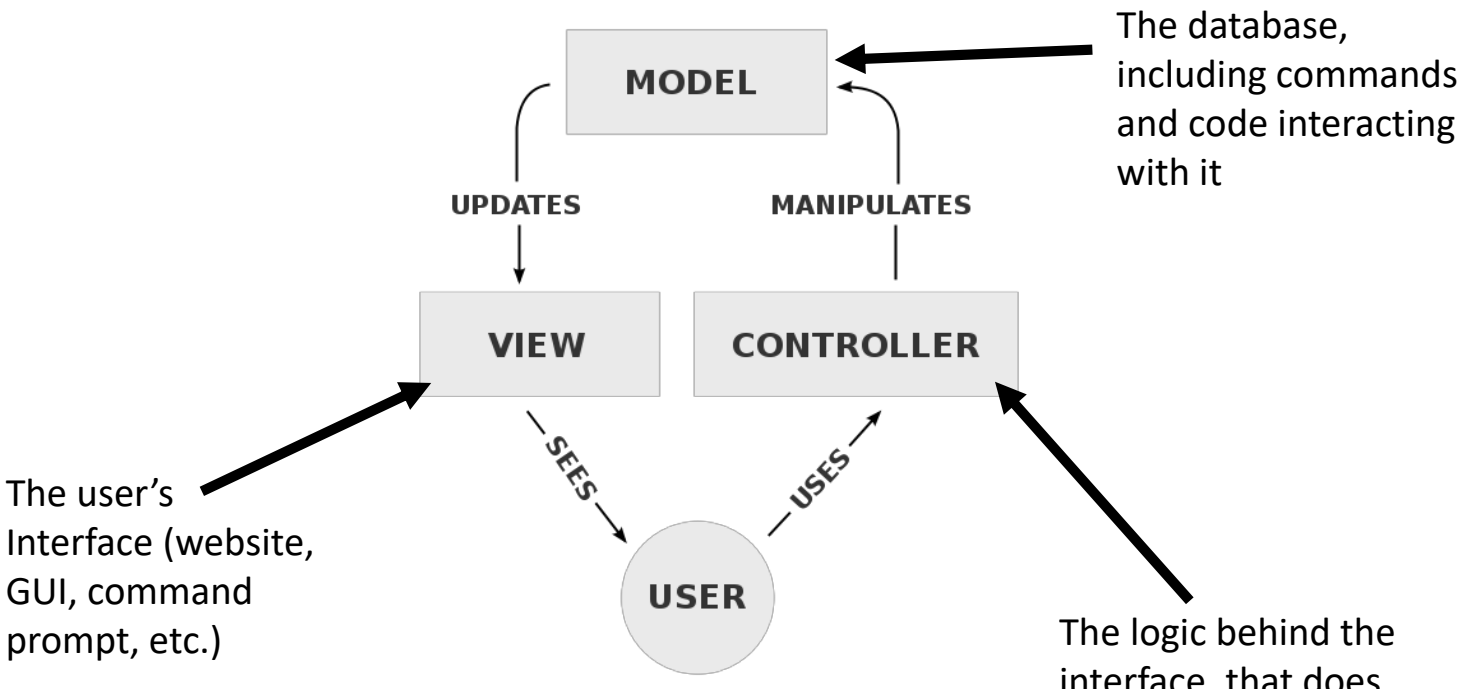

Originator
SetMemento(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
memento
Caretaker
return new Memento(state)
state = m->GetState()

Architectural Patterns

Model View Controller (MVC)

This isn't really a "design pattern" in the traditional sense, but it is an important topic. Architectural design helps create clean and consistently working code.

MVC dictates breaking the code into three key areas:
Model – The models being worked on, usually mimics the structure of your data model. This is also the layer that logic about the domain sit in.
View – This is the output representation of information that the user interacts with. Think the application screens or webpages
Controller – Accepts inputs, and converts them to commands that are delegated to either the model or view, or both.



The database, including commands and code interacting with it

The user's Interface (website, GUI, command prompt, etc.)

The logic behind the interface, that does the work or the view (the non-graphical front end)