What does Software Design Include?

- Choosing how to abstract from the real-world and what to abstract (abstraction)
- Making code that is structurally easy to work with and build off of
    - *Stable* – no errors when more features are built
    - *Extendable* – additional features can be built onto it
    - *Clean* – unrelated features aren't tangled up
    - *Communicable* – expresses your intentions about its usage
    - *Efficient* – minimize repetitive code, save coder's time, simplify groupwork
- There are design **principles**, not **rules**, because they aren't universal to all circumstances. They're general guidelines that may contradict each other; compromise is needed.

Object-Oriented Programming (OOP): Programming paradigm based on "object" with data/fields/variables and code/procedures/methods/functions. In many languages, objects are classes.

Features of OOP

Abstraction: Distilling a concept to set of essential characteristics

- Objects contain *some* features of real-world physical objects that it models
- Objects higher in inheritance hierarchy have less defined features, more "abstract"
- Objects closer to back-end more "abstract", interact less with user

Encapsulation: Accessing data indirectly with methods, and hiding internal representation

Inheritance: When subclasses inherit features from the higher class

Polymorphism: Ability for a function/expression to apply to subclasses/implementers

Coupling: How much a class is linked to another class

- *High coupling* means changing one class might change another class
- *Low coupling* is an ideal goal

Cohesion: How much a class's features belong together

- *Low cohesion* means class methods do unrelated tasks
- *High cohesion* means class methods have strongly-related functionality

Object-Oriented Design Principles:

Single Responsibility Principle

- Classes have 1 one specific job. Inner code shouldn't be relied on for multiple jobs
- Create separate classes to handle separate jobs, then create an interface class

Open/Closed Principle

- Code should be open for extension, closed for modification
- If you extend new features, you should be able to add it without modifying anything else

Liskov Substitution Principle

- Classes should be substitutable with subclasses without changing anything

Interface Segregation Principle

- Use an interface to hide irrelevant methods and classes that won't be used by the client

Dependency Inversion Principle

- Less stable components should dependent on stabler components. Both should dependent on abstract classes. Idea is to decouple the system.

- **Depend**: When a class method calls instances of another class (or its variables/methods)
    - If A depends on B, who has a child C, A "kind of" depends on C.
    - If A depends on B, who is implemented by C, A does not depend on C.

Reuse/Release Equivalence Principle

- What you publish should be reusable as a cohesive unit, not random, unrelated classes

Common Closure Principle

- Components should be related classes that change for the same reason at the same time.

Common Reuse Principle

- You shouldn't depend on a component with classes you don't need

Acyclic Dependency Principle

- You shouldn't have dependency cycles (eg. A depends on B, B depends on C, C depends on A), otherwise making changes to any component will cause problems

Version Control and Git

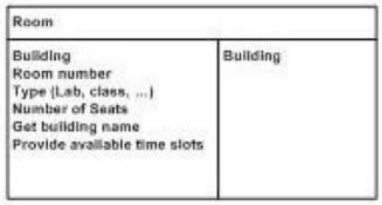
Enter commands in the program "git bash" to operate git.

Clone the master repository into a local repository *(to start your project)*	`git clone <url>`
Add any changed files onto a list of things you will commit	`git add file1 file2`
Commit the list of things you've added	`git commit -m "did something"`
Push to the master repository everything you've committed	`git push`
Pull from the master repository the changes any other people made	`git pull`

Four possible **statuses** (`git status`) of files in local repository:

1) **Untracked:** File not originating from git *(ie. you created this file in the local repository)*
2) **Tracked:** Files originating from git *(ie. you received this file from git push)*
3) **Staged:** Files affected by **add** and are ready to be committed
4) **Dirty/Modified:** Files that have been modified but not **added**

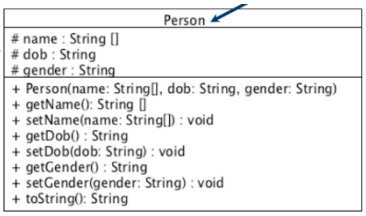
Class, Responsibility, Collaboration (CRC) Card: A way to brainstorm and visualize how many objects interact. "What" a class should do. No code should be written in this.



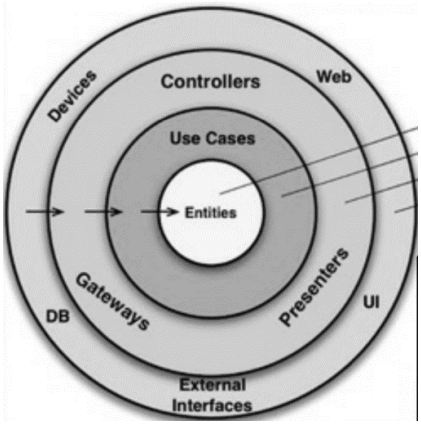
- *Top:* Name of class, inheritance/implementing
- *Left:* Responsibilities
- *Right:* Classes it collaborates with

Unified Modelling Language (UML) Diagram: Like CRC, but shows inner workings of a class. "How" a class does what it does

- *Variables:* `name: type`
- *Methods:* `name(p1: type1, p2: type2, …): returnType`
- *Visibility:* `-` (private), `+` (public), `#` (protected), `~` (package)
- *Static:* `static`
- *Abstract:* *`abstract class/method`*, `<<abstract class>>`
- *Interface:* `<<interface>>`
- *Relations:* → (point to what you inherit from), ⇢ (point to what you implement)



Clean Architecture



There are many program architectures, Clean Architecture being one of them. It divides classes into layers.

- **Dependency Rule:** Classes in a layer can depend on classes in the same layer or the layer directly below.

Entity: Classes of real-life things; the building blocks.

- Class variables should be private; methods should be getters, setters or small methods. Ideally doesn't depend on other entities – use String IDs if possible
- eg. *Student, Person, Course, Seminar, Shape*
- *BTW:* Enums count as even more low-level then entities. Think of them as Strings.

Use Case: Classes/methods focused on using entities, manipulating them meaningfully.

- *Use cases* are the situations in a program where an entity is used.
- *Use case classes* are classes that contain use case methods. Ideally depends on just 1 entity
- eg. Use cases of *Student* include enrolling/dropping courses, ordering transcripts; similar use cases should be bundled up into manager classes
- eg. *StudentManager* – stores students, changes student information like enrolled courses

Controller: Receives input in frontend, calls the necessary use cases in backend to do a task

- eg. Controller *StudentEnrollmentSystem* receives gets user input to enroll in CSC207
  It sends the input student number to use case *StudentManager*, with list of students
  StudentManager finds the right *Student* entity, checks if the student is enrollable
  Then the controller sends the same information to use case *CourseManager*.
  CourseManager checks if there's room in the course and enrolls the student.
  Then the controller returns true.

Presenter: Classes that convert back-end information to front-end display (UI)

- eg. Presenter *ColourPresenter* calls retrieves a user's particular theme by calling a *UserManager* method, then returns the right colours and theme to the UI.
- eg. Presenter *MainMenuPresenter* has children *EnglishMainMenuPresenter*, …, for different languages.

Gateway: Anything that interacts to things outside the program

- eg. Storing to .csv/.txt file or database, reading information from a file, use a printer, interact with external interfaces (eg. APIs)
- eg. Barcode scanners, retrieves physical information from outside, read it, convert it to something usable in the program

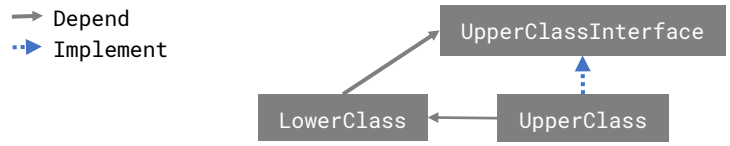
Frontend	*UI* *External Interfaces* (eg. other website APIs) *Database* (eg. csv/txt files)	*Web* (eg. auto-check weather) *Devices* (eg. printer)
	Controller Gateway	Presenter
Backend	Use Cases	
	Entities	

Interface Segregation Principle: Idea that interfaces should have minimum methods without forcing the implementation of multiple interfaces for the same service.

Dependency Inversion Principle: In Clean Architecture, this is the Dependency rule. High-level modules should not depend on low-level modules. Detailed implementations should depend on abstractions.

Introduce abstraction layers in the form of interfaces between low-level and high-level classes to reduce coupling. This way, pieces can be changed without changing other pieces

- Use when a lower layer needs to depend on an upper layer. Have the upper layer implement an interface that the lower layer depends on.



- Ideally, all Use Case/Controller classes should have an interface and only reference each other through those.
- Use when changing a lower layer requires changing an upper layer (too much coupling!)

```
interface GatewayInterface
class Gateway implements GatewayInterface
```

In a controller, to access the gateway…

```
GatewayInterface g = new Gateway();
```

In a use case class, to access the gateway…

```
public void coolMethod(GatewayInterface g)
```

Code Smells

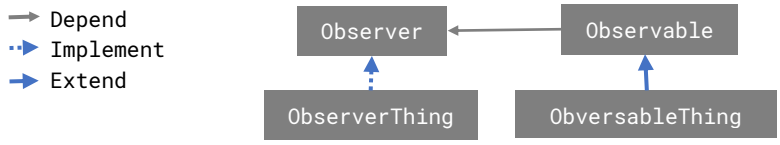
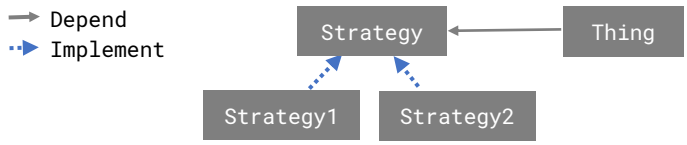
Bloater	*Long Method* *Large Class* *Primitive Obsession* *Long Parameter List* *Data Clump*	10+ lines Too many methods/variables Primitives instead of small class 4+ method parameters Identical variable/parameter groups
Object-Orientation Abuser	*Alternative Classes, Different Interfaces* *Refused Bequest* *Switch Statements* *Temporary Field*	Same class & purpose, different method names Subclass doesn't use all parent methods Switch/many ifs. Use polymorphism instead Variables filled only in special cases, else empty
Change Preventer	*Divergent Change* *Parallel Inheritance Hierarchies* *Shotgun Surgery*	Change unrelated methods when changing class Change many classes when changing class Create other subclass when creating subclass
Dispensable	*Comments* *Duplicate Code* *Data Class* *Dead Code* *Lazy Class* *Speculative Generality*	"Best comment is a good method/class name" "Merge duplicate code pls" Classes with only variables, getters/setters Obsolete, no longer used code Class too small, doesn't do enough Code created "in case" of something, never used
Coupler	*Feature Envy* *Inappropriate Intimacy* *Message Chains* *Middle Man*	Access other class's methods more than itself Class uses other class's internal data a() -> b() -> c() -> d() Classes that only delegate work to others

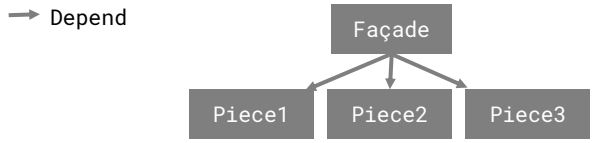
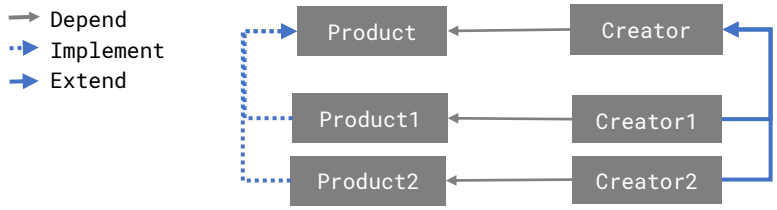
- *Note:* Many design patterns have code smells; that is fine.

Composition: When a class contains another class as a variable
Inheritance: When classes have a parent/children relationship, directly extending

- Use inheritance when you also need one class to already be an instance of the other (thus you can cast or call parent methods, etc.)

Design Patterns
Antipattern: A piece of "bad code" that a design pattern is meant to simplify/make easier

Iterator	*Need multiple ways to iterate over a class* *Hide implementation of the iterating because it's bloating/reducing cohesion* Create interface `Iterator<T>`, create implementors In your class, add `Iterator<T>` as an object
Observer	*A class has a method that should affect another class, but it shouldn't directly call/depend on the other class.* Get a class to implement `Observer`, with `update(Argument)`, which updates itself in some way. Get a class to extend `Observable`, which contains a list of `Observers`, options to edit observers, & `notifyObservers()`, which calls `Observer.update(Argument)` for each of its `Observers` You'll have to manually add `Observers` to an `Observable` somewhere else. *eg. Detecting unsaved changes made* 
Dependency Injection	**Hard Dependency:** When a class directly instantiates another class and uses it **Dependency Injection:** When a class uses another class in method arguments Prefer dependency injection over hard dependency, because… - Harder to test the class independently from the class it's inside - Make code more flexible to allow passing in subclasses
Strategy	*Many methods with same purpose, but slightly different functionality. Want them to be usable in many contexts, or decouple them. Split them into classes:* Get an interface `Strategy`, create implementors that do a thing In a main class, create `Strategy` field and setter for it. *eg. Different sorters for books (Length, alphabetically, ISBN, etc.)* 
Façade	*Big/bloated, multi-functional class with non-cohesive pieces that can be separated from each other and are big enough to stand on their own class.*

	Split the big pieces into smaller, specialized classes Create a Façade class that delegates responsibilities to those smaller classes. 
Factory Method	*A class wants to instantiate various subclasses/implementors of a parent class/interface* Create a `Product` interface implemented by many `ConcreteProducts` Create a `Creator` class with abstract `createProduct()` that returns `Product` Create many `ConcreteCreators` that inherit `Creator`.  **Abstract Factory:** For creating related "types" of different objects. - Create an `AbstractFactory` interface with multiple `createProduct()` methods that return different abstract products (that are related) - Implement `AbstractFactory` with various `ConcreteFactories` that build implemented/inherited versions of those abstract products
Builder	*We want to create a complex class in a part-by-part fashion; or, the process for creating some class is getting too bloated and we want to hide it.* Split the complex class into many classes if necessary and if not already. Create a `Builder` with methods that create the structure in the right order and returns the whole thing. Many ways to do this: 1) `Builder` has private instances of A, B, C, which can be set to something with `buildA()`, `buildB()`, `buildC()`, then `getResult()` creates a new `Result` object, sets `Result`'s properties to A, B, C, then returns it 2) `getResult()` directly calls `buildA()`, `buildB()`, and `buildC()`, and then instantiates the `Result` object and sets its properties. 3) `Builder` has a private instance of `Result` which is auto-instantiated. `buildA()`, `buildB()`, `buildC()`set A/B/C to something and also set Result's A/B/C parameter. `getResult()` only returns `Result` When you want to build something, instantiate the `Builder` and call the build/getResult methods in the right order. If you find yourself doing this repetitively, or if your building process can be done more than one way, then consider creating a `Director` class, who just has methods that call builder methods in the right order.
Adapter	*We want to re-use a class from somewhere else, but doesn't fit perfectly.* **Object Adapter** – Create an adapter class with an instance of the adaptee class. Adapter implements methods required for the program to work.

	Class Adapter – Create an adapter class that extends the adaptee class, but also implements methods required for the program to work.

Legal Accessibility Requirements

- Color isn't the only way to show visual elements like info, actions, prompts, etc.
- ≤ 3 colour flashes/second (for epilepsy)
- ≥ 3 second long audio that auto-plays should be pauseable/stoppable/controllable volume-wise.

7 Principles of Universal Design (not just software)

1) **Equitable Use**

Useful/marketable to people with diverse abilities – means of use is identical for all if possible, equivalent if not.

2) **Flexibility in Use**

Accommodates individual preferences/abilities

3) **Simple & Intuitive Use**

Easy to understand, regardless of user experience/knowledge/language/concentration

4) **Perceptible Information**

Legibly communicated information – visual contrast between info and background, compatibility with colourblind, subtitles for deaf

5) **Toleration for Error**

Minimizes hazards/accidental/unintended actions – make most-used elements accessible, shield hazardous elements

6) **Low Physical Effort**

Using it doesn't fatigue you – minimize repetition, sustained effort, awkward body positions

7) **Size and Space for Approach and Use**

Should be the right size and in the right place to be used, regardless of user's body size, posture, or mobility (not very applicable to software)

Regular Expressions (Regex)

Regex: An expression that filters strings. More rules here

^	Gets the expression to search strings from their beginning
$	Gets the expression to stop searching any string at its end
[…]	A list of acceptable characters. Pick one character from it.
[^…]	A list of unacceptable characters. Pick one character not from it.
-	Denotes a range of characters/letters (eg. a-z is all lowercase, 0-9 is all digits)
*	0+ of the thing directly before the symbol is allowed.
+	1+ of the thing directly before the symbol is allowed.
?	0 or 1 of the thing directly before the symbol is allowed.
{2}	Exactly 2 of the thing directly before the symbol is allowed.
{,2}	At most 2 of the thing directly before the symbol is allowed.
{2,}	At least 2 of the thing directly before the symbol is allowed.
{2,4}	2-4 of the thing directly before the symbol is allowed.
\	An "escape", which cancels out the syntactic meaning of a character in order to directly use it as a character in the regex
.	Any character
\d	Any digit (0-9), [0-9]
\D	Any non-digit/letter, [^\d]
\s	Any white space character [\t\n\x0B\f\r]
\S	Any non-white space character [^\s]
\w	Any word character [a-zA-Z_0-9]
\W	Any non-word character [^\w]
(…)	A capture group. Pick everything inside it.
(A(B)) \1	\1 means you must pick what you picked for A(B), the first capture group detected.
(A(B)) \2	\2 means you must pick what you picked for B, the second capture group detected.
|	Or, disjunction, union
&&	And, conjunction, intersection
?(A)X|Y	If regex A appears, X must follow; else, Y must appear

Escape Characters

\t	Tab	\'	Single quote	\(	(
\b	Backspace	\"	Double quote	\)	)
\f	Form feed (printers)	\\	Backslash	\{	{
\v or \x0B	Vertical tab			\}	}
\r	Carriage return			\[	[
\n	New line (enter)			\]	]

- `^[a-z][a-zA-Z0-9]*$` describes camelCase.
  - The first character must be lowercase. (a-z)
  - You may then have 0+ of any lowercase/uppercase character/digit.
- With ^…$ (anchors), the regex strings that fit the regex exactly.
- Without ^…$ (anchors), the regex accepts any strings with substrings that fit the regex.