Multiprocessing Creational Patterns 2: Lazy Initialization & Builder

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 13

AsyncIO and Threads

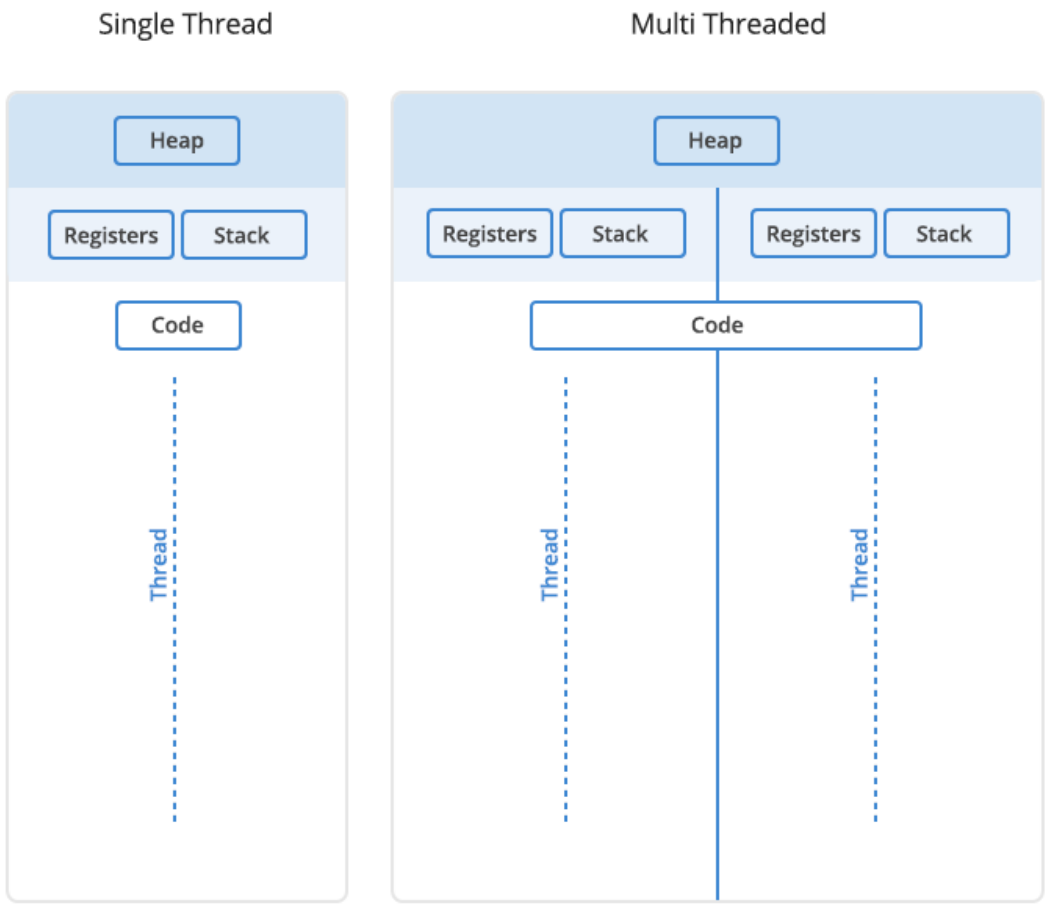
Not really concurrent

AsyncIO runs on a single thread and is sequential with perceived concurrency.

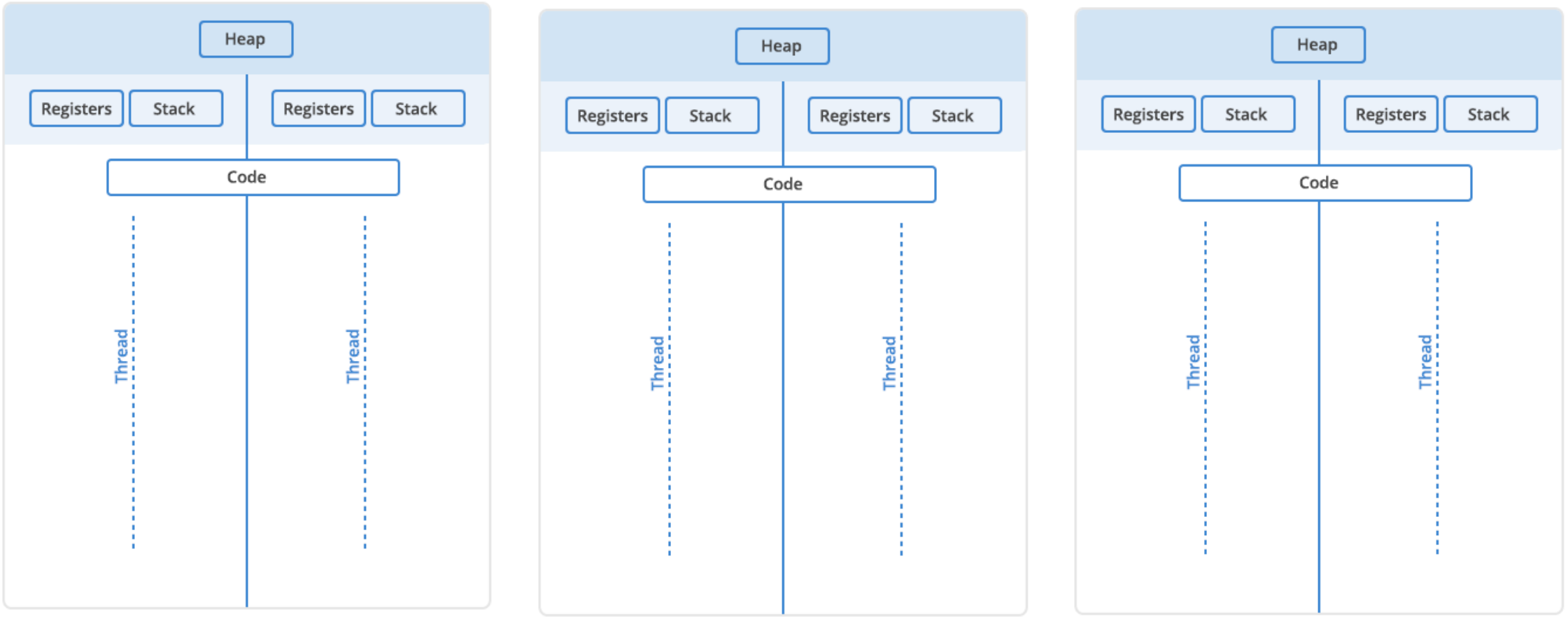
Threads, while having the ability to run in parallel on multiple cores, is limited by the Global Interpreter Lock (GIL). They run sequentially but they switch out so fast that they lend to a sense of perceived concurrency.

How can we achieve true concurrency?

What about CPU bound problems?



Solution – Multi-Processing



Multiprocessing – TRUE parallel processing

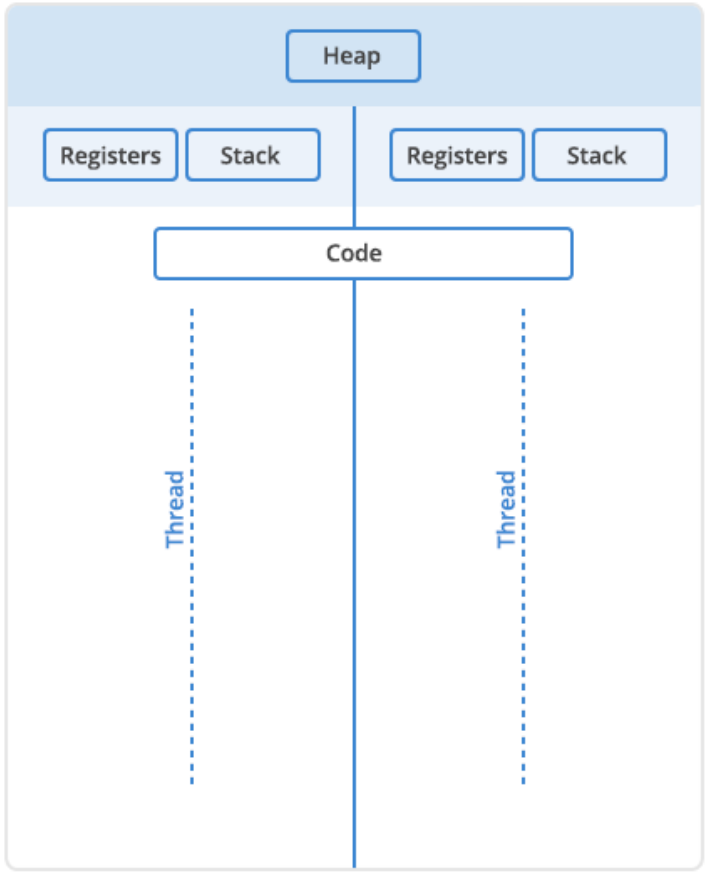
A process is a collection of resources, memory and program code.

A process contains a unit of execution called a thread which process program code in a sequence.

A process can have more than one thread.

Each process can run on a separate core and have it's own instance of the python interpreter.

This is true parallel processing. Code will actually be running in parallel and at the same time. This is what demarcates Multiprocessing from Multithreading.

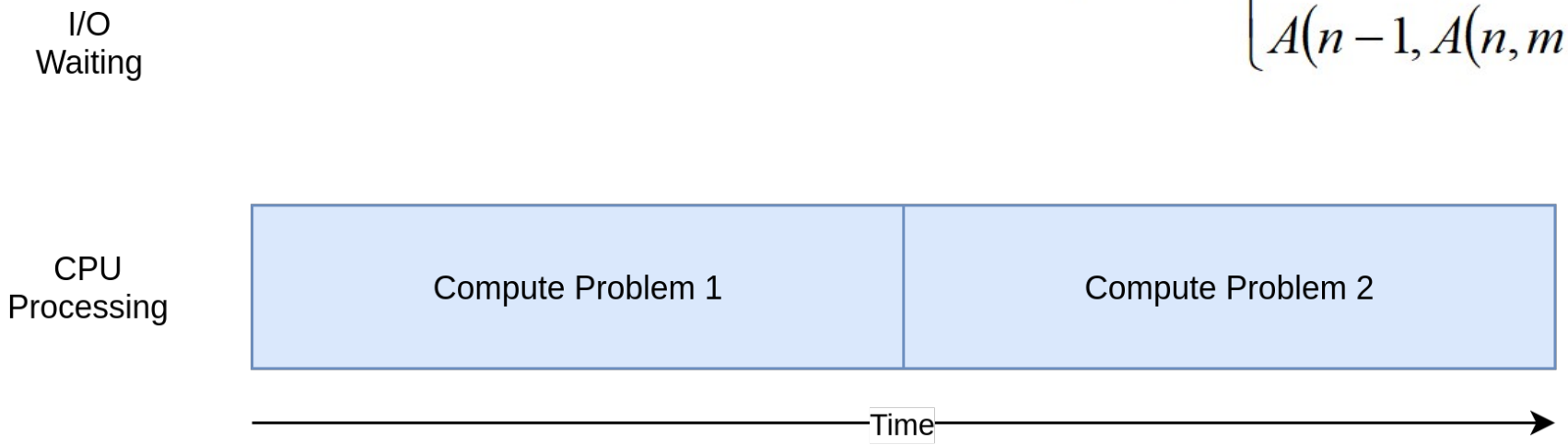


RECAP: CPU Bound

A CPU Bound **problem spends the bulk of it's time processing data** instead of waiting on Input/Output.

Remember Ackermann's Algorithm? We looked at it while learning about profiling. That is a CPU Bound problem.

$$A(n,m) = \begin{cases} m+1, if\ n = 0; \\ A(n-1,1), if\ n \neq 0, m = 0; \\ A(n-1, A(n, m-1)), if\ n > 0, m > 0. \end{cases}$$



Multiprocessing

The multiprocessing module mimics the threading module

- Can create a process object with a target function, start it and wait for it to join.

```
import multiprocessing

def my_function():
    # printing process id
    print("ID of process running worker2: {}".format(os.getpid()))

p1 = multiprocessing.Process(target=my_function)
p1.start()
p1.join()
```

Multiprocessing

- Can inherit from the Process Class and override the run method

```
import multiprocessing

class CustomProcess(multiprocessing.Process):
    def run(self) -> None:
        # method code

p1 = CustomProcess()
p1.start()
p1.join()
```

Multiprocessing

- Can create a process pool that manages multiple processes.

```
import multiprocessing

param_func_1 = 1
param_func_2 = 2
param_func_3 = 3

with multiprocessing.Pool(processes=3) as pool:
    pool.map(countdown, [param_func_1, param_func_2, param_func_3])
```

If you have 8 cores, only 8 processes can run at the same time.

Processes come with their own overhead. They are heavy and resource intensive .

Multiprocessing - Disadvantages

Processes are expensive.

It is more taxing to run multiple cores than to run a single core with threads.

Exchanging data and resources is resource-intensive. We may not see it but behind the scenes the multiprocessing module is '**`pickling`**' objects and sending it through a '**`Pipe`**' to another process.

Pipe

A Pipe* is a way of communicating across processes.

Think of it as something similar to a producer-consumer queue, but with only one producer and consumer.

A pipe is a pair of ‘connections’ that can send and receive data.

- It can be bidirectional (both connections can send and receive data)
- Or unidirectional(one connection is the producer and another connection is a consumer)

https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Pipe

Pickling

Pickled objects is a way of encoding a python object. This can be sent to another python process and the object will be directly available for use after unpickling. When an object is pickled it is serialized into a binary file.

Similar to JSON.

- Limited to the python language. Will not work with other languages
- Includes top level functions and built-ins

Multiprocessing

We won't really be exploring this aspect of concurrency. Just an introduction.

Let's look at an example

cpu_bound.py

cpu_bound_process.py

cpu_bound_thread.py

Creational Patterns 2: Lazy Initialization & Builder

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 13

Categorizing Design Patterns

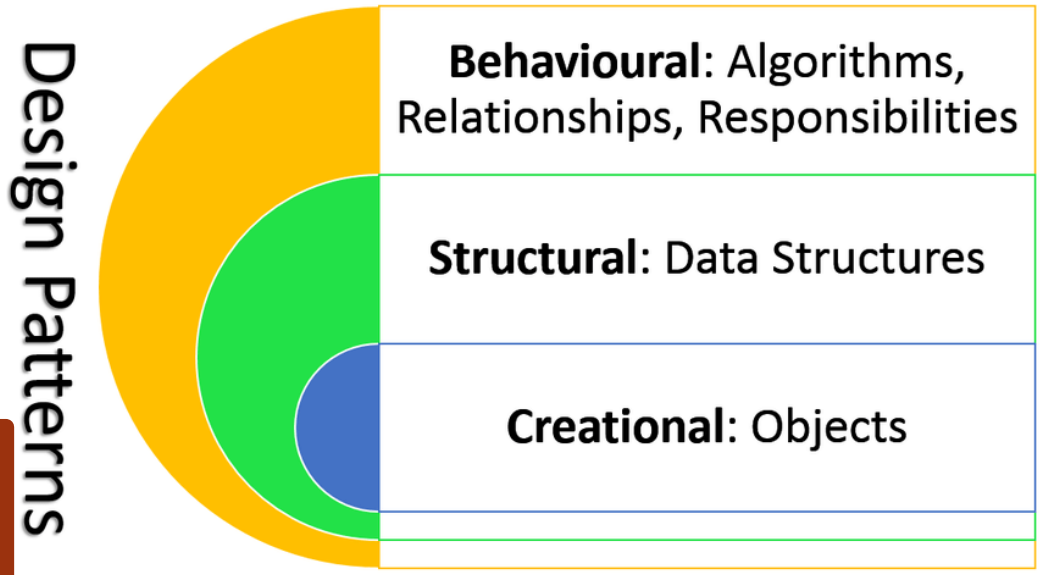
- **Behavioural**

Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational (We are looking at these!)**

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects



Today we will be looking at 2 patterns…

Lazy Initialization

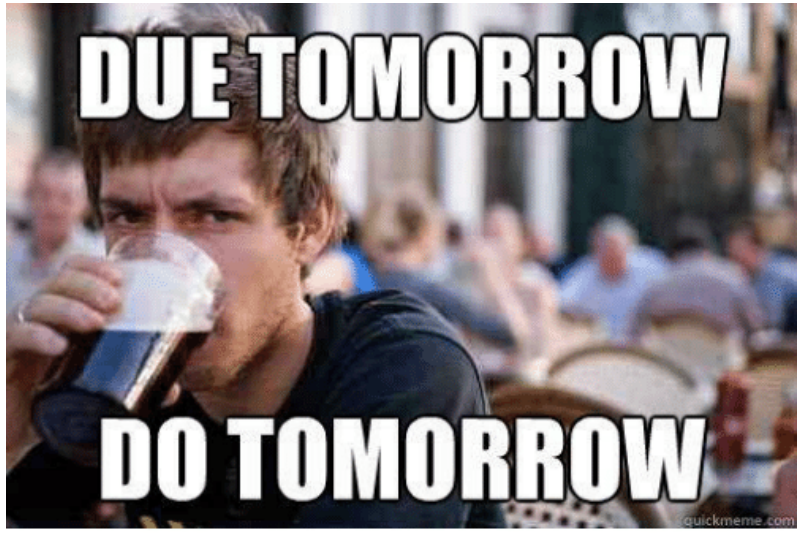
- Delay creation of an expensive or heavy resource until it is actually needed.

Builder

- Decouple creation logic of complex objects with various components from the object itself.

Lazy Initialization

Lazy Initialization



The art of procrastination.

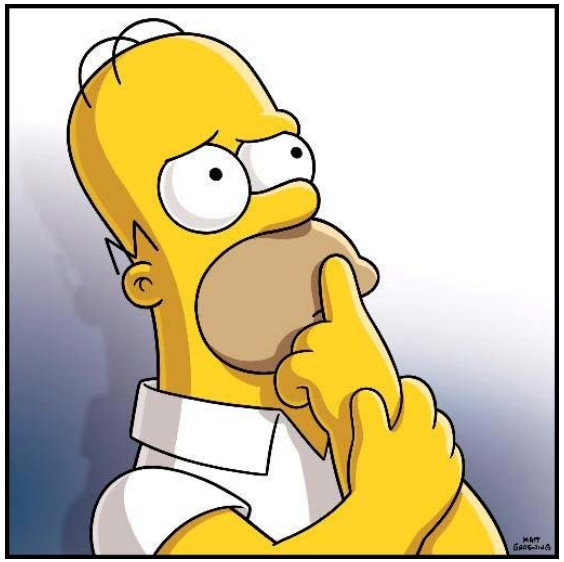
- Lazy Initialization is probably the easiest design pattern out there. It is also known as Lazy Loading.
- I question it's validity as a "pattern". (Purely personal).
- The Lazy Initialization pattern mandates that we don't initialize an object/resource unless we need it.
- Why tax the system unnecessarily?
- This is usually applied to objects that are expensive to initialize either due to memory or computation costs.
- In simple words. We procrastinate on initializing an expensive resource until we need it. Laziness is the hallmark of every good programming.

Lazy Initialization

I could continue talking about it.

But let's look at some code: lazy_initialization_sample_code.py

Cars are great, but where can I actually use this?



Ok I hear you!

Contrived examples are great for understanding but not good for application.

So, hypothetically speaking.

Just hypothetical.

Say we were writing an application that had to maintain a connection or a web session or something like that.

If we are not sending any requests, should we keep the web session open?

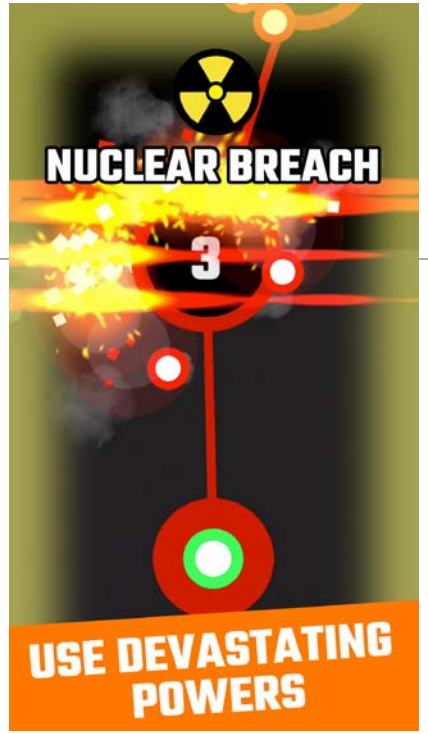
Or should we open one when we need it and say, close it if there has been no activity in 3-5 minutes or so?

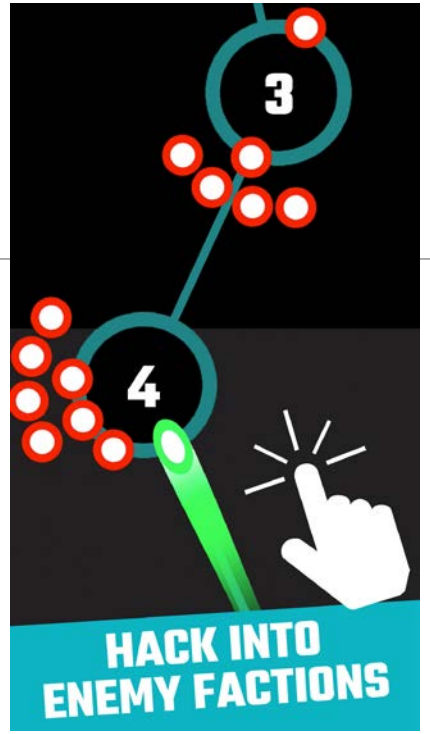
We can lazily initialize our web session and close it after a period of inactivity.

Examples

Lazy Initialization is used when rendering complex environments in games for example.

If the player is not looking at an area, don’t keep it in memory and render it.





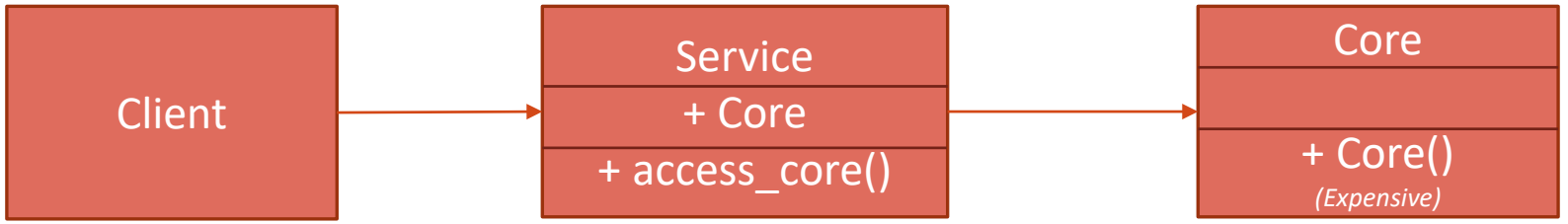
I've used lazy initialization in my games. Games have many screens with UI components. The player won't always see every screen when they play the game

Don't load all screens and save them into memory when the game starts

Load screens when the player needs to access them

Lazy Initialization

- That's it.
- That's lazy initialization.
- Let's look at it's advantages and disadvantages.
- Also, it can be implemented in so many ways, there really isn't much point in a UML diagram.
- But if you insist. Here is a possible UML diagram.



Lazy Initialization

Advantages:

- Dedicate resources (memory / processing) on a on-demand basis.

Disadvantages:

- Not a good choice if the object needs to be initialized every time it is accessed (assuming it is accessed frequently).

Builder

WHEN YOU WANT CUSTOMIZATION AND COMPLEXITY

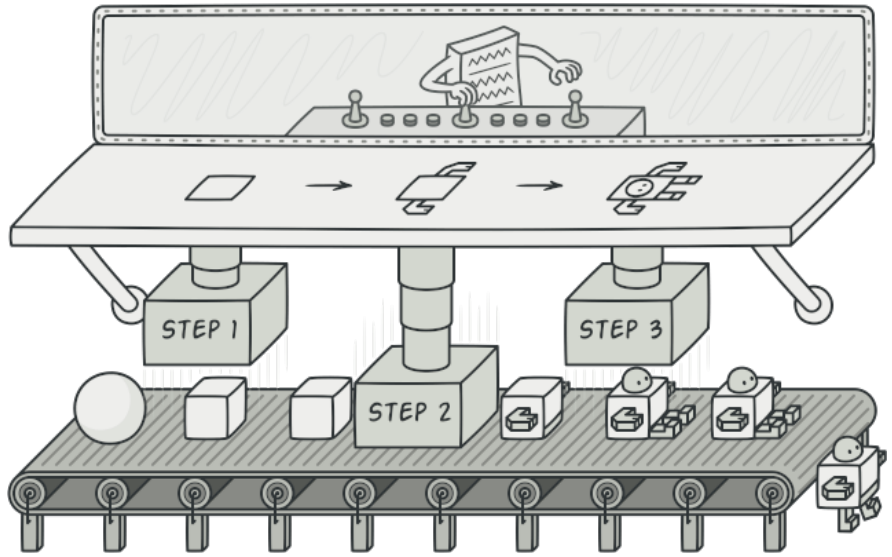
Builder

Often we need to initialize objects with a number of different parts, nested fields, optional attributes and behaviour.

This can cause either:

- The constructor of an object to become bloated. Lots of parameters, most of which may not even be used often.
- A complex inheritance hierarchy to model the optional features.

The Builder Pattern allows us to break this construction code into a separate class that decouples independent blocks of code that can be mixed and matched.



Builder

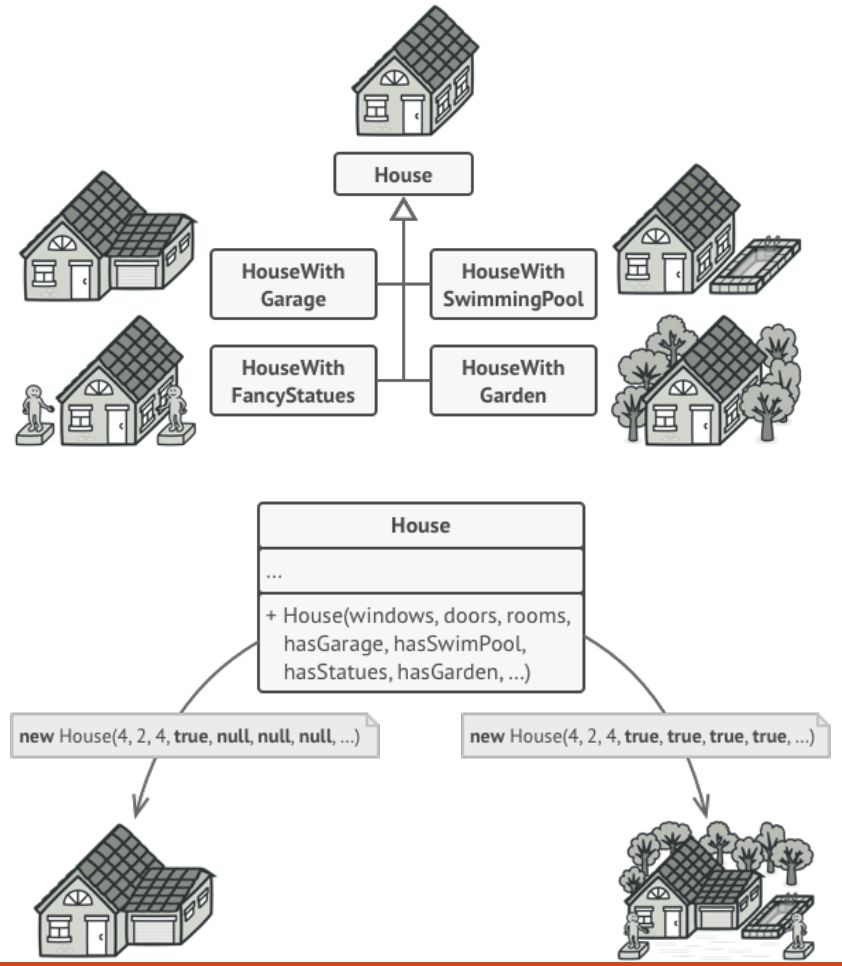
Say we were modeling and representing Houses.

Houses have a lot of components and optional parts.

- Door, walls, roof, windows, etc.

We can use inheritance to model all these cases.

Or we can avoid inheritance, and have a single class with a lot of parameters in it's constructor and unnecessary attributes to keep track of all the components. This leads to ugly constructor calls.



Builder

Let’s make it even better.

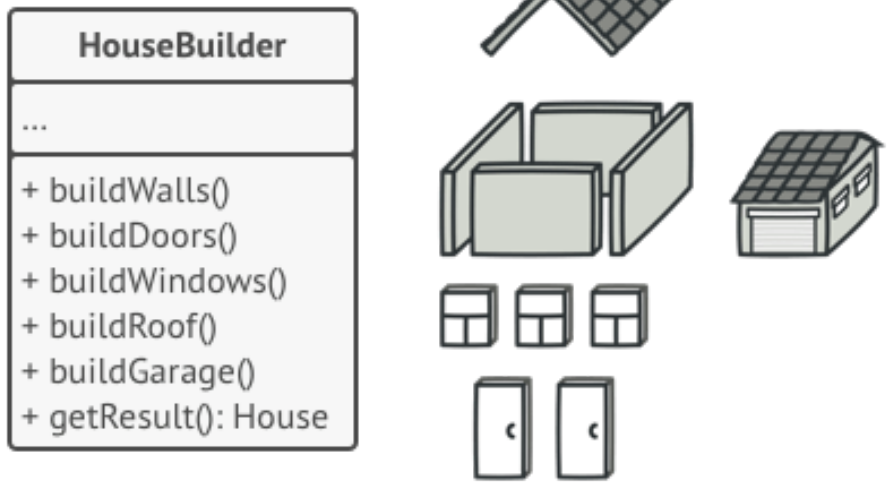
If the construction of an object is so complex, then why don’t we separate that out into its own class.

This would ensure the Single Responsibility Principle.

A class would be split into itself and its builder class with all of its construction code.

We can have a separate call for each different component that our object is aggregated/composed off.

Calling different combinations of these methods would lead to different types of homes.



Builders and Variations

This is a really powerful pattern.

We can customize the Builder Class to take in its own parameters and options.

This can allow us to instantiate different Builder objects that execute the construction differently.

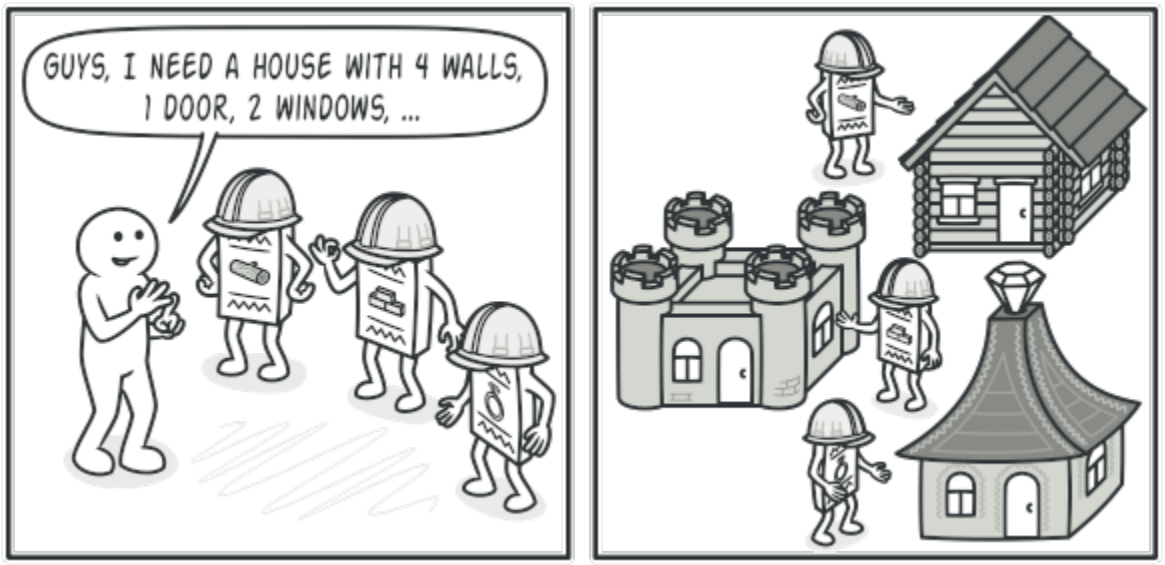
For example, we can have our builder class accept a parameter for the material we can build the house out of.

We can then make 3 builder objects:

```
builder_wood = HouseBuilder(mat="Wood")
cabin = builder_wood.build_house()

builder_stone = HouseBuilder(mat="Stone")
castle = builder_stone.build_house()

builder_paper = housebuilder(mat="Paper")
origami_house = builder_paper.build_house()
```



Builder and Director

We can **optionally** implement a manager class called a Director that can be responsible for different subroutines that call different methods in a builder class.

This Director class is optional and usually makes sense when working with complex objects that can be decoupled into different pieces/components that can be mixed and matched.

We **can even inherit** from a Builder class to create different variations of builders if the complexity demands it.

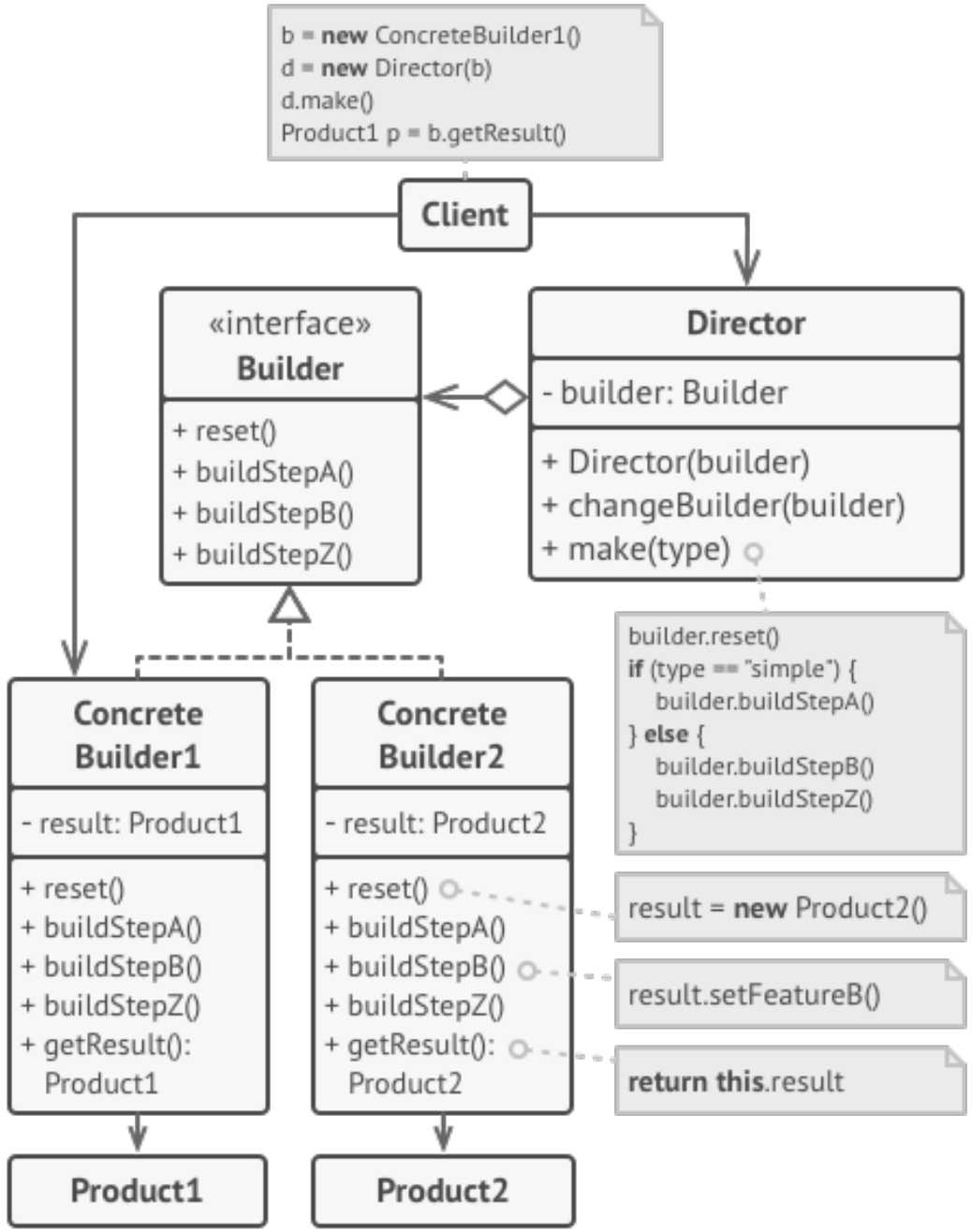
```
class Director:

    def __init__(self, builder: Builder):
        self.builder = builder

    def build_townhouse(self) -> House:
        self.builder.build_walls()
        self.builder.build_roof()
        self.builder.build_garage()
        self.builder.build_swimming_pool()
        self.builder.build_second_story()
        self.builder.build_door()
        self.builder.build_windows()
        return self.builder.product

    def build_apartment(self) -> House:
        self.builder.build_walls()
        self.builder.build_door()
        self.builder.build_windows()
        return self.builder.product
```

Builder Pattern: UML

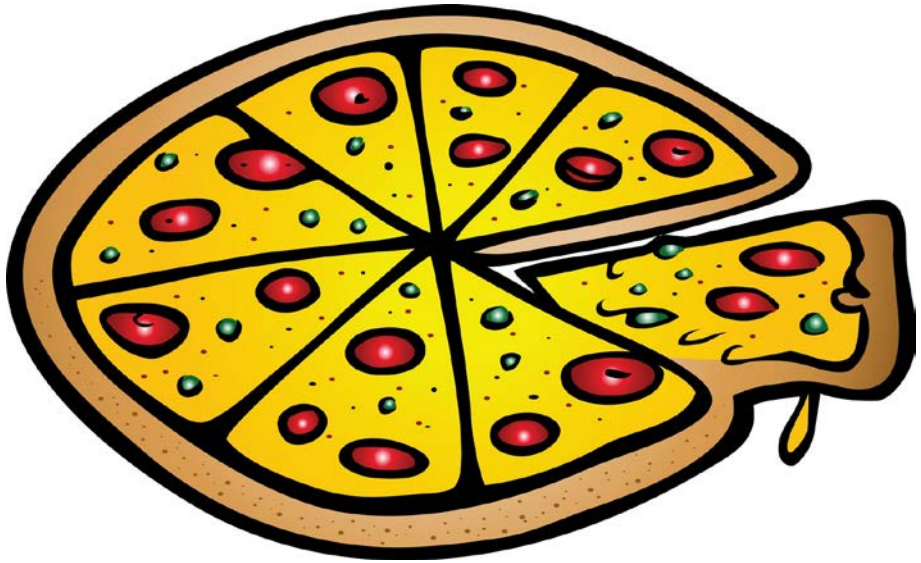


Builder Pattern

Let's look at some code:

bad_pizza.py

builder_sample_code.py



Builder Pattern: Why and When do we use it

- When you want to get rid of ugly init methods in Python or the numerous constructors you may need in other languages (*cough* Java *cough*)
- When a product is built out of many parts that can be mixed and matched. This allows you to build variations while (possibly) avoiding inheritance hierarchies
- Single Responsibility Principle comes to life as we separate construction logic from the product logic. That is, we decouple complex construction logic from the product.

Builder Pattern – Disadvantages

- Code complexity increases as we introduce more classes.
- Does not work well for products that cannot be broken down into independent parts.

That’s it for today

No more labs!

But 1 more assignment, due next week

Pair up and sign up on D2L so you know who’s looking for partners