Design pattern

Creational	Behavioral	Structural

Creational

Singleton	Factory	Builder	Prototype

Logger - to log info over the system

5 log levels:
- Debug
- Info
- Error
- Warning
- Critical

```
def critical(self, level, msg):
    self._write_log("CRITICAL",msg)

def error(self, level, msg):
    self._write_log("ERROR",msg)

def warning(self, level, msg):
    self._write_log("WARNING",msg)

def debug(self, level, msg):
    self._write_log("DEBUG",msg)

def info(self, level, msg):
    self._write_log("INFO",msg)
```

Notes:
- Save to files: access_log, error_log, info_log
- Able to save to db
- If having error, …. Logger.writelog()
- Connector - message queue
- Audit system

```
    class __SingletonLogger():
        def __init__(self):
            self.file_name = None

        def __str__(self):
            return "{0!r} {1}".format(self,self.file_name)

        def _write_log(self, level, msg):
            with open (self.file_name, "a") as log_file:
                log_file.write("[{0}] {1}\n".format(level, msg))
```

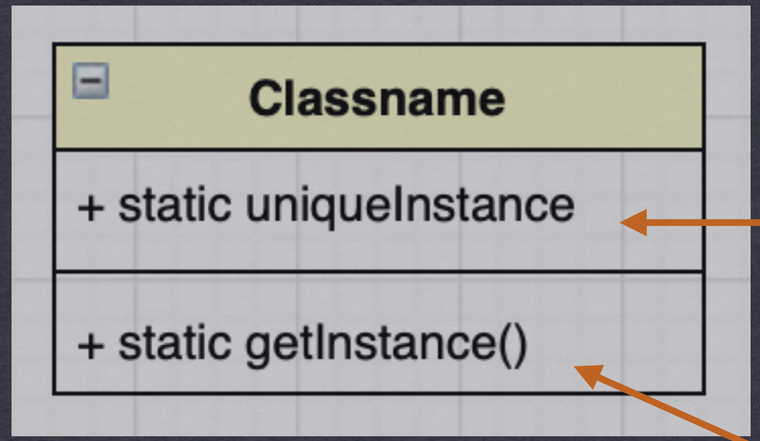
Logger.writelog()

```
    instance = None

    def __new__(cls):
        if not SingletonLogger.instance:
            SingletonLogger.instance =
SingletonLogger.__SingletonLogger()
        return SingletonLogger.instance

    def __getattr__(self, name):
        return getattr(self.instance, name)

    def __setattr__(self, name):
        return setattr(self.instance, name)
```

The uniqueInstance

Check instance

Caching, load balancing, route mapping, Find dialog, ...

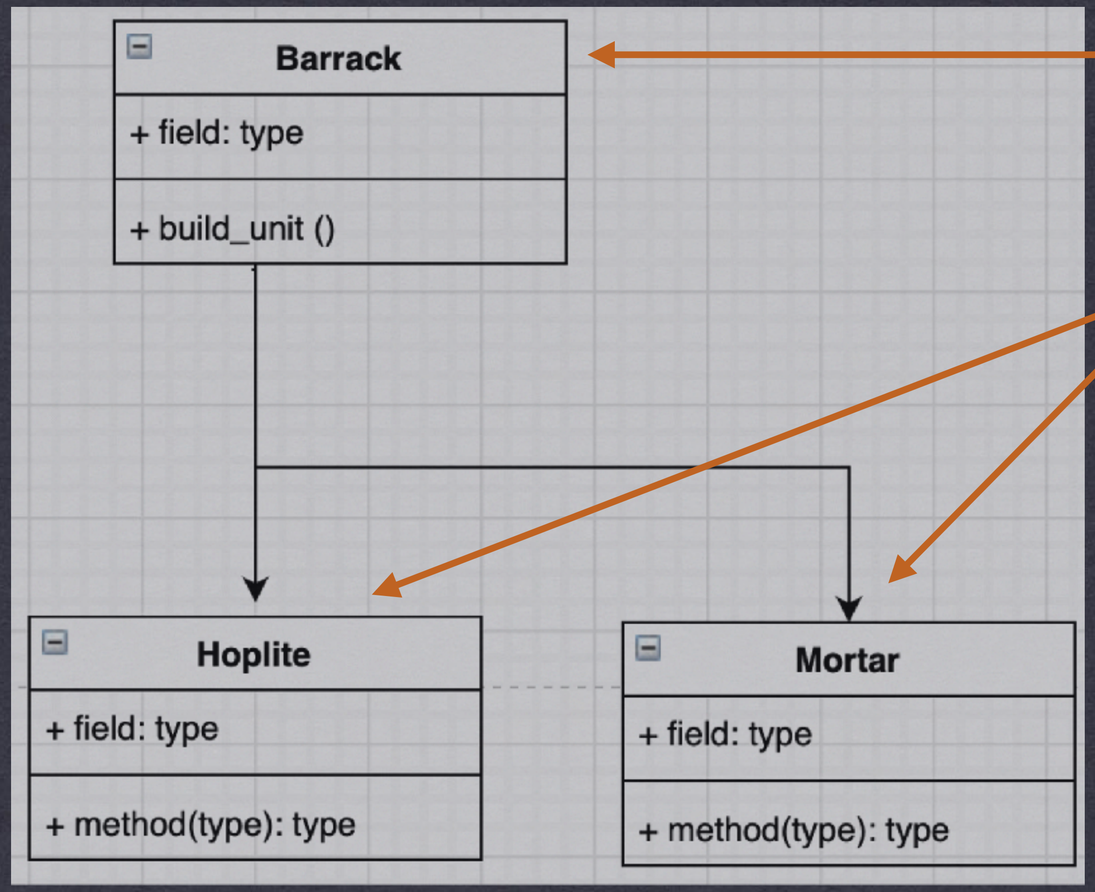


The uniqueInstance class variable holds our one and only instance of Singleton

The getInstance() method is static

Characters in game - to create many characters at the same time and with various levels

Prototype



Barrack to generate units

units

Levels

```
class Hoplite(object):
    def __init__(self, level):
        self.unit_type = "Hoplite"
        if level == 1:
            self.hit = 56
            self.armour = 1
            self.speed = 60
            self.size = 1
            self.weapon = "Spear"
        elif level == 2:
            self.hit = 60
            self.armour = 3
            self.speed = 60
            self.size = 1
            self.weapon = "Spear"
```

Can use file for level configuration

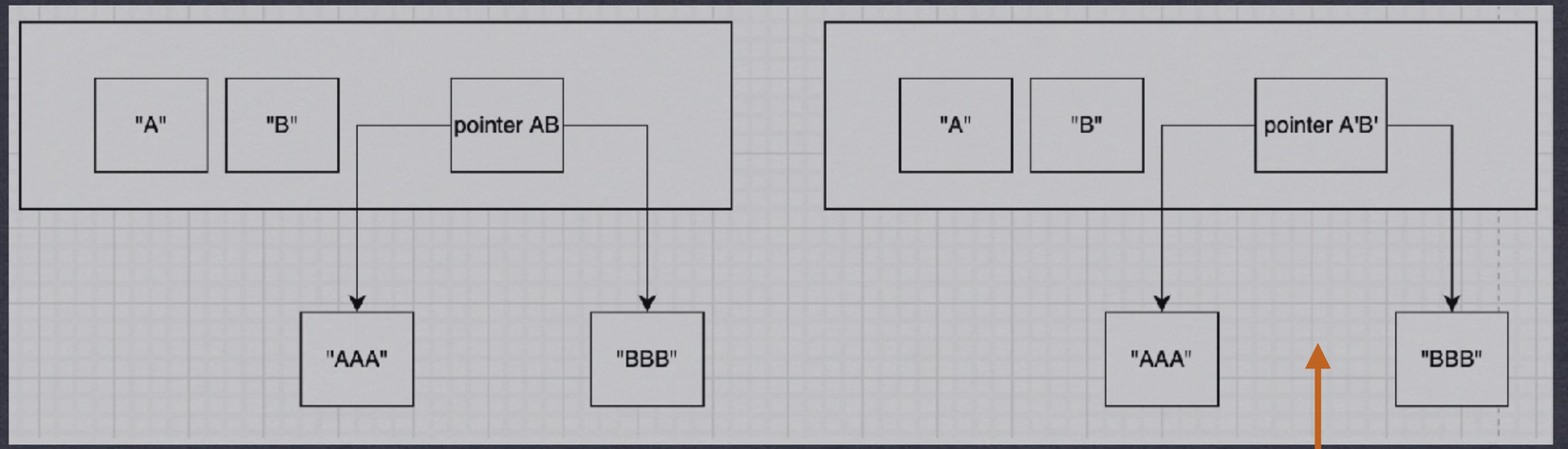
```
class Hoplite(object):
    def __init__(self, level):
        self.unit_type = "Hoplite"
        filename =
"{}{}_{}.dat".format("./properties/",self.unit_type, level)
        with open(filename, 'r') as param_file:
            lines = param_file.read().split("\n")
            self.hit = lines[0]
            self.armour = lines[1]
            self.speed = lines[2]
            self.size = lines[3]
            self.weapon = lines[4]
```

```
class Barracks(object):
    def build_unit(self, unit_type, level):
        if unit_type == "Hoplite":
            return Hoplite(level)
        elif unit_type == "Mortar":
            return Mortar(level)
```

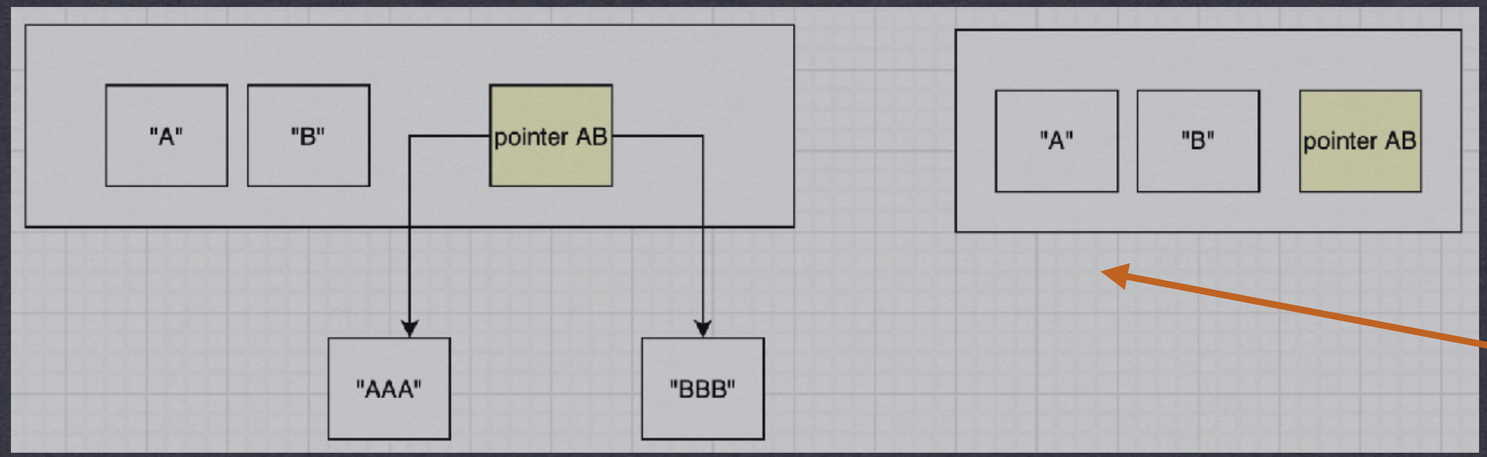
Type

Levels

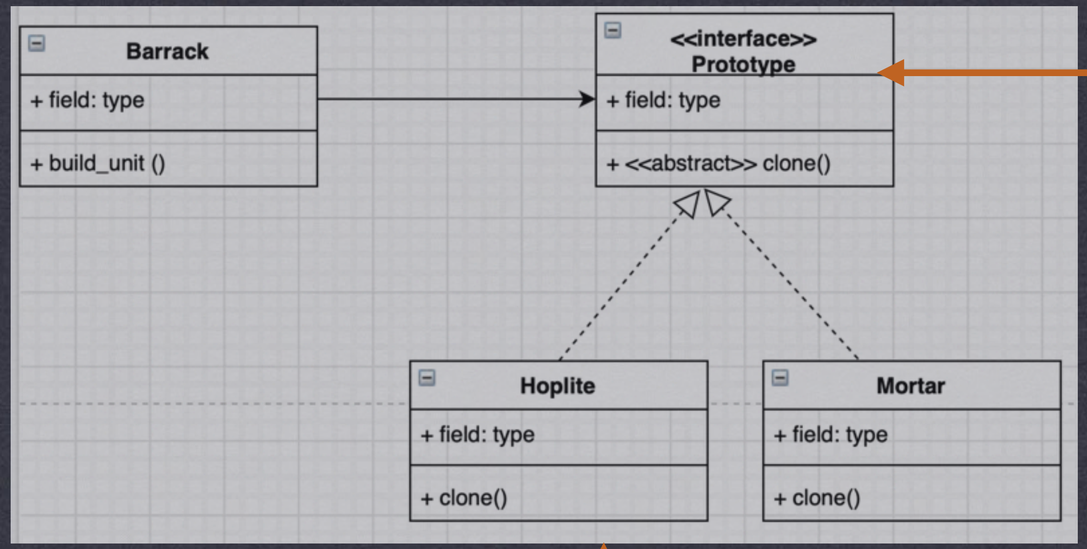
Deep & Shallow copy



Deep copy



Shallow copy




```python
class Character(metaclass=ABCMeta):
    @abstractmethod
    def clone(self):
        pass
```

```python
class Hoplite(Character):
# Nothing to change
# add method clone()

    def clone(self):
        return deepcopy(self)
```

Prototype using deep copy

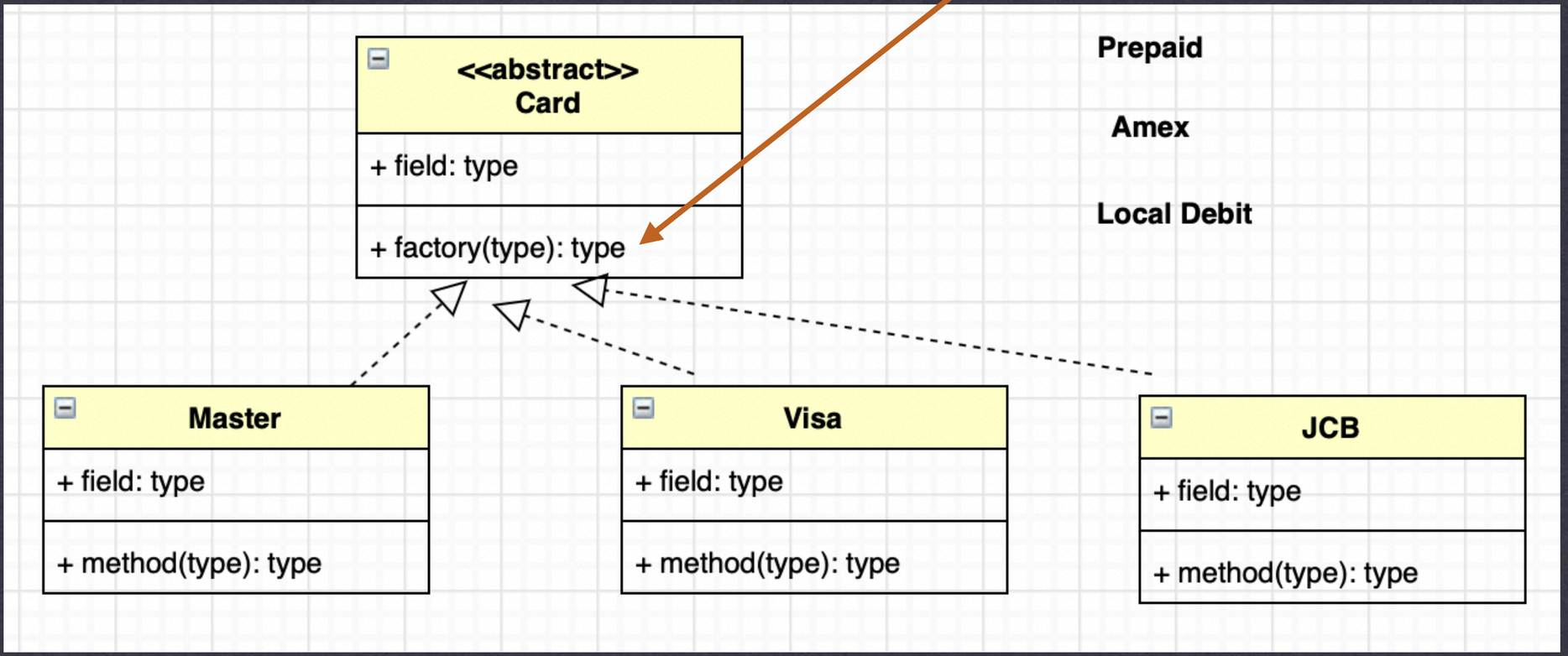
Cards generation

Factory

```
@staticmethod
def card_factory(type):
    if type == "MasterCredit":
        return MasterCreditCard()
    if type =="VisaDebit":
        return VisaDebitCard()
```

Factory method

Method to generate cards



Abstract factory