MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ

# Laboratory work 1:
# Creational Design Patterns

Elaborated: Cucos, Maria
st. gr. FAF-221

Chișinău, 2024

**Task:** Define the main involved classes and think about what instantiation mechanisms are needed. Implement at least 3 creational design patterns in your project.

I implemented the Singleton, Builder and Factory patterns in a python project that manages a shelter catalog.

**Implementation:**

**Singleton Pattern** (`ShelterManagement`):

- Ensures only one instance of the shelter management system exists
- Maintains a central registry of all animals (cats and dogs)
- Single point of control for the entire system

```
class ShelterManagement:

  _instance = None

  def __new__(cls):

    if cls._instance is None:

      cls._instance = super().__new__(cls)

      cls._instance.animals = []

    return cls._instance

  def add_animal(self, animal: Animal):

    self.animals.append(animal)

  def list_animals(self) -> List[Animal]:

    return self.animals
```

**Builder Pattern** (`AnimalProfileBuilder`):

- Handles the construction of complex `AnimalProfile` objects
- Allows step-by-step construction of animal profiles
- Makes it easy to create different representations of animals with varying attributes
- Provides a fluent interface

```
class AnimalProfile:

  def __init__(self):

    self.name = None

    self.species = None
```

```
    self.age = None
    self.medical_history = []
    self.behavior_notes = []
    self.special_needs = []
    self.vaccination_status = {}


class AnimalProfileBuilder:
  def __init__(self):
    self.profile = AnimalProfile()
  def set_basic_info(self, name: str, species: str, age: int):
    self.profile.name = name
    self.profile.species = species
    self.profile.age = age
    return self
  def add_medical_history(self, condition: str):
    self.profile.medical_history.append(condition)
    return self
  def add_behavior_notes(self, note: str):
    self.profile.behavior_notes.append(note)
    return self
  def add_special_needs(self, need: str):
    self.profile.special_needs.append(need)
    return self
  def add_vaccination_status(self, vaccine: str, date_administered:
date):
    self.profile.vaccination_status[vaccine] = date_administered
    return self
  def build(self):
    return self.profile
```

**Factory Method Pattern** (`AnimalFactory`):

- Provides an interface for creating animals
- Allows subclasses (`CatFactory`, `DogFactory`) to decide which class to instantiate
- Makes the system extensible for new animal types
- Encapsulates the object creation logic

```python
class AnimalFactory(ABC):
  @abstractmethod
  def create_animal(self, profile: AnimalProfile) -> Animal:
    pass

class DogFactory(AnimalFactory):
  def create_animal(self, profile: AnimalProfile) -> Animal:
    return Dog(profile)

class CatFactory(AnimalFactory):
  def create_animal(self, profile: AnimalProfile) -> Animal:
    return Cat(profile)
```

The `animal.py` file defines the core animal classes using abstract base classes and inheritance.

```python
class Animal(ABC):
  def __init__(self, profile: AnimalProfile):
    self.profile = profile

  @abstractmethod
  def make_sound(self):
    pass

  @abstractmethod
  def get_care_instructions(self):
    pass

class Dog(Animal):
  def make_sound(self):
    return "Woof"

  def get_care_instructions(self):
    return "Feed twice a day, walk daily, fresh water"

class Cat(Animal):
  def make_sound(self):
    return "Meow"

  def get_care_instructions(self):
    return "Feed three times a day, fresh water, clean litter box"
```

I created empty `__init__.py` files in each directory to make them Python packages. The `main.py` file serves as the entry point and demonstration of how the implemented design patterns work together in the `Animal Shelter` system. For the Singleton Pattern it ensures only one shelter management system exists throughout the application and all operations on animals go through this single instance. For the Builder Pattern it creates detailed profiles with basic information (name, species, age), medical history, behavior notes and vaccination records. And for the Factory Pattern it uses specific factories for each animal type and shows how factories handle the object creation process.

```
shelter = ShelterManagement()
cat_profile_builder = AnimalProfileBuilder()
cat_profile = (
  cat_profile_builder
  .set_basic_info("Mia", "Cat", 6)
  .add_medical_history("Vaccinated")
  .add_behavior_notes("Friendly")
  .add_vaccination_status("Rabies", date(2021, 2, 1))
  .build()
)
cat_factory = CatFactory()
new_cat = cat_factory.create_animal(cat_profile)
shelter.add_animal(new_cat)
```

Output:

```
PS D:\TMPS-labs\Animal Shelter> python main.py
Same shelter instance? True

Animal: Mia
Species: Cat
Sound: Meow
Care instructions: Feed three times a day, fresh water, clean litter box

Animal: Teddy
Species: Dog
Sound: Woof
Care instructions: Feed twice a day, walk daily, fresh water
```

**Conclusions:**

This project uses 3 design patterns: Singleton, Builder and Factory. The main benefits of using these patterns in this project are centralized management (Singleton ensures consistent access to the shelter system), flexible construction (Builder allows creating detailed animal profiles without constructor complexity) and extensible creation (Factory Method makes it easy to add new animal types). This project shows how all three patterns work together: the Builder creates the profile, the Factory creates the animal with the profile and the Singleton manages all created animals.