Designing Programs with Inheritance

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: What would be a good relationship between `Chair` and `Throne`?

1. `Throne` should be an interface implemented by `Chair`
2. `Chair` should be an interface implemented by `Throne`
3. `Throne` should be a concrete subclass of the abstract class `Chair`
4. `Chair` should be a concrete subclass of the abstract class `Throne`
5. None of the above

Inheritance vs composition

Inheritance: *is a* relationship

- A square *is a* rectangle.
- One of the classes is a subclass of the other.
- (One of the classes may be abstract, but neither should be an interface.)

Composition: *has a* relationship

- A square *has* four edges.
- One of the classes holds an instance of the other class as an instance variable.

Good object-oriented design requires knowing when to use inheritance versus composition.

Example when to use composition (*has a*)

`SocialMedia`: a class that holds information about a social media platform, including a set of users

- Correct: composition. `SocialMedia` class has a `set[User]` as an attribute
- Incorrect: inheritance. It would be wrong to make the `SocialMedia` class extend the `set[User]` class

  Both are possible to do using Python, but the inheritance version is silly:

```python
class SocialMedia(set[str]):
    pass

fb = SocialMedia()
fb.add('Mini')
fb.add('Binnie')

print(fb)  # SocialMedia({'Mini', 'Binnie'})
```

Example when to use composition (*has a*)

A `House` with a kitchen and bedroom

- Correct: composition. The `House` class should have instance variables for a `Kitchen` and a `Bedroom`
- Incorrect: inheritance. It would be wrong to make the `House` class extend the `Kitchen` class and add the features of a `Bedroom` (like a `Bed` instance variable)

Again, both options are possible to do using Python, but the inheritance version would require admitting that one's house is a specific type of kitchen.

Example when to use inheritance (*is a*)

`Cat`, `Lion`, and `HouseCat`:

- Correct: inheritance. The `Lion` and `HouseCat` classes should extend the `Cat` class
- Incorrect: composition. It would be wrong to make the `Lion` and `HouseCat` classes each have an instance variable for a `Cat` to which they outsource all of the kneading

Poll: Which of these pairs of classes should use inheritance rather than composition?

1. `VideoGame` and `Physics`
2. `UIComponent` and `TextBox`
3. `Student` and `TA`
4. `OnlineStore` and `Inventory`
5. `TextEditor` and `SpellChecker`

```python
from abc import ABC, abstractmethod

class Pet(ABC):
    @abstractmethod
    def express_affection(self) -> None:
        pass

class Cat(Pet):
    def express_affection(self) -> None:
        self.make_biscuits()

    def make_biscuits(self) -> None:
        print('Making biscuits')

class Dog(Pet):
    def express_affection(self) -> None:
        self.slobber()

    def slobber(self) -> None:
        print('Slobbering')

for pet in [Cat(), Dog(), Cat()]:
    pet.express_affection()
```

Polymorphism

This works because of *polymorphism*: the `pet` variable's ability to be both a `Cat` and a `Dog`, and for it to be treated correctly as an instance of both a `Cat` and a `Dog`.

Polymorphism

Let's create classes for `Car`, `Motorcycle`, and `Truck`.

Let's write a function that takes a fleet of vehicles as a list and returns the total fuel needed for the trip.

```python
class Vehicle(ABC):
    def __init__(self, mpg: int):
        self.fuel_used: float = 0.0
        self.mpg = mpg

    def move(self, distance: int) -> None:
        self.fuel_used += (distance / self.mpg)

    def get_fuel(self) -> float:
        return self.fuel_used

class Car(Vehicle):
    def __init__(self) -> None:
        super().__init__(26)

class Motorcycle(Vehicle):
    def __init__(self) -> None:
        super().__init__(55)

class Truck(Vehicle):
    def __init__(self) -> None:
        super().__init__(7)
```

```python
def get_total_gas(fleet: list[Vehicle]) -> float:
    return sum(veh.get_fuel() for veh in fleet)

fleet: list[Vehicle] = [
    Car(),
    Car(),
    Truck(),
    Motorcycle(),
    Motorcycle(),
    Motorcycle(),
    Motorcycle()
]

for veh in fleet: veh.move(10)

print(get_total_gas(fleet))
```

Design principle: Encapsulation

- Group attributes and methods into a single class.
- Information hiding: discourage direct access to some methods / attributes (using underscores).
    - Protects internal data from unauthorized modification
    - Promotes "modularity" by hiding unnecessary implementation details behind a simple public interface
    - Gives us more flexibility to change implementations without telling the client

Poll: Which class design best demonstrates encapsulation?

1. Expose all internal attributes to the public for flexibility
2. Create a minimal public interface with all complex logic kept private
3. Rather than having a class be a direct subclass of its interface, make it a subclass of a subclass, to add layers of privacy
4. Document internal methods thoroughly for users

Poll: Here is a poorly designed Python class:

```
class Rectangle:
    def __init__(self, width: int, height: int):
        self.width = width
        self.height = height
        self.area = width * height
```

How can we improve its encapsulation?

1. Validate in `__init__()` that `width` and `height` are not negative
2. Make all three attributes private with corresponding getter and setter methods using the `@property` decorator
3. Make `width` and `height` private with corresponding getter/setter `@property` methods, and make `area` a property only (calculated in a getter method)
4. Add docstrings to explain the attributes

Poll: Here is a poorly designed Python class:

```python
class Rectangle:
    def __init__(self, width: int, height: int):
        self.width = width
        self.height = height
        self.area = width * height
```

How should errors be handled in an encapsulated class design?

1. Raise all errors to the caller
2. Wrap all errors in `try` / `except` and return error codes instead
3. Wrap low-level internal errors in `try` / `except` and raise them as domain-specific errors
4. Log errors internally but never raise them

Design principle: Coupling vs cohesion

Cohesion: how closely related the parts of a unit are

- closely related to the Single Responsibility Principle
- Example where the unit is a function: aim for it to have a single, well-defined job
- Example where the unit is a class: aim for its methods to be very closely related

Coupling: how dependent different units are on each other

- want to **avoid** this
- Often, that means that one class is too dependent on another, and any changes to the other class will result in "ripple effects" on it.

Bad email sender with too much coupling of its tasks:

```
class BadEmailSender:
    def send_email(self,
            user: str, email_type_flag: int
    ) -> None:
        if email_type_flag == 1:
            # send a welcome email
        elif email_type_flag == 2:
            # send a password reset email
```

- Many tasks, all very dependent on each other
- Code will be repeated between branches

Better design that uses polymorphism to separate out the tasks into cohesive classes:

```
class Template(ABC):
    @abstractmethod
    def generate_content(self, user: str) -> str:
        pass

class EmailSender:
    def send_email(self,
            email_template: Template, user: str
    ) -> str:
        return email_template.generate_content(user)

class WelcomeEmail(Template):
    def generate_content(self, user: str) -> str:
        return f"Welcome {user}!"

class PasswordResetEmail(Template):
    def generate_content(self, user: str) -> str:
        return f"Reset password for {user}"
```

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?