Replace Type Code with Strategy

Replace a **mutable type code** with a **Strategy** to change behavior at runtime.

- We cannot use the previous refactoring (Replace Type Code with Subclasses), when the object code is updated **dynamically**.
- We introduce type code with strategy refactoring in this case.

Example: Logger

Before:

- We have a Logger class that uses type code to express current state.
- It has two states: stop (STATE_STOPPED) and log (STATE_LOGGING).

```
class Logger:
    STATE_STOPPED = 0
    STATE_LOGGING = 1
    ...
```

- The Logger class stores its state in the state field.
- The initial state is stop.

```
class Logger:
    ...
    def __init__(self):
        self.state = Logger.STATE_STOPPED
```

- Depending on the current state, we should choose different behavior.
- The start method changes its state to log only when it is in the stop state.

```
# in the stop state, the state is changed to log
def start(self):
    if self.state == Logger.STATE_STOPPED:
        self.state = Logger.STATE_LOGGING
    elif self.state == Logger.STATE_LOGGING: pass
```

- Likewise, the stop method changes its state only in log state.

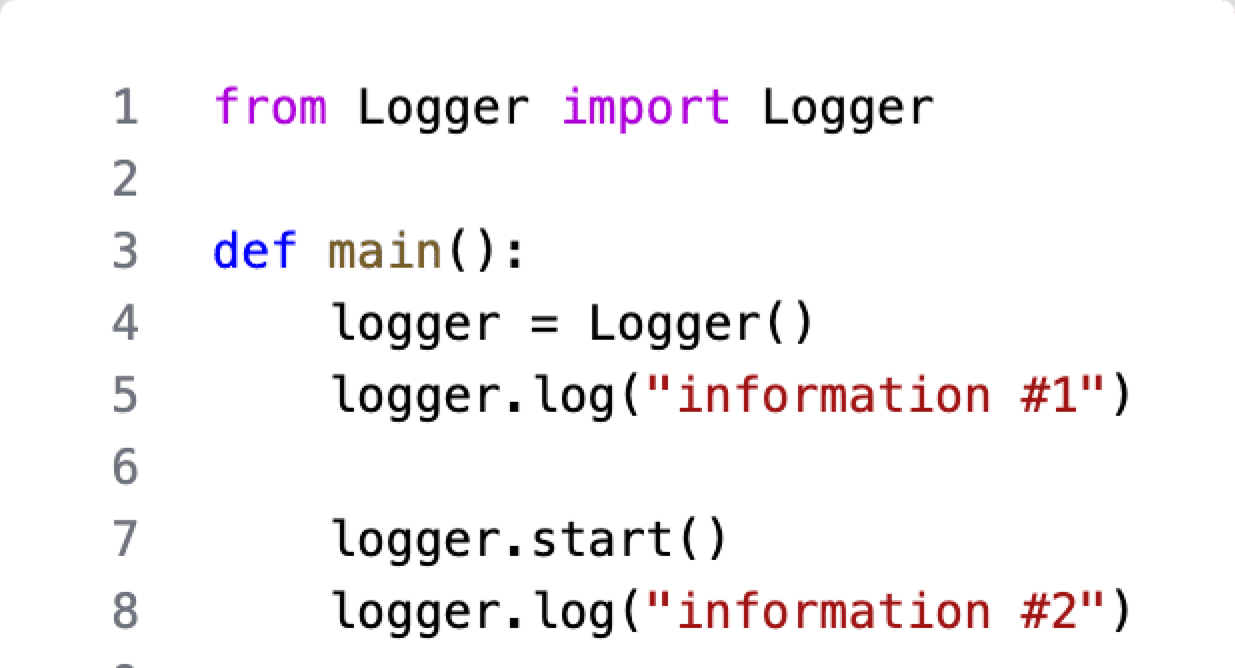
```python
def stop(self):
    if self.state == Logger.STATE_STOPPED: pass
    elif self.state == Logger.STATE_LOGGING:
        self.state = Logger.STATE_STOPPED
    else:
        print(f"Invalid state: {self.state}")
```

- The log method displays different message depending on its current state.

```python
def log(self, info: str):
    if self.state == Logger.STATE_STOPPED:
        print(f"Ignoring: {info}")
    elif self.state == Logger.STATE_LOGGING:
        print(f"Logging: {info}")
    else:
        print(f"Invalid state: {self.state}")
```

- We can start or stop logging using states.

```
Ignoring: information #1
** START LOGGING **
Logging: information #2
Logging: information #3
** STOP LOGGING **
Ignoring: information #4
Ignoring: information #5
```



```python
from Logger import Logger

def main():
    logger = Logger()
    logger.log("information #1")

    logger.start()
    logger.log("information #2")
```

After:

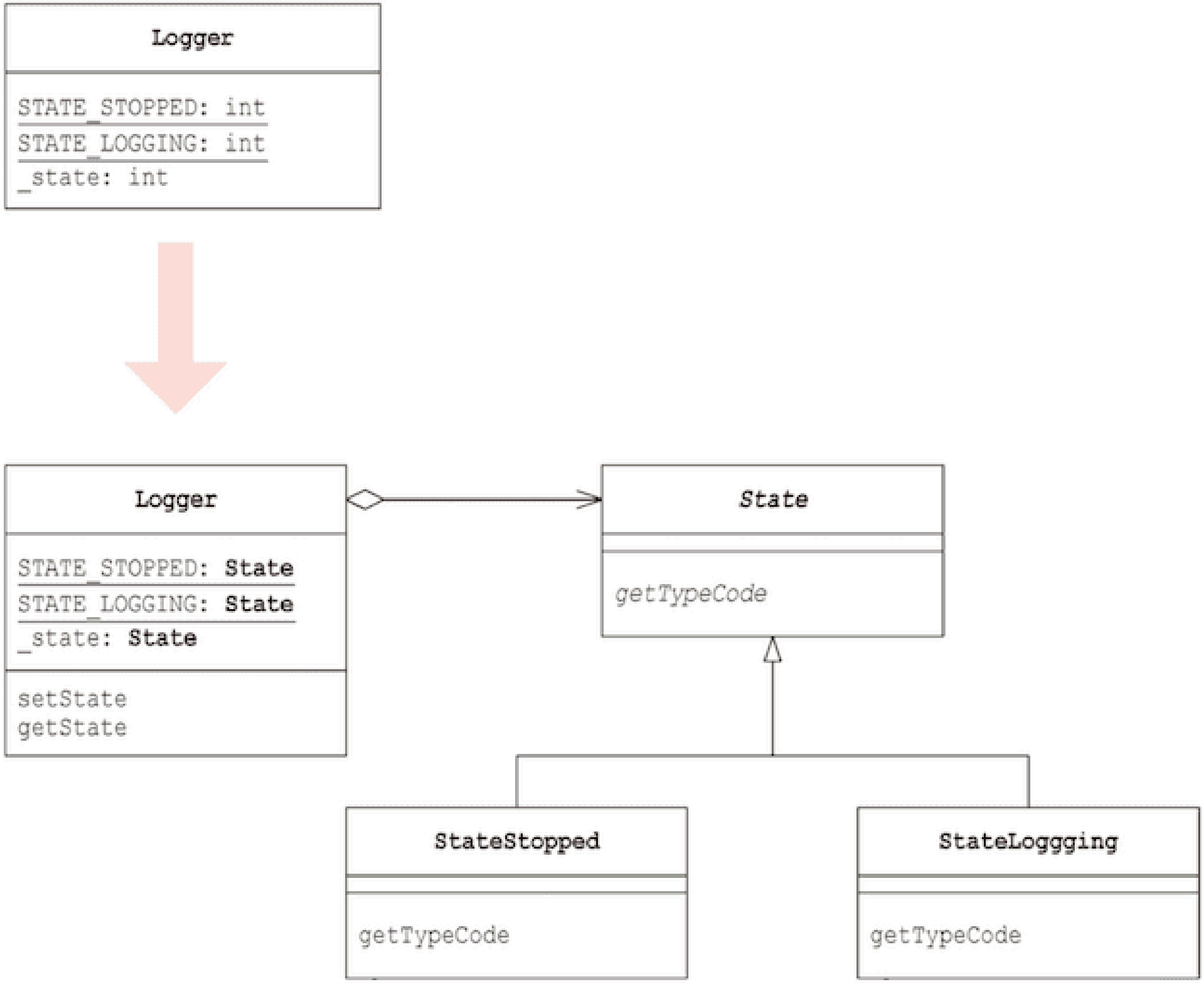
- We simplify the code using two steps.
- The first step is to introduce the state class to take responsibility of state management.

- As a first step, we isolate states (abstract and its subclasses).

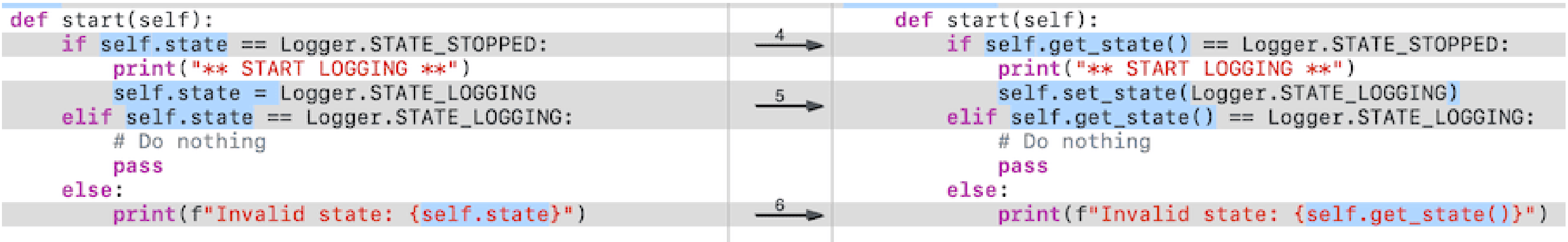
```python
class State(ABC):
    @abstractmethod
    def get_type_code(self) -> int:pass

class StateLogging(State):
    def get_type_code(self) -> int:

class StateStopped(State):
    def get_type_code(self) -> int:
...
```

- Logger uses `set_state` method to switch states.

```python
class Logger:
    ...
    def set_state(self, state: int):
        if state == Logger.STATE_STOPPED:
            self._state = StateStopped()
        elif state == Logger.STATE_LOGGING:
            self._state = StateLogging()
        else:
            print(f"Invalid state: {state}")
    def get_state(self) -> int:
        return self._state.get_type_code()
```


Logger
STATE_STOPPED: int
STATE_LOGGING: int
_state: int
Logger
STATE_STOPPED: State
STATE_LOGGING: State
_state: State
setState
getState
State
getTypeCode
StateStopped
getTypeCode
StateLoggging
getTypeCode

- The state assignment is now using the `set_state` and `get_state` method.



```
def start(self):
    if self.state == Logger.STATE_STOPPED:
        print("** START LOGGING **")
        self.state = Logger.STATE_LOGGING
    elif self.state == Logger.STATE_LOGGING:
        # Do nothing
        pass
    else:
        print(f"Invalid state: {self.state}")
```

```
def start(self):
    if self.get_state() == Logger.STATE_STOPPED:
        print("** START LOGGING **")
        self.set_state(Logger.STATE_LOGGING)
    elif self.get_state() == Logger.STATE_LOGGING:
        # Do nothing
        pass
    else:
        print(f"Invalid state: {self.get_state()}")
```

Unit tests are the same except that they use `get_state` to get current state.

The `get_state` and `set_state` are not visible (encapsulation).

```python
class TestLogger(unittest.TestCase):
    def setUp(self):
    def test_complete_scenario_from_main(self):
        # Log when stopped - state should remain STOPPED
        self.logger.log("information #1")
        self.assertEqual(self.logger.get_state(), Logger.STATE_STOPPED)

        # Start logging - state should change to LOGGING
        self.logger.start()
        self.assertEqual(self.logger.get_state(), Logger.STATE_LOGGING)
```

Refactoring again to remove the Type Code (and if statement)

- The next step is to remove the type code.
- The type code was needed because the Logger class manages the state.
- When the state is managed by the State, we don't need the type code and if statements.
- Also, we can remove the if statements.

- The State manages its state with start, stop, and log methods.

```python
class State(ABC):
    def start(self) -> None: pass
    def stop(self) -> None: pass
    def log(self) -> None: pass
```

- In the stopped state, it starts logging with the start method.

```python
class StateStopped(State):
    def start(self):
        print("** START LOGGING **")

    def stop(self): pass

    def log(self, info: str):
        print(f"Ignoring: {info}")
```

- The log state behaves accordingly.

```python
class StateLogging(State):
    def start(self):
        pass

    def stop(self):
        print("** STOP LOGGING **")

    def log(self, info: str):
        print(f"Logging: {info}")
```

- The Logger class delegates state management to its state.

```python
class Logger:
    def __init__(self): self._state = StateStopped()
    def get_state(self): return self._state
    def set_state(self, state): self._state = state
    def start(self):
        self._state.start()
        self.set_state(StateLogging())
    def stop(self):
        self._state.stop()
        self.set_state(StateStopped())
    def log(self, info: str):
        self._state.log(info)
```

Unittests Update

We need to check the object type instead of int type to represent the object.

```
self.assertEqual(self.logger.get_state(), Logger.STATE_STOPPED)

=>

self.assertIsInstance(self.logger.get_state(), StateStopped)
```

Discussion

Benefits of Replace Type Code with Strategy

1. **Runtime type changes** - can change behavior during object lifetime
2. **Eliminates conditionals** - no if/switch statements on type
3. **Composition over inheritance** - more flexible than subclassing
4. **Multiple variations** - can vary along multiple dimensions
5. **Strategy reuse** - strategies can be shared among objects

The main difference between Replace Type Code with Subclass and Replace Type Code with Strategy

Subclass: Uses **inheritance**, type is **fixed at creation**, object cannot change type.
Strategy: Uses **composition**, type can **change at runtime**, more flexible but slightly more complex.