Design patterns: Strategy, Observer, and "Data Pull"

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Design Patterns

Already covered a few *design patterns*: structures or templates that software engineers have agreed solve common software problems

- Lists: Accumulator Pattern (add up, or *accumulate*, a sequence of items)
- Design patterns for handling data: mapping, filtering, and merging dataframes

Design patterns:

- are independent of programming language
- are flexible (the overarching pattern allows modifications)
- provide us a common vocabulary to communicate "blueprints" for standard patterns

"Data Pull" pattern

Modeled after this lecture: https://neu-se.github.io/CS4530-Spring-2024/Slides/Module 05 Interaction-Level Design Patterns.pdf

Information transfer: push versus pull

Producer produces some data, Consumer uses that data:

```
class Producer:
    def get_data(self) -> int:
        return 500

class Consumer:
    def do_some_work(self) -> None:
        self.do_something(self.needed_data)

    def do_something(self, data: int) -> None:
        # Placeholder for actual work
        pass
```

How can we get the data from the producer to the consumer?

"Data pull" pattern: consumer asks producer

```python
class Producer:
    def get_data(self) -> int:
        return 500

class Consumer:
    def __init__(self, producer: Producer):
        self.producer = producer

    def do_some_work(self) -> None:
        needed_data = self.producer.get_data()
        self.do_something(needed_data)

    def do_something(self, data: int) -> None:
        # Placeholder for actual work
        pass
```

- Consumer knows about producer
- Producer has a method that consumer can call
- Consumer asks producer for the data

Example: Clock using "Data pull" pattermn

```python
class IPullingClock(ABC):
    @abstractmethod
    def reset(self) -> None:
        """Sets the time to 0."""
        pass

    @abstractmethod
    def tick(self) -> None:
        """Increments the time."""
        pass

    @abstractmethod
    def get_time(self) -> int:
        """Returns the current time."""
        pass
```

`SimpleClock` (producer) implements `IPullingClock` interface, `ClockClient` is consumer:

```python
# Producer
class SimpleClock(IPullingClock):
    def __init__(self) -> None:
        self.time = 0

    def reset(self) -> None:
        self.time = 0

    def tick(self) -> None:
        self.time += 1

    def get_time(self) -> int:
        return self.time

# Consumer
class ClockClient:
    def __init__(self, the_clock: IPullingClock):
        self.the_clock = the_clock

    def get_time_from_clock(self) -> int:
        return self.the_clock.get_time()
```

Observer pattern

Potential problem with the "data pull" clock example:

What if the clock ticks once per second, but there are dozens of clients, each asking for the time every 10 msec?

Our clock might be overwhelmed!

Can we do better for the situation where the clock updates rarely, but the clients need the values often?

Observer pattern: producer tells consumer ("push")

```
class Consumer:
    def __init__(self) -> None:
        self.needed_data = 0

    def receive_notification(
        self, data_value: int
    ) -> None:
        self.needed_data = data_value

    def do_some_work(self) -> None:
        self.do_something(self.needed_data)

    def do_something(self, data: int) -> None:
        # Placeholder for actual work
        print(
            f"Doing something with data: {data}")
```

```
class Producer:
    def __init__(
        self, consumer: Consumer
    ) -> None:
        self.consumer = consumer
        self.the_data = 0

    def update_data(
        self, input_value: int
    ) -> None:
        self.the_data = \
            self.do_something_with_input(
                input_value)
        # notify the consumer about the change:
        self.consumer.receive_notification(
            self.the_data)

    def do_something_with_input(
        self, input_value: int
    ) -> int:
        # Placeholder for actual processing logic
        return input_value * 2  # Example processing
```

Observer pattern: producer tells consumer ("push")

- Producer notifies the consumer whenever the data is updated
- Probably more than one consumer

AKA Listener pattern, Publish-subscribe pattern

The object being observed (the "subject") keeps a list of the objects who need to be notified when something changes.

- subject = producer = publisher
- observer = consumer = subscriber = listener
- If a new object wants to be notified when the subject changes, it registers ("subscribes") with the subject.

Example: Pushing clock (Observer pattern)

```
class IPushingClock(ABC):
    @abstractmethod
    def reset(self) -> None:
        """Resets the time to 0."""
        pass

    @abstractmethod
    def tick(self) -> None:
        """Increments the time and sends a notification with the
        current time to all consumers."""
        pass

    @abstractmethod
    def add_listener(self, listener: 'IPushingClockClient') -> int:
        """Adds another consumer and initializes it with the current time."""
        pass

class IPushingClockClient(ABC):
    @abstractmethod
    def receive_notification(self, t: int) -> None:
        """Notifies the client with the current time."""
        pass
```

Example: Pushing clock (Observer pattern)

```
# Producer
class PushingClock(IPushingClock):
    def __init__(self) -> None:
        self.observers: list[IPushingClockClient] = []
        self.time = 0

    def add_listener(
        self, listener: 'IPushingClockClient'
    ) -> int:
        self.observers.append(listener)
        return self.time

    def notify_all(self) -> None:
        for obs in self.observers:
            obs.receive_notification(
                self.time)

    def reset(self) -> None:
        self.time = 0
        self.notify_all()

    def tick(self) -> None:
        self.time += 1
        self.notify_all()
```

```
# Consumer
class PushingClockClient(IPushingClockClient):
    def __init__(self, the_clock: IPushingClock
    ) -> None:
        self.time = the_clock.add_listener(self)

    def receive_notification(self, t: int) -> None:
        self.time = t
```

Observer decides what to do with the notification. Another option:

```
class DifferentClockClient(IPushingClockClient):
    """A client that receives notifications from a clock and
    stores TWICE the current time."""

    def __init__(self, the_clock: IPushingClock) -> None:
        self.twice_time = the_clock.add_listener(self) * 2
        self.notifications: list[int] = []

    def receive_notification(self, t: int) -> None:
        self.notifications.append(t)
        self.twice_time = t * 2
```

Push versus pull: tradeoffs

Pull	Push
Consumer knows about the Producer	Producer knows about the Consumer(s)
Producer must have a method that the Consumer can call	Consumer must have a method that Producer can use to notify it
Consumer asks the Producer for the data	Producer notifies the Consumer whenever the data is updated
Better when updates are more frequent than requests	Better when updates are rarer than requests

Details and variants

- How does the consumer get an initial value?
  - Here we've had the producer supply it when the consumer registers
- Should there be an unsubscribe method?
- What data should be passed with the notification?
- How does the producer store its registered consumers?
  - If many consumers, this could be an issue
- "There's a package for that"

Poll: Which of these scenarios might be well-served with the Observer pattern?

1. a spreadsheet application: when a cell value changes, other dependent cells all update automatically
2. database search results: instead of loading all query results at once, it pulls data in "chunks" as needed. The user's scrolling triggers a request for the next "chunk"
3. when a news service publishes articles, it sends notifications to the subscribers (email, mobile news apps, social media, archives)
4. API rate limiting: let clients request data at their processing speed. The API server's data processing pipeline processes new jobs only when it's ready, preventing queue overflow.

Note: All options are either Observer or Data Pull pattern.

Hashing

Poll: What does this print?

```
class Course:
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

course_oakland = Course('CS', 2100)
course_boston = Course('CS', 2100)

print(course_oakland == course_boston)
```

1. True
2. False

Poll: How about now?

```
class Course:
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Course):
            raise NotImplementedError
        else:
            return self.department == other.department and self.course == other.course


course_oakland = Course('CS', 2100)
course_boston = Course('CS', 2100)

print(course_oakland == course_boston)
```

1. True
2. False

Moving on to the "hashing" topic...

If we try to add a `Course` to a `set`, it raises a `TypeError`

```
class Course:
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Course):
            raise NotImplementedError
        else:
            return self.department == other.department and self.course == other.course


course_oakland = Course('CS', 2100)

courses: set[Course] = {course_oakland}  # TypeError: unhashable type: 'Course'
```

And same thing if you try to add it as a key to a `dict`

In order to put someting in a `set` or `dict`, it needs to follow the Hashable protocol

Hashable protocol's required method: `__hash__(self) -> int`

Corresponding interface: `from collections.abc import Hashable`

Why hashing?

If we avoid the `TypeError: unhashable type: 'Course'`:

- Just use a `list` instead
- But `list`s are slower than `set`s in some situations:

```
T = TypeVar('T')

def list_contains(item: T, list: list[T]) -> bool:
    """Returns True if the item is in the list, and False otherwise"""
    for element in list:
        if element == item:
            return True
    return False
```

Can make the code smaller, but still need to check each element

And sets check containment in *constant time* (i.e., time does not depend on list length)

How hashing makes sets magically fast

Definitions:

- *hash* (verb): To map a value to an integer index
- *hash table* (noun): A list that stores elements via hashing
- *hash set* (noun): A set of elements stored using the same hash function
- *hash function* (noun): An algorithm that maps values to indexes

One possible hash function for integers: `i % length`

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value										

How hashing makes sets magically fast

One possible hash function for integers: `i % length`

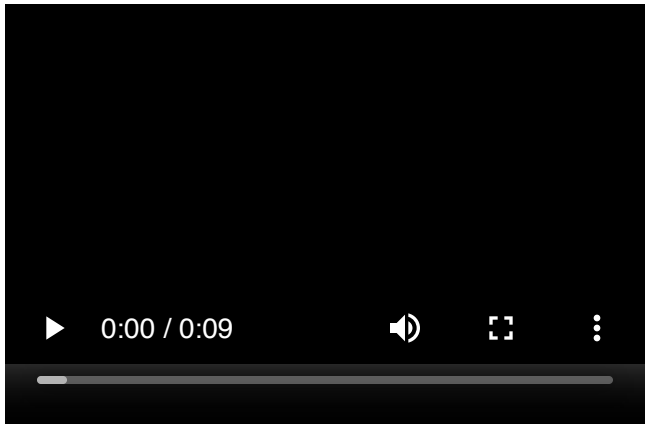
```
set.add(11)  # 11 % 10 == 1
set.add(49)  #  49 % 10 == 9
set.add(24)  #  24 % 10 == 4
set.add(7)  #  7 % 10 == 7
```

index	0	1	2	3	4	5	6	7	8	9
value		11			24			7		49

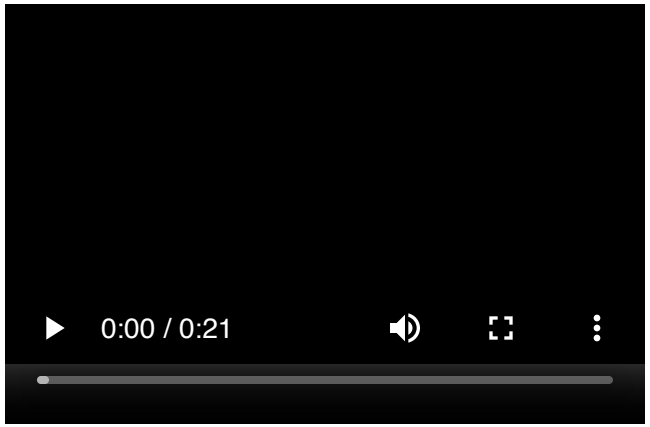
Okay, getting there, but there are some issues with that method...

dog hash collision gif

Source: https://m.vayagif.com/busqueda/0/el perro no nace agresivo/p/893



cat collision gif



no collision

Source: Tyler Yeats

How hashing makes sets magically fast

- *collision*: When hash function maps 2 values to same index

```
set.add(11)
set.add(49)
set.add(24)
set.add(7)

set.add(54)  # collides with 24
```

index	0	1	2	3	4	5	6	7	8	9
value		11			24			7		49

- *collision resolution* (noun): An algorithm for fixing collisions

How hashing makes sets magically fast

- *probing* (verb): Resolving a collision by moving to another index

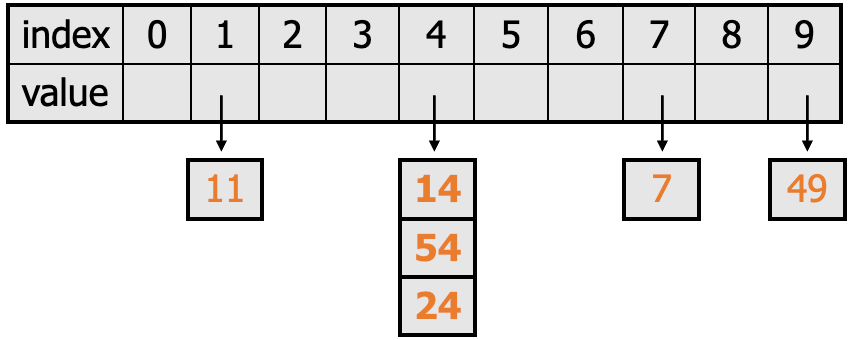
```
set.add(11)
set.add(49)
set.add(24)
set.add(7)

set.add(54)  # spot is taken -- probe (move to the next available spot)
```

index	0	1	2	3	4	5	6	7	8	9
value		11			24	54		7		49

How hashing makes sets magically fast

- *chaining* (verb): Resolving collisions by storing a list at each index
  - add / search / remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes



Poll: Which of these can be done in constant time?

1. Checking if an element is in a set
2. Adding an element to a set
3. Removing an element from a set
4. Checking if a key is present in a map
5. Getting the value associated with a key in a map
6. Changing the value associated with a key in a map
7. Checking if a value appears in a map

What makes a good hash code?

`__hash__()` should return an `int` which is relatively unique to that object (so it can be used as an index in the hash table)

Rules for `__hash__()`:

- `__hash__()` must always return the same value for a given object
- If two objects are equal, then `__hash__()` must return the same value for them

Poll: Is this a legal hash function?

```
    def __hash__(self) -> int:
        return 42
```

1. Yes
2. No

In addition to the rules for hash functions...

Desired characteristics for `__hash__()`:

- We would like different objects to have different values
- The hash function should be quick to compute (ideally constant time)

Poll: Strings, numbers, and tuples are hashable by default in Python. Lists, sets, and dictionaries are not hashable by default in Python. Why might that be?

1. Because we rarely need to add lists, sets, or dictionaries to a set, or anything that requires hashing
2. Because lists, sets, and dictionaries are mutable, which could result in a changing hash code
3. Because lists, sets, and dictionaries are rarely equal to each other, so they don't need a hash code
4. Because lists, sets, and dictionaries can hold `None` in them, which shouldn't get a hash code

```python
from collections.abc import Hashable

class Course(Hashable):
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

    def __str__(self) -> str:
        return f'{self.department}{self.course}'

    def __repr__(self) -> str:
        return self.__str__()

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Course):
            raise NotImplementedError
        else:
            return self.department == other.department and self.course == other.course

    def __hash__(self) -> int:
        return hash(str(self))


course_oakland = Course('CS', 2100)
course_boston = Course('CS', 2100)

courses: set[Course] = {course_oakland}
courses.add(course_boston)
```

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?