CS 4530: Fundamentals of Software Engineering

Module 11.1: Interaction-Level Design Patterns

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

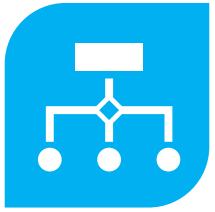
- At the end of this lesson you should be able to:
  - Explain how patterns capture common solutions and tradeoffs for recurring problems.
  - Explain and give an example of each of the following:
    - The Demand-Pull pattern
    - The Data-Push (aka Listener or Observer) pattern
    - The Singleton pattern
  - Do the same for other mini-patterns
    - Dependency Injection
    - The Delegate or Callback pattern
    - The first-time-through switch

A Different Perspective On The Three Scales of Design

PROGRAMS

PLANNING

ORGANIZING



IMPLEMENTING

The Structural Scale

- key questions: what are the pieces? how do they fit together to form a coherent whole?

The Interaction Scale

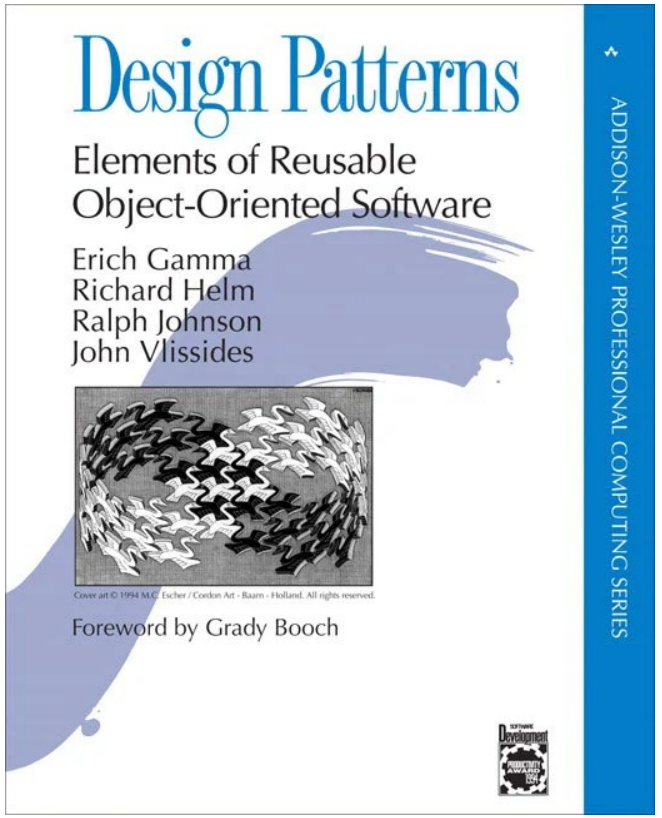
- key questions: how do the pieces interact? how are they related?

The Code Scale

- key question: how can I make the actual code easy to test, understand, and modify?

Design at the Interaction Level corresponds to "OOD Design Patterns"

- Four people in the 90's wrote a book that lists a lot of patterns.
- But this is not the be-all and end-all of patterns
- We'll see patterns at lots of different levels.



Pull and Push patterns

- Lots of times we need to get some data from one place in our program to another.
- There are two basic ways of accomplishing this, which we call "pull" and "push"
- In each situation we have two objects, which we call the "producer" and the "consumer"
- But there many variations. Let's look at a few of them.

Let's start with a very simple example

- We have a clock, which is updated every so often with the current time
  - right now, we don't care how that happens
- We have several client modules that need to know the current time.
- The clock is the producer
- The client modules are the consumers

Example: Interface For a Simple Clock

src/pullingClock/IPullingClock.ts

```
export interface IPullingClock {
  /** sets the time to 0 */
  reset(): void;

  /** increments the time */
  tick(): void;

  /** returns current time */
  currentTime(): number;
}
```

Example: Implementing a Simple Clock

```ts
export interface IPullingClock {
  /** sets the time to 0 */
  reset(): void;

  /** increments the time */
  tick(): void;

  /** returns current time */
  currentTime(): number;
}
```

src/pullingClock/simpleClockUsingPull.ts

```ts
export class SimpleClock implements IPullingClock {
  private _time = 0;
  reset() {
    this._time = 0;
  }
  tick() {
    this._time += 1;
  }
  currentTime() {
    return this._time;
  }
}
```

Example: Using a Simple Clock

src/pullingClock/simpleClockUsingPull.ts

```ts
export interface IPullingClock {
  /** sets the time to 0 */
  reset(): void;

  /** increments the time */
  tick(): void;

  /** returns current time */
  currentTime(): number;
}
```

```ts
export class ClockClient {
  private _theClock: IPullingClock;

  constructor(theClock: IPullingClock) {
    this._theClock = theClock;
  }

  get time(): number {
    return this._theClock.currentTime();
  }
}
```

Testing the simple clock

src/pullingClock/simpleClockUsingPull.test.ts

```
import { SimpleClock, ClockClient } from "./simpleClockUsingPull.ts";

test("test of SimpleClock", () => {
  const clock1 = new SimpleClock();
  expect(clock1.currentTime()).toBe(0);
  clock1.tick();
  clock1.tick();
  expect(clock1.currentTime()).toBe(2);
  clock1.reset();
  expect(clock1.currentTime()).toBe(0);
});
```

We call this the "demand-pull" pattern

- because the when the client needs some data, it *pulls* the data it needs from the server.
- Alternative names: you could call the SimpleClock the “producer” and call the ClockClient the “consumer”

But there's a potential problem here.

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

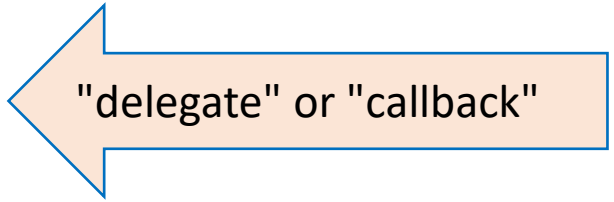
The 'data-push' pattern

- Instead, let's arrange it so that the server *pushes* the data to the consumer only when it changes
- That will make it the responsibility of the clock to keep track of the clients it needs to notify.
- We can do that in several ways; let's use one that matches one you've seen before.

src/pushingClock/IPushingClock.ts

Interface For a Clock Using the Push Pattern

```ts
export type IPushingClockListener = (time: number) => void;

export interface IPushingClock {
  /** increments the time, notifies consumers of the new time */
  tick():void

  /** resets the time to 0 */
  reset():void

  /** returns the current time */
  currentTime(): number

  /** adds a new consumer */
  addListener(listener: IPushingClockListener): void
}
```



src/pushingClock/pushingClock.ts

Implementing a Clock with the Push Pattern

```
export class PushingClock implements IPushingClock {
  private _time = 0;
  private _observers: IPushingClockListener[] = [];
  private _notifyAll() {
    this._observers.forEach((observer) => observer(this._time));
  }
  /** resets the time to 0 */
  reset() { this._time = 0; this._notifyAll(); }
  /** increments the time, notifies consumers of the new time */
  tick() { this._time += 1; this._notifyAll(); }
  /** returns the current time */
  currentTime() { return this._time; }
  /** adds a new consumer */
  addListener(observer: IPushingClockListener) {
    this._observers.push(observer);
  }
}
```

src/pushingClock/pushingClockClients.ts

Using a Pushing Clock

```
export type IPushingClockListener =
  (time: number) => void;

export interface IPushingClock {
  /** as before */
  tick():void
  reset():void
  /** adds a new consumer */
  addListener(listener: IPushing…
}
```

```
export class PushingClockClient {
  private _time: number;
  constructor(theClock: IPushingClock) {
    this._time = theClock.currentTime();
    theClock.addListener((time) => {
      this._time = time;
    });
  }
  get time(): number {
    return this._time;
  }
}
```

A More Conventional Design of a Pushing Clock

```
export interface INotifyingClock {
/** as before */
  tick(): void;
  reset(): void;
  currentTime(): number;

  /** adds a new consumer */
  addClient(client: INotifyingClockClient): void;

}

export interface INotifyingClockClient {
  /** called when the clock ticks, with the new time */
  onClockTick(time: number): void;
}
```

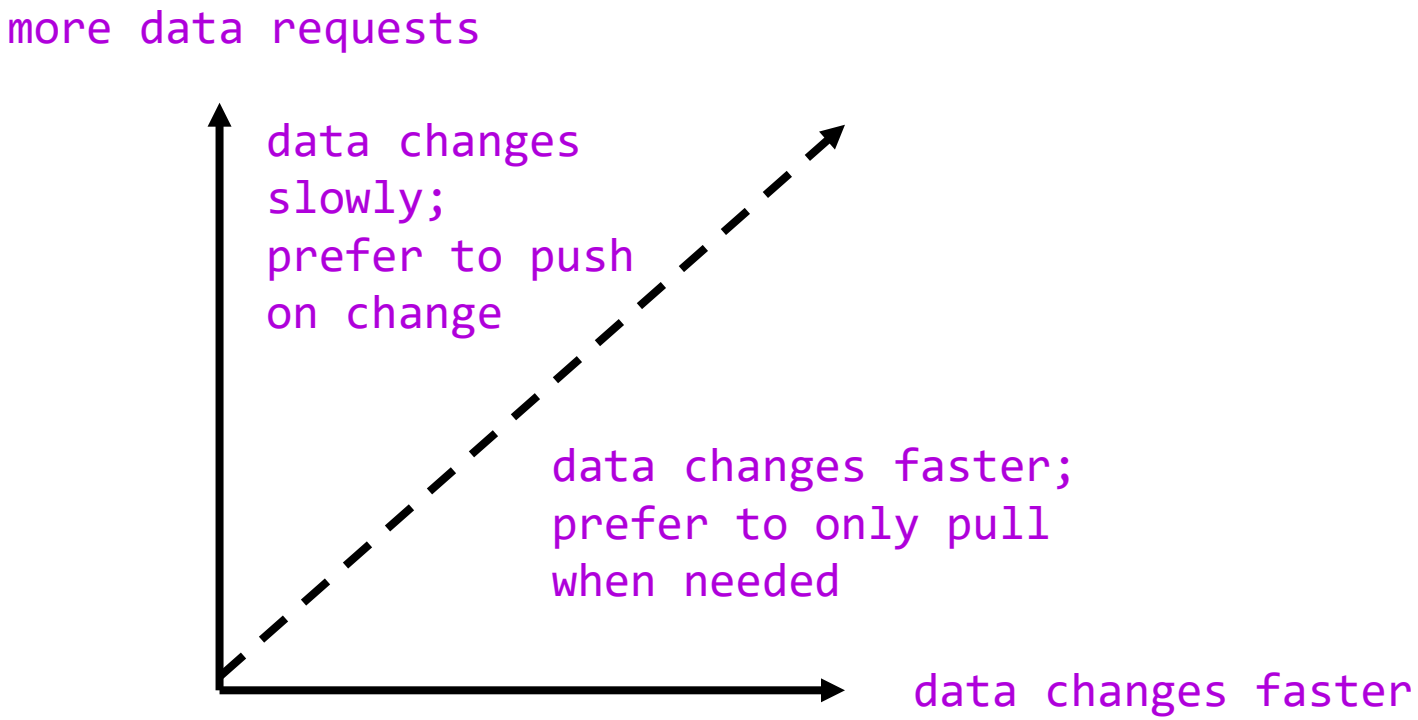
A More Conventional Client

```
export class NotifyingClockClient implements INotifyingClockClient {
  private _time: number;
  constructor(theClock: INotifyingClock) {
    this._time = theClock.currentTime();
    theClock.addClient(this)
  }


  onClockTick(time: number) {
    this._time = time;
  }


  currentTime(): number {
    return this._time;
  }
}
```

This these are all instances of the Listener or Observer Pattern

- Also called "publish-subscribe pattern” (or “pub/sub”)
  - Some people make fine-grained distinctions between pub/sub and observer/observable; we won’t do that here.
- The object being observed (the “subject”) keeps a list of the objects who need to be notified when something changes.
  - subject = producer = publisher
- When a new object (i.e., the “consumer”) wants to be notified when the subject changes, it registers with ("subscribes to") the subject/producer/publisher
  - observer = consumer = subscriber = listener

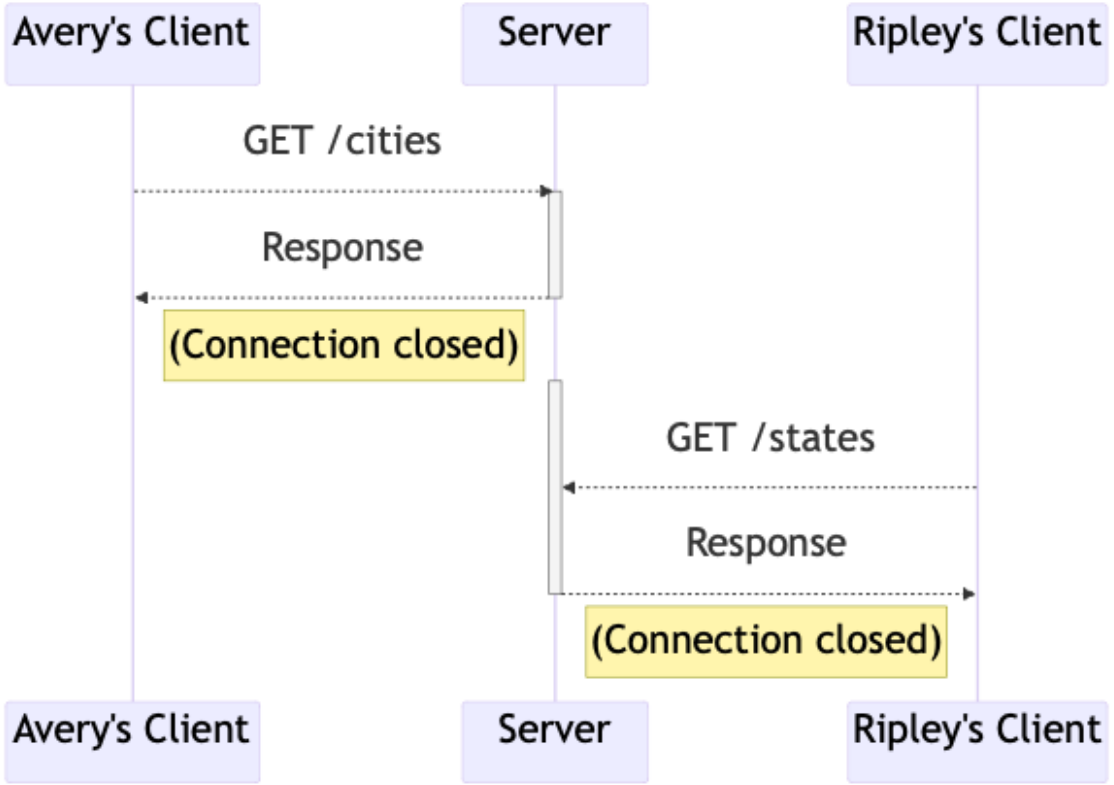
Push or Pull?


more data requests
data changes slowly; prefer to push on change
data changes faster; prefer to only pull when needed
data changes faster

Push or Pull?
We've Seen This Already (1)

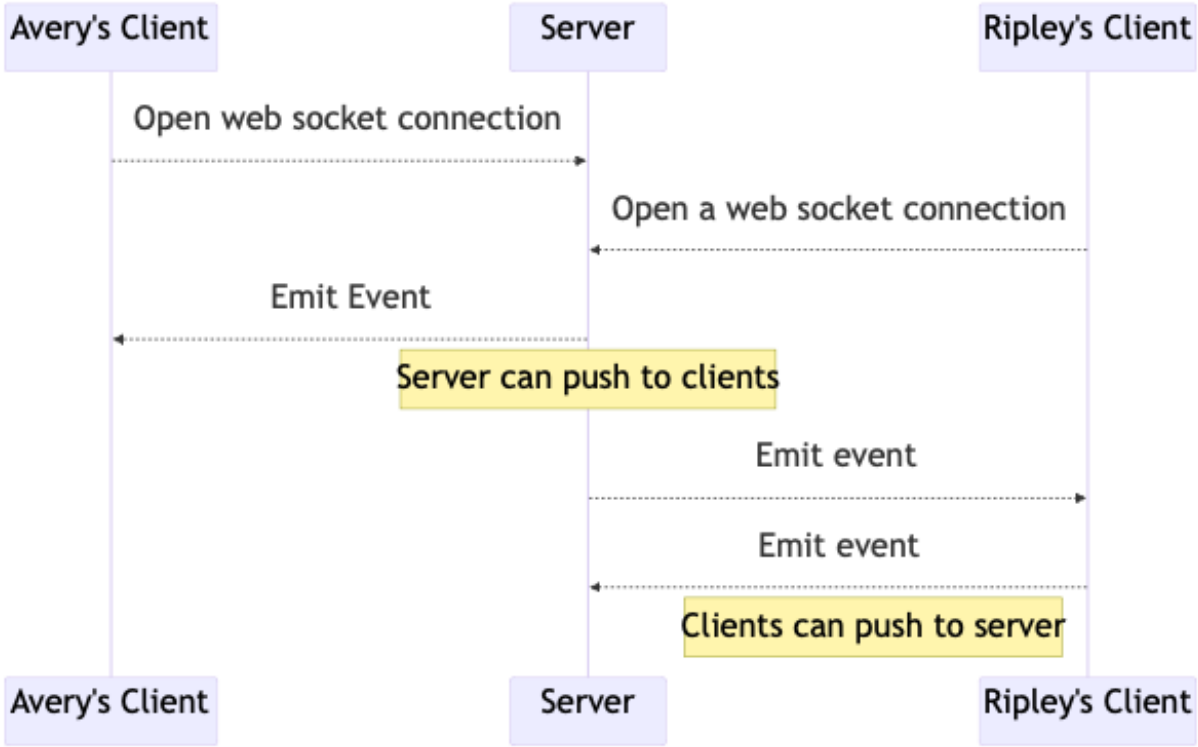
HTTP

(one-way, client initiates)



Web Sockets

(two-way after client opens socket)



Push or Pull?
We've Seen This Already (2)

- The clock and JavaScript event handling both use *callbacks* — a mini-pattern of their own — with the push pattern.
- With `onClick`, the browser is the producer/server, your app is the consumer/client

```
export class PushingClockClient {
  private _time: number;
  constructor(theClock: IPushingClock) {
    this._time = theClock.currentTime();
    theClock.addListener((time) => {
      this._time = time;
    });
  }
  get time(): number {
    return this._time;
  }
}
```

```
<button
  onClick={() => {
    navigate(`/profile/${user.username}`)
  }
>
  View Profile
</button>
```

Pattern #3: The Singleton Pattern

- Maybe you only want one clock in your system.
- You can't just say "new Clock" because that always creates a new object of class Clock.

Test-Driven Development

src/pullingclock/simpleClockUsingPull.test.ts

```
test("clocks are independent", () => {
  const clock1 = new SimpleClock();
  const clock2 = new SimpleClock();
  expect(clock1.currentTime()).toBe(0);
  expect(clock2.currentTime()).toBe(0);
  clock1.tick();
  clock1.tick();
  clock1.tick();
  expect(clock1.currentTime()).toBe(3);
  expect(clock2.currentTime()).toBe(0);
});
```

src/singletonClock/singletonClock.test.ts

```
test("clocks are NOT independent", () => {
  const clock1 = ???
  const clock2 = ???
  expect(clock1.currentTime()).toBe(0);
  expect(clock2.currentTime()).toBe(0);
  clock1.tick();
  clock1.tick();
  clock1.tick();
  expect(clock1.currentTime()).toBe(3);
  expect(clock2.currentTime()).toBe(3);
});
```

One Singleton Pattern Implementation: First-time Through Switch

```
class PrivateClock implements IPullingClock {
  private _time = 0;
  reset() { this._time = 0; }
  tick() { this._time++; }
  currentTime() { return this._time; }
}

let privateClock: null | IPullingClock = null;

export function clockInstance(): IPullingClock {
  if (!privateClock) {
    privateClock = new PrivateClock();
  }
  return privateClock;
}
```

the "first-time through" switch is a mini-pattern itself!

Singleton Pattern
We've Seen This Already

- Each repository model (GameRepo, ChatRepo, etc) keeps track of a singleton Keyv object specific to that repository
- The first time a repository method is accessed, a first-time-through switch initializes the key-value store

Review: Learning Goals for this Lesson

- You should now be able to:
  - Explain how patterns capture common solutions and tradeoffs for recurring problems.
  - Explain and give an example of each of the following:
    - The Demand-Pull pattern
    - The Data-Push (aka Listener or Observer) pattern
    - The Singleton pattern
  - Do the same for other mini-patterns
    - Dependency Injection
    - The Delegate or Callback pattern
    - The first-time-through switch

Review: Learning Goals for this Lesson

- You should now be able to:
  - Explain how patterns capture common solutions and tradeoffs for recurring problems.
  - Explain and give an example of each of the following:
    - The Demand-Pull pattern
    - The Data-Push (aka Listener or Observer) pattern
    - The Singleton pattern