

CS 4530: Fundamentals of Software Engineering

Module 5: Interaction-Level Design Patterns

Adeel Bhutta, Joydeep Mitra, and Mitch Wand
Khoury College of Computer Sciences

Learning Goals for this Lesson

- By 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 Callback or Handler pattern
 - The Typed-Emitter pattern

What is a Pattern?

- A Pattern is a summary of a standard solution (or solutions) to a specific class of problems.
- A pattern should contain
 - A statement of the problem being solved
 - A solution of the problem
 - Alternative solutions
 - A discussion of tradeoffs among the solutions.
- For maximum usefulness, a pattern should have a name.
 - So you can say “here I’m using pattern P” and people will know what you had in mind.

Patterns help communicate intent

- If your code uses a well-known pattern, then the reader has a head start in understanding your code.

Patterns are intended to be flexible

- We will not engage in discussion about whether a particular piece of code is or is not a “correct” instance of a particular pattern.

This week we will talk about the interaction scale

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?

Example: Interface for a simple clock

```
export default interface ISimpleClock {  
  
    /** sets the time to 0 */  
    reset():void  
  
    /** increments the time */  
    tick():void  
  
    /** returns the current time */  
    getTime():number  
}
```

Tests for the clock and the client describe their desired behavior

```
import { SimpleClock, ClockClient } from "../simpleClock"
```

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

```
test("test of ClockClient", () => {  
  const clock1 = new SimpleClock  
  expect(clock1.time).toBe(0)  
  const client1 = new ClockClient(clock1)  
  expect(clock1.time).toBe(0)  
  expect(client1.getTimeFromClock()).toBe(0)  
  clock1.tick()  
  clock1.tick()  
  expect(client1.getTimeFromClock()).toBe(2)  
})
```

src/clockWithPull/simpleClock.test.ts

simpleClockUsingPull.ts

```
import IClock from "../IPullingClock";

export class SimpleClock implements IClock {
  private time = 0
  public reset () : void {this.time = 0}
  public tick () : void { this.time++ }
  public get time(): number { return this.time }
}

export class ClockClient {
  constructor (private theclock:IClock) {}
  getTimeFromClock ():number {
    return this.theclock.time
  }
}
```

SimpleClock is the Producer

ClockClient is the Consumer

The code is simple...

```
import IClock from "../simpleClock.interface";

export class SimpleClock implements IClock {
  private time = 0

  public reset () : void {this.time = 0}

  public tick () : void { this.time++ }

  public getTime(): number { return this.time }
}

export class ClockClient {
  constructor (private theclock:IClock) {}

  getTimeFromClock ():number {return this.theclock.getTime()}
}
```

We call this the "demand-pull" pattern

- because the when the client needs som data, it *pulls* the data it needs from the server.
- Alternative names: you could call these the consumer and the producer.

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

The 'data-push' pattern

- We'd like to arrange it so that the server *pushes* the data to the consumer only when it changes
- To accomplish that, the consumer needs to advertise an 'update' method that the producer can call.

This is called the Listener or Observer Pattern

- Also called "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
- 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

Interface for a clock using the Push pattern

```
export interface IPushingClock {  
  
    /** resets the time to 0 */  
    reset():void  
  
    /**  
     * increments the time and sends a .nofify message with the  
     * current time to all the consumers  
     */  
    tick():void  
  
    /** adds another consumer and initializes it with the current time */  
    addListener(listener:IPushingClockClient):number  
}
```

Interface for a clock listener

```
interface IPushingClockClient {  
    /**  
     * * @param t - the current time, as reported by the clock  
     */  
    notify(t:number):void  
}
```


Tests

```
test("single observer", () => {  
  const clock1 = new PushingClock()  
  const observer1  
    = new PushingClockClient(clock1)  
  expect(observer1.time).toBe(0)  
  clock1.tick()  
  clock1.tick()  
  expect(observer1.time).toBe(2)  
})
```

```
test("Multiple Observers", () => {  
  const clock1 = new PushingClock()  
  const observer1  
    = new PushingClockClient(clock1)  
  const observer2  
    = new PushingClockClient(clock1)  
  const observer3  
    = new PushingClockClient(clock1)  
  clock1.tick()  
  clock1.tick()  
  expect(observer1.time).toBe(2)  
  expect(observer2.time).toBe(2)  
  expect(observer3.time).toBe(2)  
})
```

A PushingClock class

```
export class PushingClock implements IPushingClock {  
  private observers: IPushingClockClient[] = []  
  public addListener(obs: IPushingClockClient): number {  
    this.observers.push(obs);  
    return this.time  
  }  
  private notifyAll(): void {  
    this.observers.forEach(obs => obs.notify(this.time))  
  }  
  private time = 0  
  reset(): void { this.time = 0; this.notifyAll() }  
  tick(): void { this.time++; this.notifyAll() }  
}
```

A Client

```
export class PushingClockClient implements IPushingClockClient
{
    private time:number
    constructor (theclock:IPushingClock) {
        this.time = theclock.addListener(this)
    }

    notify (t:number) : void {this.time = t}
    getTime () : number {return this.time}
}
```

The observer can do whatever it likes with the notification

```
export class DifferentClockClient implements IPushingClockClient {
```

```
    /** TWICE the current time, as reported by the clock */  
    private twiceTime:number
```

```
    constructor (theclock:IPushingClock) {  
        this.twiceTime = theclock.addListener(this) * 2  
    }
```

```
    /** list of all the notifications received */  
    public readonly notifications : number[] = [] // just for fun
```

```
    notify(t: number) : void {  
        this.notifications.push(t)  
        this.twiceTime = t * 2  
    }
```

src/pushingClock/pushingClockClients.ts

```
    time : number { return (this.twiceTime / 2) }
```

```
}
```

Better test this, too

```
test("test of DifferentClockClient", () => {  
  const clock1 = new PushingClock()  
  const observer1 = new DifferentClockClient(clock1)  
  expect(observer1.time).toBe(0)  
  clock1.tick()  
  expect(observer1.time).toBe(1)  
  clock1.tick()  
  expect(observer1.time).toBe(2)  
})
```

Tests for .notifications method

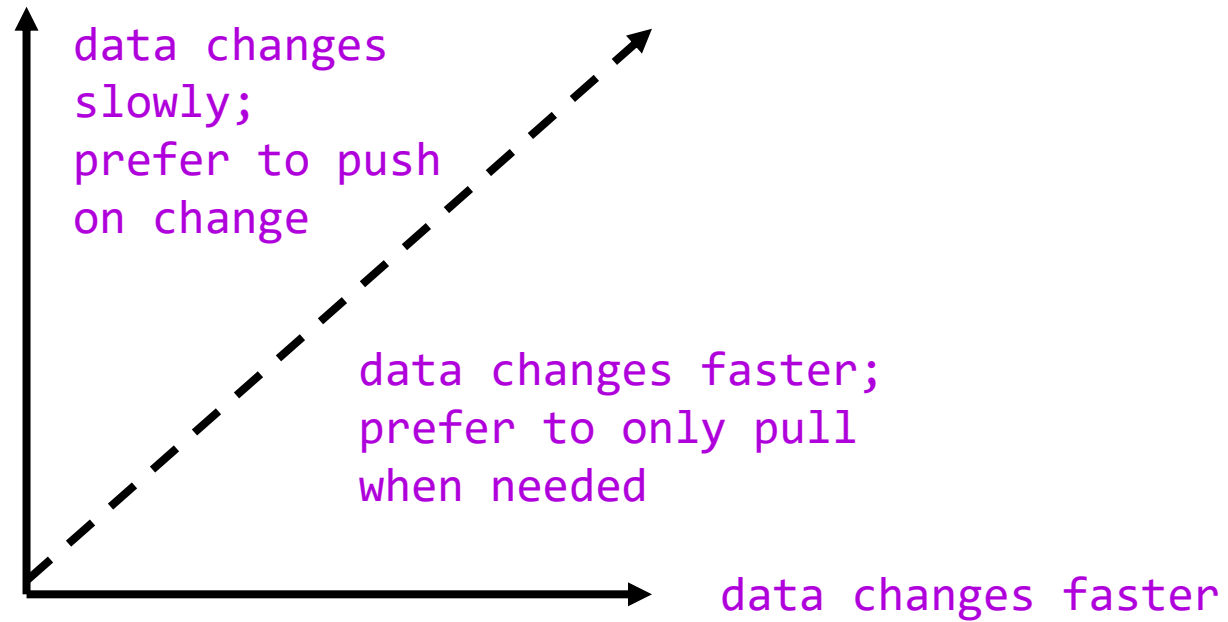
```
test("DifferentClockClient accumulates the times correctly", ()  
=> {  
    const clock1 = new PushingClock()  
    clock1.tick()  
    const differentClient = new DifferentClockClient(clock1)  
    expect(differentClient.time).toBe(1)  
    expect(differentClient.notifications).toEqual([])  
    clock1.tick()  
    clock1.tick()  
    clock1.tick()  
    expect(differentClient.time).toBe(4)  
    expect(differentClient.notifications).toEqual([2, 3, 4])  
})
```

src/pushingClock/pushingClock.test.ts

There are more tests in here;
you should look at them.

Push or Pull?

more data requests



Maybe the server doesn't want to give the client access to all of its methods

```
export class DifferentClockClient implements IPushingClockClient {
```

```
    /** TWICE the current time, as reported by the clock */
```

```
    private twiceTime:number
```

```
    constructor (theclock:IPushingClock) {
```

```
        this.twiceTime = theclock.addListener(this) * 2
```

```
    }
```

```
    /** list of all the notifications received */
```

```
    public readonly notifications : number[] = [] // just for fun
```

```
    notify(t: number) : void {
```

```
        this.notifications.push(t)
```

```
        this.twiceTime = t * 2 }
```

```
    time : number { return (this.twiceTime / 2) }
```

```
}
```

DANGER!!

src/pushingClock/pushingClockClients.ts

Pattern #3: The callback or handler pattern

- Maybe the server doesn't want to give the client access to all of the server's methods.
- the server constructs the client and gives it a *function* to call instead.
- Typically, this will be a function inside the server
- We call this function the *callback* or *handler* for the client's action.
- This pattern is used all the time in REACT.

The interface

```
export interface ICallbackServer {  
    // returns a new client that satisfies the ICallbackClient interface  
    newClient(clientName: string): ICallbackClient;  
  
    // returns the log of all push messages received  
    log(): string[];  
}  
  
export interface ICallbackClient {  
    // sends a push message to the server,  
    sendPush: () => void  
    // asks the server for list of all push messages received  
    // from all clients.  
    getLog: () => string[];  
}
```

Example: Expected Behavior

```
it('works', () => {  
  
  const server = new Server()  
  const client1 = server.newClient("A")  
  const client2 = server.newClient("B")  
  
  client1.sendPush()  
  expect(client1.getLog()).toEqual(["A"])  
  client2.sendPush()  
  expect(client1.getLog()).toEqual(["A", "B"])  
  expect(client2.getLog()).toEqual(["A", "B"])  
  // now client1 pushes again  
  client1.sendPush()  
  // now the clients can see all the pushes  
  expect(client1.getLog()).toEqual(["A", "B", "A"])  
  expect(client2.getLog()).toEqual(["A", "B", "A"])  
  
})
```

The Server

```
export class Server implements ICallbackServer {  
  
    // the log of all push messages received  
    private _log: string[] = []  
  
    // using arrow function to bind 'this' correctly  
    private pushHandler = (clientName: string) => () => {this._log.push(clientName)}  
    private logHandler = () : string[] => { return this._log }  
  
    public log(): string[] { return this._log }  
    public newClient(clientName:string): ICallbackClient {  
        return new Client(this.pushHandler(clientName), this.logHandler)  
    }  
}
```

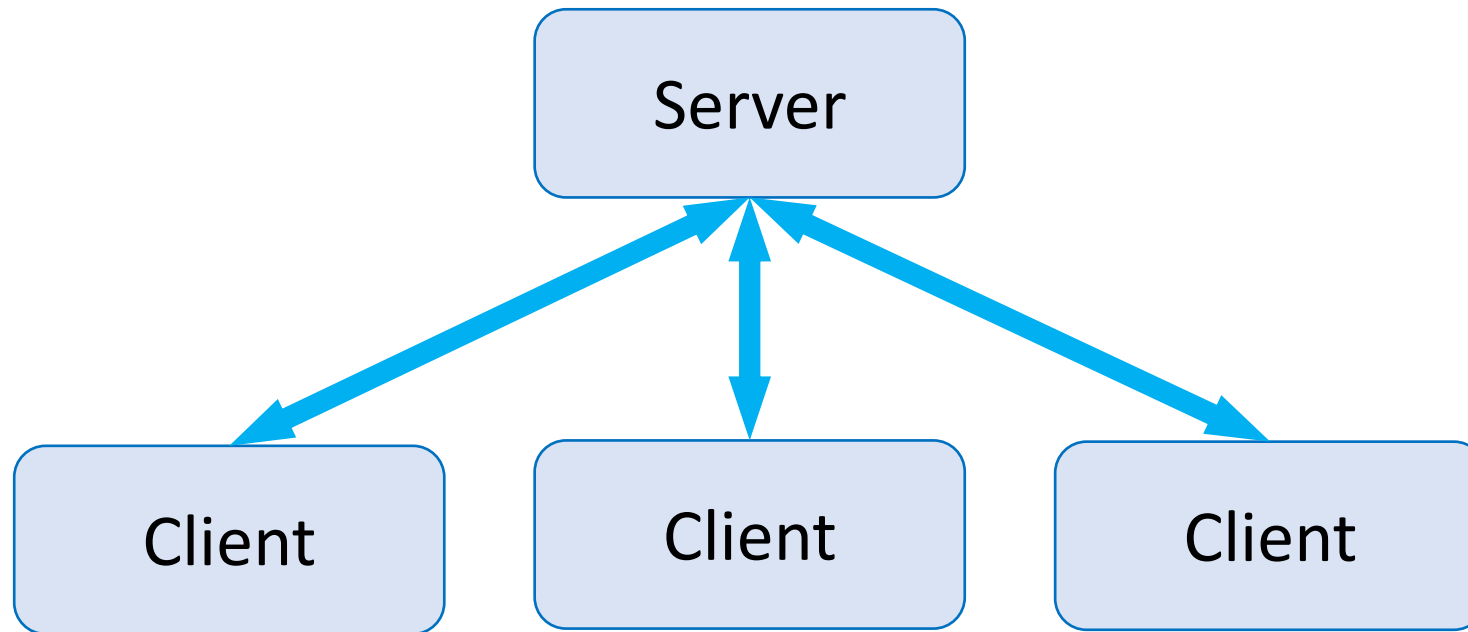
The Client

```
export class Client implements ICallbackClient {  
  
    // the client doesn't get to see the server directly; the server  
    // creates it with two callbacks.  
  
    constructor(  
        _pushHandler: () => void,  
        _logHandler: () => string[],  
    ) { this.pushHandler = _pushHandler; this.logHandler = _logHandler; }  
  
    private pushHandler: () => void  
    private logHandler: () => string[]  
  
    // the public methods just call the callbacks  
    public sendPush() { this.pushHandler(); }  
    public getLog(): string[] { return this.logHandler(); }  
  
}
```

Pattern #4: The Typed-Emitter Pattern

- What if the data source wants to notify its listeners with several different kinds of messages?
- Maybe with different data payloads?
- And what if we want to take advantage of type-checking?

Emitters use a server/client model



- Client can send a message to its server
- Server can send a message to an individual client
- ...or to some or all its clients

Typed Emitters use types to specify messages that servers and clients can exchange

```
// a simple ping-pong protocol for testing WebSocket connections.
```

```
// server starts with (ping 0)
```

```
// client replies to server 'ping n' with 'pong n' (n <= 5)
```

```
// server replies to client 'pong n' with 'ping n+1'
```

```
export interface ServerToClientEvents {  
  'ping': (count:number) => void;  
}
```

```
export interface ClientToServerEvents {  
  'pong': (clientName: string, count:number) => void;  
  'goodbye': (clientName: string) => void;  
}
```

Note: this is the interface for the socket-io implementation of emitters. Other implementations use somewhat different interfaces.

Emitters typically provide many methods

```
export interface EventEmitter {  
  
    /** The event callbacks are called with the passed arguments */  
    emit(type, ... args);  
    /** Run callback every time event is emitted */  
    on(event, callback);  
    /** Run callback when event is emitted just for the first time */  
    once(event, callback);  
    /** Removes the callback for event */  
    off(event, callback);  
    /** Removes all callbacks for event */  
    off(event);  
    /** Removes all callbacks for all events */  
    off();  
}
```

This pattern can be used across multiple machines using websockets.

- Websockets is a standard, but low-level protocol for sending messages between machines.
- [Socket.io](#) provides a typed-emitter-style programming model for webSockets.
- It also provides automatic reconnection, broadcast rooms, and other goodies

Using an emitter: at the Server end

```
function startClientHandlers(  
  socket: Socket<ServerToClientEvents, ClientToServerEvents>,  
  clientName: string  
) {  
  
  // system starts by sending 'connect'  
  socket.on('connect', () => {  
    console.log(`${clientName} connected to server on ${clientURL}`);  
  });  
  
  socket.on('ping', (n: number) => {  
    console.log(`${clientName} received ping with count ${n}`);  
    if (n <= 5) {  
      socket.emit('pong', clientName, n); // reply with pong  
    } else { // invariant violated! disconnect if count exceeds 5  
      console.log(`${clientName} received ping with count ${n} > 5`)  
      socket.disconnect();  
    }  
  })  
};
```

Look at the file
if you want to see
where the socket
comes from.

Using an emitter: at the Client end

```
function startClientHandlers(socket:
  Socket<ServerToClientEvents, ClientToServerEvents>) {

  // system starts by sending 'connect'
  socket.on('connect', () => {
    console.log(`${this.clientName} connected to server on ${clientURL}`);
  });

  socket.on('ping', (n: number) => {
    console.log(`${this.clientName} received ping with count ${n}`);
    if (n < 5) {
      this.socket.emit('pong', this.clientName, n); // reply with pong
    } else {
      this.socket.emit('goodbye', this.clientName);
      this.socket.disconnect(); // disconnect after 5 pings
      process.exit(0); // exit the process
    }
  });
}
```

Look at the file
if you want to see
where the socket
comes from.

Choreographies and Projections

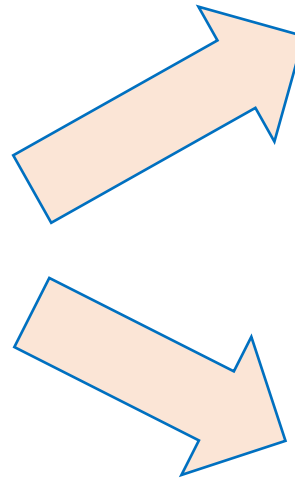
Choreography

```
// a simple ping-pong protocol for testing WebSocket connect:

// server starts with (ping 0)
// client replies to server 'ping n' with 'pong n' (n <= 5)
// server replies to client 'pong n' with 'ping n+1'

export interface ServerToClientEvents {
  'ping': (count:number) => void;
}

export interface ClientToServerEvents {
  'pong': (clientName: string, count:number) => void;
  'goodbye': (clientName: string) => void;
}
```



```
function startClientHandlers(
  socket: Socket<ServerToClientEvents, ClientToServerEvents>,
  clientName: string
) {

  // system starts by sending 'connect'
  socket.on('connect', () => {
    console.log(`${clientName} connected to server on ${clientURL}`);
  });

  socket.on('ping', (n: number) => {
    console.log(`${clientName} received ping with count ${n}`);
    if (n <= 5) {
      socket.emit('pong', clientName, n); // reply with pong
    } else { // invariant violated! disconnect if count exceeds 5
      console.log(`${clientName} received ping with count ${n} >`);
      socket.disconnect();
    }
  });
}
```

```
function startClientHandlers(socket:
  Socket<ServerToClientEvents, ClientToServerEvents>) {

  // system starts by sending 'connect'
  socket.on('connect', () => {
    console.log(`${this.clientName} connected to server on ${clientURL}`);
  });

  socket.on('ping', (n: number) => {
    console.log(`${this.clientName} received ping with count ${n}`);
    if (n < 5) {
      this.socket.emit('pong', this.clientName, n); // reply with pong
    } else {
      this.socket.emit('goodbye', this.clientName);
      this.socket.disconnect(); // disconnect after 5 pings
      process.exit(0); // exit the process
    }
  });
}
```

Look at the file
if you want to see
where the socket
comes from.

Projections

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 Callback or Handler pattern
 - The Typed-Emitter pattern