S

Single Responsibility Principle (SRP): A class or module should have only one reason to change. In Node.js, this means each module should have a single responsibility or concern.

```
class User {
    constructor(name, email) {
      this.name = name;
      this.email = email;
    }
  }

  class UserRepository {
    saveToDatabase(user) {
      // Logic to save user data to the database
      console.log(`Saving user ${user.name} to the database...`);
    }
  }

  class EmailService {
    sendWelcomeEmail(user) {
      // Logic to send a welcome email to the user
      console.log(`Sending welcome email to ${user.email}...`);
    }
  }

  const user = new User('Alice', 'alice@example.com');
  const userRepository = new UserRepository();
  const emailService = new EmailService();

  userRepository.saveToDatabase(user);
  emailService.sendWelcomeEmail(user);
```

```
avoid Use this
X X X X
class User {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }


  saveToDatabase() {
    // Logic to save user data to the database
    console.log(`Saving user ${this.name} to the database...`);
  }


  sendWelcomeEmail() {
    // Logic to send a welcome email to the user
    console.log(`Sending welcome email to ${this.email}...`);
  }
}


const user = new User('Alice', 'alice@example.com');
user.saveToDatabase();
user.sendWelcomeEmail();
```

In this refactored example:

The **`User`** class is responsible only for representing user data.

The **`UserRepository`** class is responsible only for persisting user data to the database.
The **`EmailService`** class is responsible only for sending welcome emails.

Each class now has a single reason to change, adhering to the Single Responsibility Principle. If there are any changes in how user data

O

Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In Node.js, this can be achieved through the use of interfaces and dependency injection. Rather than modifying existing code, you should extend it through inheritance or composition.

Let's discuss this principle with two examples in a Node.js context.

```
class AuthenticationProcessor {
  constructor(authenticationMethod) {
    this.authenticationMethod = authenticationMethod;
  }


  authenticateUser(credentials) {
    return this.authenticationMethod.authenticate(credentials);
  }
}
----------------------------
class UsernamePasswordAuthentication {
  authenticate(credentials) {
    // Logic to authenticate using username/password
  }
}


class OAuthAuthentication {
  authenticate(credentials) {
    // Logic to authenticate using OAuth
  }
}


class JWTAuthentication {
  authenticate(credentials) {
    // Logic to authenticate using JWT
  }
}
```

XXXX
Should avoid below logic for coding

```
class AuthenticationService {
  authenticateUser(credentials) {
    // Logic to authenticate user based on authentication method
    if (credentials.method === 'username_password') {
      // Authenticate using username/password
    } else if (credentials.method === 'oauth') {
      // Authenticate using OAuth
    } else if (credentials.method === 'jwt') {
      // Authenticate using JWT
    }
  }
}
```

Now, if we want to add support for SAML authentication, we simply create a new class implementing the authentication method interface without modifying existing code.

```
class SAMLAuthentication {
  authenticate(credentials) {
    // Logic to authenticate using SAML
  }
}
```

In this example, we extended the behavior of our authentication microservice by adding a new authentication method without modifying existing code, thus adhering to the Open-Closed Principle. Each authentication method class is responsible for its specific authentication logic, promoting high cohesion and low coupling within the system.

L

The Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In other words, subclasses should be substitutable for their base classes without altering the desirable properties of the program.

```
class File {
  constructor(name) {
    this.name = name;
  }

  getSize() {
    throw new Error('This method should be implemented by subclasses');
  }
}

class TextFile extends File {
  constructor(name, content) {
    super(name);
    this.content = content;
  }

  getSize() {
    return this.content.length;
  }
}

class ImageFile extends File {
  constructor(name, size) {
    super(name);
    this.size = size;
  }

  getSize() {
    return this.size;
  }
}
```

```
function printFileSize(file) {
  console.log(`${file.name} size: ${file.getSize()} bytes`);
}

const textFile = new TextFile('document.txt', 'Hello, world!');
const imageFile = new ImageFile('image.jpg', 2048);

printFileSize(textFile); // Output: document.txt size: 13 bytes
printFileSize(imageFile); // Output: image.jpg size: 2048 bytes
```

subclasses `TextFile`, and `ImageFile` can be substituted for their base classes `Shape` and `File`, respectively, without altering the correctness of the program. This demonstrates the Liskov Substitution Principle in action.

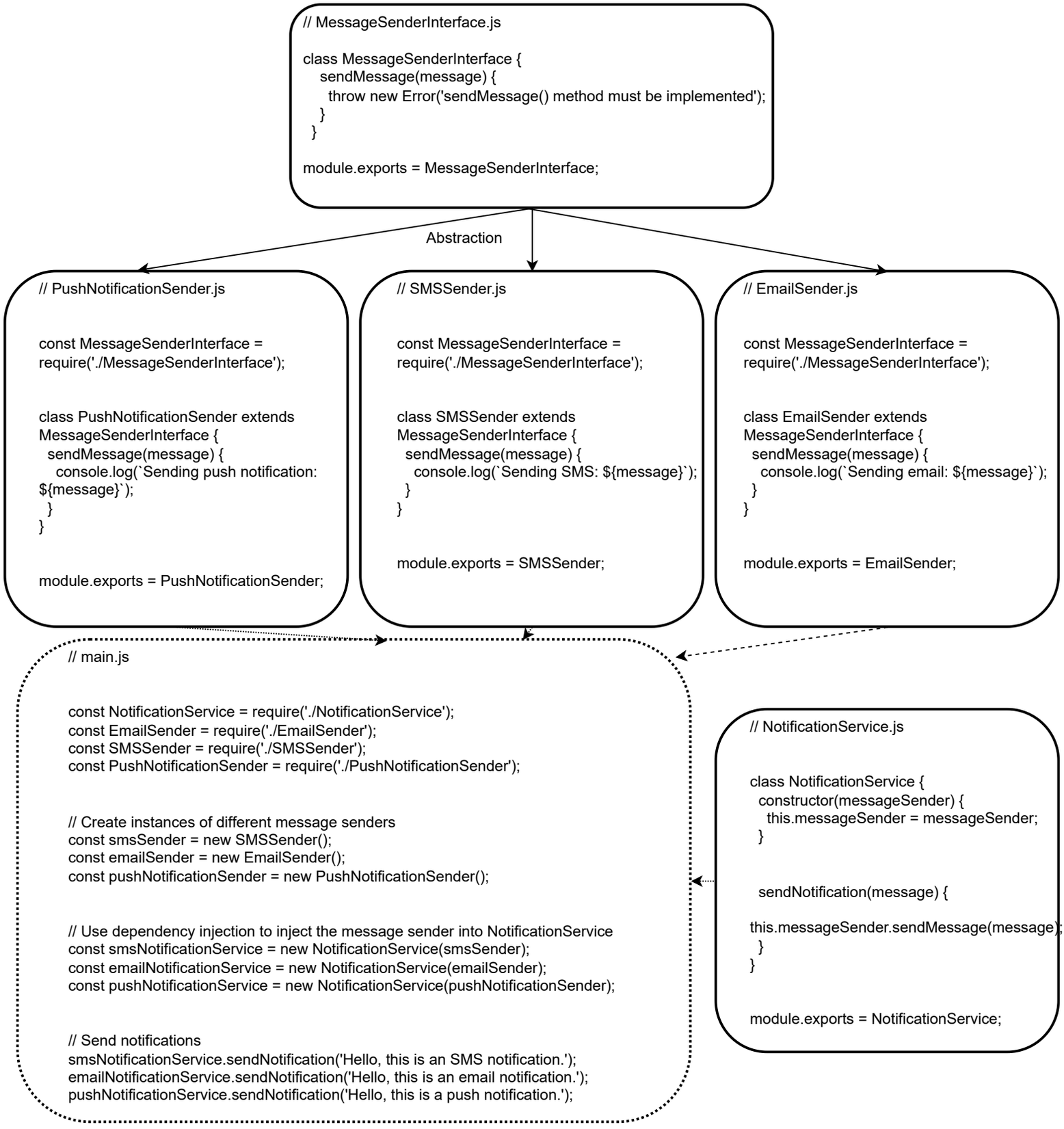
I

Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. In Node.js, this means that modules should expose only the methods or properties that are relevant to their clients. This helps to keep interfaces focused and prevents clients from being burdened with unnecessary dependencies.

```
// Interface segregation


playAudio(): void;
  recordAudio(): void;
}


interface VideoPlayer {
  playVideo(): void;
}



class AudioPlayer implements AudioPlayer {
  playAudio() {
    // Implementation for playing audio
  }


  recordAudio() {
    // Implementation for recording audio
  }
}


class VideoPlayer implements VideoPlayer {
  playVideo() {
    // Implementation for playing video
  }
}
```

```
XXXXXXXXXXX AVOID XXXXXXXX


interface MediaPlayer {
  playAudio(): void;
  recordAudio(): void;
}




class AudioPlayer implements MediaPlayer {
  playAudio() {
    // Implementation for playing audio
  }


  recordAudio() {
    // Implementation for recording audio
  }
}



class VideoPlayer implements MediaPlayer {
  playAudio() {
    // Implementation for playing audio of the video
  }


  recordAudio() {
    // This method is irrelevant for video playback
  }
}
```

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions. In Node.js, this can be achieved by using dependency injection, where dependencies are passed to a module rather than being created or instantiated within it. This allows for more flexible and testable code



Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions. In Node.js, this can be achieved by using dependency injection, where dependencies are passed to a module rather than being created or instantiated within it. This allows for more flexible and testable code.

In this example:

NotificationService.js represents a high-level module responsible for sending notifications.

MessageSenderInterface.js defines an interface for message senders.

EmailSender.js, SMSSender.js, and PushNotificationSender.js implement the MessageSenderInterface for different messaging channels.

In main.js, we create instances of different message senders and inject them into the NotificationService, demonstrating how we can easily switch between messaging channels without modifying the NotificationService code.

By following the Dependency Inversion Principle, we've ensured that the NotificationService module is not tightly coupled to specific implementations of message senders, promoting flexibility and ease of maintenance in our messaging system.