Notes - JavaScript - Design Principles - Foundational - Loose Coupling

Dr Nick Hayward

A brief introduction to loose coupling in JavaScript/TypeScript and how to apply it with dependency injection, events, and composition.

Contents

Intro

Loose coupling reduces interdependencies between parts of a system. When components know less about each other, you can change one without breaking the rest. In JS ecosystems (Node.js, browsers, React/React Native), loose coupling is often achieved with dependency injection, events, and programming to small contracts.

Related principles:

- encapsulate what varies
- favor composition over inheritance
- program to interfaces (contracts), not implementations

Loose coupling techniques (JavaScript)

1) Dependency Injection (DI)

- Pass collaborators in via constructor/factory parameters. Callers choose implementations; callees depend on contracts.

2) Observer / Pub-Sub

- Emit events and let listeners react. Publishers don't need to know who is listening.

3) Ports and Adapters (Hexagonal)

- Define a stable boundary (port) that your core code depends on; provide adapters for concrete technologies (HTTP, DB, etc.).

4) Module boundaries

- Export small public APIs. Keep internals private to avoid tight coupling.

Example — basic message service with dependency injection

```
// MessageService depends on a small contract: sender.send(message)
class MessageService {
    constructor(sender) {
        this.sender = sender;
    }
    sendMessage(message) {
        this.sender.send(message);
    }
```

```
}

class EmailSender {
    send(message) {
        console.log(`Sending email: ${message}`);
    }
}

class SmsSender {
    send(message) {
        console.log(`Sending SMS: ${message}`);
    }
}

// usage: swap implementations without changing MessageService
const emailSvc = new MessageService(new EmailSender());
emailSvc.sendMessage("Welcome to the email service...");

const smsSvc = new MessageService(new SmsSender());
smsSvc.sendMessage("Hello from SMS...");
```

Why it's loosely coupled: `MessageService` only depends on the `send(message)` contract, not on a specific sender class.

Example — observer/pub-sub with EventEmitter (Node.js)

```
// Node: replace with a minimal EventEmitter in the browser or use a library
const { EventEmitter } = require('node:events');

class NotificationBus extends EventEmitter {}

const bus = new NotificationBus();

// publishers
function onOrderCreated(order) {
    bus.emit('order:created', order);
}

// independent subscribers
bus.on('order:created', (order) => {
    console.log('Send email for order', order.id);
});
bus.on('order:created', (order) => {
    console.log('Update analytics for order', order.id);
});

// usage
onOrderCreated({ id: 'A123', total: 42 });
```

Why it's loosely coupled: the publisher doesn't know about subscribers. New listeners can be added or removed without touching the publisher.

Example — ports and adapters (boundary abstraction)

```
// Port (contract)
/** @typedef {{ save: (record: any) => Promise<void> }} Repository */

// Core service depends on the port, not a specific DB
class UserService {
    /** @param {Repository} repo */
    constructor(repo) { this.repo = repo; }
    async register(user) {
        // ... domain logic, validation, etc.
        await this.repo.save(user);
    }
}

// Adapter 1: in-memory (tests/dev)
class InMemoryRepo {
    constructor() { this.items = []; }
    async save(record) { this.items.push(record); }
}

// Adapter 2: SQL (prod) - example shape
class SqlRepo {
    constructor(db) { this.db = db; }
    async save(record) { await this.db.query('INSERT ...', [record.id, record.name]); }
}

// usage
const serviceDev = new UserService(new InMemoryRepo());
serviceDev.register({ id: 'u1', name: 'Ada' });

// const db = /* create db client */
// const serviceProd = new UserService(new SqlRepo(db));
// await serviceProd.register({ id: 'u2', name: 'Linus' });
```

Why it's loosely coupled: `UserService` depends on the repository contract. Technology choices live in adapters that can change independently.

Edge cases and trade-offs

When tighter coupling may be acceptable:

- small scripts or performance-critical hot paths where indirection is measurable
- stable dependencies that rarely change

Pitfalls to avoid:

- over-abstracting too early: don't invent ports until variation is real
- hidden coupling via shared mutable state or global singletons
- event spaghetti: untracked listeners can make flow hard to follow (name events clearly and document)

Testing benefits:

- DI enables easy mocking/stubbing of collaborators
- pub-sub can be tested by asserting emissions and handler effects
- ports/adapters let you run core logic with fast in-memory fakes

Try it

You can paste the above examples into a Node.js REPL or a `*.mjs` file and run it with Node 18+.