High-level Architectural Principles

High-level architectural principles provide the foundation for designing robust, scalable, and maintainable systems. They serve as guiding rules that shape decision-making, ensuring consistency, performance, and long-term adaptability of the architecture..

Modularity

Modularity is a cornerstone of high-level software architecture. It’s the idea of breaking down a system into smaller, independent, and interchangeable parts (modules). Each module focuses on one well-defined functionality, making the system easier to understand, maintain, and evolve..

Definition:
A modular system is composed of self-contained units (modules), each responsible for a distinct feature or concern, which can be developed, tested, and deployed independently.

Why it matters:

- **Scalability:** You can add or replace modules without touching the whole system.
- **Maintainability:** Bugs are easier to locate and fix when logic is separated.
- **Reusability:** Modules can often be reused across different projects.
- **Team efficiency:** Different teams can work on different modules in parallel.

Bad Example (violating modularity):
Here, everything — authentication, payments, and notifications — is crammed into one giant module:

```
const App = {
  login(user, password) {
    console.log("Authenticating user...");
  },
  processPayment(amount) {
    console.log("Processing payment...");
  },
  sendNotification(message) {
    console.log("Sending notification...");
  }
};
```

Problem: If you need to change the payment logic, you risk breaking authentication or notification logic because everything is tangled together.

Good Example (following modularity):

We separate responsibilities into distinct modules:

```
// Auth module
function login(user, password) {
  console.log("Authenticating user...");
}
// Payment module
function processPayment(amount) {
  console.log("Processing payment...");
}
// Notification module
function sendNotification(message) {
  console.log("Sending notification...");
}
// Composing them together
const App = {
  login,
  processPayment,
  sendNotification
};
```

Now: Each function/module has its own clear responsibility. Changing `processPayment` won't affect `login` or `sendNotification`, and new modules can be added easily.

Illustration:

```
✕ One Big Module
+--------------------------------+
|             App               |
| login() processPayment() send()|
+--------------------------------+

✔ Independent Functions
+----------+   +--------------+   +-----------------+
|  login   |   | processPayment|  | sendNotification|
+----------+   +--------------+   +-----------------+
        \             |             /
         \            |            /
              +-----------------+
              |       App       |
              +-----------------+
```

Encapsulation

Encapsulation is the practice of **hiding internal details** of a module or function, exposing only what is necessary. It protects the system from unintended interference and simplifies usage.

Definition:
A module should **control access to its internal state and logic**, exposing only public interfaces for other parts of the system to interact with.

Why it matters:

- **Prevents unintended modifications:** Internal data can't be changed accidentally.
- **Improves maintainability:** Changes inside the module don't affect external code.
- **Simplifies usage:** Users only interact with a clear, limited interface.
- **Enhances testability:** Easier to mock or replace modules.

Bad Example (violating encapsulation):
Everything is exposed globally:

```
const AuthModule = {
  loggedIn: false, // direct access from outside

  login(user, password) {
    console.log("Authenticating user...");
    this.loggedIn = true;
  },

  logout() {
    this.loggedIn = false;
  }
};

// Somewhere else in the code
AuthModule.loggedIn = true; // breaks logic
```

Problem: External code can directly modify `loggedIn`, bypassing login logic and breaking the system.

Good Example (following encapsulation):
We hide the internal state using closures:

```
function createAuthModule() {
  let loggedIn = false; // private variable
```

```
    function login(user, password) {
      console.log("Authenticating user...");
      loggedIn = true;
    }

    function logout() {
      loggedIn = false;
    }

    function isLoggedIn() {
      return loggedIn;
    }

    return { login, logout, isLoggedIn };
  }

  // Usage
  const Auth = createAuthModule();
  Auth.login("user1", "pass123");
  console.log(Auth.isLoggedIn()); // true
  // Auth.loggedIn = true; // ✖ Not allowed
```

Now: The internal state `loggedIn` is private. External code can't change it directly.

Illustration:

```
✖ Everything Exposed
 +-------------------+
 |  AuthModule       |
 | loggedIn          |
 | login()           |
 | logout()          |
 +-------------------+
 External code can modify loggedIn directly
 +-------------------+
 | loggedIn = true   |
 +-------------------+

✔ Encapsulated Module
 +-------------------+
 |  AuthModule       |
 | [loggedIn]        | <- hidden
 | login()           |
 | logout()          |
 | isLoggedIn()      |
 +-------------------+
 External code can only interact via functions
```

Coupling and Cohesion

Coupling and cohesion are complementary principles that describe **how modules relate to each other** and **how well their internal parts work together**. High cohesion and low coupling lead to maintainable and flexible systems.

Definition

- **Cohesion:** Measures how closely related the responsibilities of a single module are. High cohesion means a module has a **clear, focused purpose**.
- **Coupling:** Measures how dependent a module is on other modules. Low coupling means a module can **function independently**.

Why it matters

- **Easier maintenance:** Changes in one module are less likely to break others.
- **Improved readability:** Modules have a single, clear responsibility.
- **Better reusability:** Independent, focused modules can be reused in other projects.
- **Team efficiency:** Teams can work on separate modules with minimal conflicts.

Bad Example (tight coupling, low cohesion)

```
const App = {
  login(user, password) {
    console.log("Authenticating user...");
    this.processPayment(100); // directly calls unrelated module
  },

  processPayment(amount) {
    console.log("Processing payment...");
    this.sendNotification("Payment done"); // also calls notification
  },

  sendNotification(message) {
    console.log("Sending notification:", message);
  }
};
```

Problem:

- `login` is indirectly calling payment and notification.
- Modules are **tightly coupled** and responsibilities are **scattered**, reducing cohesion.

Good Example (low coupling, high cohesion)

```
function createAuthModule() {
  function login(user, password) {
    console.log("Authenticating user...");
    return true;
  }
  return { login };
}

function createPaymentModule() {
  function processPayment(amount) {
    console.log("Processing payment...");
    return true;
  }
  return { processPayment };
}

function createNotificationModule() {
  function sendNotification(message) {
    console.log("Sending notification:", message);
  }
  return { sendNotification };
}

// Composing App
const Auth = createAuthModule();
const Payment = createPaymentModule();
const Notify = createNotificationModule();

// Usage
if (Auth.login("user1", "pass123")) {
  Payment.processPayment(100);
  Notify.sendNotification("Payment done");
}
```

Now:

- Each module has **one focused responsibility** (high cohesion).
- Modules communicate **via public interfaces only** (low coupling).
- Easier to replace or extend any module independently.

Illustration

```
✗ Tightly Coupled & Low Cohesion
 +-------------------------------+
 |           App                 |
 | login() -> processPayment() ->|
 | sendNotification()            |
 +-------------------------------+

✓ Loosely Coupled & High Cohesion
 +---------+   +------------+   +-----------------+
 |  Auth   |   |  Payment   |   |  Notification   |
 +---------+   +------------+   +-----------------+
        \          |               /
         \         |              /
          +----------------------+
                  App
```

Design by Contract (DbC)

Design by Contract is a principle where **modules communicate through well-defined contracts**, specifying **preconditions, postconditions, and invariants**. It ensures that each module behaves predictably and errors are detected early.

Definition

A contract defines the rules a module **must satisfy**:

- **Preconditions:** What must be true before calling a function.
- **Postconditions:** What will be true after the function executes.
- **Invariants:** Conditions that remain true during the module's lifetime.

Why it matters

- **Early bug detection:** Violations of the contract reveal errors immediately.
- **Improved reliability:** Each module clearly defines expectations.
- **Clear documentation:** Contracts serve as formal, enforceable specifications.
- **Safer integration:** Modules can interact with confidence, knowing contracts are respected.

Bad Example (no contract)

```
function divide(a, b) {
  return a / b;
}

console.log(divide(10, 0)); // ✗ returns Infinity, unexpected
```

Problem:

- No contract: function assumes b is non-zero, but nothing enforces it.
- Leads to unexpected behavior and potential bugs.

Good Example (with contract checks)

```
function divide(a, b) {
  if (b === 0) throw new Error("Precondition failed: divisor must not be zero");
  const result = a / b;
  if (!isFinite(result)) throw new Error("Postcondition failed: result must
     be finite");
  return result;
}

console.log(divide(10, 2)); // ✓ works correctly
// divide(10, 0); // ✗ throws error
```

Now:

- Preconditions prevent invalid inputs.
- Postconditions ensure expected outputs.
- The contract is **explicit**, making the function predictable and safe.

Illustration

```
✗ No Contract
+-----------------+
| divide(a, b)    |
|                 |
+-----------------+
b = 0 -> unexpected Infinity

✓ With Contract
+-----------------+
| divide(a, b)    |
| Preconditions:  |
|  b != 0         |
| Postconditions: |
|  result finite  |
+-----------------+
 Violations throw errors immediately.
```

Event-Driven Design

Event-Driven Design organizes a system around **events** and **listeners**. Modules emit events when something happens, and other modules respond without tight coupling. This creates a **flexible, reactive architecture**.

Definition

A module (or component) should **emit events** for significant changes and allow other modules to **subscribe** to these events. This decouples modules and promotes responsiveness.

Why it matters

- **Low coupling:** Modules don't need direct references to each other.
- **High flexibility:** New event handlers can be added without changing existing code.
- **Reactive behavior:** System responds automatically to changes or user actions.
- **Better scalability:** Easy to extend features by adding new event listeners.

Bad Example (tight coupling)

```
const App = {
  login(user) {
    console.log("Authenticating user...");
    // Directly calls other modules
    Payment.processPayment(100);
    Notify.sendNotification("Payment done");
  }
};

const Payment = {
  processPayment(amount) {
    console.log("Processing payment...");
  }
};

const Notify = {
  sendNotification(msg) {
    console.log("Sending notification:", msg);
  }
};
```

Problem:

- `login` directly calls `Payment` and `Notify`.
- Modules are tightly coupled and cannot operate independently.

Good Example (event-driven)

```
const EventBus = {
  events: {},

  subscribe(event, listener) {
    if (!this.events[event]) this.events[event] = [];
    this.events[event].push(listener);
  },

  emit(event, data) {
    if (this.events[event]) {
      this.events[event].forEach(listener => listener(data));
    }
  }
};

// Modules
const Auth = {
  login(user) {
    console.log("Authenticating user...");
    EventBus.emit("loginSuccess", { user });
  }
};

const Payment = {
  init() {
    EventBus.subscribe("loginSuccess", ({ user }) => {
      console.log(`Processing payment for ${user}`);
    });
  }
};

const Notify = {
  init() {
    EventBus.subscribe("loginSuccess", ({ user }) => {
      console.log(`Sending notification for ${user}`);
    });
  }
};

// Initialize event subscriptions
Payment.init();
Notify.init();

// Usage
Auth.login("user1");
```

Now:

- Auth emits a `loginSuccess` event.
- `Payment` and `Notify` respond independently.
- Modules remain **decoupled** and easy to extend.

Illustration

```
✖ Tightly Coupled
+---------+
|  Auth   |
| login() |-----> Payment.processPayment()
|         |-----> Notify.sendNotification()
+---------+

✔ Event-Driven
+---------+       +-------------+
|  Auth   |       | EventBus    |
| login() |------>| "loginSuccess" |
+---------+       +-------------+
                      /       \
                     v         v
               +--------+  +---------+
               | Payment|  | Notify  |
               +--------+  +---------+
```