Behavioral and Team-Oriented Principles

Behavioral and Team-Oriented Principles focus on how software is developed in practice, not just how it's structured. They emphasize collaboration, communication, testing practices, and conventions that help teams build reliable, understandable, and maintainable systems together.

Convention Over Configuration

"Convention over configuration" means preferring sensible defaults over requiring developers to write extensive configuration for common tasks. By following conventions, teams can focus on the unique aspects of their applications instead of boilerplate setup.

Definition

Convention over configuration is a design principle where a system makes assumptions about defaults, requiring configuration only when behavior deviates from those defaults.

Why it matters

- **Less boilerplate**: Reduces repetitive setup code.
- **Consistency**: Teams follow the same patterns without extra discussion.
- **Faster onboarding**: New developers can be productive quickly.
- **Focus on essentials**: Developers spend time on unique business logic.

Bad Example (too much configuration)

Here, every function requires explicit configuration, even when defaults would work:

```
function connectToDatabase(config) {
  const host = config.host || "localhost";
  const port = config.port || 5432;
  const user = config.user || "admin";
  const password = config.password || "password";

  console.log(`Connected to DB at ${host}:${port} as ${user}`);
}

// Usage (developer must configure everything, even defaults)
connectToDatabase({
  host: "localhost",
  port: 5432,
  user: "admin",
  password: "password"
});
```

Problem: Developer has to specify values that are already the defaults. More boilerplate, more room for mistakes.

Good Example (conventions first)

Here, defaults are applied automatically, and configuration is only needed if we want to override them:

```
function connectToDatabase(config = {}) {
  const defaults = {
    host: "localhost",
    port: 5432,
    user: "admin",
    password: "password"
  };

  const finalConfig = { ...defaults, ...config };

  console.log(
    `Connected to DB at ${finalConfig.host}:${finalConfig.port}
    as ${finalConfig.user}`
  );
}

// Usage (use defaults)
connectToDatabase();

// Usage (override only what's necessary)
connectToDatabase({ user: "alice", password: "securePass" });
```

Now:

- Defaults handle the common case.
- Developers configure only when they need non-standard behavior.
- Code is shorter, clearer, and more maintainable.

Illustration

```
❌ Configuration Overload
+------------------------------+
| Must define EVERY setting    |
| even if 90% are defaults     |
+------------------------------+

✔ Convention Over Configuration
+------------------------------+
```

```
| Sensible defaults in place    |
| Only override when needed     |
+-------------------------------+
```

You Can’t Fix What You Don’t Measure

In software engineering, assumptions about performance or reliability often turn out to be wrong. Without measuring, you don’t know where the real problems are — and trying to “fix” blindly can waste time or even make things worse.

Definition
“You can’t fix what you don’t measure” means you must collect data (metrics, logs, benchmarks) before attempting to improve performance, reliability, or user experience.

Why it matters

- **Evidence-based decisions**: Changes are guided by data, not guesswork.
- **Accurate bottleneck detection**: Focus effort where it matters most.
- **Prevents wasted effort**: Avoids optimizing parts of the system that don’t need it.
- **Builds accountability**: Teams can prove improvements through measurable results.

Bad Example (guessing without measurement)
A developer receives complaints that “the app loads the user list slowly” and immediately changes the filtering function in code, assuming it’s the problem::

```
function filterUsers(users) {
  // Developer assumes this loop is slow
  let result = [];
  for (let i = 0; i < users.length; i++) {
    if (users[i].active) result.push(users[i]);
  }
  return result;
}
```

Problem:

- The developer optimizes `filterUsers` without knowing if it’s the real bottleneck.
- Maybe the actual problem is the **database query** fetching users.
- Time and effort are wasted, and the code becomes unnecessarily complex.

.

Good Example (measure, then fix)

The developer first measures API response time before touching the filtering logic:

```
async function fetchUsers() {
  console.time("fetchUsers"); // Measure total API call
  const users = await getUsersFromDB(); // Could be the slow part
  console.timeEnd("fetchUsers");

  const activeUsers = users.filter(u => u.active); // Filter in memory
  only if needed
  return activeUsers;
}

// Usage
fetchUsers();
```

Now:

- The developer sees that fetching from the database is the slow part, not filtering.
- Optimization efforts focus on **DB query or indexing**, not the loop in memory.
- Time is saved, and changes are targeted and meaningful.

Illustration

```
✕ Guessing without measurement
+----------------------------------------+
| "Filter must be slow"                  |
| Optimizing loop blindly → wasted effort |
+----------------------------------------+

✔ Measure first
+----------------------------------------+
| Measure API call / DB query            |
| Focus optimization on actual bottleneck |
+----------------------------------------+
```

Least Astonishment Principle (LAP)

The Least Astonishment Principle encourages designing software so that its behavior is intuitive and matches the expectations of its users or other developers. When code behaves in an unexpected way, it can lead to bugs, confusion, and wasted time.

Definition
Design software so that the behavior is obvious and predictable. If someone reads your code or uses your API, the results should align with common sense and expectations.

Why it matters

- **Reduces bugs**: Surprising behavior often leads to errors.
- **Improves maintainability**: Code is easier for other developers to understand.
- **Better user experience**: Users interact with predictable interfaces.
- **Encourages consistency**: Aligns with established conventions and patterns.

Bad Example (violating LAP)
A function for removing a user **silently deletes their account**, instead of returning a confirmation or throwing an error:

```
function removeUser(users, id) {
  // Deletes user silently
  const index = users.findIndex(u => u.id === id);
  users.splice(index, 1);
  // No feedback, no warning
  return users;
}

// Usage
const userList = [{id: 1, name: "Alice"}, {id: 2, name: "Bob"}];
removeUser(userList, 3); // Oops, user 3 doesn’t exist, but no warning is given
```

Problem:

- The developer or user is surprised when nothing happens or data disappears silently.
- Hard to debug issues caused by unexpected behavior.

Good Example (following LAP)
Provide clear feedback and predictable behavior:

```
function removeUser(users, id) {
  const index = users.findIndex(u => u.id === id);
  if (index === -1) {
    throw new Error(`User with id ${id} not found`);
  }
  users.splice(index, 1);
  return users;
}

// Usage
```

```
try {
  const userList = [{id: 1, name: "Alice"}, {id: 2, name: "Bob"}];
  removeUser(userList, 3);
} catch (err) {
  console.log(err.message); // User with id 3 not found
}
```

Now:

- Behavior matches expectations: you can't remove a non-existing user silently.
- Easier debugging and safer code.
- API is predictable and consistent for all developers.

Illustration

```
✕ Surprising / Astonishing behavior
+---------------------------+
| removeUser silently fails  |
| No error, no feedback      |
+---------------------------+

✔ Predictable / Intuitive behavior
+---------------------------+
| removeUser checks existence|
| Throws error if not found  |
| Feedback is clear          |
+---------------------------+
```

Test-Driven Development (TDD)

Test-Driven Development is a software development approach where tests are written **before** the actual code. The idea is to define the expected behavior first, then implement code to pass those tests. This ensures correctness and guides design.

Definition

Write automated tests **before** writing the production code. Development proceeds in short cycles:

1. Write a failing test.
2. Write code to make the test pass.
3. Refactor if needed.

Why it matters

- **Ensures correctness**: Code is validated as it's written.
- **Guides design**: Writing tests first helps you think about interfaces and responsibilities.
- **Reduces bugs**: Early detection of problems.
- **Facilitates refactoring**: Tests act as safety nets.

Bad Example (no TDD)

Code is written first without tests; issues may appear later:

```
// Production code
function addUser(users, user) {
  users.push(user);
  // Forgot to check for duplicate IDs
  return users;
}

const userList = [{id: 1, name: "Alice"}];
addUser(userList, {id: 1, name: "Bob"});
console.log(userList);
// [{id: 1, name: "Alice"}, {id: 1, name: "Bob"}] -> duplicate ID inserted
```

Problem:

- Bugs like duplicate entries go unnoticed until later.
- Harder to track down problems in larger systems.

Good Example (following TDD)

Write tests first, then implement the functionality:

```
// Test first
function testAddUser() {
  const users = [{id: 1, name: "Alice"}];
  const newUsers = addUser(users, {id: 2, name: "Bob"});
  if (newUsers.length !== 2) throw new Error("User not added");
  console.log("Test passed!");
}

// Production code to pass test
function addUser(users, user) {
  const exists = users.some(u => u.id === user.id);
  if (exists) throw new Error(`User with id ${user.id} already exists`);
  return [...users, user];
}

// Run test
testAddUser();
```

Now:

- Duplicate IDs are prevented automatically.
- You know immediately when a new change breaks functionality.
- Encourages writing modular, testable code.

Illustration

```
❌ No TDD
+----------------------------+
| addUser blindly            |
| Duplicate IDs possible     |
| Bugs found later           |
+----------------------------+

✔ TDD
+----------------------------+
| Write test first           |
| Implement code to pass test |
| Refactor safely            |
| Bugs caught immediately    |
+----------------------------+
```

Behavior-Driven Development (BDD)

Behavior-Driven Development is an evolution of TDD that emphasizes **specifying the behavior of software in a readable, human-friendly way**. It focuses on collaboration between developers, testers, and non-technical stakeholders to ensure the software does what users expect.

Definition

Write specifications (tests) that describe **expected behavior** in plain language. Code is then implemented to satisfy these behavior specifications.

Why it matters

- **Improves communication**: Everyone understands what the system should do.
- **Ensures correct behavior**: Tests are expressed as user scenarios.
- **Facilitates acceptance testing**: Non-technical stakeholders can validate behavior.
- **Encourages better design**: Thinking in terms of behavior often produces cleaner, modular code.

Bad Example (no BDD)

Code is written without considering expected behavior from the user's perspective:

```
// Function without clear behavior specification
function calculateDiscount(user) {
  if (user.membership) return 0.1;
  if (user.vip) return 0.2;
  return 0;
}

console.log(calculateDiscount({membership: true, vip: true})); // 0.1 -> unexpected
```

Problem:

- Ambiguous rules: Which discount applies?
- Confusing results for users.
- Harder to communicate intent to stakeholders.

Good Example (using BDD approach)

Write behavior specifications first in human-readable form, then implement code:

```
// Behavior specification
// "VIP users get 20%, members get 10%, VIP+Member gets 20%"

// Implement code to satisfy behavior
function calculateDiscount(user) {
  if (user.vip) return 0.2;
  if (user.membership) return 0.1;
  return 0;
}

// Test scenarios
function testDiscount() {
  const tests = [
    {input: {vip: true, membership: true}, expected: 0.2},
    {input: {vip: false, membership: true}, expected: 0.1},
    {input: {vip: false, membership: false}, expected: 0}
  ];

  tests.forEach(({input, expected}) => {
    const result = calculateDiscount(input);
    if (result !== expected) throw new Error(`Failed: ${JSON.stringify(input)}`);
  });
  console.log("All BDD tests passed!");
}
```

```
// Run tests
testDiscount();
```

Now:

- Rules are clear and behavior-driven.
- Everyone (developers, testers, stakeholders) can understand what is expected.
- Reduces ambiguity and unexpected results.
- Encourages modular and maintainable code.

Illustration

```
❌ No BDD
+--------------------------+
| calculateDiscount()      |
| Rules unclear, results   |
| may confuse users        |
+--------------------------+

✔ BDD
+--------------------------+
| Define behavior first    |
| Implement code to meet   |
| behavior                 |
| Tests verify expectations|
+--------------------------+
```