Coupling Clarity:

Using Connascence to Write Maintainable Code

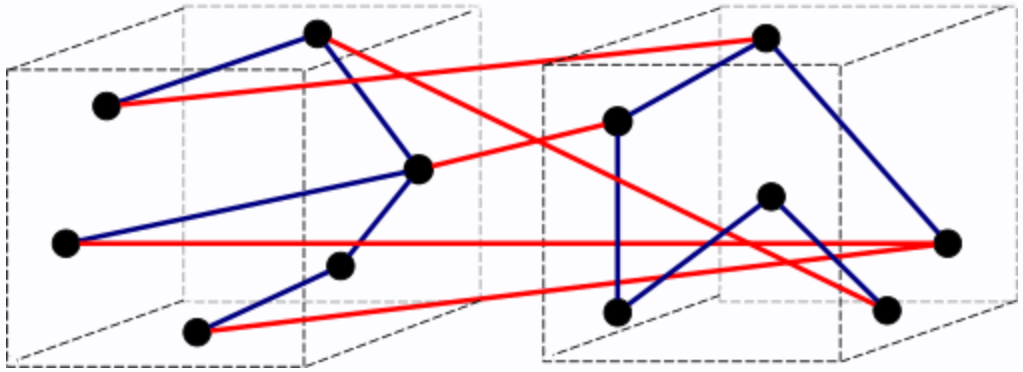
by Tom Wetjens

The Challenge of Software Design

Writing maintainable, scalable software requires understanding the relationships between components.

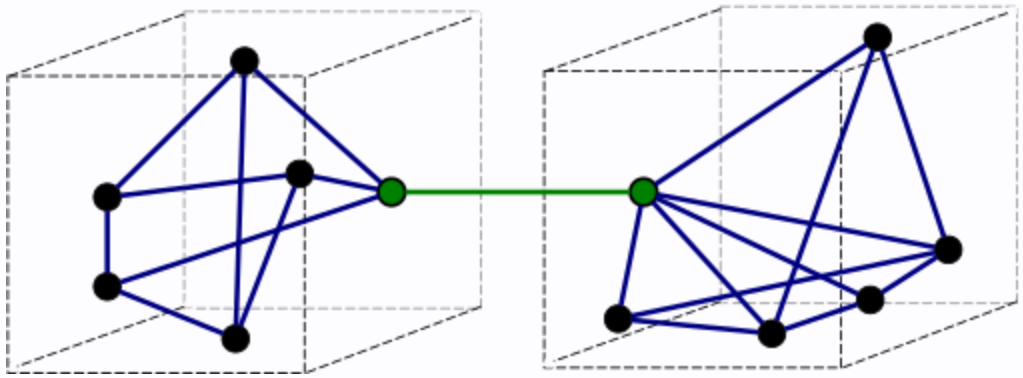
Low Coupling, High Cohesion

The golden rule of software architecture



Problematic: High coupling, low cohesion

- Components tightly bound together
- Changes ripple throughout the system
- Difficult to test and maintain



Ideal: Low coupling, high cohesion

- Components are loosely connected
- Changes are localized
- Easy to test and maintain

What is Connascence?

Connascence quantifies the **degree** and **type** of dependency between software components.

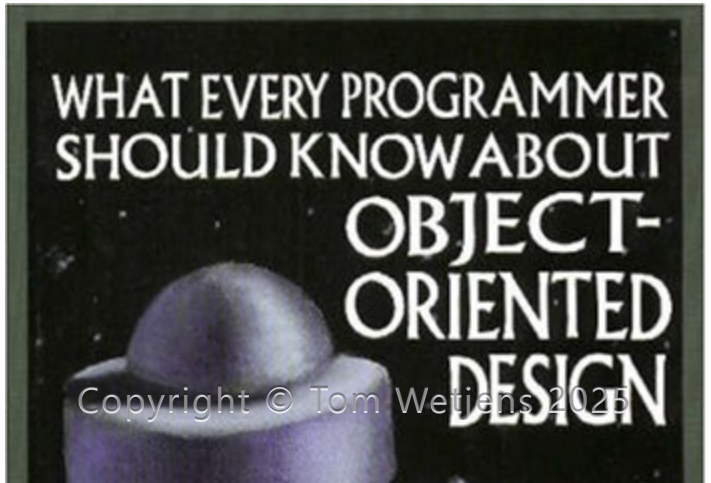
It evaluates their **strength** (difficulty of change) and **locality** (proximity in the codebase).

Historical Context

The term "connascence" was first introduced by **Meilir Page-Jones** in:

Comparing techniques by means of encapsulation and connascence (1992)

and later expanded in:



Classification Framework

Connascence types can be categorized as:

Static: Analyzable at compile-time
Dynamic: Only detectable at runtime

Connascence Types Overview

Static	Dynamic
Name	Execution
Type	Timing
Meaning	Value
Position	Identity
Algorithm	

Static Connascences

Detectable at compile-time

```
class Programmer {
    void work() {}
}
```

```
void main() {
  new Programmer().work();
}
```

Connascence of Name

Multiple components must agree on the name of something (e.g. type, method, parameter, field)

```
class Coffee {
    void brew() {}
}
```

```
class Programmer {
    void drink(Coffee coffee) {
        coffee.brew();
    }
}
```

Connascence of Type

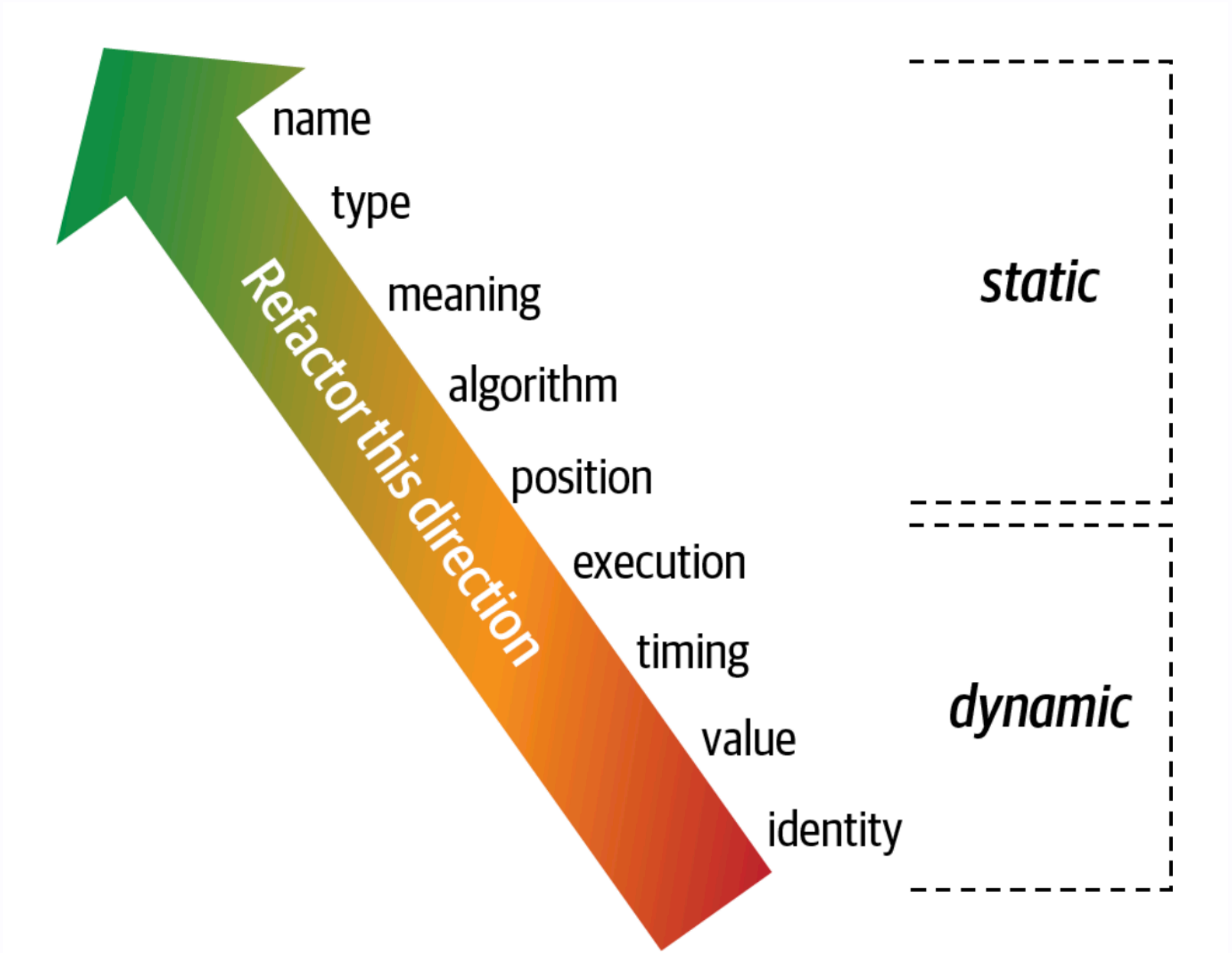
Multiple components must agree on the type of something (e.g. parameter, return value, field)

```
class Programmer {
    void drink(Coffee coffee) {
        coffee.brew();
    }

    void drink(Tea tea) {
        tea.brew();
    }
}
```

```
interface Brewable {
    void brew() {}
}
class Coffee implements Brewable {}
class Tea implements Brewable {}
```

```
class Programmer {
    void drink(Brewable brewable) {
        brewable.brew();
    }
}
```



```
class Coffee {
    int getStrength() {
        return 3;
    }
}
```

```
class Programmer {
    void drink(Coffee coffee) {
        if (coffee.getStrength() > 2)
            throw new IllegalArgumentException("too strong");
    }
}
```

Connascence of Meaning

Multiple components must agree on the meaning of particular values.

Refactoring *Connascence of Meaning* to *Connascence of Name*

```
class Coffee {
    static final int WEAK = 1, MEDIUM = 2, STRONG = 3;

    int getStrength() {
        return STRONG;
    }
}
```

```
class Programmer {
    void drink(Coffee coffee) {
        if (coffee.getStrength() > Coffee.MEDIUM)
            throw new IllegalArgumentException("too strong");
    }
}
```

Another example:

```
class CreditCardNumber {
    boolean isValid(String str) {
        if (str.equals("9999-9999-9999-9999")) {
            // Test credit card number
            return true;
        }
        /* else normal validation */
    }
}
```

```
class CreditCardNumber {

    static final String TEST_VALUE = "9999-9999-9999-9999";

    boolean isValid(String str) {
        if (str.equals(TEST_VALUE)) {
            // Test credit card number
            return true;
        }
        /* else normal validation */
    }
}
```

Yet another example:

```
class Order {
    enum Status { PLACED, PAID, FULFILLED }
    Status getStatus() { /*...*/ }
}
```

```
class Client {
    void checkOrder(Order order) {
        if (order.getStatus() != FULFILLED) {
            System.out.println("order not completed yet");
        }
    }
}
```

Let's add an enum value:

```
class Order {
    enum Status { PLACED, PAID, FULFILLED, ARCHIVED }
    Status getStatus() { /*...*/ }
}
```

```
class Client {
    void checkOrder(Order order) {
        if (order.getStatus() != FULFILLED) { // BUG!
            System.out.println("order not completed yet");
        }
    }
}
```

Refactor idea 1

```
class Order {
    enum Status { PLACED, PAID, FULFILLED, ARCHIVED }
    Status getStatus() { /*...*/ }
    boolean isComplete() { /*...*/ }
}
```

```
class Client {
    void checkOrder(Order order) {
        if (!order.isComplete()) {
            System.out.println("order not completed yet");
        }
    }
}
```

Refactor idea 2

```
class Order {
    enum Status {
        PLACED, PAID, FULFILLED, ARCHIVED;
        static final Set<Status> COMPLETE = EnumSet.of(FULFILLED, ARCHIVED);
    }
    Status getStatus() { /*...*/ }
}
```

```
class Client {
    void checkOrder(Order order) {
        if (!Status.COMPLETE.contains(order.getStatus())) {
            System.out.println("order not completed yet");
        }
    }
}
```

Connascence of Position

```
class Customer {
    Customer(String id, String name, String email) { /*...*/ }
}
```

```
class Controller {
    void create() {
        new Customer("1", "john@doe.com", "John Doe"); // BUG!
    }
}
```

using stronger types:

```
class Customer {
    Customer(UUID id, String name, Email email) { /*...*/ }
}
```

we force the correct positions on the caller

or using a builder pattern:

```
class Controller {
    void create() {
        Customer.builder()
                .id("1")
                .email("john@doe.com")
                .name("John Doe")
                .build();
    }
}
```

the position doesn't matter

Connascence of Algorithm

Multiple components must use the same algorithm or computational method.

```
class Coffee {
    void brew() {
        int brewTime = 10 + getStrength() * 5;
        /*...*/
    }
}
```

```
class Programmer {
    int getBreakTime(Coffee coffee) {
        return 10 + coffee.getStrength() * 5 + 300;
    }
}
```

```
class Coffee {
    void brew() {
        int brewTime = getBrewTime();
        /*...*/
    }
    int getBrewTime() {
        return 10 + getStrength() * 5;
    }
}
```

```
class Programmer {
    int getBreakTime(Coffee coffee) {
        return coffee.getBrewTime() + 300;
    }
}
```

Another example:

```
class UserTest {
    User user = new User("John Doe");
    void test() {
        assertThat(user.fingerprint())
                .isEqualTo(MD5.digest(user.name));
        // connascence of algorithm!
    }
}
```

```
class UserTest {
    User user = new User("John Doe");
    void testEqual() {
        assertThat(user.fingerprint())
                .isEqualTo(new User(user.name).fingerprint());
    }
    void testNotEqual() {
        assertThat(user.fingerprint())
                .isNotEqualTo(new User("Someone Else").fingerprint());
    }
}
```

Dynamic Connascences

Connascence of Execution

When the order of execution across multiple components is important.

```
class Database {
    private boolean initialized;
    void initialize() {
        initialized = true;
    }
    void saveUser(User user) {
        if (!initialized)
            throw new IllegalStateException("not initialized");
        /*..*/
    }
}
```

```
void main() {
    var db = new Database();
    db.initialize();
    db.saveUser(user);
}
```

```
class Database {
    private boolean initialized;
    private void initialize() {
        initialized = true;
    }
    void saveUser(User user) {
        if (!initialized)
            initialize();
        /*..*/
    }
}
```

```
void main() {
    var db = new Database();
    db.saveUser(user);
}
```

Connascence of Timing

real life example:

1. Cron job A places file in S3 bucket at 12:00
2. Cron job B reads file from S3 bucket at 13:00

often found in tests of async behavior:

```
void test() {
    produceEvent();

    Thread.sleep(500);
    assertThat(event).wasProcessed();
}
```

a way to solve it:

```
void test() {
    produceEvent();

    await().untilAsserted(() ->
            assertThat(event).wasProcessed());
}
```

Connascence of Value

When values in different components must change together.

```
class DatabaseService {
    void connect(Map<String, String> config) throws SQLException {
        DriverManager.getConnection(
                "jdbc:postgresql://%s%s".formatted(config.get("db.host"), config.get("db.port")),
                config.get("db.user"),
                config.get("db.password")
        );
    }
}
```

```
void main() {
    var config = new HashMap<String, String>();
    config.put("db.host", "localhost");
    config.put("db.port", "5432");
    config.put("db.user", "postgres");
    config.put("db.password", "postgres");
    new DatabaseService().connect(config);
}
```

You can refactor *Connascence of Value* to *Connascence of Name*:

```
class DatabaseConfig {
    String host, user, password;
    int port;
}
class DatabaseService {
    void connect(DatabaseConfig config) {/*..*/}
}
```

```
void main() {
    var config = new DatabaseConfig();
    config.host = "localhost";
    config.port = 5432;
    config.user = "postgres";
    config.password = "postgres";
    new DatabaseService().connect(config);
}
```

another example

```
class ProductFixture {
    static final Product PRODUCT = new Product("Eggs");
}
```

```
void test() {
    var cart = new ShoppingCart(ProductFixture.PRODUCT);

    assertThat(cart.items()).containsExactly("Eggs");
}
```

```
void test() {
    var cart = new ShoppingCart(ProductFixture.PRODUCT);

    assertThat(cart.items())
            .containsExactly(ProductFixture.PRODUCT.getDescription());
}
```

yet another example

```
class ContractFixture {
    Contract createTestContract() {
        return new Contract(Status.ACTIVE);
    }
}
```

```
void test() {
    var contract = ContractFixture.createTestContract();

    contract.deactivate();

    assertThat(contract.getStatus()).isEqualTo(Status.INACTIVE);
}
```

```
void test() {
    var contract = ContractFixture.createTestContract(Status.ACTIVE);

    contract.deactivate();

    assertThat(contract.getStatus()).isEqualTo(Status.INACTIVE);
}
```

Connascence of Identity

When multiple components must reference the exact same entity.

```
class UserService {
    User currentUser;
}
```

```
class SecurityFilter extends GenericFilterBean {
    UserService userService;
    void doFilter() {
        /*..*/ userService.currentUser = new User();
    }
}
```

```
class AuditLogger {
    UserService userService;
    void auditLog() {
        log("Current user: {}", userService.currentUser);
    }
}
```

another example

```
class Example {
    private static final Long DEFAULT = 1L;

    void doSomething(String str) {
        if (Long.valueOf(str) == DEFAULT) {
            /*..*/
        }
    }
}
```

Key Principles

- **Strength Hierarchy**: Identity > Value > Timing > Execution > Algorithm > Position > Meaning > Type > Name
- **Locality Matters**: Stronger connascence is more acceptable within the same module
- **Refactoring Direction**: Always refactor from stronger to weaker forms
- **Static vs Dynamic**: Static connascence is generally preferable (compile-time detection)

Practical Guidelines

- **Minimize** connascence across module boundaries
- **Prefer** static over dynamic connascence
- **Use** stronger types to eliminate position connascence
- **Extract** constants to convert meaning to name connascence
- **Encapsulate** algorithms in single locations
- **Design** APIs to minimize execution order dependencies

Benefits of Connascence Awareness

- **Better Design Decisions**: Quantify coupling objectively
- **Improved Refactoring**: Know which direction to refactor
- **Clearer Communication**: Common vocabulary for discussing dependencies
- **Maintainable Code**: Reduce hidden dependencies and brittleness

Learn more about connascence at:

https://connascence.io/