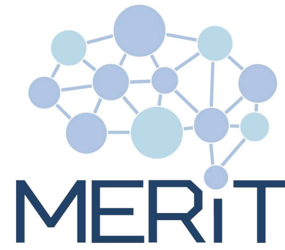
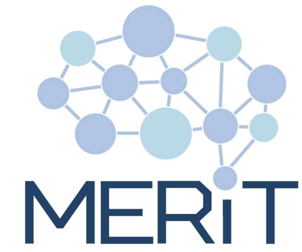


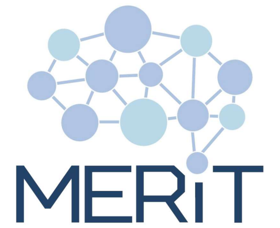
Design 2

Design Principles



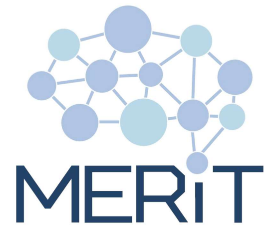
Design Principles

Definition

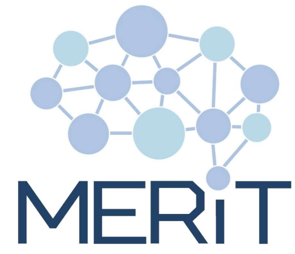


- Design principles are **heuristics based on experience**, which consist of **guiding rules** and **best practices** that help structure systems effectively.
- At their core, they exist to **manage complexity**.
- Every design principle is a distilled **lesson from failure.**

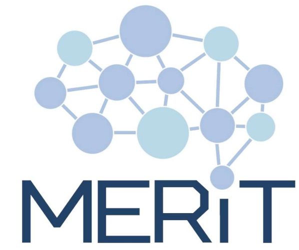
Purpose



- Slow the **natural drift toward chaos**, where everything is entangled and nothing can be touched without breaking something else.
- **Prepare** systems for **change** by isolating responsibilities, abstracting dependencies, and localizing impact.
- Close the gap between **intention and implementation**, making the system behave as conceptually designed.



SOLID

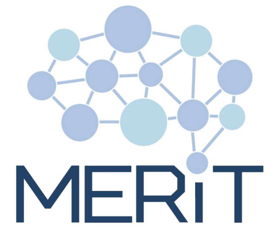


S: Single Responsibility Principle (SRP)

Definition

- “A class, module, or component should have one, and only one, reason to change.” — Robert C. Martin (Uncle Bob)
- Every software unit should have **ONLY one** clearly defined role.
- If not, changes to one part can inadvertently affect the other.

Core Idea



- SRP isn't about writing small code but about ensuring each piece of the system has a focused role, so:
  - ✔ It's easier to understand
  - ✔ Changes are localized
  - ✔ Bugs are isolated
  - ✔ Testing is more precise

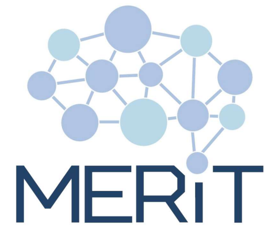
Code – SRP Violation

```
class Report:
    def generate(self):
        # logic to generate a report
        pass

    def save_to_file(self):
        # logic to save report to disk
        pass

    def send_email(self):
        # logic to send report via email
        pass
```

Code – SRP Applied

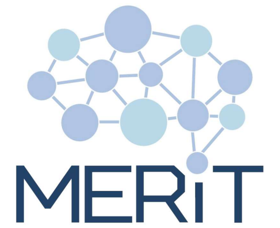
```
class ReportGenerator:
    def generate(self):
        pass

class FileSaver:
    def save(self, report):
        pass

class EmailSender:
    def send(self, report):
        pass
```

Examples

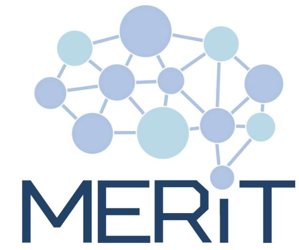


- Architectural Design: Each component serves one business domain role (e.g., AuthService, BillingService).
- Logical Design: Each module handles a single concern (e.g., InvoiceCalculator, PDFExporter).
- Git: Each command (e.g., git commit, git status, git push) performs one task.

Benefits

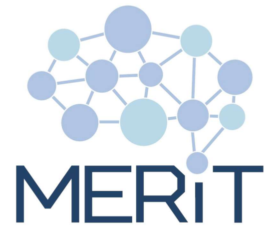


- Easier to maintain and refactor
- Fewer merge conflicts in teams
- Targeted unit testing
- Lower chance of introducing bugs



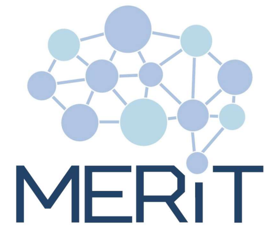
O: Open/Closed Principle (OCP)

Definition



- “Software entities (classes, modules, functions) should be open for extension but closed for modification.” — Bertrand Meyer
- New behavior should be **added** to a system **without altering** the existing code, instead **extending** it.

Core Idea



- Change is inevitable, so design for it
- OCP encourages using abstraction to define stable interfaces or base classes and extension mechanisms (like inheritance, delegation, or strategy) to add new behavior without touching existing code.

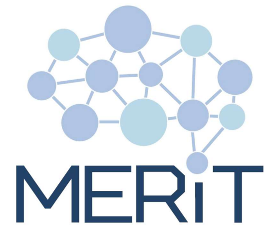
Code – OCP Violation

```
def calculate_discount(customer_type, amount):
    if customer_type == "regular":
        return amount * 0.95
    elif customer_type == "vip":
        return amount * 0.90
    elif customer_type == "employee":
        return amount * 0.85
```

Code – OCP Applied

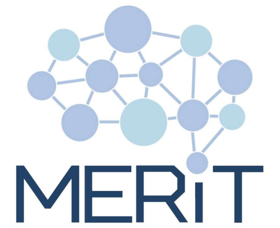
```
class DiscountStrategy:
    def apply(self, amount):
        raise NotImplementedError

class RegularCustomer(DiscountStrategy):
    def apply(self, amount):
        return amount * 0.95

class VIPCustomer(DiscountStrategy):
    def apply(self, amount):
        return amount * 0.90

class Employee(DiscountStrategy):
    def apply(self, amount):
        return amount * 0.85
```

Examples

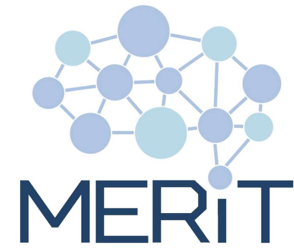


- Architectural Design: Plug-in Frameworks, Extensible Microservices, Feature Toggles
- Logical Design: Strategy Pattern, Polymorphism, Interface-Based Design
- Event-Driven Systems: Adding new subscribers without changing publishers.

Benefits

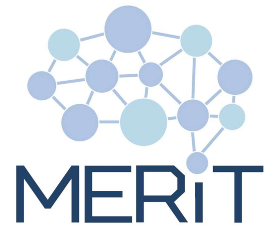


- Encourages modular, decoupled code
- Supports agile change with minimal risk
- Reduces code churn and regression
- Enables extensibility in plugins, features, integrations



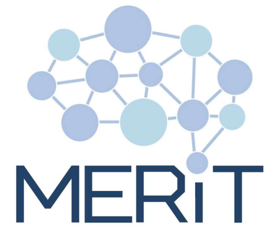
L: Liskov Substitution Principle (LSP)

Definition



- “Objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program.” — Barbara Liskov, 1987
- If **S** is a subtype of **T**, then you should be able to use **S** wherever **T** is expected, and the program should still behave correctly.

Core Idea



- LSP ensures that inheritance or interface implementation doesn't break the system.
- Subclasses must honor the contract defined by their base class. They can extend behavior, but not contradict expectations.

Code – LSP Violation

MERIT

```
class Bird:
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def fly(self):
        raise Exception("Ostriches can't fly!")
```

Code – LSP Applied

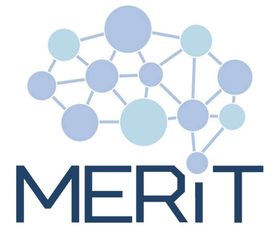
MERIT

```
class Bird:
    pass

class FlyingBird(Bird):
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def run(self):
        print("Running fast")
```

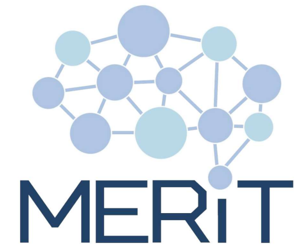
Examples

- Architectural Design: Interface design for substitutable services/plugins.
- Logical Design: Class Hierarchies, Base Interfaces, Inheritance Trees
- RESTful APIs: If v2 of your API breaks v1 contracts, clients break. LSP violated.
- UI Frameworks: A subclassed widget must still behave as a standard widget.

Benefits

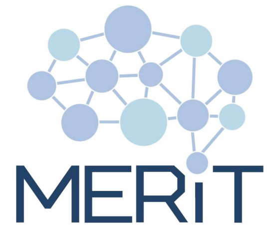


- Prevents fragile inheritance hierarchies
- Ensures consistent interface contracts
- Promotes safe polymorphism
- Enables cleaner substitution and extension



I: Interface Segregation Principle (ISP)

Definition



- “Clients should not be forced to depend on interfaces they do not use.” — Robert C. Martin (Uncle Bob)
- Keep interfaces small and specific.
- Don’t make a class or component implement methods it doesn’t need.

Core Idea

- Interfaces (or abstract base classes) should be tailored to the role of each client. If one interface tries to do too much, it creates:
  - ✔ Unused code
  - ✔ Fake implementations
  - ✔ Fragile dependencies
- ISP encourages splitting "fat" interfaces into smaller, role-specific ones.

Code – ISP Violation

MERiT

```
class Worker:
    def work(self):
        pass

    def eat(self):
        pass

class Robot(Worker):
    def eat(self):
        raise NotImplementedError("Robots don't eat")
```

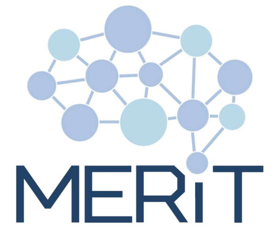
Code – ISP Applied

```
class Workable:
    def work(self):
        pass

class Eatable:
    def eat(self):
        pass

class Human(Workable, Eatable):
    def work(self): ...
    def eat(self): ...

class Robot(Workable):
    def work(self): ...
```

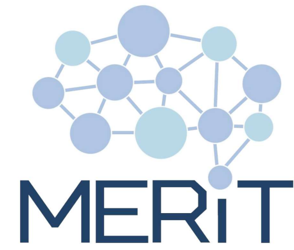
Examples

- Architectural Design: Microservices should expose focused, role-specific APIs.
- Logical Design: Split large interfaces into fine-grained abstractions.
- Frontend Services: Should consume only the API endpoints they require, not a massive shared API.
- IoT Firmware: Devices shouldn't be forced to include unused capabilities to conform to a broad interface.

Benefits



- Reduces coupling and bloated contracts
- Enables focused, role-specific implementations
- Makes systems easier to evolve and refactor
- Improves interface reusability



D: Dependency Inversion Principle (DIP)

Definition

- “High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.” — Robert C. Martin (Uncle Bob)
- Depend on interfaces, not implementations.

Core Idea

- This principle protects high-level policy from being tightly bound to low-level implementation.
- Without DIP, you get rigid, hard-coded dependencies.
- With DIP, you introduce abstraction barriers (interfaces or abstract classes) that decouple logic from implementation, making the system more modular and flexible.

Code – DIP Violation

```
class FileLogger:
    def log(self, message):
        with open("log.txt", "a") as f:
            f.write(message)

class OrderService:
    def __init__(self):
        self.logger = FileLogger()

    def place_order(self):
        # ...
        self.logger.log("Order placed")
```

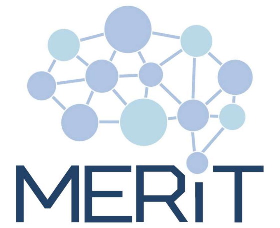
Code – DIP Applied

```
class Logger:
    def log(self, message):
        pass

class FileLogger(Logger):
    def log(self, message):
        with open("log.txt", "a") as f:
            f.write(message)

class OrderService:
    def __init__(self, logger: Logger):
        self.logger = logger

    def place_order(self):
        # ...
        self.logger.log("Order placed")
```

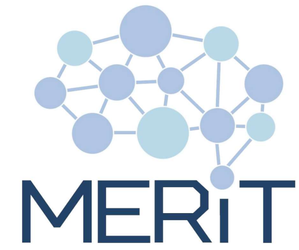
Examples

- Architectural Design: Services depend on contracts/interfaces, not on concrete services.
- Logical Design: Classes or modules receive dependencies via injection, not instantiation.
- Microservices: Depend on well-defined APIs, not on the internal logic of other services.
- Testing: You can mock or stub interfaces without touching real implementations.

Benefits

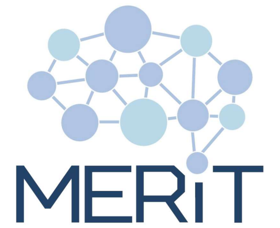


- Enables easy unit testing via mocks/stubs
- Encourages modular and loosely coupled code
- Increases reusability and configurability
- Supports inversion of control and dependency injection



DRY – Don’t Repeat Yourself

Definition



- “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” — Andy Hunt & Dave Thomas
- Avoid duplication of logic, data, and structure.
- If you change something in one place, you shouldn't have to change it in multiple places.

Core Idea

- DRY isn’t just about avoiding copy-pasted code; it’s about centralizing knowledge and responsibility.
- Duplication creates inconsistency and maintenance risks. If one copy changes while others do not, bugs may arise.

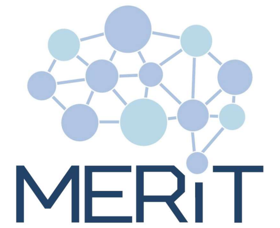
Code – DRY Violation

```
def calculate_order_total(quantity, price):
    tax = 0.18
    total = quantity * price + (quantity * price * tax)
    return total

def print_receipt(quantity, price):
    tax = 0.18
    total = quantity * price + (quantity * price * tax)
    print(f"Total: {total}")
```

Code – DRY Applied

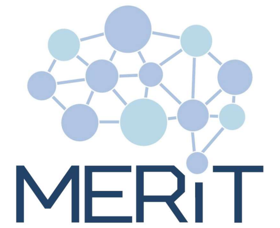
```
def calculate_total(quantity, price, tax=0.18):
    subtotal = quantity * price
    return subtotal + subtotal * tax

def print_receipt(quantity, price):
    total = calculate_total(quantity, price)
    print(f"Total: {total}")
```

Examples



- Architectural Design: Avoid repeating the same logic across services/modules (e.g., shared utility library).
- Logical Design: Common modules/functions for repeated workflows.
- Detailed Design: Extracting repeated code blocks, avoiding magic constants.
- Design Systems: Define reusable UI components instead of repeating styling.

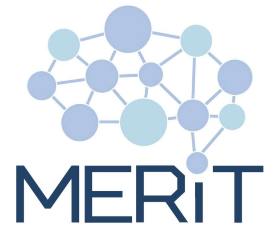
Benefits



- Reduces bugs and maintenance effort
- Centralizes change
- Improves consistency
- Encourages abstraction and modularity

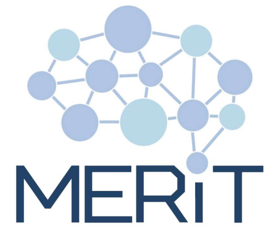
KISS – Keep It Simple, Stupid

Definition



- Design and build things in the simplest way that solves the problem.
- Avoid unnecessary complexity, clever tricks, or abstract layers that don’t provide real value.
- Simplicity makes systems easier to understand, change, and trust.

Core Idea

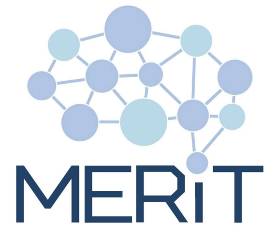


- Complexity is the enemy of clarity, maintainability, and change.
- KISS emphasizes that software should do what it needs to do (no more, no less) and it should do so in the simplest way that works.

Code – KISS Violation

```
def calculate_shipping(order_type):
    strategies = {
        'domestic': DomesticShippingStrategy(),
        'international': InternationalShippingStrategy()
    }
    return strategies[order_type].calculate(order)
```

Code – KISS Applied

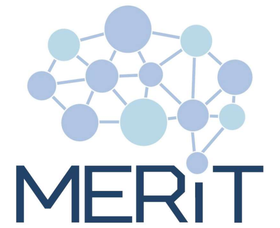



```
def calculate_shipping(order_type):
    if order_type == 'domestic':
        return 10
    elif order_type == 'international':
        return 25
```

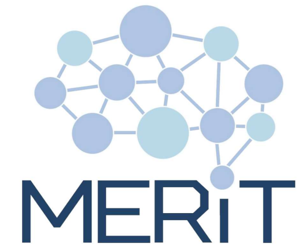
Examples

- Architectural Design: Choose the minimal viable architecture for the scale needed (e.g., monolith over microservices for MVPs).
- Logical Design: Avoid overusing inheritance, abstract factories, or overgeneralizing.
- Detailed Design: Prefer simple functions, small methods, and clear conditionals over “clever” tricks.
- MVP: Build just enough to test value.

Benefits

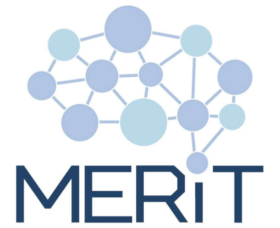


- Faster to develop and debug
- Easier for new developers to understand
- More predictable behavior
- Easier to extend later



YAGNI – You Aren’t Gonna Need It

Definition



- Don't build functionality until there is a real, current need for it — not just a prediction.

Core Idea

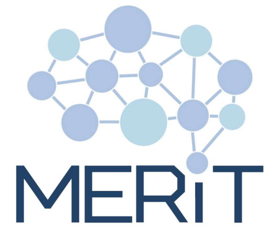
- YAGNI is about resisting the urge to over-plan or over-build. It pushes you to defer non-essential functionality until:
  - ✔ It is confirmed by the requirements
  - ✔ It delivers immediate value
  - ✔ It is worth the added complexity and cost
- This principle counters the natural developer instinct to “build it now, just in case.”

Code – YAGNI Violation

MERIT

```
class PaymentService:
    def process_stripe(self): ...
    def process_paypal(self): ...
    def process_square(self): ...
    def process_bank_transfer(self): ...
```

Code – YAGNI Applied

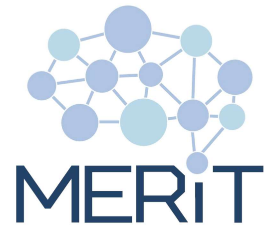



```
class PaymentService:
    def process_stripe(self): ...
```

Examples

- Architectural Design: Choose simpler architectures unless scaling demands complexity (e.g., don’t start with microservices for an MVP)
- Logical Design: Avoid designing modules that handle speculative cases or multiple modes of operation
- Detailed Design: Avoid extra parameters, branching, and feature flags unless justified by real usage

Benefits



- Reduces development and maintenance effort
- Focuses teams on delivering real, current value
- Prevents unnecessary complexity
- Increases speed and flexibility