

# Clean Code Rust Kata 詳細指南

## Kata 1: 溫度轉換器 🌡️

### 學習目標

- 良好的命名規範：函數和變數名稱要表達意圖
- 型別安全：使用 enum 而非原始型別
- 單一職責原則：每個函數只做一件事

### 設計思路

#### ✗ 不好的設計

```
rust
// 糟糕：不清楚的命名和參數
fn conv(t: f64, m: char) -> f64 {
    if m == 'c' {
        return t * 9.0/5.0 + 32.0;
    } else if m == 'f' {
        return (t - 32.0) * 5.0/9.0;
    }
    t
}
```

#### ✓ Clean Code 設計

```
rust
// 使用 enum 提供型別安全
pub enum TemperatureUnit {
    Celsius(f64),
    Fahrenheit(f64),
    Kelvin(f64),
}

// 每個轉換都是獨立、清晰的方法
impl TemperatureUnit {
    pub fn to_celsius(&self) -> f64 { /* ... */ }
    pub fn to_fahrenheit(&self) -> f64 { /* ... */ }
}
```

### Clean Code 原則應用

## 1. 表達性命名

- `to_celsius()` 比 `convert_c()` 更清楚
- `TemperatureUnit` 比 `TempType` 更完整

## 2. 避免魔術數字

rust

```
const ABSOLUTE_ZERO_CELSIUS: f64 = -273.15;  
const FAHRENHEIT_OFFSET: f64 = 32.0;  
const FAHRENHEIT_RATIO: f64 = 9.0 / 5.0;
```

## 3. 單一職責

- 每個轉換函數只負責一種轉換
- enum 封裝了值和單位

## 實作重點

- Pattern Matching：使用 `match` 處理不同單位
- 不可變性：轉換方法不修改原值，返回新值
- Display trait：提供友善的格式化輸出

## 測試策略

rust

```
#[test]  
fn test_freezing_point() {  
    let celsius = TemperatureUnit::Celsius(0.0);  
    assert_eq!(celsius.to_fahrenheit(), 32.0);  
    assert_eq!(celsius.to_kelvin(), 273.15);  
}  
  
#[test]  
fn test_absolute_zero() {  
    let kelvin = TemperatureUnit::Kelvin(0.0);  
    assert!((kelvin.to_celsius() + 273.15).abs() < 0.001);  
}
```

---

## Kata 2: 字串處理器

### 學習目標

- 函數分解：將大函數拆成小函數
- DRY 原則：避免重複程式碼
- 高階函數：善用 Iterator 方法

## 設計思路

### ✗ 不好的設計

```
rust

fn analyze(text: &str) -> (usize, usize, usize) {
    let mut words = 0;
    let mut lines = 0;
    let mut chars = 0;

    // 所有邏輯擠在一個函數中
    for line in text.lines() {
        lines += 1;
        for word in line.split_whitespace() {
            words += 1;
        }
    }
    chars = text.len();

    (words, lines, chars)
}
```

### ✓ Clean Code 設計

```
rust

pub struct TextStatistics {
    pub word_count: usize,
    pub line_count: usize,
    pub character_count: usize,
}

// 每個統計都是獨立函數
fn count_words(text: &str) -> usize { /* ... */ }
fn count_lines(text: &str) -> usize { /* ... */ }
fn count_characters(text: &str) -> usize { /* ... */ }
```

## Clean Code 原則應用

## 1. 單一職責函數

- `count_words()` 只計算字數
- `count_lines()` 只計算行數
- 每個函數都可以獨立測試

## 2. 使用結構體而非元組

rust

```
// 不好：(usize, usize, usize) - 不清楚每個值代表什麼  
// 好：TextStatistics 有明確的欄位名稱
```

## 3. Iterator 鏈式調用

rust

```
text.split_whitespace()  
  .filter(|word| !word.is_empty())  
  .count()
```

## 實作重點

- 邊界條件處理：空字串應該返回 0
- Unicode 支援：使用 `chars()` 而非 `len()` 計算字元
- Builder Pattern：可擴展為支援選擇性統計

## 進階擴展

rust

```
// 可配置的統計器  
pub struct TextAnalyzer {  
    count_words: bool,  
    count_lines: bool,  
    count_sentences: bool,  
}  
  
impl TextAnalyzer {  
    pub fn analyze(&self, text: &str) -> TextStatistics {  
        // 根據配置執行不同統計  
    }  
}
```

## Kata 3: 簡單計算機

### 學習目標

- 錯誤處理最佳實踐：使用 Result 而非 panic
- 自定義錯誤型別：提供有意義的錯誤信息
- 防禦性編程：處理邊界情況

## 設計思路

### ✗ 不好的設計

```
rust

fn divide(a: f64, b: f64) -> f64 {
    if b == 0.0 {
        panic!("Cannot divide by zero!"); // 不要 panic
    }
    a / b
}

fn calculate(expr: &str) -> f64 {
    // 假設輸入總是正確的
    let parts: Vec<&str> = expr.split(' ').collect();
    let a = parts[0].parse().unwrap(); // unwrap 很危險
    let b = parts[2].parse().unwrap();
    // ...
}
```

### ✓ Clean Code 設計

```
rust

#[derive(Debug, PartialEq)]
pub enum CalculatorError {
    DivisionByZero,
    Overflow,
    InvalidInput(String),
}

pub fn divide(dividend: f64, divisor: f64) -> Result<f64, CalculatorError> {
    if divisor == 0.0 {
        Err(CalculatorError::DivisionByZero)
    } else {
        Ok(dividend / divisor)
    }
}
```

## Clean Code 原則應用

## 1. 明確的錯誤型別

- 每種錯誤都有專門的 variant
- 錯誤訊息包含上下文資訊

## 2. 使用 Result 而非 Option

rust

```
// 不好：Option 不說明失敗原因
fn parse_number(s: &str) -> Option<f64>

// 好：Result 提供錯誤資訊
fn parse_number(s: &str) -> Result<f64, CalculatorError>
```

## 3. 早期返回模式

rust

```
pub fn evaluate(expression: &str) -> Result<f64, CalculatorError> {
    let parts = parse_expression(expression)?; // 提早處理錯誤
    let a = parse_number(parts[0])?;
    let b = parse_number(parts[2])?;

    match parts[1] {
        "+" => Self::add(a, b),
        "-" => Self::subtract(a, b),
        // ...
    }
}
```

## 實作重點

- 浮點數特殊值：檢查 `is_infinite()` 和 `is_nan()`
- 錯誤鏈：使用 `?` 操作符傳播錯誤
- 錯誤恢復：某些錯誤可以恢復，某些不行

## 錯誤處理模式

rust

```
// 使用 map_err 轉換錯誤型別
parts[0].parse::<f64>()
.map_err(|_| CalculatorError::InvalidInput("Invalid number".to_string()))?

// 使用 and_then 鏈式操作
parse_number(input)
.and_then(|n| validate_range(n))
.and_then(|n| calculate_result(n))
```

## Kata 4: 使用者驗證系統 🛡️

### 學習目標

- **Newtype Pattern**：包裝原始型別增加型別安全
- **驗證邏輯封裝**：在建構時驗證，保證物件始終有效
- **資訊隱藏**：私有欄位，公開方法

### 設計思路

#### ✗ 不好的設計

```
rust

struct User {
    username: String, // 可能是無效的
    email: String,    // 可能不是 email
    password: String, // 可能太弱
}

fn validate_user(user: &User) -> bool {
    // 驗證邏輯散落各處
    user.username.len() >= 3 &&
    user.email.contains('@') &&
    user.password.len() >= 8
}
```

#### ✓ Clean Code 設計

```
rust

// Newtype 保證型別安全
pub struct Username(String);
pub struct Email(String);
pub struct Password(String);

impl Username {
    // 只能通過驗證才能建立
    pub fn new(username: &str) -> Result<Self, ValidationError> {
        // 驗證邏輯集中在這裡
    }
}
```

## Clean Code 原則應用

### 1. Parse, Don't Validate

rust

```
// 不好：驗證後仍是 String
fn validate_email(email: &str) -> bool

// 好：解析成強型別
fn parse_email(email: &str) -> Result<Email, ValidationError>
```

## 2. 不可能的狀態不可表示

rust

```
// Username 型別保證：
// - 長度在 3-20 之間
// - 只包含合法字元
// 一旦建立，就不可能是無效的
```

## 3. 建造者模式確保有效性

rust

```
pub struct UserAccountBuilder {
    username: Option<Username>,
    email: Option<Email>,
    password: Option<Password>,
}

impl UserAccountBuilder {
    pub fn build(self) -> Result<UserAccount, ValidationError> {
        // 確保所有必要欄位都存在
    }
}
```

## 實作重點

- 驗證規則集中化：所有規則在 `new()` 方法中
- 私有建構函數：防止繞過驗證
- 不可變性：一旦建立就不能修改

## 安全性考量



rust

```
impl Password {  
    // 不實作 Display 或 Debug  
    // 避免意外洩漏密碼  
  
    pub fn verify(&self, input: &str) -> bool {  
        // 使用恆定時間比較  
        constant_time_eq(&self.0, input)  
    }  
}  
  
// 清理敏感資料  
impl Drop for Password {  
    fn drop(&mut self) {  
        // 覆寫記憶體  
        self.0.zeroize();  
    }  
}
```

## Kata 5: 圖形面積計算器

### 學習目標

- 開放封閉原則：對擴展開放，對修改封閉
- 多型與 trait：使用 trait 定義共同介面
- 里氏替換原則：子類型可以替換父類型

### 設計思路

#### ✗ 不好的設計

rust

```
enum Shape {  
    Circle { radius: f64 },  
    Rectangle { width: f64, height: f64 },  
}  
  
fn calculate_area(shape: &Shape) -> f64 {  
    match shape {  
        Shape::Circle { radius } => PI * radius * radius,  
        Shape::Rectangle { width, height } => width * height,  
        // 新增圖形需要修改這個函數！  
    }  
}
```

## ✓ Clean Code 設計

rust

```
pub trait Shape {  
    fn area(&self) -> f64;  
    fn perimeter(&self) -> f64;  
}  
  
// 新增圖形不需要修改現有程式碼  
pub struct Triangle { /* ... */}  
impl Shape for Triangle { /* ... */}
```

## Clean Code 原則應用

### 1. 依賴反轉

rust

```
// 高層模組不依賴具體實作  
pub fn total_area(shapes: &[Box<dyn Shape>]) -> f64 {  
    shapes.iter().map(|s| s.area()).sum()  
}
```

### 2. 介面隔離

rust

```
// 可以有多個 trait  
trait Drawable {  
    fn draw(&self);  
}  
  
trait Measurable {  
    fn area(&self) -> f64;  
    fn perimeter(&self) -> f64;  
}
```

### 3. 工廠模式

rust

```
pub fn create_shape(shape_type: &str, params: &[f64]) -> Result<Box<dyn Shape>, String> {  
    match shape_type {  
        "circle" => Ok(Box::new(Circle::new(params[0]?)),  
        "rectangle" => Ok(Box::new(Rectangle::new(params[0], params[1]?)),  
        _ => Err("Unknown shape".to_string()),  
    }  
}
```

## 實作重點

- 驗證建構參數：半徑和邊長必須為正
- 精確計算：使用適當的數學公式
- trait object vs 泛型：權衡動態分派和靜態分派

## 進階模式

rust

```
// 使用泛型實現零成本抽象  
pub struct ShapeCollection<S: Shape> {  
    shapes: Vec<S>,  
}  
  
// 使用 trait object 實現執行時多型  
pub struct DynamicShapeCollection {  
    shapes: Vec<Box<dyn Shape>>,  
}  
  
// Visitor 模式  
trait ShapeVisitor {  
    fn visit_circle(&mut self, circle: &Circle);  
    fn visit_rectangle(&mut self, rectangle: &Rectangle);  
}
```

---

## Kata 6: 簡單的日誌系統

### 學習目標

- 依賴注入：降低耦合度
- 策略模式：可替換的演算法
- 介面隔離原則：客戶端不應依賴不需要的的方法

### 設計思路

## ✗ 不好的設計

rust

```
pub struct Application {  
    // 直接依賴具體實作  
    console_logger: ConsoleLogger,  
    file_logger: FileLogger,  
}  
  
impl Application {  
    fn log(&self, message: &str) {  
        // 硬編碼的日誌邏輯  
        if DEBUG_MODE {  
            self.console_logger.log(message);  
        } else {  
            self.file_logger.log(message);  
        }  
    }  
}
```

## ✓ Clean Code 設計

rust

```
pub trait Logger: Send + Sync {  
    fn log(&self, message: &LogMessage);  
    fn minimum_level(&self) -> LogLevel;  
}  
  
pub struct Application {  
    // 依賴抽象而非具體實作  
    logger: Box<dyn Logger>,  
}
```

## Clean Code 原則應用

### 1. 控制反轉 (IoC)

rust

```
impl Application {  
    // 注入依賴而非建立依賴  
    pub fn new(logger: Box<dyn Logger>) -> Self {  
        Application { logger }  
    }  
}
```

## 2. 單一職責

- `Logger` trait：定義日誌介面
- `ConsoleLogger`：負責控制台輸出
- `FileLogger`：負責檔案寫入
- `Application`：業務邏輯

## 3. 組合模式

rust

```
pub struct MultiLogger {
    loggers: Vec<Box<dyn Logger>>,
}

impl Logger for MultiLogger {
    fn log(&self, message: &LogMessage) {
        for logger in &self.loggers {
            logger.log(message);
        }
    }
}
```

## 實作重點

- 執行緒安全：`Send + Sync` trait bounds
- 日誌等級過濾：在 logger 層級處理
- 格式化彈性：不同 logger 可以有不同格式

## 測試策略

rust

```
// 使用 Mock Logger 進行測試
pub struct MockLogger {
    messages: Arc<Mutex<Vec<String>>>,
}

#[test]
fn test_application_logging() {
    let mock = MockLogger::new();
    let app = Application::new(Box::new(mock.clone()));

    app.run();

    let messages = mock.get_messages();
    assert!(messages.contains(&"Application started".to_string()));
}
```

## 進階功能

```
rust

// 非同步日誌
trait AsyncLogger {
    async fn log_async(&self, message: &LogMessage);
}

// 結構化日誌
#[derive(Serialize)]
struct StructuredLogMessage {
    timestamp: DateTime<Utc>,
    level: LogLevel,
    message: String,
    context: HashMap<String, Value>,
}

// 日誌裝飾器
struct RateLimitedLogger<L: Logger> {
    inner: L,
    rate_limiter: RateLimiter,
}
```


---

## Kata 7: 迷你 TODO 應用

### 學習目標

- Clean Architecture：分層架構設計
- 領域驅動設計 (DDD)：領域模型隔離
- 六邊形架構：端口與適配器模式

### 設計思路

 不好的設計

rust

// 所有東西混在一起

```
struct TodoApp {  
    tasks: Vec<(String, bool)>, // 沒有領域模型  
  
    fn save_to_file(&self) {  
        // 業務邏輯和基礎設施混合  
        let json = serde_json::to_string(&self.tasks);  
        fs::write("tasks.json", json);  
    }  
}
```

## ✓ Clean Code 設計

rust

// 清晰的分層結構

```
mod domain { // 領域層：純粹的業務邏輯  
    struct Task { /* ... */ }  
}  
  
mod use_cases { // 應用層：業務用例  
    struct AddTaskUseCase { /* ... */ }  
}  
  
mod ports { // 端口：定義介面  
    trait TaskRepository { /* ... */ }  
}  
  
mod infrastructure { // 基礎設施層：具體實作  
    struct FileTaskRepository { /* ... */ }  
}
```

## Clean Code 原則應用

### 1. 依賴規則

Domain ← Use Cases ← Infrastructure  
(內層不依賴外層)

### 2. 領域模型純粹性

rust

```
// Task 不知道如何被儲存
impl Task {
    pub fn complete(&mut self) {
        // 只包含業務邏輯
        self.status = TaskStatus::Completed;
        self.completed_at = Some(Utc::now());
    }
}
```

### 3. Use Case 協調

rust

```
impl<R: TaskRepository> CompleteTaskUseCase<R> {
    pub fn execute(&mut self, task_id: TaskId) -> Result<(), UseCaseError> {
        // 1. 從儲存庫獲取
        let mut task = self.repository.find_by_id(task_id)?;

        // 2. 執行業務邏輯
        task.complete();

        // 3. 保存回儲存庫
        self.repository.update(task)?;

        Ok(())
    }
}
```

## 架構層級詳解

### Domain 層



rust

// 純粹的業務實體

```
pub struct Task {  
    id: TaskId,  
    title: String,  
    priority: Priority,  
    status: TaskStatus,  
}
```

// 值物件

```
pub struct TaskId(u32);
```

// 領域事件

```
pub enum TaskEvent {  
    TaskCreated(TaskId),  
    TaskCompleted(TaskId),  
}
```

## Use Case 層

rust

// 每個用例一個結構

```
pub struct AddTaskUseCase<'a, R: TaskRepository> {  
    repository: &'a mut R,  
    event_bus: &'a dyn EventBus,  
}
```

// 清晰的輸入輸出

```
pub struct AddTaskRequest {  
    pub title: String,  
    pub description: String,  
    pub priority: Priority,  
}
```

```
pub struct AddTaskResponse {  
    pub task_id: TaskId,  
}
```

## Infrastructure 層

rust

// 多種儲存實作

```
pub struct InMemoryTaskRepository { /* ... */ }
```

```
pub struct PostgresTaskRepository { /* ... */ }
```

```
pub struct RedisTaskRepository { /* ... */ }
```

// 都實作同一個介面

```
impl TaskRepository for PostgresTaskRepository {
```

```
    fn save(&mut self, task: Task) -> Result<(), RepositoryError> {
```

```
        // SQL 操作
```

```
    }
```

```
}
```

## 測試策略

rust

// 領域層測試：純邏輯測試

```
#[test]
```

```
fn test_task_completion() {
```

```
    let mut task = Task::new(/* ... */);
```

```
    task.complete();
```

```
    assert_eq!(task.status(), TaskStatus::Completed);
```

```
}
```

// Use Case 測試：使用 Mock Repository

```
#[test]
```

```
fn test_complete_task_use_case() {
```

```
    let mut mock_repo = MockTaskRepository::new();
```

```
    let mut use_case = CompleteTaskUseCase::new(&mut mock_repo);
```

```
    let result = use_case.execute(TaskId::new(1));
```

```
    assert!(result.is_ok());
```

```
    assert_eq!(mock_repo.update_called_times(), 1);
```

```
}
```

## Kata 8: 簡單的 Parser 🔍

### 學習目標

- 組合子模式：小 parser 組合成大 parser
- 函數式編程風格：不可變性、純函數
- 遞迴下降解析：結構化的解析方法

### 設計思路

## ✗ 不好的設計

rust

```
fn parse_json(input: &str) -> Value {
    let mut index = 0;
    let mut stack = Vec::new();

    // 大量的可變狀態和複雜的迴圈
    while index < input.len() {
        match input.chars().nth(index) {
            // 混亂的狀態管理
        }
    }
}
```

## ✓ Clean Code 設計

rust

```
// Parser 是一個函數：Input -> Result<(Output, Remaining), Error>
type ParseResult<T> = Result<(T, &str), ParseError>;

// 組合子：小 parser 組成大 parser
fn parse_value(input: &str) -> ParseResult<JsonValue> {
    parse_null(input)
        .or_else(|_| parse_bool(input))
        .or_else(|_| parse_number(input))
        .or_else(|_| parse_string(input))
        .or_else(|_| parse_array(input))
        .or_else(|_| parse_object(input))
}
```

## Clean Code 原則應用

### 1. 單一職責 Parser

rust

```
fn parse_null(input: &str) -> ParseResult<JsonValue>
fn parse_bool(input: &str) -> ParseResult<JsonValue>
fn parse_number(input: &str) -> ParseResult<JsonValue>
// 每個 parser 只處理一種情況
```

### 2. 不可變性

rust

```
// 不修改輸入，返回剩餘部分
fn parse_string(input: &str) -> ParseResult<JsonValue> {
    // input 不被修改
    let (string_value, remaining) = extract_string(input)?;
    Ok((JsonValue::String(string_value), remaining))
}
```

### 3. 組合子庫

rust

```
// 基本組合子
fn many<T>(parser: impl Fn(&str) -> ParseResult<T>)
-> impl Fn(&str) -> ParseResult<Vec<T>> {
    move |input| {
        let mut results = Vec::new();
        let mut remaining = input;

        while let Ok((value, rest)) = parser(remaining) {
            results.push(value);
            remaining = rest;
        }

        Ok((results, remaining))
    }
}
```

## Parser 組合子模式

### 基本組合子

rust

// 選擇：嘗試多個 parser

```
fn or<T>(p1: Parser<T>, p2: Parser<T>) -> Parser<T>
```

// 序列：依序執行 parser

```
fn and_then<A, B>(p1: Parser<A>, p2: Parser<B>) -> Parser<(A, B)>
```

// 映射：轉換 parser 結果

```
fn map<A, B>(parser: Parser<A>, f: impl Fn(A) -> B) -> Parser<B>
```

// 重複：解析多次

```
fn many<T>(parser: Parser<T>) -> Parser<Vec<T>>
```

```
fn many1<T>(parser: Parser<T>) -> Parser<Vec<T>> // 至少一次
```

// 可選：解析 0 或 1 次

```
fn optional<T>(parser: Parser<T>) -> Parser<Option<T>>
```

## CSV Parser 實作要點

rust

```
impl CsvParser {
```

// 處理引號和逸出字元

```
fn parse_quoted_field(&self, input: &str) -> ParseResult<String> {
```

// 1. 檢查開頭引號

// 2. 累積字元直到結束引號

// 3. 處理逸出序列

```
}
```

// 處理不同的換行符號

```
fn parse_line_ending(input: &str) -> ParseResult<()> {
```

parse\_string("\r\n")(input)

.or\_else(|\_| parse\_char('\n')(input))

.or\_else(|\_| parse\_char('\r')(input))

.map(|(\_, rest)| ((), rest))

```
}
```

```
}
```

## JSON Parser 實作要點

rust

```
impl JsonParser {
    // 遞迴結構
    fn parse_array(input: &str) -> ParseResult<JsonValue> {
        // 1. 解析 '['
        // 2. 解析 values (遞迴呼叫 parse_value)
        // 3. 解析 ']'

        parse_char('[')(input)
            .and_then(|(_, rest)| {
                parse_comma_separated(parse_value)(rest)
            })
            .and_then(|(values, rest)| {
                parse_char(']')(rest)
                    .map(|(_, final_rest)| (JsonValue::Array(values), final_rest))
            })
    }
}
```

## 錯誤處理與回復

rust

```
// 詳細的錯誤資訊
#[derive(Debug)]
pub struct ParseError {
    position: usize,
    expected: String,
    found: String,
}

// 錯誤恢復策略
fn parse_with_recovery<T>(
    parser: impl Fn(&str) -> ParseResult<T>,
    recovery: impl Fn(&str) -> ParseResult<T>,
) -> impl Fn(&str) -> ParseResult<T> {
    move |input| {
        parser(input).or_else(|_| recovery(input))
    }
}
```

## 效能考量

rust

// 避免過多的字串複製

```
fn parse_identifier(input: &str) -> ParseResult<&str> {
    let end = input
        .char_indices()
        .find(|(i, c)| !c.is_alphanumeric())
        .map(|(i, _)| i)
        .unwrap_or(input.len());

    Ok((&input[..end], &input[end..]))
}

// 使用 Cow 處理可能的修改
use std::borrow::Cow;

fn parse_escaped_string(input: &str) -> ParseResult<Cow<str>> {
    if input.contains("\\") {
        // 需要處理逸出，建立新字串
        Ok((Cow::Owned(process_escapes(input)), remaining))
    } else {
        // 不需要修改，直接借用
        Ok((Cow::Borrowed(input), remaining))
    }
}
```

---

## 總結：Clean Code 核心原則在 Rust 中的應用

### 1. 命名規範

- 使用 `snake_case` 表示函數和變數
- 使用 `CamelCase` 表示型別
- 避免縮寫，除非是廣為人知的（如 `ctx`, `impl`）

### 2. 函數設計

- 函數應該做一件事，並且做好
- 函數名稱應該說明它做什麼
- 參數不超過 3 個，考慮使用結構體

### 3. 錯誤處理

- 使用 `Result<T, E>` 而非 `panic!`
- 提供有意義的錯誤訊息
- 使用 `?` 操作符簡化錯誤傳播

## 4. 型別系統

- 善用 Rust 的型別系統確保正確性
- Newtype pattern 增加型別安全
- 使用 enum 表示有限的可能性

## 5. 所有權與借用

- 優先使用借用而非轉移所有權
- 明確生命週期關係
- 避免不必要的 clone

## 6. 測試

- 單元測試放在同一檔案的 `#[cfg(test)]` 模組
- 整合測試放在 `tests/` 目錄
- 使用 property-based testing 找邊界情況

## 7. 文件

- 為公開 API 撰寫文件註解
- 提供使用範例
- 說明錯誤情況

## 8. 模組組織

- 相關功能放在同一模組
- 使用 `pub(crate)` 限制可見性
- 清晰的模組階層

這些 Kata 涵蓋了從基礎到進階的 Clean Code 實踐，透過實作和重構這些練習，你將能夠寫出更清晰、更可維護的 Rust 程式碼。