Design, implementation & testing

PTS2 / FMFI UK / 25.11.2024

Jana Kostičová

Design

- Level of granularity below the software architecture
- Modeling internal structure of individual software components

Common approaches

- Object-oriented design
- Functional programming design

Object-oriented design

- Objects, their interactions and hierarchies
- UML often used
  - UML class diagrams, UML sequence diagrams, ..
- Based on
  1. Abstraction
  2. Inheritance
  3. Encapsulation
  4. Polymorphism
- Important to follow basic principles
  - SOLID principles, High cohesion-low coupling, …
- Design patterns:
  - Singleton, Factory, Observer, Adapter,...
- Bottom-up approach
  1. Objects and their relationships are defined
  2. They are combined to larger components and eventually the entire system

PTS1

Functional (programming) design

- Computation = evaluation of mathematical functions
- Focuses on pure functions, immutable data and higher-order functions
- Gaining popularity especially in the area of data processing (Scala, Python)
- Top-down approach
    1. Overall system's goals are defined
    2. The goals are expressed by smaller, more manageable functions
    3. These functions are further decomposed into smaller functions until the desired level of granularity is reached

Functional (programming) design - example

- Data flow diagram
  - Raw Data → [Clean Data] → [Analyze] → [Aggregate] → Result

→ Shows the flow of data through pure functions.

- Composition of function (Scala):

```
val cleanData = (data: List[String]) => data.filter(_.nonEmpty).map(_.trim)
val analyze = (data: List[String]) => data.map(word => (word, word.length))
val aggregate = (data: List[(String, Int)]) => data.groupBy(_._1).mapValues(_.size)

val processData = cleanData andThen analyze andThen aggregate
```

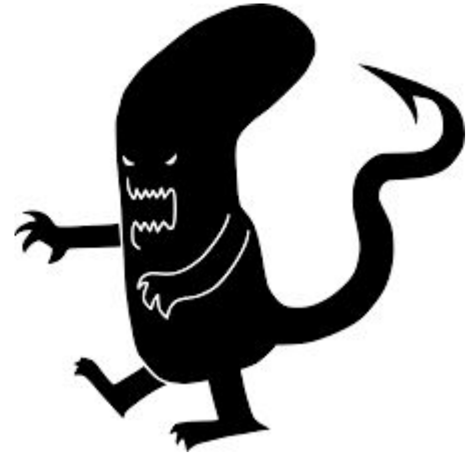
`val cleanData = (data: List[String]) => data.filter(_.nonEmpty).map(_.trim)`

- `val cleanData = ...` - declares a variable that is immutable once assigned
- `(data: List[String]) => ...` - defines a function taking a `List[String]` as input
- `data.filter(_.nonEmpty):` - filters the input list, keeping only the non-empty strings (`_.nonEmpty` is a shorthand notation for a function that checks if a string is not empty)
- `map(_.trim):` - maps each non-empty string to its trimmed version (`_.trim` is a shorthand notation for a function that trims whitespace from a string)

	O-O design	Functional programming design
Focus	Objects and their relationships	Functions and their composition
Design approach	Bottom-up	Top-down
Flexibility	More flexible	Less flexible
Modularity	More modular	Less modular
State	Stateful objects	No shared mutable state
Concurrency and parallelism	Typically more issues	Typically less issues
Debugging and testing	More complex	Simpler

Implementation & integration

- Write readable code
- Follow project structure and style guide
- Document your code properly
- Keep code DRY (Don't Repeat Yourself)
- Practice YAGNI (You Aren't Gonna Need It)
- Avoid optimizing up front
- Handle edge cases explicitly
- Use version control
  - Commit in small, logical steps
  - Commit only compilable and tested code
  - Write good commit messages !
- Practice code reviews
- Leverage CI / CD pipelines



"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth, 1974

Examples

```
def multiply_by_four(x):
return x * 4
```

vs.

```
def multiply_by_four(x):
return x << 2
```

```
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n - 1)
```

vs.

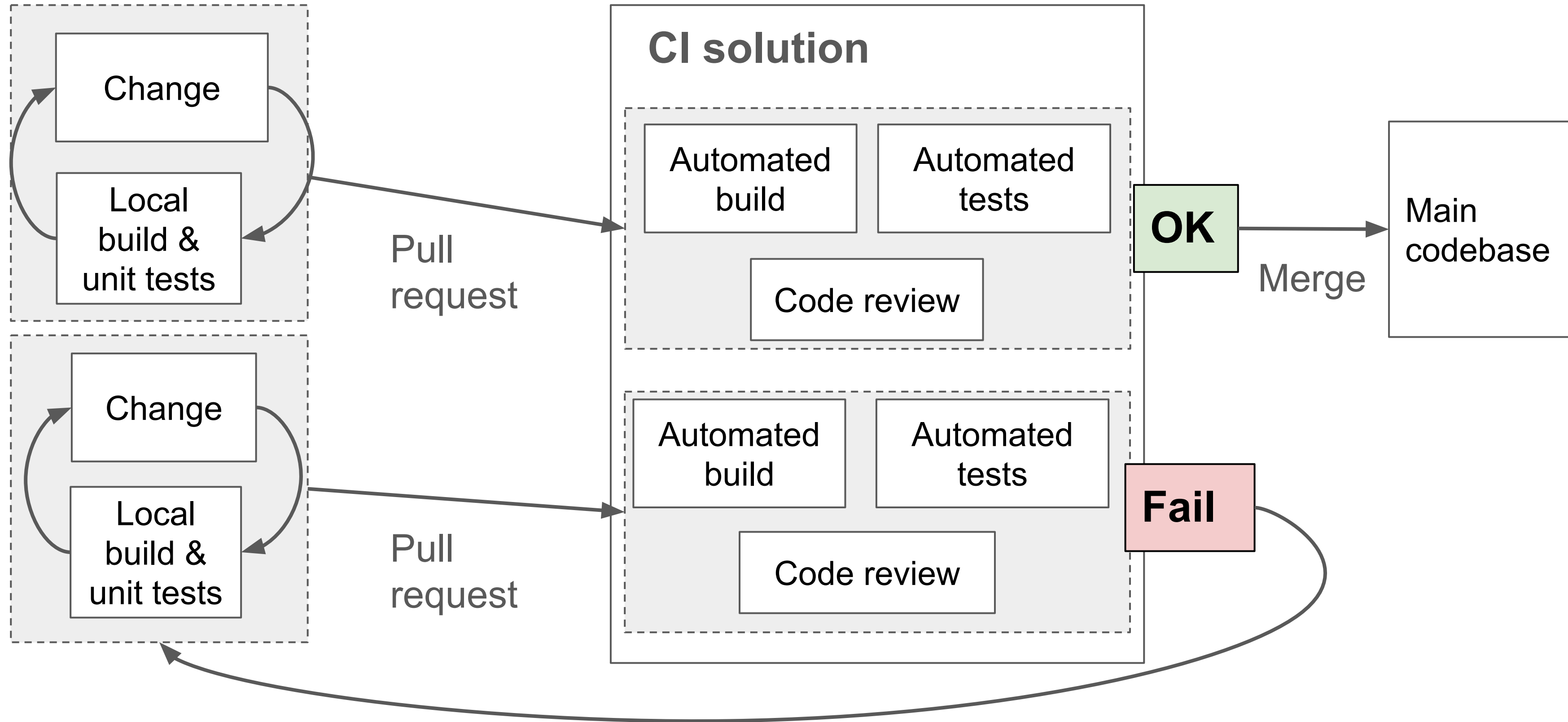
```
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

```
result = "a" + "b" + "c" + "d"
```

vs.

```
result = ''.join(["a", "b", "c", "d"])
```

Continuous Integration (CI) workflow



Testing

- Unit testing
  - Testing individual units of code (functions, classes, modules) in isolation
- Integration testing
  - Testing groupings of integrated components
- System (end-to-end) testing
  - Testing entire system (product)
- GUI tests
  - Tools for automated GUI tests: Selenium, Appium, Playwright..
  - Example: Selenium Test Case

Written mostly by developers (Unit testing, Integration testing)

Written mostly by QA engineers (System testing, GUI tests)

- CI / CD workflow: tests run automatically before merge
- Test-driven development: First write tests, then code

Resources

- Martin, R.C.: Clean code - A Handbook of Agile Software Craftsmanship, 2009.
- Hunt, A., Thomas D.: The Pragmatic Programmer - From Journeyman to Master, 2000.
- Knuth, D.E. Structured Programming with go to Statements. ACM Comput. Surv. 6, 4 (Dec. 1974), 261–301.
- Whittaker, J.A.: “What Is Software Testing? And Why Is It so Hard?” IEEE Software, vol. 17, no. 1, 2000, pp. 70–79.

- Relevant PTS1 lectures