Software Development Principles

Dave van Soest

February 13, 2014

Introduction

DRY

KISS

Premature Optimization

SOLID

Best Practices

Questions & Discussion

Introduction

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     printf("Hello world!\n");
5
6     return 0;
7 }
~
~
~
helloworld.c            4,1-4        All
```

Goals

- Write software that contains less errors.
- Write software that is more readable and better maintainable.
- Write software faster.
- Follow well-defined patterns.
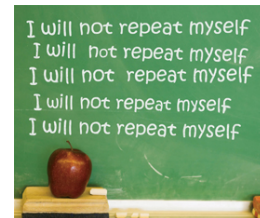- Focus on solving the issues that create value.

Reducing dependencies

Every software development principle, best practice, pattern and valuable technique is about reducing dependencies.

Fewer dependencies between parts of a system makes it easier to...
- modify behavior.
- optimize.
- refactor.
- (unit) test.
- deploy.

DRY

DRY: Don't Repeat Yourself

- Don't Repeat Yourself
- DON'T REPEAT YOURSELF!.
- No seriously, do not repeat yourself, ever.

DRY: Why?

- "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."
- Implicit dependencies can and will eventually cause inconsistencies in your system.

DRY: In practice

- Remove duplicate (i.e. 100% identical) code.
- Code should not contain the same logic twice, even if it's just a single statement.
- Prevent fields with derivative data in model design.
- Prefer parameterized helpers over logical code units that are more than 50% similar.
- Write short and concise function bodies. This helps you detecting similar code.
- Does not only apply to code, but to all project sources. E.g.: database schemas, configuration definitions, build/C.I. system, documentation, ...
- Use code/documentation/... generators in order to keep your sources free from repetition.
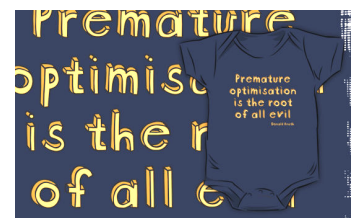- Check for repetition during code review.

KISS

KISS: Keep It Simple, Stupid!

- "Write programs for people first, computers second."
- Maximize design simplification.
- Write clear code. (Decompose statements; use temporary variables; ...)
- Maintainability: "What you can't comprehend, you can't change with confidence."

Premature Optimization

Avoid premature optimization

- Writing optimized code takes longer.
- Optimized code is more complex, thus less readable and harder to maintain.
- Optimized code often introduces dependencies.
- Adds no value without an actual performance need.
- Most operations aren't CPU-bound. Optimizing a particular piece of code may have a neglectable impact on overall performance.
- Developers are notoriously bad at estimating what to optimize.
- Compilers and interpreters are far better at optimizing code. Some seeming optimizations may even block the compiler/interpreter from applying it's own optimizations.
- Beware: Don't pessimize prematurely! If it doesn't take longer to implement and doesn't reduce code clarity, please do it.

How to optimize code, when the need is proven

- Measure first.
- Define optimization goals.
- Look for algorithmic optimization first.
- Try to encapsulate and modularize the optimization.
- Write a comment explaining the reason of the optimization.
- Keep the unoptimized code, both as a reference and to write correctness tests against.
- Run the optimized code in different scenarios, not only in the scenario you're trying to optimize.

SOLID

SOLID: Single responsibility principle

- Give each *entity* (variable, function, class, namespace, module, library, system) one well-defined responsibility.
- Entities with multiple responsibilities are hard to design and implement.
  - The number of combinations of the entity's state and behavior easily explodes.
- Always describe the responsibility of an entity in a comment inside the code. For variables, a descriptive name can be sufficient.
  - Rule of thumb: If the responsibility can't be described without conjunctives or disjunctives, then it violates this principle.

SOLID: Open/closed principle

- Software entities should be open for extension, but closed for modification.
- *Open*: Extension points of an entity and their correct usage are clearly defined.
- *Closed*: The behavior of the entity can be modified without changing its source code.
  - But only the parts that are designed to be modifiable.
  - No need for recompilation, code reviews, running unit tests of the original entity.

SOLID: Liskov substitution principle

- "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
- This principle imposes a number of requirements on the definition and implementation of subtypes and their methods.

SOLID: Interface segregation principle

- "No client should be forced to depend on methods it does not use."
- Use concise, specific interfaces instead of bulky multi-purpose interfaces.

SOLID: Dependency inversion principle

- High-level modules should not depend on low-level modules.
- Abstractions should not depend on details. Details should depend on abstractions.
- Useful techniques:
  - Inversion of Control.
  - Dependency Injection.
  - Plugins.
  - Interfaces.

Best Practices
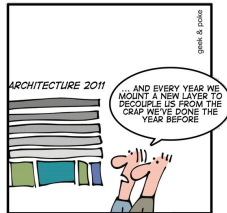
Best practices: Data

- Minimize global and shared data.
- Hide information.
  - Protects the client by preventing inconsistent state; strenghthens invariants.
  - Makes client interfaces clear.
  - Decreases coupling.
  - Supports future implementation changes.

Best practices: Programming style

- Avoid magic numbers and string literals.
  - Make semantics clear by assigning the value to a constant with a descriptive name.
  - Avoid repetition, reuse the same constant. (DRY!)
  - Avoid implicit dependencies.
  - Avoid suggestive dependencies.
- Avoid long functions and deep nesting.
- Prefer build-time errors to run-time errors.
- Avoid cyclic dependencies.

Best practices: Comments

- Write comments well.
- Know what to comment... and what not.
- Always document classes and functions using comments.
- Document a variable if its meaning is not clear from its name and context.
- Add a between-code comment in case the code could work unexpectedly different than it might appear.
- Add a between-code comment to document assumptions.
- Leave out obvious comments.
- Prefer clear code and assertions over comments.
- Prefer DRY comments. Don't repeat (parts of) other comments. Don't repeat logic that is already clear from the code.

Best practices: Design patterns

- Stand on the shoulders of giants!
- Most challenges are generic, with generic solutions already available.
- Common vocabulary to use in design discussions.

Questions & Discussion