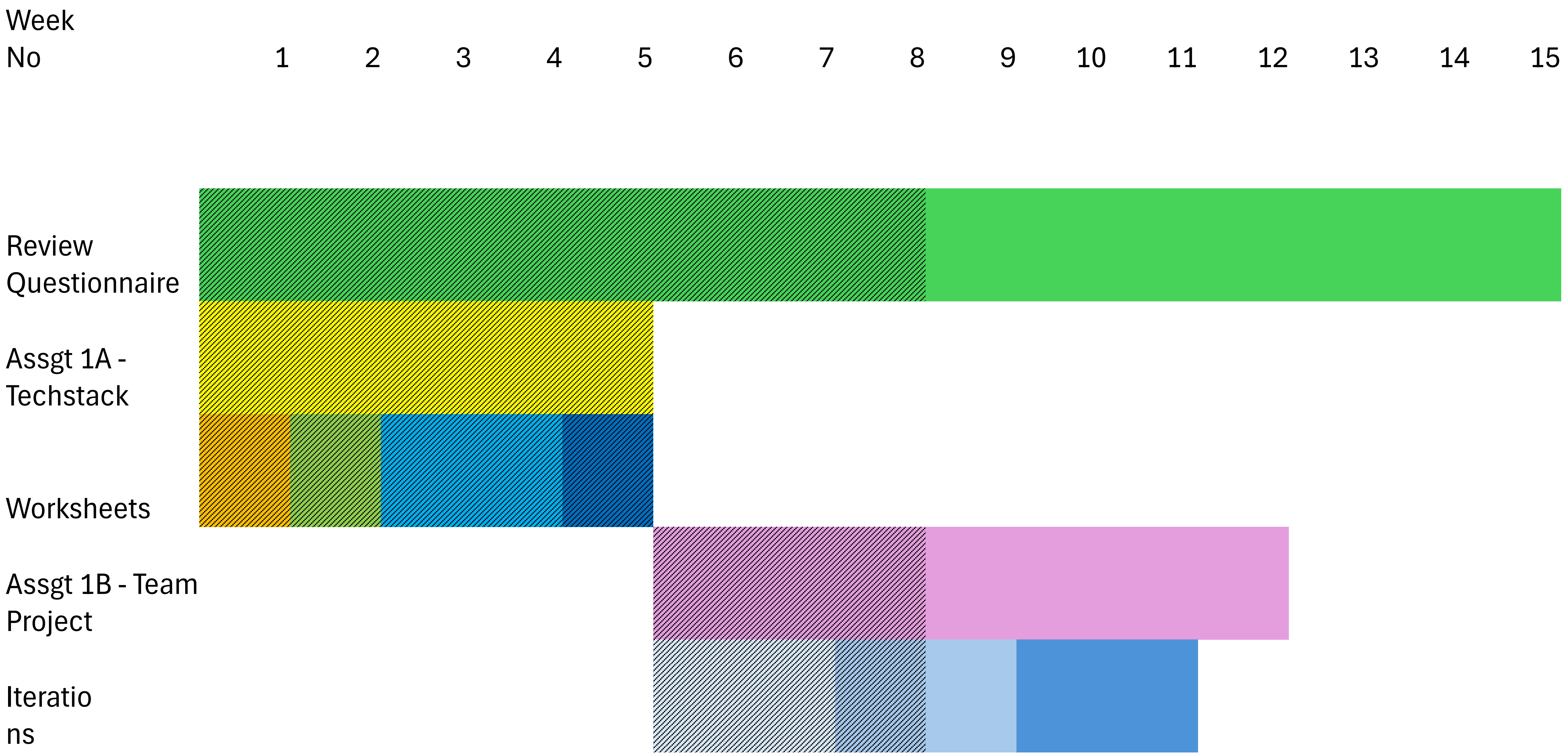
Code Craft and Code Quality

Week 9

Taking Stock



Quality of the code

Behaves as expected – no bugs **Unit tests as a "contract"**

Easy to change

Easy to understand how it works and change

The intention of the code is clear **Naming and structure**

Change is predictable – impact is limited **Small code structures**

Quality of the product

Is useful to users – solves a problem

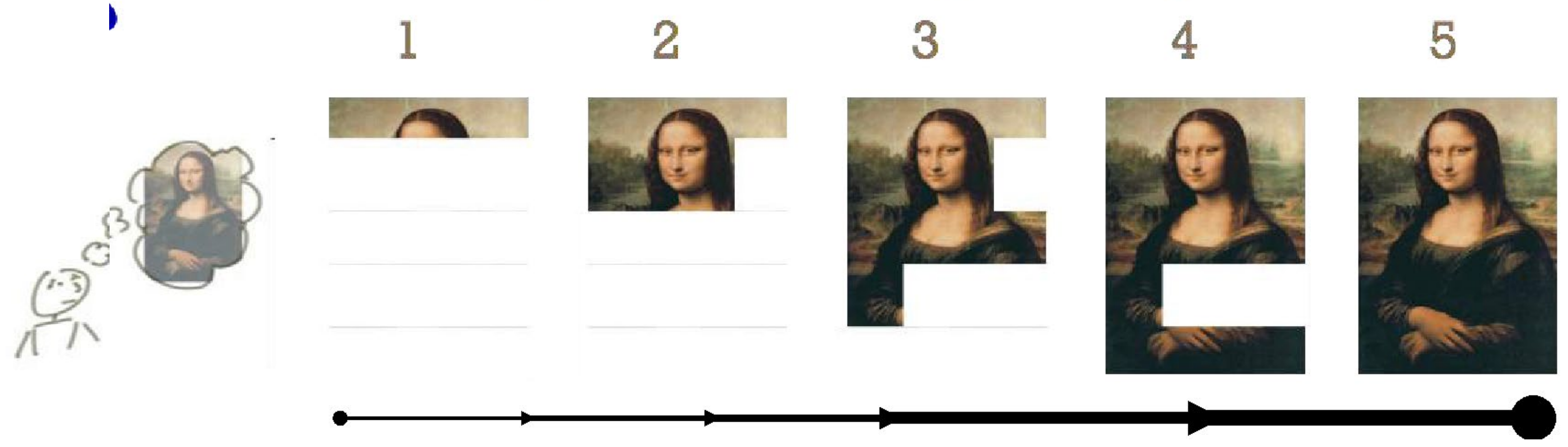
Behaves as expected

Specifications and Scenarios
Based on Acceptance crieria

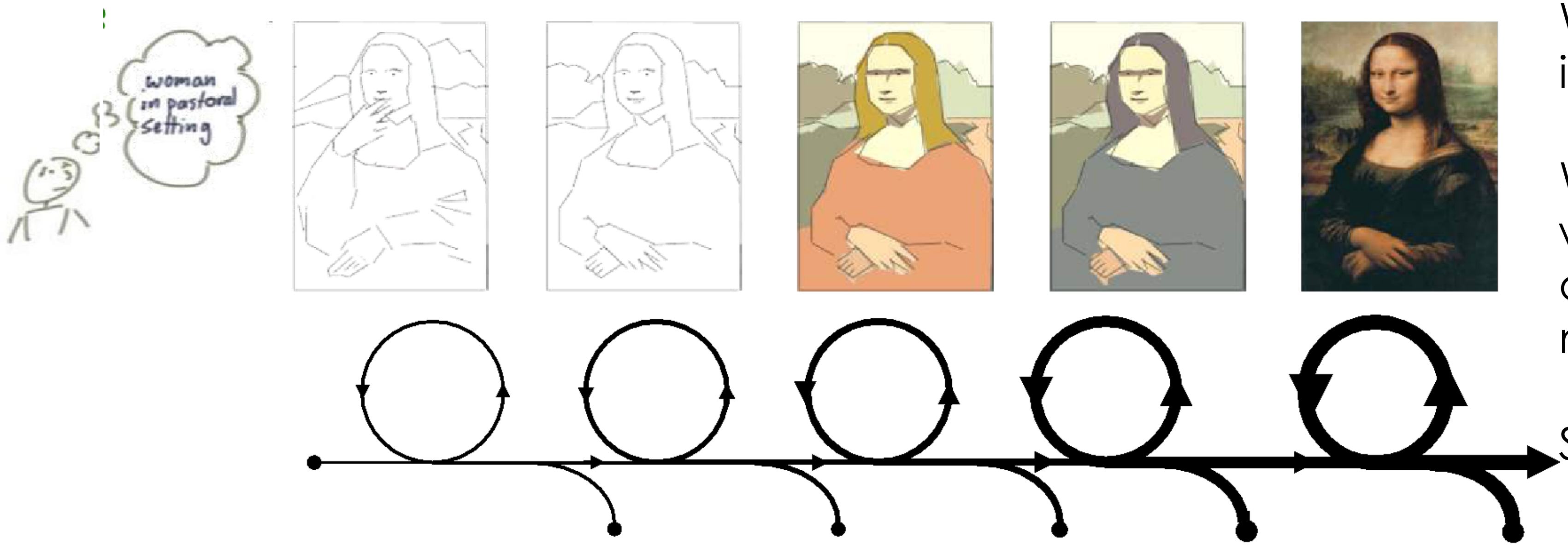
Techniques with strong empirical and anecdotal evidence of improving code quality

Test Driven Development
CI/CD
Pair/mob programming
Code Reviews
Explicit Coding standards –Static code checkers - Linters, SonarQube etc

Coding is a craft
Crafting code is different to just writing it!



We do NOT create a full design then build from the ground up until we have the finished product.



We start with a sketch, iteratively adding detail.

We revise, extend and refine - working at different levels of abstraction until the software meets someone's needs.

Software is never really finished

Why it is important to have well-crafted clean code?

Quality software is developed in teams

CODE is read more often than it is written

Other people will need to read and understand how your code works to extend it, debug it, change it or remove it.

You may need to do the same a day later, two weeks later, 6 months later

THINK ABOUT WHO WILL COME NEXT!
BE A GOOD TEAM MATE!

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. " — Martin Golding

So how can I craft my code so it is easier for me and others to understand how it works?

Code (and think) small!

"The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that." — Robert C. Martin

Function bodies should rarely be more than 20 line long and mostly less than 10 lines

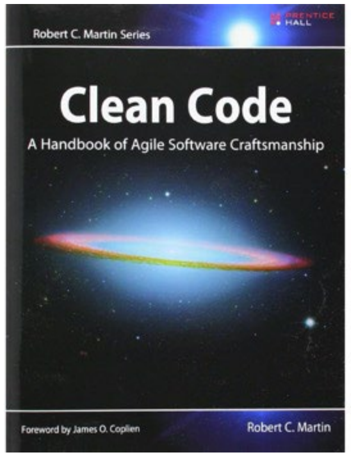
Functions should take as few arguments as possible, preferably none

Functions should do one thing — and do it well

Classes should be sized so they are responsible for one thing only

Easier to follow and understand – low cognitive load

The Single Responsibility Principle (SRP) – the "S" in SOLID principles


Robert C. Martin Series
Prentice Hall
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

e.g. a function that fetches, manipulates and stores data should be split into three smaller functions

WARNING: Too many tiny classes can be difficult to understand and change

Make code Self-documenting (readable, its intention is clear, understandable)

“Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.” — Robert C. Martin

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

Do NOT use magic numbers
65 should be replaced with

Const minAgeForBenefits = 65

Gets refactored to :

```
if (employee.isEligibleForFullBenefits())
```

- The comment is removed
- The conditional logic is encapsulated into a method
- Because a method is used and not a free-standing function, instance variables can be used, creating a zero-argument method call
- The method is given a descriptive name, making its responsibility super clear

https://medium.com/better-programming/clean-code-5-essential-takeaways-2a0b17ccd05c

Example of readability of a function

```
const handleSubmit = (event) => {
event.preventDefault();
NoteAdapter.update(currentNote)
.then(() => {
setCurrentAlert('Saved!')
setIsAlertVisible(true);
setTimeout(() => setIsAlertVisible(false), 2000);
})
.then(() => {
if (hasTitleChanged) {
context.setRefreshTitles(true);
setHasTitleChanged(false);
}
});
};
```

https://itnext.io/tips-for-writing-self-documenting-code-e54a15e9de2

```
const showSaveAlertFor = (milliseconds) => () => {
setCurrentAlert('Saved!')
setIsAlertVisible(true);
setTimeout(
() => setIsAlertVisible(false),
milliseconds,
);
};
const updateTitleIfNew = () => {
if (hasTitleChanged) {
context.setRefreshTitles(true);
setHasTitleChanged(false);
}
};const handleSubmit = (event) => {
event.preventDefault();
NoteAdapter.update(currentNote)
.then(showSaveAlertFor(2000))
.then(updateTitleIfNew);
};
```

Single responsibility

C# code

```
class User
{
    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
                                 ex.ToString());
                         Errors.txt", ex.ToString());


}
```

```
class Post
{
    private ErrorLogger errorLogger = new ErrorLogger();

    void CreatePost(Database db, string postMessage)
    {
        try
        {


                       eption ex)

            errorLogger.log(ex.ToString())
        }
    }
}

class ErrorLogger
{
    void log(string error)
    {
     db.LogError("An error occured: ", error);
     File.WriteAllText("\LocalErrors.txt", error);
    }
}
```

In the article Principles of Object Oriented Design, Robert C. Martin defines a responsibility as a 'reason to change', and concludes that a class or module should have one, and only one, reason to be changed.

CreatePost() can create a new post, log an error in the database, and log an error in a local file

https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4

It’s all in the name…

```
const fStuNms = stus.map(s => s.n)
```

Whaaaat?

```
const filteredStudentNames = students.map(student => {
return student.name;
});
```

- Use intention-revealing names — e.g, **int elapsedTimeInDays**, not **int days**
- Use pronounceable names — e.g., **Customer**, not **DtaRcrd102**
- Avoid encodings — don’t use an m_ prefix for members and don’t use Hungarian notation
- Pick one word per concept — don’t fetch, retrieve, get for the same concept

Common naming conventions

If your value is a boolean, start with **is** or **has**, like **isEnrolled: true**

If your value is storing an array, the name should be plural, eg **students**

Numbers should start with min or max if possible

For functions, there should be a helpful verb in front,
like **createSchedule** or **updateNickname**

Naming standards for Java
https://google.github.io/styleguide/javaguide.html#s5-naming

https://itnext.io/tips-for-writing-self-documenting-code-e54a15e9de2

Write (and read) Useful Test Descriptions

const **getDailySchedule** = (**student**, **dayOfWeek**) => {

It retrieves the daily schedule; if the day of the week is a weekend it returns an empty array; if the student has detention it sticks it onto the end of the schedule; and if the student isn’t enrolled in the school, it prints a link to a the school website.

describe('getDailySchedule tests', () => {
it("*retrieves the student's full schedule*", () => {
it('*returns an empty array if given a weekend day*', () => {
it('*adds detention if a student got one that day*', () => {
it('*prints a school website link if student not enrolled yet*', () => {

https://itnext.io/tips-for-writing-self-documenting-code-e54a15e9de2

Techniques for crafting clean code…

Refactoring is the process of restructuring existing computer code without changing its external behavior.

Test-driven development is a process where requirements are turned into specific test cases, then the code is added so the tests pass.

The process of crafting software might look something like this:

1. Write failing tests that verify the required but unimplemented behaviour.

2. Write some (potentially bad) code that works and makes those tests pass.

3. Incrementally refactor the code, with the tests continuing to pass, making it more clean with each development iteration.

Design Patterns

Software design patterns provide templates and tricks used to design and solve recurring software problems and tasks. Applying time-tested patterns result in extensible, maintainable and flexible high-quality code, exhibiting superior craftsmanship of a software engineer.

https://www.educative.io/courses/software-design-patterns-best-practices

Design Patterns have become an object of some controversy in the programming world in recent times, largely due to their perceived ‘over-use’ leading to code that can be harder to understand and manage.

The Gang of Four and 23 Design Patterns

Creational Patterns

- ○ Builder Pattern
- ○ Singleton Pattern
- 🔒 Prototype Pattern
- 🔒 Factory Method Pattern
- 🔒 Abstract Factory Pattern

Structural Patterns

- 🔒 Adapter Pattern
- 🔒 Bridge Pattern
- 🔒 Composite Pattern
- 🔒 Decorator Pattern
- 🔒 Facade Pattern
- 🔒 Flyweight
- 🔒 Proxy Pattern

Behavioral Patterns

- 🔒 Chain of Responsibility Pattern
- 🔒 Observer Pattern
- 🔒 Interpreter Pattern
- 🔒 Command Pattern
- 🔒 Iterator Pattern
- 🔒 Mediator Pattern
- 🔒 Memento Pattern
- 🔒 State Pattern
- 🔒 Template Method
- 🔒 Strategy Pattern
- ○ Visitor Pattern

Common OOP problem and solution patterns

Singleton
The singleton pattern is used to limit creation of a class to only one object. This is beneficial when one (and only one) object is needed to coordinate actions across the system. There are several examples of where only a single instance of a class should exist, including caches, thread pools, and registries.

Factory Method
A normal factory produces goods; a software factory produces objects. And not just that — it does so without specifying the exact class of the object to be created. To accomplish this, objects are created by calling a factory method instead of calling a constructor.

Strategy **Observer** **Builder** **Adapter** **State**

https://www.geeksforgeeks.org/category/design-pattern/
https://www.geeksforgeeks.org/software-design-patterns/

https://medium.com/educative/the-7-most-important-software-design-patterns-d60e546afb0e

Code reviews … another code crafting enabler

https://medium.com/better-programming/how-to-review-code-in-7-steps-98298003b7ec

DRY (WET), YAGNI

YAGNI (You Aren't Gonna Need It)
Do not implement something until you are going to need it

DRY (Don't Repeat Yourself)
A piece of code should be implemented in just one place in the source code

You can create a common function or abstract your code to avoid any repetition in your code.

(WET= Write everything Twice!)

Making OOP with a SOLID Design

Introduced by Robert C. Martin (Uncle Bob), in his 2000 paper *Design Principles and Design Patterns*.
The actual SOLID acronym was, however, identified later by Michael Feathers ("Working with Legacy Code").

S — Single responsibility principle
every module or class should have responsibility over a single part of the functionality provided by the software.

O — Open/closed principle
utilize inheritance and/or implement interfaces that enable classes to polymorphically substitute for each other

L — Liskov substitution principle
objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

I — Interface segregation principle
no client should be forced to depend on methods it does not use

D - Dependency inversion principle
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.

SOLID Design Principles

Introduced by Robert C. Martin (Uncle Bob), in his 2000 paper *Design Principles and Design Patterns*.
The actual SOLID acronym was, however, identified later by Michael Feathers ("Working with Legacy Code").

https://en.wikipedia.org/wiki/SOLID

In software engineering, **SOLID** is a mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable.

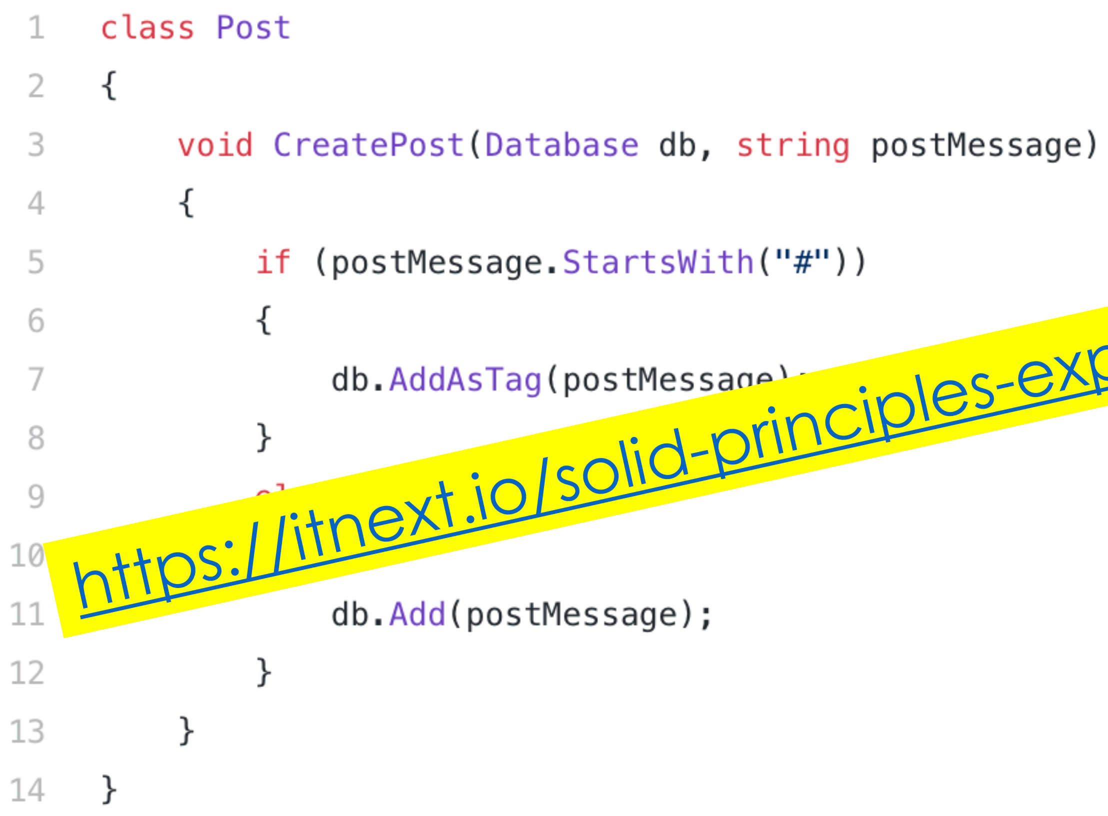
Sandi Metz (May 2009). "SOLID Object-Oriented Design". YouTube.
Archived from the original on 2021-12-21.
Retrieved 2019-08-13. Talk given at the 2009 Gotham Ruby Conference.

https://www.youtube.com/watch?v=v-2yFMzxqwU

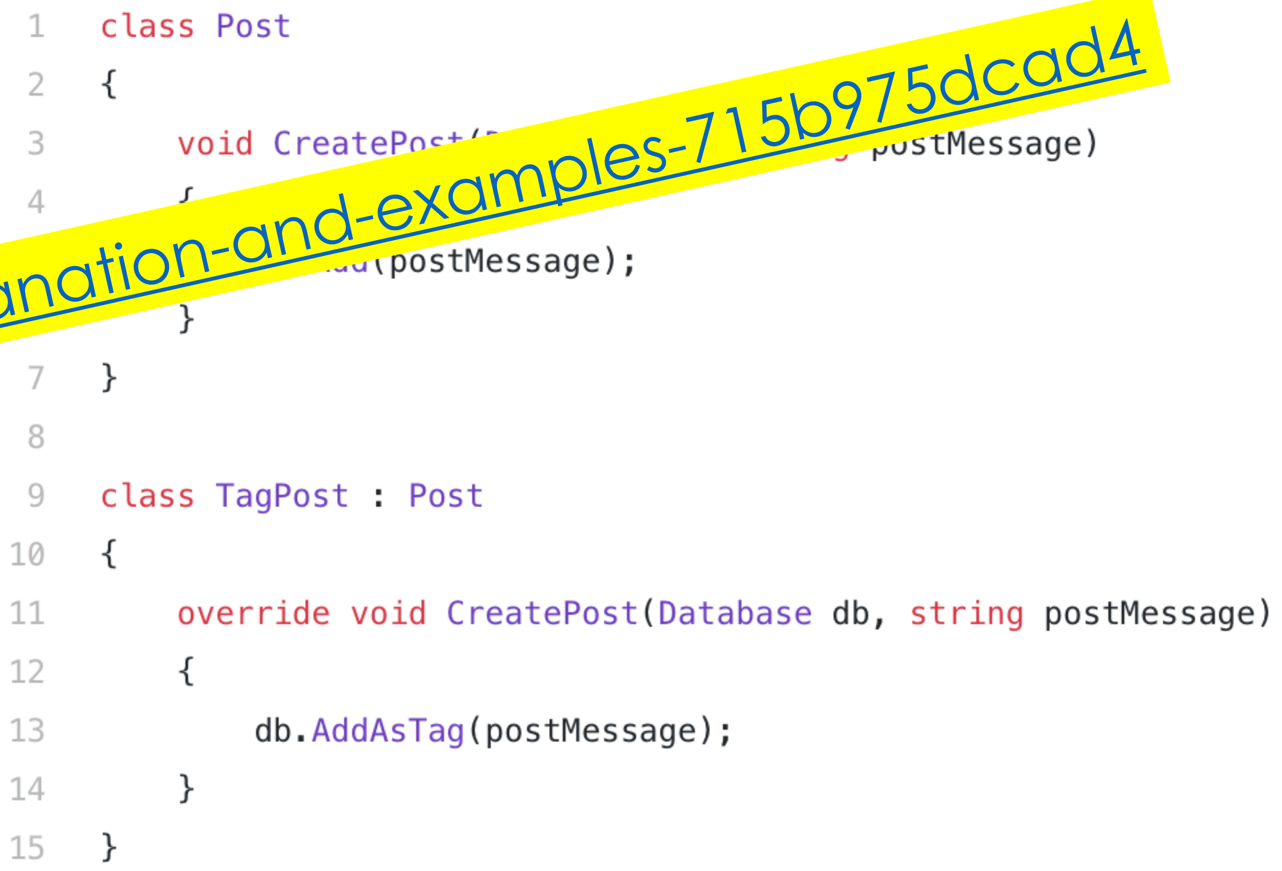
https://www.youtube.com/watch?v=6Bia81dI-JE (check out 9 mins 30 ff.)
Building on SOLID foundations - Steve Freeman & Nat Pryce
Authors of: *Growing Object-Oriented Software, Guided by Tests*

O — Open/closed principle

We can make sure that our code is compliant with the open/closed principle by utilizing inheritance and/or implementing interfaces that enable classes to polymorphically substitute for each other.



```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        if (postMessage.StartsWith("#"))
        {
            db.AddAsTag(postMessage)[illegible]
        }
        [illegible]
        [illegible]
            db.Add(postMessage);
        }
    }
}
```



```
class Post
{
    void CreatePo[illegible]postMessage)
    [illegible]
    [illegible](postMessage);
    }
}

class TagPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        db.AddAsTag(postMessage);
    }
}
```

https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4

do something specific whenever a post starts with the character ‘#’.

If we later wanted to also include mentions starting with ‘@’, we’d have to modify the class with an extra ‘else if’ in the CreatePost() method

The evaluation of the first character ‘#’ will now be handled elsewhere

The 12 Factor App

https://12factor.net

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity
Keep development, staging, and production as similar as possible

XI. Logs
Treat logs as event streams

XII. Admin processes
Run admin/management tasks as one-off processes

Roadmap Resources - Topics

Skill Based

https://roadmap.sh/react

https://roadmap.sh/javascript

https://roadmap.sh/typescript

Role Based

https://roadmap.sh/frontend

https://roadmap.sh/backend

e.g. Architectural Patterns - 12 Factor Apps

https://www.youtube.com/watch?v=FryJt0Tbt9Q

roadmap.sh is a community effort to create roadmaps, guides and other educational content to help guide the developers in picking up the path and guide their learnings.

https://roadmap.sh/

I. Codebase

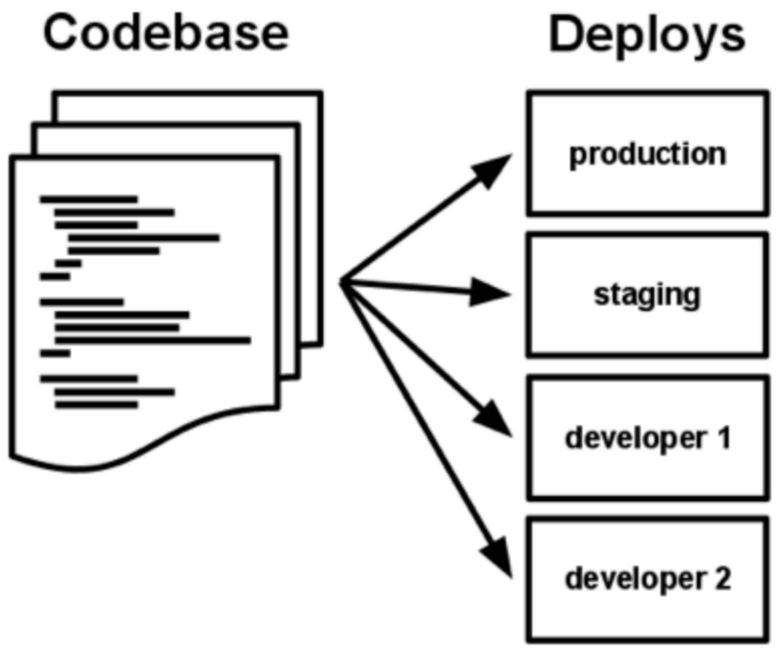
One codebase tracked in revision control, many deploys

A twelve-factor app is always tracked in a version control system, such as Git, Mercurial, or Subversion. A copy of the revision tracking database is known as a *code repository*, often shortened to *code repo* or just *repo*.

A *codebase* is any single repo (in a centralized revision control system like Subversion), or any set of repos who share a root commit (in a decentralized revision control system like Git).

There is always a one-to-one correlation between the codebase and the app:

- If there are multiple codebases, it's not an app – it's a distributed system. Each component in a distributed system is an app, and each can individually comply with twelve-factor.
- Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the dependency manager.



There is only one codebase per app, but there will be many deploys of the app. A *deploy* is a running instance of the app. This is typically a production site, and one or more staging sites. Additionally, every developer has a copy of the app running in their local development environment, each of which also qualifies as a deploy.

The codebase is the same across all deploys, although different versions may be active in each deploy. For example, a developer has some commits not yet deployed to staging; staging has some commits not yet deployed to production. But they all share the same codebase, thus making them identifiable as different deploys of the same app.

"Clean code is not written by following a set of rules. You don't become a software craftsman by learning a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines." — Robert C. Martin

Try to read this sort of stuff every day

https://medium.com/better-programming/10-must-read-books-for-software-engineers-edfac373821b

https://www.makeuseof.com/tag/basic-programming-principles/
https://www.geeksforgeeks.org/7-common-programming-principles-that-every-developer-must-follow/

https://medium.com/better-programming/clean-code-5-essential-takeaways-2a0b17ccd05c

https://medium.com/better-programming/how-to-review-code-in-7-steps-98298003b7ec

https://medium.com/young-coder/is-it-time-to-get-over-design-patterns-8851864a6834

Questions and Comments....