LECTURE 6: DESIGN PRINCIPLES - SOLID

ANDREY.BOCHARNIKOV@GMAIL.COM

TELEGRAM: @RICKO_X

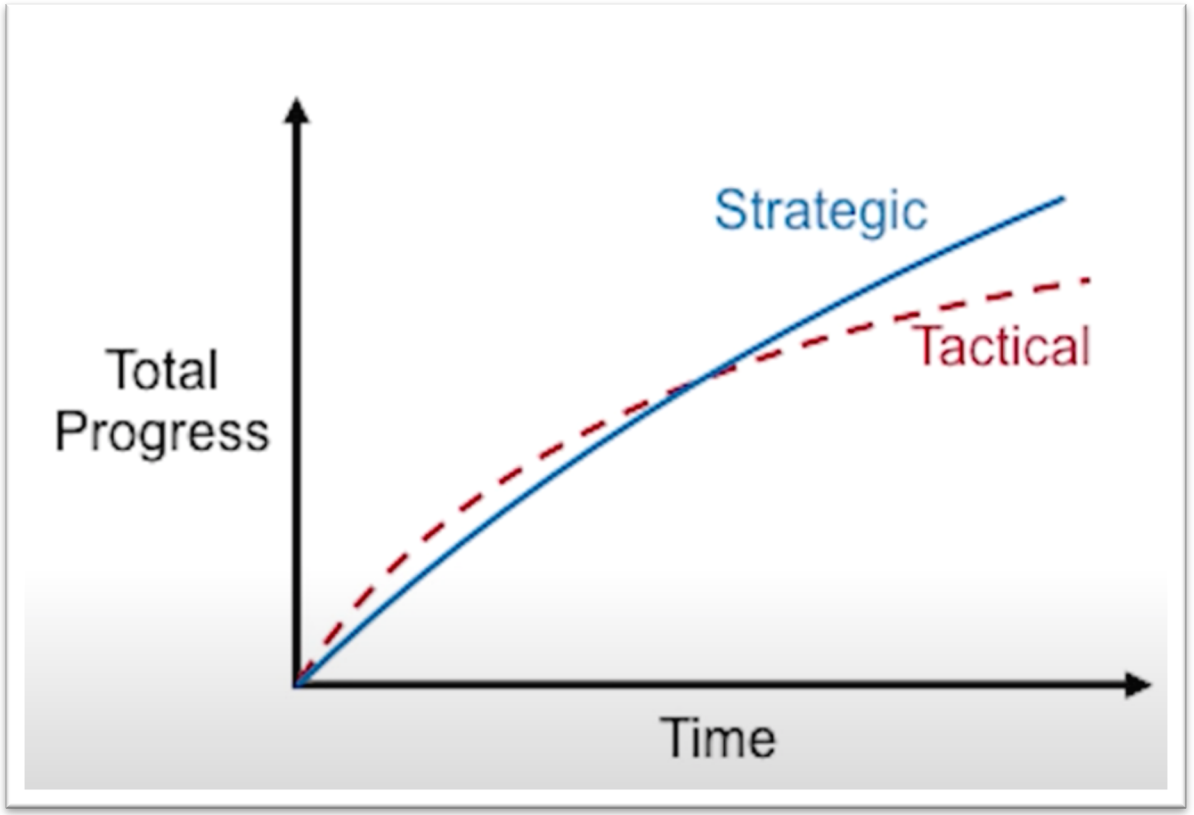
TACTICAL VS STRATEGIC PROGRAMMING

- Tactical programming
  - Goal: get next feature/bug fix working ASAP
  - A few shortcuts and gludges are OK?
  - Result: bad design, high complexity
  - Exterme: tactical tornadoes
- Complexity is incremental

- **Working code isn't enough**

TACTICAL VS STRATEGIC PROGRAMMING

- Strategic programming
  - Goal: produce a great design
  - Simplify future development
  - Minimize complexity
  - Must sweat the small stuff
- Investment mindset
  - Take extra time today
  - Pays back in the long run



In order to go fast, we need to go well

ARHCITECTURE

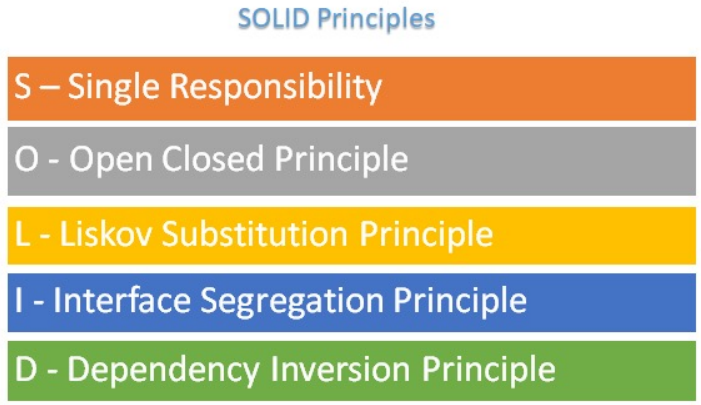
- The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.
- Expert developer's shared understanding of the system design.
- The set of design decisions that must be made early (the decisions that you wish you could get right early on)
- The decisions that are hard to change
- Shared understanding + Hard to change => the important stuff (whatever that is)

ECONOMICS VS CRAFTMENSHIP

- Economics:
  - We need to put less effort on quality so we can build more features for our next release
- vs. craftmenship:
  - We gotta make decent job, moral reasons

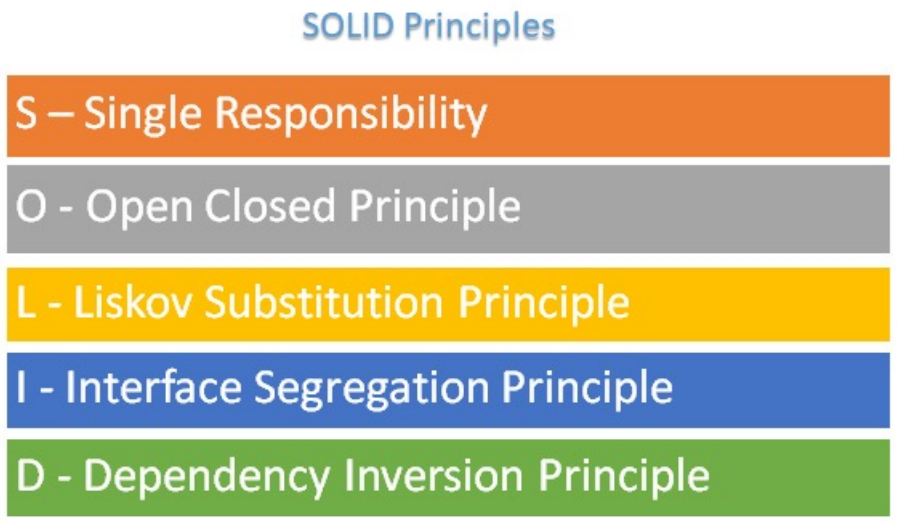
SOLID

- The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected
- The goal of the principles is the creation of mid-level software structures that:
  - Tolerate change,
  - Are easy to understand, and
  - Are the basis of components that can be used in many software systems.



SOLID

- `SRP:` Each software module has one, and only one, reason to change
- `OCP:` Software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code
- `LSP:` Build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another
- `ISP:` Avoid depending on things that you don't use
- `DIP:` High-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies



*module – just a source file

SRP: THE SINGLE RESPONSIBILITY PRINCIPLE

- It doesn't mean: that ~~every module should do just one thing~~
- Instead we split code up based on the social structure of the users using it
- 3 definitions, same meaning. Let's rephrase it:
  - A module should have one, and only one, reason to change
  - A module should be responsible to one, and only one, user or stakeholder
  - A module should be responsible to one, and only one, actor
- Example…

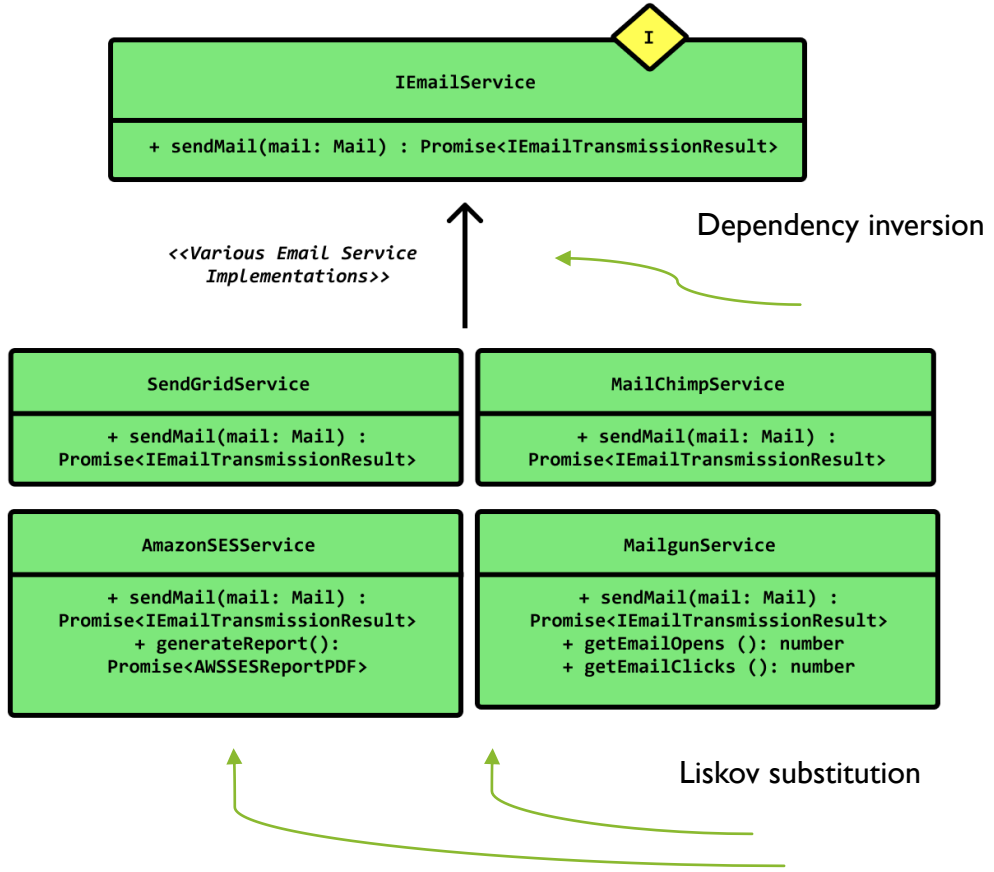
*module – just a source file (functions + data structures)

OCP: THE OPEN-CLOSED PRINCIPLE

- "A software artifact should be **open** for extension but **closed** for modification" (in 1988 by Bertrand Meyer)
  - Bob Martin: "OCP is the most important principle of object-oriented design"
- Higher level-components should be protected from changes to lower level components.
- How?
  - Properly separating the things that change for different reasons (the **S**ingle Responsibility Principle)
  - Organizing the dependencies between those things properly (the **D**ependency Inversion Principle)
  - Being able to swap implementations as long as the same interface is being depended on (the **L**iskov Substitution Principle)

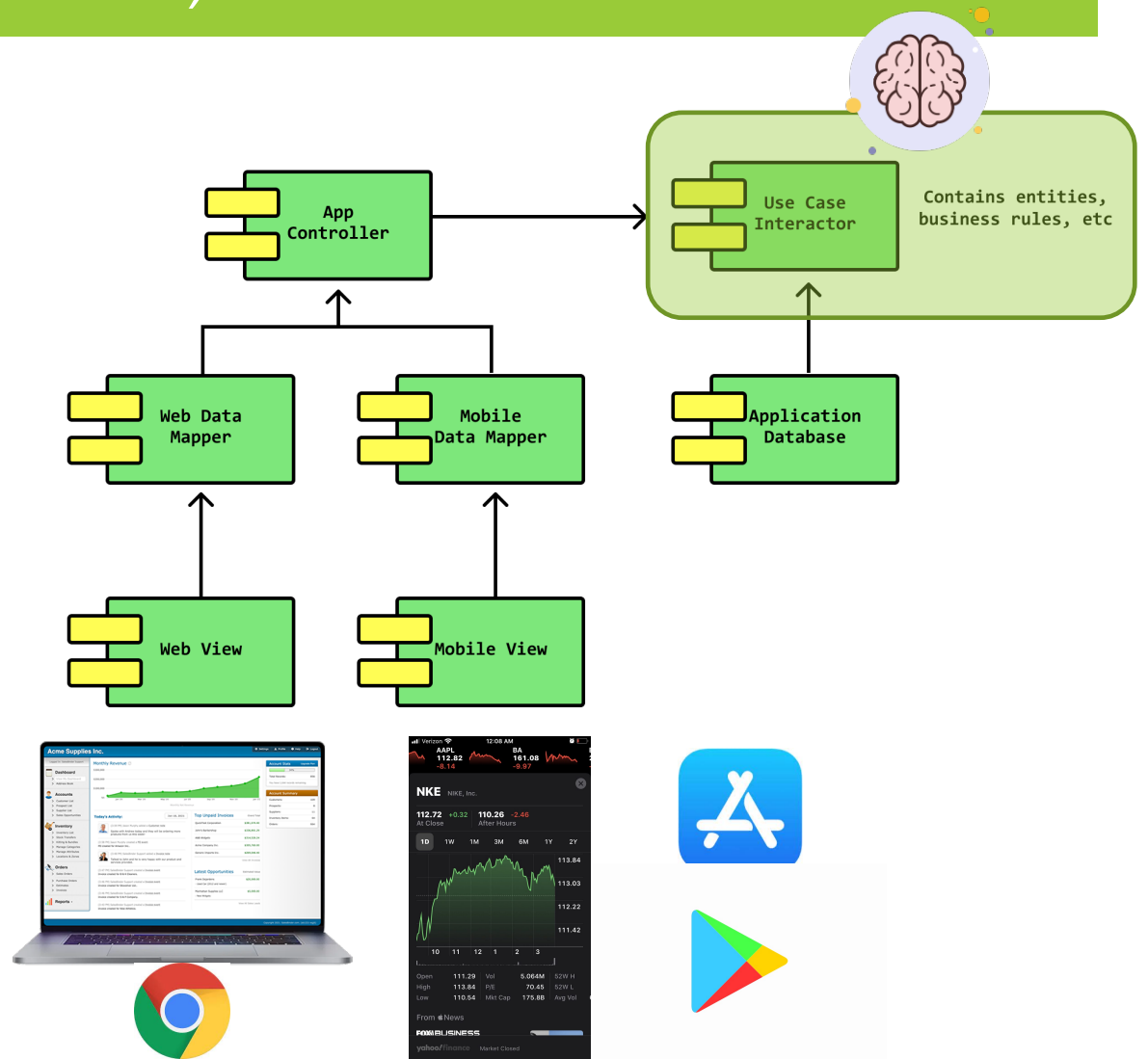
OCP: EXAMPLE (WRITING MODULES)

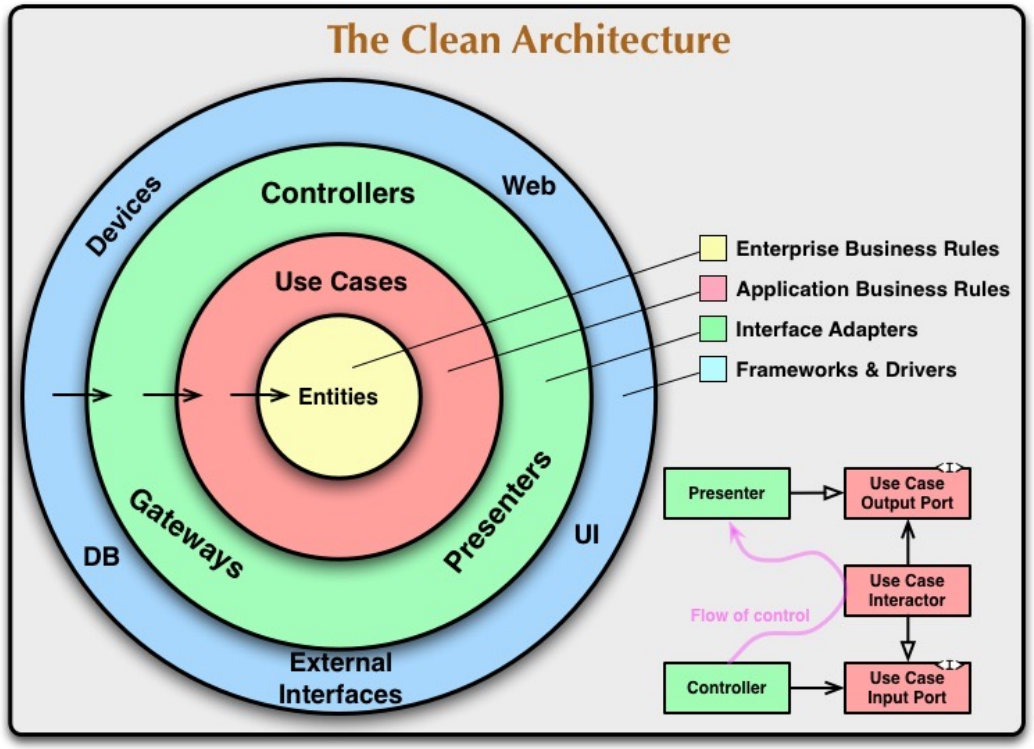
- Task 1: use SendGrid for sending emails.
- Task 2 (3months later): use MailChimp instead because SendGrid is too expensive.
- 2 options:
  - Changing and potentially breaking a lot of code
  - Or define IEmailService interface with 2 implementations
    - 4 implementations depend on IEmailService (DIP)
    - Implementation classes are interchangeable (LSP)



OCP: EXAMPLE (WRITING COMPONENTS)

- Higher level components – Use cases
- Low level components – Database, User interfaces
- Higher level-components are protected from changes to lower level components:
  - When we change Use Case component – it likely affect all lower level components (DB, UI…)
  - But if we change the Web UI, it's a less likely that that we need to change business logic




The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port

LSP: THE LISKOV SUBSTITUTION PRINCIPLE

- "What is wanted here is something like the following substitution property: If for each object **o1** of type S there is an object **o2** of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when **o1** is substituted for **o2** then S is a subtype of T"

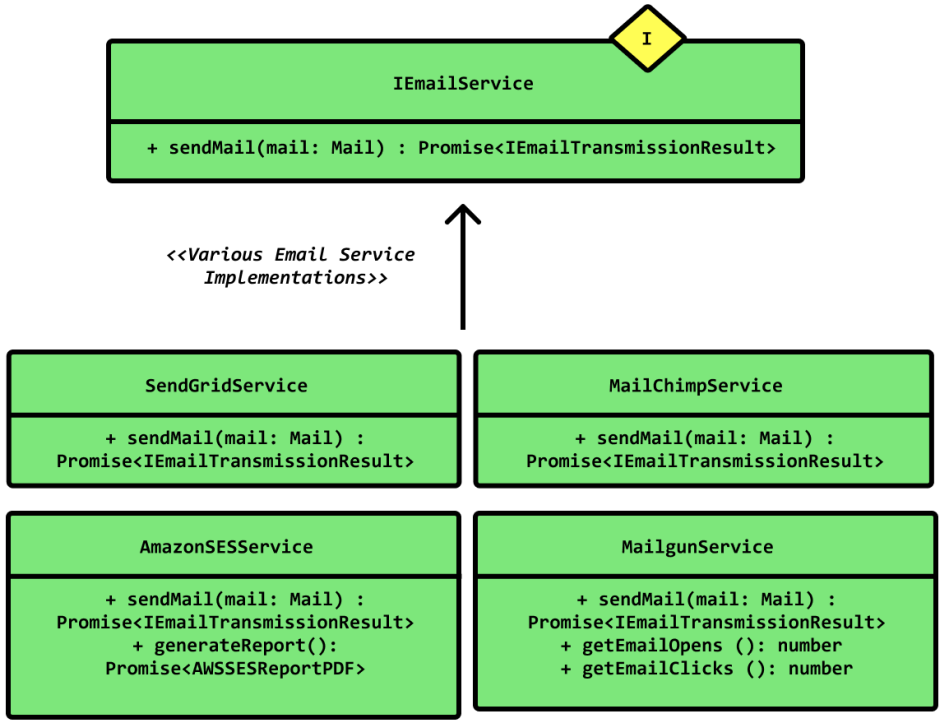
  by Barbara Liskov, 1988

- "To build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another"

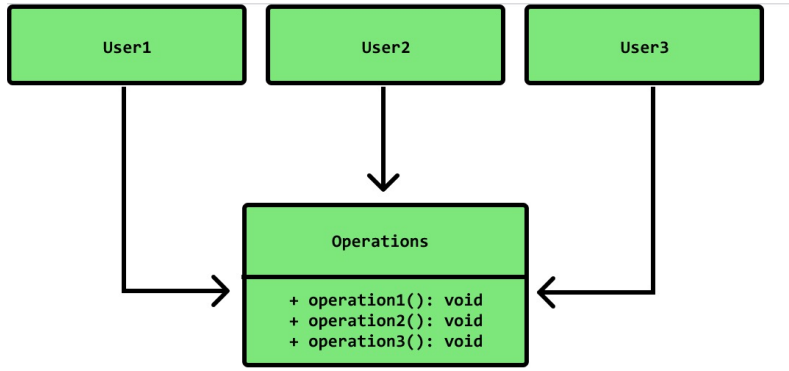
  by Bob Martin

- Or simply: "We should be able to *swap* one implementation for another"
- Mail example…

LSP EXAMPLE

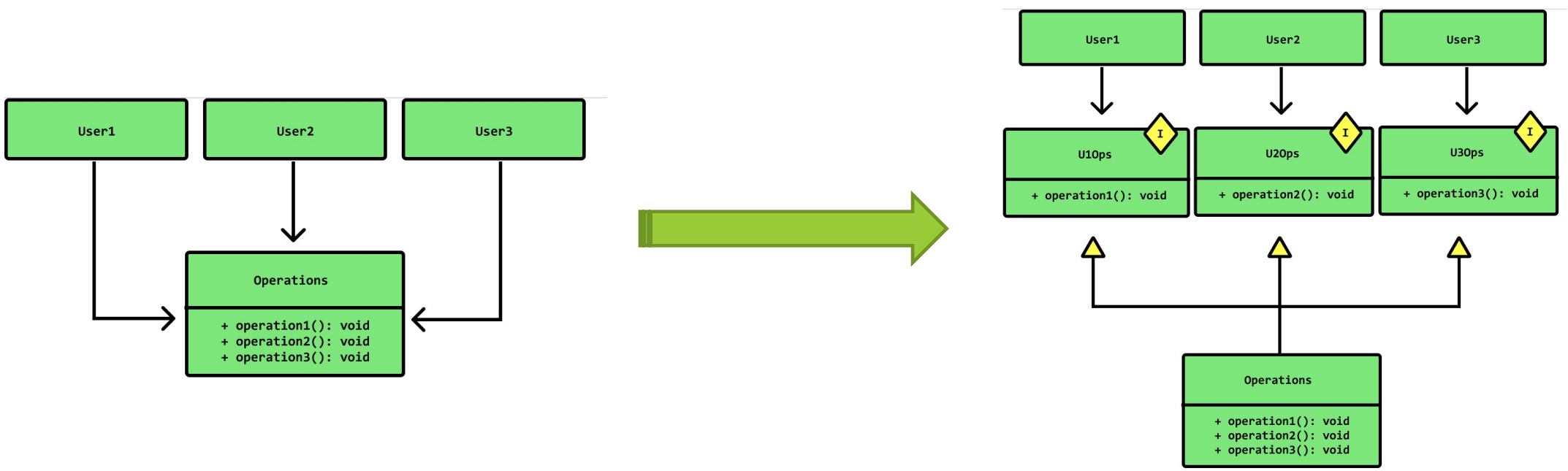

I
IEmailService
+ sendMail(mail: Mail) : Promise<IEmailTransmissionResult>
<<Various Email Service Implementations>>
SendGridService
+ sendMail(mail: Mail) : Promise<IEmailTransmissionResult>
MailChimpService
+ sendMail(mail: Mail) : Promise<IEmailTransmissionResult>
AmazonSESService
+ sendMail(mail: Mail) : Promise<IEmailTransmissionResult>
+ generateReport(): Promise<AWSSESReportPDF>
MailgunService
+ sendMail(mail: Mail) : Promise<IEmailTransmissionResult>
+ getEmailOpens (): number
+ getEmailClicks (): number

ISP: THE INTERACE SEGREGATION PRINCIPLE



- Prevent classes from relying on things that they don't need
  - Split up(segregate) the unique functionality into interfaces
  - Depend only on interfaces or abstract classes (Dependency Inversion Principle)
- In general, it is harmful to depend on modules that contain more than you need

SEPARATE INTERFACES

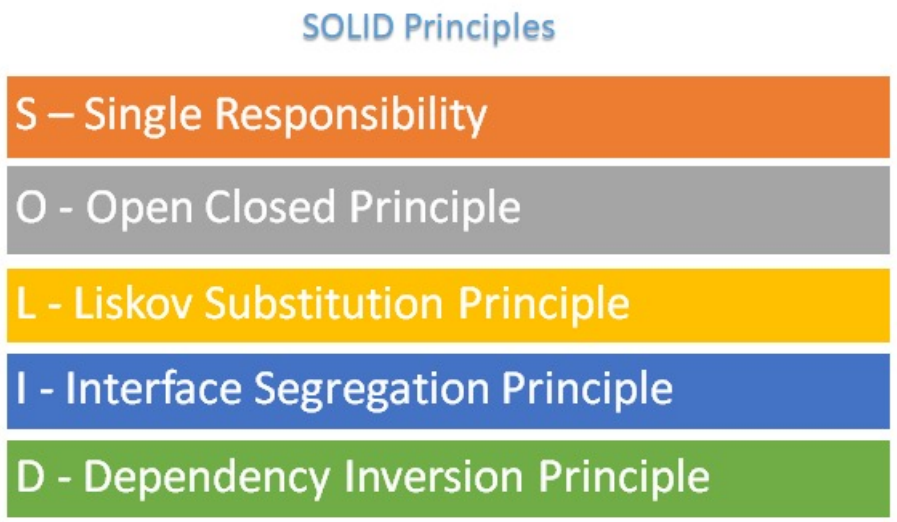


DIP: THE DEPENDENCY INVERSION PRINCIPLE

- Abstractions should not depend on details. Details should depend on abstractions.
  - Abstraction – interface, abstract class (contract, protocol)
  - Details – concrete class (implementation)
- Don't confuse Dependency *inversion* and Dependency *injection*

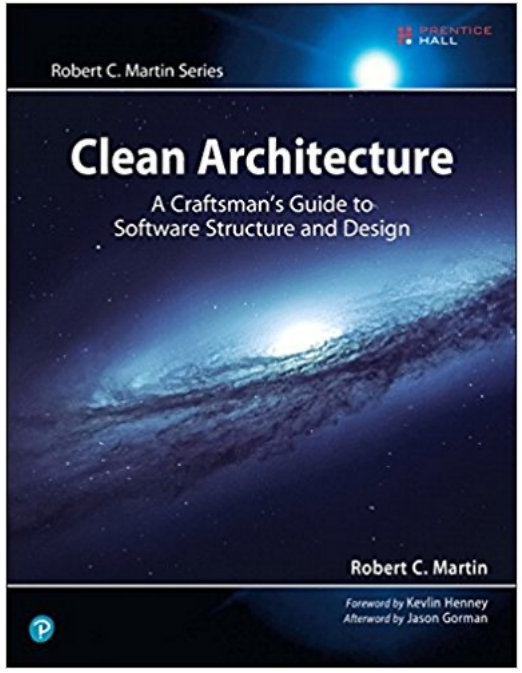
SOLID

- SRP: Each software module has one, and only one, reason to change
- OCP: Software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code
- LSP: Build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another
- ISP: Avoid depending on things that you don't use
- DIP: High-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies



BOOKS

- Clean Architecture: A Craftsman's Guide to Software Structure and Design
  - By Robert C. Martin



WHAT'S NEXT

- May 15 – Saturday, 12:30 Q&A, short seminar on regular expressions in Python, Labyrinth task
- May 19 – Wednesday, 14:30 Lecture
- May 22 – Saturday, 12:30 Q&A
- May 26 - Wednesday, 14:30 Lecture
- May 29 - Saturday, 12:30 Q&A, Labyrtinth task **deadline**