Effective Software Design

COMP 1531

Aarthi Natarajan

Week 07

“Why is design quality important?”

- Poor design leads to lower quality software systems, less functionality to clients, higher costs (for everyone), loss of business and misery for developers.
- Poor software quality costs more than $150 billion per year in U.S. and greater than $500 billion per year worldwide – Casper Jones
- Systems Sciences Institute at IBM found that it costs four- to five-times as much to fix a software bug after release, rather than during the design process
- A research from Cambridge University’s Judge Business School revealed that companies around the globe spend more than $300 billion debugging their software

Why does **Software Rot**?

We write bad code

Why do write bad code ?

- Is it because **do not know** how to write better code?
- Requirements change in ways that original design did not anticipate
- But changes are not the issue –
  - changes requires refactoring and refactoring requires time and we say we do not have the time
  - Business pressure - changes need to be made quickly – "quick and dirty solutions"
  - changes may be made by developers not familiar with the original design philosophy

Bad code, in fact slows us down

Why do software developers **fear change**?

- Reluctance to make changes to requirements after design and implementation
- Requires refactoring and lots of code needs to re-written
- Changes impact other parts of the system

But, one of the key principles of Agile Manifesto states…

- Change is a natural and inevitable part of software development life-cycle
- “**Welcome changing requirements**”

Design Smells

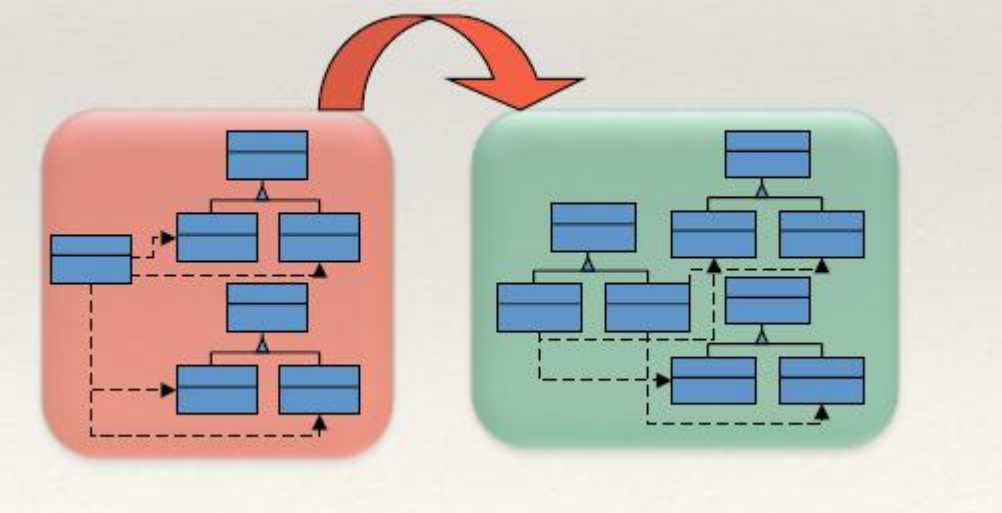
When software rots
it smells…

A design smell

- is a symptom of poor design
- often caused by violation of key design principles
- has structures in software that suggest refactoring

Refactoring

The process of restructuring (changing the internal structure of software) software to make it *easier to understand* and *cheaper to modify* without changing its *external, observable behaviour*



Design Smells (1)

Rigidity

- Tendency of the software being too difficult to change even in simple ways
- A single change causes a cascade of changes to other dependent modules

Fragility

- Tendency of the software to break in many places when a single change is made

Rigidity and fragility complement each other – aim towards minimal impact, when a new feature or change is needed

Design Smells (2)

Immobility

- Design is hard to reuse
- Design has parts that could be useful to other systems, but the effort needed and risk in disentangling the system is too high

Viscosity

- Software viscosity – changes are easier to implement through ‘hacks’ over ‘design preserving methods’
- Environment viscosity – development environment is slow and in-efficient

Opacity

- Tendency of a module to be difficult to understand
- Code must be written in a clear and expressive manner

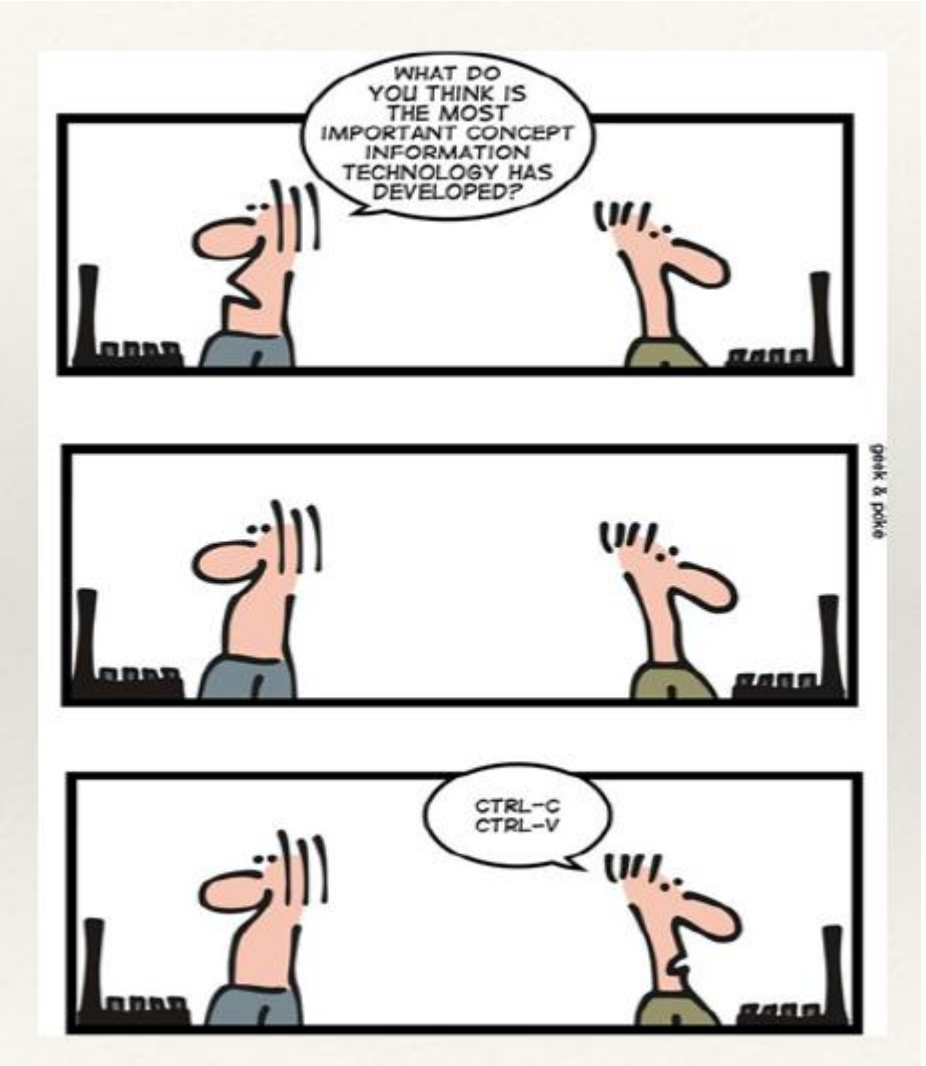
Design Smells (3)

Needless complexity

- Contains constructs that are not currently useful
- Developers ahead of requirements

Needless repetition

- Design contains repeated structures that could potentially be unified under a single abstraction
- Bugs found in repeated units have to be fixed in every repetition



“Applying design principles is the key to creating high-quality software

What do we mean by “design principles”?

“Design principles are key notions considered fundamental to many different software design approaches and concepts.”

- SWEBOK v3 (2014)

"The critical design tool for software development is a mind well educated in design principles"

- Craig Larman

Characteristics of Good Design

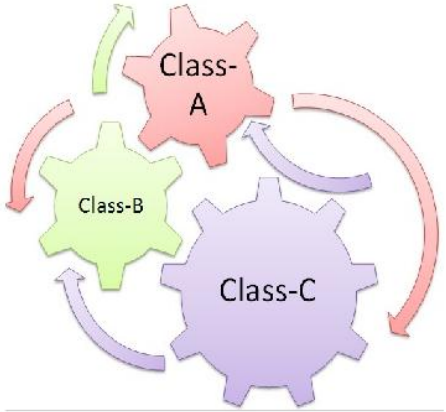
The design quality of software is characterised by

1. Coupling
2. Cohesion

Good software aims for building a system with loose coupling and high cohesion among its components so that software entities are:

- Extensible
- Reusable
- Maintainable
- Understandable
- Testable

Coupling

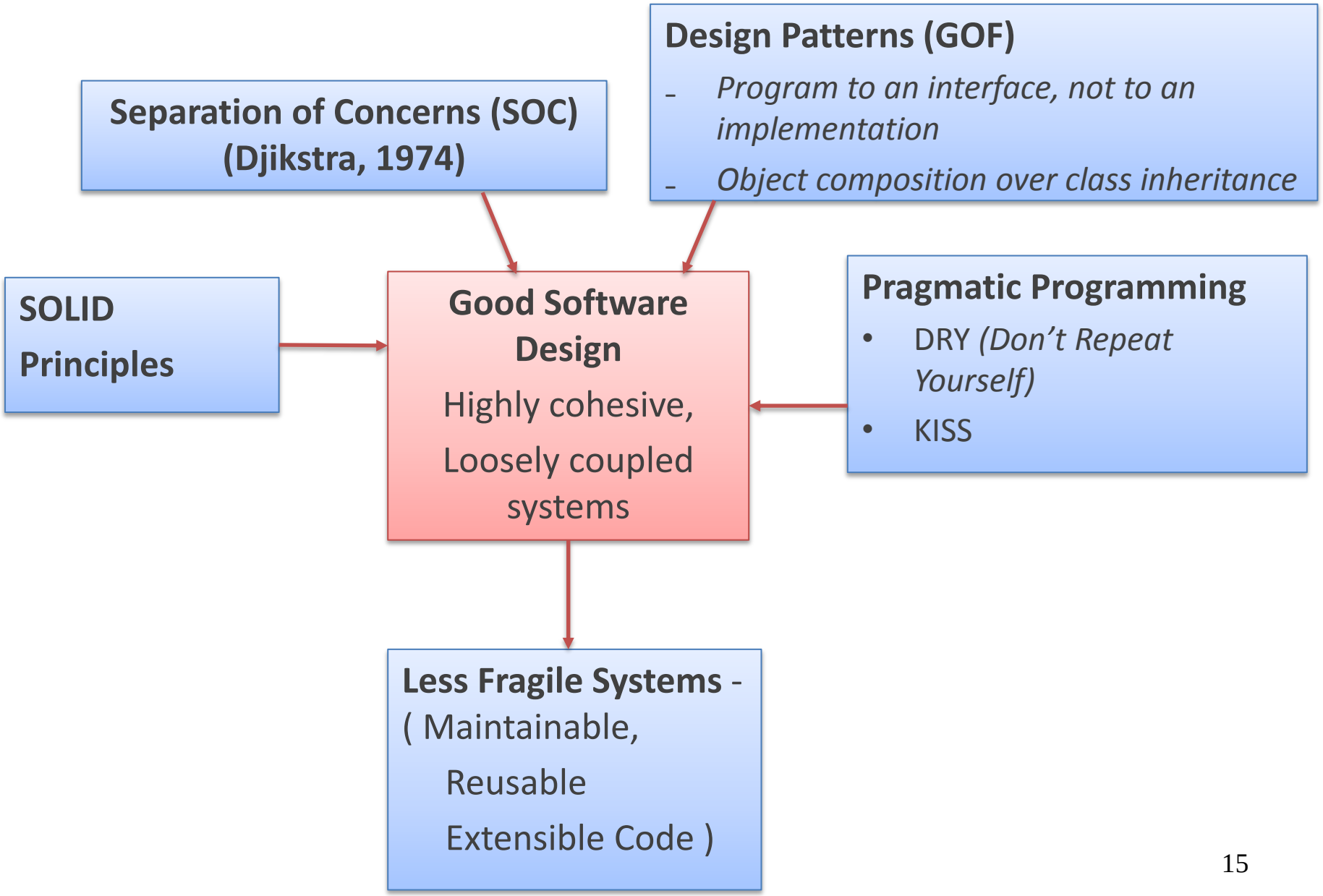


- Is defined as the degree of interdependence between components or classes
- High coupling occurs when one component **A** depends on the internal workings of another component **B** and is affected by internal changes to component **B**
- High coupling leads to a complex system, with difficulties in maintenance and extension…eventual software rot
- Aim for loosely coupled classes - allows components to be used and modified independently of each other
- But "zero-coupled" classes are not usable – striking a balance is an art!

Cohesion

- The degree to which all elements of a component or class or module work together as a functional unit
- Highly cohesive modules are:
  - much easier to maintain and less frequently changed and have higher probability of reusability
- Think about
  - How well the lines of code in a method or function work together to create a sense of purpose?
  - How well do the methods and properties of a class work together to define a class and its purpose?
  - How well do the classes fit together to create modules?
- Again, just like zero-coupling, do not put all the responsibility into a single class to avoid low cohesion!

Several Design Practices…One Goal



When to use design principles

- Design principles help eliminate design smells
- But, don't apply principles when there no design smells
- Unconditionally conforming to a principle (just because it is a principle is a mistake)
- Over-conformance leads to the design smell – needless complexity

Applying SOLID principles to achieve High Cohesion and Low Coupling

SOLID Principles

A set of five guiding design principles to avoid design smells:

- **S**RP – Single Responsibility Principle
- **O**CP – Open Closed Principle
- **L**SP – Liskov Substitution Principle
- **I**SP – Interface Segregation Principle
- **D**IP – Dependency Inversion Principle

Single Responsibility Principle

"A class should have ***one*** reason to change"

- Cohesion says: how good a reason the elements of a module have to be in the same module
- Cohesion and SRP: the forces that cause the module to change

Rationale behind SRP

- Rationale behind SRP:
  - Changes in requirements -> changes in responsibilities
  - A 'cohesive responsibility' represents a single axis of change -> a class should have only one responsibility
- A class with several responsibilities:
  - Creates unnecessary couplings between the those responsibilities
  - Changes to one responsibility may impair the class's ability to meet the others.
  - Leads to fragile designs that break in unexpected ways when changed.

SRP analogy with a water tap

A tap with 2 valves : 1 for hot water, 1 for cold water.

- If we turn the hot water valve: flow up, temp up
- If we turn the cold-water valve: flow up, temp down

What is the problem?

- What if you wanted to change the degree of movement to alter the flow control?
- With the current design – this would impacts the temperature too, but you do not want to the behaviour of the temperature too change!!

SRP analogy with a water tap

We want to achieve following things using valves:

(a) control flow (b) control temperature

A better design:

Two separate valves, with one just controlling flow and the other controlling temperature

SRP Misconception

- **Note:**
  - Single responsibility ≠ Single method in a class!

- One function can invoke several other functions, but it should not be responsible for how these functions are implemented
- Consider this example…

```
class Reporter(object):

    def email_report_hours(self,email, time_period,emp_id):

        report_data = self.get_report_data(time_period, emp_id)
        body = self.format_report(report_data)
        self.send_email(email, body)
```

Violation of SRP – (1)

Mixing of business logic and work orchestration

```
class Reporter(object):

    # This class knows too much about how the report is generated, formatted and emailed
    # This class has many reasons to change i.e. many responsibilities
    # e.g., if the business logic behind the report changes, then the class is changed
    # e.g., if the configuration to the email server changes, the class is changed

    def email_report_hours(self,email, time_period,emp_id):

        report_data = self.get_report_data(time_period, emp_id)
        body = self.format_report(report_data)
        self.send_email(email, body)

    def get_report_data(self,time_period,emp_id):
        # Open connection to database
        # Prepare a SQL query based on business logic
        # Run the SQL query and parse the result set
        print("Generating the report data")

    def format_report(self,report_data):
        print("Formating the report in PDF")
        return "formatted_report"

    def send_email(self,email,body):
        print("Configuring smtp server...sending report to:" + email)
```

A better design that conforms to SRP

```
class Reporter(object):

    # This class is only responsible for the orchestration of the process to email a report

    def email_report_hours(self,email, time_period,emp_id):
        report_data = ReportReader.get_report_data(time_period, emp_id)
        body = ReportFormatter.format_report(report_data)
        EmailService.send_email(email, body)

# Decompose responsibilities into separate independent classes
class ReportReader:
    def get_report_data(time_period,emp_id):
        # Open connection to database
        # Prepare a SQL query
        # Run the SQL query and parse the result set
        print("Generating the report data")

class ReportFormatter:
    def format_report(report_data):
        print("Formating the report in PDF")
        return "formatted data"

class EmailService:
    def send_email(email,body):
        print("Configuring smtp server...sending report to:" + email)
```

Violation of SRP – (2)

Grouping of business rules and persistence control

- These two responsibilities should never be mixed
- Business rules change frequently
- Persistence does not change often

```
class Employee(object):

    # This class groups business rules and persistence

    def calculate_pay(self):
        # some business logic to calculate pay

    def save(self):
        # code to persist(store) employee details
```

Is SRP violated?

```
class Modem(object):
    def call(self, number):
    def disconnect(self):
    def send_data(self,data):
    def recv_data(self):
```

Do we need to de-couple the responsibilities ?

```
class ConnectionManager(object):
    def call(self,number):
    def disconnect(self):

class DataTransmitter(object):
    def send_data(self,data):
    def recv_data(self):
```

- A responsibility is an axis of change only if changes occur
- If there is no symptom of change, no need to decouple responsibilities

Advantages of SRP

One of the fundamental design principles, simple to state, yet difficult to get it right. If implemented correctly, SRP helps to achieve "highly cohesive" classes that have:

Readability:

- Easier to focus on one responsibility and you can identify the responsibility

Reusability:

- The code can be re-used in different contexts

Testability:

- Each responsibility can be tested in isolation
- When a class has encapsulates several responsibilities, several test-cases are required

Consider this example…

```
class ReportFormatter:
    'A class to format a report into PDF format'

    def format_report(self,report_data):
        print("Formating the report in PDF")
```

What if you had to format the report in several different ways, HTML, PDF…?

Open Closed Principle

"Software entities (classes, modules, functions) should ***be open for extension*** but ***closed for modification***"

Open Closed Principle

- Modules that satisfy OCP have two primary attributes:
  - *Open for extension*: As requirements change, the module can be extended with new behaviours to adapt to the changes
  - *Closed for modification:* Extending the behaviour of the module must not require changing the original source, or binary code of the module.
- OCP reduces rigidity
  - a change does not cause a cascade of related changes in dependent modules
- Our previous example does not comply with OCP - cannot use a different formatter without modifying the function's code

"How do we write software that is ***open for extension*** but ***closed for modification***"

- Using abstraction and dynamic binding
- abstractions are implemented as abstract base classes, that are fixed yet, represent an unbounded group of possible behaviours.
- The unbounded group of possible behaviours (or the extensions) are provided by possible derived classes (sub-classes)

Complying with OCP:

To make our previous example comply with OCP

- Create `ReportFormatter` as an abstract class
- Define a new derived class for each type of formatting method and override the `format_report` method

```
class ReportFormatter(ABC):

        @abstractmethod
        def format_report():
                pass

class PDFFormatter(ReportFormatter):
        def format_report(self,report_data):
                print("formatting for pdf")
                return "pdf formatted data"

class HTMLFormatter(ReportFormatter):
        def format_report(self,report_data):
                print("formatting for html")
                return '<HTML><BODY>......report_data </BODY></HTML>'

class Reporter(object):

    # This class is only responsible for the orchestration of the process to email a report

    def email_report_hours(self,email, time_period,emp_id):
        report_data = ReportReader.get_report_data(time_period, emp_id)
        formatter = HTMLFormatter()
        body = formatter.format_report(report_data)
        EmailService.send_email(email, body)
```

Our revised work orchestration class, Reporter

```
class Reporter(object):

    'A class that orchestrates the work to email a report'

    def email_report_hours(self,email, time_period,emp_id):

        #Generate the report data
        report_data = ReportReader.generate_report_data(time_period, emp_id)

        # Create a new HTML formatter and invoke the format_method
        formatter = HTMLFormatter()
        formatted_data = formatter.format_report(report_data)

        # Email the report
        EmailService.send_email(email, formatted_data)
```

Output:

```
Formating the report in HTML
<HTML><BODY>.....report_data </BODY></HTML>
```

Conformance to OCP

Not Easy

- A skill gained through experience by knowing the users, industry to be able to judge the various kinds of changes
- Educated guesses could be right or wrong. If wrong, you loose time

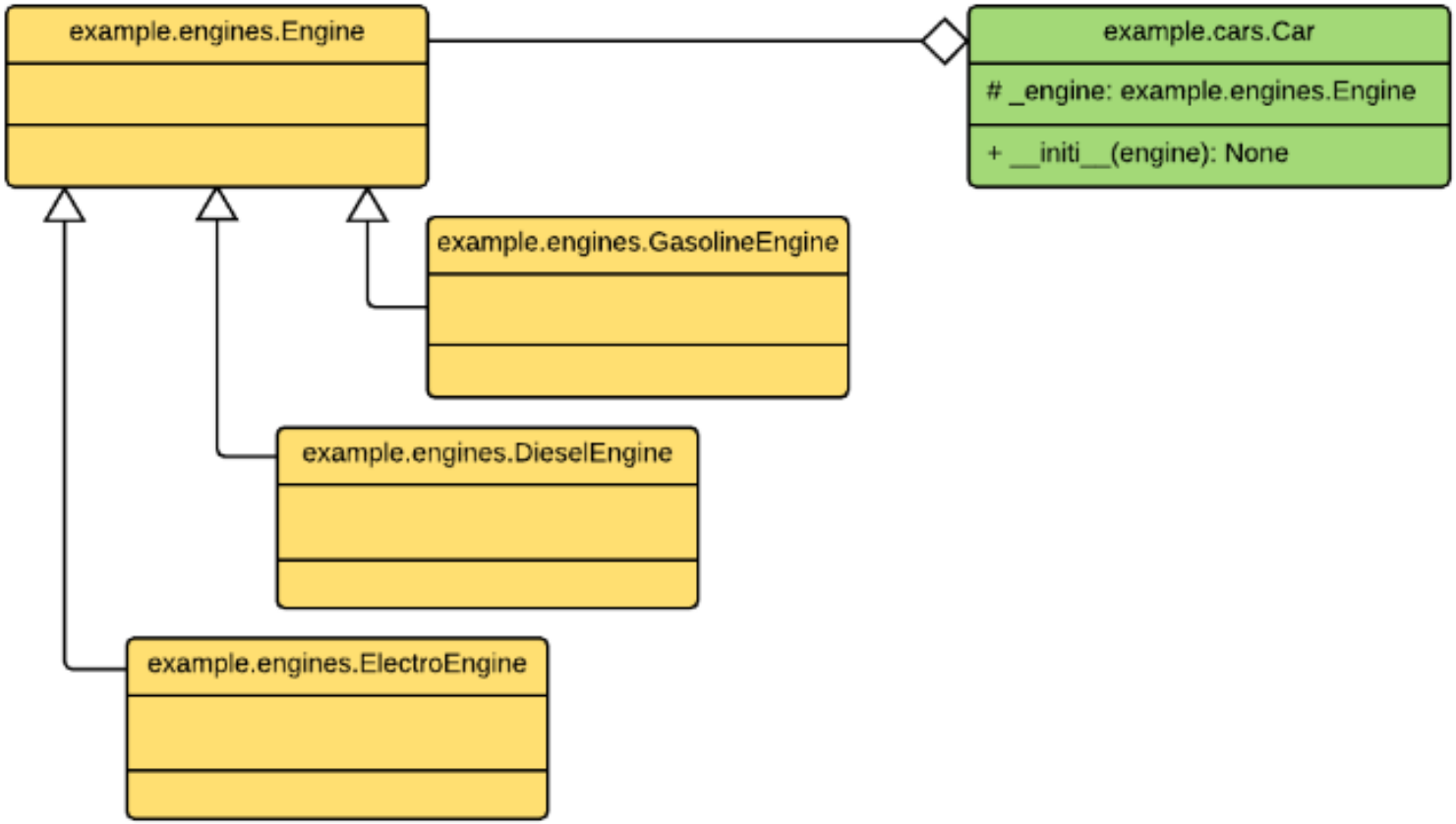
Expensive:

- Abstractions increase the complexity of software design
- Takes development time and effort to create the appropriate abstraction
- Apply OCP only when it is needed for the first time

Yet yields great benefits:

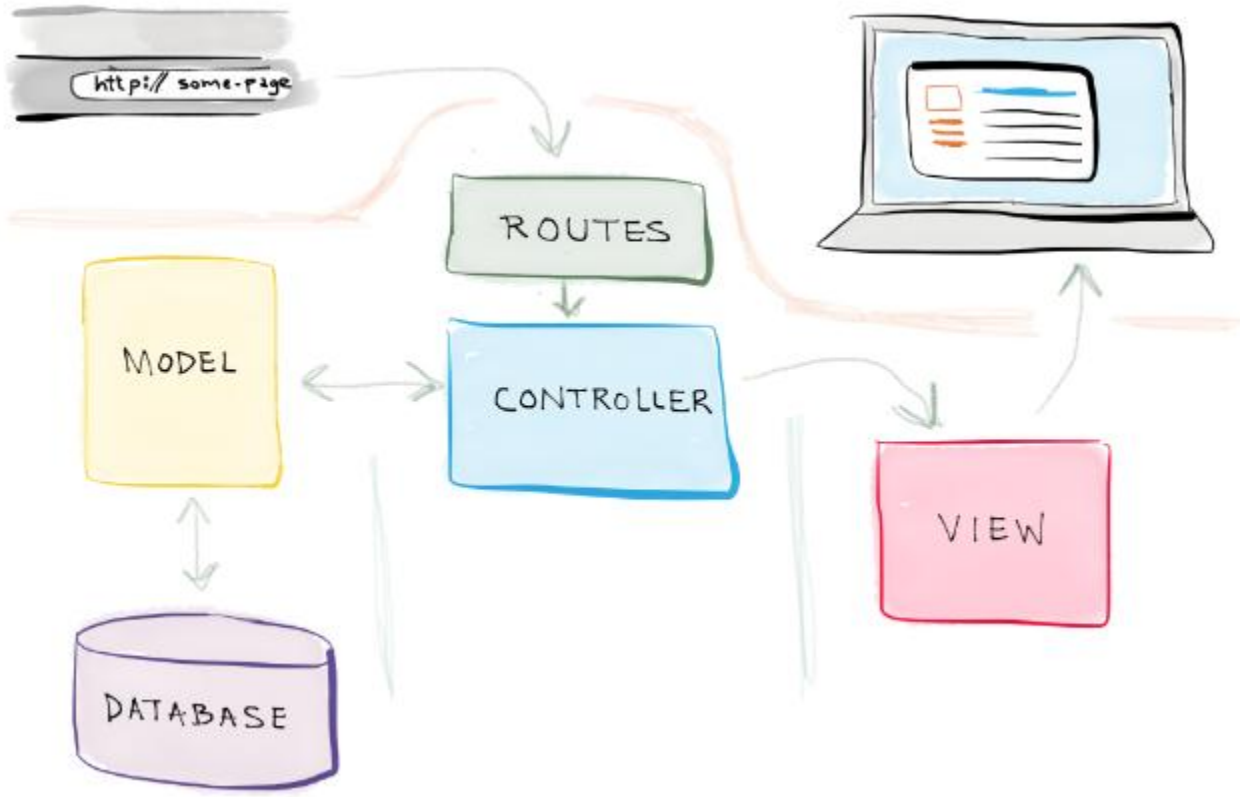
- Flexibility, Reusability and Maintainability

Class Exercise: Design a set of classes that implement the OCP and DIP for the class diagram below

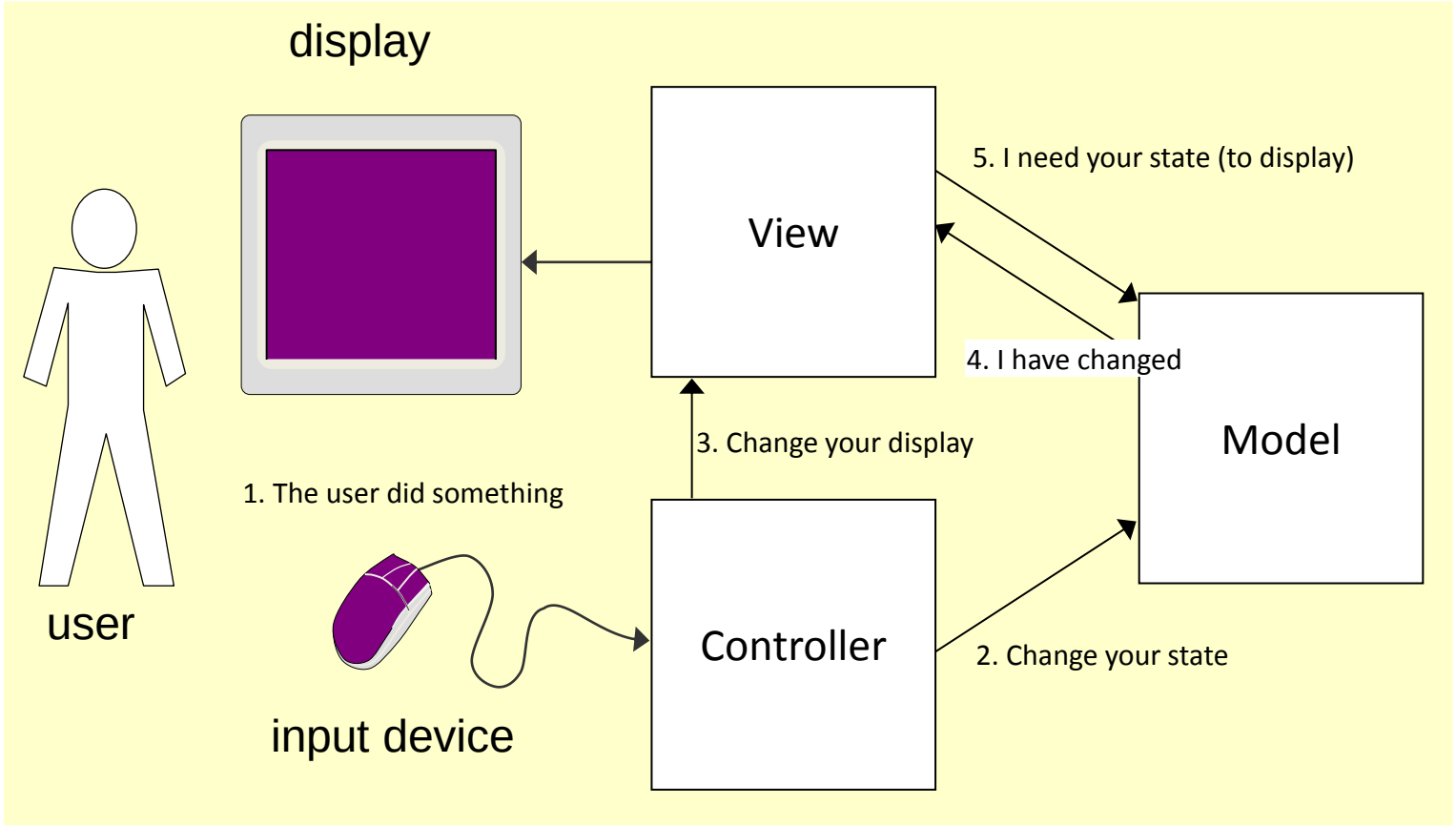


Separation of Concerns Using MVC (1) (Model-View-Controller)

A software architectural pattern that decouples data access, application logic and user interface into three distinct components



Separation of Concerns Using MVC (2)
(Model notifies View)

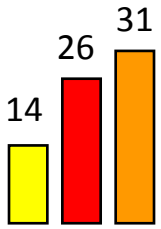


View

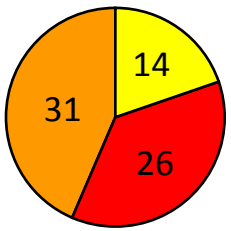
- This is the presentation layer provides the interaction that the user sees (e.g a web page).
- View component takes inputs from the user and sends actions to the controller for manipulating data.
- View is responsible for displaying the results obtained by the controller from the model component in a way that user wants them to see or a pre-determined format.
- The format in which the data can be visible to users can be of any ‘type‘ such as HTML or XML depending upon the presentation tier.
- It is responsibility of the controller to choose a view to display data to the user.

Model: array of numbers [14, 26, 31]

➔ Different Views for the same Model:



versus



Model

- Holds all the data, state
- Responds to instructions to change of state (from the controller)
- Responds to requests for information about its state (usually from the view),
- Sends notifications of state changes to “observer” (view)

Controller

- Glue between user and processing (Model**)** and formatting (View) logic
- Accepts the user request or inputs to the application, parses them and decides which type of Model or View should be invoked

Benefits:

- Abstraction of application architecture into three high-level components (Model, View, and Controller). Model has no knowledge of the View that is provided to the user.
- Supports multiple views of the same data on different platforms at the same time
- Enhances testability

Weakness:

- Complexity
- Cost of frequent update – an active model that undergoes frequent changes could flood the views with update requests

Useful resources

Bob Martin's SOLID Principles of Object Oriented Design:
https://www.youtube.com/watch?v=TMuno5RZNeE

https://code.tutsplus.com/tutorials/solid-part-1-the-single-responsibility-principle--net-36074

https://code.tutsplus.com/tutorials/solid-part-2-the-openclosed-principle--net-36600