(CS360) Fundamentals of Software Engineering

Overview and design principles

Lecture 1

Dr. Qobiljon Toshnazarov

Outline

New Uzbekistan University

- Administrivia
  - About instructor
  - Course logistics
  - Syllabus overview
- Introduction
  - General program design principles
  - Coding standards
  - Object oriented principles

Course instructor

New Uzbekistan University

- Instructor
  - Name: **Qobiljon Toshnazarov**
  - Email: q.toshnazarov@newuu.uz
  - OH: Mondays 14:00–18:00, UCA 218
- Academic background
  - (2024) PhD in Energy Engineering, KENTECH
  - (2020) MS in ECE, Inha University
  - (2018) BS in CSE, Inha University in Tashkent

Course instructor

New Uzbekistan University

- Professional experience (part of it)
  - (2024 ~ present) Senior lecturer at New Uzbekistan University
  - (2022 ~ present) Software engineer at Toptal
  - (2018 ~ 2024) Researcher at Intelligent Mobile Computing Lab

- Courses taught
  - Fundamentals of Software Engineering @ New Uzbekistan University
  - Object Oriented Programming @ New Uzbekistan University
  - Artificial Intelligence @ New Uzbekistan University
  - Application Programming with Java @ Inha University
  - Computer Networks @ Inha University
  - Wireless Communications @ Inha University
  - Data Structures @ KENTECH
  - Data Science @ KENTECH

Course instructor

New Uzbekistan University

Toptal Developers

Kobiljon Toshnazarov

Verified Expert in Engineering

Back-end Developer

Tashkent, Tashkent Province, Uzbekistan

Toptal member since May 5, 2022

EXPERTISE

Coding | Machine Learning | Technical Writing | Algorithms | Data Visualization | Web Development | Artificial Intelligence | Cloud Engineering | Twitter Bootstrap | Visual Studio Development | Python | Django | REST API | Linux | Database | Miro Development | Healthcare Data Science Sector

BIO

Kobiljon is an engineering professional known for his exceptional critical thinking, problem-solving abilities, and

Course logistics

New Uzbekistan University

- Communication
  - Communication method: email
- Course materials
  - Lecture slides
  - Laboratory manual
  - Assignments
  - Textbook (e-versions)
  - Google doc updated weekly: **tiny.cc/cs360**
  - Also uploaded on **HeRo LMS** platform
- Weekly assignments
  - Assignments are submitted via **HeRo LMS** platform

Course materials:
tiny.cc/cs360

Schedules of exams & HW

New Uzbekistan University

WEEK	TOPIC
1	-
2	-
3	Assignment 1
4	Assignment 2
5	Assignment 3
6	Assignment 4
7	Assignment 5
8	Midterm exams
9	-
10	Assignment 6
11	Assignment 7
12	Assignment 8
13	Assignment 9
14	Assignment 10
15	-
16	Final exams

⇦ We are here!

Topics in syllabus (overview)

New Uzbekistan University

WEEK	TOPIC
1	Overview and design principles
2	Design documentation and patterns
3	Software architecture, HTTP and REST
4	Async functions
5	Testing
6	User interfaces
7	-
8	-
9	Maintenance
10	Code improvement
11	Distributed systems & security
12	Continuous development (CD)
13	Engineering equitable software
14	Estimation and productivity
15	-
16	-

⇦ We are here!

Late policy for assignments

New Uzbekistan University

Your work is late if it is *not* turned in by the deadline.

- 10% will be deducted for late HW assignments turned in within 24 hours after the due date
- HW assignments submitted more than 24 hours late will receive a zero
- If you're worried about being busy around the time of a HW submission, **please plan ahead and get started early**

Grading policy

New Uzbekistan University

- Assignments: **30%**
- Midterm exam: **30%**
- Final exam: **40%**

SE overview and design principles

What is software engineering?

New Uzbekistan University

- Software engineering (SE) encompasses the tools and processes that we use to design, construct and maintain programs over time.

Good code is necessary but not sufficient

New Uzbekistan University

- Developing software is a ***systems enterprise***
  - There are many stakeholders
  - How to determine the requirements?
  - How to design code for:
    - Reusability
    - Readability
    - Scalability and other factors?
  - How to organize the development process?
  - How to make sure you've built the thing right?
  - How to make sure you've built the right thing?

SE includes tools and processes

New Uzbekistan University

- The answers to those questions will depend on things like:
  - Size of the team
  - Size of the product
  - Longevity of the product
- There's no one "right" way; there are always tradeoffs.
- But there are best practices, which we will expect you to follow.

Learning objectives for this course:

New Uzbekistan University

- By the end of this course you will
  - Be able to define and describe the phases of the software engineering lifecycle.
  - Be able to explain the role of key processes and technologies in modern software development.
  - Be able to productively apply instances of major tools used in elementary SE tasks.
  - Design and implement a portfolio-worthy software engineering project in a small team environment that can be showcased to recruiters.

Technology

New Uzbekistan University

In our tutorials we use:

- TypeScript + Python as implementation languages
- React / VueJS for web pages (or some other tool with SSR is fine as well)
- Some library for design elements (Plugins of TailwindCSS, Frameworks like Chakra UI, etc.)
- WebStorm, PyCharm, Visual Studio Code
- + Git, etc...

Academic integrity

New Uzbekistan University

- Students must work individually on all homework assignments.
- We encourage you to have high-level discussions with other students in the class about the assignments, however, we require that when you turn in an assignment, it is only your work. That is, copying any part of another student's assignment is strictly prohibited.
- Keep in mind: if you’re stealing someone else's work, you’re failing the class.
- You are also responsible for protecting your work from being copied. If someone uses your work, with or without your permission, you fail the class.

Academic integrity

New Uzbekistan University

- You are free to reuse small snippets of example code found on the Internet (e.g. via stack overflow) provided that it is attributed.
- Same goes with ChatGPT as well. If you use code provided by ChatGPT, you must do so by attributing it.
- If you are concerned that by reusing and attributing that copied code it **may appear that you didn't complete the assignment yourself**, then please raise a discussion with the instructor.
- If you are in doubt whether using others' work is allowed, you should **assume that it is NOT allowed** unless the instructors confirm otherwise.

General Program Design Principles

The Challenge: Controlling Complexity

- Software systems must be comprehensible by humans
- Why? Software needs to be maintainable
  - continuously adapted to a changing environment
  - Maintenance takes 50–80% of the cost
- Why? Software needs to be reusable
  - Economics: cheaper to reuse than rewrite!

The challenge: controlling code complexity

- Software systems must be comprehensible by humans
  - How? Make programs readable.
  - How? Make programs flexible.
  - How? Make programs modular.

The biggest obstacle: coupling

- Two pieces of code are *coupled* if a change in one demands a change in the other.
- A coupling represents an agreement between the two pieces of code.
  - They may agree on:
    - names
    - order (e.g. of arguments)
    - meaning (e.g. meaning of data)
    - algorithms
- The more two pieces of code are coupled, the harder they are to understand and modify: you have to understand both to understand either of them.

There's a fancy word for this: *connascence* (meaning "born together")

More coupling means less readability, less modifiability

Five general-purpose principles

Five general principles
1. Use good names
2. Design your data
3. One method/one job (single responsibility)
4. Don't repeat yourself
5. Don't hardcode things that are likely to change

Good idea: make a sticky note with this list and keep it on your laptop screen.

Principle 1. Use good names

- The name of a thing is a first clue to the reader about what the thing means.
  - Often, it's the only clue ☹️
- Use good names for
  - Constants
  - Variables
  - Functions/methods
  - Data types

Good names for constants

- Replace magic numbers with good names

```
let salesPrice = netPrice * 1.06
```

```
const salesTaxRate = 1.06
let salesPrice = netPrice * salesTaxRate
```

Where did that 1.06 come from?

Oh, it's the sales tax? Are there many occurrences of that 1.06 in your code? (Probably!) Will the sales tax rate ever change? (Probably!)

Let's fix it!

But use good names!

```
int a[100]; for (int i = 0; i <= 99; i++) a[i] = 0;
```

Even if you search for 100, you'll miss the 99!

```
int SIZE = 100;
int a[SIZE]; for (int i = 0; i <= SIZE-1; i++) a[i] = 0;
```

~~int ONE_HUNDRED = 100;~~
~~int a[ONE_HUNDRED]; ...~~

No.....!

But use GOOD names!

Good names for variables and types

```
var t : number
var l : number
```

```
var temp : number
var loc  : number
```

```
var temp : Temperature
var loc  : SensorLocation
```

What do these variables mean?

Better names would give the reader a clue.

Does 'temp' mean 'temporary', or 'temperature', or something else?

Good names for the data types solves the problem.

Good names for functions and methods

```
function checkLine () : boolean
```

```
function isLineTooLong () : boolean
```

What are you checking it for? Length? Illegal Syntax? or what?

Ahh, now we know!

Good names for functions and methods

- Use noun-like names for functions or methods that return values, e.g.

```
let c = new Circle(initRadius)
let a = c.diameter()
```

- not:

```
let a = c.calculateDiameter()
```

- Reserve verb-like names for functions or methods that perform actions, like

```
table1.addItem(student1,grade1)
```

Your workplace should have coding standards for things like this.

Principle 2. Design your data

- You need to do three things:
  1. Decide what part of the information in the "real world" needs to be represented as data
  2. Decide how that information needs to be represented as data
  3. Document how to interpret the data in your computer as information about the real world

Example:

- Assume I am wearing a red shirt, and I've decided I need to represent that fact in my program.
- How should I represent that in my program?
- I need to represent the color red. Possibilities:
  - "red" (English text)
  - "RED" (English text)
  - "Lāla" (Hindi, according to Google)
  - #ff0000

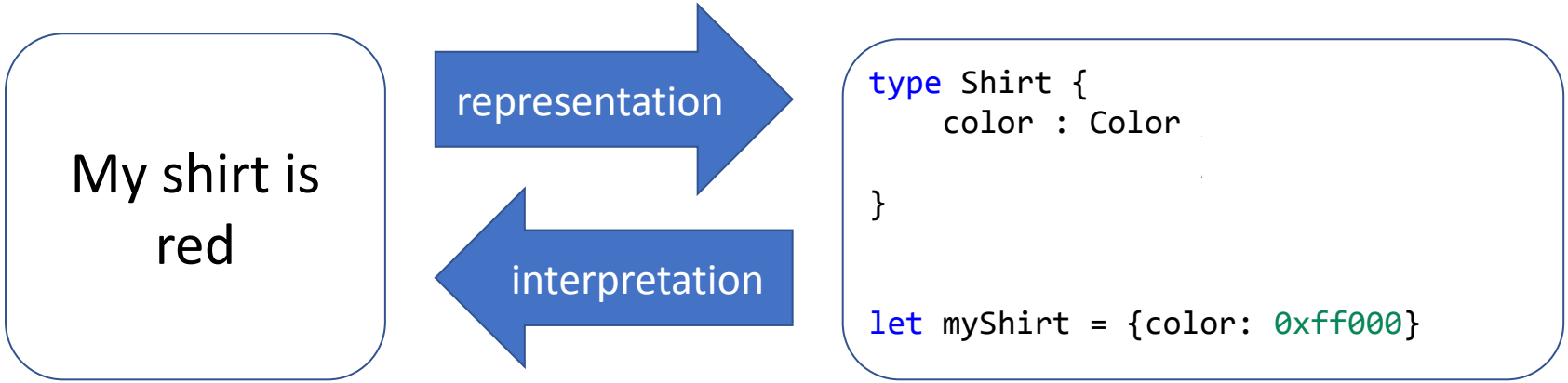
Example:

- And of course we also need to represent my shirt.
- In that representation, we have to represent its color.
- Here's one of many possibilities:

```
type Shirt {
    color : Color   // the color of the shirt
}

let myShirt = {color: 0xff000} // my shirt
```

The big picture



- How do we know that these are connected?
- Answer: we have to write down the interpretation
- In our Typescript infrastructure, we do that with the comments.

Principle 3: One method/one job

- Each class, and each method of that class, should have one job, and only one job
- If your method has more than one job, split it into 2 methods. Why?
  - You might want one part but not the other
  - It's easier to test a method that has only one job
- You call both of them if you need to.
  - or write a single method that calls them both
- Same thing for classes.

The fancy name for this is "The Single Responsibility Principle". You can use this if you want to impress your coop interviewer.

Principle 4: Don't repeat yourself

- If you have some quantity that you use more than once, give it a name and use the name.
- That way you only need to change it in one place
- And of course you should use a good name
- If you have some task that you do in many places, make it into a procedure.
- If the tasks are slightly different, turn the differences into parameters.

We saw this before with the sales tax and array bound examples.

This is called "Single Point of Control"

A real example

```
function testequal <T> (testname: string, actual: T, correct: T){
    it(testname, function (){
        assert.deepEqual(actual, correct)
    })
}

describe('tests for count_local_morks', function () {
    testequal('empty crew', count_local_morks(ship1), 0)
    testequal('just Mork', count_local_morks(ship2), 1)
    testequal('just Mindy', count_local_morks(ship3), 0)
    testequal('two Morks', count_local_morks(ship4), 2)
    testequal(
        'drone has no Morks',
        count_local_morks(drone1),
        0
    )
})
```

Think of how much typing this saves!

Plus, if I ever need to change what testequal does, I can do it all in one place.

Principle 5:
Don't hardcode things that are likely to change

- "No magic numbers" and "Don't Repeat Yourself" are already examples of this.
- General strategy: If there something that might change, give it a name
  - if it's not already a "thing", refactor to make it a "thing"
  - many strategies for this; let's look at one of them

Example

- Imagine we are computing income tax in a state where there are four rates:
  - One on incomes less than $10,000
  - One on incomes between $10,000 and $20,000
  - One on incomes between $20,000 and $50,000
  - One on incomes greater than $50,000
- You might write something like in the following slide

You might write something like

```
function grossTax(income: number): number {
    if ((0 <= income) && (income <= 10000)) { return 0 }
    else if ((10000 < income) && (income <= 20000))
    { return 0.10 * (income - 10000) }
    else if ((20000 < income) && (income <= 50000))
    { return 1000 + 0.20 * (income - 20000) }
    else { return 7000 + 0.25 * (income - 50000) }
}
```

This also violates one function/one job:
it finds the right bracket AND calculates the appropriate tax

- What might change?
  - The boundaries of the tax brackets might change
  - The number of brackets might change

Not so terrible..

Ouch! Do you really want to dive into that conditional?

So let's represent our data differently

```
// defines the tax bracket for income lower < income <= upper.
// if upper is null, then lower < income  (no upper bound)
type TaxBracket = {
    lower: number,
    upper: number | null,
    base : number
    rate : number
}

let brackets : TaxBracket[] = [
    {lower:0,     upper:10000, base:0, rate:0},
    {lower:10000, upper:20000, base:0, rate:0.10},
    {lower:20000, upper:50000, base:1000, rate:0.20},
    {lower:50000, upper: null, base:7000, rate:0.25}
]
```

The brackets are now a "thing". All the data is in one place, so we have a Single Point of Control

And now it's easy to rewrite our function

```
// defines the incomes covered by a bracket
function isInBracket(income:number, bracket:TaxBracket) : boolean {
    if (bracket.upper == null)
    { return (bracket.lower <= income) }
    else
    { return ((bracket.lower <= income) && (income < bracket.upper))}
}

function taxByBracket(income:number,bracket:TaxBracket) : number {
    return bracket.base + bracket.rate * (income - bracket.lower)
}

function grossTax2 (income:number, brackets: TaxBracket[] ) : number {
    return taxByBracket(income,income2bracket(income,brackets))
}
```

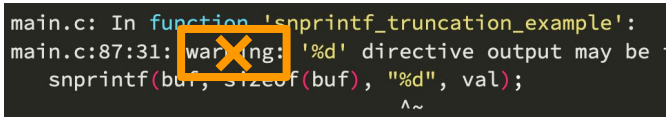
And we are back to one function/one job.

Coding standards

Coding standards

New Uzbekistan University

- Reduce warnings
  - If possible reduce them to zero

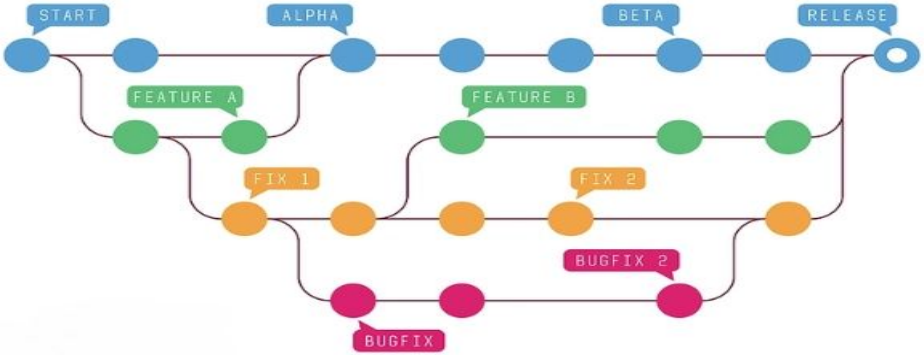


- Include project requirements in the codebase
  - e.g., package.json, requirements.txt, makefile, etc.

Coding standards

New Uzbekistan University

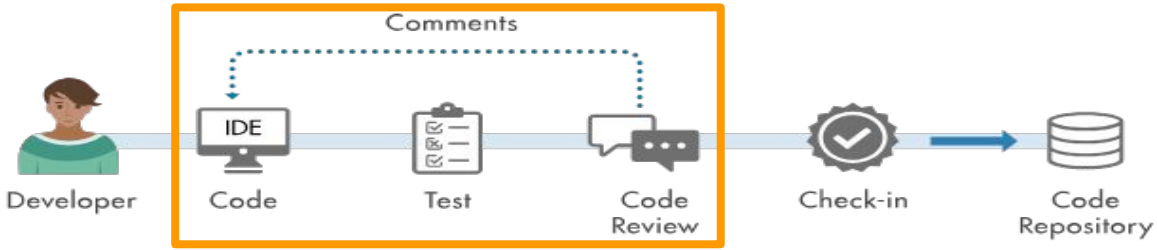
- Use of version control system (VCS) is mandatory
  - For instance, you can use Git as your project's VCS
  - "The palest of ink is better than the best memory"
  - Note: the code in the version control system should always compile, so, do not break the build!



Coding standards

New Uzbekistan University

- Re-read code!
  - Read code from good programmers to learn and benefit.
  - More eyes will help make better quality.
  - It’s best to do code reviews in writing, a simple email can suffice.



Correctness

New Uzbekistan University

- Correct is better than fast.
- Simple is better than complex.
- Prefer clarity over cuteness.
- Write code for people first, then machines.

Optimization

New Uzbekistan University

- Avoid premature optimization
  - i.e., trying to make the program run faster before ensuring the codebase works correctly.
  - “Premature optimization is the root of all evil”.
  - Do not optimize prematurely!
- Remember: “It is far easier to make a correct program fast than to make a fast program correct”.

Coding style

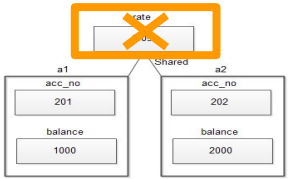
New Uzbekistan University

- Use “const” proactively, i.e., immutability
  - (for languages that support this)
- Handle variables properly
  - Avoid magic numbers / strings / etc, i.e., hardcoded numeric/character/etc. literals in the code.
  - Declare variables as locally as possible.
  - Always initialize variables (at least to null)

Coding style

New Uzbekistan University

- Minimize global and shared data
  - Do not use “static” keyword in your code until explicitly asked (or required)
  - Thread-local - avoid sharing resources between threads whenever possible.



- Do not expose internal information from an entity that provides an abstraction
  - a.k.a., information hiding

SW engineering goals

New Uzbekistan University

Your software should be:

- Expandable
- Maintainable
- Understandable
- Stable
- And preferably implemented, tested, and reviewed by more than one person…

Object oriented principles

Single responsibility

Open-closed

Liskov substitution

Interface segregation

Dependency inversion

SOLID principles

New Uzbekistan University

- Set of guidelines or **best practices** for object-oriented programming that help to create software that is maintainable, flexible, and scalable.

- Term “SOLID” is an acronym for five different principles focusing on a different aspect of SW development:
  - Single responsibility principle (SRP)
  - Open-closed principle (OCP)
  - Liskov substitution principle (LSP)
  - Interface segregation principle (ISP)
  - Dependency inversion principle (DIP)

Single responsibility principle

New Uzbekistan University

- One class should have **only one responsibility.**
  - Implementation: divide the responsibilities of a class into smaller, more focused classes.
- It means that one class should have only one reason to change.
- Leads to low coupling between classes and makes the code more maintainable and scalable.

Example: single responsibility

New Uzbekistan University

```
class Customer {
private:
 string name;
 int id;
 vector<Item> items;
 float totalAmount;
public:
 void setName(string name);
 string getName();
 void setId(int id);
 int getId();
 void addItem(Item item);
 void removeItem(Item item);
 float calculateTotalAmount();
 string generateInvoice();
};
```

Interface of a `Customer` class.

Has several responsibilities, such as:
(1) maintaining customer details,
(2) calculating the final bill, and
(3) generating an invoice.

Responsibility: maintaining customer details

```
class CustomerDetails {
private:
 string name;
 int id;
public:
 void setName(string name);
 string getName();
 void setId(int id);
 int getId();
};
```

Responsibility: calculating the final bill

```
class BillingCalculator {
private:
 vector<Item> items;
public:
 void addItem(Item item);
 void removeItem(Item item);
 float calculateTotalAmount();
};
```

Responsibility: generating an invoice

```
class InvoiceGenerator {
public:
 string generateInvoice(
  CustomerDetails customerDetails,
  BillingCalculator billingCalculator
 );
};
```

Open-closed principle

New Uzbekistan University

- The open/closed principle states that a class must be both open and closed.
  - Open: means it has the ability to be extended.
  - Closed: means it cannot be modified other than by extension.
- i.e. we would like to be able to modify the behaviour without touching its source code

Open-closed principle

New Uzbekistan University

- Once a class has been approved for use after having gone through code reviews, unit tests, and other qualifying procedures, you don't want to change the class very much, just extend it.

- Changing base class can complicate all derived classes.

Example: open-closed

New Uzbekistan University

```
class Shape {
public:
 virtual double area() = 0;
};

class Rectangle : public Shape {
private:
 double width;
 double height;
public:
 Rectangle(double width, double height);
 double area() override;
};

class Circle : public Shape {
private:
 double radius;
public:
 Circle(double radius);
 double area() override;
};
```

Existing `Shape` class hierarchy.

With OCP, one that wants to modify this hierarchy must only do so by extending the existing classes.

Create a new class `Triangle` that inherits from the Shape class and implements the area() method. In this way, we are adding new functionality **without modifying the existing code**.

```
class Triangle : public Shape {
private:
 double base;
 double height;
public:
 Triangle(double base, double height);
 double area() override;
};
```

Liskov substitution principle

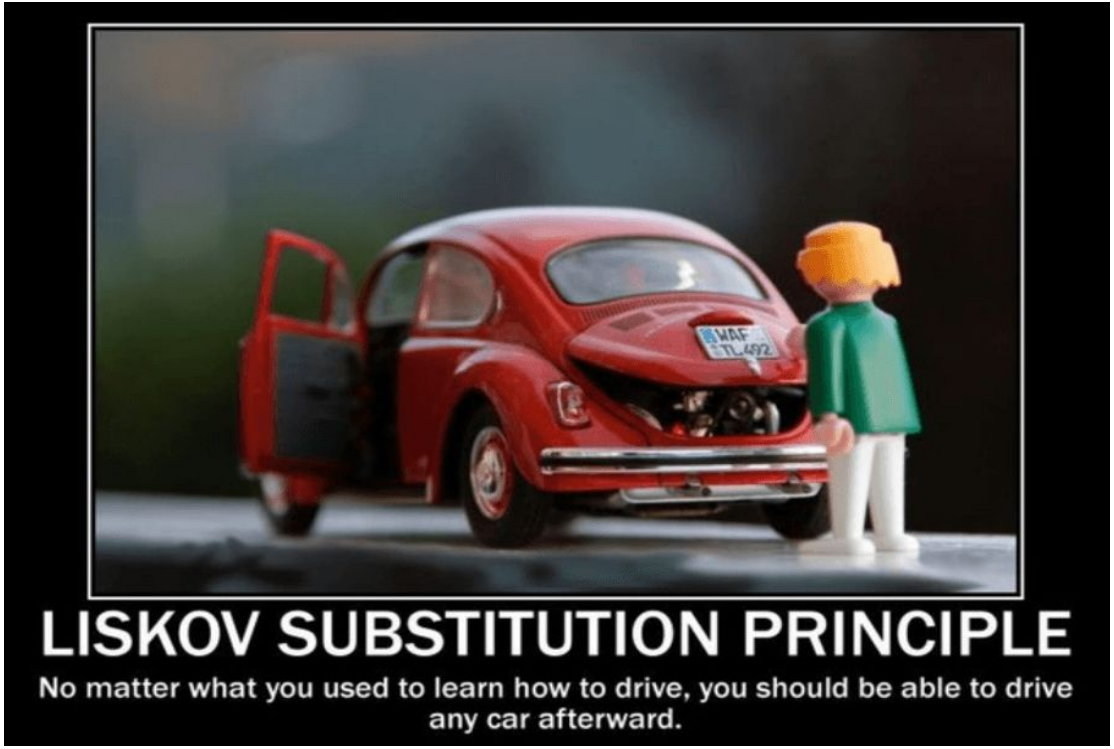
New Uzbekistan University

- If S is a subtype of T then objects of type T may be replaced with objects of type S

- Student is a subtype of Person, so anywhere I can have a Person, I can have a Student instead.

Liskov substitution principle

New Uzbekistan University



Example: Liskov substitution

New Uzbekistan University

```
class Bird {
public:
 virtual void fly() = 0;
};

class Eagle : public Bird {
public:
 void fly() override;
};

class Penguin : public Bird {
public:
 void fly() override; // throws exception
};

class Ostrich : public Bird {
public:
 void fly() override; // throws exception
};
```

`Bird` class hierarchy.

With LSP, all objects of base type may be replaced with objects of derived types.

Suppose you have following function that invokes base `Bird` class's `fly()` method:

```
void makeBirdFly(Bird& bird) {
 bird.fly();
}
```

With LSP, you should be able to replace objects of `Bird` with objects of any **derived** type (of `Bird`).

```
Eagle eagle;
Penguin penguin;
Ostrich ostrich;

makeBirdFly(eagle); // okay, eagle can fly
makeBirdFly(penguin); // error, penguin can't fly
makeBirdFly(ostrich); // error, ostrich can't fly
```

Example: Liskov substitution

New Uzbekistan University

To solve the issue in previous slide, we can set up the hierarchy more appropriately.

We can have a intermediate superclass called `FlyingBird` that represents the birds that can fly.

```
class Bird {
   public:
        virtual bool canFly() = 0;
};

class FlyingBird {
   public:
        bool canFly() override {
            return true;
        }
        virtual void fly() = 0;
};

class Crow : public FlyingBird {
   public:
        void fly() override {
            cout << "Crow is flying" << endl;
        }
};

class Penguin : public Bird {
   public:
        void canFly() override {
            return false;
        }
};
```

Interface segregation principle

New Uzbekistan University

- Client should not be forced to depend on interfaces that it does not use.
  - Implementation is somewhat similar to SRP for classes, but it is for interfaces.
  - Implementation: break down larger interfaces into smaller, more specific interfaces.
- In other words, a class should not have to implement methods that it does not need.

Example: interface segregation

New Uzbekistan University

```
class Printer {
public:
 virtual void printPDF() = 0;
 virtual void printHTML() = 0;
 virtual void printPlainText() = 0;
};
```

`Printer` interface, an abstract class.

Problem: client has to implement all methods to use even just one.

```
class Printable {
public:
 virtual void print() = 0;
};
```

Client can use following class to print PDFs.

```
class PDFPrintable : public Printable {
public:
 void print() override;
};
```

Client can use following class to print HTML.

```
class HTMLPrintable : public Printable {
public:
 void print() override;
};
```

Client can use following class to print text.

```
class PlainTextPrintable : public Printable {
public:
 void print() override;
};
```

Dependency inversion principle

New Uzbekistan University

- High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features.

- To achieve that, you need to introduce an **abstraction** that decouples the high-level and low-level modules from each other.

Dependency inversion principle

New Uzbekistan University

- High-level modules should not depend on low-level modules.
- Instead, both should depends on abstractions.
- Depend upon abstractions, and not upon concrete classes.

Example: dependency inversion

New Uzbekistan University

Problem: A high-level `Database` class depends on a low-level `MySQLConnector` class. So, high-level logic depends on a very specific database type.

What if you want to switch to a different database e.g., MongoDB?

We would have to **re-write the high-level code** due to changes in low-level code!

High-level class.

```
class Database {
public:
 void addUser(
   const string& name,
   const string& email
 ) {
   MySQLConnector connector;
   connector.connect();
   string query = "INSERT INTO users (name, email)"
                  + "VALUES ('" + name + "', '"
                  + email + "')";
   connector.query(query);
 }
};
```

Depends on a low-level class!!

Low-level class.

```
class MySQLConnector {
public:
 void connect();
 void query(const string& query);
};
```

Example: dependency inversion

New Uzbekistan University

Solution: the high-level `Database` class depends on an abstraction `IDatabaseConnector`, instead of concrete classes.

```
class Database {
public:
 Database(
   IDatabaseConnector& connector
 ) :connector_(connector) {}

 void addUser(
   const string& name,
   const string& email
 ) {
   connector.connect();
   string query = "INSERT INTO users (name, email)"
                + "VALUES ('" + name + "', '"
                + email + "')";
   connector.query(query);
 }
};
```

High-level class.

Depends on an abstraction.

Now, the low-level concrete classes can be be easily switched, **without affecting the high-level code.**

```
class IDatabaseConnector {
public:
 virtual void connect() = 0;
 virtual void query(
  const string& query
 ) = 0;
};
```

Abstraction.

Low-level classes:

```
class MySQLConnector
  : public IDatabaseConnector
```

```
class MongoConnector
  : public IDatabaseConnector
```

```
class PostgresConnector
  : public IDatabaseConnector
```

Unit testing

New Uzbekistan University

- Unit testing is a procedure used to validate that individual units of source code are working properly.
- Unit = smallest testable part of an application
  - For instance, in TypeScript smallest units are functions and classes
- Goal: isolate each part of the program and show that individual parts are correct.

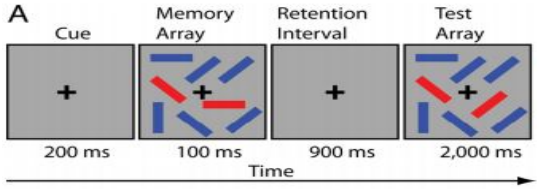
Wrap up

Multitasking and learning

New Uzbekistan University

- Distraction & divided attention: texting while driving?

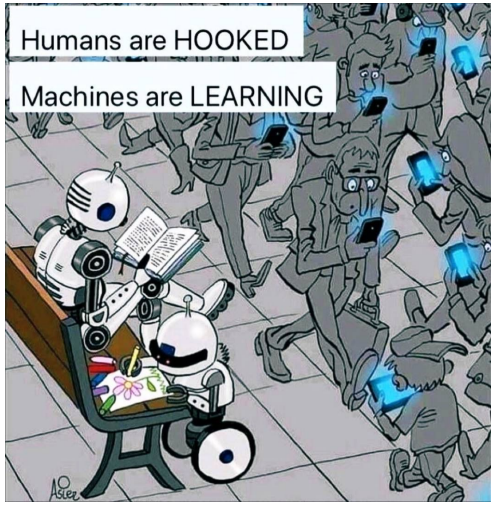
- Multitaskers are inferior in cognitive control: filtering irrelevant info and task switching (Ophir et al., 2009)



- Lower scores (w/ laptop use) of the user and nearby users (secondhand smoking) (Cepeda et al. 2013)

Notes on the use of LLMs for learning

New Uzbekistan University



- While you may use LLMs for learning purposes (e.g., understanding terminologies, concepts)
- Verify everything, and avoid blind use of LLMs to complete tasks!!!

Reference	Supporting Statement
[96]	ChatGPT suffers from generalization problems because it can not perform well on new and unseen issues or new problems.
[82]	Concerning accuracy, ChatGPT faces challenges in generating correct answers. Roughly 30% of them produced inaccurate solutions that remained unsolved even with feedback provided by leetcode.
[8]	and evolution to enhance LLMs' coding chatbot abilities to have an effective and powerful tool for programming tasks in computing education.
[10]	Another study reported that the explanations generated by Codex cover a significant portion (90%) of the code despite some inaccuracies (67.2% of explanation lines were correct). They also observed that in most cases, the erroneous lines contained only minor mistakes that an instructor or teaching assistant could quickly fix.
[14], [75]	On the contrary, some studies explore the quality of the code generated by Copilot and compare the programming code written by humans to see the differences between both. The team infers that Copilot's code quality is weaker than human-generated code.
[81]	ChatGPT requires more efficiency in utilizing information to generate accurate outputs.
[48]	Although ChatGPT appears to be excellent in various tasks, it is not adept at providing textbook
[28]	Around half of the correct solutions generated by ChatGPT contain complex and useless
[28]	ChatGPT can occasionally report incorrect codes as correct ones. Furthermore, it may not anticipate edge cases that could cause the code's functionality to fail in some circumstances, and it may not always
[57], [64]	Codex was worse on Parsons's problems; in around 50% of cases, Codex could not resolve the issues correctly. However, GPT-4 seems good at solving Persona problems, completing 96.7% of the tasks.
[50], [59], [83], [88]	MCQ-type questions containing coding snippets make the prompt more challenging for GPT models to understand. GPT-3 only correctly solved 199 problems out of 530 questions, and GPT-3.5 was observed to be more successful since this model solved 341 questions correctly. However, GPT-4 remained most capable in solving MCQ queries by solving 446 questions correctly, which reflects that GPT models improved noticeably over generations and GPT-4 is good at grading feedback and task synthesis.
[38]	Despite being discussed as more successful than GPT 3.5, GPT-4 still struggles with fine-grained formatting requirements for both the output and code.
[74]	The previous versions of LLMs like GPT-3 should not be directly handed over to students because code explanations of fewer quality outputs in the correctness of code descriptions.

FYI: Limitations of using LLMs as coding assistants for Computer Science students (Pirzado et al., 2024)

Thoughts on teaching

New Uzbekistan University

I will not show off by asking advanced questions during lecture.
I will not show off by asking advanced questions during lecture.
I will not show off by asking advanced questions during lecture.
I will not show off by asking advanced questions during lecture.
I will not show off by asking advanced questions during lecture.
I will not show off by asking advanced questions during lect
I will not show off by asking advanced questions during lect
I will not show off by asking advanced questions during lect
I will not show off by asking advanced questions during lect
I will not show off by asking advanced questions during
I will not show off by asking advanced questions during

answer advanced questions.


Redo
Undrln
Q & A
Find
J
C
V
B
N