

Software Engineering

Book: MIT 6.031 Software Construction

Install Gradle - Connecting to GitHub (learning how to use git is also part of this code)

Lecture 1:

- ❑ Google's Key Software Engineering Practices
- ❑ Static checking
- ❑ Testing

Google's Key Software Engineering Practices (Check out Henderson 2017 - end of the slides)

- Source code files contain a total of 2 billion lines of source code
- Most of their code is in the same unified source-code repository
- Git is a software used by many companies such as Google
 - Github is a company itself that provides a smoother use of Git
- Google uses Blaze → build system (we use Gradle)
- Build System, responsible for
 - Compiling
 - Linking
 - Running tests
- Unit testing in Java is called JUnit
- Don't get too attached to any specific language (C++, Java, Python, JavaScript..)
- Code is never static, it is always changing → Google for example changes/improves most of its code every year

Static Checking

- The main difference between python and java is defining variables
 - **Java** (explicitly defining variables): `int a = 3`
 - **Python** (does not define): `a = 3`
- **Java** is a **statically typed** language
 - The type of all variables are known before compiling
- **Python** is a **dynamically typed** language
 - Checking is deferred until runtime
- **Static typing is a form of static checking**
 - The bug is found automatically (with compiler errors) before the program runs
 - Examples, Wrong: syntax, names, number of arguments, argument types, return types
 - Traps in java:
 - **Integer division:** `5/2` does not return a fraction (or double), it returns an integer
 - **Integer overflow:** `int` and `long` types are **FINITE** sets of integers with max and min values

- **Special values in floating-point types:** (double for example) can have special values that are NOT real numbers e.g NaN (“Not a number”)
 1. Programming world \neq mathematical world
 2. You can assign double a = Double NaN
 3. $(2^{31}) - 1$ is the highest number in Java
- **Hacking vs. Engineering**
 - In the real world hacking is bad, long code, not commented, not documented
 - Engineering is good, it is constantly testing, commented, etc.
- **Testing**
 - A way to validate the code, to debug and find problems
 - Ways of testing:
 - **Verification** (mathematical way), test using a formal proof
 - **Code Review:** A human reviews your code
 - **Testing:** an automatic test made by you → you need to define what to test, define the inputs and the outputs.
 - Testing is very **hard**, you can't just test every possible combination
 - For example: for a 32-bit floating point multiply operation $a*b$ there are 2^{64} possible test cases (exhaustive testing)
 - **Haphazard testing** (randomly trying to find a bug) → not efficient
 - **Random or statistical testing:** doesn't work well for software. Software is not continuous so you may have boundary cases that are not representative of a random behaviour.
 - **Aim:**
 - When you are coding: goal is to make the program work
 - When you are testing: goal is to make it fail
 - **Test-First Programming**
 - When you write a function:
 1. Write a specification for the function
 2. Write tests that exercise the specification
 3. Write the actual code (if the code passes the test you are done)
 - Specification: describes the input and output behavior of the function
 - Iterative test-first programming: iteratively go through steps 1-3
 - **Choosing test cases by partitioning:**
 - Small individual tests that cover most of the program, these should run fast
 - Partition can also be called subdomain
 - Example: function with two inputs. Split the two inputs into subdomains. Choose one test case for each subdomain. The test case should represent the entire subdomain.
- **Example: BigInteger.multiply**
 - Create two BigIntegers (a and b) (inputs), multiply a times b, the output is a BigInteger.
 - BigInteger x BigInteger → BigInteger

- Input space is two-dimensional
- Partitions: (4 Subdomains) → this is a way to do it, there are other ways as well
 - a and b are both positive
 - a and b are both negative
 - a is positive, b is negative
 - a is negative, b is positive
- **Special Cases:**
 - 0, 1 or -1
 - It could also be that the implementer of BigInteger may try to use int or long internally when possible → making it faster. This could happen if a or b is small, and a or b is smaller than the Long.MAX_VALUE
 - Guess how the person implemented the code, and think of the mistakes they may have made.

Test Suite: collection of all test cases

Example: max(int a, int b)

- a and b are two arguments. Max() returns the greatest argument between the two
- Define the possible values of a and b (e.g if a and b are integers) → include as test cases:
 - Value of a: a = 0, a < 0, a > 0, a = minimum integer, a = maximum integer
 - Value of b: b = 0, b < 0, b > 0, b = minimum integer, b = maximum integer
- Partitions:
 - a < b
 - a = b
 - a > b
- Test Suite: 3 test cases
 - (a,b) = (1, 2) to cover a < b
 - (a,b) = (9,9) to cover a = b
 - (a,b) = (-5, -6) to cover a > b
- Test Values examples:

Test Values (<i>test cases</i>)	Test Classes (<i>subdomains</i>)
(1, 2)	a < b, a > 0, b > 0
(-1, -3)	a > b, a < 0, b < 0
(0, 0)	a = b, a = 0, b = 0
(Integer.MIN_VALUE, Integer.MAX_VALUE)	a < b, a = minint, b = maxint
(Integer.MAX_VALUE, Integer.MIN_VALUE)	a > b, a = maxint, b = minint

Including Boundaries in the Partition (practice these in ASSIGNMENT 1)

- Bugs often occur at **boundaries between subdomains**, for example:
 - **0** is a boundary between positive and negative numbers
 - **Maximum** and **minimum** types of numeric values (e.g int and double)
 - **Emptiness** (empty string, empty list, empty array)
 - **First** and **last** element of a **collection**
- **Common off-by-one mistakes:**
 - Using <=, instead of <
 - Initializing counter to 0, instead of 1
- **Always consider special cases**

- **Consider places of discontinuity**

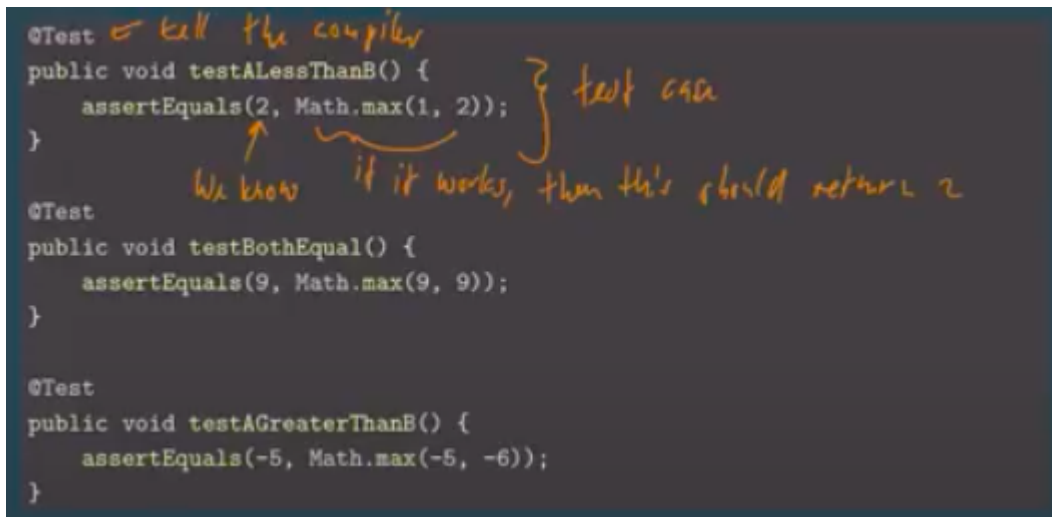
- E.g If an int variable grows beyond its maximum positive value, it becomes a negative number.

Two Extremes for covering the partition

- You can make a specific test for every possible combination that is legal. Or you can make test cases that cover multiple subdomains at the same time

Automated unit testing with JUnit (for Java)

- Unit test: tests an individual module in isolation
- It is a Java unit testing library
- Procedure:
 - Calls to module
 - Checks results using assertion methods like: “**assertEquals**”, “**assertTrue**”, “**assertFalse**”...
- Tag: tells the compiler that this is a unit test → **@Test**
 - Every test case has a tag
- Example: Math.max():



The image shows a screenshot of Java code for JUnit tests. The code is as follows:

```
@Test
public void testALessThanB() {
    assertEquals(2, Math.max(1, 2));
}

@Test
public void testBothEqual() {
    assertEquals(9, Math.max(9, 9));
}

@Test
public void testAGreaterThanB() {
    assertEquals(-5, Math.max(-5, -6));
}
```

Handwritten annotations in orange include:

- Next to **@Test**: "tell the compiler"
- Next to the first **assertEquals**: "test case"
- Below the first **assertEquals**: "We know if it works, then this should return 2"

- **assertEquals(arg1, arg2)** → the arg1 represents the answer that the test should give us. Arg2 is what we want to test.
- You have to import certain classes to use the JUnit test (find this in the GitHub classroom)

Documenting your testing strategy (examples in slide 56, 57):

- Short description
- An example
- Define boundary cases
- Define subdomains
- Define strategy
- Define the input parameters
- Define the output parameters
- Each test method should have a comment above it saying how its test case was chosen

Black box testing (you don't know what is inside the implementation)

- Test cases only from the specification
- Example: testing the methods like “multiply” or “max”

Glass box testing (you know what is in the code → more elaborate testing)

- Implementation selects different algorithms depending on the input → then you should partition according to those domains.

Coverage tool (we will use the one on Gradle) → how much your test cases cover your code → always aim for everything to be green

- Green: executed by the test suite
- Yellow: line containing a branch that has been executed in only one direction
- Red: line not yet covered → never executed in the test suite
- Categories of coverage (the more you cover the better):
 - **Statement coverage** → every statement is covered by a test
 - **Branch coverage** → if and while statements → cover both true and false directions
 - **Path coverage** → cover every possible combination of branches, all the paths in the program

Other definitions:

Integration test: combination of modules → testing the entire program (we won't cover this in this class).

Regression: introducing bugs when you are trying to fix other bugs

Regression testing: running all tests after every change

Lecture 2: Introduction to software engineering and Management

Overview of important topics in software engineering (Broader perspective)

What is software engineering?

- It is a process
- Designing, creating and deploying software
- Quality needs to be assured
- Tests need to be performed
- Responsibilities have to be clear → good communication
- Requirements must be met and deadlines must be set
- Finance and administration

Production grade source code → free of obvious bugs, tested

Software Team:

- Product Owner
- Project Manager
- Software architect
- Developers
 - Back end



- Try to solve the business problem
 - Use database
- Front end
 - Mainly HTML, CSS, javascript → designers
- Full-stack (both front and back end)
- Experience designers
- QA or testers responsible for quality assurance
- Business Analyst → uncover what users want and need from the product → improve the product

We will look at:

- How to design a software
- How to organize development
- How to use design patterns and architecture
- Writing modular software with API's

Tools for software teams:

- Databases, IDE's, Versioning Systems, Platforms and Frameworks such as Spring/apache, Build Automation Systems, Style Guides, Formatting tools, Project management tools.
- Java Style Guides: oracle, twitter, google

Agile Principles: (not specific in exams, but understand them)

- Satisfy the customer through early and continuous delivery of valuable software
- Code has to welcome changing requirements, even late in development
- Deliver working software frequently
- Business people and developers must work together daily throughout the project
- Motivation → work with motivated people create a safe and supported environment
- Face-to-face conversation (but not really long meetings every day)
- Working software
- Simplicity
- Self-organizing teams

Software Development Models

- Generally, we write software for a client
- Customers usually change their minds → they have very different expectations
 - Beforehand we need to define the “requirements” (define what the user wants) → then we can write software

Functional Requirements

- Business Rules, Administrative functions, Authentication, Authorization levels, External interfaces, Certification requirements, Reporting requirements, legal requirements

Non Functional Requirements

- Performance, scalability, capacity, availability, reliability, recoverability, Maintainability, Serviceability, security etc...

Issues of non-developers:

- Non-technical people, Users that not commit to the set of written requirements, slow communication, users that are technically unsophisticated, users that do not understand the development process.

Issues of developers:

- Starting implementation before requirements are clear
- Different vocabularies between technical personnel and end-users

Techniques

- Rapid prototyping, modelling and sketching, Write use cases, agile development (turning use cases into solutions)

Software Development

Waterfall development

Requirements → product requirements document

Design → software architecture

Implementation → software

Verification

Maintenance

1. Analyse problem
2. Draft functional, technical and test specifications
3. Implement modules and perform unit testing +integrate modules and perform system testing
4. Release and Maintain

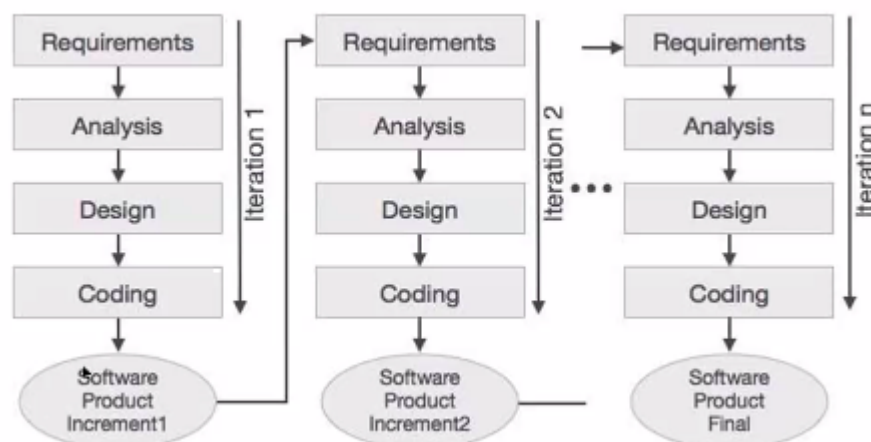
When to use waterfall method:

- You go from idea to execution
- Application is not big or complicated, project is short
- Requirements are specific
- Development environment is stable
- Resources are adequately trained and available
- Example: software for a wifi router, streetlights (there is no interaction with the user necessary)

Issues:

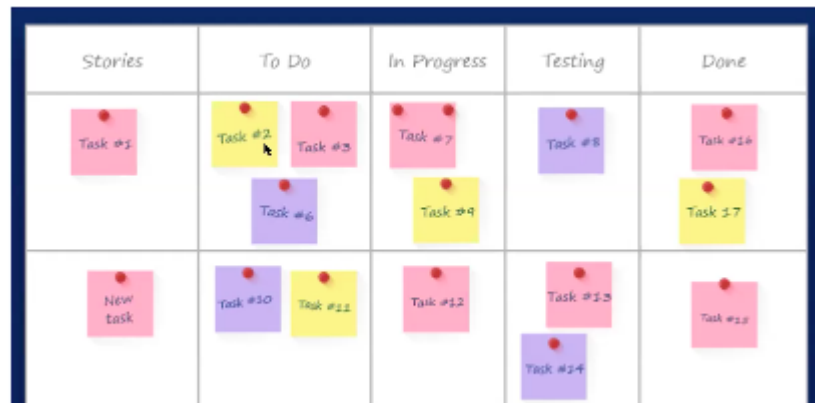
- Once you move to the next step you can't go back
- Sometimes these projects take years and requirements change, thus when the product is released they are not as useful as they could have been anymore.

Improvement: **Iterative Incremental Model (Plan Driven):** you never deliver, you write the whole software and then you reassess.



Agile development

- **Based on iterative incremental development** → Requirements and solutions evolve through team collaboration
- Doing the waterfall steps for every small part of the software (constantly delivering these small parts)
- Agile Manifesto
- Examples of agile development:
 - Kanban
 - Sprint to Kanban board



- - Scrum
 - Facilitate the agile **process**:
 - **Sprint**: takes either 2 weeks or one month
 - After the sprint is some we are going to **release** something (the part of the product we were working on) → show this **increment** to the client
 - **Sprint review**
 - **Sprint Retrospective** → “what went well?”, “What didn’t we finish?”
 - But the things that we still need to implement in the **Product Backlog**
 - Plan a **new meeting** from the BackLog → decide new Sprint requirements
 - **Repeat cycle**
 - **Sprint Planning**: the work to be performed is planned by the Scrum Team
 - **Daily Scrum Meeting** → For the Scrum team to synchronize the activities and create a plan for the day
 - **Sprint Review**: help at the end of the Sprint to inspect the Increment and make changes to the Product Backlog, if needed.
 - Sprint Backlog

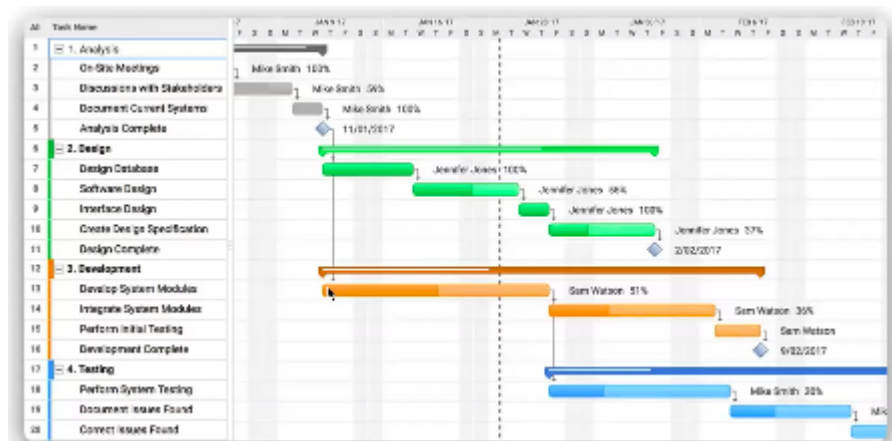


- **Scrum Estimation:** is in terms of the degree of difficulty → this is assessed using a particular scale. E.g:
 - Numeric sizing (1-10), T-shirt Sizes (XS, S, M, L, XL, XXL, XXXXL), Dog breeds (Chihuahua,...,Great Dane)
- **Common Mistakes:** underestimating the complexity, not respecting promised commitments made to the stakeholders
 - Lean (M.V.P)
 - DevOps

Roles:

Project Management

- Tasks can be bundled up into projects, these have a set of goals defined
- Deliverables: The set of goals for the project
- Program: the combination of multiple projects with overlapping goals
- **Strict Management:**
 - Budget
 - First, make an estimate on how much you are going to spend (talk to the team)
 - Start the project and check whether the estimates were correct or not
 - Time
 - Done with Gantt Charts → tasks (bigger tasks divided into subtasks), assign tasks, the estimate of time, dependencies, progress bar, milestone.



- A simplified version can be made in Excel

Dependencies (in a Gantt Chart):

- **Finish-to-Start**
 - A must finish before B starts
- **Finish to Finish**
 - B can't finish before A finishes
- **Start to Start**
 - B can't start before A starts
- **Start to Finish**
 - B can't finish until A begins

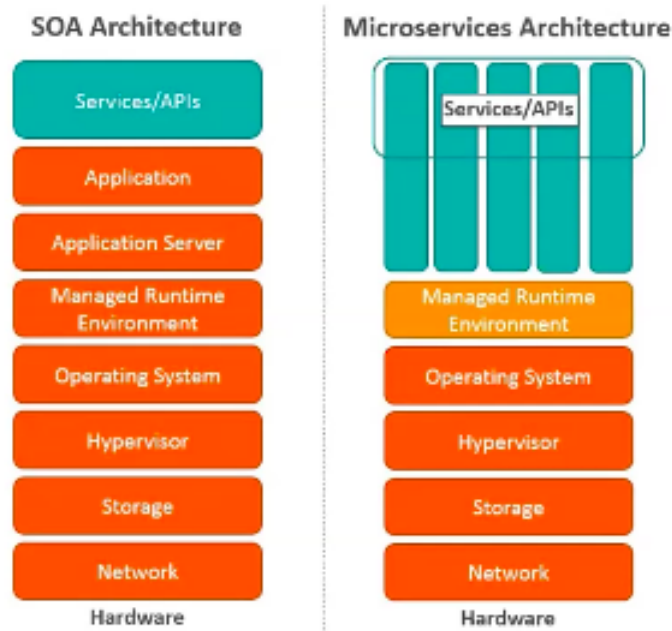
Project Manager

- Checking on the status of a deliverable
- Reviewing and identifying new work, monitoring existing tasks
- Creating a plan to reach a specific milestone
- Ensure that project work meets the quality and reliability requirements that had been identified at the beginning of the project

Software Architecture:

- Architecture is part of the design → done before the system has been built
- Gives a basis for analysis and for re-use of elements and decisions
- Facilitates communication with stakeholders
- Helps in risk management, enables cost reduction
- **SoC: Separation of Concerns**
 - Software should be separated based on the kinds of work it performs
- **Important:** “Changes in lower layers should never “break” the upper ones”
 - If the interface is done first, it should break if there are changes in the backend
- **Multi-Layered Architecture**
 - MVC - Model View Controller

- Model is the central component, defines how data is stored, class model
 - Database
 - File Storage
- View: what is presented to the user and what the user interacts with.
 - Website, Gui, Terminal
- Controller: Business logic of the program, connects Model and View, accepts inputs.
 - Algorithms
 - Business Logic
 - Machine learning model
 - Data transforms
- Interactions between MVC
 - These need to be clear
- **Model-View-Controller: Scaffolding**
 - Method of creating barebone controllers based on a defined model
 - Creates CRUD operations (create, read, update, delete)
- **Advantages of MVC:**
 - Simultaneous development
 - Ease of modification
 - Testability
 - The communication between the view and the controller is different compared to what we have seen before, for example, it can be made through the use of an API
- **Disadvantages of MVC:**
 - Inevitable clustering (hard to scale)
 - Not applicable to all types of software
 - New functionalities must be adjusted to MVC patterns
- **Service-Oriented Architecture:**
 - Self contained
 - Consumer doesn't have to be aware of the service's inner workings
 - May consist of other underlying services
 - Different services that depend on each other
 - Each server is some kind of business activity
- **Microservices:**
 - SoC is achieved by splitting the software into loosely-coupled independent services
 - Cloud platforms are built on Microservices architecture
 - Provide API's virtual machines and containers that are developed for a specific service
 - Containers: small Virtual machine (machine in a machine) that runs a piece of software.
 - E.g. Popular method for deploying services → Docker
 - Software packages
 - You can turn your own computer into a multi service provider
 - Each container does not depend on anything else
 - And combining these containers together is the future of software engineering



- **Overall:**
 - Design Inputs
 - Design activities
 - Design Outputs

Lecture 3:

- ☐ Code Review
- ☐ Version Control
- ☐ Specifications

Code Review:

- Study of source code by people who are NOT the original authors of the code
- Constructive feedback
- Purposes:
 - Improve the code
 - Improve the skills of the programmer who wrote the code
- Widely practiced in open source projects
 - Examples: Apache, Mozilla
- “Smelly” code - badly structured, not documented
 - Duplicated code - risk to safety → **DRY** (Don’t repeat yourself)
- Comments:
 - **Specification** (a crucial comment)
 - **Specifications document assumptions**
 - **Provenance** of code → link from where the code was copied/inspired by
 - By doing this you can avoid violations of copyright
 - Careful: code can fall out of data or websites can change
 - **Bad:** some comments are too obvious
 - In Java → Javadoc

- From 9 - 17 slide number:
- Don't use global variables
- Snapshot diagrams
 - Local
 - Instance
 - Static

Version Control:

- Example: Alice works on an assignment, before handing it in she makes a change and knows everything is broken → now she needs to figure out what is not collaborating with her change.
 - She could retrieve a past version to check when this change had been made and where.
- Standard software tools exist for comparing text - linux and mac diff
- **Overall** a programmer would like to:
 - **Revert** to a past version
 - **Comparing** two different versions
 - **Pushing** full version history to another location
 - **Pulling** history back from that location
 - **Merging** versions that are offshoots of the same earlier version
- Git does everything for you
- Distributed vs. Centralized:
 - **Centralized (CVS and subversion)**
 - One master server → there is a cloud between the group and everyone can access it
 - Users only communicate with the master
 - **Distributed (Git, for example)**
 - Multiple repositories are created equal, spread out all over the place
 - Every single user is also by itself a cloud → sharing and distributing code at the same time
 - Every person has the full history and information of their source code on their laptop already.
 - If everybody in your group other than one, loses all of their source code, they could just synchronize with the one person that still has all of the information, and start up already.
 - You don't rely on any specific location on the web to store your code, every laptop is an individual server by itself.

Git (invented by the same guy who invented linux)

- Three state: (all local operations → nothing on the cloud)
 - **State 1: Working directory → files on a working computer**
 - Modified: you have changed the file but have not committed it to your database yet
 - **State 2: Staging area**
 - Staged: you have marked a modified file in its current version to go into your next commit snapshot
 - All the things that were stage will go to a specific version. This is done and decided by us, it is not automatic.

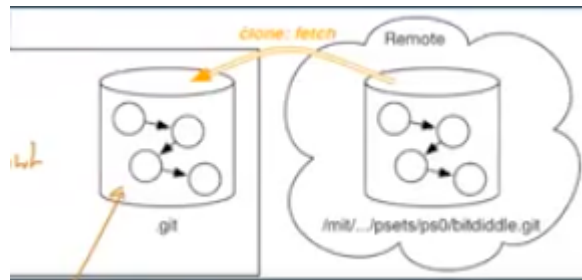
- (this is something new to us)
- **State 3: Git-Directory (repository)**
 - Committed: data is safely stored in your local database

Steps for Git Clone operations (LAB) (needs to be installed)

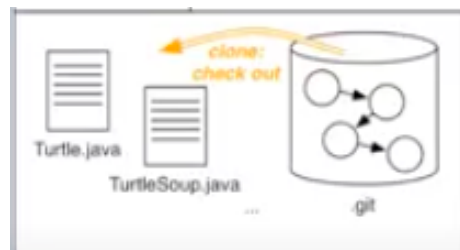
`git clone https://github.com/.../KEN1520-Assignment-3.git a3`

What happens after having run this operation on the terminal:

- **Step 1: Creates an empty local directory a3, and a3/.git**
 - On terminal: command **git clone <URL>**
 - a3 is the name of the directory where we want to store. Creates a folder called a3 (empty directory) and a hidden file .git
 - Git itself is the tool, while github is the host/cloud
- **Step 2: Retrieves the version control tree (graph)**
 - Stores it in the hidden file .git (that is in the background of the empty folder that we just created in step 1)
 - Calls git fetch → fetches the graph (this is invisible)



- Calls git Checkout → copy info from the graph into the folder → extracts the latest version into the directory (makes it visible) (The graph still contains all of the versions)



The graph (DAG - directed acyclic graph - NO cycles)

- Every arrow goes to one direction
- Each node is a commit with unique ID
 - Version number: version ID
- Root node - latest version
- As you go down the graph, you go back in time to previous versions
 - The last node is the first version
- Two or more nodes have the same parent: diverged from a common previous version
- A node has two parents: tie divergent histories back together



- Git stores file efficiently, it only changes the file when there is actual change in the file, if the file is the same from version to version these will not be copied → it will just link to the same physical file:
 - Git object graph stores each version of an individual file once
 - Multiple commits share that one copy

Git add and Git Commit

- Makes changes into the history

Git push and Git pull

- When we clone a repository, we obtain a copy of the history graph
 - We remember where we retrieved the repository from → github
 - We can use this information to upload our local changes to the cloud
- In our case our server is github
- Using git commit, we add new commits to the local history on the branch master
- Send those changes back to the origin remote (github)
 - Command: *git push origin master* (the master branch is now called "Main")
 - Copies the DAG to the remote cloud

Merging (Alyssa and Ben work on a project together): Example in lecture: 1:26:30

1. Both Alyssa and Ben clone
2. Alyssa creates hello.scm and commits
3. Ben creates hello.rb and commits
 - a. Since alyssa was the first to push her change up to the remote, Ben's push will be rejected (otherwise Alyssa's commit would disappear)
4. Ben pulls to merge his changes with Alyssa
5. Merges Ben's history with Alyssa's, creating a new commit that joins together the two histories
6. Ben can git push
7. Alyssa can git pull to obtain Ben's work

→ BEFORE YOU START WORKING, ALWAYS git pull → minimizes the likelihood of having to merge later.

→ Git has a lot of commands, these can be seen in a GIT CHEAT SHEET

Specifications:

- Acts as a contract between the implementer and the client
 - *Implementer*: responsible for meeting the contract
 - *Client*: can rely on the contract
- Contract will bind both parties to some agreement
- Both good for the client and the impmenter
 - *Client* doesn't need to read the code, it is giving the input and it expects an output
 - *Implementer* can do the computation from the given input and will return an output. They can change the implementation without having to tell the client
- Structure:
 - Definition/declaration of the method
 - Requires: define input (outside of Java)

- Effects: define output (outside of Java)
- Specification of a method consists of several clauses
 - Precondition → requires
 - Postcondition → effects
- If both adhere to the contract (input is correct), then you have to return the agreed postcondition
 - Appropriate values, Throwing specified exceptions, Modifying or not modifying objects
- If the precondition is violated (not correct), then the method is free to do anything
 - Not terminating, throwing only some exceptions, returns arbitrary results.
- In Java the preconditions and the postconditions are not checked by the compiler → they are stated in the comments
 - @param → precondition
 - @return → postcondition
- Test cases MUST obey the contract
- **Exceptions for special results**
 - Instead of handling special results (inputs) by returning special values, you should throw an exception.
- **Two Types of exceptions in Java**
 - **Unchecked** → the compiler doesn't check, but easy to check by the user
 - *Errors*
 - *RuntimeException*
 - **Checked** → compiler checks, but difficult to check by the user
 - *Exception*
 - *Throwable*
- Always include the exceptions in the specification using: @throws, but don't need to define them in the method declaration
 - DO NOT mention unchecked exceptions that signal bugs

Lab 4: Coding exercises on github, multiple choice quiz on canvas (visible after the lab), install git on computer before lab. PDF of the assignment should be available

Git Commands:

<https://www.notion.so/Introduction-to-Git-ac396a0697704709a12b6a0e545db049>

- *git clone*
- *git init* → initialize a repository
- *git status* → you run it after making changes on the file(s) → tells you which files have changed.
- *git add*
 - *git add .* = all files that have been changed or have been created new will be staged (sometimes you don't want that and you want to stage only specific files)
 - *git add <fileName>* (remember to run git status after that and see if the modified in red turned green)
- *git pull origin master*

- *git commit -m "<message>"* → In the message say what changes have been made
 - *git push origin master* → pushes changes into the original repository
 - *git log* → see all the commits that were made for the project
 - Will show details of each commit: author name, the generated hash, date and time of the commit, commit message
 - *git checkout <commit-hash>* → to go back to the previous state of your project code that you committed (the commit hash can be seen with the git log command)
 - *git checkout master*: go back to the latest commit (the newest version of the project code)
 -
-
- *Untracked files - not in the registry of git* → e.g if you add a file to the cloned repository - needs to be staged and committed.
 - *Staging: preparing a file to be committed*

Lecture 4:

- ❑ Object-Oriented Design
 - ❑ Class Design
- ❑ Modelling
 - ❑ UML
- ❑ Design Patterns

Object-Oriented Design

- **First step:** Modelling state first
- Strictness depends on what language you are using (Java, C++, Python etc.)
- OOP Key Concepts:
 - Encapsulation
 - Better control of attributes and methods, making sure that object change only how we want them to.
 - In setter methods we can do some validation and check inputs that will change instance variables
 - **Access Modifiers**
 - **Public:** accessible for all classes (also outside of the package)
 - **Private:** accessible only within the declared class
 - **Default:** accessible within the same package
 - **Protected:** accessible within the same package and subclasses
 - Inheritance - hierarchy of relationships
 - Polymorphism - the ability of objects to take many forms
 - Creating subclasses from superclasses - overriding classes and methods
 - Abstraction - Abstract Classes and Methods
- Interfaces vs. Abstract classes

- You can implement multiple interfaces in one class, while you can only extend to one class
- To call a non static method from a static method, you have to create an object from the class where the non static method is found, and call the method as an extension of the object

ABSTRACTION (missing)

STATIC (missing)

Modelling/UML

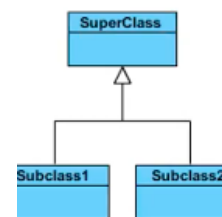
Unified modelling language (UML)→ watch for LAB as well

Classes

- Class name at the top
- Variables under class name
- Methods under Variables
- Class Variable and methods
 - + : public
 - - : private
 - # : protected
- Example: <modifiers><name><parameters>:<type>

Relationships

- Inheritance
 - “Generalisation”
 - “Is - a” relationship
 - **ABSTRACT CLASSES**: *Title in italics*
 - <<Interfaces names are shown in <<brackets>>>>
 - Open triangle



- Aggregation
 - “Part of “ relationship
 - Class 2 **is part** of Class 1
 - Or, Class 1 is **using** Class 2
 - Example:
 - Class Message Queue has a private ArrayList containing Messages (from Message Class)
 - Multiplicities: In this example there is one message Queue and many messages.



- Multiplicities (Cardinality):
 - Any number (0 or more) : *
 - One or more : 1..*
 - Zero or one: 0...1
 - Exactly one: 1
- Association
 - More general type

- Associations can have roles
- Defines how classes relate to each other
 - Example (Bidirectional): A Course has students can be represented as:
 - COURSE has as participant STUDENT
 - And, STUDENT registers for COURSE



- Example (directed)
 - Navigation is unidirectional
 - Message doesn't know about message queue containing it
 - One-to-Many relationship



- Composition: Special type of aggregation → the lifetime of an object is connected to the lifetime of another object
 - Parts are destroyed when the whole is destroyed
 - Can have a cardinality
 - Example: MESSAGE QUEUE are permanently contained in MAILBOX



- Example: A BATHROOM is a composition of HOUSE, the bathroom cannot exist without the house
- Interface (implementation) (dotted line with an open arrow)



●

Overall : UML Diagrams

- **UML Class Diagrams:** Good for describing the STRUCTURE of a system
- **UML Sequence Diagrams:** Good for describing the BEHAVIOUR of systems
- **UML State Diagrams:** more abstract diagram describing classes with different states (in some cases useful for AIs)

Design Pattern

What is it?

- Structural thing that you can implement in your code to make sure that what you are doing is good design.
- Standard solution to common programming problems
- Technique for making code more flexible
- A high-level programming idiom
- Program organization
- They can be language specific

Elements of a Design Patterns:

- Name
 - Introduces vocabulary, allows abstraction of ideas
- A problem that it applies to
 - Algorithm, class structure, conditions
- A solution to the problem
 - Relationships, responsibilities and collaborations between the elements
- Consequences
 - Side-effects and trade-offs of using the pattern

Covering three types of Design patterns:

1. Creational Patterns
 - The Factory Method:
 - Provides an interface for creating objects in a superclass
 - Subclass is allowed to alter the type of objects that will be created
2. Structural Patterns
3. Behavioral Patterns

Refactoring Guru is a good website with many examples regarding design patterns

Lecture 5:

- ❑ Designing Specifications
- ❑ Debugging

Designing Specifications:

Overall definition: Specification of a method or a class and defines what the method or the class is supposed to do. Sort of a contract between the programmer and the client.

Remember: Java doesn't check the specs at a compiler level. The specification is more at a comment level. Additional requirements in the code are sometimes needed in order to follow and satisfy the specs.

1st Dimension Deterministic vs Underdetermined Specification

Deterministic Specification

- Example: **static in findExactlyOne(int [] arr, int val)**
- **Inputs (requires):** val occurs exactly once in arr
- **Output:** returns index i such that arr[i] = val
- **Deterministic** because:
 - Only **one return value** and **one final state** is possible
 - There are no valid inputs for which there is more than one valid output (ONLY ONE VALID OUTPUT)
- **Careful:** even if we require something (in the specs) and implement our code based on that, we can assume that sometimes the user (using the code) will still violate the requirement or misinterpret the specs.
 - It is the responsibility of the user, using the code, to follow the requirements of the specification.

Not Deterministic/ Undetermined

- Example: **static int findOneOrMoreAnyIndex(int [] arr, int val)**
- **Inputs (requires):** requires that val occurs in arr (we don't require exactly one)
- **Output (effects):** returns index i such that arr[i] = val
- **Undetermined** because:
 - **Multiple valid outputs** for the **same input** are allowed
- **ATTENTION (language):** not deterministic ≠ nondeterministic. To avoid confusion we use undetermined.

2nd Dimension Declarative vs. Operational Specifications

Operational: Series of steps that the method performs

Declarative (more freedom to the programmer): just properties of the final outcome, and how it is related to initial state (Overall Declarative is more beneficial). How you link input and output doesn't matter, as long as you give the required output

- **Preferable:** easier to change implementation, and no technical language required to write down the spec.
- Example: **static boolean startsWith(String str, String prefix)**
 - There are many ways to express this in a declarative way. These are *three examples* (but there are many others).
 - **Effects:** returns true if and only if there exist String suffix such that prefix + suffix = str
 - **Effects:** returns true if and only if there exists integer i such that str.substring(0, i) = prefix
 - **Effects:** returns true if the first prefix.length() characters of str are the characters of prefix, else false.

There are ways to write a spec, such that they are a mix of declarative and operational.

3rd Dimension Stronger vs. Weaker Specifications

Question: What is a strong and what is a weak spec? We look at this from a client perspective (this can also be another programmer using our code)

Stronger: **safer** for the client

Example 1: **S1 is the first Spec, S2 is the second Spec:**

- **If Spec S2 is stronger or equal that spec S1 (Both following points have to hold):**
 - S2's precondition (requirement/input) is weaker than or equal to S1's
 - S2's postcondition (effects/outputs) is stronger than or equal to S1's (for the states that satisfy S1's precondition)
- **For the implementer's perspective:**
 - You can always weaken the precondition
 - You can always strengthen the postcondition

	<u>pre</u>			<u>post</u>		
	weaker	equal	stronger	weaker	equal	stronger
stronger spec	X	X			X	X

Example 2: **Static int find(int[] a, int val)**

- **S1:**
 - **Requires:** val occurs **exactly once** in a
 - **Effects:** returns index i such that a[i] = val
- **Replace S1 With S2: (S2 is stronger than S1 because the precondition (requires) is weaker and the postcondition (effects) stays equal.** As a client, if you don't read the specs and you give an input that has more than one occurrence, the code would still guarantee that the right answer will be outputted)
 - **Requires:** val occurs **at least once** in a
 - **Effects** (stays the same as before)
- **Replace S2 with S3: (S3 is stronger than S2, because the precondition (requires) stays equal and the postcondition (effects) is stronger.** Give a more specific return value (better from the client's perspective))
 - **Requires** (same as S2)
 - **Effects:** returns lowest index i such that a[i] = val

Example 3: **static int find(int[] a, int val)**

- **S1:**
 - Requires nothing
 - Effects: returns index i such that a[i] = val, or -1 if no such i
- **If we replace S1 with S2: (In this case we are weakening both the pre- and postconditions. → INCOMPARABLE CASES)**

- Requires: val occurs at least once in a
- Effects: returns lowest index i such that a[i] = val
- **Incomparable: ≠ weaker (we cannot say whether it is weaker or stronger)**
 - Precondition and Postcondition are both weaker.

Diagramming Specifications: (SCREENSHOT SLIDE 13, 14 ANNOTATED)

We can see that by strengthening the postcondition, the number of possible implementations will also narrow down.

Overall: Specifications should be

- Coherent
 - Single and complete unit
 - Avoid lots of different cases
- Do NOT: use long argument lists...
 - Bad Example 1:
 - Bad Example 2:
 - Problem: returns an ambiguous return value that doesn't tell you what happened inside the function
 - Bad Example 3:

Debugging - Avoiding Debugging:

Term invented by Grace Hopper (in the 60s), student at Harvard, when she found a Moth (bug) stuck in a circuit which was the reason why they were having problems.

Remember: KISS (keep it simple stupid), initially you should care about elegance or speed, always choose the easier way.

First Defense:

- Static checking
- Dynamic checking: Java compiler automatically produces an error for:
 - Overflows
 - OutofBounds exceptions
- Use **final** keyword when possible for immutable references
 - If a line of code will try to change an immutable (final) variable, the compiler will detect this bug for you. (So less work for you)

```

final char[] vowels =
    new char[] { 'a', 'e', 'i', 'o', 'u' };

▪ Which of the following statements are illegal?

^ vowels = new char[] { 'x', 'y', 'z' }; illegal
L vowels[0] = 'z'; legal

```

- **Careful in this example:** you cannot change the array vowels as it is constant.
 - However you can change the single elements in the array because they are not constant.

Second Defense: Localise bugs

- Use Preconditions/requires and Postconditions/effects
 - In Java: @param @return
- **Example:** (we know that we have to do something if the user inputs a negative double). A good practice is to throw an **unchecked exception** to prevent bug from propagating

```

/**
 * @param x requires x >= 0 ✓ pre, requires
 * @return approximation to square root of x ✓ post, effects
 */
public double sqrt(double x) { ... }

```

- You can also use and **assertion**:
 - **Assert x >= 0 : "x is " + x**
 - At this point x should be non-negative
- Do not use this assertion: check if the JVM is doing the correct procedure
 - Example: x = y+1;

Reproduce the Bug:

- Step 1: reproduce the bug in the simplest possible way you can find (where the program fails)
 - Reduce the size
 - Create small test case(s)
 - Start fixing the bug
 - Check if it fixed the original problem (if not, repeat)

Scientific method to debugging:

1. **Study the data**

- Stack trace from an exception
- Test input
-

2. **Hypothesize**

- **Slicing**: finding the parts of the program that contributed to computing a particular value. Each of these parts could be the reason for the bug. Usually you do this manually yourself.
- **Delta Debugging**: Testing base debugging procedure
- **Prioritize Hypotheses**

3. **Experiment**

- Experiment should be chosen to **test the prediction**
- Best experiment is a **probe**:
 - Print statements
 - Logging (logging framework in Java is Log4)
 - Assertions
 - Breakpoint with debugger
- The experiment should not change the code
- **Swapping components**: another useful strategy
- Don't fix yet, try to understand where it comes from and why it is a bug, try to create a test case for it

4. **Repeat**

- You may have to repeat this many times

5. **Fix the bug**

- Fix bug
- Add a test case

Preparation for monday Lab:

Try to figure out in my ide (integrated development environment), how to use the debuggers

Extra notes from MIT Lecture 5:

http://web.mit.edu/6.031/www/fa18/classes/07-designing-specs/#declarative_vs_operational_specs

<https://web.mit.edu/6.005/www/fa16/classes/08-avoiding-debugging/>

<https://web.mit.edu/6.031/www/sp20/classes/09-avoiding-debugging/>

<http://web.mit.edu/6.005/www/sp16/classes/11-debugging/>

<http://web.mit.edu/6.031/www/fa17/classes/13-debugging/>

Lecture 6: Application Programming Interfaces - APIs

YT video: "What is a REST API"

- **Example**: You work in an ice cream shop and want to create a web application to show the ice cream flavours that are in stock each day. However you also want the

workers to be able to make updates on that flavour → you can do this by using a REST API:

- The Web page communicates with a cloud-based server via a REST API
- Have and **endpoint**: <http://icecream.com/API/Flavours>
 - API: API portion of the endpoint
 - Flavours: signifies a Resource
- “Request”: sent from the client to the server
 - (CRUD) - In API the equivalent is HTTP methods or Operation
 - Create → (in API) **Post**
 - Read → (in API) **Get**
 - Update → (in API) **Put**
 - Delete → (in API) **Delete**
 - In the request you might want to send: Operations, Endpoint, Parameters/Body, Headers(may have API key or authentication data)
- “Response”: sent from the server to the client
 - EXAMPLES:
 - **Request**: “Get Flavours”
 - Operation: GET
 - Endpoint: /api/flavours
 - **Response**: An array of those flavours

Update the flavours

- **Request**: “Update”
 - Operation: PUT
 - Endpoint: /api/flavours/1
 - ID: “1” if you want to replace
 - Parameter/Body: Chocolate (replace mint with chocolate, for example)
 - **Response**: acknowledges that ID of 1 is replaced with the flavour of chocolate
- **Rest API:**
 - *REST: “Representational State Transfer”*
 - Standardised software architecture style
 - **“Communication”** between client and server
 - **“Restful web service”**: a service that uses REST APIs to communicate
 - **Benefits of REST APIs:**
 - Simple and standardised approach to communication
 - Don’t have to worry about how to format data → everything is standardised and industry used.
 - Scalable and Stateless
 - Easily make modifications
 - High Performance
 - Supports caching

Lecture 6 Part 1 - Slides for web development using APIs (MIT course)

API: Application Programming Interface

- Connects two pieces of software together
- Services that have already been provided for you so that you don't have to re-invent them
- REST API: type of API that you can access over a network

APIs for web development - VIEW

- A web API has no dependencies because it is a server in itself
- Difference between API and Application
 - Public API
 - A **way** your code can **ask** the application/library/service to **do things**
 - **Boundary** of the application/library through which **requests** go in and **responses** come out.
 - Application
 - Example: A java library
 - "A blob of code that does things"
- Difference between API and UIs (User Interface)
 - Your code (communicates) \longleftrightarrow API (in the same way as the user communicates) \longleftrightarrow with the UI
 - **Software libraries** have APIs (they are intended to be used by other code)
 - **Applications** often have UIs (they are intended to be used by humans)
 - **Modern web applications** usually have both APIs and UIs
- HTTP (90s web): communicate through HTTP protocol with APIs
 - HTTP is an API for talking to Web Servers
 - 6 main methods (that we can ask the web server):
 - OPTIONS \rightarrow "Tell me about yourself"
 - GET \rightarrow **retrieve** a resource
 - POST \rightarrow **upload** a new resource
 - PUT \rightarrow **replace** an existing resource
 - PATCH \rightarrow **modify** a resource
 - DELETE \rightarrow **remove** a resource
 - Method Parameters
 - Whenever we **send** something to a web server we have to post a **request**:
 - **URL**: What we want to access in the web server
 - **Body**: what are we sending to the web server (contents to PUT/POST/PATCH)
 - **Headers**: Specific things we want to include in the request
 - Referrer: the page making the request
 - Accept: acceptable response data types (i.e: text, pdf etc.)
 - Cookie: contains specific user data
 - **Response** - what the web server sends back
 - **Status code**: i.e 200 OK, 301 redirect, 401 unauthorised, 500 server error
 - **Body**: the Content requested (i.e web page)

- **Headers:**
 - Content - type: of the returned object (text, pdf, etc..)
 - Set-Cookie: the user data that has to be stored in browser
 - Location: Instruction to look somewhere else
 - Serialization - to use the content in a structured manner
 - Network carries a stream of bytes/text
 - Each HTTP **request is serialized** into a big string with specific syntax
 - **Web server receives and deserializes** the string back into a structured request
 - Server then **serializes the response** that returns to client
 - This is then **deserialized** into response data
 - Methods that are run in an HTTP server
 - Safe methods: no side effects
 - **GET**: fetches a resource
 - **OPTIONS**
 - Idempotent methods: same side effects every time (repetition doesn't matter)
 - **PUT**: if method is repeated it just replaces the same (new) content again
 - **DELETE**: repetition doesn't matter cause you cannot delete twice
 - Neither safe nor idempotent methods: Different side effects each time
 - **POST**: creates a **different** new resource each time
 - **PATCH**: applies changes that can accumulate
 - Overall, the two methods that matter are GET and POST
 - **GET**: arguments are in a query string of url → easy to serialise whole request into url string, great for a few short method parameters
 - **POST**:
- **AJAX & REST (00s Web)**
 - **Asynchronous Javascript and XML**
 - AJAX is a part of javascript
 - XML is not replaced by JSON
 - Communicates with an HTTP server
 - The classic web application model:
 - **User actions** in interface trigger an HTTP **request** to a web server
 - Process of the **server**: Retrieves data, crunches numbers, talks to various legal systems → overall: **returns** an HTML page to the client
- **JSON**
 - Javascript objects - a way of defining objects in a text based format
 - A serialization of an object
 - **We will have to work with JSON a lot in the future**
- **Web APIs**
 - Idea: can code interact with web pages (and manipulate them?)

- The image illustrates the difference between a web page and its API representation. On the left is a screenshot of the GitHub web interface for the 'karger' repository. On the right is the JSON API response for the same repository, showing structured data like 'login', 'id', 'node_id', 'avatar_url', 'followers_url', etc.

- Less generic than HTTP API
- Exposes **app-specific methods**: login, updateMap, addToCart
 - For each method there is usually one url route
 - E.g <http://api.github.com/user> → gets user info
 - Way of connecting the url to the code that you are writing as a web developer
- Arguments in GET path (search parameters)
 - /user/karger, or
 - /user?kager
- The **result** is contained in the **body** of the **response**
 - Response format: JSON (mainly, sometimes for input arguments as well)
- **Encapsulation:**
 - Application API call is encapsulated inside HTTP API call
 - You need to think about what info is important and encapsulate all of it into a SINGLE call.
- **RESTful APIs**
 - Representational State Transfer
 - Keeps all ephemeral state (irrelevant information) in the client (rather than the server)
 - Every Method call, must include all the state the server needs to provide the method
 - Server holds the persistent state (everything you store in a database / relevant info):
 - Your file uploads
 - Your purchases

- **Security and Authorization:**
 - Companies sell APIs - they know who is using their service, because this service has to be purchased.
 - The problem with REST APIs is that it is stateless. This makes it less secure because anyone could use this service. For this reason, it has to be maintained secure somehow.
 - The client needs to be responsible for authorization
 - **Authorization with Tokens:**
 - Give the client a token that signifies that they are authorized to do something (for example: request username and password)
 - This token is used to identify the client → lets the server decide whether the client is authorised.
 - The token cannot be stolen
 - **Web token types:**
 - URL parameters
 - <http://site/?user=david&pwd=foo> (bad idea as this info can be intercepted and it can be dangerous for the client)
 - Cookies (consistent with REST, but also not the best idea as the cookies/text file is stored somewhere and it can be stolen)
 - Associated with a particular domain
 - It is a small text file that you store in your computer
 - Server: sends **Set-Cookie** (a header in the response from that domain)
 - Client: sends **Cookie** (a header with EVERY request to that domain)
 - HTTP Basic Auth header
 - HTTP Digest Auth header (hash username and password (cannot be reverted back to its original state → more secure))
 - HTTP Bearer Auth header
 - API keys
 - Note: users can still steal shared API keys
 - **Acquire Tokens**
 - They need to prove who they are
 - The initial authentication must be something that only you can access (e.g username/password (mind), authentication with text message (phone), the site emails you a link with a token (email))
 - **OAuth providers (servers)** → (used by google or apple, for example, to authenticate users during login) you can use them for APIs

Lecture Part 2 (The **EXAMINABLE** material) (how to create RESTful APIs + clients) (in java or Python or Node.js)

Java - REST Spring Boot APIs:

Resources.

- <https://www.youtube.com/watch?v=9SGDpanrc8U>
- <https://spring.io/guides/gs/consuming-rest/>

- <https://spring.io/guides/gs/rest-service/>
- **Spring Boot** is a framework to build applications (very well-known)
 - Gives you everything you need → connect to any database
 - Easy to learn
 - It is production ready
- Project overview: 3 layers
 - API Layer - GET, POST, PUT, DELETE
 - Service Layer - business logic
 - Data access layer - connects to any database
- **Spring Initializr**: where we can **bootstrap** any given spring boot application
 - Select: Project (Gradle), Language (Java), Spring Boot (Spring Boot version)
 - Dependencies: Spring Web, Spring Data JPA, PostgreSQL Driver
 - Download and open with IntelliJ
 - Test and Java folders
 - Resources:
 - application.properties
 - Where we configure all the properties for our application + environment specific properties
 - Static and Templates folders are for web development, for example, with HTML and CSS
- Create a Simple RESTful API
 -

APIs for web development - CONTROLLER

Lecture 7: Abstract Data Types

Questions: How can they help with specifications? How do they facilitate good code?

Mutable vs. Immutable Types

Mutable: Types that you can internally change from the outside

- **StringBuilder Class**

- `StringBuilder sb = new StringBuilder("a");`
 - To change the content of the variable: `sb.append("b")`
 - The content is changing inside the original reference.

Immutable

- **String Class**

- **String s = "a"** if you want to mutate this string, there are different ways to do this:
 - `s = s.concat("b");`
 - Adds character b to a, so now s = "ab"
 - Even though the variable name is still the same, the old pointer to the string "a" is deleted and it points to a newly created location that contains string "ab".
 - So we cannot go back to an object, but we have to delete it and create a new object. Once you delete the reference to the old string, (In Java) it goes into the *garbage collector*.

Why does this matter?

- Example:
 - `String s = "a";`
 - `String t = s;`
 - `t = t + "c";` (s and t now are NOT the same)
 - Now, s points to "a" while t points to "ac"
 - In total we created 2 objects, 2 references (2 of which were deleted)
 - `StringBuilder sb = new StringBuilder("a");`
 - `StringBuilder tb = sb;`
 - `tb.append("c");`
 - Now, both sb and tb point at "ac"
 - In total we created 1 object and 2 references
 - If you change tb, you are also changing sb (careful, because it can cause problems)

Then, why do we use mutable types?

- Many languages completely avoid them
- But **advantages:**
 - Optimizing Performance:
 - Speed is always important. Mutable types can be beneficial you want to concatenate a large number of strings together
 - Sharing (Communication):
 - Two parts of the program can communicate by deliberately sharing a common mutable data structure.

Risks of Mutation (Examples in lecture)

Specifications for Mutating Methods

- It is crucial to include mutations in the method's spec (in the effects). If the effects does not explicitly say that the inputs can be mutated, then the implementor should assume that the mutation of the input is disallowed.

Useful Immutable Types

- For date: use immutable type for java.time
- Immutable collections for some known values:
 - List.of
 - Set.of
 - Map.of
- Methods for obtaining unmodifiable views of mutable collections:
 - Collections.unmodifiableMap
 - Collections.emptyList
- Unmodifiable copies of mutable collections:
 - List.copyOf

Abstract Data Types

Lecture 8:

- ☐ SOLID design principles
- ☐ GRASP design principles
- ☐ Code Refactoring

GRASP and SOLID principles

- Both explain (in general terms) how object oriented software should be designed
- GRASP (Not part of the course): General Responsibility Assignment Software Patterns
 - Provides more hands-on principles
- SOLID
 - Provides more general guidelines to agile development
- There are other principles

SOLID

- **Single responsibility principle**
 - "A class should have one, and only one, reason to change"
 - The class should only have one responsibility - one purpose
 - Example from lecture:
 - Instead of implementing methods calculate() and isValid() in AgeCalculator Class → implement isValid() in a class called AgeValidator().
 - Reduces number of bugs, improves development speed. We spend more time reading code (trying to understand it) rather than writing code. So Single Responsibility principle attempts to make your code more efficient.

- *LOOK AT SLIDES
- **Open/Closed Principles**
 - “Software entities (Classes, modules, functions, etc.) should be open for extension, but closed for modification”
 - “We should write our modules so that they can be extended, without requiring them to be modified. In other words, we want to be able to change what the modules do, without changing the source code of the modules”.
 - What we can do in this case is use Polymorphism:
 - Define super objects and subobjects and we can create new instances and new modules without changing the original code.
 - Write code in a way such that you can add new functionality without changing the existing code.
 - A class is *closed* → should not be open for new extensions → not be able to change them
 - But it is also *open* so that subclasses can use it as a parent and add new features.
 - To make it open you can add a layer of abstraction
- Liskov Substitution
- Interface Segregation
- Dependency Inversion Principle

Link with Design patterns: Design patterns are written in such a way that these principles will hold. They make sure that you adhere to these principles.