

# CS-665

Git/GitHub and UML – Intro Session

# Agenda

---

- Git and GitHub fundamentals
- UML basics
- Will address any assignment related questions

# History of Git

---

- What is Git?
  - It is a distributed source control system
  - Widely used modern version control system
  - OpenSource project – no license necessary to use it
- Other Popular SCM Tools:
  - VSS – Visual source safe
  - CVS- Concurrent version system
  - Rational Clear Case
  - SVN- Subversion
  - Perforce
  - TortoiseSVN
  - IBM Rational team concert
  - IBM Configuration management version management
  - Razor
  - Quma version control system
  - SourceAnywhere

**\* *SCM = Source Control Management Tool***

# Traditional SCM tools Limitations

---

1. Hard to Branch and Merge.
2. Hosting multiple developers on SAME task/work was tedious.
3. Developers were forced to use way too many plug-ins and face upgrade challenges - AnkhSVN, SlikSVN, TortoiseSVN etc.
4. Different code review processes, disconnected tools & many ways of enforcing it - it was painful.
5. Most tools support POST-Commit reviews and pollute trunk/master with code Reviewer changes/comments etc.
6. Often developers felt that it is not worth the effort to branch and merge because of SCM limitations.

# Git to Rescue the Developers!

## His Vision (Year 2005) :

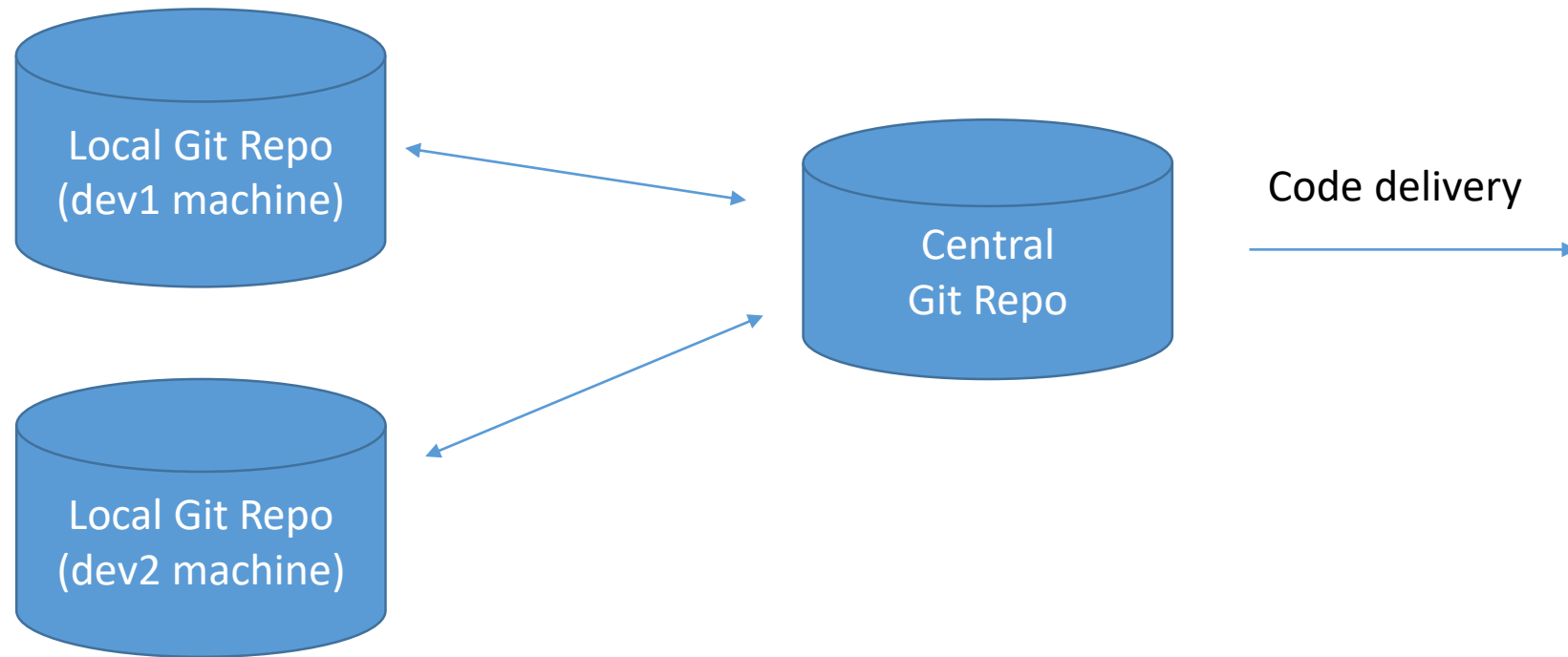
- Developers should have the freedom to have their OWN copies of SCM database on their machines
- When developers are ready to publish their code, reviewers should be able to REVIEW them prior merging it with the Master copy
- Reviewers and Developer(s) should be able to collaborate on review comments
- When the code is clean and ready , it should be merged with Master for delivery.

Author of Git &  
Principal developer of Linux kernel



# Distributed SCM Concept

---



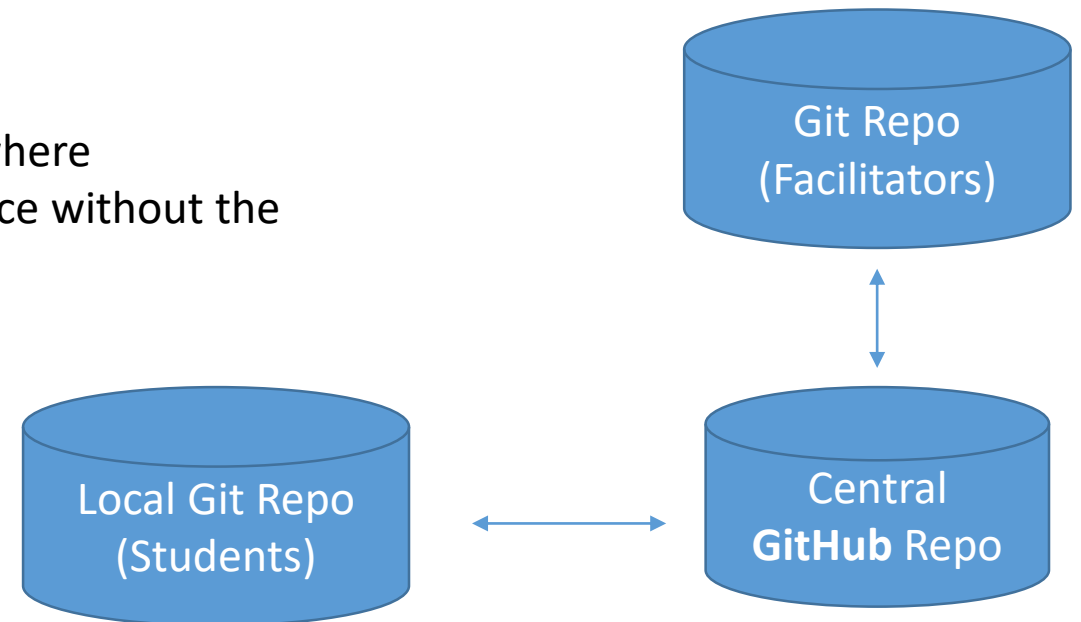
# Difference between Git and GitHub?

---

**Git** – Is the version Control system

**GitHub** – is a hosting service for Git repositories.

In simple terms , think of it as a SaaS solution where developers can HOST their code in a central place without the need to have their OWN git central server.



# Basic code Edit

---

- Take a copy of Central Repository
- Make Changes to the code
- Save Changes to Git (local)
- Publish your Changes to a central repository (for other developers to grab)



# Code Edit – for the first-time

---

1. Take a copy of Central/Remote/Origin Repository to your local machine(**clone**)
  2. *Make Your Code Changes << and follow Maven instructions to compile your Java source>>*
  3. Stage your changes (**add**)
  4. Save Changes to Git local (**commit**)
  5. Public Changes to a central/remote/origin repository (for other developers to grab your changes) (**push**)
- To pull others changes (needed for collaborative environment) – (**fetch/pull**)

# Incremental Edits

---

- ~~1. Take a copy of Central/Remote/Origin Repository to your local~~  
~~(clone)~~
  2. *Make Your Code Changes*
  3. *Stage your Changes* (add)
  4. Save Changes to Git local (commit)
  5. Public Changes to a central/remote/origin repository (for other developers to grab your changes) (push)
- 
- To Pull others changes (needed for collaborative environment) – (fetch/pull)

# Command Line Example:

**Note** : Make sure to download git client from here : <https://git-scm.com/downloads>

## Step 1: Clone

CA Command Prompt

```
C:\CS665\Assignment1\commandline_example>git clone https://github.com/metcs/met-cs665-summer-2018-assignment-1-sudakar21.git
Cloning into 'met-cs665-summer-2018-assignment-1-sudakar21'...
remote: Counting objects: 146, done.
remote: Compressing objects: 100% (62/62), done.
remote: Total 146 (delta 38), reused 140 (delta 32), pack-reused 0
Receiving objects: 100% (146/146), 22.30 KiB | 1.17 MiB/s, done.
Resolving deltas: 100% (38/38), done.

C:\CS665\Assignment1\commandline_example>
```

## Step 2: Make Code Edits and check the status

```
C:\CS665\Assignment1\commandline_example\met-cs665-summer-2018-assignment-1-sudakar21>git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   src/main/java/edu/bu/met/cs665/Main.java


no changes added to commit (use "git add" and/or "git commit -a")

C:\CS665\Assignment1\commandline_example\met-cs665-summer-2018-assignment-1-sudakar21>
```

# Command Line Example:

Step 3: Stage your changes and commit in single line (**add & commit**)

```
C:\CS665\Assignment1\commandline_example\met-cs665-summer-2018-assignment-1-sudakar21>git commit -a -m "adding new changes"
[master fa76106] adding new changes
1 file changed, 1 insertion(+), 1 deletion(-)
```

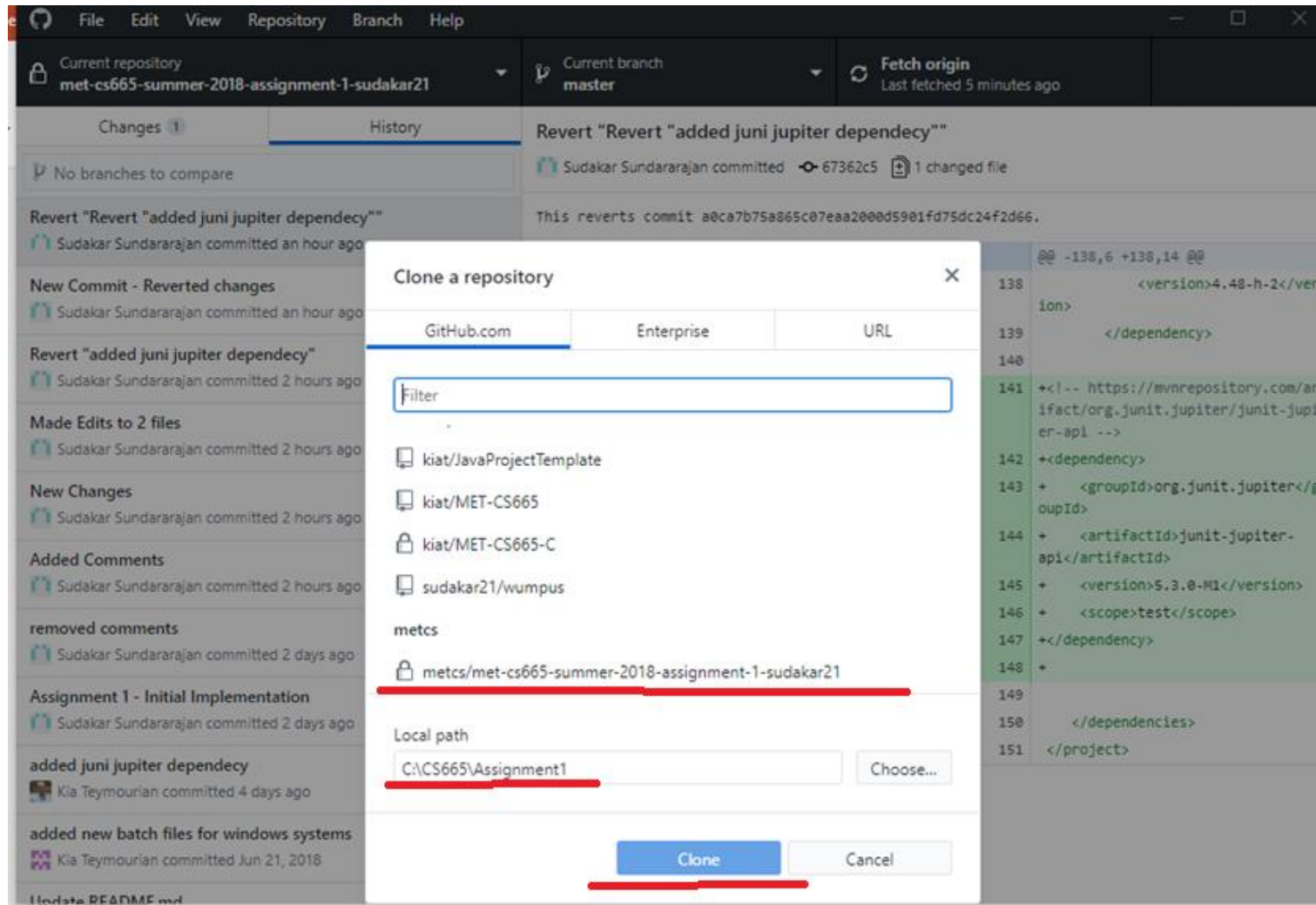


Code comments

Step 4: Make sure all changes went to central repo

```
C:\CS665\Assignment1\commandline_example\met-cs665-summer-2018-assignment-1-sudakar21>git pull
Already up to date.
```

# Review of GitDesktop (alternative option)



# Review of GitDesktop

(alternative option)

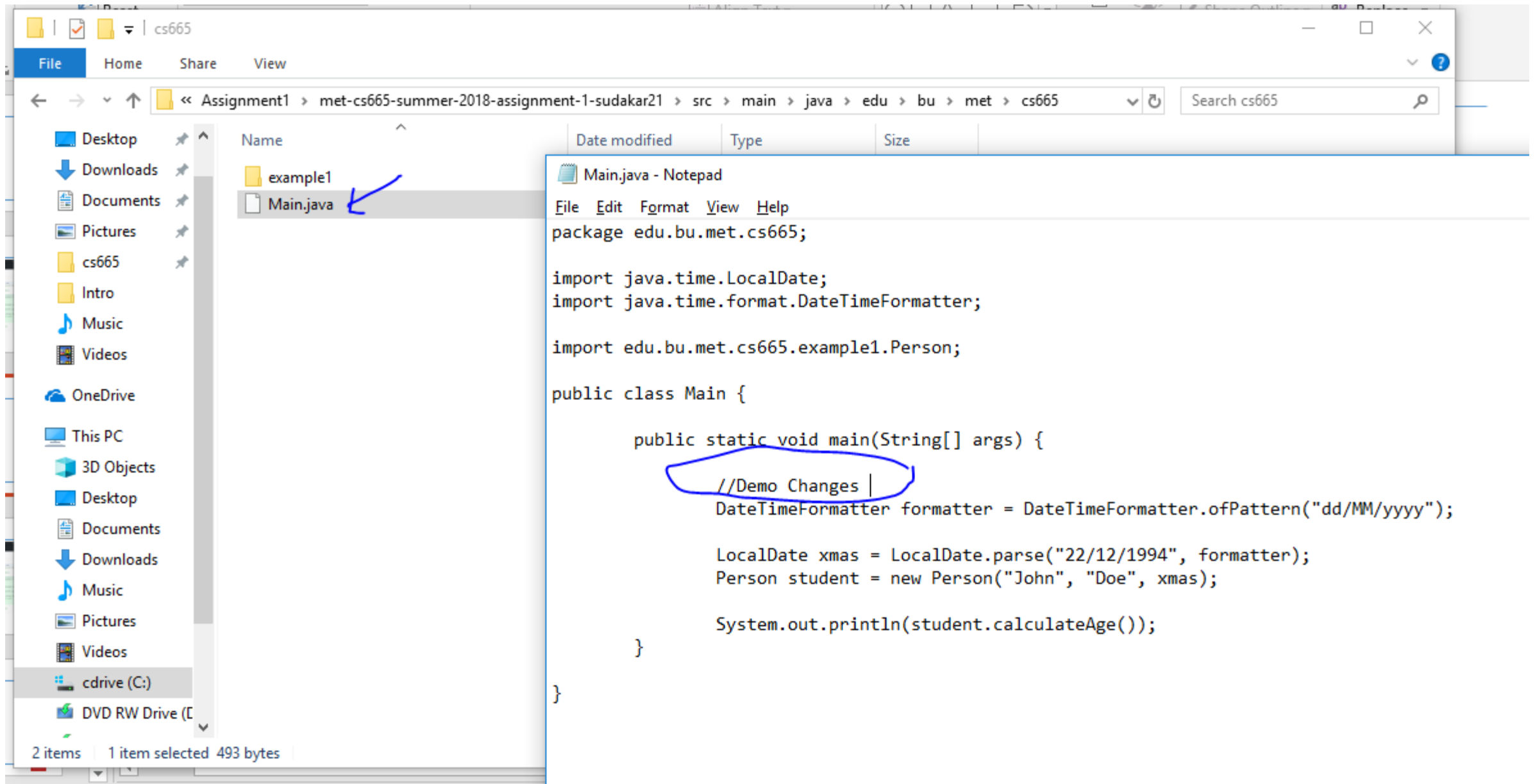
The screenshot shows the Git Desktop application interface. The top bar includes a menu (File, Edit, View, Repository, Branch, Help) and status information: Current repository (met-cs665-summer-2018-assignment-1-sudakar21), Current branch (master), and Fetch origin (Last fetched 3 minutes ago).

The main area is divided into two panes. The left pane shows the commit history, with the following entries:

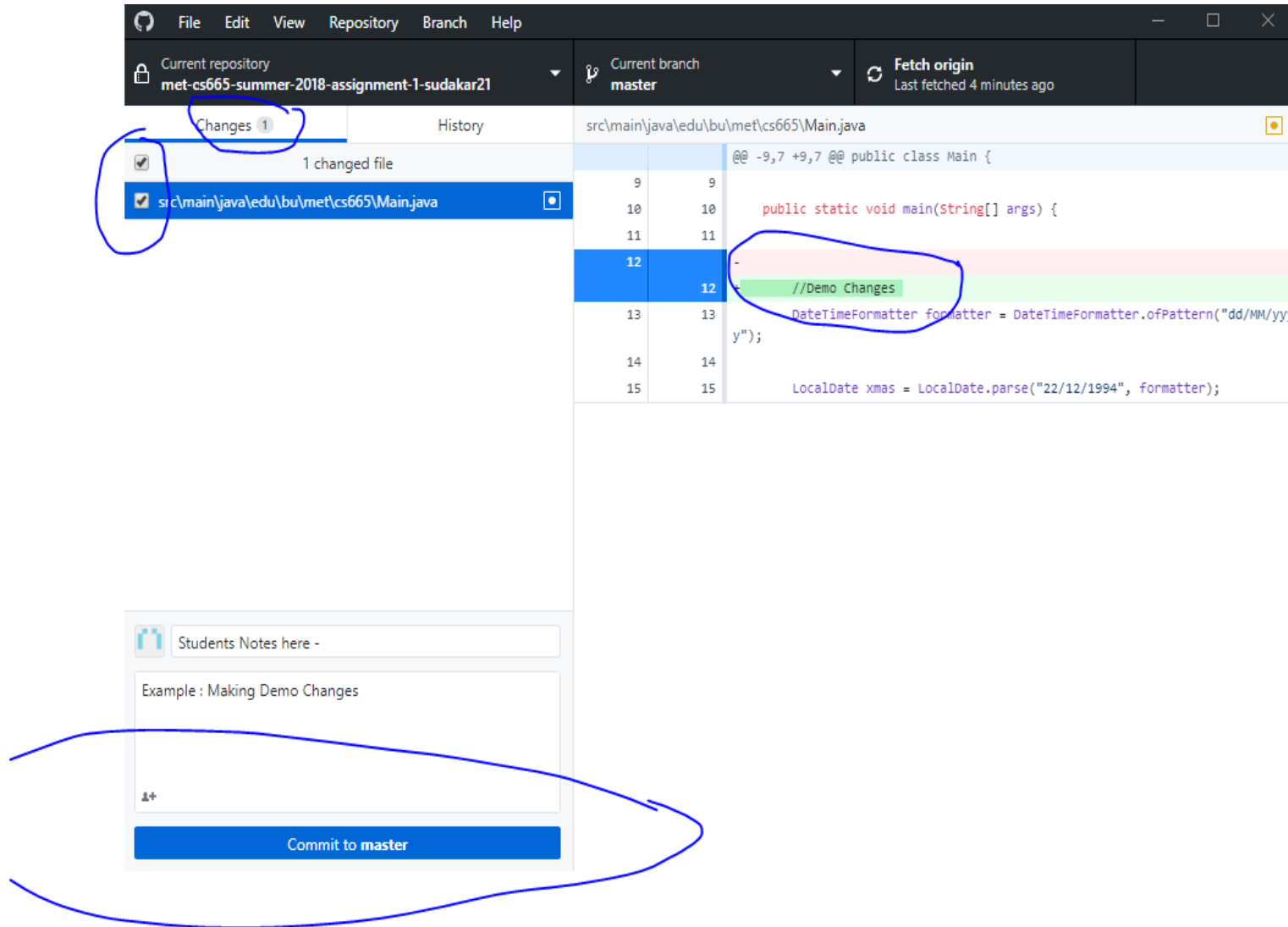
- Revert "Revert "added juni jupiter dependency"" (Sudakar Sundararajan committed an hour ago)
- New Commit - Reverted changes (Sudakar Sundararajan committed an hour ago)
- Revert "added juni jupiter dependency" (Sudakar Sundararajan committed 2 hours ago)
- Made Edits to 2 files (Sudakar Sundararajan committed 2 hours ago)
- New Changes (Sudakar Sundararajan committed 2 hours ago)
- Added Comments (Sudakar Sundararajan committed 2 hours ago)
- removed comments (Sudakar Sundararajan committed 2 days ago)
- Assignment 1 - Initial Implementation (Sudakar Sundararajan committed 2 days ago)
- added juni jupiter dependency (Kia Teymourian committed 4 days ago)
- added new batch files for windows systems (Kia Teymourian committed Jun 21, 2018)

The right pane shows the diff for the selected commit, titled "Revert 'Revert 'added juni jupiter dependency''". It indicates that this commit reverts commit a0ca7b75a865c07eaa2000d5901fd75dc24f2d66. The diff shows changes to the file pom.xml, with a summary of @@ -138,6 +138,14 @@.

```
@@ -138,6 +138,14 @@
138      <version>4.48-h-2</version>
139      </dependency>
140
141      +<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
142      +<dependency>
143      +  <groupId>org.junit.jupiter</groupId>
144      +  <artifactId>junit-jupiter-api</artifactId>
145      +  <version>5.3.0-M1</version>
146      +  <scope>test</scope>
147      +</dependency>
148      +
149
150      </dependencies>
151    </project>
```



# Review of GitDesktop (alternative option)





# Branching and Release Strategy

- (not needed for this class)

---

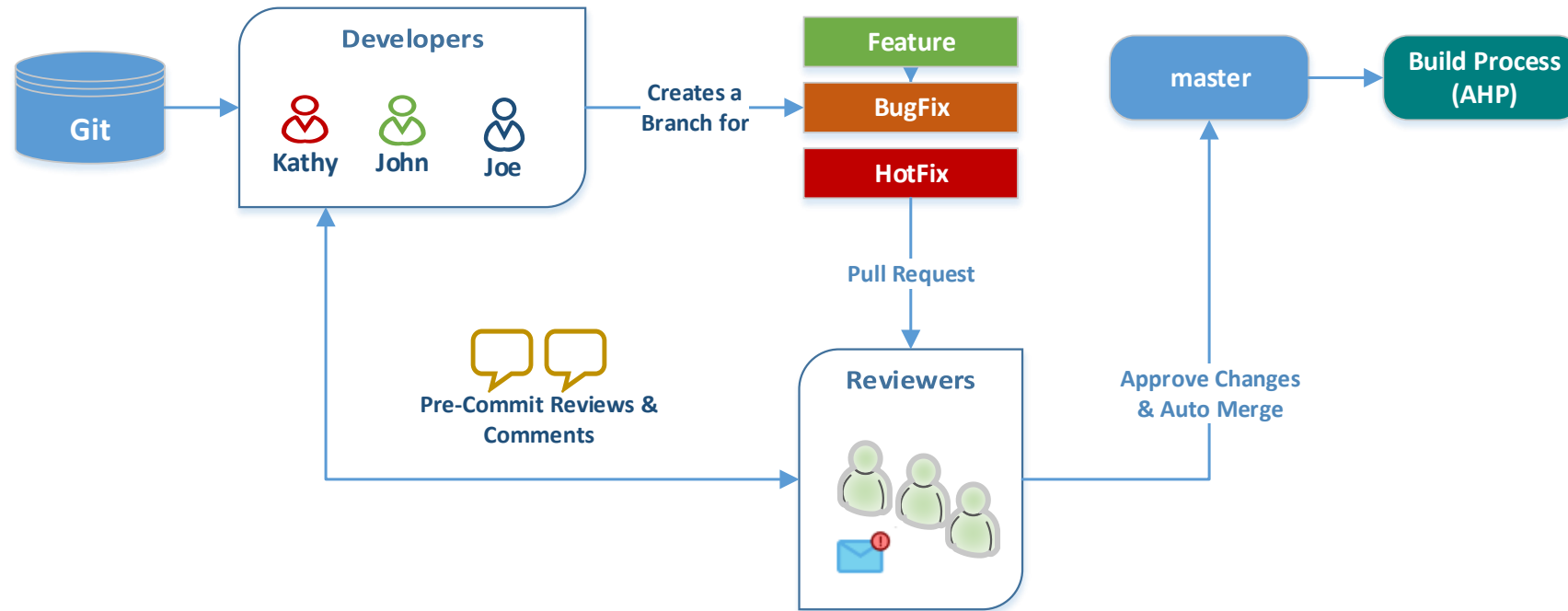
A mature software product can be in different stages :

- Already in Production (Prod Version)
- Bug fixes to current production version.
- New Features (future releases ex: 2019 release)

➔ This is where GitFlows are used.

# Branching and Release Strategy

- (not needed for this class)



# Git commands Summary

---

1. `git clone https://github.com/metcs/met-cs665-summer-2018-assignment-1-your\_git\_id.git`
2. *<<make your code edits using notepad/ide and Save>>*
3. `git status`
4. `git commit -a -m "your comments here"`
5. `git push`

# Git references

## Atlassian (creator of JIRA)

<https://www.atlassian.com/dam/jcr:8132028b-024f-4b6b-953e-e68fcce0c5fa/atlassian-git-cheatsheet.pdf>

<code>git config --global --edit</code>	Open the global configuration file in a text editor for manual editing.
<b>git log</b>	
<code>git log &lt;limit&gt;</code>	Limit number of commits by <limit>. E.g. <code>git log -5</code> will limit to 5 commits.
<code>git log --oneline</code>	Condense each commit to a single line.
<code>git log -p</code>	Display the full diff of each commit.
<code>git log --stat</code>	Include which files were altered and the relative number of lines that were added or deleted from each of them.
<code>git log --author= "pattern"</code>	Search for commits by a particular author.
<code>git log --grep="pattern"</code>	Search for commits with a commit message that matches <pattern>.
<code>git log &lt;since&gt;..&lt;until&gt;</code>	Show commits that occur between <since> and <until>. Args can be a commit ID, branch name, HEAD, or any other kind of revision reference.
<code>git log -- &lt;file&gt;</code>	Only display commits that have the specified file.
<code>git log --graph --decorate</code>	<code>--graph</code> flag draws a text based graph of commits on left side of commit msgs. <code>--decorate</code> adds names of branches or tags of commits shown.

<code>git reset &lt;commit&gt;</code>	Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone.
<code>git reset --hard &lt;commit&gt;</code>	Same as previous, but resets both the staging area & working directory to match. Deletes uncommitted changes, and all commits after <commit>.
<b>git rebase</b>	
<code>git rebase -i &lt;base&gt;</code>	Interactively rebase current branch onto <base>. Launches editor to enter commands for how each commit will be transferred to the new base.
<b>git pull</b>	
<code>git pull --rebase &lt;remote&gt;</code>	Fetch the remote's copy of current branch and rebases it into the local copy. Uses git rebase instead of merge to integrate the branches.
<b>git push</b>	
<code>git push &lt;remote&gt; --force</code>	Forces the <code>git push</code> even if it results in a non-fast-forward merge. Do not use the <code>--force</code> flag unless you're absolutely sure you know what you're doing.
<code>git push &lt;remote&gt; --all</code>	Push all of your local branches to the specified remote.
<code>git push &lt;remote&gt; --tags</code>	Tags aren't automatically pushed when you push a branch or use the <code>--all</code> flag. The <code>--tags</code> flag sends all of your local tags to the remote repo.



Visit [atlassian.com/git](https://atlassian.com/git) for more information, training, and tutorials

# UML & Class Diagram- Refresher

# Overview

---

## What is UML ?

The Unified Modeling Language (UML) is a general-purpose, developmental, visual annotation language.

In layman's term:

Think of it as a blue print tool(modeling language) to build software.

# UML Types:

---

- **Structure Diagrams** include the **Class Diagram**, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.
- **Behavior Diagrams** include the Use Case Diagram (used by some methodologies during requirements gathering); **Activity Diagram**, and State Machine Diagram.
- **Interaction Diagrams**, all derived from the more general Behavior Diagram, include the **Sequence Diagram**, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

# Class diagram - Basics

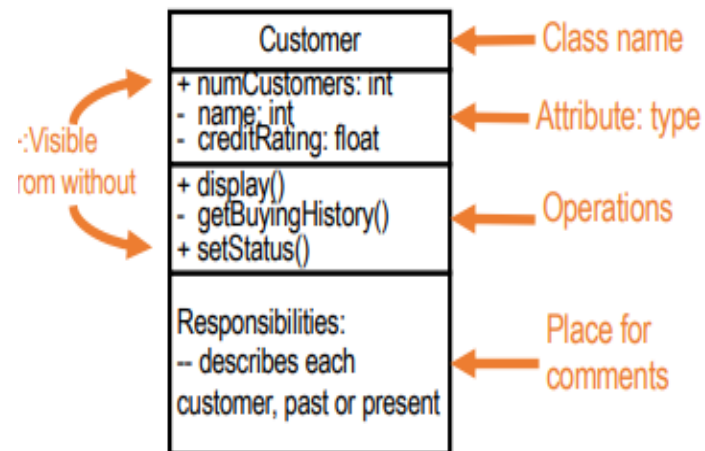
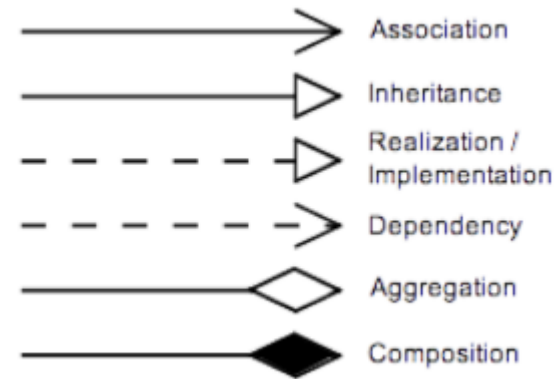
## Class Structure



## Visibility

- private  
+ public  
# protected  
~ package level

## Relationships



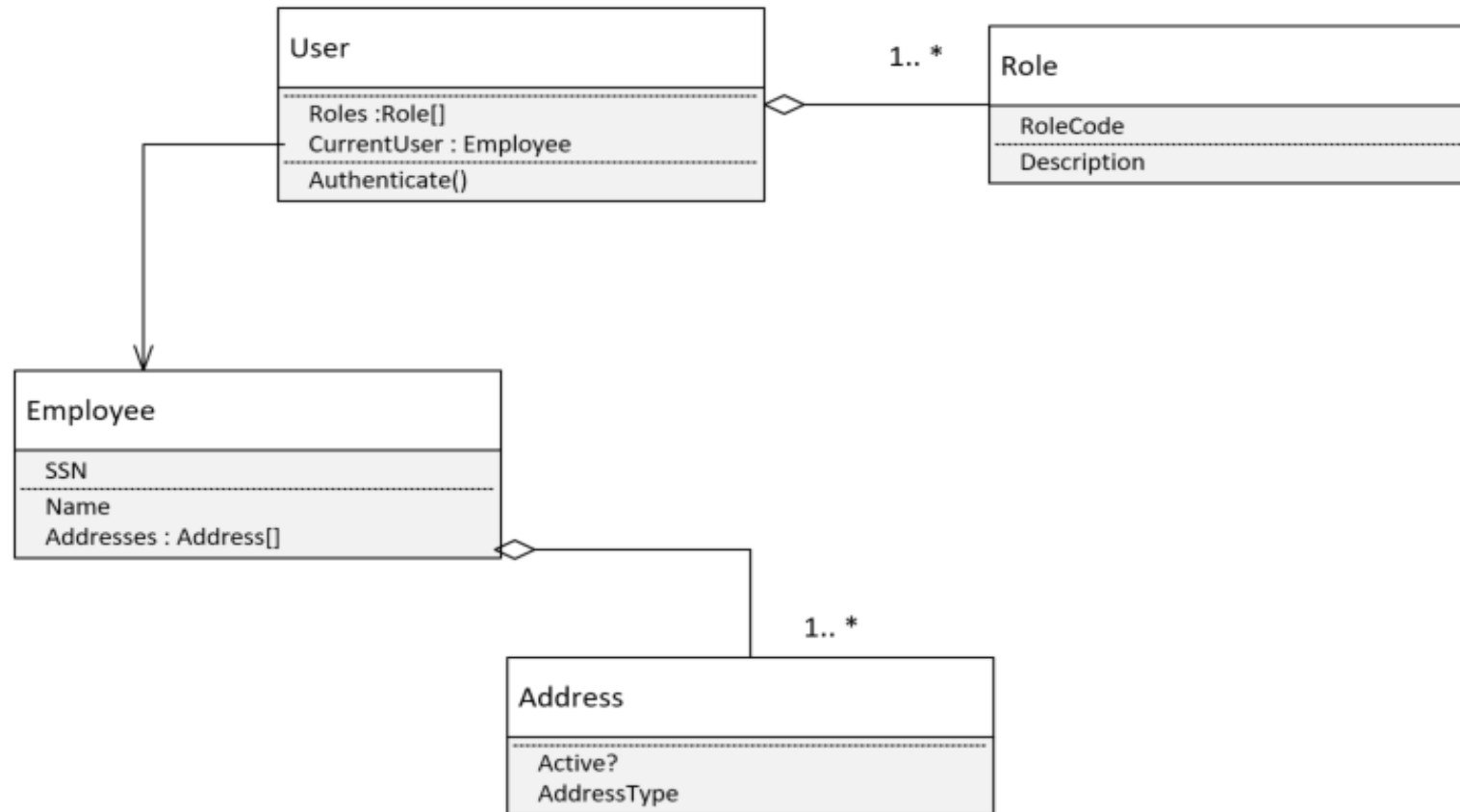


# Design a Tax System :

---

1. **Tax Payers** can file a return for themselves (each year)
2. A Tax Payer could hire a CPA
3. A **CPA** can file taxes on behalf of a taxpayer
4. A **Fiduciary** or **Trustee** can file taxes
5. IRS **Employee** can login to the system and act as a Taxpayer – Customer service
6. **Employees** can file their OWN personal taxes
7. A **CPA** could file their own **Personal** taxes or **Business** tax

# Example 1 : Simple Authentication



# Security Requirements

---

- Record Read/Writes are strictly controlled by SSN# and Access
- Fiduciary acting on behalf of a taxpayer should be able to act as a Taxpayer
- CPA should be able to login as a taxpayer
- IRS Employees
  - Auditors and Customer Service Rep should be able to login as the taxpayer
  - Restrict employees accessing their own returns during business hours (conflict of interest)

# Identify Tax System Actors

---

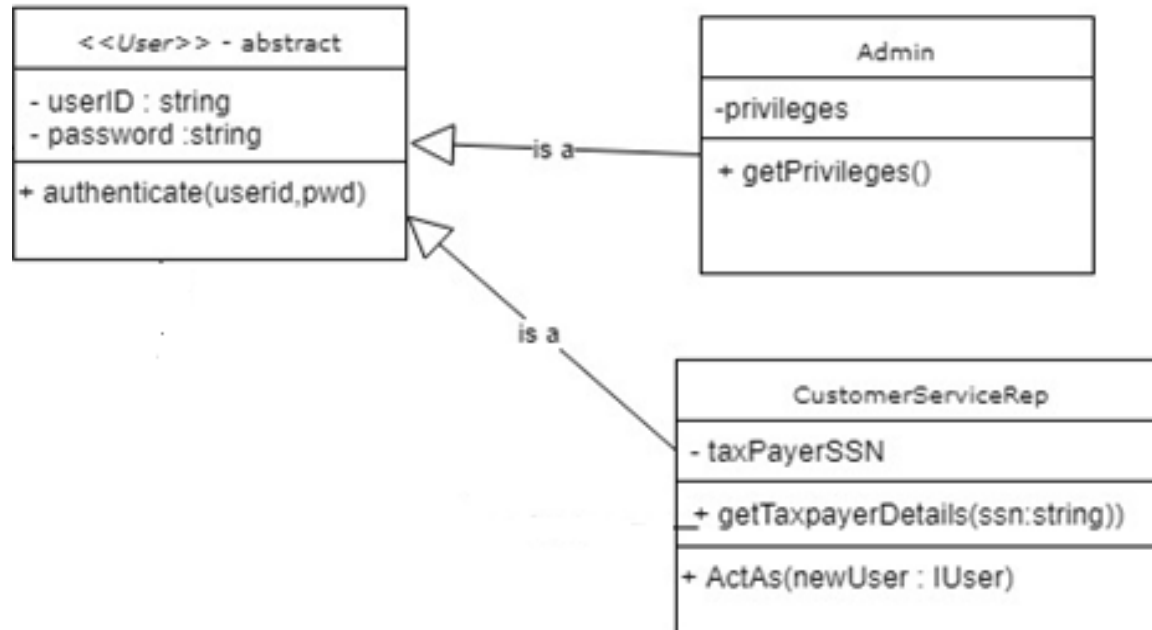
- Individual Taxpayer
- Fiduciary
- Trustees
- CPA's
- IRS Employees acting as Customer Service Rep
- IRS System Admins
- IRS Employees filing their own taxes (on-premise access not allowed during regular work hours)
- Business Taxpayer
  - Ability to allow access to Multiple Employees with in their organization

# Inheritance

Inheritance enables new classes to receive—or *inherit*—the properties and methods of existing classes.

## Business Rule:

- Admins Should be able to Login but should not be able to access TaxInfo
- Customer Service Rep Should be able to Login, but should not be able to access AdminInfo



## Design Intention:

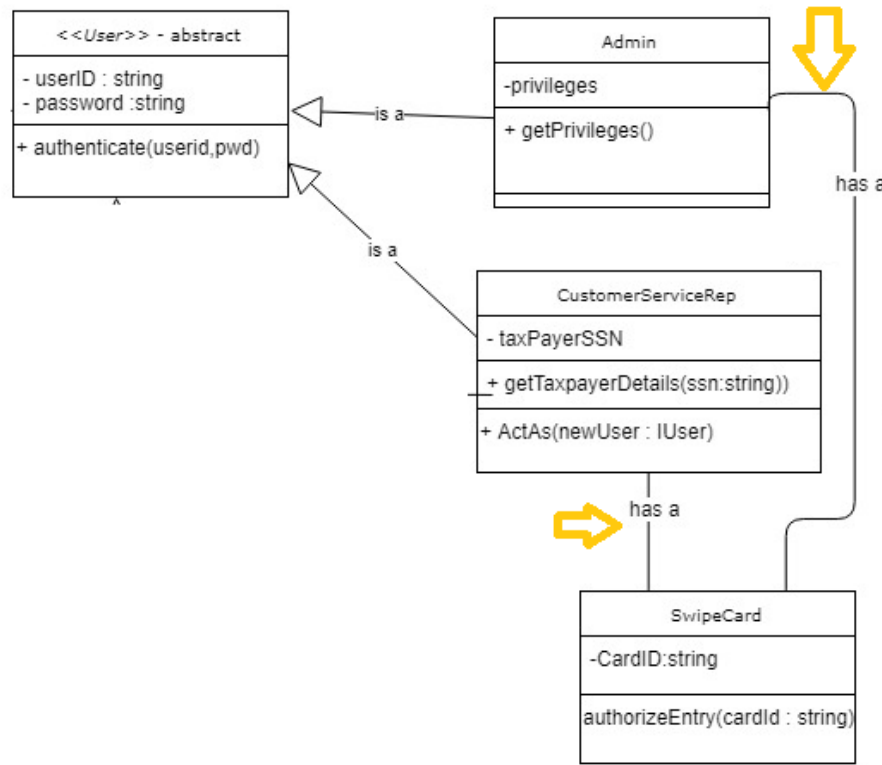
1. Let the user authenticate, but
2. Prevent admin accessing TaxInfo
3. Prevent CustomerRep accessing AdminInfo

# Association

**association** defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf.

## Business Rule:

**Admin or CustomerService Rep will have a Swipe Card to Enter the building**



## Design Intention:

1. Admin "has a" Swipe Card
2. Customer Rep "has a" Swipe Card.
3. Remote Employee may not need a swipe card.

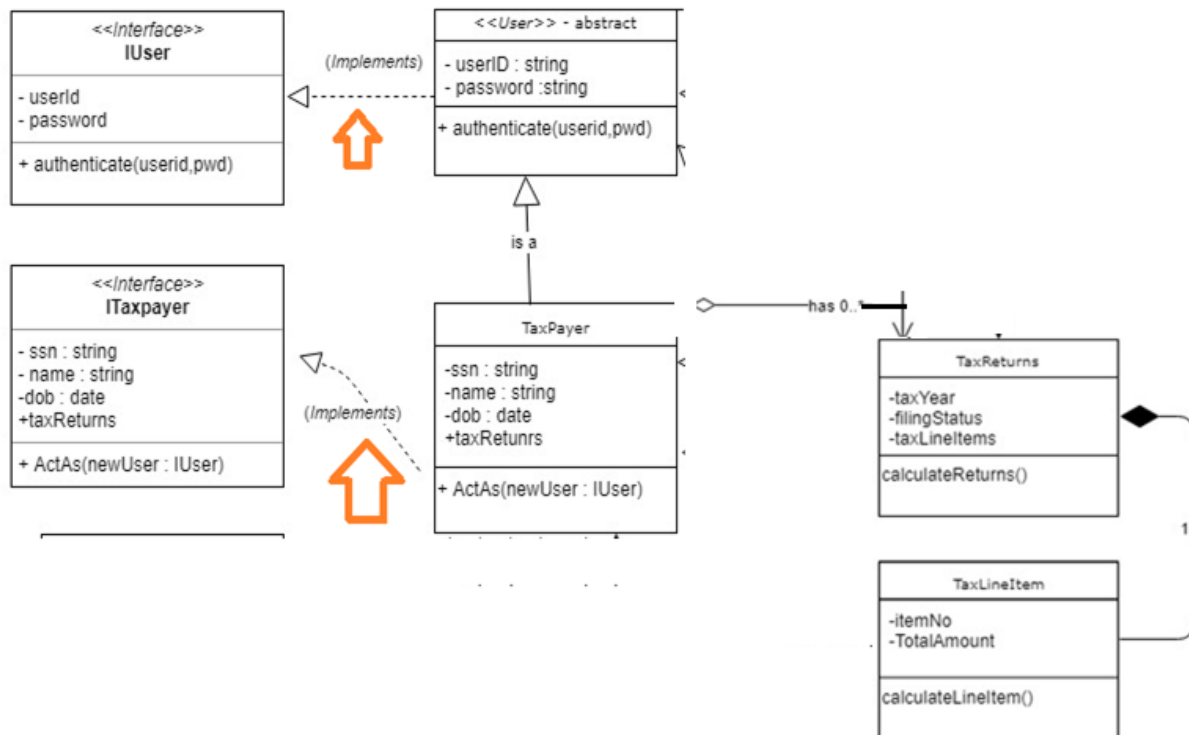
i.e: Employee and Swipe Card objects can exist without each other.

# Realization

- Realization is where a class satisfies an interface

## Business Rule:

Every TaxPayer has to have SSN, Name and DOB and should force the calling class to implement “ActAsUser() Method.



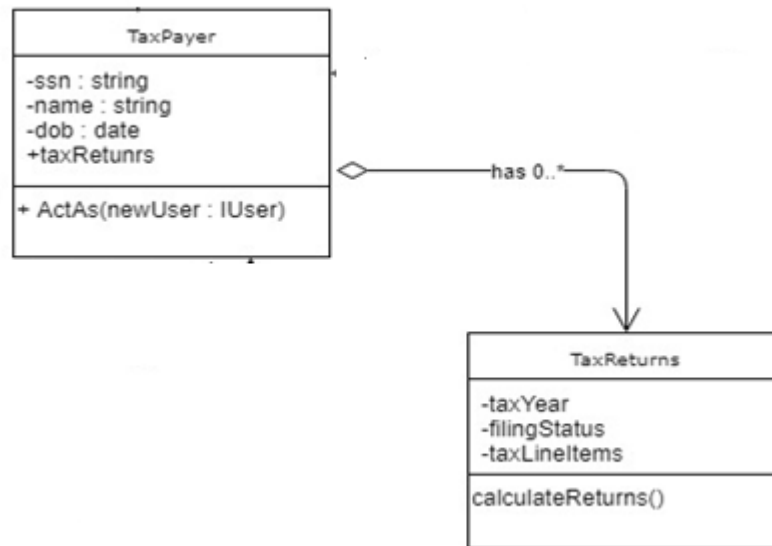
## Design Intention:

1. A taxpayer object should be able to “ActAs()” another taxpayer

# Aggregation

**Aggregation** implies a relationship where the child can exist independently of the parent

**Business Rule: A Taxpayer is a user & may file tax return for a specific tax year.**



**Design Intention:**

1. A **TaxPayer** may have 0..\* tax returns

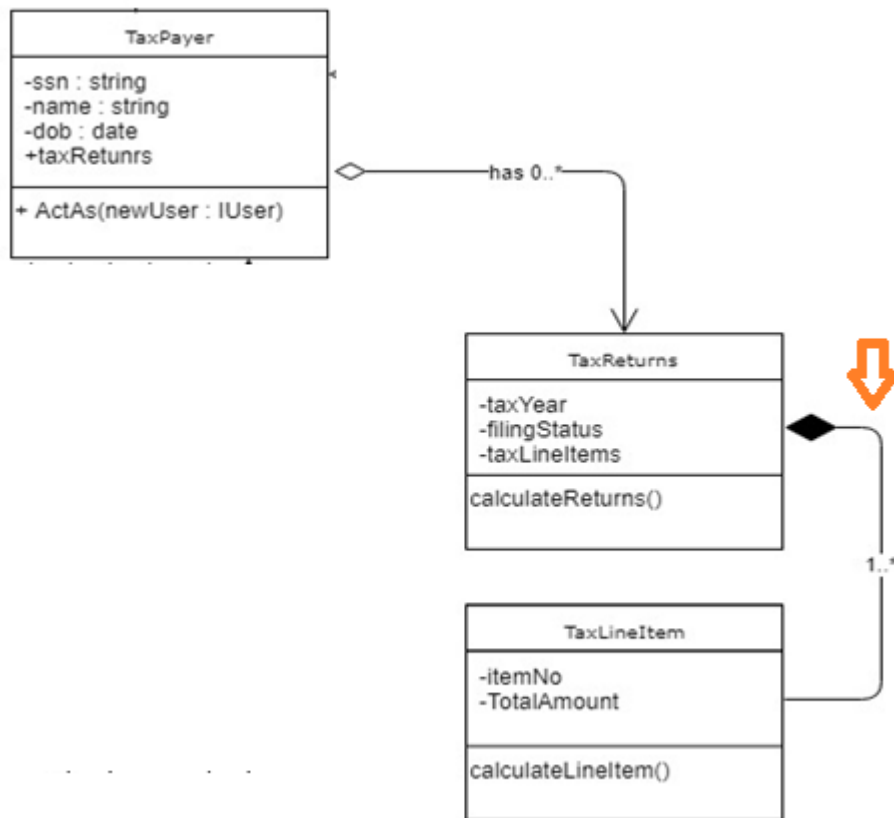
**TaxPayer "uses" TaxReturns = Aggregation :**  
TaxReturns exists independently (conceptually) from TaxPayer



# Composition

**Composition** is a "strong" form of aggregation. if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it

**Business Rule: A *TaxLineItem* cannot exist without a *TaxReturn***



## Design Intention:

1. A **TaxPayer** may have zero to many tax returns
2. But a **TaxReturn** must have at least one **TaxLineItem**

## Difference between Composition vs Aggregation:

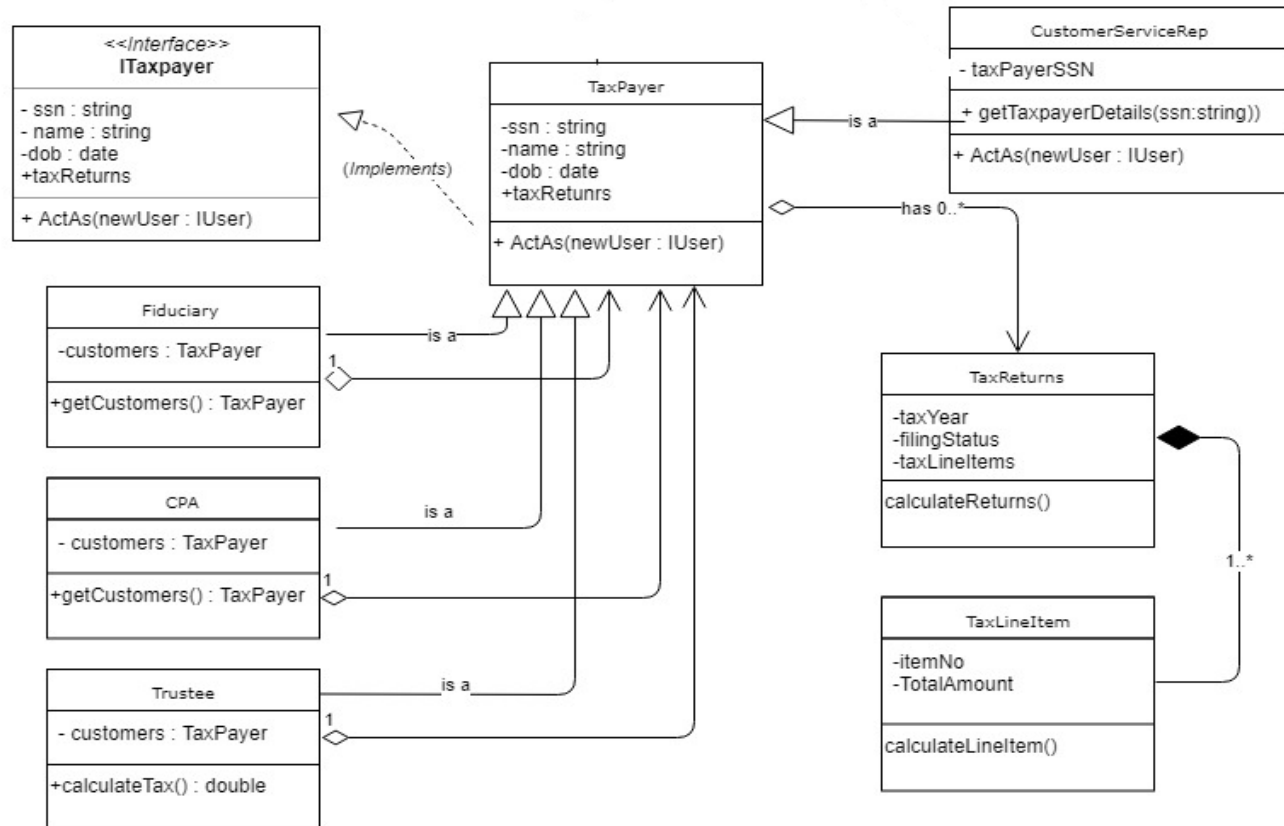
**TaxReturns** "owns" **TaxLineItems** => **Composition**

**TaxLineItem** has no meaning or purpose in the system without **TaxReturn**

**TaxPayer** "uses" **TaxReturns** => **Aggregation** : **TaxReturns** exists independently (conceptually) from **TaxPayer**

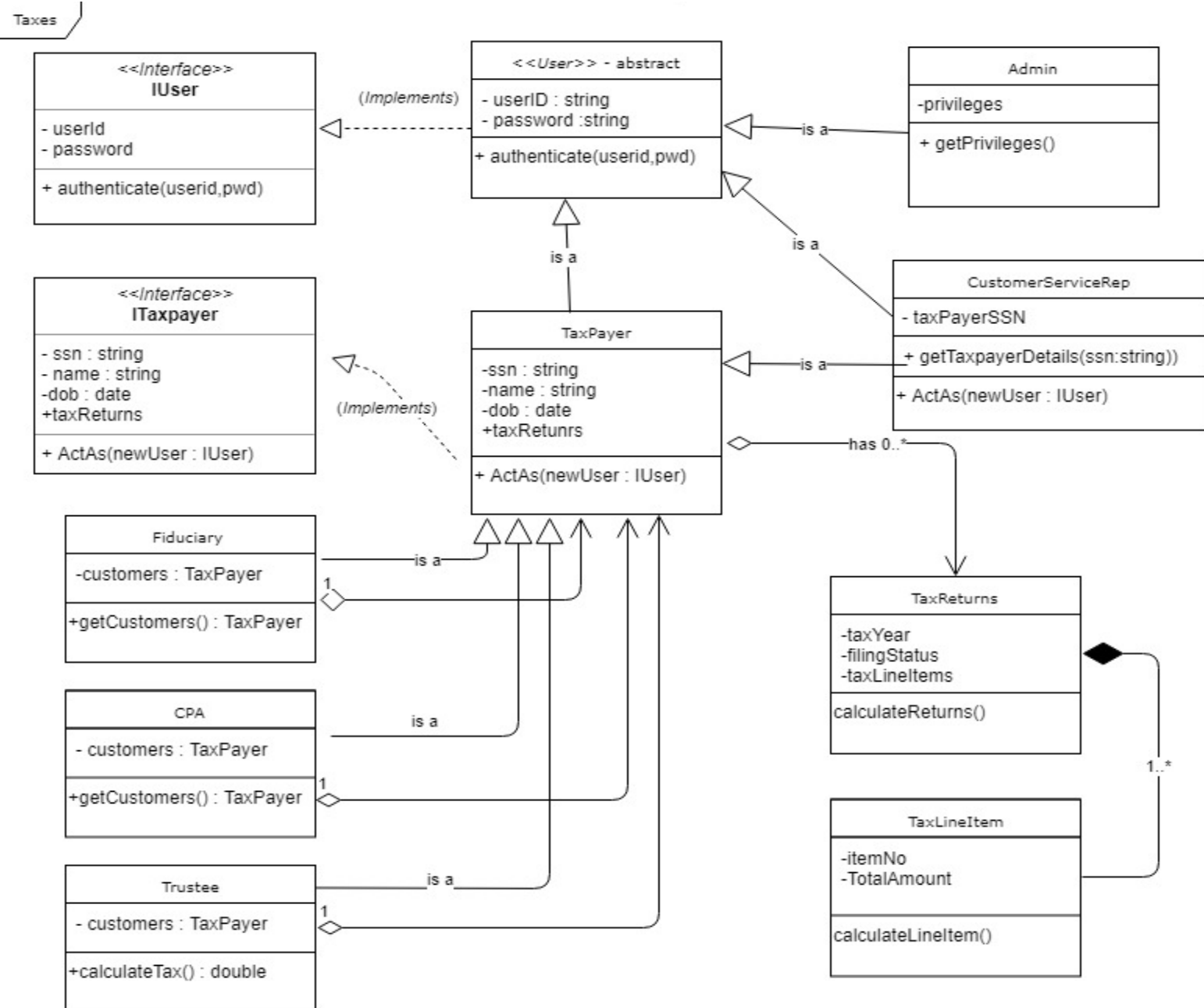
# Different Taxpayer types

**Business Rule: CPA, Fiduciary, Trustees and Employees should be able to access tax return for themselves or their customers**



## Design Intention:

- Employee should be able to proxy as i.e “ActAs” any taxpayer
- CPAs, Trustees and Fiduciary users should be able to access their own returns and their customer returns.



# How to Draw Class diagrams?

---

<https://draw.io>

# Assignment 1 - Review

---