Git Guide

Monday, 18 February 2019 10:40 PM

Setting up git in your local machine

Linux

Git generally comes pre-installed with Linux. You can find out by typing

```
git status
```

If you get a 'fatal: Not a git repository (or any of the parent directories): .git', you have git.
If you get something like 'bash: git: command not recognized' then go to
https://git-scm.com/download/linux
And follow the relevant instructions to set up git.

Mac

Install homebrew first, and then type

```
brew install git
```

Authenticating

Open the terminal, and type in (with the quotes)

```
git config --global user.name "Your Name"
git config --global user.email "email@example.com"
```

If you do not have an SSH key, "No such file or directory" will pop up. If so, type in

```
ssh-keygen -t rsa -C "your_email@example.com"
```

and accept the defaults by pressing **ENTER**.
Next, type in

```
cat ~/.ssh/id_rsa.pub
```

You will see a very large hash key. That is your SSH key and one way git can verify that it is indeed you.
Go to your git account > Settings > SSH and GPG Keys > New SSH Key
Paste that ssh key and give it a title that tells you which machine it is, for example "CSE Machine", "personal-mac", etc.

Getting Familiar with git

Git knows what you did last summer.

Git, as you may have heard, is a version control system.

In the company that you work for, the software engineers (and that includes yourself) have worked hard to get a minimum viable product running, and it was a huge success. After tasting success in version 1.0 of your product, you now want to add more features to it when version 2.0 comes out. And thus you delve into your mugs of coffee and work even harder to add the features. Version 2.0 is now ready and you release it.

As it turns out, version 2.0 has some serious bugs in it that somehow passed your tests and is now running in production. You guys now need to roll back to version 1.0, which is a stable version and is yet to produce any serious bugs.

Such a scenario is quite common in any place where software is used, and thanks to git, we can roll back to a stable version of our choice and keep working on fixing the bugs before releasing it.

Create a new repository in your remote git account (the online GitHub/bitbucket account) . For now, let's go to github.com and login. Then from your dashboard, locate the button "New Repository". Fill in the name and description fields and create the repository as private.

Initialize local repository

Initialize a git repository in your local machine. First create a folder for your new local repository.

```
mkdir proj_name
cd proj_name
```

Now, type the following command to initialise an empty repository.

```
git init
```

Running the command should display the following message:

```
Initialized empty Git repository in /home/path/to/project/.git/
```

If you run, **`ls -a`**, you should be able to see a new **`.git`** folder in the current project directory. Git stores all the tracking information in that folder.

Let's add a ReadMe file so we have something to work with. Open a ReadMe file and write "Starter ReadMe File"

2. Type touch **`ReadMe.txt`** to create a file named **`ReadMe.txt`**
3. Now open the file with your preferred text editor and add the line **`initial edit`**
4. Save the file and close it
5. Git isn't aware of this file yet. To track this file using git, type **`git add ReadMe.txt`**
6. Commit your changes with a meaningful message. To do that, run the command **`git commit -m "Added a new read me file"`**

Type **`git log`** to check your commits. You should be able to see your last commit. As you add features in your project or fix bugs, add and commit your changes. This will allow you to go back to your working code if your current code doesn't work as intended.

Initialize remote repository and add to local

If you want to work on a group project, you'll need to collaborate with your friends. This is where centralised repository hosting services like GitHub or BitBucket come into play. They provide version control and source code management based on Git and add extra features to assist the development cycle. While Git is a command line tool, these separate online services provide a web-based graphical interface and several collaboration features such as task management tools. In this course, we will be using GitHub.

Before you start working on GitHub, make sure you have everything setup first. There are instructions for getting started at the top of this week's lab.

Now, create a new repository in your remote git account (GitHub account in this case) so that you can link it with your local repository. To do this, go to github.com and login. Then from your dashboard, locate the button "New Repository". Fill in the name and description fields and create the repository as private. This online GitHub repository will now be referred to as your remote project.

In your remote project, you will see a button "Clone or download". Click on it, click on "Use SSH".

Copy that url. It will typically look like this: **`git@github.com:Your_id/Repository_name.git`**
Run the commands:

```
git remote add origin remote_repository_url
git remote -v (to verify)
```

Now, let's push the ReadMe file that we added before to Github so that everyone can see it.

```
git push -u origin master
```

And you have made your first push. Now let us add some code to it so that we can see the actual functionalities of git.

Cloning

Cloning an existing repository is fairly straightforward. Go to the website, and find the green 'Clone or Download' button. Click on it and copy the clone url, which looks likehttps://github.com/username/repository.git

After that, open up a terminal and type

```
git clone <remote_repository_url>
```

The repository will be downloaded into your local machine.

Committing and Adding

Let's add a very simple minimal Hello World application. For this, create a file named "hello_word.py" and inside, write **`print("Hello, World!")`**
Save the file and do the following

```
git add . (Stage the directory)
```

```
git commit -m “Added a minimal application”(Add a commit message)
git push (Push to remote)
```

If you look into your remote git account, you should be able to see the new file added

Branching out

If you are to collaborate on a project, or you want to add a new feature but don't want to mess up the old features, you would want to create a new branch and make your changes.
To create a branch, type in

```
git checkout -b new-branch-name
```

Note that the **-b** flag is needed only when creating a branch. After it is created, you can just switch into it by typing in

```
git checkout new-branch-name
```

Now, make some changes to your file. For instance, add a comment like “A change exclusive to this branch only”.

```
git add .
git commit -m “Added a comment in this branch”
git push
```

Notice that it will show that everything is up to date, even though you had made changes. That is because your new branch exists only locally, and not remotely. Not yet anyway. So by default git tried to push local master to remote master. In order to tackle that, do

```
git push --set-upstream origin new-branch-name
```

or

```
git push -u origin new_branch_name
```

(Note that this is only needed when you are pushing for the first time from a branch. After that, git push will suffice).
To notice the differences, type in

```
git diff master
```

At any point, to check which branch you are on, type in

```
git branch
```

To check changes that you are yet to commit, type in

```
git status
```

Merge conflict

A git pull a day keeps the conflicts away.

Merge conflicts happen when two people try to push together (not necessarily at the same time, but after the same pull) and they have different code in certain parts of the code, then git doesn't know which change to accept, so it throws merge conflicts. The larger the project is, the harder it is to fix. Let us look at an example.

By default, all git repositories have the branch ‘master’. Let us make another branch called **dev**.

Before making the branch **dev**, open a hello-world.py file and put in
print(“This is a git branch”)

Add, commit and push.

Then run

```
git checkout -b dev
```

This will create a new **dev** branch and move you to the new branch. If you do **git branch** now, you'll notice that your current branch is **dev**. While in dev, underneath line 1, add the following line
“This is branch dev”

Add, commit and push to origin dev.

Go back to **master** by running **git checkout master** and just as before, add the line
“This is branch master”

Add, commit and push to origin master.

Now, type in the following commands

```
git merge dev
```

This should bring up a merge conflict message. In order to fix your merge conflicts, go to your file and locate the '<<<<<<<' and '>>>>>>>' and pick the one that you think is appropriate. You can even keep both if you wish. Remember to remove the merge conflict markers when you update the file.

Add, commit with the message "Fixed merge conflict from branch dev" and push to **`origin master`**.

There are several other ways to get merge conflicts.

In order to avoid merge conflicts, make sure you don't leave uncommitted changes at the end of the day and most importantly, at the start of the day, do a git pull.

From <https://webcms3.cse.unsw.edu.au/static/uploads/course/COMP1531/19T1/6df740f5b0f42613d5d84512e9d572b22a372d709502c5f91dba4d80eea7d71c/Git_Basics_Guide.html>

Introduction to Software Engineering

Tuesday, 19 February 2019 4:13 PM

What is software engineering?

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software." [IEEE]

Software Engineering is a discipline that enables customers to achieve **business goals** through developing software-based systems to solve their **business problems** e.g., develop a course enrolment application or a software to manage inventory.

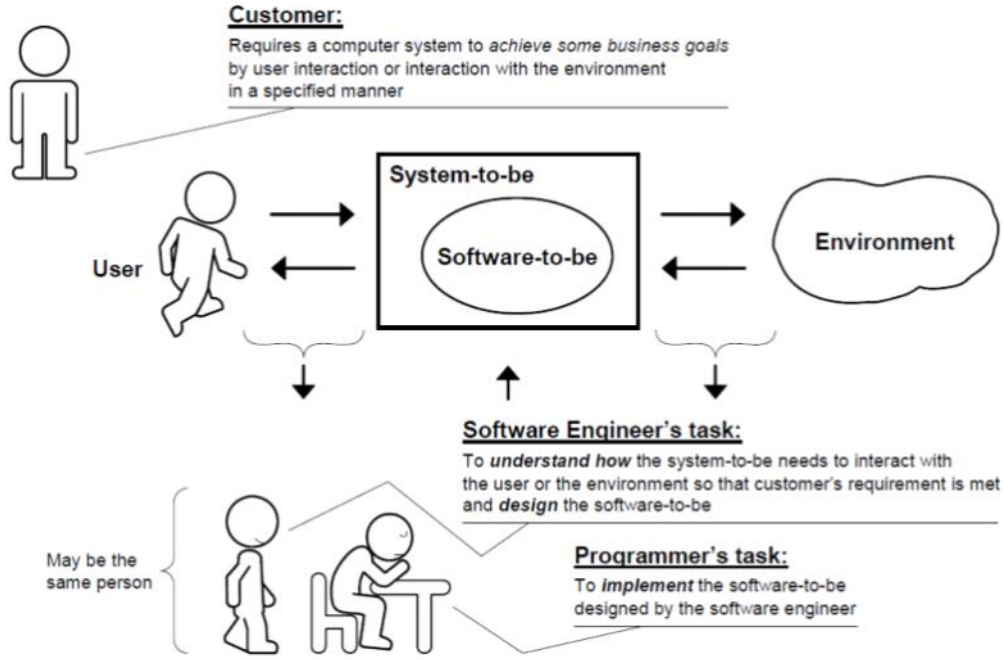
This discipline places great emphasis on the methodology or the method for managing the development process. The methodology is commonly referred to as Software Development Life-Cycle (SDLC).

Software Engineering vs Programming

Software Engineering	Programming
• **Understanding** the business problem (understanding the interaction between the system-to-be, its users and environment) • **Creative formulation** of ideas to solve the problem based on this understanding • **Designing** the "blueprint" or architecture of the solution	• **Implementing** the "blueprint" designed by the software engineer

Role of a Software Engineer

A software engineer acts as a bridge from customer **needs** (problem domain) to programming **implementation** (solution domain). This enables the software engineer to design solutions that accurately target the customer's needs, that is, deliver value to the customer.



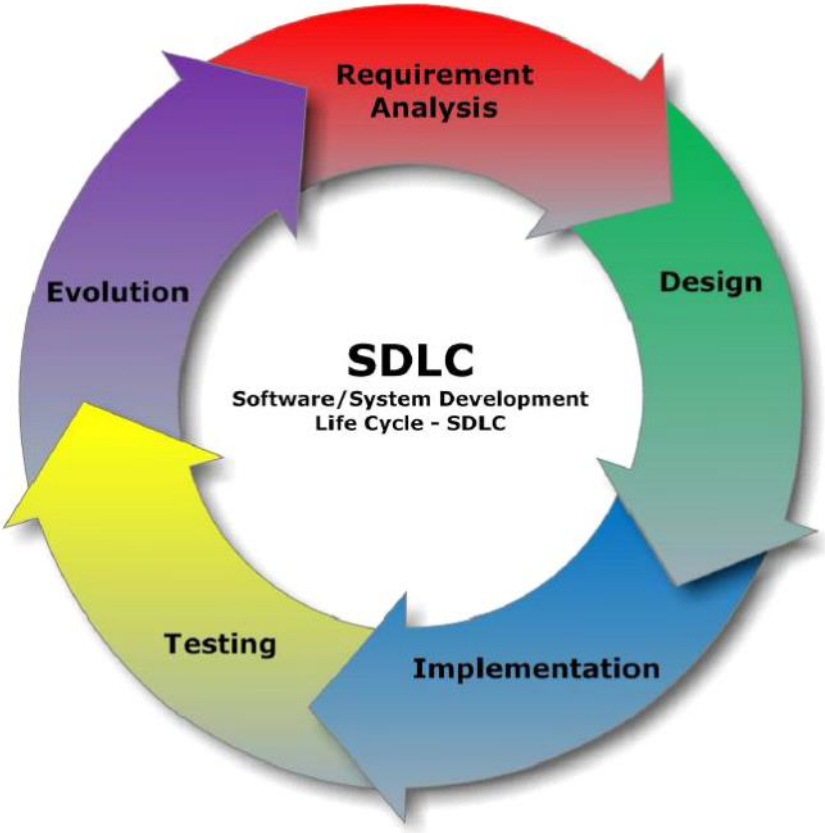
Software Development Life-Cycle

We described software engineering as a complex, organised process with a great emphasis on ***methodology***. This ***methodology*** is essentially a framework to structure, plan and control the development of the software system and typically consists of the following phases:

- Analysis and Specification
- Design
- Implementation
- Testing
- Release & Maintenance

Each of the above phases can be accompanied by an artifact or deliverable to be achieved at the completion of this

phase.



SDLC - Requirement Analysis

1. Analysis:
    - Discover and learn about the **problem domain** and the "**system-to-be**" where software engineers need to:
        - Analyse the problem, understand the problem definition – what features/services does the system need to provide? (***behavioural characteristics or external behaviour***)
        - Determine both functional (inputs and outputs) and non-functional requirements (performance, security, quality, maintainability, extensibility etc.)
        - Use-case modelling, user-stories are popular techniques for analysing and documenting the customer requirements
2. Design:
    - A problem-solving activity that involves a "creative process of ***searching how to implement all of the customer's requirements***"
    - Plan out the system's "internal structure or structural characteristics" that delivers the system's "external behaviour" specified in the previous phase
    - Produce ***software blue-prints*** (design artifacts e.g., domain model)
    - Often the design phase overlaps with previous phase
3. Implementation:
    - Encode the design in a programming language to deliver a software product
4. Testing:
    - Verify that the system works correctly and realises the goals
    - Testing process encompasses:
        - Unit tests (individual components are tested)
        - Integration tests (the whole system is testing)
        - User acceptance tests (the system achieves the customer requirements)
5. Release and Maintenance:
    - Deploy the system, fix defects and add new functionality

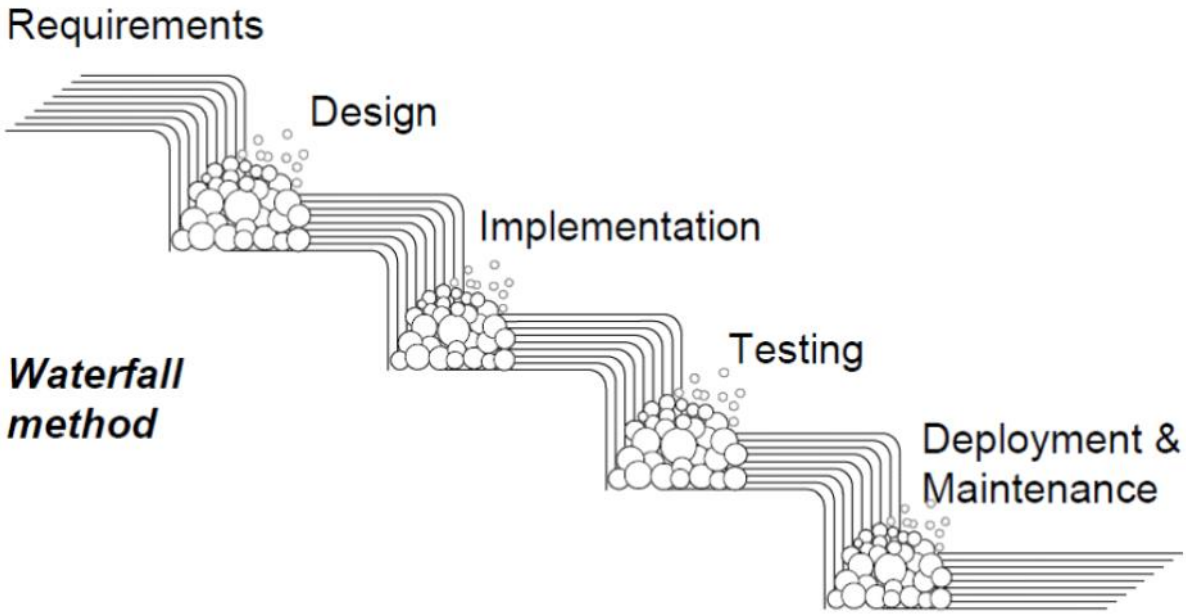
Software Development Methodologies

A software development methodology prescribes how the different phases in SDLC are carried out and can be grouped into two broad categories.

- ***Waterfall Process*** - a linear process, where the various SDLC phases are carried out in a sequential manner.

- ***Iterative & Incremental Process*** - development increments functionality and repeats the process in a feedback loop. e.g. Rational Unified Process, Agile methods (e.g. SCRUM, XP)

Waterfall Model (1970's)



Waterfall process for software development.

The Waterfall Model is a linear, sequential life-cycle (*plan-driven development model*) characterised by detailed planning.

1. The problem is identified, documented and designed
2. Implementation tasks are identified, scoped and scheduled
3. Development cycle is followed by testing cycle

Each phase must be fully completed, documented and signed off before moving on to the next phase. It is simple to understand and manage due to ***project visibility***. This approach is suitable for projects that require minimised risk with **stable** product statements, well-known requirements with no ambiguities, clear technical requirements. It is also suitable where resources ample or for mission-critical applications, where requirements are more likely to be clearly defined (e.g. NASA)

The drawbacks of the Waterfall Model include:

- No working software produced until late into the software life-cycle
- Rigid and not very flexible
    - No support for fine-tuning of requirements through the cycle
    - Good ideas need to identified upfront; a great idea during the release cycle is a **threat**!
    - All requirements are set in stone at the end of the requirements phase, once the application is implemented and in the "testing" phase, it is difficult to retract and change something that was not "well-thought out" in the concept phase or design phase
- Heavy documentation (typically model based artifacts, UML)
- Incurs a large management overhead
- Not suitable for projects where requirements are at a moderate risk of changing

Software Development Challenges

Software is:

- probably, the most complex artifact
- intangible and hard to visualise
- the most flexible artifact – radically modified at any stage of software development when customer changes requirements

The Waterfall Model prescribes a sequential process, but this linear order does not always produce best results. Easier to understand a complex problem by implementing and evaluating pilot solutions.

Incremental and Iterative Project Life-Cycle

Incremental and iterative approaches:

1. Break the big problem down into smaller pieces and prioritize them.

2. An “iteration” refers to a step in the life-cycle
3. Each “iteration” results in an “increment” or progress through the overall project
4. Seek the customer feedback and change course based on improved understanding at the end of each iteration

An incremental and iterative process

- seeks to get to a working instance as soon as possible.
- progressively deepen the understanding or “visualization” of the target product

Incremental and Iterative Models

Unified Software Development Process (Ivar Jacobson, Grady Booch and James Rumbaugh)

- an iterative software development process where a system is progressively built and refined through multiple iterations, using feedback and adaptation
- each iteration will include requirements, analysis, design, and implementation
- Iterations are timeboxed

Rational Unified Process (A specific adaptation of Unified Process), evolved at IBM

Agile Methodologies (e.g., XP, SCRUM), that are more aggressive in terms of short iterations

Requirements Engineering Introduction

Tuesday, 26 February 2019 11:21 PM

A requirement is ***a condition or capability needed by a user to solve a problem or achieve an objective***.

A requirement is a short, concise statement that describes an aspect of what the proposed system should do or describe a constraint, and it must help solve the customer's problem.

A set of requirements as a whole represents a negotiated agreement among all stakeholders.

Functional vs Non-Functional Requirement

Functional Requirements	Non-Functional Requirements
Defines the functionality of the "system-to-be". It is the ***set of services*** provided by the system • What inputs the system should accept and under what conditions • The behaviour of the system • What outputs the system must produce and under what conditions	Describe the ***quality attributes*** of the "system-to-be" • The constraints of the functionality provided by the system (e.g. security, reliability, maintainability, efficiency, portability, scalability)

Example: a cell phone
Functional requirements: calling, texting, emailing, taking photos.
These features are based on the user-interaction with the phone.
Non-functional requirements: good battery life, access to networks, large internal memory, will it break when you drop it?
These are features the should perform in the user's environment.

Metrics of Non-Functional Requirements
Non-functional requirements are quantifiable and must have a measurable way to assess if the requirement is met (metrics)

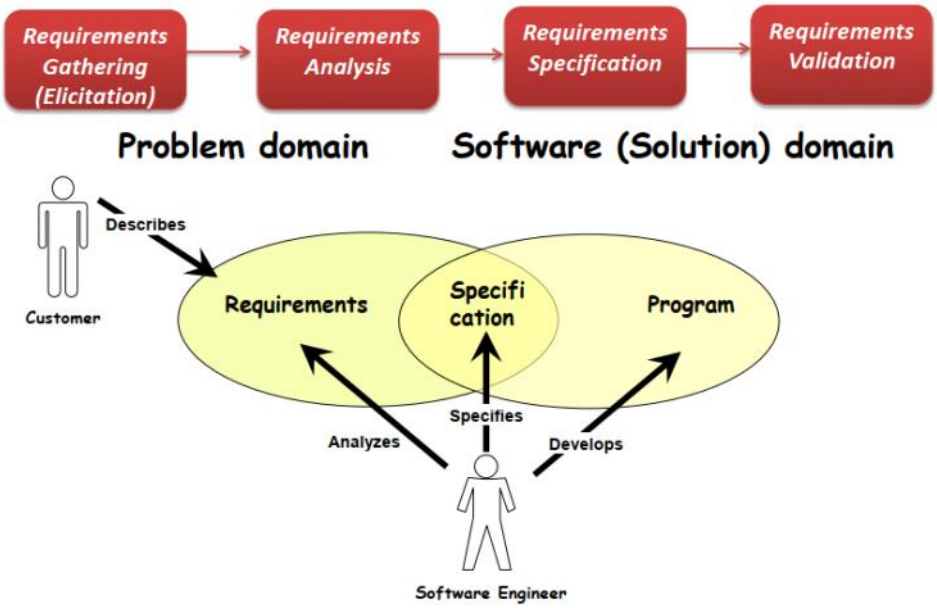
- Performance (user response time or network latency measured in seconds, transaction rate (#transactions/sec)
- Reliability (MTBW – mean time between failures, downtime probability, failure rate, availability)
- Usability (training time, number of clicks)
- Portability (% of non-portable code)

Requirements Engineering

Requirements engineering is a ***set of activities*** concerned with identifying and communicating the purpose of a software system and the context it will be used.
It is a ***negotiation process***, where potential users of the system decide what they want and need the system to do. Software engineers formulate a well-defined problem to solve. The problem consists of a set of criteria (requirements) which proposed solutions would either definitely solve or fail to solve.

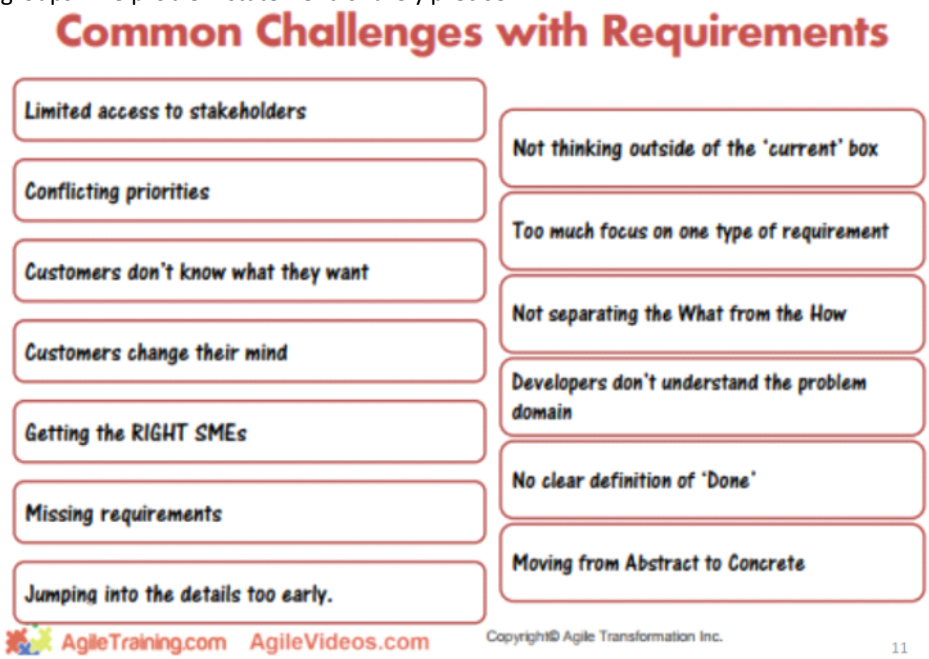
End users interested in the system, customers (business owners interested in cost and timelines) and a design team (software engineers, architects and developers) are those who typically participate in requirements engineering

Phases of Requirements Engineering



1. ***Requirements gathering*** is a process, where customers and end-users articulate, discover and understand their requirements.
   The customers specify:
   - What is required
   - How will the intended system fit into the context of their business
   - How the system will be used on a day to day basis

   Developers understand the business context through meetings, market research, questionnaires and focus groups. The problem statement is rarely precise.



2. ***Requirement analysis*** starts with the customer statements of requirements or the ***vision statement***. It is where we refine and reason about the requirements mentioned. Here, we scope the project, negotiate with the customer to determine the priorities (what is important, what is realistic). It is where we identify dependencies, conflicts and risks.

   There are two popular techniques in requirements analysis
   1. ***Use-case modelling*** - build a set of use-cases to describe the task to be performed by the system. Each use-case represents a dialog between user and system, helping the user achieve a goal. In each dialog, the user initiates an action and the system responds with a reaction. We can build elaborate user-scenarios for each use-case that describe the interaction between user and system.
   2. Developing ***user-stories*** (agile requirements analysis)
3. ***Requirements specification*** is where we document the functional and non-functional requirements using formal, structured notations or graphical representation to ensure clarity, consistency and completeness. This is often a combination of use-cases, user-stories, prototypes, formal mathematical models or a formal SRS (System Requirements Specification)
4. ***Requirements validation*** is the process of confirming with the customer or user of the software that the

specified requirements are valid, correct, and complete. This ensures that the developer's understanding of the problem matches the customer's expectations.

Note: the logical ordering of the above does not imply they are performed sequentially.

Sometimes we are given requirements that are relatively complex or are compound requirements. We can simply split the requirement into simpler requirements.

Test-Driven Development

TDD (test-driven development) specifies writing tests for requirements during the requirements analysis.

These tests are known as ***User Acceptance Tests (UAT)***. They capture the customer's assumptions about how the requirement will work and what could go wrong. They are also defined by the customer or developer in collaboration with the customer.

The customer can work with the developer to write the actual test cases. A ***test case*** is a particular choice of input values that will be used to test a program. The program will pass a test case if it provides expected output values. A ***test*** is a finite collection of test cases.

Agile Requirements Analysis and Specification With User-Stories

User Story

User stories are an approach used in agile software development to create requirements. It is a short and simple description of a feature or requirement narrated from the perspective of the person who desires the capability.

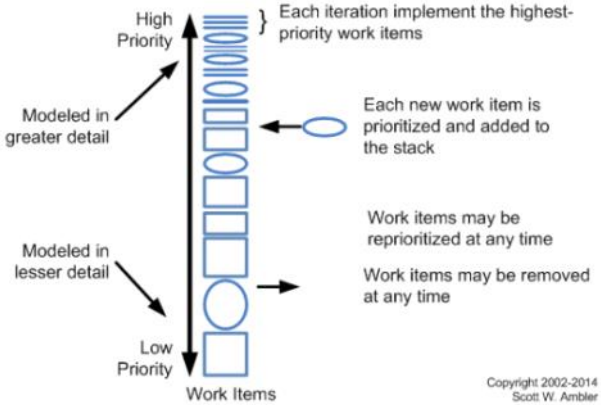
User stories use a ***RGB (Role-Goal-Benefit)*** template:
As a <type of user>, I want <some goal> so that <some reason>

User stories are a reminder to have a conversation with your customer. It shifts the focus from writing about the features to discussing what they need and why they need it. Anyone can write user stories; a product owner or even a team member, but it is the product owner's responsibility to make sure a ***product backlog of user stories*** exist.

User stories also use the simplest tools. They can be written on index cards or sticky notes, stored in a shoe-box, and arranged on walls or tables to facilitate panning and discussion.

Epic user stories cover a large amount of functionality and are generally too large for an agile team to complete in one iteration. A ***theme*** is a collection of related epic user story. Note that an epic user story does not necessarily have to cover what a specific user wants, but an overall collection of requirements, that go together.
User stories split an epic user story into multiple smaller atomic stories, so that the story is small enough to be coded and tested in one iteration. Then each user story should have ***confirmations*** or ***acceptance criteria*** that define when a story is "completed".

User stories not only capture the user's vision but also impact the ***planning process*** in two key areas; ***estimating*** and ***scheduling***.



Some important things to consider when writing user stories:

- Assign each user story a ***unique identifier***
- Remember non-functional requirements
    - ***Indicated the estimated size*** - assign user story points to each card and give a relative indication of how long it will take a pair of programmers to implement the story
- ***Indicate the priority***

Techniques to Write a User Story

As mentioned before, a common technique is to use the ***Role-Feature-Reason*** template or ***RGB (Role, Goal, Benefit)***, developed by Mike Cohn of Mountain Goat Software, 1991
*As a **< type of user >**, I want **< some goal >** so that **<some reason >***

Keep the focus on ***who***, ***what***, and ***why:***

- **Role (who):** describes *who* will be benefited by the feature, must clearly identify the specific type of user e.g., a manager, administrator, librarian, trainer, student etc.
- **Goal (what):** describes ***what*** the user wants from the perspective of the user and **not** from the perspective of the developer who will be coding it e.g., feature = “search for a book”, “search the course offering”
- **Benefit (why):** states why the user wants this feature. What benefit the user will get out of this feature? e.g. goal = “improve customer service”, “find an offering that interests me”. (If the value or benefit can’t be

articulated, it might be something that's not necessary)

The three C's model (Ron Jeffries, 2001), identifies three primary components of a user-story that helps business and technical team reach an agreement on the meaning of the user-story.

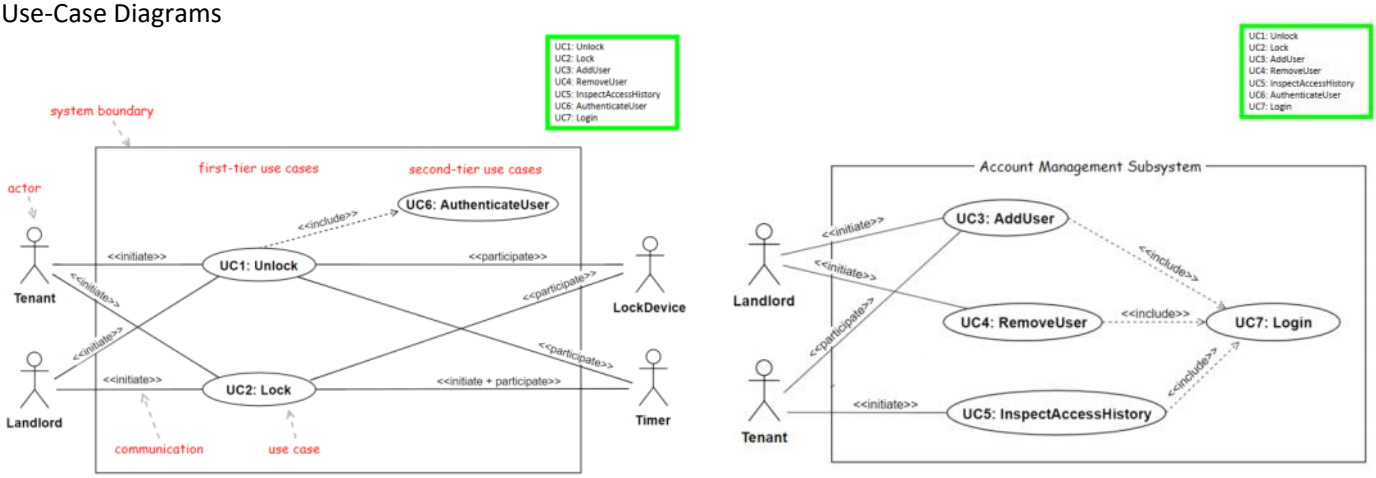
- a ***"Card"***, a simple statement usually written in the RGB format, addressing the "who", "what" and "why" on an index-card
- a ***"Conversation"***, detailing the simple requirement; *conversation* can take place at different times in the project, enables development team to obtain a clearer understanding of how the feature will work in different situations, including error conditions, the value being provided; conversation is largely verbal but most often supplemented by documentation
- the ***"Confirmation"***, developers need to get *confirmation* regarding the acceptance criteria from the product owner; define the acceptance tests, that will be used to show that the story has been implemented correctly

Use-Case Modelling

Use-Case Modelling is building a set of ***use-cases*** that describe the task to be performed by the "system-to-be" and relation between these tasks and the outside world. A ***use-case*** description represents a ***dialog*** between the user and the system, with the aim of helping the user achieve a ***goal***. Use-cases signify ***what*** the system needs to accomplish not ***how***;

Since use-cases help a user achieve ***goals***, each use-case name must include a ***verb*** capturing the goal achievement e.g. "withdraw cash".
Each use-case description represents a ***dialog***, where the user initiates ***actions*** and the system responds with ***reactions***.
Each use-case specifies what information must pass the boundary of the system in the course of a dialog (without considering what happens inside the system)

Use-Case Diagrams

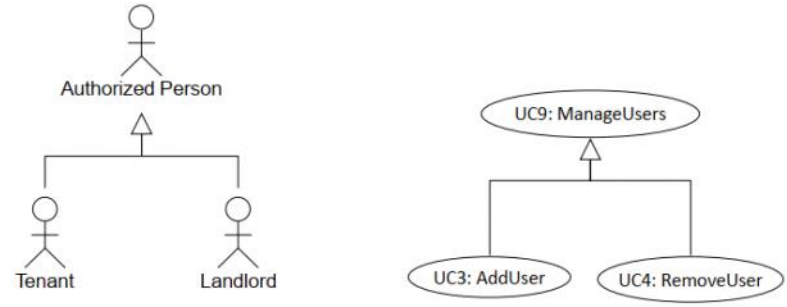


Types of actors:

- ***Initiating actor*** (aka ***primary actor*** or ***user***) initiates the use-case to achieve a goal
- ***Participating actor*** (aka ***secondary actor***) participates in the use-case but does not initiate it. It helps the system-to-be complete the use-case

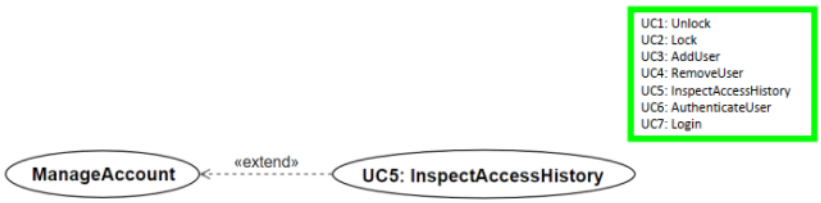
<<initiate>>
<<participate>>
<<include>>
<<extend>>

Use Case Generalisation:
More abstract representations can be derived from particular representations. For example, in our use-case diagram above, our tenant and landlord can simply be an authorised person. For our use-cases, adding and removing users can simply be managing users.



Actor Generalization Use Case Generalization

Optional Use Cases: «extend»



Key differences between «include» and «extend» relationships

	Included use case	Extending use case
Is this use case optional?	No	Yes
Is the base use case complete without this use case?	No	Yes
Is the execution of this use case conditional?	No	Yes
Does this use case change the behavior of the base use case?	No	Yes

Traceability Matrix

Mapping: System requirements to Use cases

REQ1: Keep door locked and auto-lock
REQ2: Lock when "LOCK" pressed
REQ3: Unlock when valid key provided
REQ4: Allow mistakes but prevent dictionary attacks
REQ5: Maintain a history log
REQ6: Adding/removing users at runtime

UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: AuthenticateUser
UC7: Login

Req't	PW	UC1	UC2	UC3	UC4	UC5	UC6	UC7
REQ1	5	X	X					
REQ2	2		X					
REQ3	5	X					X	
REQ4	4	X					X	
REQ5	2	X	X			X		
REQ6	1			X	X			X
Max PW		5	2	2	2	1	2	1
Total PW		15	3	2	2	3	2	3

Traceability refers to the property of a software artefact (use-case, a class etc) being traceable to the original requirement that motivated its existence.
Traceability matrices are continued through the domain model, design diagrams etc. In the context of use-cases, the matrix serves to:

- Check that all requirements are covered by use-cases
- Check that none of the use-case is introduced without a reason
- Priorities the work on use cases

Usage Scenario: Schema for Detailed Use Cases

Use Case UC-#:	Name / Identifier [verb phrase]
Related Requirements:	List of the requirements that are addressed by this use case
Initiating Actor:	Actor who initiates interaction with the system to accomplish a goal
Actor's Goal:	Informal description of the initiating actor's goal
Participating Actors:	Actors that will help achieve the goal or need to know about the outcome
Preconditions:	What is assumed about the state of the system before the interaction starts
Postconditions:	What are the results after the goal is achieved or abandoned; i.e., what must be true about the system at the time the execution of this use case is completed

Flow of Events for Main Success Scenario:

→	1.	The initiating actor delivers an action or stimulus to the system (the arrow indicates the direction of interaction, to- or from the system)
←	2.	The system's reaction or response to the stimulus; the system can also send a message to a participating actor, if any
→	3.	...

Flow of Events for Extensions (Alternate Scenarios):
What could go wrong? List the exceptions to the routine and describe how they are handled

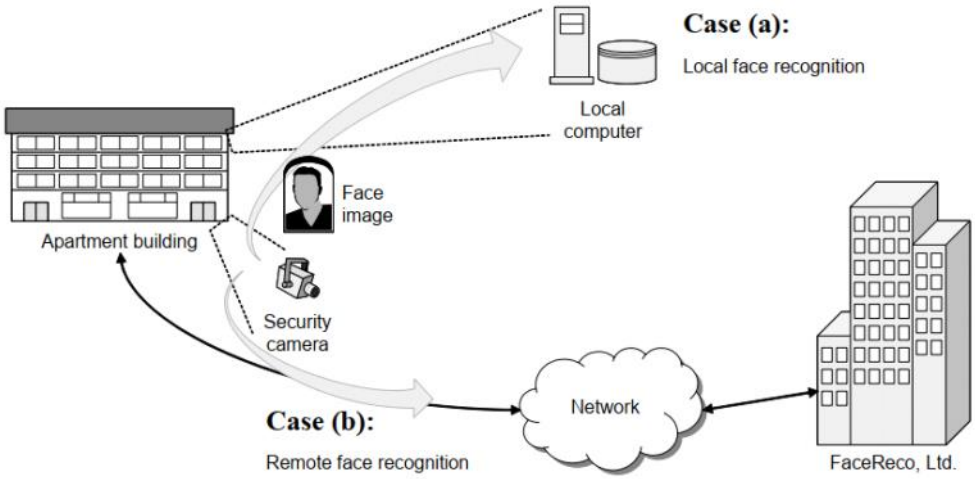
→	1a.	For example, actor enters invalid data
←	2a.	For example, power outage, network failure, or requested data unavailable
	...	

The arrows on the left indicate the direction of interaction: → Actor's action; ← System's reaction

Usage scenarios outline the sequence of events to achieve requirement and

System Boundary & Subsystems

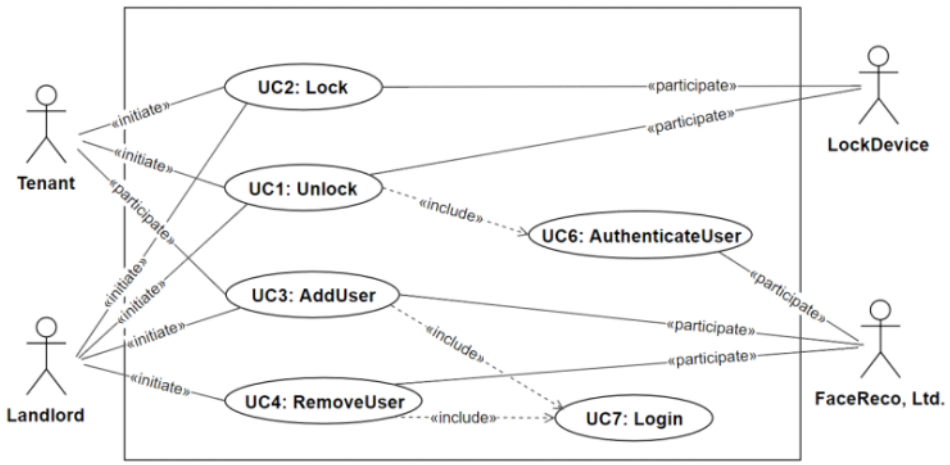
Use Case Variations Example:



Modified Use Case Diagram

Authentication subsystem (FaceReco, Ltd.)
is externalised from the system-to-be:

UC1: Unlock
UC2: Lock
UC3: AddUser
UC4: RemoveUser
UC5: InspectAccessHistory
UC6: AuthenticateUser
UC7: Login



As a customer, I should be able to register for training courses.
As a

The Art of Writing User Stories

Friday, 8 March 2019 4:43 PM

A basic user story has a simple template - a high level description of a desired functionality and goal.

Attributes of a good user story

The **WHO**, **WHAT** and **WHY** of user-story must be at the right granularity and add value to the business and user. **INVEST** is an acronym that helps evaluate whether you have a high quality user story.

INVEST

Independent - Ideally, a user-story must be self-contained, independent (there should be no inherent dependency on other Product Backlog Items (PBI)). If all stories are independent, then any PBI can be developed independently and delivered separately, but this is not always feasible!
So, identifying the right granularity, prioritizing PBI and minimising dependencies results in a better backlog.

Negotiable - Leave room for negotiation, and avoid having too much detail until you have more context. Make sure the user stories are not fleshed out completely, but that you have enough information to allow prioritisation. Higher priority user stories should be more precisely defined, while lower priorities can be less detailed. The user story should also be discussable for later and flexible in case of any changes.

Valuable - The user story must have value to the customer and the business.

Estimable - User stories need be clear enough to estimate, without being too detailed. If a user story is not at the right granularity, or is too vague, it becomes difficult how long the task will take to complete.

Small - Stories should be able to be built in a short/small amount of time (a few person-days).

Testable - The user story should be defined with clear acceptance criteria, both for the correct functionality and error conditions which leads to test cases. Acceptance criteria are tests that define "conditions of satisfaction" from the perspective of the customer. They are used to determine what is required for the business and product owner to accept the user story as "complete".

User-Stories vs Use-Cases

One obvious difference between user stories and use-cases are their scope. User stories describe the goal and its benefit, while use-cases describe the flow of events for a goal to be a success.

User stories and use-cases serve different purposes. A use-case serves as a document contract, while user stories are placeholders for conversation.

User stories and use cases also differ in their longevity. A use-case tends to be permanent, while a user story is disposable.

Domain Modelling using OO Design Techniques

Friday, 8 March 2019 4:43 PM

A ***domain model*** is also referred to as a ***conceptual model*** or a ***domain object model***. It provides a visual representation of the problem domain by decomposing the domain into key concepts or objects in the real world and identifying the relationship between these objects.

Domain Modelling vs Requirements Analysis

Requirements analysis determines the *external* behaviour of the system. What are the features of the system-to-be and who requires these features (actors)

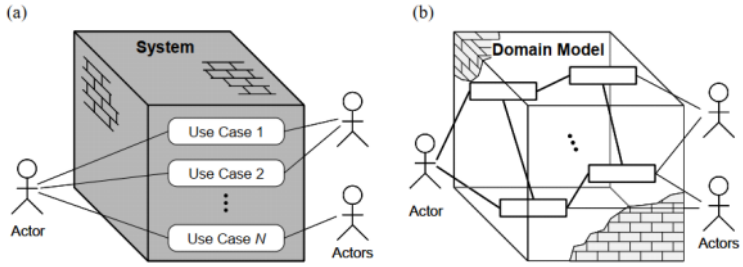
Domain modelling determines the *internal* behaviour of the system. How elements of the system-to-be interact to produce the external behaviour.

Requirements analysis and domain modelling are mutually dependent. The domain model provides clarification for the requirements, while the requirements help build the domain model.

Use-Case vs Domain Model

In **use case analysis**, we consider the system as a "**black box**"

In **domain analysis**, we consider the system as a "**transparent box**"



Benefits of a Domain Model

- Triggers high level ***discussion about what is central to the problem*** (the core domain) and relationships between sub-parts (sub-domains)
- Ensure that the system-to-be reflects a deep, ***shared understanding of the problem domain***, since objects in the domain model will represent domain concepts
- Unambiguous shared understanding of the problem domain and requirements among business visionaries, domain experts and developers

Unified Modelling Language (UML)

UML is an open source, graphical language to model software solutions, application structures, system behaviour and business procedures. It has several uses:

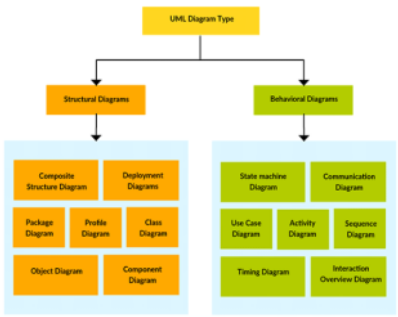
- As a design that communicates aspects of the system
- As a software blueprint
- Sometimes used for auto code-generation

There are two types of UML diagram categories; ***structure diagram*** and ***behaviour diagram***.

Structure diagrams show the ***static structure*** of the system, its parts and how these parts relate to each other. They are said to be static because the elements are depicted irrespective of time.

Behaviour diagrams show the ***dynamic behaviour*** of the objects in the system (i.e. a series of changes to the system over a period of time). A subset of these diagrams are referred to as ***interaction diagrams***, which emphasise the interaction between objects.

UML Diagram Types



Domain Modelling using OO Design Paradigm

Object Oriented Design

Objects are real-world entities and could be something:

- Tangible and visible (car, phone, apple, pet)
- Intangible (an account, time)

Every object has:

- ***Attributes*** - properties of the object. e.g. model number, colour, registration of a car, age, breed of a dog
- ***Behaviour*** -what the object can do (or ***methods***). e.g. a duck can fly, a dog can bark, you can withdraw or deposit into an account

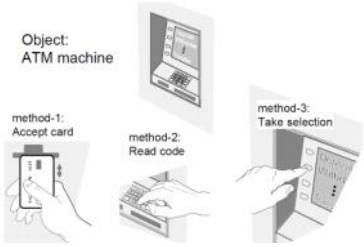
Each object encapsulates some ***state*** (the currently assigned values for its attributes). This gives the object its identity since the state of one object is independent of another.

Objects interact and communicate by sending ***messages*** to each other. Objects play a ***client*** and ***server*** role and could be located in the same memory space or on different servers. If object A wants to invoke a specific behaviour of object B, it sends a message to object B requesting that behaviour.

A message is typically made of three parts:

1. The ***object*** to whom the message is addressed (e.g. John's "account" object)
2. The ***method*** you want the object to invoke (e.g. deposit())
3. Any ***additional information*** needed (e.g. amount to be deposited)

An object's interface is the set of object methods that can be invoked on the object. The interface is the fundamental means of communication between objects.



Objects and Classes

OO Approach vs functional approach

- Functional: thinking about the operation, not the data
- Revolves around the operation you want to achieve.

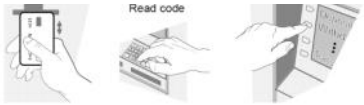
OO thinks about the data (objects) and how they interact with each other

Object instance has state, while a class does not.(it is pretty much a template)

Contents of the attributes define the object state

You do not need to model actors as classes. The only time you [illegible] is if you need to store information about them.

An object's interface is the set of object methods that can be invoked on the object. The interface is the fundamental means of communication between objects.



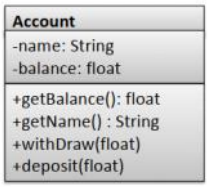
Objects and Classes

Many objects are of the same *kind* but have different identities. E.g. there are many account objects belonging to different customers, but they all share the same attributes and methods. You can logically group objects that share some common properties into a ***class***. A class serves as a *blue-print* defining the attributes and methods (behaviour) of this logical group of objects. A class can sometimes be referred to as an object's type.

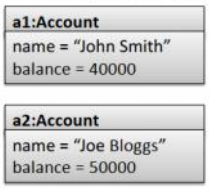
Defining a class does not actually create an object. When an object is ***instantiated*** from a class, it is said to be an ***instance*** of the class. An object instance is a specific realisation of a class. An object has state, but a class does not.
Two object instances from the same class share the same attributes and methods, but have their own ***object identity*** and are independent from each other.

Representing classes in UML

class (class diagram)



object instances (object diagram)



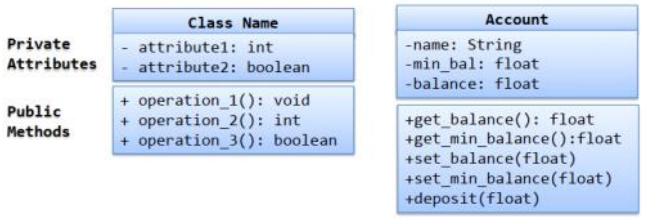
(Two Key) Principles of OO Design

1. Abstraction

Abstraction helps you to focus on common properties and behaviours of objects. Good abstraction helps us to accurately represent the knowledge we gathered about the problem domain (and discard anything unimportant and irrelevant).
Example: For a car, we can create a class for each brand, or we can abstract and focus on the common essential qualities of the object and focus on the current application context.

2. Encapsulation

Encapsulation implies ***hiding*** the objects state (attributes). An object's attributes represent its individual characteristics or properties, so access to the object's data must be restricted. Methods provide explicit access to the object. E.g. the use of `getter` and `setter` methods to access or modify the fields.



UML notation for a Class

The importance of encapsulation:

- Encapsulation ensures that an object's state is in a ***consistent state***.
- Encapsulation increases ***usability***.
  Keeping data private and exposing the object only through its interface (public methods) provides a clear view of the *role of the object* and increases usability.
  It is clear contract between the invoker and the provider guaranteeing consistent behaviour of the method invoked (if the client invokes the method correctly)
- Encapsulation ***abstracts the implementation***, reducing the dependences so that a change to a class does not cause a rippling effect on the system
- Makes the components that you build more ***reusable***

Relationships Between Objects

Relationships between objects can be broadly classified as; ***inheritance*** and ***association***.

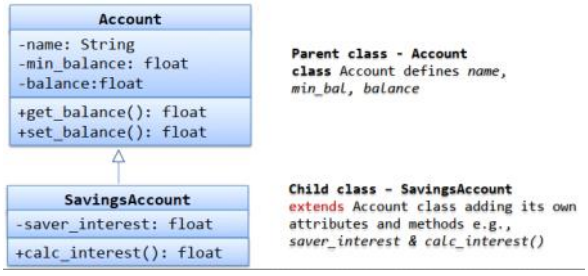
Inheritance

Inheritance models a relationship between classes, in which one class represents a more *general* concept (***parent*** or ***base class***) and another more specialised class (***sub-class***). Inheritance models a *"is-a"* type of relationship. e.g.

- A dog *is a* type of pet
- A savings account *is a* type of bank account
- A manager *is a* type of employee
- A rectangle *is a* type of shape

To implement inheritance we, create a new class (sub-class), that inherits common properties an behaviour from a base class (parent-class or super-class). We say that the child ***inherits***/***is-derived from*** the parent class.
A sub-class can ***extend*** the parent class by defining additional properties and behaviour specific to the inherited group of objects. A sub-class can also ***override*** methods in the parent class with their own specialised behaviour.

To denote that a sub-class inherits the parent class we use an arrow from the sub-class pointing to the parent class it is derived from

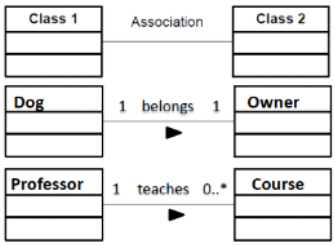


Associations

Association is a special type of relationship between two classes, that shows that the two classes are

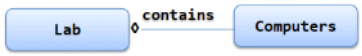
- Linked to each other
- Combined into some kind of *"has a"* relationship, where one class *contains* another class

They are modelled in UML as a line between the two classes.

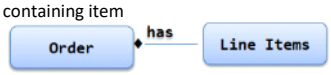


Associations can be further refined as:

- *Aggregation* relationship (hollow diamond symbol ◊ in UML diagrams): the contained item is an element of the a collection but ***can exist on its own***.



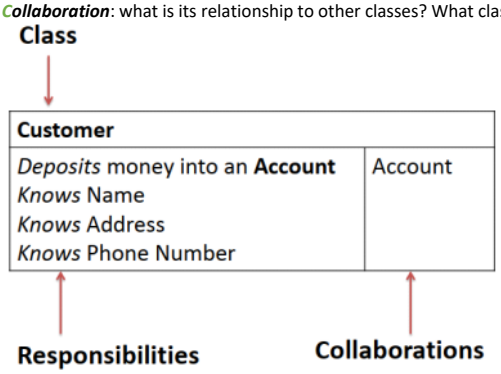
- *Composition* relationship (filled diamond symbol ♦ in UML diagrams): the item contained is an ***integral part*** of the containing item



CRC Cards and Models

CRC cards are a simple tool for system analysis and design. It is part of the OO design paradigm and features prominently as a design technique in XP programming. It is written in 4 by 6 index cards, where an individual CRC card is used to represent a domain object. The output of this tool is the definition of classes and their relationship.

CRC stands for
***C*lass**: an object-oriented class name. Represents a collection of similar objects
***R*esponsibility**: what does this class know and what does it do?
***C*ollaboration**: what is its relationship to other classes? What classes does this class use?



A *CRC Model* is a collection of CRC cards, that specifies OO design of the system.

How to create a CRC model?

You are given a description of the requirements of a software system. Gather around a table with a set of index cards and pen. As a team:

- Read the description
- Identify core classes (look for nouns)
- Create a card per class (start with just class names)
- Add responsibilities (look for verbs)
- Add collaborations
- Add more classes as you discover them
- Refine by identifying inheritance etc

How can you tell it works?

By a ***scenario walk-through***

- Select a set of scenarios (use cases).
- Choose a plausible set of inputs for each scenario.
- Manually "execute" each scenario.
- Start with initial input for scenario and find a class that has responsibility for responding to that input.
- Trace through the collaborations of each class that participates in satisfying that responsibility.
- Make adjustments and refinements as necessary.

Evolve CRC domain models into UML class diagrams, where:

- The concepts are represented as classes
- Collaborations between the classes are established as relationships
- Depending on the kind of relationship, we can use the different notations that we've used for associations - non-hierarchical, aggregation, inheritance.

OO Programming with Python

Saturday, 9 March 2019 12:26 AM

Defining a class in Python

Basic Python Syntax

- To create a class, use the keyword **class** followed by the name of the class e.g., **Account**

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

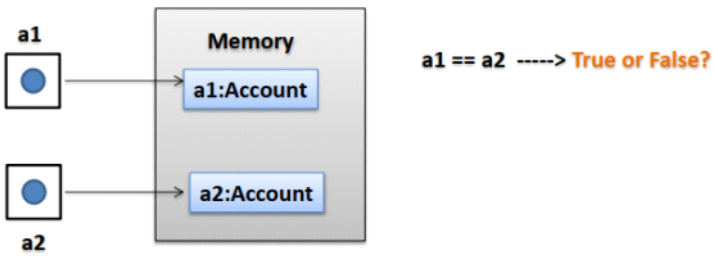
```
class Account:
      'Common base class for all bank accounts'
```

- An ***object instance*** is a specific realization of the class
  - Defining a class, does not actually create an object
  - Create an instance of the Account class as follows:

Creating object instances

- A class is sometimes referred to as an object's type
- An object instance is a specific realization of the class
- create an instance of the Account class as follows

```
class Account:
      'Common base class for all bank accounts'

# a1 and a2 are object instances
a1 = Account()
a2 = Account()
```

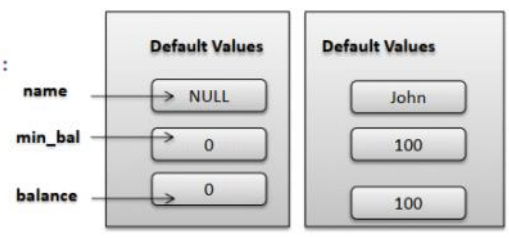


a1 == a2 -----> True or False?

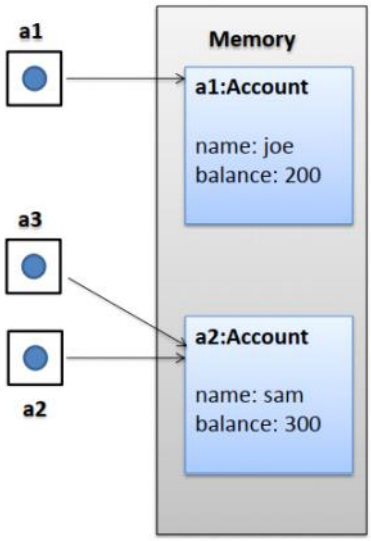
Constructor & Instance Variables

- A special method that creates an object instance and assign values (initialisation) to the attributes (instance variables)
- Constructors eliminate default values
- When you create a class without a constructor, Python automatically creates a default "no-arg" constructor for you

```
class Account:

   def __init__(self,name,min_bal):
       self.name = name
       self.min_bal = min_bal
       self.balance = min_bal

# a1 is an object instance
a1 = Account("John",100)
```



Object References



a1 == a2 -----> True or False?

Consider,
a3 = a1
a3 == a1 -----> True or False?

Instance Methods

Similar to instance variables, methods defined inside a class are known as **instance methods**

Methods define what an object can do.

In Python, every instance method, must specify **`self`** (the specific object instance) as an argument to the method including the constructor (`__init__()`)

```
class Account:

   def __init__(self,name,min_bal):
       self.name = name
       self.min_bal = min_bal
       self.balance = min_bal

   def deposit(self, amount):
       self.balance += amount;

# a1 is an object instance
a1 = Account("John",100)
a1.deposit(120)
```

46

Encapsulation in Python

Python does not support strong encapsulation. Attribute names are simply prefixed with a single underscore e.g., _name to signal that these attributes are **private** and must not be directly accessed by clients

Double underscore not single underscore __name

```
class Account:

    def __init__(self,name,min_bal):
        self._name = name
        self._min_bal = min_bal
        self._balance = min_bal

    def get_name(self):
        return self._name

    def get_min_bal(self):
        return self._min_bal

    def set_min_bal(self,min_bal):
        self._min_bal = min_bal

    def get_balance(self):
        return self._balance

    def deposit(self, amount):
        self.balance += amount;
```

47

Recall why we need encapsulation important

1. Encapsulation ensures that an object's state is in a consistent state

```
class Account:

    def __init__(self,name,min_bal):
        self._name = name
        self._min_bal = min_bal
        self._balance = min_bal

    # define the getter and setter methods
    # ...

    def deposit(self, amount):
        self.balance += amount;

    def withdraw(self, amount):
        if self._balance - amount <= self._min_bal:
                print("Minimum balance must be maintained")
        else:
            self.balance -= amount

a1 = Account("John",100)
a1.withdraw(50)

a1._balance = 10
print("Current balance: {0}".format(str(a1._balance)))
```

breaking encapsulation and direct assignment of the *balance* attribute, potentially set the *balance* to an amount less than minimum balance, violating the business constraint

encapsulation enforces that *balance* is hidden and can only be changed through *deposit* and *withdraw* methods

48

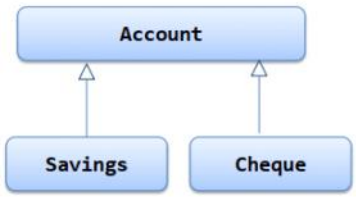
Implementing Inheritance in Python

```
class Account(object):

        def __init__(self, name=None, min_bal=0):
             self._name = name
             self._balance = min_bal

        def get_name(self):
                return self._name
        def get_balance(self):
                return self._balance
        def set_balance(self,amount):
                self._balance = amount


class SavingsAccount(Account):
        def __init__(self,name,amount):
                Account.__init__(self,name,amount)
                self._saver_interest = 0.05
        def get_interest(self):
                return self._saver_interest

a2 = SavingsAccount("joe",1000)
print("{0}'s balance is {1} building interest at {2}:"
      .format(a2.get_name(),a2.get_balance(),a2.get_interest()))
```

Abstract Classes in Inheritance

In the example below

- `Savings` and `Cheque` both inherit from the base class `Account`
  - But `Account` is really not a real-world object
  - `Account` is a *concept* that represents some real-world objects like a Savings Account), so `Account` is said to be an abstract class
  - It does not make sense to create an instance of an abstract class



Abstract class – Vehicle
Defines common attributes e.g., *make, model, year, miles, wheels*

Software Testing

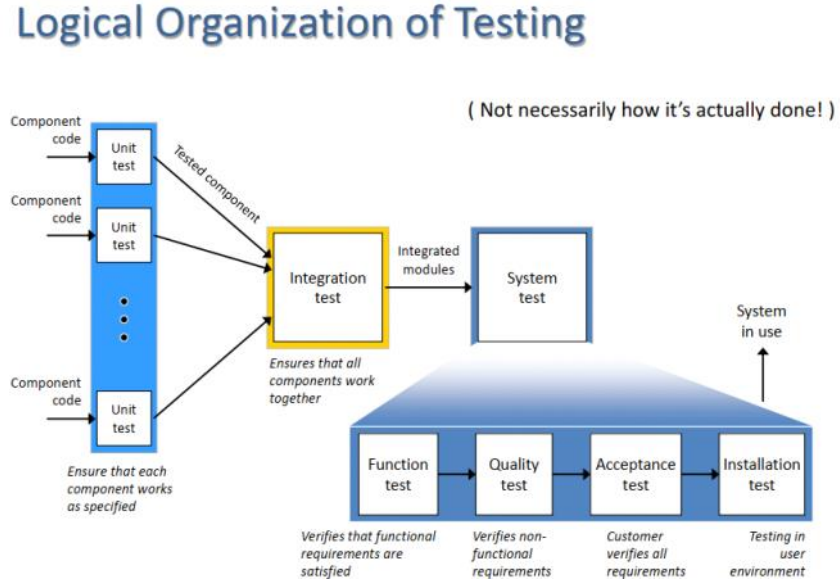
Tuesday, 19 March 2019 4:07 PM

A ***fault***, also called a ***defect*** or a ***bug*** is an erroneous hardware or software element of a system that can cause the system to fail; that is to behave in a way that is not desired or even harmful.

Any software artifact can be tested including requirements specification, domain modelling, and design specification. Testing activities should be started as early as possible. An extreme form of this approach is ***test-driven development (TDD)***, one of the practices of Extreme Programming (XP), in which development *starts* with writing tests.

Testing works by probing the program with different combinations of inputs to detect faults. Therefore, testing shows only the presence of faults, not their absence. Showing absence of faults requires exhaustively trying all possible combination of inputs (or following all possible paths through the program).

A key tradeoff of testing is between testing as many as cases as possible while keeping the economic costs limited. Our goal is to find faults as cheaply and quickly as possible. Ideally, we would design a single *right* test case to expose each fault and run it. In practice, we have to run many *unsuccessful* test cases that do no expose any faults.



Testing is usually guided by the hierarchical structure of the system as designed in analysis and design phases. We may start by testing individual components, which is known as ***unit testing***. These components are incrementally integrated into a system. Testing the composition of the system components is known as ***integration testing***. ***System testing*** ensures that the whole system compiles with the functional and non-functional requirements. The customer performs ***acceptance testing*** of the whole system. As always, logical organisation does not imply testing steps should be ordered in time, instead the development lifecycle evolves incrementally and iteratively, and corresponding cycles will occur in testing as well.

Black box testing is a testing approach commonly adopted by customers (such as user acceptance tests). It tests a running program with a set of inputs without looking at the implementation.

White box testing chooses test data to tests the program with knowledge of the implementation such as knowledge of the system architecture, used algorithms, or program code. This testing approach assumes that the code implements **all** parts of the specification, even with programming errors.

Regression testing seeks to expose new errors, or "*regressions*", in existing functionality after changes have been made to the system. A new test is added for each fault discovered, and tests are run after every change to the code. Regression testing is useful for verifying that software, that was previously developed and tested, still performs after the program or interface has changed

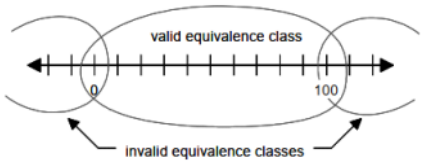
Test Coverage

Test coverage measures the degree to which the specification or code of a software program has been exercised by tests. ***Code coverage*** measure the degree to which the source code of a program has been tested.

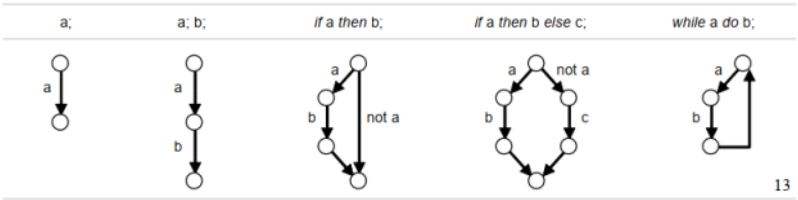
Code coverage criteria include:

- ***Equivalence testing*** - is a black box testing method that divides the space of all possible inputs into equivalence groups such that the program *behaves the same* on each group. There are two steps to equivalence testing:
  1. Partitioning the values of input parameters into equivalence groups
  2. Choosing the test input values

Equivalence classes:



- ***Boundary testing*** - is a special case of equivalence testing that focuses on the boundary values of input parameters. It is based on the assumption that developers often overlook special cases at the boundary of equivalence classes. To do this we select elements from the *edges* of equivalence classes, or *outliers* such as:
  - Zero
  - Min/max values
  - Empty sets
  - Empty strings
  - Null
  - Confusions b/w > and >=
- ***Control-flow testing***
  Statement coverage selects a tests set such that every elementary statement in the program is executed at least once by some test case in the test set.
  Edge coverage selects a test set such that every edge (branch) of the control flow is traversed at least once by some test case. We construct the ***control graph*** of a program so that statements become the graph edges, and the nodes connected by an edge represents entry and exit to/from the statement.
  Condition coverage (predicate coverage) selects a test set such that every condition (boolean statement) takes TRUE and FALSE outcomes at least once in some test case
  Path coverage determines the number of distinct paths through the program that must be traversed at least once to verify correctness.



- ***State-based testing*** - defines a set of abstract states that a software unit can take and tests the unit's behaviour by comparing its actual states to the expected states.

Unit Testing Frameworks

PyUnit

Python has a version of JUnit, which is used for Java testing. It uses the module `unittest` to support test automation.

Important concepts of `unittest`:

- ***Text fixture***: preparation tasks/clean up actions e.g. create temporary databases
- ***Test case***: the *smallest unit of testing*, that checks for a specific response to a particular set of inputs. This uses a base class `TestCase`, to create new test cases
- ***Test suite***: a collection of test cases, test suites or both, which are used to aggregate tests that should be executed together
- ***Test runner***: orchestrates the execution of tests and provides the outcome to the user

Important points for writing a single test

- Every class is a sub-class of `unittest.TestCase`
- Every test function should start with `test` name
- Use `assert` functions to check for an expected result
- Define initialisation tasks by overriding `setup()` method, which is called before a test method is run
- Define clean-up tasks by overriding `teardown()` method, which is called after a test method is run
- Run Test with `python -m unittest -v test_module`

```
#arithmetic functions

def multiply(a,b):
    return a*b

def add(a,b):
    return a+b

def divide(a,b):
    return a/b
```

```
import unittest
from arith import multiply,add

class AddTestCase(unittest.TestCase):

    def test_add_with_correct_values(self):
        self.assertEqual(add(2,3),5)

class MultiplyTestCase(unittest.TestCase):
    def test_multiply_with_correct_values(self):
        self.assertEqual(multiply(3,6),18)

if __name__ == '__main__':
        unittest.main()
```

PyTest

Python also has a third-party testing framework; pytest
Important concepts (similar to `unittest`):

- ***Text fixture***: preparation tasks/clean up actions e.g. create temporary databases, directories. Use `@pytest.fixture` wrapper around a function
- ***Test case***: the *smallest unit of testing*, that checks for a specific response to a particular set of inputs.

Fixtures are a way of setting up your test cases.
Pytest knows that it needs to run

- ***Text fixture***: preparation tasks/clean up actions e.g. create temporary databases, directories. Use `@pytest.fixture` wrapper around a function
- ***Test case***: the *smallest unit of testing*, that checks for a specific response to a particular set of inputs.
- ***Test suite***: a collection of test cases, test suites or both, which are used to aggregate tests that should be executed together
- ***Test runner***: orchestrates the execution of tests and provides the outcome to the user. Use the `pytest` command

Fixtures are a way of setting up your test cases.
Pytest knows that it needs to run @pytest.fixture before it runs the test cases

Important points for writing a single test:

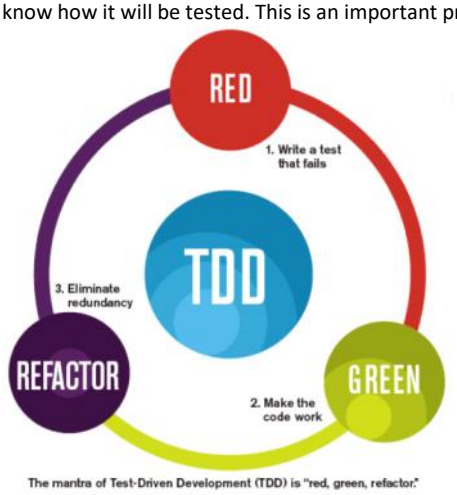
- Need to install Pytest: `pip install pytest`
- Every test function should start with `test` name
- Use `assert` functions to check for an expected result
- Set up fixtures to reuse variables between tests
- Run Test with `pytest -v test_module`

```
# Arithmetic functions

def multiply(a, b):
    return a*b

def add(a,b):
    return a+b

def divide(a,b):
    return a/b
```

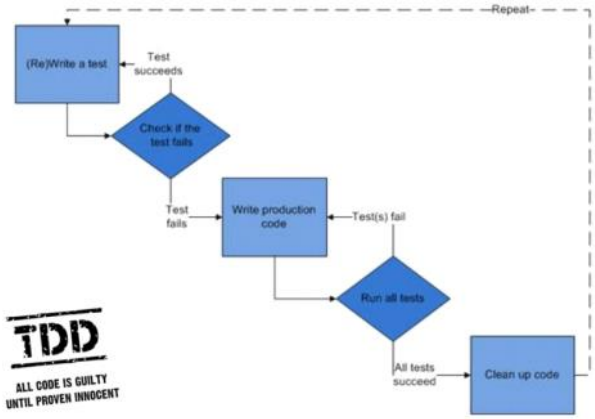
```
import pytest
from arith import multiply,add

def test_add_with_correct_values():
    assert(add(2,3) == 5)

def test_multiply_with_correct_values():
    assert(multiply(3,6) == 18)
```

Test Driven Development

In test driven development, every step in the development process must start with a plan of how to verify that the results meet a goal. The developer should not create a software artifact (a UML diagram, or source code) unless they know how it will be tested. This is an important principle in XP, Scrum



The mantra of Test-Driven Development (TDD) is "red, green, refactor."



In general:

1. Write a test case that fails
2. Write (just enough) code to implement the function
3. Test the code
4. Refactor code to remove redundancies
5. Repeat

Refactoring is a technique of modifying the internal structure of software to make it more maintainable, reusable and flexible **without changing** the external behaviour of software.

There are many cases to consider and unit tests help to validate the complex logic and check for regression errors. This also brings emphasis that every unit of code is tested.

Exception Handling

An ***exception*** is an ***error*** that happens during the execution of a program, causing the program to terminate abruptly. This can be caused by providing the wrong input data, running out of memory, file or network resources. They are different to ***program bugs***; an error in the code.

Exception handling enables us to handle these situations gracefully and avoid intermittent failures. It is critcal for creating robust and stable applications

In Python, when an error occurs:

- An exception is raised through creating a Python object `Exception`
- The normal flow of the program is disrupted
- This exception must be handled, or else the program terminates
- We can use python's `try/except` clause to handle exceptions

Syntax of a Python <try-except-else> block:

```
try:
      You do your operations here;
      ......................
except ExceptionI:
      If there is ExceptionI, then execute this block.
except ExceptionII:
      If there is ExceptionII, then execute this block.
      ......................
else:
      If there is no exception then execute this block.
finally:
      Always, execute this block.
```

If you have a `return` in the `try` segment, your value will only be returned after `finally` is executed

Common exceptions in Python

Exception	Occurence
IOError	If the file cannot be opened
ImportError	If python cannot find the module
ValueError	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value
EOFError	Raised when the end of file is reached.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.

ModuleNotFoundError	

A `try` statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
try:

except (RuntimeError, TypeError, NameError):
```

A class in an except clause is compatible with an exception if it is the same class or a base class. For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass
for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

The raise statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

The sole argument to raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

Assert in Python

Assert is a powerful debugging aid to test conditions. We use assertions as an *internal self-check* to identify unrecoverable errors potentially caused by a program bug. Assertions are usually used to check preconditions. It is **NOT** a mechanism for handling run-time errors such as "`File Not Found`". An `AssertError` is raised if the assert condition fails.
Python's assert syntax: `assert expression1 ["," expression2]`

Do **NOT** use asserts for data validation or data processing. Asserts can be turned off globally, which can cause dangerous side effects. For example:

```
def delete_product(product_id, user):
    assert user.is_admin(), 'Must have admin privileges to delete'
    assert store.product_exists(product_id), 'Unknown product id'
    store.find_product(product_id).delete()
```

Instead use validation exceptions:

```
if not user.is_admin():
    raise AuthError('Must have admin privileges to delete')
```

Use assertions during development to ensure pre and post conditions are satisfied. When the code goes in to production, it is safe to remove the assertions.

There is a base exception class in python. You can design a custom exception that extends the exception
Class ExceptionYouExtend:
  pass

Raise nameOfException(additional parameters)

Client calling a function that can raise an exception must handle it; this can be done using the try-except block e.g lecture
Arguments in the exception are stored as tuples

It is necessary to write custom exceptions to suit the error; invalid authentication, ordered too much

If exceptions are raise, you usually want to store them in a log file.

Agile Development Methodologies

Tuesday, 26 March 2019 4:08 PM

Software engineering is a complex, organised process with great emphasis on ***methodology***.
A ***software development methodology*** lays out a prescriptive process by mandating a sequence of tasks such as:

- Analysis and specification
- Design
- Implementation
- Testing
- Release and maintenance

Elaborate processes are rigid, plan-driven documentation heavy methodologies. e.g. the Waterfall approach. It is unidirectional, where you must finish one step before moving to the next.
Iterative and Incremental processes are processes, which develop increments of functionality and repeat in a feedback loop. e.g. Rational Unified Process (RUP) and agile methods such as SCRUM and XP (which are more aggressive in terms of short iterations)

Incremental and Iterative Project Life-Cycle

Incremental and iterative approaches:

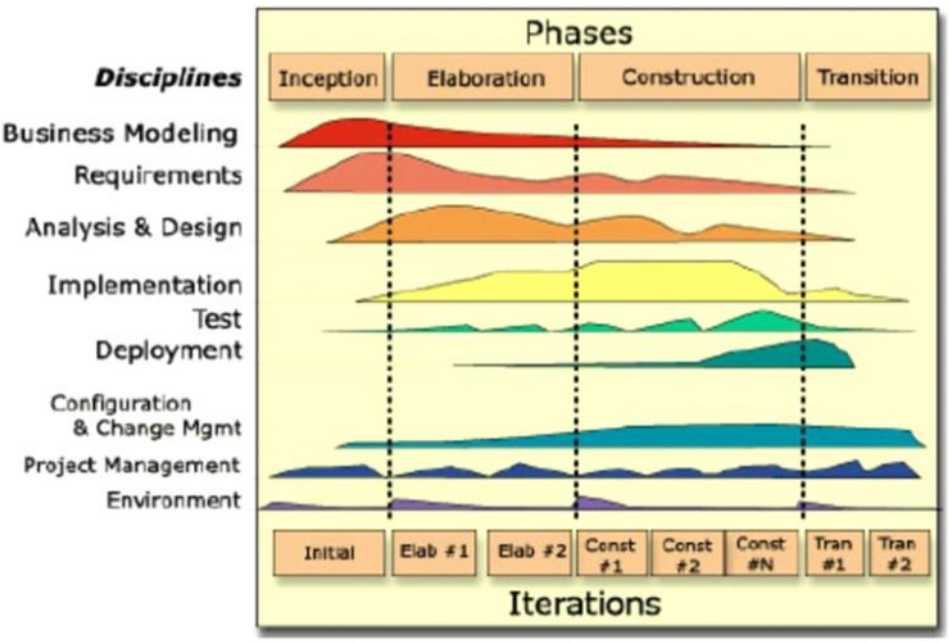
1. Break the big problem down into smaller pieces and prioritise them
2. An ***iteration*** refers to a step in the life-cycle
3. Each *iteration* results in an ***increment*** or progress through the overall project
4. Seek the customer's feedback and changes course based on improved understanding at the end of each iteration

An incremental and iterative process seeks to get a working instance as soon as possible. It progressively deepens the understanding or *visualisation* of the target product.

Rational Unified Process (RUP)

RUP is an iterative software development process that has four major phases:

1. ***Inception***: scope the project, identify major players, what resources are required, architecture and risks, estimate costs
2. ***Elaboration***: understand the problem domain, analysis, evaluate in detail the required architecture and resources
3. ***Construction***: design, build and test the software
4. ***Transition***: release the software into production



Agile Software Development Methodologies

Some agile methodologies include Scrum, DSDM, FDD, Lean, Crystal and eXtreme Programming.

Benefits of agile methods:

- 93% increased productivity
- 88% increased quality
- 83% improved stakeholder satisfaction
- 49% reduced costs

- 66% three-year, risk-adjusted return on investment

Through agile methods, we have come to value:

- ***Individuals and interactions*** over processes and tools
- ***Working software*** over comprehensive documentation
- ***Customer collaboration*** over contract negotiation
- ***Responding to change*** over following a plan

Extreme Programming (XP)

XP is a prominent agile software engineering methodology that

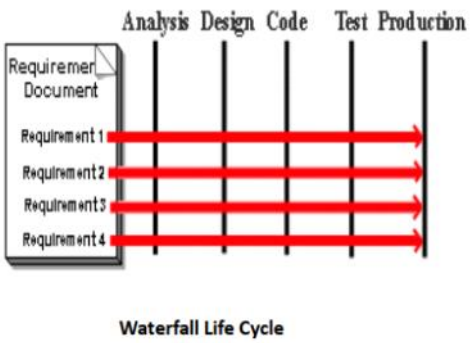
- focuses on providing the highest value for the customer in the fastest possible way
- acknowledges changes to requirements as a natural and inescapable aspect of software development
- places higher value on ***adaptability*** (to changing requirements) over ***predictability*** (defining all requirements at the beginning of the project) – being able to adapt is a more realistic and cost-effective approach
- aims to lower the cost of change by introducing a set of basic ***principles*** (high quality, simple design and continuous feedback) and ***practices*** to bring more flexibility to changes

Core principles of XP:

1. ***High quality***
   Pair programming - code is written by pairs of programmers, working at the same workstation. One member *codes*, while the other *reviews*
   Continuous integration - programmers check their code in and integrate it several times a day
   Sustainable pace - a moderate and steady pace
   Open workspace - as the name suggests
   Refactoring - a series of tiny transformation to improve the structure of the system
   Test-driven development - unit-testing and user acceptance testing
2. ***Simple design***
   Focus on the stories in the current iteration and keeps the designs
   simple and expressive. Migrate the design of the project from iteration to iteration to be the best design for the set of stories currently implemented. Spike solutions, prototypes, CRC cards are popular techniques during design
   Three design mantras for developers:
   - *Consider the simplest possible design* for the current batch of user stories (e.g., if the current iteration can work with flat file, then don't use a database)
   - *You aren't going to need it* – Resist the temptation to add the infrastructure before it is needed (e.g., "Yeah, we know we're going to need that database one day, so we need put the hook in for it?)
   - *Once and only once* – XP developers don't tolerate duplication of code, wherever they find it, they eliminate it
3. ***Continuous feedback***
   An XP team receives intense feedback in many ways, in many levels (developers, team and customer)
   - Developers receive constant feedback by working in pairs, constant testing and continuous integration
   - XP team receives daily feedback on progress and obstacles through daily stand-up meetings
   - Customers get feedback on progress with user acceptance scores and demonstrations at the end of each iteration
   - XP developers deliver value to the customer through producing working software progressively at a "steady heartbeat" and receive customer feedback and changes that are "gladly" accepted.

XP vs Waterfall Life-Cycle

Traditional software development is linear, with each stage of the lifecycle requiring completion of the previous stage.
XP turns the traditional development process sideways, where we visualise activities and keep the process itself constant.



Waterfall Life Cycle

XP Life Cycle

Key Steps in XP Planning:

1. ***Initial exploration***: the developer and customer have *conversations* about the system-to-be and identify significant features (but not ALL features, these can be discussed later). Each feature is broken into one or more *user stories*. The developers estimate the user story in *user story points* based on the team's ***velocity***, which becomes more accurate through iterations.

- Stories that are too large or too small are difficult to estimate. A *epic* story should be split into pieces that aren't too big
- Developers complete a certain number of stories each week. The sum of the estimates of the completed stories is a metric known as ***velocity***
- Developers have a more accurate idea of ***average velocity*** after 3 or 4 weeks, and using this, can provide better estimate times for ongoing iterations
  User stories impact the planning process in two key areas; ***estimating*** and ***scheduling***

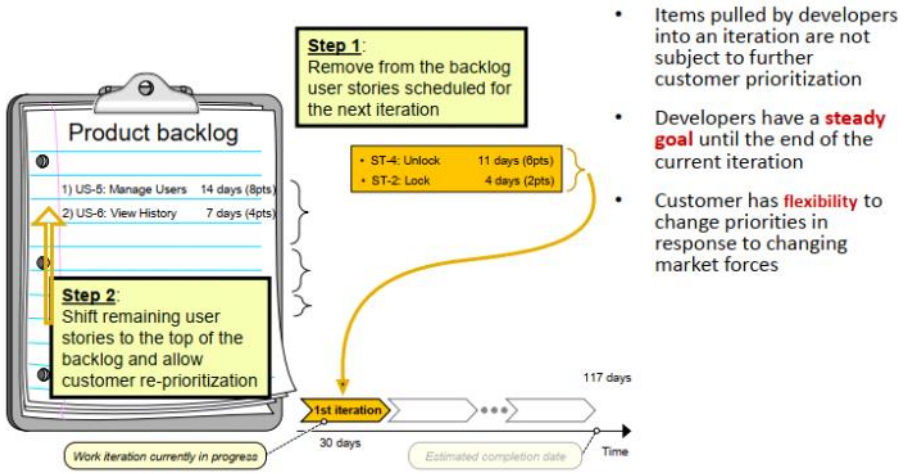
2. ***Release plan***: this is where a release date (6 or 12 or 24 months in the futures) is negotiated. Customers specify which user stories are needed and the order for the planned date. Customer cannot choose more user stories than will fit according to the current velocity plan. Selection is crude, as initial velocity is inaccurate. The release plan can be adjusted as the velocity becomes more accurate.

   The project velocity is used to plan:

- By ***time***, where we compute the number of user stories that can be implemented before a given date (multiply the number of iterations by the project velocity)
- By ***scope***, where how long a set of stories will take to finish divided by the total weeks of estimated user stories by the project velocity.

3. ***Iteration planning***: this uses the release plan to create ***iteration plans***. Developers and customers choose an iteration size: typically 1 or 2 weeks. The customer prioritises user stories from the release plan in the first iteration, but the selection must fit the current velocity. Customers cannot change the stories in the iteration once it has begun, but they can change or reorder any other story in the project). The iteration ends on a specified date, even if all the stories aren't done. A story is not done, until it passes all acceptance test. Estimates for all the completed stories are totaled, and the velocity for that iteration is calculated.
4. ***Task planning***: Developers and customers arrange an **iteration planning meeting** at the beginning of each iteration. Customers choose user stories for the iteration from the release plan but **must** fit the current project velocity. User stories are broken down into programming tasks and order of implementation of user stories within the iteration is determined ***(technical decision).*** Developers may sign up for any kind of tasks and then estimate how long task will take to complete (*developer's budget – from previous iteration experience*). Each task estimated as 1, 2, 3 (or even ½) days of ideal programming days. Tasks < 1 day grouped together, tasks > 3 days broken down. Project velocity is used again to determine if the iteration is over-booked or not. Time estimates in ideal programming days of the tasks are **summed** up, and this must not exceed the project velocity (initial or from the last iteration). *If the iteration has too much - the customer must choose user stories to be put off until a later iteration (snow plowing). If the iteration has too little then another story can be accepted.*
   The velocity in task days (iteration planning) overrides the velocity in story weeks (release planning) as it is more accurate. Team holds a meeting halfway through iteration to track progress

The total project effort is estimated based on the culminative story points of all user-stories



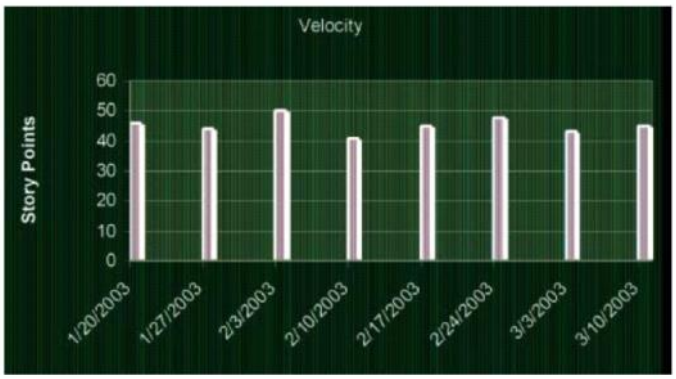
Project Velocity

The total work size estimate is the sum of all user story points for each user story.
The estimate ***project velocity*** (or the team's productivity) is estimated from the number of use story points that the team can complete in a particular iteration. This enables customers to obtain an idea of the cost of each story, its business value and priority.

The estimate project duration is (total work size)/(project velocity)

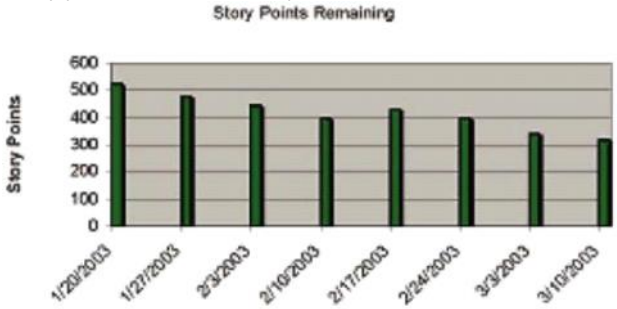
Project tracking is the recording of results in each iteration. The results are used to predict what will happen in the next iteration. Tracking the total amount of work done during each iteration is the key to keep the project running at a *sustainable, steady pace*. XP teams use a ***velocity chart*** or ***burn-down chart*** to track the project velocity which shows how many story points were completed (i.e. user acceptance tests passed).

Velocity Chart



Burn-Down Chart

A burn-down chart shows the week-by-week progress. The slope of the chart is a reasonable predictor of the end-date. The bars in the burn-down chart indicate how many user story points are left to implement.



Drawbacks of Agile Methods

- ***Daily stand up meetings, close collaboration*** – not ideal for development outsourcing, clients and developers separated geographically, or business clients who simply don't have the manpower, resources
- ***Emphasis on modularity, incremental development, and adaptability*** – not suited to clients desiring contracts with firm estimates and schedule
- ***Reliance on small self-organized teams*** makes it difficult to adapt to large software projects with many stakeholders with different needs and neglects to take into account the need for leadership while team members get used to working together.
- ***Lack of comprehensive documentation*** can make it difficult to maintain or add to the software after members of the original team turn over
- ***Agile development*** – Need highly experienced software engineers who know how to both work independently and interface effectively with business users

XP should be used when the problem domain requires change, and when customers do not have a firm idea of what they want. XP practices mitigate risk and increase the likely-hood of success. It is ideal for project group sizes of 2-12. XP requires an extended development team comprising of managed, developers and customers all closely collaborating. It also places great emphasis on *testability* and stresses creating automated unit and acceptance tests. XP projects also deliver greater productivity.

Deciding on which method to use

This depends on what the customer wants. Successful software development strategies understand all three processes in depth and take the parts of each that are most suited to your particular product and environment.
SaaS (Software as a Service) and Web 2.0 applications that require moderate adaptability are likely to be suited to agile style.
Mission-critical applications such as military, medical that require a high degree of predictability are more suited to waterfall

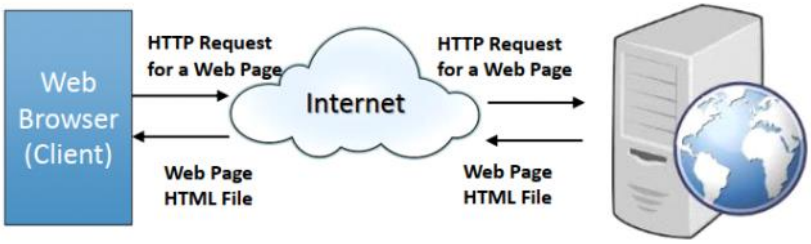
WWW, Decorators & Flask

Friday, 29 March 2019 12:45 AM

The Internet is a global network of computers connected with the purpose of sharing information. Users typically access a network through a computer called a ***host*** or ***node***. A node that provides information or a service is called a ***server***. A computer or other device that requests services is called a ***client*****.**

The **World Wide Web (WWW)**, or simply **Web**, is an information sharing model that is built on top of the Internet, where information is structured as documents called "web pages" written in HTML and not accessed in a linear fashion, but connected using **hypertext links** forming a huge "web" of connected information. WWW is based on the **HTTP** protocol and is just one way of accessing information over the Internet.

Web Architecture:



Each document on the World Wide Web is referred to as a **Web page** and it is basically a text file written in HTML (Hypertext Markup Language) and stored on a Web Server. Each web page has a special address, **URL (Uniform Resource Locator)**

HTML files are viewed using a web browser.

A Web Server is a computer that runs a special server software that enables communication between computers through the HTTP protocols and these servers make available any web page to any device connected to the Internet
e.g., Apache HTTP server, Microsoft IIS server
A Web Browser is a software application running on the client for retrieving and rendering a web page to an end-user
e.g., Internet Explorer, Chrome

How a web page is assembled:

1. A client requests a web page by specifying the URL or clicking on a hyper link
2. Browser sends a HTTP request to the web server named in the URL and requests for the specific document
3. The **web server** locates the requested file and sends a HTTP response.
   a. If document is not found, an error message "404, Not found" is returned
   b. If the document is found, the server returns the requested file to the browser
4. The browser parses the HTML document and assembles the page
   a. If the page contains images, the browser requests the server for the image, inserts the image into the document in the position indicated and displays the assembled page

HTTP Requests and Responses

```
HTTP Request Structure
1 GET /home.html HTTP/1.1
2 Host: www.yoursite.com
```

A HTTP request message consists of:

- Request Line (GET /home.html HTTP/1.1)
- Headers
- An optional message body

Common HTTP request methods include HEAD, GET, POST, DELETE

HTTP defines eight possible request methods:

1. HEAD
2. GET
3. POST
5. DELETE
6. TRACE
7. OPTIONS

```
HTTP Response Structure
1 HTTP/1.1 200 OK
2 Date: Sun, 28 Jul 2013 15:37:37 GMT
3 Server: Apache
4 Last-Modified: Sun, 07 Jul 2013 06:13:43 GMT
5 Transfer-Encoding: chunked
6 Connection: Keep-Alive
7 Content-Type: text/html; charset=UTF-8
8 Webpage Content
```

A HTTP response consists of:

- A Status Line that includes a status code
  - Success: 2xx
  - Redirection: 3xx
  - Client-Error: 4xx
  - Server-Error: 5xx
- Headers
- An optional message body

4. PUT 8. CONNECT

HTTP is a stateless protocol. This means that the server and client are only aware of each other during a current request. Neither client nor server can retain information between different requests across web pages It is possible for a user to customise the content of a website by using:

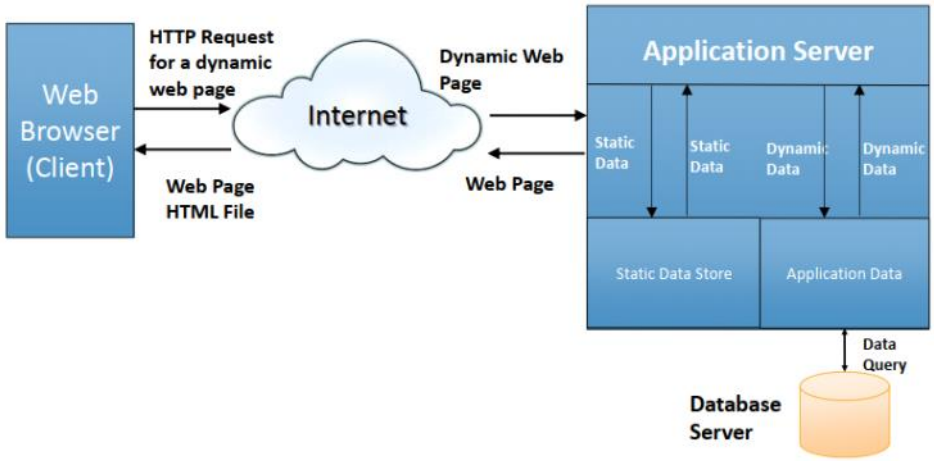
- Cookies (a small piece of text stored on user's computer)
- Sessions
- Hidden variables (when the current page is a form)

Server-Side processing (e.g., PHP, Perl, J2EE, FLASK, Django, Ruby on Rails) is when a server receives dynamic web page request and the server processes user input, renders the dynamic web page to be returned to the client for display on browser.

Client-Side Processing (e.g., JavaScript) is processing needs to be "executed" by the browser to:

- Complete the request for the dynamic page (e.g. valid user-input)
- Create the dynamic web page
- More responsive UI and lowers the bandwidth cost

Extended Web Architecture:



Intro to Flask

Python Decorators

In python, functions are like other data types (e.g. number, string, list) which means we can do a lot of useful operations on them:

1. Assigns function to variables
2. Define functions inside another function
3. Functions can be passed as parameters to other functions
4. Functions can return other functions

Assign Function To Variable

```
#Example1: Assigning a function to a variable

def hello_world(name):
        return "Hello World!" + name

my_function = hello_world
print(my_function("Sam"))
```

Nesting Functions

```
#Example2: Nesting functions inside functions

def nested_hello_world(name):
        def greet():
                return "Hello World! "
        return greet() + " " + name

print(nested_hello_world("Jack"))
```

Functions Can Be Passed As Parameters To Other Functi

```
#Example3: Passing functions as parameters to other functions

def greet(name, lang):
        if lang == "French":
                return "Bonjour " + name
        else:
                return "Hello World!"+ name

def another_hello_world(func):
        my_name = "Jack"
        my_lang = "French"
        return func(my_name,my_lang)

print(another_hello_world(greet))
```

This example returns the same result as invoking the function greet() directly

Functions can return other Functions

```
#Example4: Functions can return other functions
def hello_world():
        def greet(name):
                return "Hello there! " + name
        return greet

my_function=hello_world()
print(greet("Maya"))
```

Applying what we have learnt so far, we can no build a function decorator. ***Function decorators*** are simply wrappers to existing functions. They are useful for extending the behaviour of functions without exactly having to modify them.

A decorator function basically takes a function as an argument and generates a new function that uses the work of the original function as an argument and returns the newly generated function.

```
#Applying the above ideas, we build a function decorator
def say_hello():
        return "Hello World! "

def my_decorator(func):
        def wrapper():
            name = "jack"
            return func() + name
        return wrapper

decorated_func = my_decorator(say_hello)
print(decorated_func())
```

Python's decorator syntax below provides a neater shortcut, by specifying the decorating function before the function can be decorated.

```
#Using Python's neat decorator syntax
def my_decorator(func):
        def wrapper():
            name = "jack"
            return func() + name
        return wrapper

@my_decorator
def say_hello():
        return "Hello World! "

print(say_hello())
```

Instead of hard-coding the variable **name**, it could actually be passed in as an argument to the decorator function.
e.g., the function **my_decorator** can be wrapped inside another function **tag**, which could take in the name argument

```
def tag(name):
        def my_decorator(func):
                def wrapper():
                        name = "jack"
                        return func() + name
                return wrapper
        return my_decorator

@tag("Jack")
def say_hello():
        return "Hello World! "

print(say_hello())
```

Flask

Flask is a micro web application framework written in Python by Armin Ronacher.
A framework is essentially a collection of tools, software and libraries all collaborating together in helpful way for the developer. It allows us to develop applications in a more structured and defined way. Flask is a micro framework because it is not as fully featured; it provides fundamental building blocks of web application and allows more flexibility on the developer's part in terms of implementation details.
As a micro-framework, aims to provide a simple, solid core but designed as *extensible* framework e.g., no native support for databases, authenticating users etc., but these key services available through *extensions* that integrate with the core package. As a developer, you pick the extensions that are relevant to your project or even write your own custom extensions.
Flask relies on two main **dependencies:**

- Werkzeug which supports routing (request and response), utility functions such as debugging and WSGI (Web Server Gateway Interface – a standard interface between web server and web applications
- Jinja2, a powerful templating language to render dynamic web pages

HTML and CSS

Monday, 27 May 2019 3:53 PM

HyperText Markup Language
<h[1-6]> header
<p> paragraph
<table>
<tr> table row
<td> table data
<th> table heading
<!-- comment -->
bold text
italic texts
Unordered lists

Ordered list

A div is a block element. The next tag that follows will be on a new line

Span is an inline element. The next tag that follows will be on the same line

Heading and paragraph tags are block elements
Bold and italics tags are inline elements

<img src="relative url"></img>

Relative absolute referencing

- Relative refrenceing includes information relative to the current folder/directory the file is in. it si often used to refer to a document within the same folder system as the current file. e.g "images/profile.png"
- Abosulte refrences includes infromation from afixed place on the internet. It alawys starts with a protocol (http or https

Hyperlinks

1. Links

link to be clicked
Link opens link on a new tab

1. anchor - linking to another section in the same document

Link to another section will jump to the div with id footer

Cascading Style Sheets
Each html tag has its own unique set of properties such as colour size and width. CSS can alter the value of each property.
Every line of css is going to change the value of a property "property: value; "

There are three ways to apply CSS:

1. Inline - styling each element inline within the tag
2. Internally - put <style></style> in the head tag of the document
   <head>
     <style>
       div {

```
            color: red;
        }
    <style>
</head>
```

3. Externally - create a file exclusively for css and write the styling in there. You can make the html file refer to this style by typing in the document's head tag

```
<link rel="stylesheet" type="text/css" href="style.css">
```

Styling a collection of tags

There are multiple ways to style a collection of tags.

1. Tag/Element
   ```
   h1{
      color: blue;
   }
   ```
2. Class
   ```
   <h1 class="greeting"></h1>
   .greeting {
      color: blue;
   }
   ```
3. Id
   ```
   <div id="hello"></div>
   #hello {
      color: red;
   }
   ```
4. Pseudo classes - gives a certain type of tag, id or class a different appearance when some action is done to it
   ```
   .greeting:hover {
      background-color: yellow;
   }
   ```
5. Pseudo element - alters the behaviour of a specific part of the tag, id or class
   ```
   .greeting::first-letter {
         Background-color: yellow;
   }
   ```

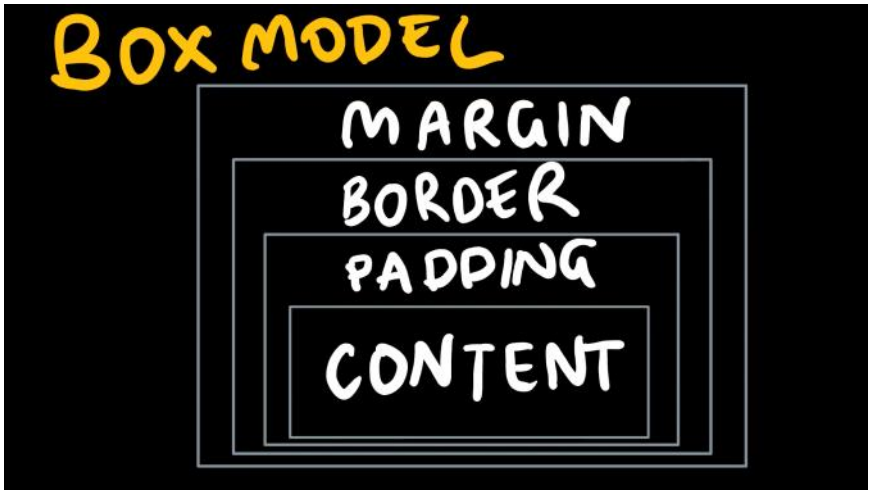
When you specify conflicting CSS, there is an order of precedence.

1. Inline
2. Internal
3. Linked/external
4. Browser

Higher order precedence can override any styling in lower precedence

CSS properties of the container in which the tag comes in

- Width, height
- Margin, padding
- Border



The box model states that each html tag is a piece of content, that has three types of boxes around it

There are pre-made tools such as bootstrap and materialize CSS that can make your webpage look good with minimal effort.
They essentially have CSS file that has already been made for you. All you have to do is include the CSS reference from their website.
Bootstrap: https://getbootstrap.com/
Materialize CSS: https://materializecss.com/

Effective Software Design

Wednesday, 3 April 2019 9:33 PM

Poor design leads to lower quality software systems, less functionality to clients, higher costs (for everyone), loss of business and misery for developers.
Systems Sciences Institute at IBM found that it costs four- to five-times as much to fix a software bug after release, rather than during the design process

We write bad code. Why?

- Because we don't know how to write better code?
- Requirements change in ways that the original design did not anticipate
- Changes require refactoring and refactoring requires time and we might not have time
- Business pressure causes us to make quick and dirty solutions
- Changes may be made by developers not familiar with the original design philosophy

Design Smells

A ***design smell*** is a symptom of poor design. It is often caused by violation of key design principles. Design smells have structures in software that suggest refactoring.

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

Design smell symptoms:

- Rigidity - the tendency of the software being too difficult to change even in simple ways. A single change causes a cascade of changes to other dependent modules
- Fragility - the tendency of the software to break in many places when a single change is made
- Rigidity and fragility complement each other – aim towards minimal impact, when a new feature or change is needed
- Immobility - the design is hard to reuse. The design has parts that could be useful to other systems, but the effort needed and risk in dis-entangling the system is too high
- Viscosity
    - Software viscosity – changes are easier to implement through 'hacks' over 'design preserving methods'
    - Environment viscosity – development environment is slow and in-efficient
- Opacity - the tendency of a module to be difficult to understand. Code must be written in a clear and expressive manner
- Needless complexity - contains constructs that are not currently useful. Developers ahead of requirements
- Needless repetition - design contains repeated structures that could potentially be unified under a single abstraction. Bugs found in repeated units have to be fixed in every repetition

Characteristics of Good Design

The design quality of software is characterised by

1. Coupling
2. Cohesion

Coupling is the degree of interdependence between components or classes. ***High coupling*** occurs when one component A depends on the internal workings of another component B and is affected by internal changes to component B. High coupling leads to a complex system, with difficulties in maintenance and extension; eventually the software will rot. we aim to write ***loosely coupled*** classes. This allows for components to be used and modified independently of each other. At the same time ***zero coupled*** classes are not usable/useless. We need to strike a balance between the two.

Cohesion is the degree to which all elements of a component or a module work together as a functional unit. ***Highly cohesive*** modules are much easier to maintain, less frequently changed and have higher probability of reusability. Do not put all the responsibilities into a single class to avoid low cohesion.
Cohesion means that the class/component should only have one responsibility. This does not mean you can only have one method per class. It means that all the functions in the module must work together towards the same purpose.

SOLID Principles

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

- **S**RP - single responsibility principle
- **O**CP - Open closed principle
- **L**SP - Liskov Substitution Principle
- **I**SP - Interface Segregation Principle
- **D**IP - Dependency Inversion Principle

(Last three are not covered in this course)

Single Responsibility Principle - *a class should have **one** reason to change*

Cohesion asks whether the elements in a modules have a good reason for being there. Cohesion and SRP are the forces that causes modules to change.

Changes in requirements results in changes in responsibilities. A ***cohesive responsibility*** represents a single axis of change. A class should only have one responsibility.

A class with several responsibilities creates unnecessary couplings between those responsibilities. Changes to one responsibility may impair the class's ability to meet the others. This leads to fragile designs that break in unexpected ways when changed.

A common misconception is that a single responsibility means a single method in a class. One function can invoke several other functions, but it **should not** be **responsible** for how these functions are implemented.

```
class Reporter(object):

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

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

The above code is an example of ***work orchestration***. It does not violate the single responsibility principle. Its only responsibility is to email report hours, and it utilised other functions, that help it achieve that responsibility. It does not need to know how these helper functions are implemented

Violation of SRP – (1)

Mixing of business logic and work orchestration

```
class Reporter(object):

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

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

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

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

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

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

A better design that conforms to SRP

```
class Reporter(object):

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

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

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

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

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

Violation of SRP – (2)

Grouping of business rules and persistence control

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

```
class Employee(object):

    # This class groups business rules and persistence

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

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

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

↓ Do we need to de-couple the responsibilities ?

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

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

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

Advantages of SRP

SRP is one of the fundamental design principles. It is simple to state, but difficult to get right. If implemented correctly, SRP helps to achieve *highly cohesive* classes that have:

- ***Readability*** - easier to focus on one responsibility and you can identify the responsibility
- ***Reusability*** - the code can be re-used in different contexts
- ***Testability*** - each responsibility can be tested in isolation. When a class has encapsulated several responsibilities, several test cases are required.

Open Closed Principle - *software entities should be open for extension but closed for modification*

Modules that satisfy OCP have two primary attributes:

- *Open for extension*: As requirements change, the module can be extended with new behaviours to adapt to the changes
- *Closed for modification:* Extending the behaviour of the module must not require changing the original source, or binary code of the module.

OCP reduces rigidity. A change does not cause a cascade of related changes in dependent modules

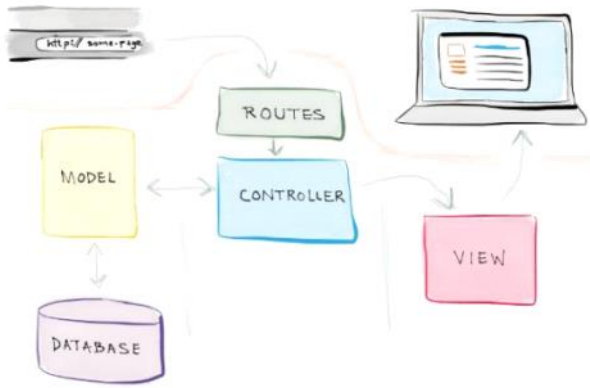
We can write software that is open for extension but closed for modification by using *abstraction* and *dynamic binding*. Abstractions are implemented as abstract base classes, that are fixed yet, represent an unbounded group of possible behaviours. The unbounded group of possible behaviours (or the extensions) are provided by possible derived (sub-) classes.

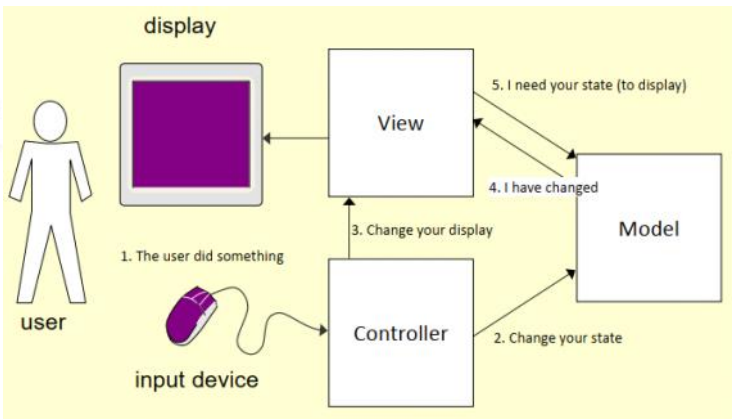
Conformance to OCP is:

- ***Not easy*** - OCP is a skill gained through experience by knowing users and industry to be able to judge various kinds of changes. Educated guesses can be wrong or right; if they are wrong, you lose time
- ***Expensive*** - abstractions increase the complexity of software design. It takes development time and effort to create the appropriate abstraction. We apply OCP only when it is needed for the first time; hence it is usually unnecessary when you first implement a system, but once requirements change, it is recommended to use OCP
- ***Beneficial*** - despite being difficult to implement and expensive, OCP provides flexibility, reusability and maintainability in the long run

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

MVC is a software architectural pattern that decouple data access, application logic and user interface into three distinct components



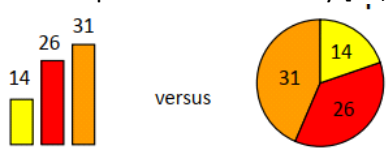


View

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

It is responsibility of the controller to choose a view to display data to the user.

For example we have an array [14, 26, 31] and we can have different views for the same model



Model

The model holds all the data and state of the system. It responds to instructions to change of state (from the controller). It also responds to requests for information about its state (usually from the view). It sends notifications of state changes to the *observer* (view)

Controller

The controller is the glue between the user and processing (*model*) and formatting (*view*) logic. It accepts the user request or inputs to the application, parses them and decides which type or mode or view should be invoked.

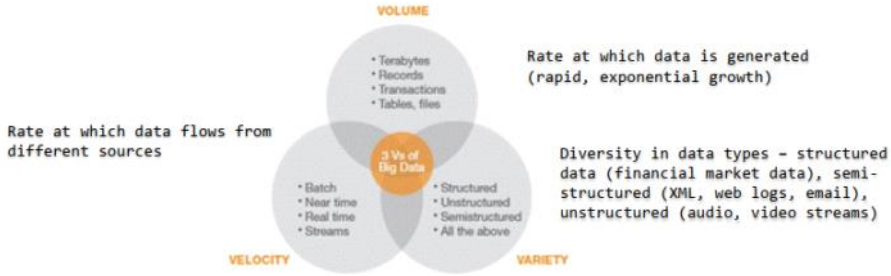
Benefits	Weakness
• Abstraction of application architecture into three high-level components (Model, View, and Controller). Model has no knowledge of the View that is provided to the user. • Supports multiple views of the same data on different platforms at the same time • Enhances testability	• Complexity • Cost of frequent updates - an active model that undergoes frequent changes could flood the views with update requests

Databases, ER Modelling

Tuesday, 9 April 2019 4:30 PM

Data science is about being able to analyse data and extract useful information to make important decisions.

Data is facts that can be recorded and have implicit meaning. Today, data is being generated at an exponential rate creating what we call big data



Data itself is not very useful, but when given context, we can transform data into ***information***. This data needs to be

- ***Stored*** in a structured format
- ***Manipulated*** efficently and usefully
- ***Shared*** by very many users (concurrently)
- ***Transmitted***

The first three points are typically handled by ***databases*** while the last is handled by ***networks***.

Nearly every computer application uses a database. These databases need to be built effectively (efficiency, security, scalability, maintainability, availability, integration, new media types). A ***database*** represents a logically coherent collection of related data.

A database management system (DBMS) is a software application that allows users to:

- Create and maintain a database (DDL - data definition language)
- Define ***queries*** to retrieve data from the database
- Perform ***transactions*** that cause some data to be written or deleted from the database (DML - data manipulation language)
- Provide concurrency, integrity, security to the database

A database and DBMS are collectively referred to as a ***database system***.

A ***data model*** describes how the data is structured in the database
There are several types of data models

- ***Relational model*** - a data structure where data is stored as a set of records known as tables. Each table consists of rows of information (also called a tuple). Each row contains fields known as columns
- ***Document model*** - data is stored in a hierarchical fashion e.g., XML
- ***Object-oriented model*** - a data structure where data is stored as a collection of objects
- ***Object-Relational model*** - a hybrid model that combines the relational and the object-oriented database models

No SQL databases facilitate the storage and analysis of unstructured data

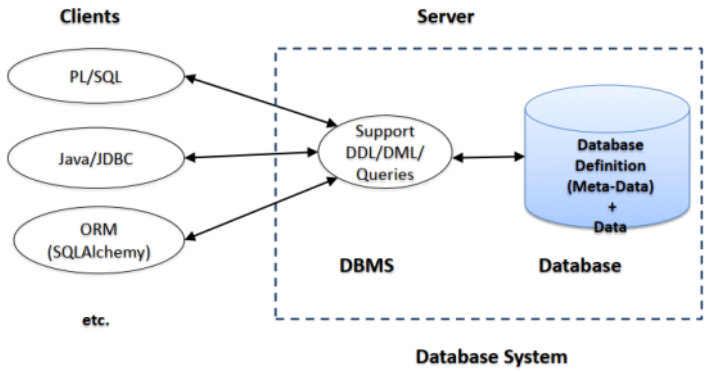
A ***database schema*** adheres to a data model and provides a logical view of the database, defining how the data is organised and the relationships between them. It is typically created before implementing a data model.
A ***database schema instance*** is the state of the database at a particular instance of time

A Relational Database Management System (RDBMS) is a DBMS that is based on a relational data model i.e., stores data as *tuples* or *records* in *tables*. It allows the user to create *relationships* between tables. Examples of relational database systems:

- Open Source - PostgreSQL, MySQL, SQLite
- Commercial - Oracle, DB2 (IBM), MS SQL Server, Sybase

Database System Architecture

Typical environment for a modern database system



SQL Queries and results travel along the client <->server links

Data Modelling for Databases

Database design

Typical steps in database design:

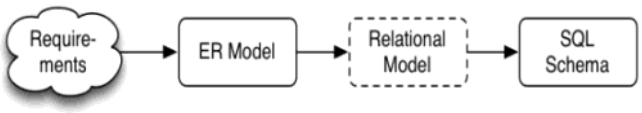
1. ***requirements analysis*** (identify data and operations)
2. **data modelling** (high-level, abstract) - an important early stage of database application development (aka "database engineering")
3. ***database schema design*** (detailed, relational model/tables)
4. ***database implementation*** (create instance of schema)
5. ***build operations/interface*** (SQL, stored procedures, GUI)
6. ***performance tuning*** (physical re-design)
7. ***schema evolution*** (logical schema re-design)

Data modelling consists of building:

- ***Logical models*** - abstract models. e.g. ER Model, OO Model
- ***Physical models*** - record-based models. e.g. relational model, classes which deal with the physical layout of data in storage

A ***data-modelling strategy*** for designing a database is to:

1. Design using an abstract model (***conceptual-level modelling***). That is, perform initial conceptual modelling with entity relationship (ER) models
2. Map to a physical model (***implementation-level modelling***). Transform the conceptual ER model into a relational model.



The aims of data modelling is to:

- Describe what ***data*** is contained in the database (e.g. entities: students, courses, accounts, branches, patients, ...)
- Describe ***relationships*** between data items (e.g. John is enrolled in COMP3311, Paul's account is held at Coogee)
- Describe ***constraints*** on data (e.g. 7-digit IDs, students can enrol in no more than 30UC per semester)

Data modelling is a design process which converts requirements into a data model.

Some Design Ideas

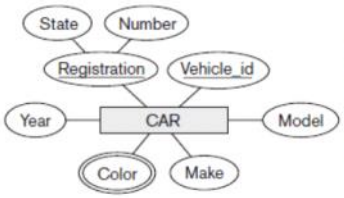
Consider the following during design:

- start simple ... evolve design as problem better understood
- identify objects (and their properties), then relationships
- most designs involve kinds (classes) of people
- keywords in requirements suggest data/relationships
  (rule-of-thumb: nouns → data, verbs → relationships)
- don't confuse operations with relationships
  (operation: he **buys** a book; relationship: the book **is owned** by him)
- consider all possible data, not just what's available

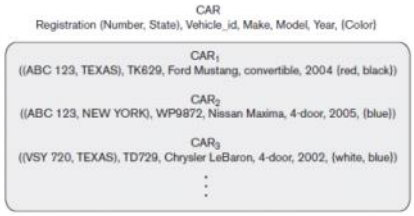
Entity Relationship Diagrams

The world is viewed as a collection of inter-related entities. ER modelling uses ***three*** major modelling constructs:

- ***entity***: a **thing** or **object** of interest in the real-world and is distinguishable from other objects
- ***attribute***: a **data item** or property of interest describing the entity *e.g., Joe (entity) described by name, address, age (attributes)*
- an **entity-set** (aka: entity-type) can be viewed as either:
  - a set of entities with the same set of attributes
  - an abstract description of a class of entities e.g., students, courses, accounts



An ER diagram showing an entity-set CAR with two key attributes (registration and vehicle_id), three single-valued attributes (year, model, make) and a multi-valued attribute (color)



An entity set CAR with three entities

Attributes in an ER model can be:

- ***Simple*** (attribute cannot be broken into smaller sub-parts) e.g., age attribute for entity type Employee
- ***Composite*** (have a hierarchy of attributes) e.g., entity type EMPLOYEE has a composite attribute Address
- ***Single-valued*** (have only one value for each entity) e.g., an vin_chassis attribute for an entity type CAR
- ***Multi-valued*** (have a set of values for each entity) • e.g., a Colors attribute for CAR = (blue,black)
- ***Derived*** (made from the value of other attributes)

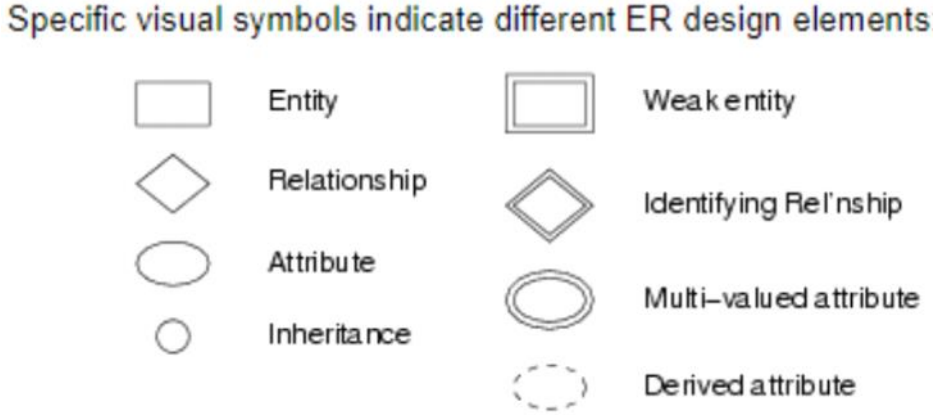
If two entities have the same set of attribute values, they are considered the same entity. So each entity must have a distinct set of attribute values. One approach would be to define a ***key (super-key)***: it is any set of attributes, whose set of values are distinct over an entity set. It can be natural (e.g. name + address + birthday) or artificial (social security number).
A ***candidate key*** is any super key, such that no subset is also a super key. e.g. (name + address) is a super key, but not (name) or (address)
A ***primary key*** is a candidate key designated by the database designer that uniquely identifies an entity.

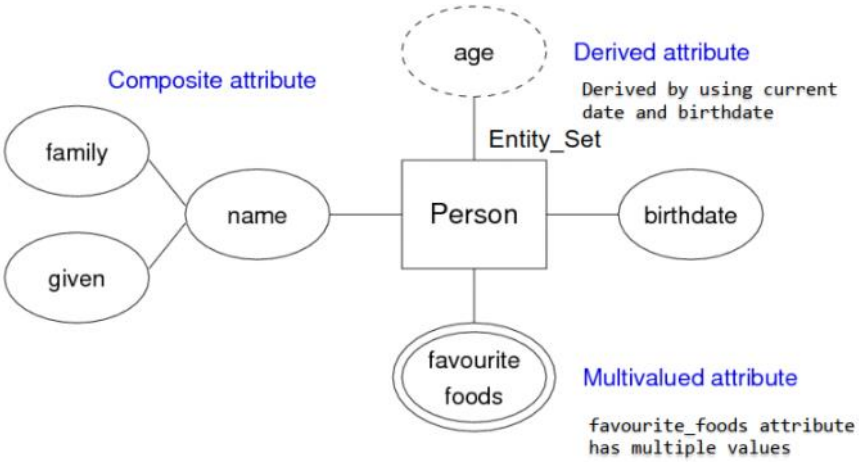
A relationship relates two or more entities. A relationship set (aka relationship type) is a set of similar relationships, associating entities belonging to one entity-set to another. The ***degree*** of a relationship is the number of entities involves in the relationship (in ER model it is always ≥ 2). The ***cardinality*** of a relationship is the number of associated entities on each side of the relationship.

We can relate ER models to an OO model, where entities are like an object instance and an entity set is a class. ER models and OO models differ mainly through the fact that an ER model models relationships not behaviour, operations or methods.

Entity Relationship Diagrams

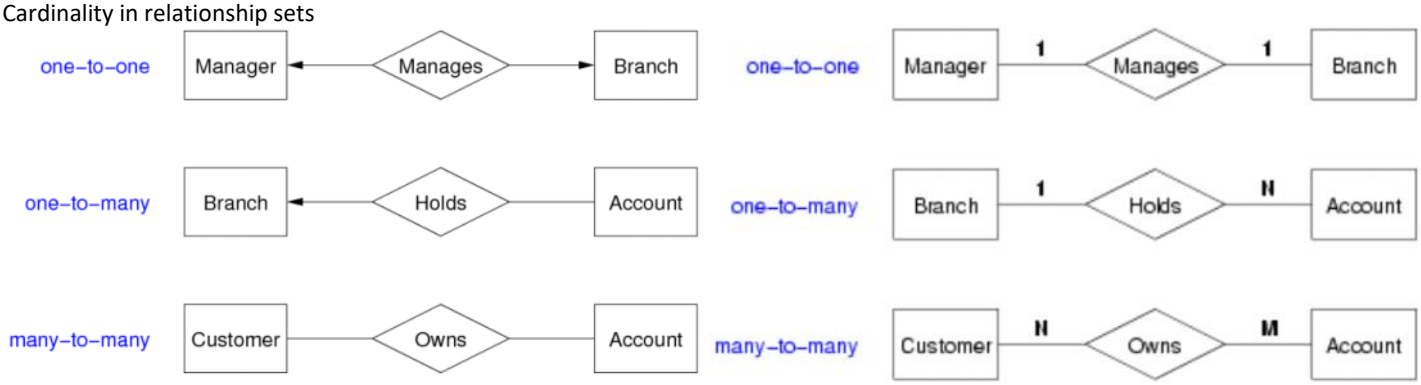
ER diagrams are a graphical tool for data modelling. An ER diagram consists of a collection of ***entity set*** definitions, a collection of ***relationship set*** definitions, ***attributes*** associated with entity and relationship sets and connections between entity and relationship sets.





Relationship Sets in ER Diagrams

Cardinality in relationship sets



The ***level of participation constraint*** is a type of relationship constraint defined as:
The participation in a relationship set ***R*** by entity set ***A*** may be

- Total - every $a \in A$ participates in more than one relationship in ***R***
- Partial - only some $a \in A$ participate in relationships in ***R***

Example:

every bank loan is associated with at least one customer

not every customer in a bank has a loan

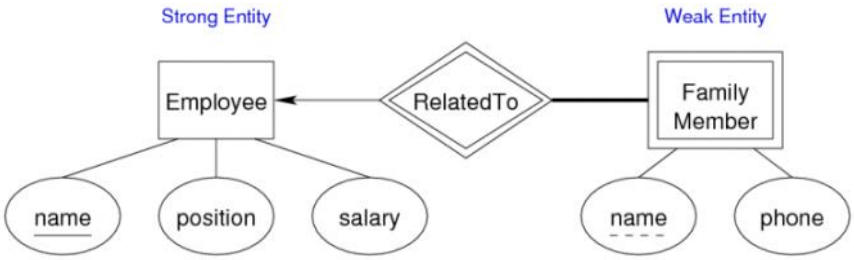
In some cases, a relationship needs associated attributes

Example:

(price and quantity are related to products in a particular shop)

Weak Entity Set

A weak entity set does not have the idea of a key, although it may have a partial key. The instances of a weak entity cannot be uniquely identified unless they are linked to an associated strong entity. Hence, they exists only because of associations with strong entities.



Subclasses and Inheritance

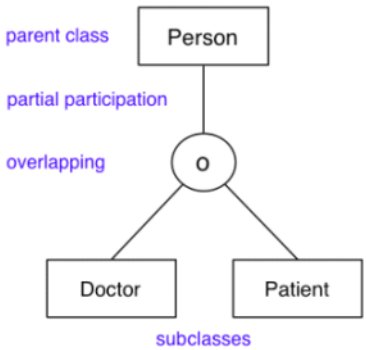
A ***subclass*** of an entity set ***A*** is a set of entities, with all attributes of ***A*** plus usually its own attributes. It is involved in all of ***A***'s relationships plus its own relationships

Properties of subclasses:

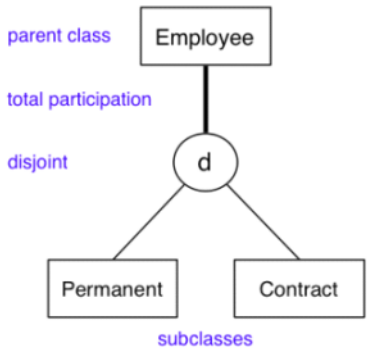
- overlapping or disjoint (can an entity be in multiple subclasses?)
  Overlapping inheritance: the entity can be this **or** that or both
  Disjoint inheritance: the entity is must be one or the other but not both
- total or partial (does every entity have to also be in a subclass?)
  Partial participation does not have to be who inherits it.
  Total participation means you cannot be the parent

If an entity has only one subclass (e.g. B is a A) we call it a specialisation

A person may be a doctor and/or may be a patient or may be neither



Every employee is either a permanent employee or works under a contract



Design considerations when using the ER model

- should an "object" be represented by an attribute or entity?
- is a "concept" best expressed as an entity or relationship?
- should we use *n*-way relationship or several 2-way relationships?
- is an "object" a strong or weak entity? (usually strong)
- are there subclasses/superclasses within the entities?

ER diagrams are typically too large to fit on a single screen (or a single sheet of paper, if printing)
One commonly used strategy is to define entity sets separately, their showing attributes. Then combine entities and relationships on a single diagram (but without showing the entity's attributes). If the design is very large, you may use several linked diagrams.

3-Tier Client Server Architecture.

1. Client sends request
2. Middle app which processes the request
3. Back end data tier which persists (stores) your data

ER and Relational Model

Thursday, 18 April 2019 10:24 PM

The relational model describes the world as a collection of inter-connected ***relations*** (or ***tables***). The goal of the relational model is a simple, general data modelling formalism, which maps easily to file structures (i.e. it is implementable). Relational models have two styles of terminology

Mathematical	Relation	Tuple	Attribute
Data-oriented	Table	Record (row)	Field (column)

The relational model has one structuring mechanism ...

- a relation corresponds to a mathematical "relation" and can also be viewed as a "table"

Each relation (table) (denoted *R,S,T,...*) has:

- a name (unique within a given database)
- a set of attributes (or column headings)

Each attribute (denoted *A,B,...* or *a1,a2,...*) has:

- a name (unique within a given relation)
- an associated domain (set of allowed values)

DB definitions also make extensive use of constraints

A ***tuple (row)*** is a set of ***values (attribute or column values)***. Attribute values:

- Are ***atomic*** (there are no composite or multi-valued attributes). Derived attributes do not need to be modelled in a relational model
- Belong to a ***domain***, which has a name, data type and format. A distinguished NULL value belongs to all domains. A NULL has several interpretations; none, don't know, irrelevant

Column Header	Domain Name	Domain Data Type, Format and Constrain
phone_number	local_phone_numbers - (set of phone numbers valid in australia)	character string of the format *(dd) ddddddd*, where each *d* is a numeric (decimal) digit and the first two digits form a valid telephone area code.
age	employee_age (set of possible ages for employees in the company)	An integer value between 15 and 80

A ***relation (table)*** is a set of tuples. Since a relation is a set, there is ***no ordering*** of rows. Normally we define a standard ordering on components of a tuple. The following are different representations of the same relation:

branchName	accountNo	balance
Downtown	A–101	500
Mianus	A–215	700
Perryridge	A–102	400
Round Hill	A–305	350
Redwood	A–222	700

accountNo	branchName	balance
A–305	Round Hill	350
A–222	Redwood	700
A–215	Mianus	700
A–102	Perryridge	400
A–101	Downtown	500

Each relation generally has a primary key (a subset of attributes, unique over the relation).

A ***database*** is a set of relations (tables).

Expressing Relational Data Model Mathematically

Given a relation (table) *R* which has:

- *n* attributes $a_1, a_2, \ldots A_n$
- with corresponding domains $D_1, D_2, \ldots D_n$

We define:

- Relation Schema of *R* as: $R(a_1{:}D_1, a_2{:}D_2, \ldots a_n{:}D_n)$
- Tuple of *R* as: an element of $D_1 \times D_2 \times \ldots \times D_n$ (i.e. list of values)
- Instance of *R* as: subset of $D_1 \times D_2 \times \ldots \times D_n$ (i.e. set of tuples)
- Database schema : a collection of relation schemas.
- Database (instance) : a collection of relation instances

The **degree** (or **arity**) of a relation is the number of attributes *n* of its relation schema, so a relation schema *R* of degree *n* is denoted by $R(a_1, a_2, \ldots a_n)$

Constraints

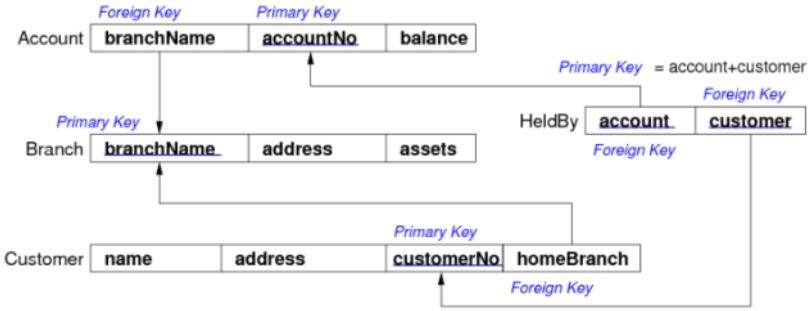
Relations are used to represent *real-world entities* and *relationships* between these entities. To represent real-world

problems, we need to describe:

- What values are/aren't allowed
- What combinations of values are/aren't allowed

Constraints are logical statements that do this. We have:

- ***Domain constraint*** - this specifies that within each tuple, the value of each attribute $(a_1, a_2, \ldots a_n)$ must be an atomic value from the corresponding domain $D_1, D_2, \ldots D_n$. Note that NULL satisfies all domain constraints (except NOT NULL)
- ***Key constraint*** - a ***key*** attribute is one whose value can be used to uniquely identify each tuple in the relation. A relation can have more than one key, so each key is a ***candidate key***. One of the candidate keys is designated as the primary key of the relation. An ***entity integrity constraint*** states no primary key value can be NULL
- ***Referential integrity*** - describes references between relations (tables). They are related to the notion of a ***foreign key*** (FK). A set of attributes F in relation R_1 is a foreign key if the attributes in F correspond to the attributes in the primary key another relation R_2 or the value for F in each tuple R_1 either occurs as a primary key in R_2 or is entirely NULL. Foreign keys are critical in relational databases for linking individual relations and are a way to assemble query answers from multiple tables.
  Example: The notion that the branchName (in Account) must refer to a valid branchName (in Branch) is a ***referential integrity constraint***



A DBMS should provide capabilities to enforce these constraints

A relational database schema is viewed as: a set of relation schemas *{ R1, R2, ... Rn }* , and a set of integrity constraints
A relational database instance is a set of relation instances *{ r1(R1), r2(R2), ... rn(Rn) }.* where all of the integrity constraints are satisfied
One of the important functions of a relational DBMS is to ensure that all data in the database satisfies constraints

The relational model is a ***mathematical construct*** giving a representation for data structures with constraints on relations/tuples and an *algebra* for manipulating relations/tuples (union, intersect…)

Relational Database Management Systems (RDBMS) provide an implementation of the relational model. It uses SQL (Structured Query Language) as language for data definition, query, updates

Relational schemas can be described using SQL (Structured Query Language), which provides formalism to express relational schemas. SQL provides a ***Data Definition Language*** for creating relations.

```
CREATE TABLE TableName (
attrName1 domain1 constraints1 ,
attrName2 domain2 constraints2 , ...
PRIMARY KEY (attri,attrj,...)
FOREIGN KEY (attrx,attry,...)
REFERENCES OtherTable (attrm,attrn,...)
);
```

To remember:

- DBMS-level ... database names must be unique
- database-level ... schema names must be unique
- schema-level ... table names must be unique
- table-level ... attribute names must be unique

Sometimes it is convenient to use same name in several tables

We distinguish which attribute we mean using qualified names
e.g. Account.branchName vs Branch.branchName

Features of RDBMS

- Support large-scale data-intensive applications
- Provide efficient storage and retrieval (disk/memory management)
- Support multiple simultaneous users (privilege, protection)
- Support multiple simultaneous operations (transactions, concurrency)
- Maintain reliable access to the stored data (backup, recovery)
- Use SQL as language for:
  - data definition (creating, deleting relations i.e. tables)
  - relation query (selecting tuples)
  - relation update (changing relations)

Mapping ER designs to Relational Model

A formal mapping between ER and relational models has been developed

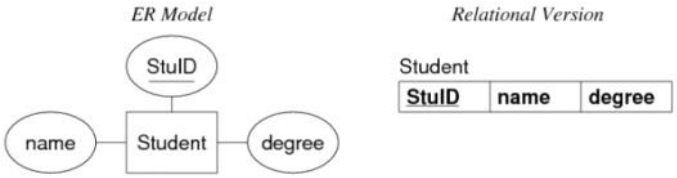
ER Model	Relational Model
ER attribute	Attribute (atomic)

ER entity-instance ER relationship-instance	Tuple (row)
ER entity-set ER relationship	Relation (table)
ER key	Primary key of relation

There are also differences between relational and ER models. Compared to ER, the relational model:

- Uses *relations* to model *entities* and *relationships*
- Has **no** *composite* or *multi-valued* attributes (only atomic)
- Has **no** *object-oriented notions* (e.g. subclasses, inheritance)

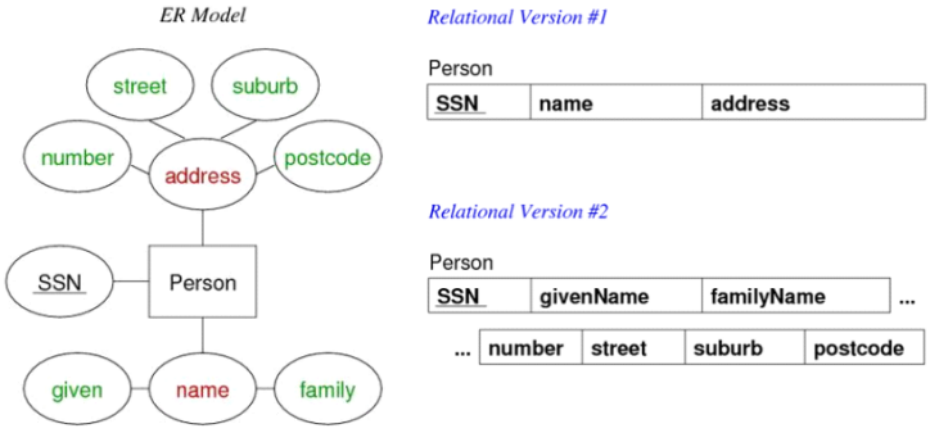
1. **Mapping Strong Entities**
   An entity consists of a collection of attributes; attributes are ***simple***, ***composite*** and ***multi-valued***
   A relation schema consists of a collection of attributes; all attributes have ***atomic*** data values.
   An obvious mapping is an entity set *E* with atomic attributes a_1, a_2, … a_n maps to a relation (table) *R* with attributes (columns) a_1, a_2, … a_n



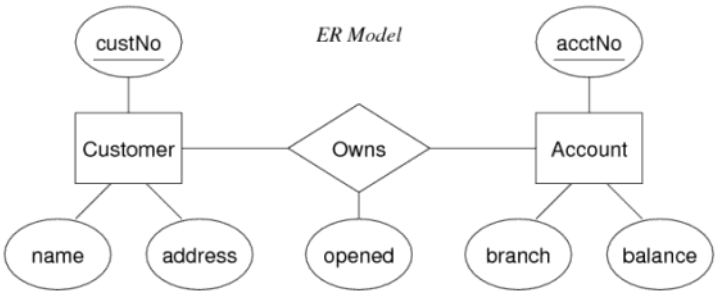
(Note: the key is preserved in the mapping)

2. **Mapping Composite Attributes**
   ER supports composite (hierarchical) attributes. The relational model supports only atomic attributes.
   Composite attributes consist of
   - structuring attributes (non-leaf attributes)
   - data attributes (containing atomic values)

   Approach 1: remove structuring attributes. Map atomic components to a set of atomic attributes (possibly with renaming)
   Approach 2: concatenate atomic attribute values into a string. Requires extra extraction effort if components are required and cannot exploit efficient query capabilities on components



3. **Mapping Relations**
   Identify one entity as "parent" and other entity as "child". As a general rule, *the PK of parent is added to child as FK*. Any attributes of the relationship are added to ***child*** relation
   a. **Mapping N:M Relationships**



Relational Version

Customer

custNo	**name**	**address**

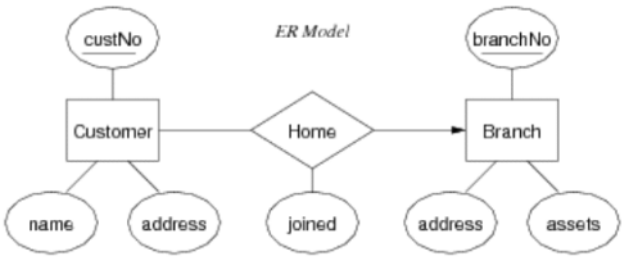
Account

acctNo	**branch**	**balance**

Owns

custNo	**acctNo**	**opened**

a. Mapping 1:M Relationships



Relational Version

Generic Mapping

Customer

custNo	**name**	**address**

Branch

branchNo	**address**	**assets**

Home

custNo	**branchNo**	**joined**

Optimised Mapping

Customer

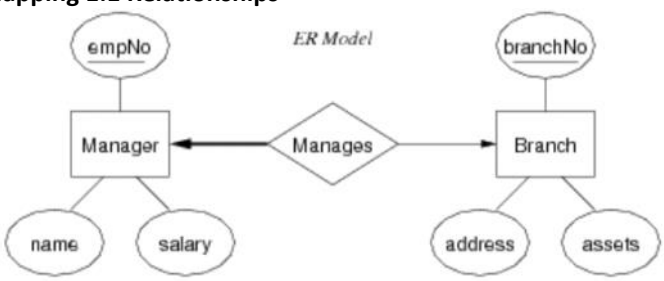
custNo	**name**	**address**	**branchNo**	**joined**

Branch

branchNo	**address**	**assets**

64

a. Mapping 1:1 Relationships

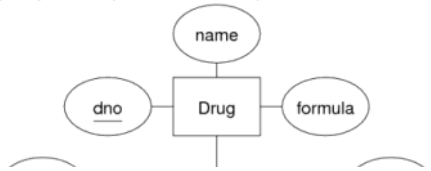


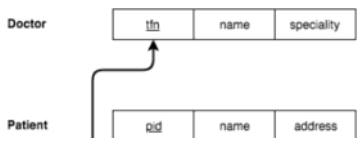
Handled similarly to 1:N relationships

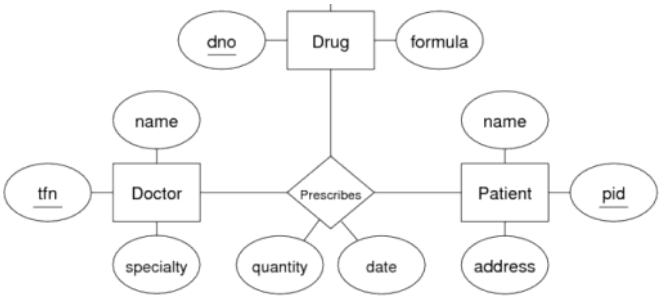
For a 1:1 relationship between entity sets *E* and *F* (*S* and *T*):

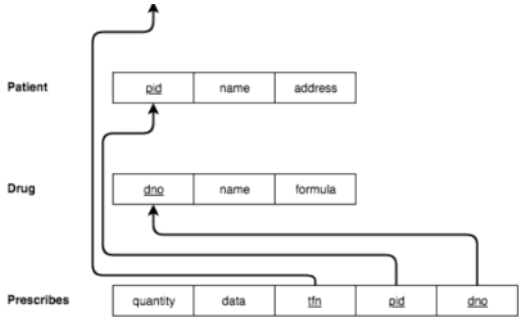
- choose one of *S* and *T* (e.g. *S*) (*Note : Choose the entity set that participates totally, if only one of them does*)
- add the attributes of *T*'s primary key to *S* as foreign key
- add the relationship attributes as attributes of *S*

a. Mapping n-way Relationships







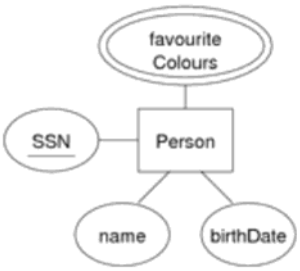


4. **Mapping Multi-valued Attributes**

Treat like an N:M relationship between entities and values. So we create new relation where each tuple contains:

- The primary key attributes from the entity
- One value for the multi-valued attribute from the corresponding entity

ER Model



Relational Version

Person

SSN	name	birthdate

FavColour

SSN	colour

Example: the two entities

```
Person(12345, John, 12-feb-1990, [red,green,blue])
Person(54321, Jane, 25-dec-1990, [green,purple])
```

would be represented as:

```
Person(12345, John, 12-feb-1990)
Person(54321, Jane, 25-dec-1990)

FavColour (12345, red)
FavColour(12345, green)
FavColour(12345, blue)
FavColour(54321, green)
FavColour(54321, purple)
```

5. **Mapping sub-classes**

   a. **ER style**

Each entity becomes a separate table, containing attributes of the subclass and a foreign key to the super class

Relational Model

Person

SSN	name	address

Employee

SSN	salary	position

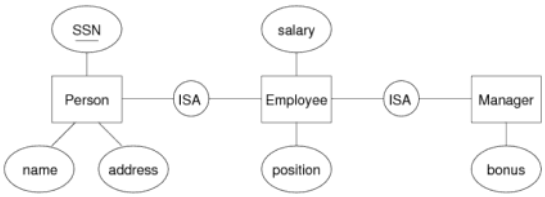
Manager

SSN	bonus

   a. **OO style**

Each entity becomes a separate table, inheriting all attributes from all superclasses

Entity–Relationship Model



Relational Model

Person

SSN	name	address

Employee

SSN	name	address	salary	position

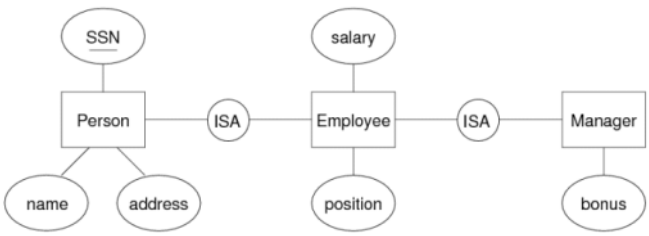
Manager

SSN	name	address	salary	position	bonus

   a. **Single-table style**

The whole class hierarchy becomes one table, containing all attributes of all subclasses (NULL, if unused)

Entity–Relationship Model



Relational Model

Person

SSN	name	address	salary	position	bonus

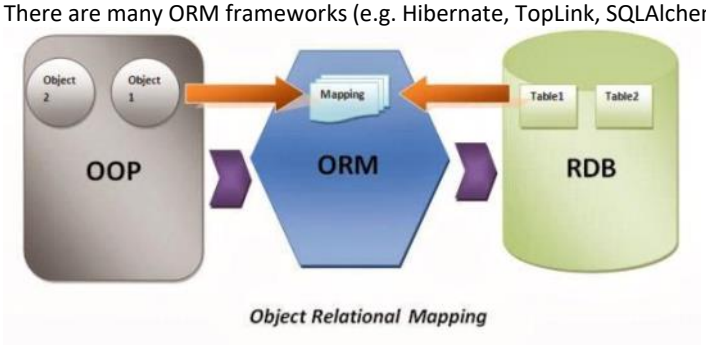
NULL for Person who is not Employee

NULL for Employee who is not Manager

Which mapping is best depends on how data is to be used.

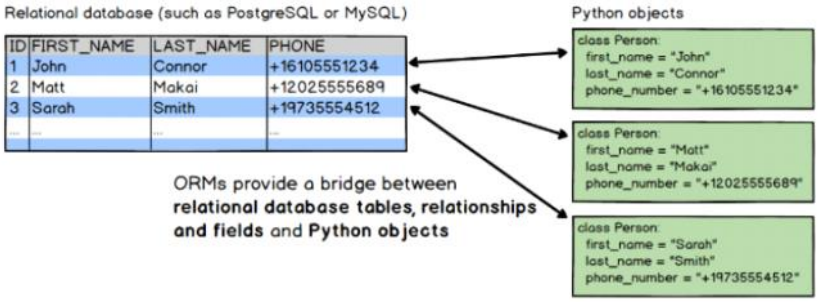
Introduction ORM (Object Relational Mapper)

ORM is a high-level abstraction framework that maps a relational database system to objects. To automates all CRUD (Create/Retrieve/Update/Delete) operations. ORM is agnostic to which relational database is used (in theory). There are many ORM frameworks (e.g. Hibernate, TopLink, SQLAlchemy)



Object Relational Mapping

ORM shields the developer from having to write complex SQL statements and focus on the application logic using their choice of programming language. Harmonisation of data types between the OO language and the SQL database. ORM automates the transfer of data stored in relational database tables into objects.



Software Architecture

Tuesday, 23 April 2019 4:06 PM

One power outage is enough to cripple your whole system in model telephone architectures.
Old telephone systems worked with analogue signals that were sent through copper wires. Nowadays telephone signals are digital, meaning that you will lose the any ability to communicate if the system is down.

The POTS system is reliable while the modern telephone system allows extensions on its functionality.

3 tier
- Client - talks to the app server, initiates requests
- Application server - talks to the database tier
- Database -

As software systems increase in size, complexity and become distributed, the design problem extends beyond the algorithms and data structures of computation. It becomes increasingly vital to specify the overall system structure.

Software architecture as a concept has its origins in the research of Edwards Dijkstra in 1968 and David Parnas in the early 1970s, who emphasised that the structure of a software system matters and getting the structure right is critical

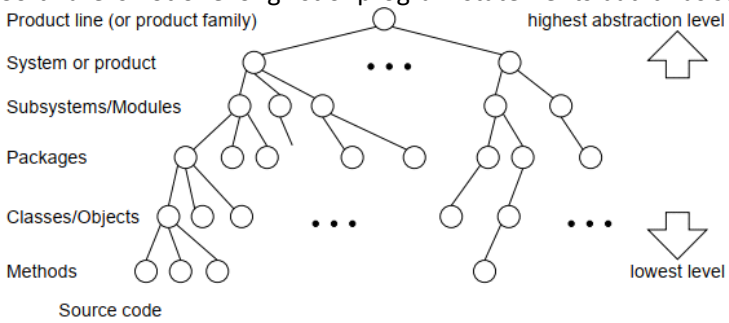
Software architecture is the *big picture* or macroscopic organisation of the system to be built. It partitions the system into logical subsystems or parts, then provides a high-level view of the system in terms of these parts and who they relate to form the whole system.

Architecture vs Design

Architecture focuses on non-functional requirements ("cross-cutting concerns") and the decomposition of functional requirements.
Design focuses on implementing the functional requirements

Hierarchical Organisation of Software

Software is not one long list of program statements but it has structure



Reasons why we decompose systems:

- ***Understanding and communication*** - this enables everyone to understand how the system works as a whole (various stakeholders, users of the system, developers, architects). It also provides an understanding of the macro properties of the system to learn how the system intends to fulfil the key functional and non-functional requirements. Through this we can pre-determine key system properties (scalability, reliability, performance, usability, etc.)
- ***Divide and conquer complexity*** - this allows people to work on individual pieces of the system in isolation which can later be integrated to form the final system
- ***Prepare for extension*** of the system - this supports flexibility and future evolution by decoupling unrelated parts, so each can evolve separately ("*separation of concerns*"). Subsystems envisioned to be part of a future release can be included in the architecture, even though they are not to be developed immediately
- ***Focus on creative parts*** and avoid *reinventing the wheel* - discover components obtained from past projects and identifying which have the highest potential for reusability

Some key questions we are faced with:

1. How to decompose the system? What components are needed in the system. The architectural style will dictate

how the system will be broken down
2. How the parts relate to one another?
3. How to document the system's software architecture? The architectural view will dictate how the system will be documented

Software Architectural Styles

Software architectural styles are defined as a family of systems in terms of a pattern of structural organisation. Patterns were described as abstract representations with an aim to:

- Make it easy to communicate among stakeholders
- Document early design decision
- Allow for reuse

An architectural style is defined by:

1. ***Components*** - a collection of computational units or elements that *do the work* (e.g. classes, databases, tools, processes etc.)
2. ***Connectors*** - enable communication between different components of the system (e.g. function all, remote procedure call, event broadcast etc) and uses a specific protocol
3. ***Constraints*** - define how the components can be combined to form the system. It defines where the data may flow in and out of the components/connections. Topological constraints define the arrangement of the components and connectors

Client-Server architecture

When there is a need to share data between a client and a service provider distributed geographically across different locations a client-server architecture is used.

Client-server architecture is a basic architectural style for distributed computing. There are two distinct components:

- A server, which provides specific services, waits for and handles connections
- A clients that initiates connections to requests services provided by a server

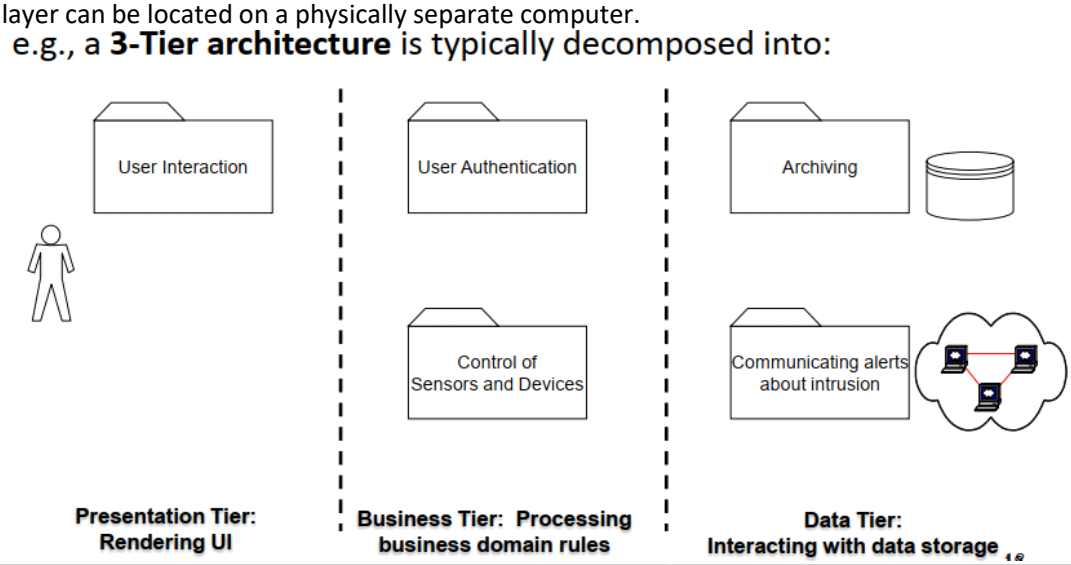
The client and the server can be on the same machine or it can be on different machines.
The connector is based on a ***request-response*** model.

Examples include a file server, database server, and email server

The biggest problem with client server architecture is the single point of failure. If the server goes down, then the client cannot make any requests

3-Tier/N-Tier Architecture

3-Tier/N-Tier Architecture separated the deployment of software components into multiple logical layers, so that each layer can be located on a physically separate computer.

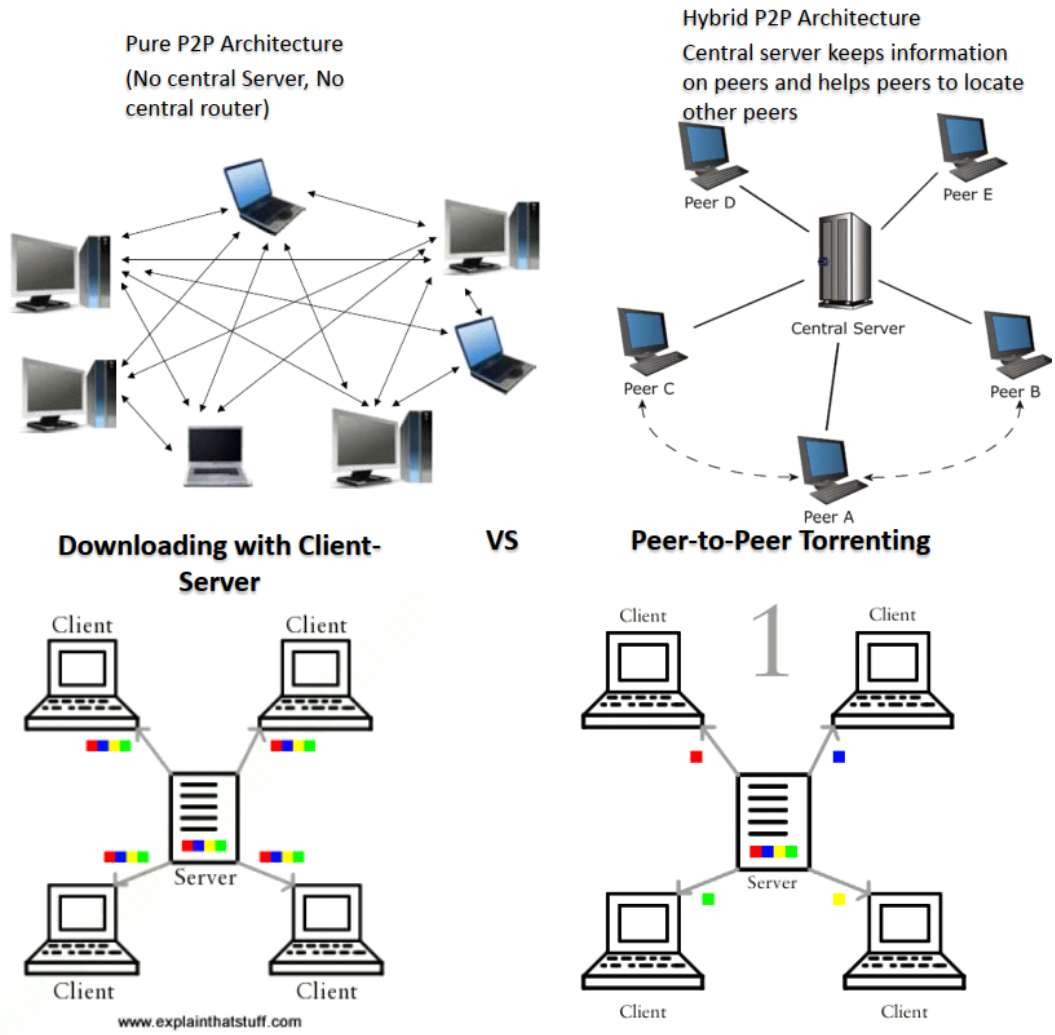


Benefits	Weaknesses
• Makes effective use of network and distribution of data is straightforward • Roles and responsibilities of system distributed among	• Single point of failure – when a server is down, client requests cannot be met • Network congestion, when large number of client requests

several independent machines • Easy to add new servers or upgrade existing servers • Deployment of modules to different servers enhancing security, scalability and performance	are made simultaneously to the server • Complex and expensive infrastructure

Peer-to-Peer (P2P)

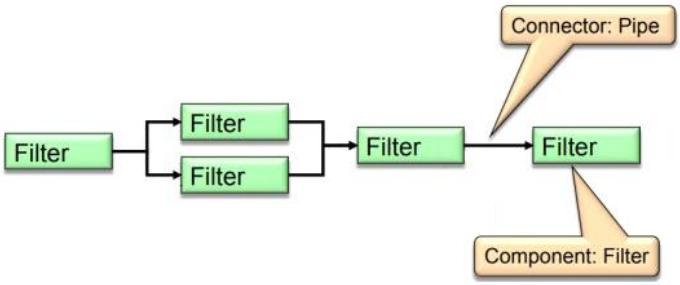
P2P resolves the network congestion and single point of failure that could result in a client-server model.
Each peer component can function as both a client and a server. Information is distributed among peers and any two components can set up a communication channel to exchange information as needed.



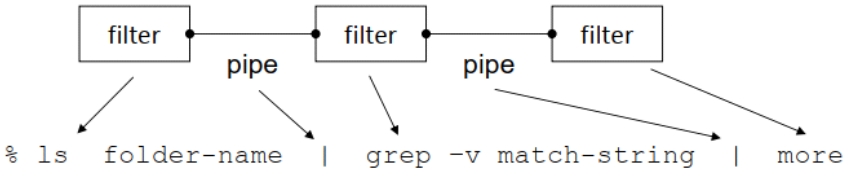
Benefits	Weaknesses
• Efficiency: More efficient as all clients provide resources • Scalability: Unlike client-server, capacity of the network increases with number of clients • Robustness: – Data is replicated over peers – immune to single point of failure, e.g., if a node failed to download a file, the remaining nodes still have the data needed to complete the download	• Architectural complexity • Distributed resources are not always available. It requires peers to be stay online for chunks of files to exist • More demanding peers

Pipe-and-Filter

Pipe-and-Filter is a style suitable for processing and transforming data streams. It aims to produces an output stream by transforming the input data stream.
Its component is a ***filter***, which reads input streams. Each filter encapsulates a data processing step to transform data and produce output streams.
Its connector is a ***pipe***, serving as a conduit for data. The data transformed by one filter is sent to other filters for further processing using the pipe connector



In Pipe-and-Filter, data is processed incrementally as it arrive. Output can begin before input has been fully entered. Filters must be independent; they do not share state and do not know upstream or downstream filters. E.g. UNIX shell commands and pipers, compilers



Benefits	Weaknesses
• Easy to understand the overall input/output behaviour of a system as a simple composition of the behaviours of filters – Different data processing steps are decoupled so filters can evolve independently of each other • Flexible and support reuse - any two filters can be recombined, if they agree on data formats • Flexible and ease of maintenance - filters can be recombined or easily replaced by new filters • Support concurrent processing of data streams	• Highly dependent on the order of filters. If an intermediate filter crashes, so does the system • Not appropriate for interactive applications

Central Repository (Database)

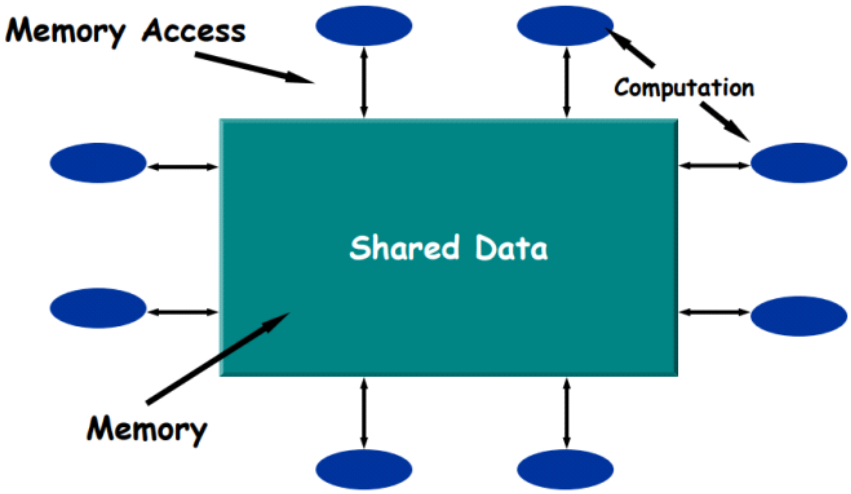
This style is used when a complex body of knowledge, that needs to be established, persisted and manipulated in several ways.

It has two distinct types of components:

- Central data repository - a central, reliable, permanent data structure that represents the state of the system
- Data accessors - a collection of independent components that operate on the central data; components do not interact directly with each other, only through the central repository

The connectors are read/write mechanisms (e.g. procedure calls, or direct memory accesses)

Components access the repository whenever they want. The knowledge sources do no change the shared data. Examples include: graphical editors, IDEs, database applications, document repositories



Benefits	Weaknesses
• Efficient way to share large amounts of data	• All independent components must agree upon a

• Centralised management of repository - concurrency access and data integrity - security - backup • Components do not need to be concerned with how data is produced	repository data model • Distribution of data can be a problem • Connectors implement complex infrastructure

Publish-Subscribe Style (Event based style)

This style has two distinct components:

- A publisher - components which generate or publish events
- A subscriber - components that register interest in published events and are invoked when these events occur

The subscriber is aware of the publisher while the publisher is not aware of the subscriber.

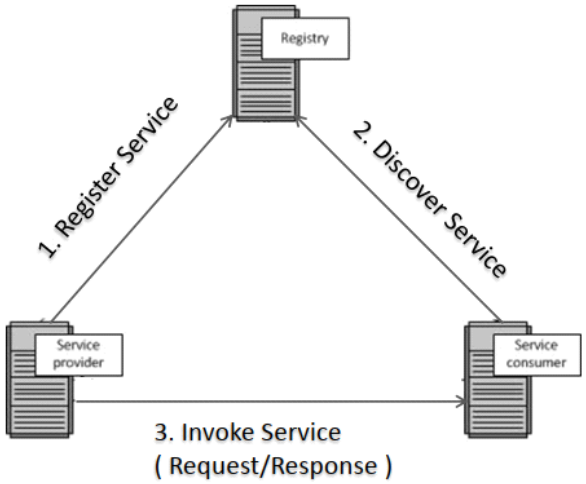
A few examples include:

- User interface frameworks
- Traders registering interest to particular stock prices and receive updates as prices change
- Wireless sensor networks

Service Oriented Architecture (SOA)

Software components are created as autonomous, platform-independent, loosely coupled services and is unaware of the underlying implementation of technology.

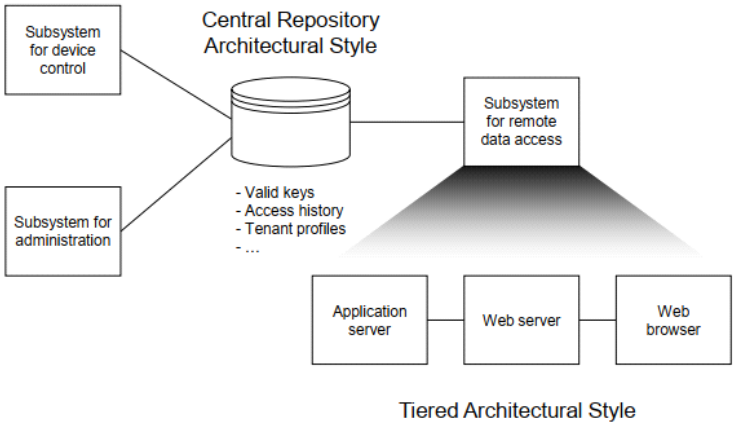
One type of service is a web service. Applications include Business-to-Business (B2B) services



Every system has an architecture, but not every architecture suits the application. The "best" design for a system:

- Depends on what criteria are used to decide the "goodness" of a design
- System requirements (functional needs, quality needs (e.g. performance, security)
- Business priorities (alignment to requirements)
- Available resources, core competences, target customers, competitors' moves, technology trends, existing investments, backward compatibility, …
- Hard to change architecture later. This does not mean BDUF (big design up front) but, need to think "enough"

A real system is more likely to be a combination of styles



Tiered Architectural Style

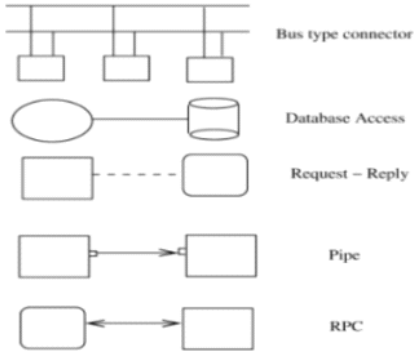
Architectural Views

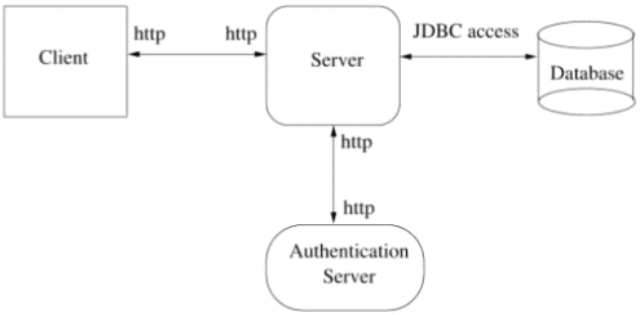
The ***model view*** decomposes the *functionality* into a coherent set of software (code) units. It is the logical break-down into sub-systems often shown using ***package diagrams***. Interactions among components at run-time are shown using sequence diagrams, activity diagrams. Data shared between low-level components and collaborations are typically expressed as UML class diagrams.

The ***component and connector view*** describes at runtime the structure of the system (components, data stores, connectors (e.g. pipes, sockets that enable interaction between components). This can be modelled using box-and-line diagrams (informal), UML component diagrams (formal)

Component and Connector View: Informal Box-Line diagram

- An informal approach to creating a component and connector view
- Often drawn on a white-board



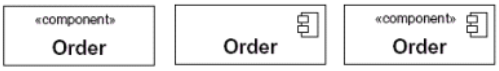


44

Component & Connector View: UML 2 component diagram

Use a formal notation to visualise the static implementation of a system and understand the wiring of the various components of the system, which includes:

Component: An independent, autonomous unit within a system or sub-system that represents a software module of classes, web service or some software resource, typically implemented as a replaceable module and can be deployed independently



A rectangle with a visual stereotype in the top right corner or a textual stereotype of <<component>> or both

Component Interfaces:

A UML 2 component defines two sets of interfaces:

Provided interfaces : a collection of one or more methods that define the services offered by the component and represent a formal contract with a consumer

Required interfaces : dependency services required by the component in order to perform its function

Provided interfaces: OrderEntry, AccountPayable

Lollipop notation (straight line with an attached circle)

Required interfaces: Person

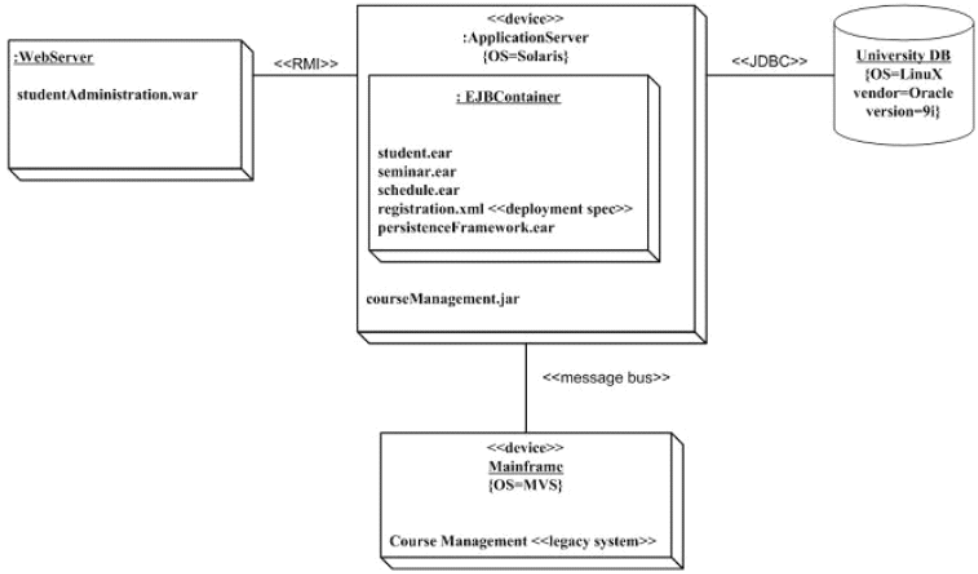
(A straight line with a half circle)

45

The ***allocation view*** describes how the software units map to the environment (hardware resources, file-systems and people). It exposes properties like which process runs on which processor. It is typically expressed as a UML deployment diagram. It is a static view of the run-time configuration of the processing nodes and the components that

run on these nodes. It shows the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another. It is ideal for applications deployed to several machines e.g., thin-client network computer which interacts with several internal
servers behind your corporate firewall

Example of a UML 2 deployment diagram



MVC VS Software architecture

- Software is more about the infrastructure hardware layer
- MVC is more of an application architecture. How the software components are broken down into sub models. Bringing a