|  |  |
| --- | --- |
| 1. **Description of main programming language used techniques** | **C#**: Open source framework owned by **Microsoft**. Based off **Mono**, C# is a modern high-level language that implements **Microsoft’s .NET framework**. Versatile and extensive in its **modularity**, C# is a well performing interim between **scripting languages** and lower level frameworks like C/C++ which is used across all **Microsoft platforms**.  **Java**: Another **open source framework**, Java is a heavy-duty language that is designed to be **written once and ran in any environment**. Owned by **Oracle** and is currently subject to heavy commercialization, Java’s popularity is questionable now but its **modularity** and **open sourced code** is undeniably **practical**. Currently an industry standard for programming.  **HTML**: Hyper Text Markup Language. **Scripting language** used as the foundation for all modern web technologies. Features a tagging syntax which **eXtensible Modifying Language**(XML) is heavily based off. Modern browser technologies allow for real-time modification of HTML so anything can be considered an IDE which begs the question why there still isn’t a C# IDE better (less bloated) than Visual Studio.  **XML**: **eXtensible Modifying Language**(XML), **tag based framework** that is heavily shilled by **Microsoft**. Somehow became an **industry standard** even when the concept is that of a **framework that can break standards whenever it wants** and tends to do so for any arbitrary reason just to make my life more difficult.  **JavaScript**: Another **web-based scripting language**, JavaScript(JS) allows for **more potent programmability** than HTML and is commonly used to **automate behaviours** of web services like **event handling** on the **client side**. Easy to learn, **impossible to master** (don’t bother).  **PHP**: **Hypertext Preprocessor**, PHP is a framework which is used as the **intermediary between client and server** for web services. It allows for **greater programmability** of web technologies and processing is done **server-side** which allows for **less invasive web programming**.  **Electron**: A **development framework** that runs off **Google’s V8** web rendering engine, Electron combines web technologies with a programmable API to make **applications that can be rendered anywhere** a web browser could. Although Electron is using what could be considered **immature technology**, it is an **industry pioneer** and leader which makes use of **pre-existing and heavily proven frameworks** (Node.js, Bootstrap, etc.). |
| 1. **Description of aspects of system** | **Client/Workstation**: The machine which is responsible for the **lowest level of use** and tends to see interaction mainly from **non-primary stakeholders**. Key considerations to make when developing software are the **wildly varying hardware/software specifications** of workstation machines as well as **broad-scale deployment** when it comes to licensing issues, timeframes, etc.  **Server**: The machine which is usually responsible for **hosting remote services**, the servers are the cornerstone of any organization that relies on **remote connectivity** in any form. Interactions with server machines tend to be **strictly limited at a low level** so servers are extremely useful at **automating menial tasks** like fetching and storing. Considerations to make for software developers to make are **functional redundancies** (backups, restorations, etc.), physical properties of the **network**, scalability and accounting for **downtime**.  **Network**: Used to describe the **connection** made between two computers during any interaction with each other. Networks can be **physical** (Local Area Network) or **virtual** (Wide Area Networks, or World Wide Web) but will always involve **computer interaction** of some sort. Important considerations to make are the **physical properties** of a network (bandwidth, physical distance, routing, etc.) and **security**.  **Computer**: A machine that is used as a **tool** to solve **complex arithmetic problems** and can be used for **processing logical actions**. You’re probably sitting in front of one right now.  **Domain**: The space in which an entity **“exists”** in. Either physical, conceptual or metaphysical, a domain refers to a **workable space** in terms of networking. In web development, it also refers to a **web space** which is used to display or render **web services**.  **Entity**: A **conceptual manifestation of information**, data or an object of any sort. May not necessarily be used to refer to something **virtual** but in terms of **ICT** it is very heavily referenced as an **idealization of a concept**. In the scope of **database development**, an Entity refers to a **dataset or table** before its **realization into a database model**.  **Database**: The **repository** for any **information** being used in any sort of **information processing job**. The database tends to support the entire **backend** of a business model and is the cause of many **headaches**. Uses **SQL** as a **programmable interface**. Considerations to make as a software developer include **Database Management Systems** (DBMS) as well as **data concurrency and accuracy**. |
| 1. **Description of UML** | **UML**: Universal Modelling Language, not so much a language but a **set of jargons** which are used by ICT workers to describe a system in a more **comprehensible** manner to the layman (as hard as they may try). Only recently became a **standard**. Most diagrams used in **software development** heavily use UML so it is a good language to **try and** **understand**.  **System**: A very **broad** and overly **basic term** generally used to describe an **overarching** conglomeration of **logical objects** and **behaviours** which support the **business case** being presented. Trevor hates this **word** as much as I hate the word **“flavourful”**.  **Actors:** Term used to describe any **person or thing** that **interacts with a programmable system**. The aim of the game is to condition the Actor’s **behavior** so that almost all typical actions done by the actor will be funneled into something which is hopefully in the **scope** of the system being **designed** or **described**. Represented in UML diagrams as a **stick figure**.  **Domains**: The name of a **conceptual dataset** and usually only seen or referred to in the **early stages** of design. The **predecessor** to a class. Represented by a **vertically compartmentalized box** (sideways Italian flag).  **Classes**: Name given to a **Domain** UML object that contains a **domain name**, **parameters** and/or **behaviours**. This is the more **completed** version of a Domain and generally **proceeds** the Domain after the Domain is conceived. Class objects in UML **describe** and are meant to more directly translate into classes in an **OOP** sense. Also represented by a **vertically compartmentalized box**.  **Entities**: A UML **object** that represents a **Class** in any sort of **behavioural diagram**. Represented by a **circle sitting on a flat line**.  **Interfaces**: A UML object that represents a **programmable layer** between the **system** and **actors** it is designed to **influence** (interface). Usually describes a **User Interface**(UI) or **Application Programmable Interface**(API) from within a system. Represented by a **circle leaning against a wall/post/stick/line**.  **Controller**: A UML object that describes an **entity** that models the **behaviors** of a given **system**. Deals with the concept of a **separation of concerns** in a software developing sense (read more on **Model View Controller** patterns). Represented by a **loading symbol**.  **Use Case (Diagram)**: A UML object that **summarises** the **behaviours** of the system in the **context** of an Actor’s **actions**. Represented by **ellipses** and generally only found in a **Use Case Diagram**, Use Cases are a **basic** way to represent **modelled behaviours** and outside the context of UML can also refer to any possibly **meaningful interaction** the user/Actor has with a **system**. They are a good way to **visually represent** which behaviours are **associated** with which Actors in a system.  **Domain/Class Diagram**: A visual representation of a program’s **basic (physical) architecture**, the Class Diagram splits a system into **Classes** which falls in line with **Object Oriented principles**. The diagram models the **associations** one class may have with another which **directly translates** into an approach a **programmer** has to take when **making the class** when using an **OOP language**. Extremely **useful reference** for a programmer when he/she finally gets to cut some **code**.  **Generalization:** A UML term and **programming concept** that is used to describe the **inheritance of properties** from one class to another. Represented in UML by a **solid-lined arrow with a hollow triangular tip** and verbosely described as a **“Is a”** kind of relationship.  **Aggregation/Composition:** A UML term and **programming concept** that represents the **dependence** of one class with another by means of **acquaintance**. An **Aggregation** relationship suggests a **dissociative** nature where the permanence of child class **does not rely on the** existence of the parent class, whereas a **Composition** relationship suggests a **dependent** nature where the permanence of the child class **does rely** on the existence of the parent class. Represented in UML as an **arrow with either a hollow or filled diamond tip** and is verbosely described as a **“Has a”** kind of relationship.  **Sequence Diagram:** A form of UML diagram that visually represents a program’s **behavioural (logical) architecture**, a Sequence Diagram is infinitely useful when trying to **make sense of how data/parameters are manipulated** during an ordinary sequence of events. The **physical components** of a system are scrawled horizontally along the top (in most cases) and the **logical flow of events** is visually represented by **a line passing through walls** that represent the system’s components. Mostly describes **behavioural attributes** and **input/output parameters**. Visually resembles [**畫鬼腳**](https://en.wikipedia.org/wiki/Ghost_Leg)**.**  **State Flow Chart:** A form of UML diagram that **visually represents** the **conditional attributes of behaviours** and the **association with Classes**. Useful for keeping track of how **instantiated Classes** are meant to be **shaped** during its lifetime. Features a **starting node**, a line which represents the lifetime of an object and can have **multiple conclusions**.  Visually represented by a **line, a bunch of circles, boxes and boxes in boxes with semi-circles in boxes**. |
| 1. **Description of SDLC in Agile / Waterfall Model** | **Agile**: A **modern method** of software development that involves a **flexible** working schedule and working through **iterations** instead of the completion of a **project** in one. This remains as a **popular approach** to **software development** as the kind of **people** who would willingly want to become **software developers** also heavily lack **commitment** to anything.  **Waterfall**: The more **traditional** and **structured approach** to software development where the entire process is clearly **defined in segments** which is planned to be performed in **sequence**. Waterfall offers a more **robust** approach to working on a project but **lacks the flexibility** that most software developers crave.  **Scrum**: A **newer** spin on the Agile methodology, Scrum is **very much Agile** but with **regular structured meetings** for more **regular updates** on the **project’s health**. Involves **frequent reiterations** of the project through **sprint periods** where a **backlog** of incomplete and/or outstanding work is **maintained** throughout the **project’s lifetime**.  **Analysis**: The first step in the **Software Development Life Cycle**(SDLC) and the **initiating phase** of a software development project, the analysis phase involves **a lot of communication** between the **primary stakeholders** and the **project team** as well as **intensive studies** on the **business case** being presented to assess the **feasibility** of the project. Happens **before anything intensive** goes underway to give the project team **enough time** to either **renegotiate** or **abandon negotiations** as to **minimize losses**.  **Planning**: One of the more **intensive** phases in Software Development (it doesn’t involve much coding either) and the next step in the SDLC. Planning encompasses the **documentation** in its entirety and planning of the **development/testing/deployment process** in order to make the actual development/testing/deployment much more **efficient** when it finally gets to happen. Planning this **early on in a project’s lifetime** also helps to **highlight** any **deficiencies** that the prior research may have which gives the project team **ample time** to do further study if needed.  **Development**: The most **active** phase in software development and the third step in the SDLC, development is the part where you **actually get to feel like you’re doing work**. It involves the **feature development** of software and any **fixing** that may come along with it **before the deployment** happens.  **Testing**: The **other half** of the Development phase and a step that is generally **not considered to be its own entity**, Testing involves the **testing of the developing product** to ensure that **mission critical functionality** remains **functional**. Either happens **concurrently with development** or in one go at the **end of the Development phase**.  **Deployment**: The phase which most would consider to be **the conclusion of the SDLC** or one of its iterations, Deployment involves the **installation** of the (hopefully) **completed product** onto the **client’s machines**. Usually comes with its own set of **headaches** despite the step being pretty straightforward. Good luck.  **Maintenance**: Post-deployment **care** or the **joys** of getting called into work at early AM hours because **something unexpected** happened to the thing. This is the phase that people who like **Sudoku puzzles** really enjoy and **should really be limited** when it comes to negotiating the **SLA** in your contract/charter. Good services means **repeat customers** though.  **Iterations/Sprints**: Unique to **Agile and all its plentiful clones**, an Iteration or Sprint is a **fancy** way of describing a **regular working week** for a software development. Usually structured so the **start** of an Iteration/**Sprint involves a briefing** to keep team members more **up-to-date** more **frequently**. |
| 1. **Description of OOA** | **Object Oriented:** A **philosophy** in programming that involves the **structure** of data in organized and workable **chunks** called “objects”, Object Oriented Programming offered a **larger scale** solution to software **architecture** that required **less effort** to type out. Now a **widely adopted standard**, nobody **remembers** or has heard about any **alternatives** (functional programming) because the **people who pay** your salary benefit more from OOP’s **ease** of programming **large scale solutions**.  **Object:** A **conceptual form** of data in the **OOP philosophy**, an Object is a **modifiable dataset** surrounded in **pointers** (bloat) to make the data as **interactive** as possible. Theoretically, the OO approach does fall inline with what **Psychologists** understand about **human psyche** so it may be more **natural** for people to think in terms of **organized datasets** like Objects, but most people just **program functionally** in the end anyway.  **Encapsulation:** The **OO concept** that an information to be **manipulated** in a behavior’s **iteration** is only to be **contained** within itself as to **avoid the corruption** and **mismanagement of data** residing in a **scope** larger than itself. Can either be set using **locality** (encapsulated in methods) or by setting an Object/variable’s **property** to **“private”** in most **OOP frameworks**.  **Iteration:** A fancier name for a **method**, Iterations contain the **actiony parts** of software. Typically defined by **curly braces**.  **Modularity:** An **OO concept** that dictates that the **code** made using in any **OO framework** should be **recycled throughout the entire program** to **reduce the workload** needed to create the software (the original intent of OOP). A good example is **writing a public method** to reuse elsewhere or creating **a class** that is solely used to represent **one object** and all the necessary functionality **unique** to said class.  **Separation of Concerns:** A **programming philosophy** where **one** **feature** being made should be designed to handle only **one function** and that **no two features should have overlapping functions** in order to **reduce redundant code**. Often requires more **planning** than what most people would care for but has been made **more simplistic** with the concept of **compartmentalized development patterns** like **MVC**(Model View Controller) or **MVVM**(Model View Viewmodel).  **MVC:** A **design pattern** in programming, MVC projects **separate** the **programmable actions** into three different **categories**. Models represent entities or **objects to be manipulated**, Views represent the **front end/interface/GUI** of the software and Controllers represent the **body of logic** behind the software’s **functionality**. Great for **rapid development** of small scale projects **without much needed** in terms of functionality. |
| 1. **Description of software tools** | **IDE**: Stands for “**Integrated Development Environment**”, an IDE is the **technical term** for a **code editor**. What separates IDEs from **plain text editors** is the **added functionality** that benefits **software developers** (for most part) and can also feature **automation of common tasks** like **refactoring** or **deployment**. Good examples of IDEs are **Visual Studio** and **Android Studio**.  **Prototyping tools**: Tools that may not strictly be for **software development** but can be **repurposed** to facilitate **rapid prototyping** of proposed **software products**. There are **purpose-built prototyping tools** that exist, but anything with **graphical functionality** can be used to **draw a prototype** and the sky’s the limit.  **Version control tools**: Software development tools that **automate** the burden of **retaining iterations of work** done in case of those **uh-oh moments**. Version control allows for greater **project redundancy**, a **failsafe** in case the project **goes astray** at any point and it also **provides** the project team the option of **repurposing** any given **iteration** of their product for the future.  **Frameworks**: A general purpose term for **a suite of software** that specialises in **performing a task** or providing the necessary **functionality** for performing said task. For example, Microsoft’s **.NET framework**.  **Project planners/diaries**: Not necessarily limited to software development once again but nonetheless a **useful tool** for managing the **progress** of any given **project** as well as providing a means of **planning** **assistance** for the future and a **chronicle** of events to **learn from**.  **Body of Knowledge**: A general purpose term for a **repository** of relevant **information**, a Body of Knowledge is the **source** of most of your **answers** to any **questions you may have** (or should be the source in most cases). A Body of Knowledge takes on **many forms** and there is **no limit** as to how one may **expand on their knowledge repository**. A **good** resource that is available digitally and is **free** in most instances is **Lynda**, a repository for **tutorials** of all kind by **highly talented educators** like **Kym Bond**. |