Jacob Taggart
Week 1 Day 3 Research HW

Day 3 Research Homework

1. Define the following design principles: Singleton, Factory, Builder, Facade, Prototype.

   A: The Singleton design principle is a principle that allows a class to only have one instance of itself created. The instance may be altered, but in a Singleton case the instance itself is the only instance capable of being created whether the instance gets changed or not.

   The Factory design principle is a principle that when used creates a default object. All of the functional details (nuts and bolts) are hidden from the end user of the program. Often times the factory method is used to create instances based on a user's input.

   The builder design principle when in use allows for a customizable object creation based on user input. In practice, the actual builder class will generally be an abstract class with children classes to allow for more customizability and a wider range of variables that can be accounted for - both of which factor into the complexity and specific makeup of an object being built.

   Much like the builder design principle, the facade principle is very focused on hiding the complexity and details of an environment from the end user. However, the difference here is that the builder principle is concerned with the creation of instances. Facades, on the other hand, are concerned with hiding the details of various options the client has. For example, a map of an amusement park may tell you how to get to rides and may even provide some details of the ride, but until you visit that specific ride you will never know exactly what the ride appears to be like. Generally this is carried out via interfaces which hide the exactness of the nature of each potential facade even if there is a general idea of what it may be about. It's all about hiding the complexity of a system from the client and making their actions simple to decide from.

   The prototype principle when used is much like the previous two in that the idea is to hide the inner-workings of the system from the client while allowing for greater simplicity on the client side. Since creating a completely new object is resource heavy and can eat up time and power, the practice of coding so that the system detects the object wanting to be constructed is already in existence somewhere else. Thus, instead of creating the object from scratch which can be costly to resources and power, the system will just copy the existing object and return the new instance this way. This principle not only hides the nuts and bolts from the user, but also provides a more efficient experience.

2. What is the differences in ART and Dalvik?

   A: The biggest and most important difference between ART and Dalvik is that ART compiles ahead of runtime (hence, A.R.T.) while Dalvik would compile on a just-in-time manner - meaning that Dalvik would compile things exactly at runtime or when necessary. ART is the new way of compiling, while Dalvik is old.

3. What is the android manifest used for?

   A: The Android manifest contains sensitive information regarding the true nuts and bolts of whatever package is being worked on/used. More specifically, the information inside the manifest contains crucial information about all the basic components of Android including the big four (i.e. activities, services, broadcast receivers, and content providers). On top of this, the manifest has a duty to the package and to itself to disallow any end user or non-developer from accessing or modifying these basic components.

4. Define the difference in Runtime and Compile Time.

   A: Compile time is when the code written is ready to be carried out while runtime is when the program is actually being executed. Thus, an important detail here is that with Android, certain aspects of the code can be compiled ahead of time with the use of ART while runtime is always fixed to when the program is ran. Often times certain compiling actions can't be carried out until runtime, but the idea is to make as much be compiled ahead of runtime without losing any functionality or simplicity.

5. How does each of the following units of measure for view work: sp, dp, px, pt, in, mm?

   A: The unit "sp" stands for "scale-independent pixels". Generally, this is mainly used for font sizes. The only thing that scale-independent pixels change for based on the form factor is depending on the screen density or a user's own choice. The unit is more abstract than most others as it only bases changes on the device's resolution or a user specifically requesting a change.

   The unit "dp" is similar to "sp" and stands for "density-independent" pixels. Like "sp", "dp" sizes are directly related to screen density. Relative to a 160dpi screen, one "dp" is one pixel. “Dp” tends to be consistent with “sp” in terms of resizing based on screen density. The difference is that “dp” is independent of user preference.

Technically speaking, “px” stands for pixels and merely means that it is the smallest area of brightness possible on a screen. A pixel is constant on any single screen which means that depending on the screen size, the screen can hold a relatively small amount of pixel units or - on something like a TV where the screen size is large - a large amount of pixels.

A “pt” is a constantly defined unit which means “point” and is exactly 1/72 of an inch. Thus, to find out how many points a screen can hold, one would multiply 1/72 by the screen size in inches.

“In” stands for “inches”, of course. An inch is a constantly defined unit of measurement roughly equal to 2.54cm. Thus, the amount of inches a screen can hold is directly determined by the size of the screen.

Like the previous two units, a “mm” is a constantly defined unit which stands for “millimeter”. Also like the previous two, the amount of millimeters on a screen is directly determined by the screen size. All three of these constantly defined units can be used interchangeably, optimally using the unit best suited to represent the size of the screen (e.g. not using mm for a massive TV screen, but instead using inches).

6. Describe what each section of the Android Platform arch. Details.

A: Firstly, all sections included in the Android Platform stack include (from bottom to top) the Linux kernel, the HAL (hardware abstraction layer), the native C/C++ libraries alongside the Android runtime platform, the Java API framework, and finally at the top is system applications.

The Linux kernel only provides the base level operations such as power management, process management, memory management, multitasking capabilities, and hardware device drivers. This layer provides abstraction between itself and all the other layers of the stack, most directly with the hardware abstraction layer. It’s important to note that ONLY the Linux kernel is being used in the Android Platform. Aside from the kernel, nothing else is Linux-based in the stack. ART completely relies on the kernel for its most basic functions.

Next, the hardware abstraction layer is the layer which provides general interfaces for specific types of hardware to be used. There are various modules in this layer that define different types of hardware. This abstraction layer allows the above platforms such as the API frameworks to be able recognize certain hardwares immediately based on the HAL. Thus, instead of installing hardware from scratch, the system can recognize it and assimilate the hardware based on its abstraction identifiers.

Android Runtime (ART) is an improved version of Dalvik runtime in that its platform is concerned with compiling as much information as possible before executing operations (running the program). In addition to this, ART is also highly concerned with getting rid of unnecessary information and allocating memory in the most efficient way possible. By using DEX (Dalvik executable files) - a bytecode format made for Android which is used to use memory as efficiently as possible. Thus, this layer is mostly concerned with providing as seamless and efficient of an experience as possible when runtime is initiated.

The native C/C++ libraries are alongside the ART in the Android Platform stack. The reason this specific layer exists is because certain parts of the stack need more base-level programming from more basic languages than the higher-level ones like Java and Kotlin. Without these libraries, there would be plenty of functionality which are generally felt as givens which would not be existent. In fact, without the native libraries in C and C++, HAL would not be used to its designed purpose and neither would ART. This layer is all about providing various functions as built-in features for the Java API Framework. Essentially, the more complex and base-level programming in this layer allows the higher language use that Android is programmed with. Without this layer, the whole stack would collapse as is the same with all other layers.

The Java API Framework layer is one that focuses on simplifying and adding another layer of abstraction to the native C and C++ libraries below it as well as simplifying and abstracting the ART layer as well. By abstracting, this is to say that it’s taking these lower level layers and simplifying as well as organizing the functionalities provided below in the stack. For example, a Java API Framework layer provides the capabilities of building from a largely diverse list of User Interface (UI) features like lists, grids, buttons, text formatting, etc. This layer also allows for quicker and more readily assimilated resources as it provides a resource manager tool that the user and even developer don’t necessarily need to know the ins and outs of in order to use or make an app access resources in a readily available and efficient manner. This layer allows for streamlined notification management, activity management (lifecycle of apps, services being ran or not being ran), and allows content providers to access and transmit data to other apps. For example, content providers - due to this layer - can allow an email app to access contacts via a contacts application or from other applications such as social media apps.

Finally, the apps layer is the set of core applications which Android OS's automatically provide to the device user. These generally include apps for text messaging, calls, contacts, internet browsing, settings, etc. When you open a new phone and begin using it on an Android OS, this layer provides anything that is immediately and readily available in terms of applications already on the UI which work efficiently due to the lower layers.

7. What is reflection in JAVA?

A: In java, reflection allows for the program to examine and modify the way applications are acting at runtime. The major upside to using reflection is that certain external classes and methods may be invoked at runtime that otherwise would not be available. This is done when the system recognizes when an abstract object may be extensible and therefore will be instantiated in order to provide certain otherwise inaccessible features. Reflection is an extremely powerful, but advanced tool that can result in some nightmare problems if used incorrectly. For example, because reflection can access private variables and methods which can cause unexpected behavioral and code changes in the program if the code is not written in a very efficient and organized manner. Also, it's important to keep the code updated to the correct and up-to-date platform stack (using the correct code and abstractions for a specific API for example). In addition, reflection requires a security-relaxed environment as it requires a runtime permission which many security managers do not allow to be used. Thus, depending on the security restrictions reflection may not be a viable option. Because reflection allows access to private fields/activities, that means the abstraction between classes is punctured and can cause major behavioral changes depending on the situation.

8. How does gradle work behind the scenes?

A: Gradle goes through 3 lifecycle stents during the build. These include the initialization phase, the configuration phase, and the execution phase.Via the use of settings.gradle, the gradle attempts to identify the amount of projects being built. The more projects to build, the more processing, power, and time it will eat up for the gradle to build. The reason settings.gradle can recognize the number of projects is via the amount of module folders included in the project (i.e. "app" folder is one that gradle considers a project...thus any module at the same place in the file hierarchy will be recognized as a project itself to gradle). Based on this, the gradle will build a file that organizes the API to reflect the project(s) being initialized.

The next phase is the configuration phase. This is the phase in which the gradle takes the organized API file(s) it created for the project(s) and configures the build script based off that API file. For multiple projects being built at once, each project gets its own build script. The build script is then executed based on the previous phase and initialization file(s). One optimization in the configuration phase is that the gradle can recognize irrelevant or redundant code to ignore and will indeed ignore if it's determined to be either of those qualifiers. Based on the previous configuration steps and the reading of the build script, the gradle analyzes the different tasks in the build script and creates a mathematical algorithm that streamlines each task object in which the execution needs to streamline. This algorithm is known as a Directed Acyclic Graph (DAG) which in effect streamlines the build tasks by containing no loops, which is to say that there are no redundant operations that will be carried out in the execution.

During the final phase, the execution phase, the gradle refers to the previously made DAG to decide which tasks need to be carried out. Depending on their dependency order in the DAG, the tasks are carried out in order of what needs to be done first to allow the next task to be possibly executed. The compilation of source code is all carried out during this phase in addition to the actual build. Everything from generating classes, optimizing the build directory files, making archives available for use, and file archiving is all done in this phase.