LE/EECS 4443 - Mobile User Interfaces
Winter 2023
Recitation Set (Sample Version)

S.Toyonaga

April 19, 2023

Contents

1 Layouts, UI Building Blocks, Testing

1.1 Layouts

1.1.1 Introduction

- A layout defines the **visual structure** for a user interface, such as the UI for an activity or app widget.
- We can define a layout in the XML vocabulary or through Java code. Do note that the XML method is preferred as it has a better separation of presentation from the application code.
- A layout in XML serves as the **root element**.
- **Elements** are denoted by $\langle element = \cdots \rangle$
- **Attributes** of an element are nested within the element. They typically have the form of $android : property = \cdots$ or $app : property = \cdots$
- We must set the layout (load the XML resource) in our activity by calling $setContentView(R.layout.\cdots)$ in the $onCreate(\cdots)$ method.

1.1.2 Introduction to the Identifier Attribute

- android:id="@+id/α"
  - @ $\rightarrow$ Symbolizes that the string is expanded as an XML resource.
  - + $\rightarrow$ Symbolizes that we must create and add this resource to the R.java class.
  - **Note:** R.java is created automatically. It connects the XML to the Java code. Lastly, it is generated during the build.
- We use IDs from R.java with $findViewById(R.id.\alpha)$ to assign Views to attributes in our Source Activity classes.

1.1.3 LayoutParams

- Attributes take the form: $android : layout\_? = \cdots$
- Layout Parameters are used by the Views to tell their parents how the want to be laid out.
- The base LayoutParams class just describes how big the view wants to be for both width and height.
- Settings:

- $FILL\_PARENT$: The view wants to be as big as its parent (minus padding)
- $WRAP\_CONTENT$: The view wants to be just big enough to enclose its content (plus padding).

1.1.4 Common Layouts

1.1.5 LinearLayout

- Arranges other views either:
  1. Horizontally in a single column
  2. Vertically in a single row
- We use *android* : *orientation* to define the prior behaviour.
- *android* : *gravity* is used to control how linear layouts align all the views that it contains. This will affect the horizontal and vertical alignment of all child views within a single row or column.
- Weights:
  - You can set *android* : *layout_weight* on individual child views to specify how LinearLayout divides the remaining space amongst the views it contains. We call this the, "importance"
  - All views are by default assigned a weight of 0.
  - A larger weight allows the view to fill any remaining space in the parent view. Views that have custom weights obtain any remaining space in the ViewGroup that is assigned to children in the proportion of their declared weight.
  - Examples:
    1. If there are three text fields and two of them declare a weight of 1, while the other is given no weight, the third text field without weight doesn't grow. Instead, this third text field occupies only the area required by its content. The other two text fields expand equally to fill the space remaining after all three fields are measured.
    2. If there are three text fields and two of them declare a weight of 1, while the third field is then given a weight of 2, then it's now declared more important than both the others, so it gets half the total remaining space, while the first two share the rest equally.

1.1.6 RelativeLayout

- A ViewGroup that displays child views in relative positions.
- Relative to Sibling Elements:
  - $android : layout\_above$
  - $android : layout\_below$
  - $android : layout\_toRightOf$
  - $android : layout\_toLeftOf$
- Relative to Parent:
  - $android : layout\_alignParentTop$
  - $android : layout\_alignParentBottom$
  - $android : layout\_alignParentRight$
  - $android : layout\_alignParentLeft$
  - $android : layout\_centerVertical$
  - $android : layout\_centerHorizontal$

1.1.7 Preface: Introduction to Adapters

- We use an Adapter object to bridge data between an AdapterView and the underlying data for that view.
- Adapters will:
  1. Provide access to data items
  2. Make a View for each item in the dataset
- We only use adapters on a layout that subclasses AdapterView.

1.1.8 Array Adapter

- Used if the data source is an array
- Creates a View for each array item by calling *toString*() on each item and placing the contents in a TextView. If we want something besides a TextView, we must extend $ArrayAdapter$ and override $getView()$ to return the desired object.

1.1.9 AdapterView: ListView

- Displays a vertically scrollable collection of Views, where each view is positioned immediately below the previous View in the list.
- Requests Views on demand from a ListAdapter as needed, such as to display new Views as the user scrolls up or down.
- **Use:** $listview.setAdapter(myAdapter)$; **as such.**

1.1.10 AdapterView: GridView

- Shows items in a two-dimensional scrolling grid. All you have to do is specify the dimensions (android:numColumns) and the underlying code handles any GUI scaling.
- Uses a ListAdapter.

1.2 UI Elements

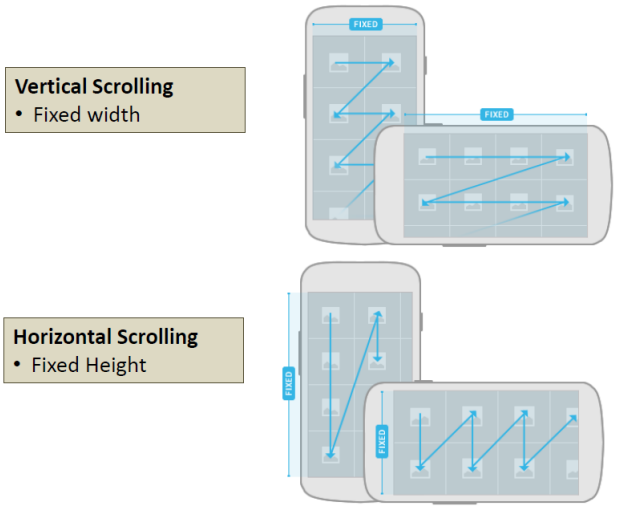
1.2.1 Tabs

- $Fixed \rightarrow$ Tabs are equal-width based on the widest label. All items are displayed concurrently. Each tab uses a Fragment.
- $Scrollable \rightarrow$ Suitable when there are many items. Every tab produces a new View via TabLayout and Fragments.

1.2.2 Lists

- A section divider organizes content into sections and facilitates scanning.
- A line item is nested within a section divider. It has many arrangements such as single or multi-line, custom icons, checkboxes, etc.,

1.2.3 Grid Lists (2D)



1.2.4 Scrolling

- Indicator → Only visible when scrolling
- Index Scrolling → Indicator is persistent. Dragging the indicator causes a pop-up similar to a dictionary tag.

1.2.5 Spinners (Combo Boxes)

- A quick way to select a value from a set
- Good for space management and when you don't need all of the options present on the screen.
- We can populate a Spinner by using a string-array and an array adapter.

1.2.6 Button

- Text Alone → Avoids ambiguity. Background is generally unnecessary. It's useful when space is limited and users are experts or will figure it out eventually.
- Image Alone → Works best when the symbol is well understood. Backgrounds are not accustomed because users are used to interacting with objects.

- Image + Text → Most appropriate when the two complement each other.

1.2.7 TextFields

- Allows users to edit text into an app.
- Properties define what is allowed to be entered and how.

1.2.8 Seek Bars / Sliders

- Used for selecting a continuous or discrete range by moving a slider. It's useful for representing an intensity of some kind (i.e., Sound)

1.2.9 Progress (Bar) Indicator

- Range should be known such that it never moves backwards. It should always be between 0 - 100.

1.2.10 Activity Indicator

- For situations where the length of operation is indeterminate.
- Can be linear or circular in representation

1.2.11 Switches

- Checkboxes are used for multi-selection
- Radio buttons ($\in$ *RadioGroup*) are used for singular selection with every option present. If the latter is not a requirement, consider a spinner.
- On/Off uses a binary selection of states.

1.2.12 Dialogs

1.2.13 Information Dialog

- Prompts the user for decisions or additional information required by the app to continue a task.
- i.e., Cancel/Ok, Adjust Settings, Enter Text
    1. Left Button → Dismissive
    2. Right Button → Affirmative

1.2.14 Alerts

- Informs the user about a situation that requires their confirmation or acknowledgement before proceeding
- Differs in appearance based on severity
- Common: $Report, Ok, Cancel, Erase$

1.2.15 Popups

- Requires a single selection form the user
- No explicit buttons.
- Selection advances the workflow. Touching outside of the popup dismisses it.

1.2.16 Feedback

- Simply used to provide feedback
- One button to confirm receipt of the message

1.2.17 Toast

- Lightweight popup
- Automatically disappears (fades) after some time

1.2.18 Pickers

- Nuanced way to select a value from a set by tapping, swiping, or through the keyboard.
- Requires a lot of steps if the range is very high.

1.3 Testing

- A formal process which is performed by a specialized testing team. They are planned in advance to preserve quality assurance and serve as an agreement between the client and developer.
- A software unit, multiple integrated software units, or an entire software system is examined by viewing or executing code in a computer.
- Associated tests are performed according to approved testing processes for the approved test cases.
- Direct and Indirect Goals:

- Direct: Detects and reveals as many errors as possible. It brings the tested software to an acceptable level of quality.
- Direct: Keeps testing in an efficient and accurate manner within budget and time constraints.
- Indirect: Provides error data to be used for the prevention of future error. Do note that we cannot ensure that all errors are eliminated.

1.3.1 Big Bang

- The goal is to test the entire system once it is completed.

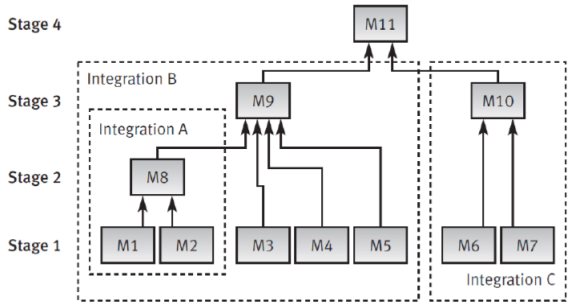
1.3.2 Increment

The general template follows:

1. Unit Testing: We test the components of the system.
2. Integration Testing: We test groups of system components according to their dependencies.
3. System Testing: We test the entire system, after all components are developed, tested, and integrated.

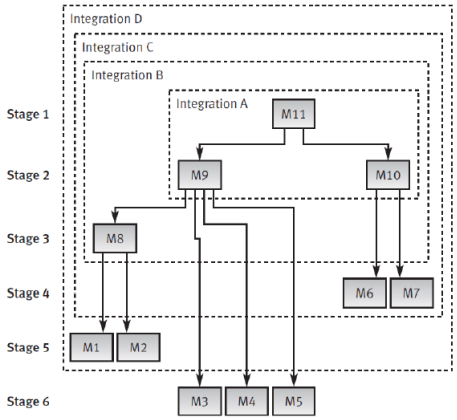
We can also perform testing modules in a Bread-First or Depth-First ordering.

1.3.3 Bottom-Up (Back-End)



- We use drivers to replace dependencies. They are used instead of high-level modules that are not yet available.
- Drivers pass test-data to low-level modules and accept computed data.
- Easy to perform, however, the results are very slow.

1.3.4 Top-Down (Front-End)



- We use stubs to replace low-level modules that are not yet available.
- They accept data from high-level modules and pass computed data.
- We get earlier results and detection of errors, but the results are difficult to analyze on account of many missing dependencies.

At the end of the day, the testing approach depends on the development process. Do note that a module has to be tested just after its implementation.

1.3.5 Comparison of Big Bang and Incremental

- Big Bang is appropriate for small and simple software. As the complexity grows, it becomes more difficult to detect and correct errors.
- Incremental is appropriate and preferred in most cases. The modules are small and simple, and thus, it's easier to detect and correct errors. One consequence is that it takes more operations to test it, thus, more resources are required.

1.4 Classification of Tests

- The classification of tests are dependent on the design of tests, the transparency of the system, and lastly, the requirements or quality criteria.

1.4.1 Black-Box Testing (Obfuscated)

- Tests execute code with certain input data and compare the results with the expected results. The internal instructions of the software are ignored.
- Allows for testing more quality criteria and aspects of the software such as reliability, performance, and security, documentation.
- One caveat is that there exists a possibility for a coincidental aggregation of errors that result in correct outputs.
- They are less costly than White-box testing.

1.4.2 White-Box Testing (Transparent)

- Requires knowledge of the entire system, including both code and documentation.
- Examines paths of internal calculation. As a result, it is limited in criteria analysis insofar as it focuses on functional accuracy, maintainability, and reusability. By implication, it also allows to test for data processing, calculation accuracy, and software qualification.
- Test Coverage:
  - Path Coverage: Focuses on measuring the percentage of tested paths to the total number of paths. This can be very expensive if we have multiple conditional instructions. It requires a lot of effort, time, and money to test them all.
  - Line Coverage: The goal becomes to test all of the codes of line at least once. Coverage is the percentage of tested lines to all code lines.
- Cyclomatic Complexity is a metric which estimates the complexity of white-box testing. It calculates the maximum set of independent paths to achieve full coverage through: $V(G) = E - N + 2$

  Advantages / Disadvantages:

- It allows for detecting more errors while also maintaining progress metrics. That is, it allows for completion of testing by revealing all lines that have not yet been analyzed.
- Very appropriate for testing critical systems whose accuracy and the elimination of errors is of the utmost importance.
- It requires an immense quantity of resources. Additionally, it does not allow for testing of all quality criteria.

1.4.3 Equivalence Classes

- Recall in Black-Box testing that it is not always necessary to test all values. This is because multiple inputs can produce the same type of output.

  The Rule of Three:

  1. Happy Path (Expected)
  2. Boundary Path (Range)
  3. Exceptional Path (Invalid / Exponential Inputs)

- We define representative values for the three paths. Then we define test cases that cover the equivalence classes (valid and invalid classes).
  - One test case per invalid class. If a test covers more than one invalid class, it is not possible to distinguish between the different sources of the error.
  - One test case per boundary class
  - α test cases per happy path

1.5 Implementing Tests

1.5.1 Automated Testing

- Advantages with respect to perceived costs, accuracy, and performance.
- Requires more effort for the design, planning, and preparation of tests
- Tools exist for automating the execution of tests, preparation of reports, regression tests, and lastly, the comparison of reports.
- Note: If regression tests have to be repeated more than 1-3 times, automated testing is more advantageous.

1.5.2 Alpha and Beta Testing

- Alpha: An incomplete version of the software is given to clients to test. And increased number of errors is expected. The client reports these errors and reveals as many bugs as possible.
- Beta: A completed version of the software which is untested is given to clients. Testing uncovers how the users will use the software in addition to deployment factors. The client will detect the errors and provide a report to the developer.

- Alpha and Beta testing have advantages in detecting unpredictable errors and reduced costs. However, both of them have non-systematic testing in addition to low quality reports. That is, replication is difficult.
- Alpha testing precedes beta testing.

1.5.3 UI Testing

UI Testing involves the verification of:

- The design of the UI according to specifications.
- The interaction between the user and UI

Generally speaking, we only do UI testing if you need to confirm visual interactions. If functionality can be tested without the user's input, but simply with test data, we rely on software testing.

In general, apps employ a combination of UI testing and Software Testing.

1.5.4 Types of UI Testing

1. Manual Testing
2. Record and Replay
3. Exploratory ($Period \rightarrow Goals$)
4. User Experience Testing(Alpha/Beta: Experience and Particular Needs / Patterns)
5. Scripted Testing (Pre and Post Conditions), Test-Driven Development)

1.5.5 Automated UI Testing with Espresso

- An automated testing framework which uses special classes through:
  - Matcher: Text, Attributes, etc.,
  - ViewAction: Clicking, Typing, Scrolling, etc.,
  - ViewAssertion: Content, State, Position
- We can also test on intents (Bundles) for states.
- "Record and Replay" $\rightarrow$ Try to follow the test case script and test only one action.
- Note: Espresso tries to capture your exact interaction with the application. However, certain details may not remain true for any subsequent runs. You have to modify your code to handle these cases.

1.5.6 Anatomy of a Test Class

1. $@LargeTest \rightarrow$ Test Class that executes multiple test cases.
2. $@RunWith \rightarrow$ Indicates the tool/library with which the tests will be executed.
3. $@Rule \rightarrow$ Used to launch the activity under test.
4. $publicvoid\alpha(\cdots)\{\cdots\}$ is the entire test interaction encapsulated within a said method.
5. $ViewInteraction \rightarrow$ The interaction within a single UI element.

1.5.7 Programmatically Tracing the UI

- The Layout Inspector (from the Tools Menu) is useful to access the UI elements programmatically when we need to fix the autogenerated code from the Espresso Recorder.

1.5.8 Idling Resources

- For tasks where the UI depends on loading data, waiting for a response/callback, we should avoid artifical delays. They can be device specific, and contaminate the performance profile of your app.
- **Solutions:**
  1. CountIdlingResource (Semaphore)
  2. UriIdlingResource: Similar to a semaphore, except that the counter needs to be zero for a specific period of time before the resource is considered idle.
  3. IdlingThreadPoolExecutor: A custom implementation of ThreadPoolExecutor that keeps track of the total number of running tasks within the created thread pools. It relies on CountingIdlingResource.
  4. IdlingScheduledThreadPool: Similar to item (3), however, it also keeps track of tasks that are scheduled for the future or are scheduled to execute periodically.

1.5.9 Custom Idling Resource Implementation

- We must register and unregister idling resources in the app.
- We use a registry of idling resources as a repository and an easy way to activate them.

- While you can code your idling resources within the app, it is better to define build variants and activate the resources only when testing.

```
public void isIdle() {
    // DON'T call callback.onTransitionToIdle() here!
}

public void backgroundWorkDone() {
    // Background work finished.
    callback.onTransitionToIdle() // Good. Tells Espresso that the app is idle.

    // Don't do any post-processing work beyond this point. Espresso now
    // considers your app to be idle and moves on to the next test action.
}
```

1.5.10 Espresso: Add-Ons

- Espresso-Intents: Invoke activity transitions and validate intents.
- Espresso Lists: Test adapters for lists and interact with RecyclerViews
- Espresso Web: Test hybrid applications that use the browser.
- Multiprocess, Accessibility, etc., $\cdots$

2 Intro to Profiling and Benchmarking

2.1 Performance

Performance is very important because it is directly tied to user experience when using the system. This is so much the case that we have developed standards.

2.1.1 Standard: Performance Efficiency (ISO 25010)

- This standard represents the performance relative to the amount of resources used under stated conditions.
- **Sub-Characteristics:**
    1. **Time Behaviour:** The degree to which the response, processing times, and throughput rates of a product or system, when performing its functions, meet requirements.
    2. **Resource Utilization:** The degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
    3. **Capacity:** The degree to which the maximum limits of a product or system parameter meet requirements.

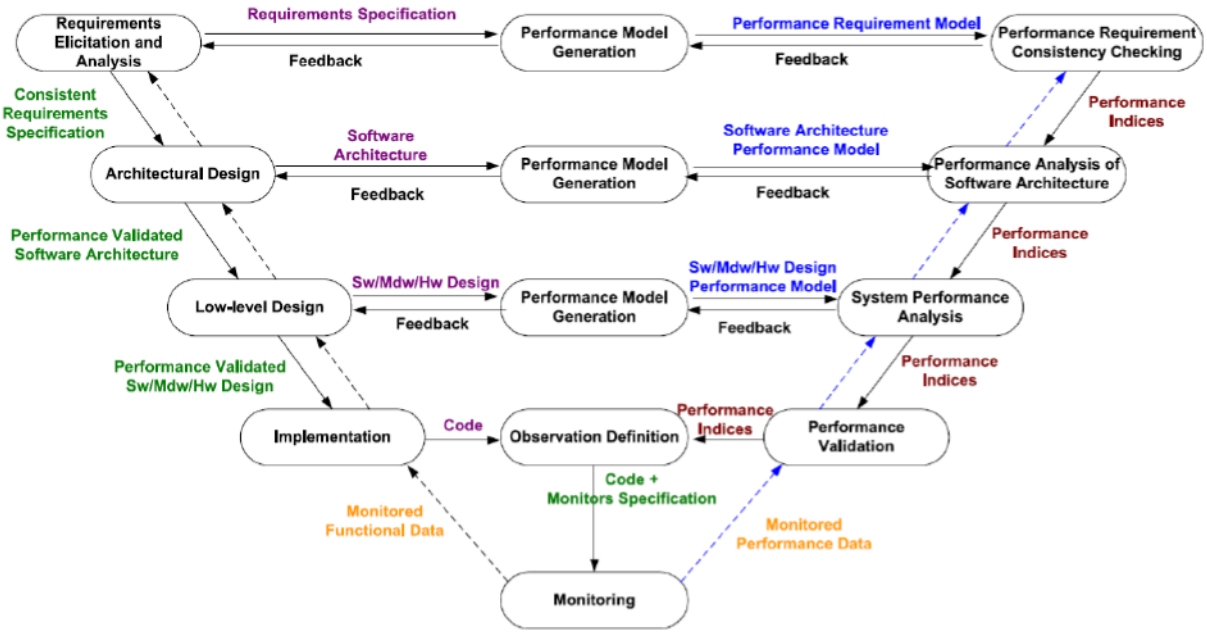
2.2 Software Performance

- Performance measures the **efficiency** of software with respect to constraints relative to **time** and **resource allocation.**
- Performance Metrics can be defined to support many different objectives. Some common examples can include but are not limited to:
    1. **Response Time:** Total time that a task takes to traverse a part of the system.
    2. **Throughput:** Number of requests or tasks completed per unit of time by a part of the system.
    3. **Utilization:** Percentage of time during which a part of the system is busy processing.
- **Note:** Per notation, there are unique indexes in the context of certain performance models.
- It is possible to consider the performance of the entire system: $(\Sigma Hardware + \Sigma Software)$, or just part of a system, like an Activity/Interaction.

2.2.1 Software Performance Engineering (SPE)

- Represents the sum of techniques, tools, and methodologies to ensure the conformity of non-functional requirements relative to performance according to specified indexes during the entire lifecycle of the software.
- **Non-functional Requirements (NFRs)** define system attributes such as security, reliability, performance, maintainability, scalability, and usability.
- Also known as a **systematic, quantitative approach** for the efficient development of software with the goal of conforming to NFR.
- The goals of SPE generally reduce to:
    1. Increasing revenue to ensure that the system processes all transactions on time.
    2. Eliminate failures that would require complete re-engineering, optimization, and additional and useless costs such as maintenance or to acquire more hardware due to the instantiation of performance problems.
    3. Identify bottlenecks by simulating a prototype.

2.2.2 Process During the Software Development Lifecycle (SDLC)



2.2.3 Requirements Phase

- Understand the business goals (requirements), documentation, and the available means to achieve them.
- Revise performance metrics of the version in production.
- Determine Non-Functional Requirements. Also take time to determine the goals of the system performance and the metrics to evaluate them.
- Determine the tools, resources, and infrastructure. This allows for the allocation of budget and time to install and train personnel.
- Confirm and resolve any potential conflicts between the consistency of the NFRs between themselves and with relation to the functional requirements.

2.2.4 Software Architecture and System Design Phase

- Evaluate the alternative designs and architectural styles with respect to performance indexes.
- Determine the capacity of the required infrastructure. Combining the architecture with the NFR can define the infrastructure requirements.

- Define the performance goals of the developers: The goals for a user are defined to be used in performance testing.

2.2.5 Implementation Phase

- Monitor the performance unit tests
- Develop workload models as follows:
  1. **Utilization:** How the users will use the system to achieve their goals. A time and size for the peak load and regular load must be asserted. **Infrastructure:** The load for infrastructure resources, like CPU, memory, network, etc.,
- Install and configure monitoring tools for the software / infrastructure.

2.2.6 Advantages of SPE

- $Clear\ set\ of\ NFR \rightarrow$ Success of the development
- Focus on system performance during all phases prevents late and expensive future modifications (fixes).
- Monitoring of performance in production maintains performance and the reliability of the system. It also allows for scaling of the capacity before it is surpassed.
- Avoids problems and focuses on development rather than constant resolution of problems.
- The client will receive the total value from the system due to delivering a functional and performant system.

2.2.7 Disadvantages of SPE

- Favouring budget and time-to-market constraints lessens the importance of performance in the SDLC.
- SPE Inefficiency $\rightarrow$ Knowledge-Gap between Developers and QA Experts:
  - $(Developers \wedge Features) \wedge (QA \wedge (Stability \wedge Availability))$

    This is mediated through an error budget. When the budget is depleted, the focus shifts towards new development.
  - Another problem involves the difficulty in matching the function to the non-functional requirements.

- Performance is perceived by the users. Developers are only aware of functionality while QA is only aware of performance.

  This is mediated through automation. It eliminates the need for qualified people to construct methods and performance models. It also reduces the time and effort to validate performance.

2.3 Software Performance Measurement

2.3.1 Concept

- Provide input data to the SPE models to verify and validate them.
- Evaluate conformance to performance goals
- Monitor performance during SDLC.
- Metrics:
  - Static: No need to execute the system.
  - Dynamic Metrics: Requires that the system is executed during the measurement. Performance metrics are often dynamic.
  - Internal: Special code for event detection and performance data logging is required in the system code.
  - Measurement software is independent from the measured software.
- Performance is easier to test with blackbox testing.

2.3.2 Goals

1. Understanding the system by measuring it. We can compare it to similar systems or previous versions.
2. Generating specification of models by providing simulated workloads and usage estimations.
3. Updating the models to improve accuracy and precision through improved measurements.
4. Verifying and validating models through a comparison of performance models to the actual observed performance. We also check conformity to the performance goals or to specific problems to fix.
5. Evaluate the performance of the software to monitor performance and identify improvements.

2.3.3 Process

- Understand the goal of measurement (Questions / Hypothesis)
- Identify the required data to answer the questions in addition to the tools and measurement techniques.
- Identify the control and experiment variables.
  - *Control* $\rightarrow$ Use to answer and compare against the execution which answers our hypothesis. Also; output variables.
  - *Experiment* $\rightarrow$ Ones which must be defined to set up the environment. Also; input variables.
- Define the test cases: workload, software components, and the environment of each test. This is critical for repeating the testing environment.
- Execute the tests and collect the data
- Analyze, interpret, and present the results.

2.3.4 Factors That Inhibit Measurement

1. System Perturbation (Observer Effect)
2. Capture Rate (Rate of Monitoring Time During Perturbation)
3. System Overload ($\Sigma Time$ of Perturbation)
4. Timing of Measurements (Internal System Clock, Sampling Rate)
5. Reproducible Results (System is not isolated, and other processes may affect the monitoring and the performance.)
6. Representative Time Intervals
7. Typical Behaviour Averages
8. Workload Generation

2.4 Performance Testing

2.4.1 Introduction

- All tests and methodologies for for measuring, verifying, and validating the performance of a system.
- A part of SPE, such that its goals include the demonstration of:
  - Conformity of the system to performance criteria
  - Comparing two systems to find the most efficient option.
  - Measuring and identifying components that cause performance problems.

2.4.2 Generated Questions

1. Scope of Testing: Components, Subsystems, Interfaces
2. Number of Expected Concurrent Users For Each Functionality (Regular versus Peak Usage)
3. Configuration of the System Architecture
4. Workload Mix for Each Functionality of Each Application
5. Workload Mix for Each Application of the System
6. Processing Time Requirements (Peak versus Regular)

2.4.3 Testing Types

1. **Load Testing:** Tests the performance of the system during the expected load.
2. **Stress Testing:** Tests the capacity limits of the system.
3. **Endurance Testing:** Tests the system under the expected load for a longer time.
4. **Spike Testing:** Tests the reaction of the system by suddenly increasing or reducing the generated load by a large number of users.
5. **Capacity Testing:** Tests the system to find the maximum capacity.
6. **Configuration Testing:** Tests the effect of various configurations or of the changes in configuration.

2.4.4 Testing Process

1. Decide if you will use internal or external resources to perform the tests.
2. Elicit the performance requirements from the users.
3. Develop the SPE plan.
4. Develop a detailed plan for the performance testing.
5. Choose the testing tools.
6. Specify the test data and the required effort.
7. Prepare the validation scenarios.
8. Configure the testing environment.
9. Execute the tests.
10. Analyze the results.

2.4.5 Testing: Common Mistakes and Traps

- Common Mistakes:
  - Applying Performance testing in the last step of development.
  - Applying more hardware to solve every performance problem.
  - Having the mindset that if it works now, it will always work. This is untrue as the software and architecture are constantly changing under updates.
  - Believing that one set of tests is enough
  - Believing that $Testing(\Sigma Components) = \Sigma(System)$
  - That load testing is sufficient.
- Common Traps:
  - Unrealistic Cache Interceptions. If data or file recovery is way too fast, we need to fix this by restarting the program and erasing the memory.
  - Using Sorted Test Data
  - Performance Variance Due to Shared Resources (We must consider the testing time in the results and use local resources.)

2.5 Profiling

2.5.1 Introduction

- Profiling is a **(low-level)** form of dynamic analysis that measures the complexity of the system in terms of space (Memory) and time, or the frequency and the duration of function calls.
- Profiling applications:
  - Optimization of the program
  - Efficient management of resources
  - Understanding the behaviour of a program **(processes, events, functions, threads, and at a micro level, bottlenecks.)**
  - Evaluating and comparing the performance of different architectures.
- Done during development to identify problems with performance in the form of bottlenecks.
- The two important components we will be studying is instrumentation and sampling.

2.5.2 Instrumentation

- The addition of code (probes) in the profiled program to collect performance data.
- We can add probes in multiple levels of the system:
  - Source Code (Manually or Automatically)
  - Compilation
  - Binary Code (Cleaner: Only Domain Code!)
- The motivation is to collect data accurately and infer the **locality** of the data. We also want to be able to control the **granularity (method, class, activity, layout, etc.,)** of the data and measurement process by activating or deactivating the probes.
- Note: While it is possible to collect data with external tools, the data is not detailed or granular enough.
- **Instrumentation Design:**
  1. Identify the events to measure. That is, the important events for each scenario, including the beginning and end of key functions.

     Android → onCreate, onResume, onRestart, onTransition, ...

2. Granularity level(s):
   - All Events → Costly
   - Selective Probe Activations in Code or for Some Software Components
   - Probe activations for calculating averages, variances, and distributions.
3. Dynamically selecting the data to be logged at run-time or through the instrumentation parameters to vary the metrics and the granularity (i.e., Application Levels).

2.5.3 Instrumentation Traps

1. Adding code in the beginning and the end of an operation to calculate the duration of its runtime → Adds overhead
2. The instrumentation does not calculate the overhead and subtract it from the execution time for more accurate results.
3. If the operation is too short or too fast, the overhead becomes significant and the profiler cannot accurately compare the time between short and long operations. This may cause false positives (bottlenecks) that do not exist.
4. Instrumentation is an intrusive process. It can lead to heisenbugs which are the observer effects.

2.5.4 Automated Profiling

- Automated profiling facilitates optimization and guarantees continuous integration and quality assurance. (Android Profiler / Espresso Testing)
- Reduces the cost of optimization by quickly identifying points which need optimization.
- Profiling tools are capable of calculating a large number of metrics and of producing detailed reports **(traces)**. Be sure that the method you choose is not intrusive.

2.5.5 Sampling

- Sampling does not affect the execution of the program.
- No code is added to the source code nor to the compiled code.

- The OS interrupts the CPU in frequent intervals and the profiler logs the instruction that is being executed. The profiler correlates the instruction with the corresponding point in the source code.
- We have to repeat the profiling with sampling multiple times to obtain statistical significance. This is because snapshots could be affected by other competing processes running in the said background.

2.5.6 Sampling versus Instrumentation

- Sampling is less accurate (No granularity, landing, or exact measurements) but more efficient than instrumentation.
- Sampling is an external process so it does not inhibit the performance of the software and does not add overhead.
- Since sampling captures snapshots of the CPU, it can lose information. For example, we don't know who called the instruction. Lastly, if the profiled operation is too short, the sampling will not capture it (No available stack trace). Even if we were to increase the sample rate, this would increase the number of CPU interrupts leading to the same problems that Instrumentation suffers from.
- If the profiled operations or system are slow enough, we prefer instrumentation because the added overhead is insignificant compared to the runtime of the operation.

2.6 Monitoring

2.6.1 Introduction

- Monitoring is the act of measuring the software during its execution on a **production environment** (after deployment)
- Key Difference:
  - Profiling (controlled) → Measuring at runtime during development
  - Monitoring (uncontrolled) → Measuring at runtime after deployment. There is no expectation about the said load.
- Gives you an (almost) complete picture of the system's performance as it is intended to be used. It can control the "health" of the system by causing alerts in cases where the capacity of the system is surpassed due to an increased load.
- Health → Adaptive or Corrective Actions at Runtime

2.6.2 Monitoring at Different Levels

1. Application
   - Android Processes: Throughput, Responsiveness, Actions, etc.,
2. Operating System
   - Resource Manager(s)
3. Host
   - Virtual Machines / Containers
4. Hardware
   - Device, Server (Energy, Temperature)

2.6.3 Monitoring: Challenges

- Different levels may imply different challenges...
  1. Hardware, Host
     - Multitenancy (when the host is used by multiple users/applications at the same time) may affect the **reliability of the measurements** for each independent application. (i.e., Cloud Servers)
  2. Operating System
     - Co-existing processes may compete with each other which effects the **reliability of the measurements**.
  3. Application
     - We can measure response time only from our perspective and not how it is perceived by the client/user.
     - There is a network delay that we cannot always control.
- As is the case with profiling, monitoring can be intrusive or external.
  - Live Monitoring ("Pull Method"): The monitoring module queries the system at real time. It is intrusive and can affect the measurement, but is also very precise. It is like instrumentation, but without code.
  - External Monitoring ("Push Method"): Uses sampling. The measurements are stored in a database at frequent intervals and the external monitoring module queries this database asynchronously instead.

2.7 Benchmarking

2.7.1 Introduction

- "A test, or set of tests, designed to compare the performance of one computer system against the performance of others"
- $Benchmark = Metrics + Workloads + Measurement$

  Measurement → Data, Metrics, Process, etc.,

2.7.2 Types of Benchmarks

1. Specification-Based
   - **Only a framework is given:** Describes functions, along with their input parameters and expected outputs that need to be realized and tested.
   - Implementation is left to the individual running the benchmark.
   - Focuses on the business problem, allowing us to test and compare different implementations of solutions for the same problem.
2. Kit-Based (Android Tools)
   - Provides an implementation that is required for the general benchmark execution.
   - Does not allow for alternative implementations, unless packaged with the benchmark.
   - Reduces the time and effort to design and execute the benchmark by providing a minimum configuration, load-and-go (execution, workload, configurations.)
3. Hybrid
   - Most of the benchmark is provided as a clip.
   - Allows for some alterations of the implementation by acting like a framework with frozen spots (kit) and hot spots (implementation alterations)
4. Synthetic Benchmarks (Path Control)
   - A simulated execution (i.e., Espresso) of the system
   - May miss unanticipated scenarios
   - Treats the functions or scenarios in isolation
5. Microbenchmarks (Assignment/Project)

- Focuses on a specific part/function of the system (I/O, processing unit, memory management unit)
- Allows to focus on a bottleneck and then quantify its impact. We use the profiler on the other hand to identify a bottleneck!

6. Kernel Benchmark
   - A small/core version of the application under tests that captures the basic functionality or main use case of the app.
   - Reduces the cost-to-coverage ratio, but may miss bottlenecks or fail to stress the resources.
7. Application Benchmarks
   - Uses a complete application as a representative of a larger class of applications.
   - Real workload and real behaviour
   - Usually a smaller version of an application with artificial or small data.

2.7.3 Benchmarking Strategies

1. Fixed-Work
   - Measures and compares two systems or versions of the same system in terms of their time behaviour.
   - It can evaluate the speed-up of an optimization.
2. Fixed-Time
   - Measures and compares two systems or versions of the same system mostly in terms of their resource behaviour.
   - It can evaluate the scalability of the system through throughput.
3. Variable-Work / Variable-Time
   - Tests the system freely under a continuous and variable workload for a longer period (Endurance / Load Testing)
   - Good approximation for more realistic scenarios.

2.7.4 What Makes a Good Benchmark?

1. Relevance (Behaviours of Interest to Users)
2. Reproducibility (Consistent Results with the Same Test Configuration)

3. Fairness (No Biasing of Results)
4. Verifiability (Accuracy)
5. Usability

2.7.5 Workload Generation

- Workloads are the components that trigger **behaviour** and **load** on the system under test.
- Workloads have two parts:
  - Executable Parts → Tasks, Requests, Interactions,...
  - Non-Executable Parts → Configuration (Repeated Execution, Emission Rates, Delays, Order, etc.,)
- Typical behaviours can cover:
  - Natural → Execution Traces of Similar Systems → Available Data Sets (We may have to configure existing ones to fit our said requirements.)
  - Synthetic → Simulated workloads that capture basic properties (expected distribution) or are based on abstract descriptions of the workload (not necessarily on the realistic use cases).

2.7.6 Types of Workloads

1. Batch: A single long-running executable work unit
2. Transactional: A series of small work units, usually repeated over a long time, given a start and end time. It causes a specific amount of load on the app.
3. Closed: Assumes a fixed number of users, each sending transactions to the application independently. The user waits for a response before sending another (Think-Time Property). The amount of workload generated also depends on the performance (speed) of the application under test.
4. Open: Users enter the system at time and exit once they have achieved their goal. The number of varying users can be replaced for simplicity with an arrival rate (work units per time unit) which could be stable or malleable.

For access to the remaining notes, please contact me over LinkedIn or GitHub.