MONASH University

Information Technology

FIT5183: Mobile and Distributed Computing Systems (MDCS)

Lecture 6
Android Design Fundamentals

Overview

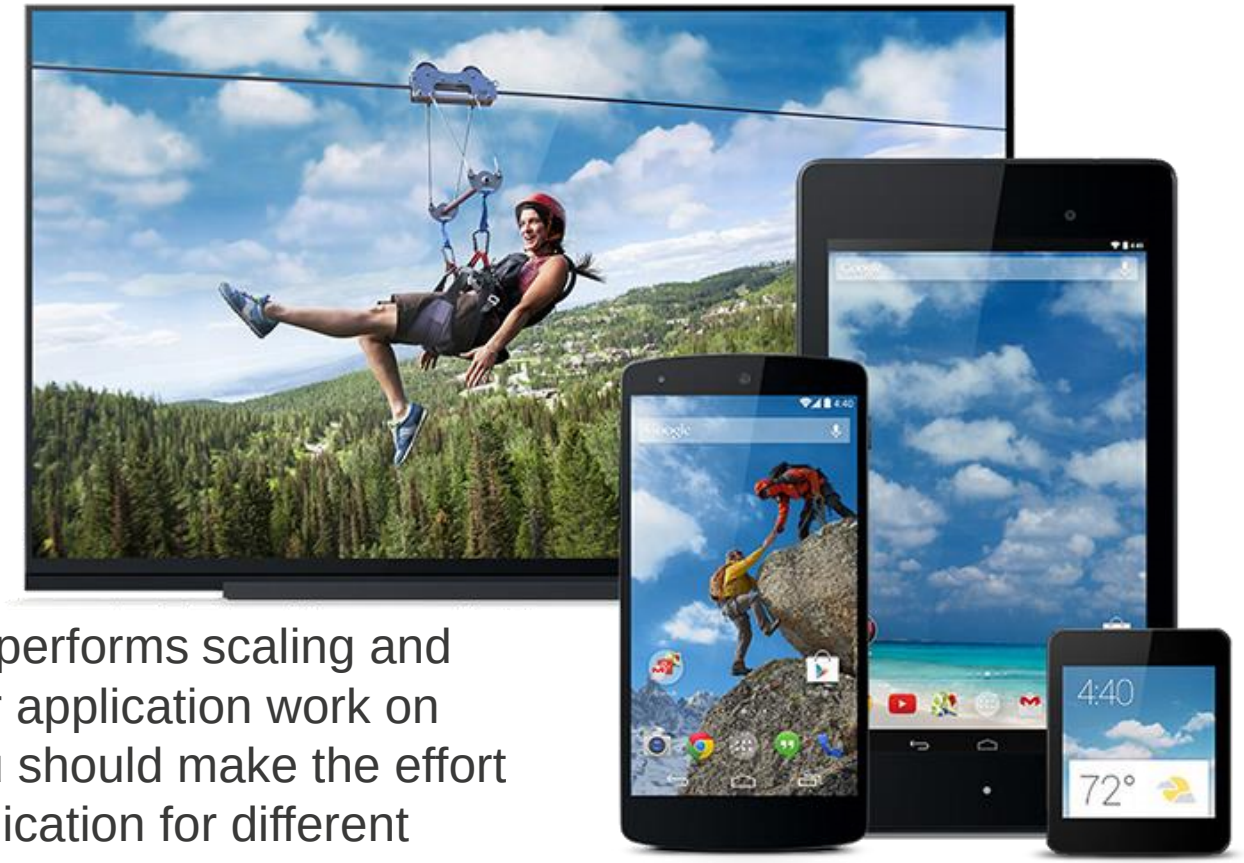
- Mobile interface design challenges
- Model-view-controller architecture and design pattern
- Android Layouts and Common Interface Controls
- Identifying Layouts
- Common interface design patterns

Mobile Interface Design Challenges

Designing for Mobile Interfaces

- Mobiles present many challenges when attempting to design an effective user interface for an application.
  - **Hard to present a huge amount of data** within a limited screen size on a mobile phone.
  - **Usability** is key, when applications look or become confusing to use, users will complain.
  - Should a tablet app use the exact same interface as a phone app? Could it be **optimised**?
- If you think of any app that has sold well on any mobile platform, chances are it had a really usable and visually appealing interface.
- In Android, we often need to create different views for various interface controls via layout files for generic interface components or custom view classes.

Supporting Multiple Screens



- Although the system performs scaling and resizing to make your application work on different screens, you should make the effort to optimize your application for different screen sizes and densities.
- You need to create an application that displays properly and provides an optimised user experience on all supported screen configurations, **using a single .apk file**.

Small Screens First!

Human Interface Guidelines (HIG)

- Provides developers with recommendations regarding how user interfaces and interactions should be setup on different platforms for an optimal user experience.
  - Goal is to help developers create applications which conform to platform interface design conventions to increase usability.
  - Having said that, these are recommendations and you can break away from them if you have a valid reason.
- The guidelines for the Android platform: http://developer.android.com/design/get-started/principles.html

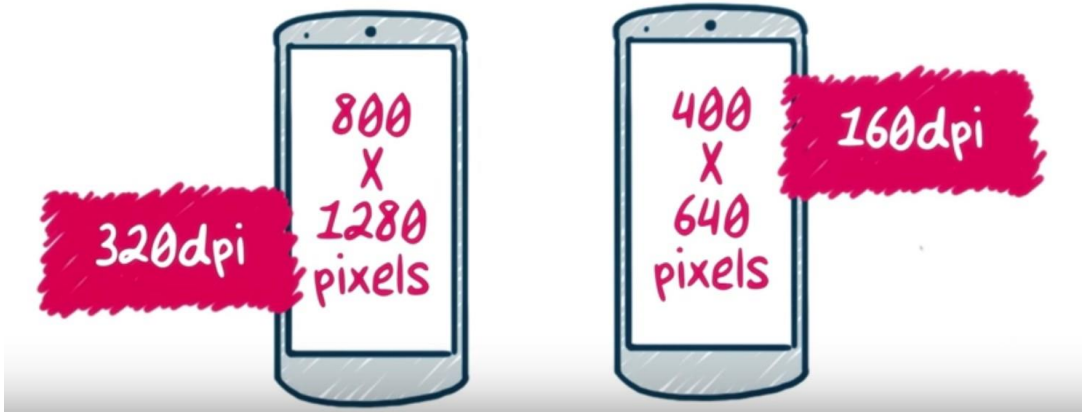
Android Best Practices
作者：Godfrey Nolan · David Truxall · Raghav

Terms and Concepts

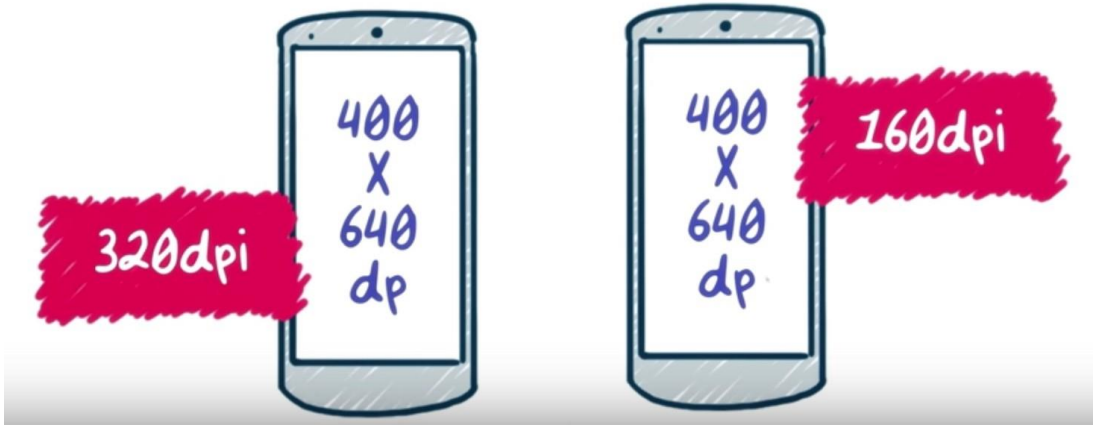
- Screen **size**: Actual physical size, measured as the screen's diagonal.
- Screen **density**: The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.
- **Orientation**: The orientation of the screen from the user's point of view.
  - This is either landscape or portrait, meaning that the screen's aspect ratio is either wide or tall, respectively. Be aware that not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.
- **Resolution**: The total number of physical pixels on a screen. When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

Density-independent pixel (dp)

- A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent

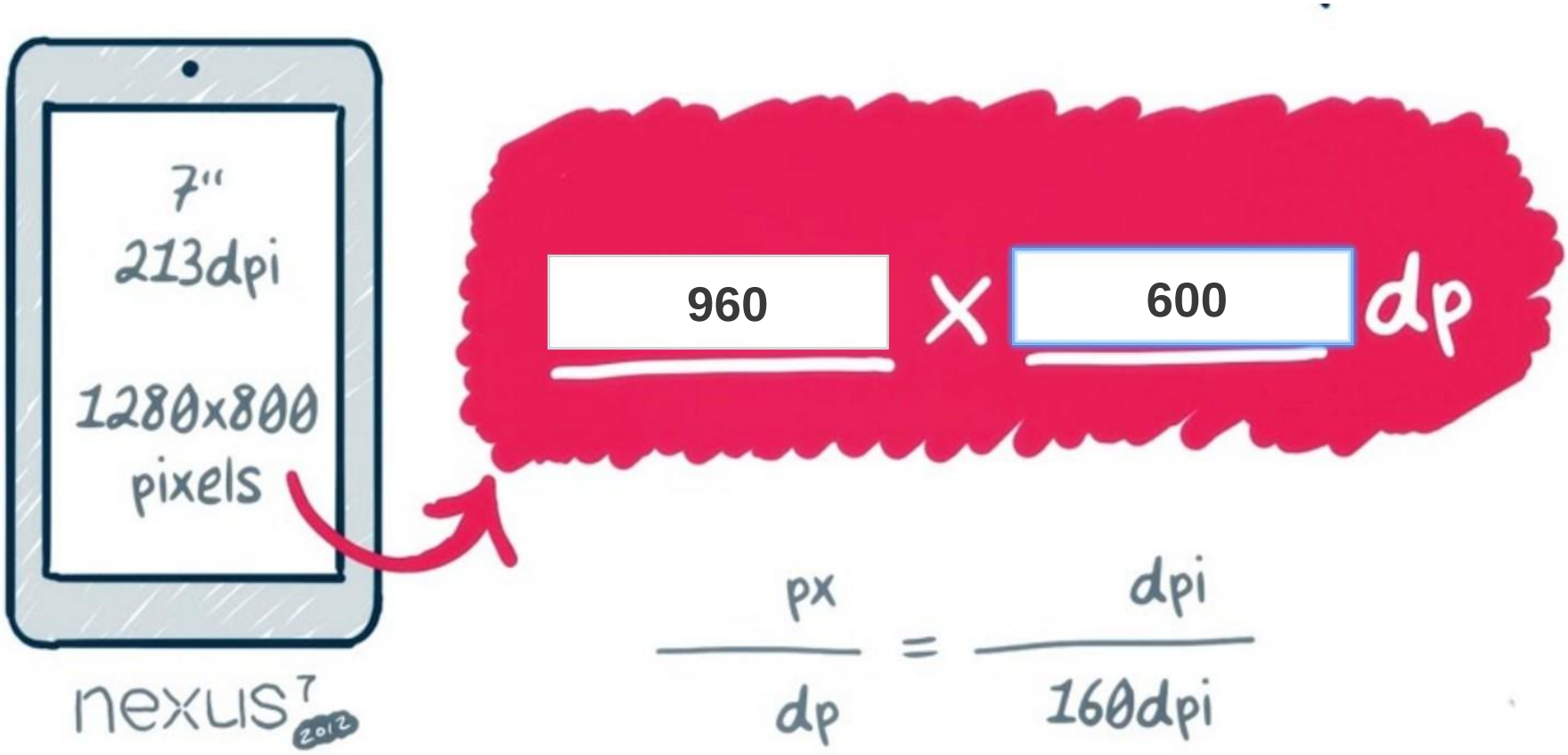


px = dp * (dpi / 160)



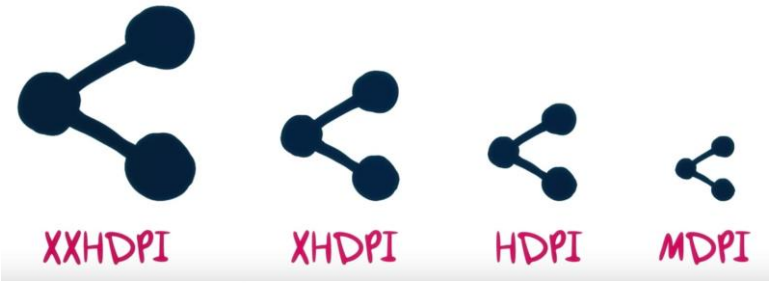
Source: www.udacity.com/course/

Calculating Dips (dp)



Source: Material Design for Android Developers @ www.udacity.com/course/

Density Buckets



LDPI	120dpi	.75x
MDPI	160dpi	1x
HDPI	240dpi	1.5x
XHDPI	320dpi	2x
XXHDPI	480dpi	3x
XXXHDPI	640dpi	4x

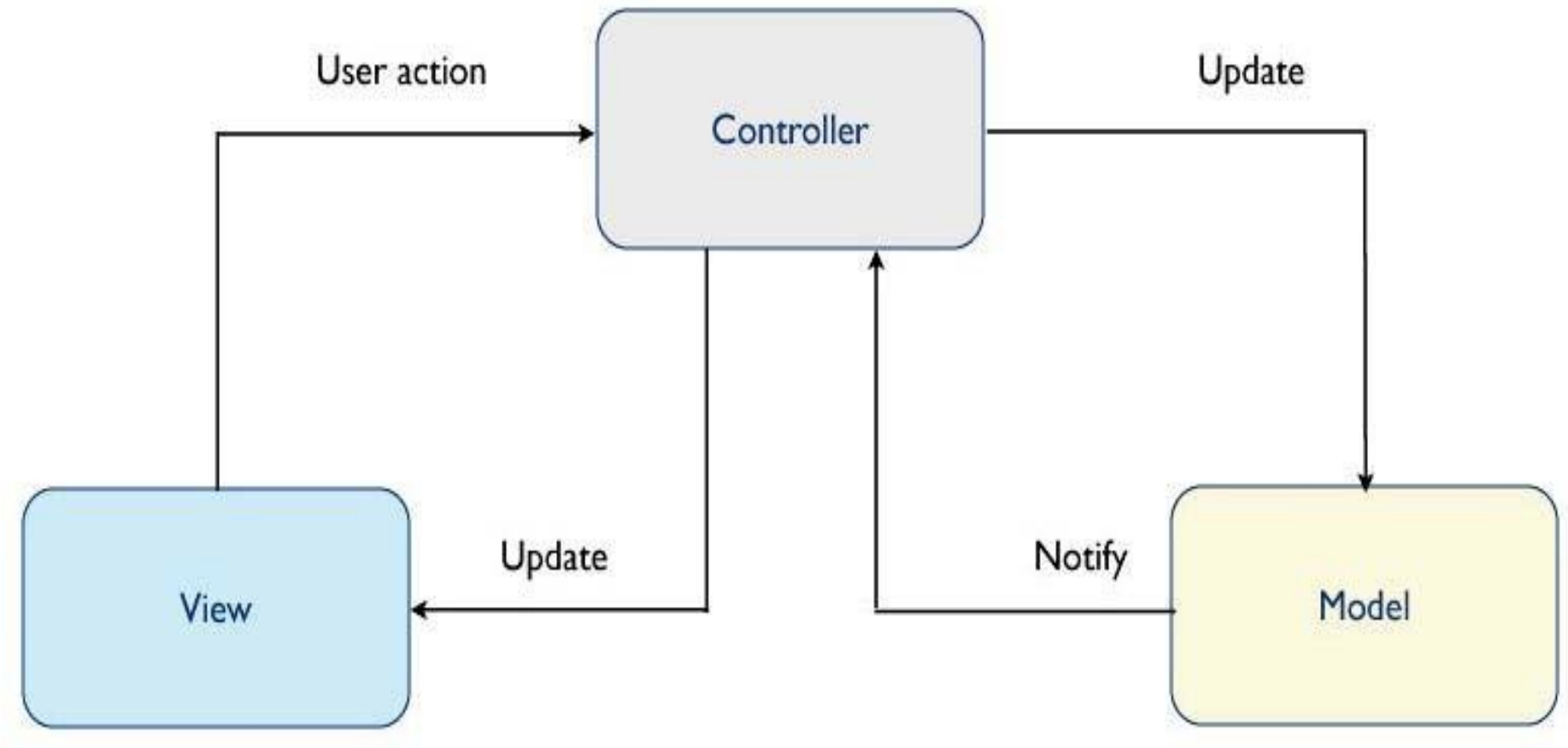
Source: www.udacity.com/course/

Model View Controller (MVC) Design Architecture

Model-View-Controller Design Pattern

- A software design pattern that focuses on the separation of core tasks in an application and to promote code reusability.
- Generally speaking, the pattern splits a program into three sections:
  - ***Model**:* Defines the data and state of the program
  - ***View**:* Defines how the user sees and interacts with the program
  - ***Controller**:* Provides business logic to interface the *View* to the *Model* and can modify the properties of either component based on user interactions or updated data.
- It is important to note that there is a no hard and fast rule regarding the MVC design pattern – it can be viewed as guiding principles to good program design.
  - Many frameworks and developers implement it differently, which were about to see when reviewing Android.

MVC: Architecture Overview



The Model

- The model component of MVC relates to the data that is accessed and stored by the application.
- The model can take many forms:
  - It could be a simple class that stores program state.
  - It could be a class that interacts with database tables.
  - It could be a mix of the above.
- MVC dictates that the **model should only interact with the controller**.
  - The view (UI) should **never** update the model directly, it should notify the controller of an interaction which can update a model.
  - Long story short: The model should never be concerned about how the data is being presented!

The View

- The view is a visualisation of a model and is the user interface that is presented to the user.
  - It handles the way data is presented and the manner in which users can interact with our apps.
- In Android, Views are best represented via *Layouts* (an XML definition of interface controls).
- Again, it is important to stress that the View (UI) has no direct link with the model that it represents.

The Controller

- The controller acts as the connector between the Model and View components.
  - It is responsible for making changes to the model and reflecting said changes in the view.
- In Android, controllers can be represented in one of two ways:
  - ***Activities*** which represents a “screen” of content and can control the flow of the application.
  - ***Fragments*** can be seen as “sub-controllers” where they each can have their own assigned view and behaviour. They can be nested in Activities for greater modular design.

Related Architectural Concept: ECB Pattern

- When identifying the elements for a scenario of system behavior, you can align each participating element with one of three key perspectives: **Entity**, **Control**, or **Boundary**. Although specifics of languages, frameworks, and heuristics of quality design will drive the final design, a first cut that covers required system behavior can always be assembled with elements of these three perspectives.
- This pattern is similar to the Model View Controller pattern, but the Entity Control Boundary (ECB) pattern is not solely appropriate for dealing with user interfaces, and it gives the controller a slightly different role to play.

http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

Entity elements

- An entity is a long-lived, passive element that is responsible for some meaningful chunk of information. This is not to say that entities are "data," while other design elements are "function." Entities perform behavior organized around some cohesive amount of data.

- An example of an entity for a customer service application is a Customer entity that manages all information about a customer.

- A design element for this entity would include data about the customer, behavior to manage the data, behavior to validate customer information and to perform other business calculations, such as "Is this customer allowed to purchase product X?"

- The identification of the entities as part of this pattern can be done many times at different levels of abstraction from the code, at different levels of granularity in size, and from the perspectives of different contexts.

http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

Entity elements

- An entity is a long-lived, passive element that is responsible for some meaningful chunk of information. This is not to say that entities are "data," while other design elements are "function." Entities perform behavior organized around some cohesive amount of data.

- An example of an entity for a customer service application is a Customer entity that manages all information about a customer.

- A design element for this entity would include data about the customer, behavior to manage the data, behavior to validate customer information and to perform other business calculations, such as "Is this customer allowed to purchase product X?"

http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

Control elements

- A control element manages the flow of interaction of the scenario. A control element could manage the end-to-end behavior of a scenario or it could manage the interactions between a subset of the elements. Behavior and business rules relating to the information relevant to the scenario should be assigned to the entities; the control elements are responsible only for the flow of the scenario.

- CreateMarketingCampaign is an example of a control element for a customer service application. This design element would be responsive to certain front-end boundary elements and would collaborate with other entities, control elements, and back-end boundary elements to support the creation of a marketing campaign.

http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

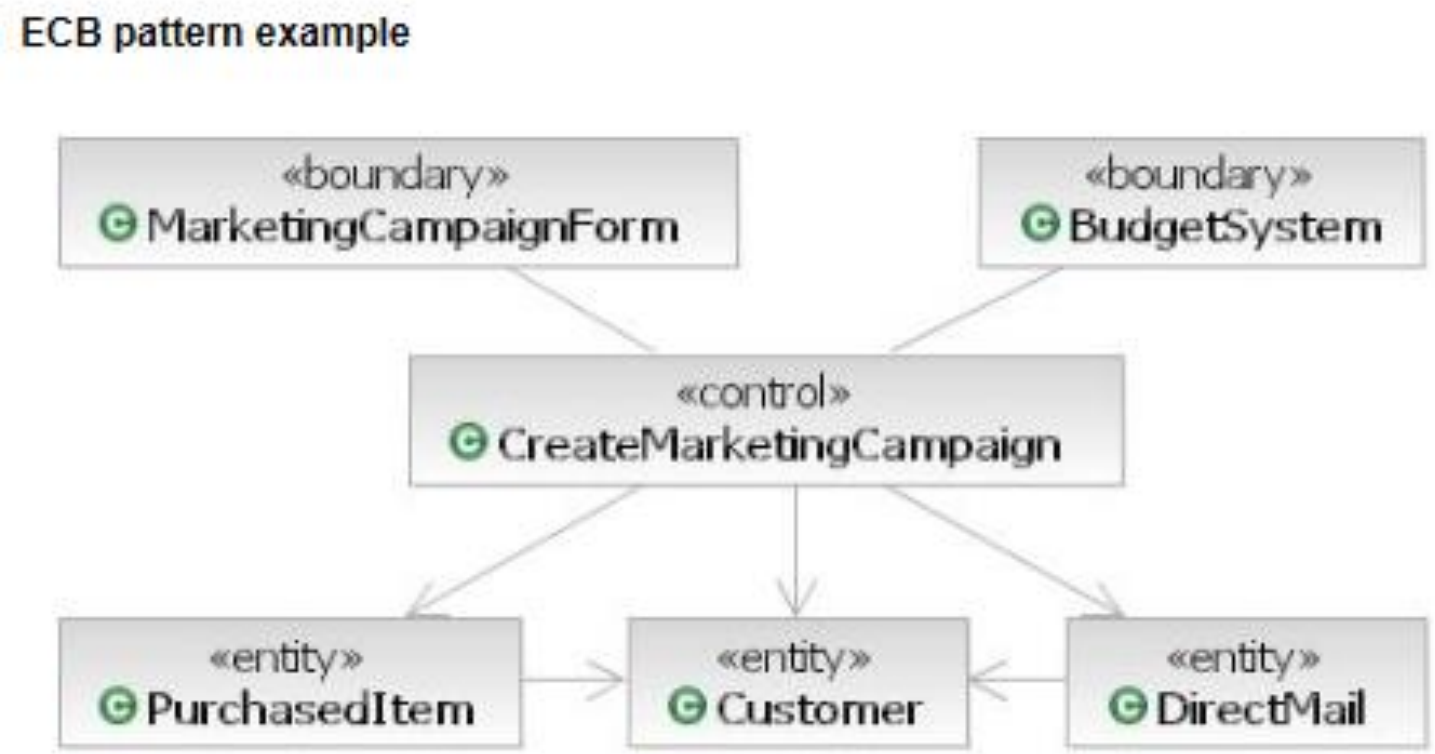
Boundary elements

- A boundary element lies on the periphery of a system or subsystem, but within it. For any scenario being considered either across the whole system or within some subsystem, some boundary elements will be "front end" elements that accept input from outside of the area under design, and other elements will be "back end," managing communication to supporting elements outside of the system or subsystem.

- Two examples of boundary elements for a customer service application might be a front end MarketingCampaignForm and a back end BudgetSystem element. The MarketingCampaignForm would manage the exchange of information between a user and the system, and the BudgetSystem would manage the exchange of information between the system and an external system that manages budgets.

http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

ECB example



http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

ECB linkages

- There are certain appropriate relations between the participating elements. An element can communicate with other elements of the same kind. Control elements can communicate with each of the other two kinds, but entities and boundary elements should not communicate directly.
- This table shows appropriate links between design elements.

	Entity	Boundary	Control
Entity	X		X
Boundary			X
Control	X	X	X

http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/guidelines/entity_control_boundary_pattern_C4047897.html

Layouts and Common Interface Controls

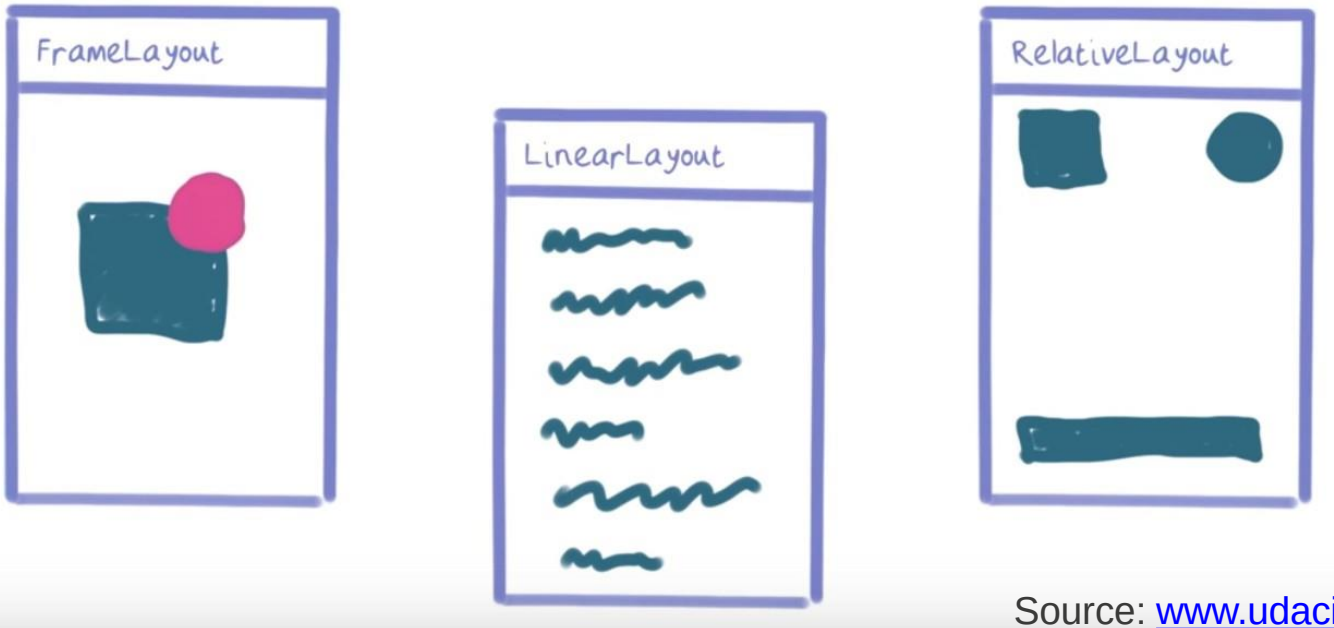
Layouts

- Layouts help define how the user interface is structured on **Views** and how each view is positioned.
- Layouts serve as the visual structure for user interfaces in Android.
  - Declared by XML or instantiated during runtime.
  - When the application is being compiled, each layout XML declaration is compiled into a *View* resource.
- Layouts can range from entire Activity Screens to individual components such as List Items.
  - We use the ***LayoutInflater*** class to instantiate a layout XML declaration into View objects.
  - This allows us to build View objects that use a new layout and instantiate values for interface controls (such as TextView).
- Layouts loaded in *onCreate* method in an Activity and can be set during runtime via:

```
setContentView(R.layout.layout_name);
```

Layouts

- Multiple layouts can be nested within each other.
- Common layouts:
  - LinearLayout
  - RelativeLayout
  - GridLayout



Source: www.udacity.com/course/

LinearLayout

- Can arrange views both horizontally (row) and vertically (column) via setOrientation.
  - Generally speaking, this is the most common layout you will work with.
- Child views can have their alignment manipulated via the gravity property via **setGravity**.
- Weight property allows elements to take a proportion of the remaining space in the layout in the specified orientation.
  - Example: Two buttons in a horizontal linear layout.
    - Button A has a weight value of 1 and Button B has a weight value of 2.
    - Button A takes 1/3 of the layouts width while Button B takes 2/3 of the width.

RelativeLayout

- Can arrange views relatively based upon the position of each element.
  - Generally speaking, this allows you to position views around the layout with a relationship between other views.
  - Can be more flexible when you don't require a strict layout (LinearLayout can create lots of whitespace).
- The gravity property still applies to child views that are nested within the layout.

GridLayout

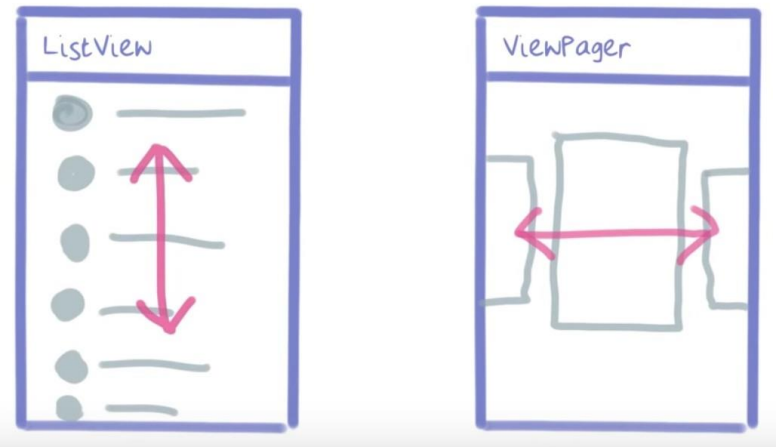
- Introduced from Android 4.0+, this layout allows you to place child views in a rectangular grid with customisable rows and columns.
- Ideal use cases would be views with similar or exact height and width layout proportions (such as images or square buttons).
  - Does become cumbersome to use when you have variable width / height views trying to work with the layout itself.

Common Interface Controls

- So what controls can we use that are common to the Android platform?
  - Buttons
  - Checkboxes
  - Text fields
  - Radio buttons
  - Switches
  - And so on!
- For a full overview, it is best to consult the Android developer guidelines on http://developer.android.com/

ListViews

- Lists are a common format shared across mobile platforms and a very powerful tool in Android.
- ListViews in Android are able to display a scrollable list of views containing content from simple Strings to complex data structures.
  - You are able to create your own layout for content items you wish to represent.
- ListViews each have a supporting ListAdapter which handles inserting data and creating the view for each row.



ListAdapters

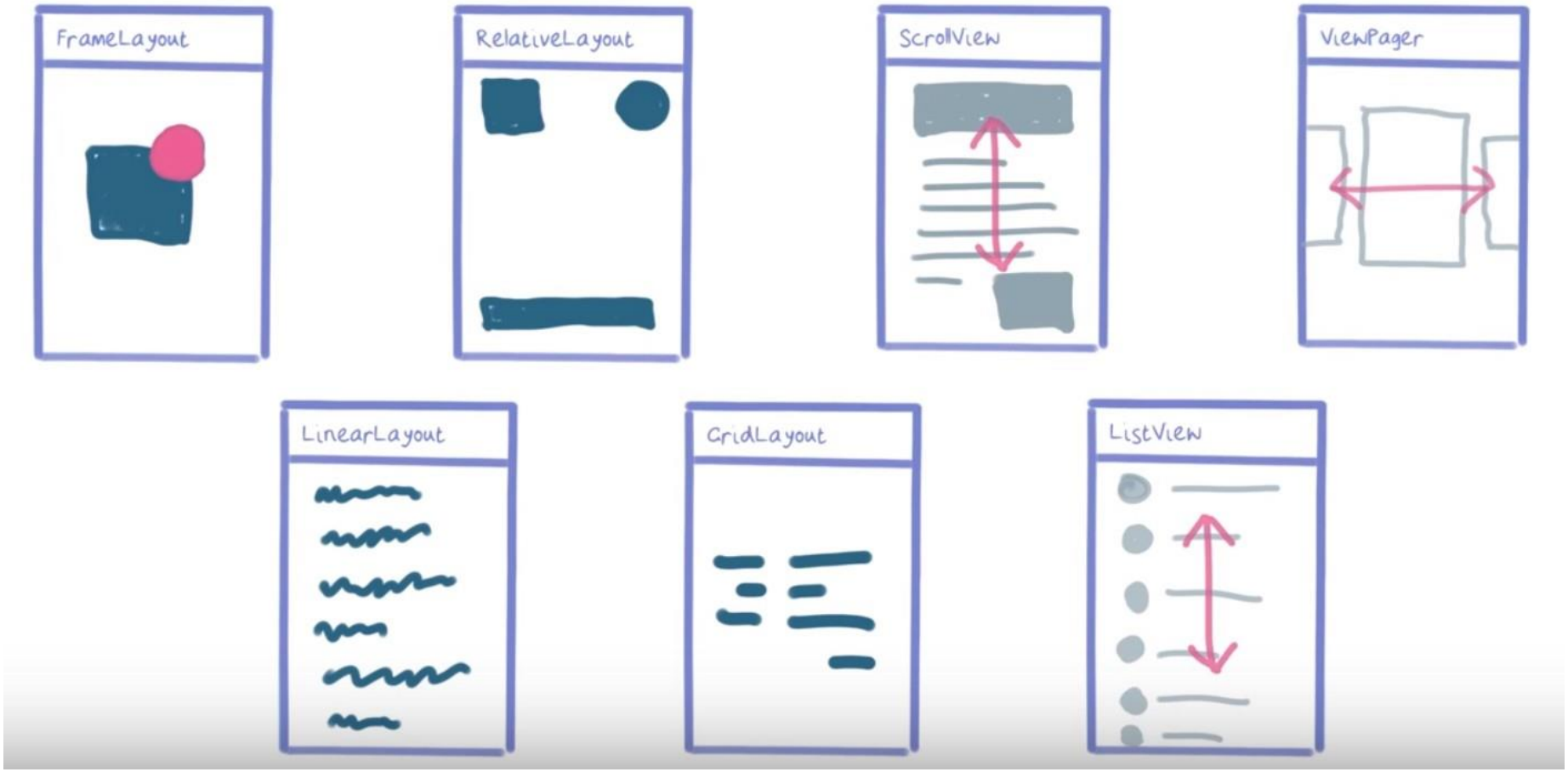
- Many applications often have unique data structures which need to be represented in different ways.
- There are a few common adapters we can use in Android:
  - **ArrayAdapter**: used when the data source is an array.
    - Calls toString() on array objects and feeds the returning String into a TextView.
  - **SimpleCursorAdapter**: used to map results from a Cursor (returned with database queries, we will explore this next week).
  - **BaseAdapter**: Common base class which all adapters
    - extend from. Often used for creating adapters for custom app requirements.

Custom ListAdapters

- The default adapters can become quite limited, especially if you want to output specific object information that may come in different formats such as string and image resources.
- **BaseAdapter** is a good adapter to build from since it covers the core functionallity required for an adapter to function in a supporting view (such as ListView).
  - This also means we will need to create a new layout for our rows to make it truly flexible.
  - Most lists are not just standard Strings, they can encapsulate other resources such as images and custom views.
  - You can extend BaseAdapter and create our own layout for a list row to demonstrate how to use a custom dataset.

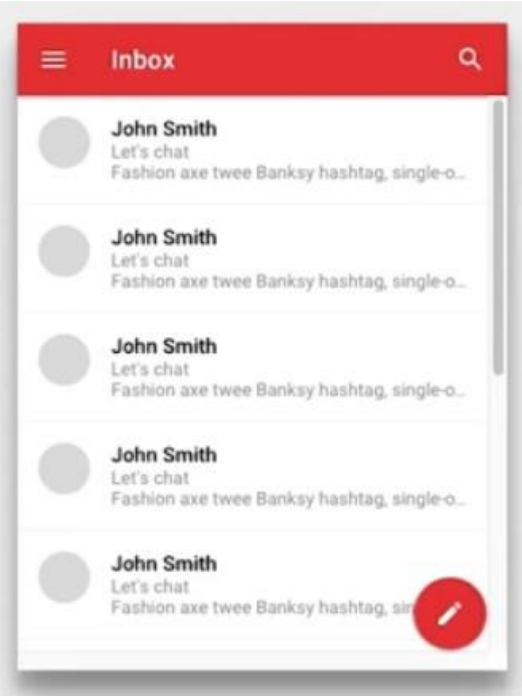
Identifying Layouts

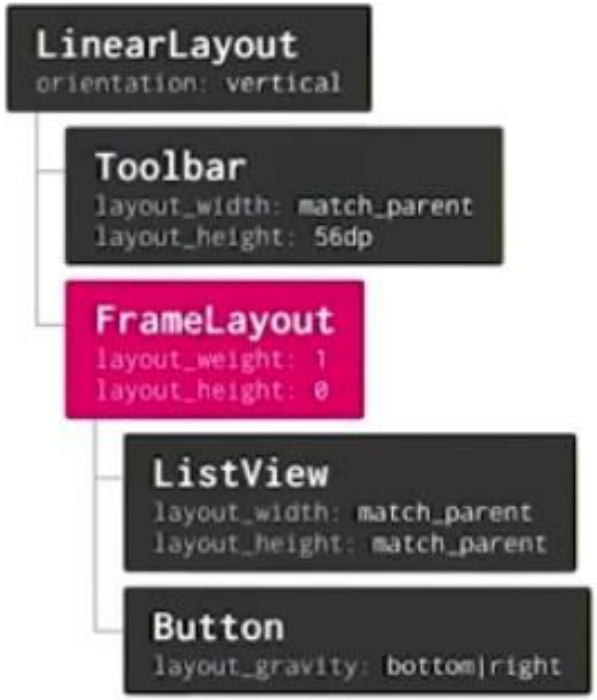
Kinds of Layouts

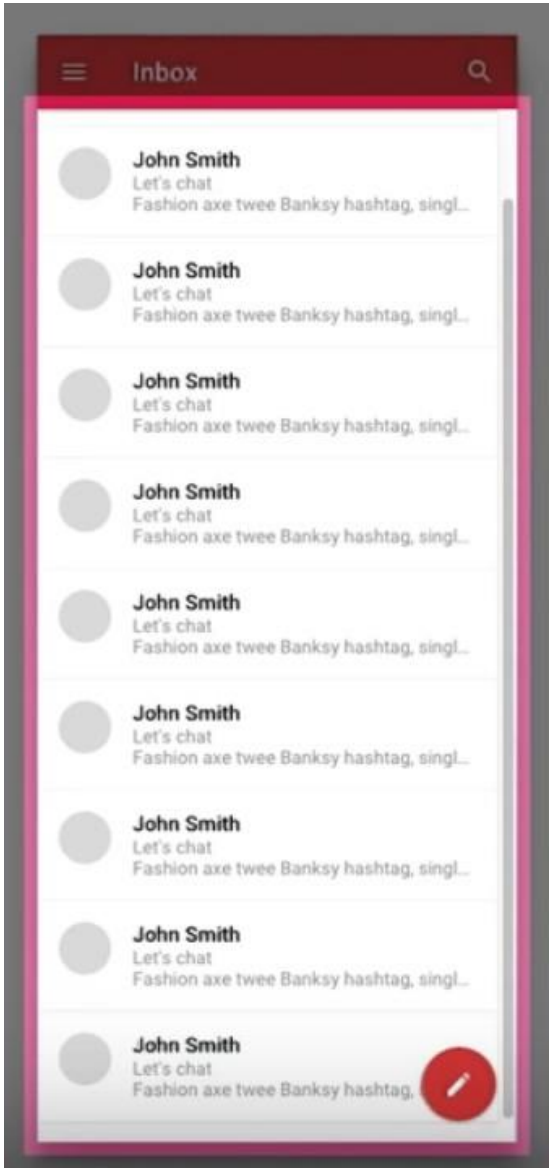


Source: www.udacity.com/course/

Layout Case Study One

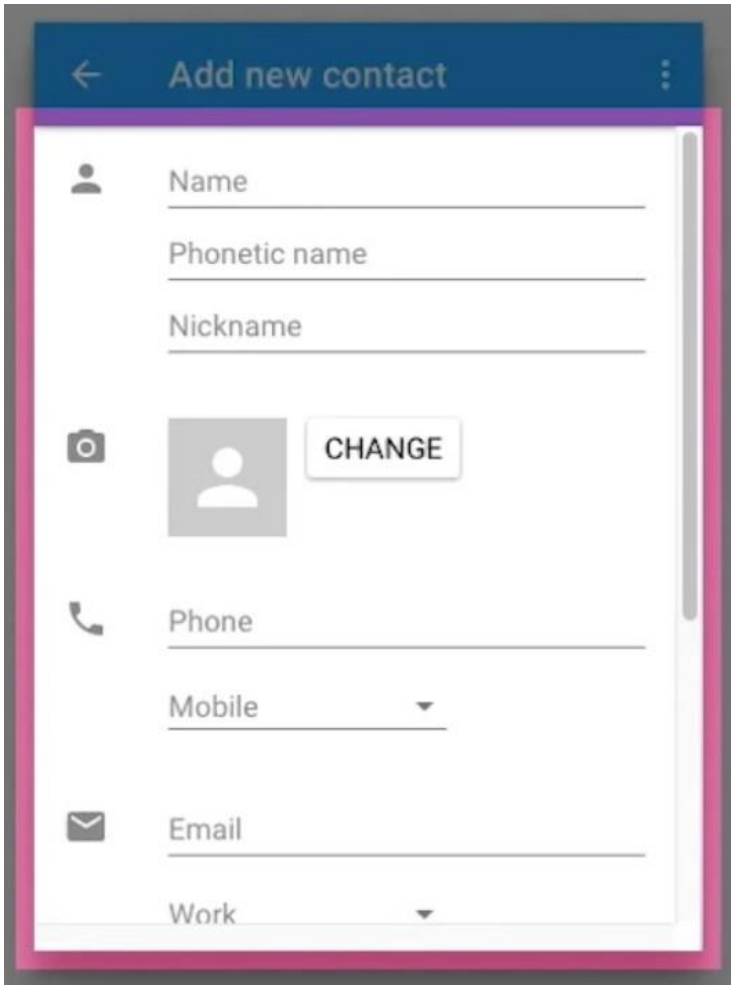


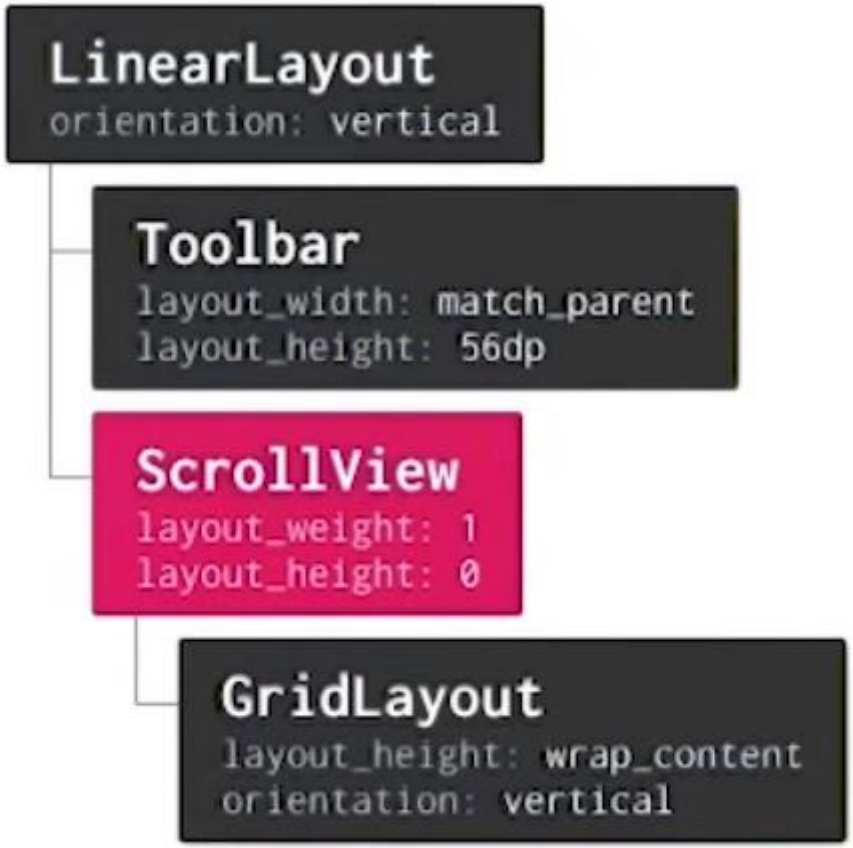




Source: www.udacity.com/course/

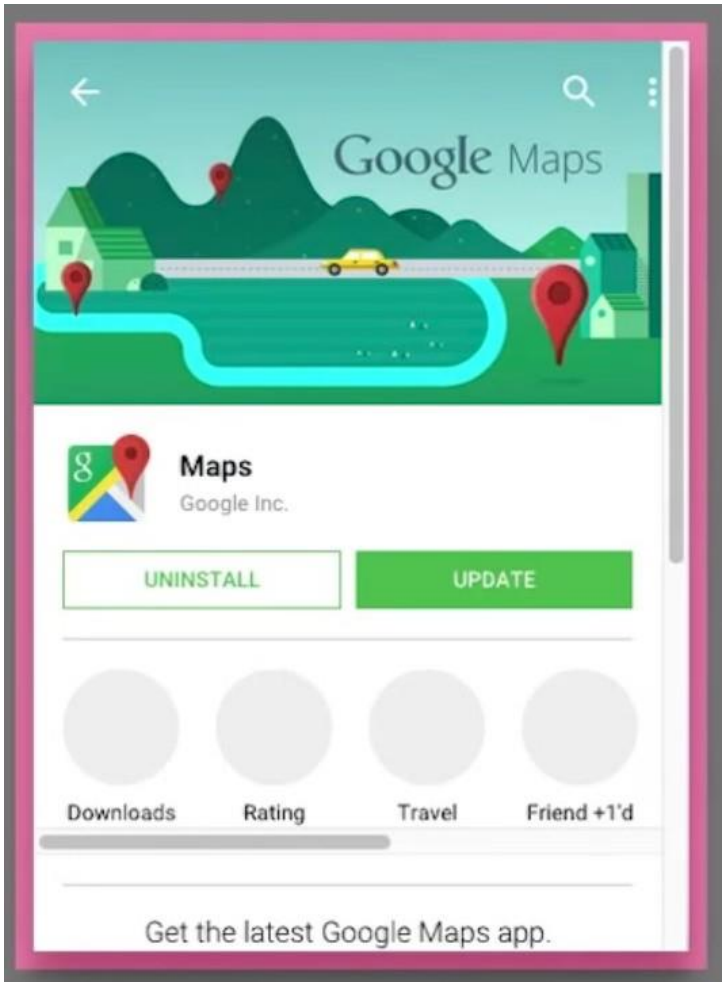
Layout Case Study Two

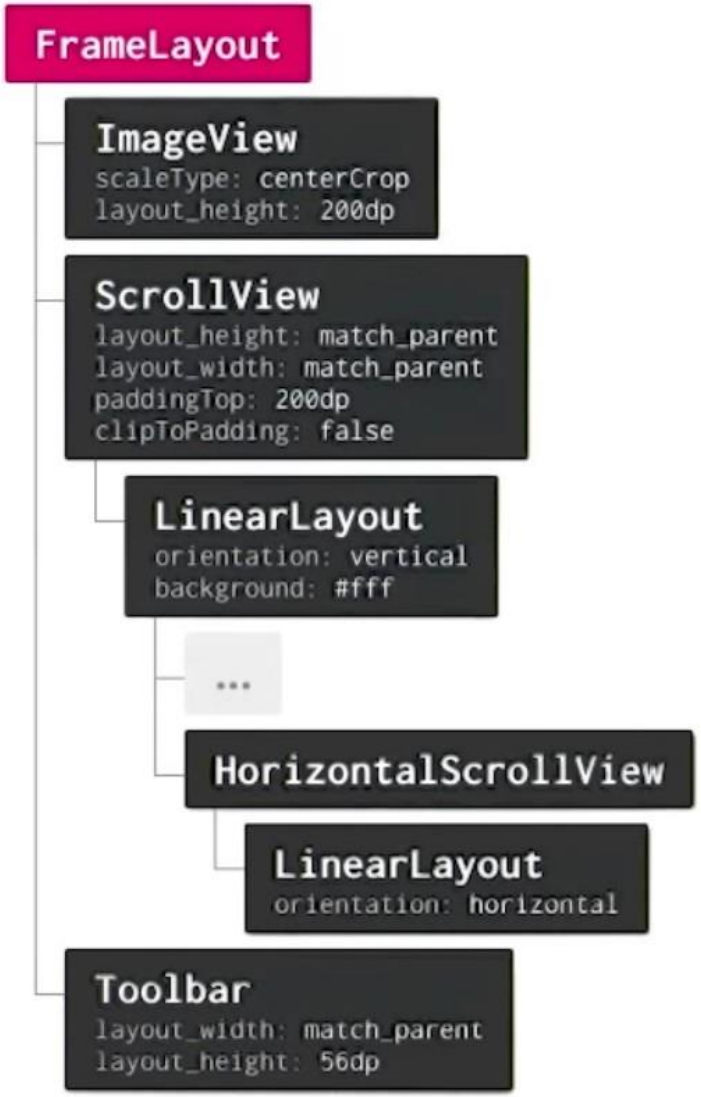




Source: www.udacity.com/course/

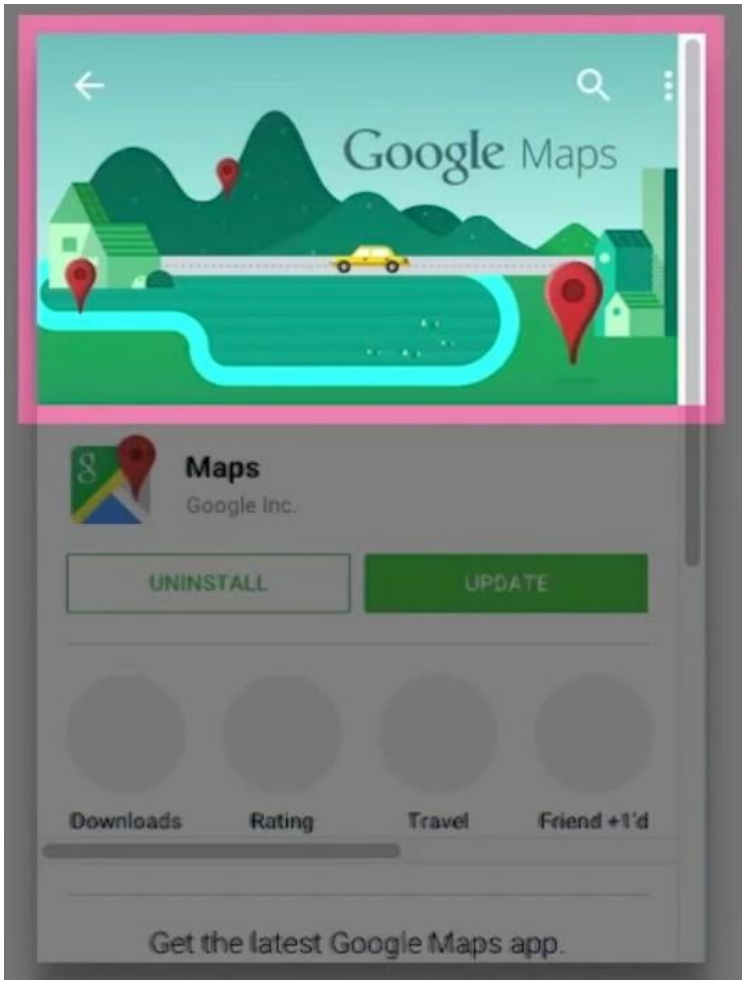
Layout Case Study Three

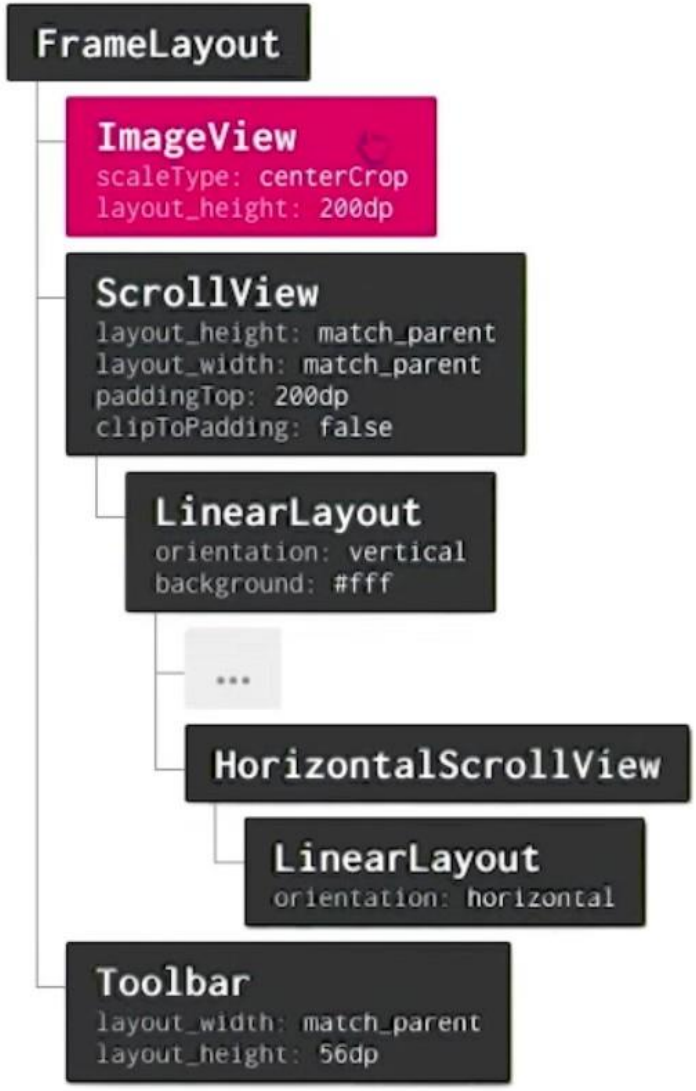




Source: www.udacity.com/course/

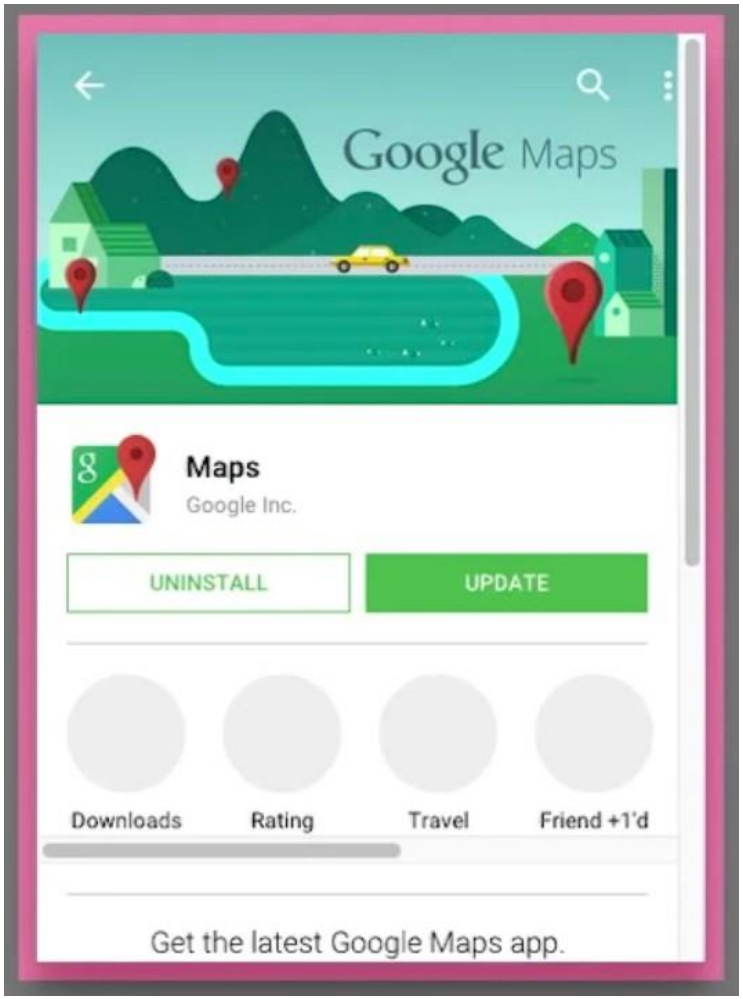
Layout Case Study Three

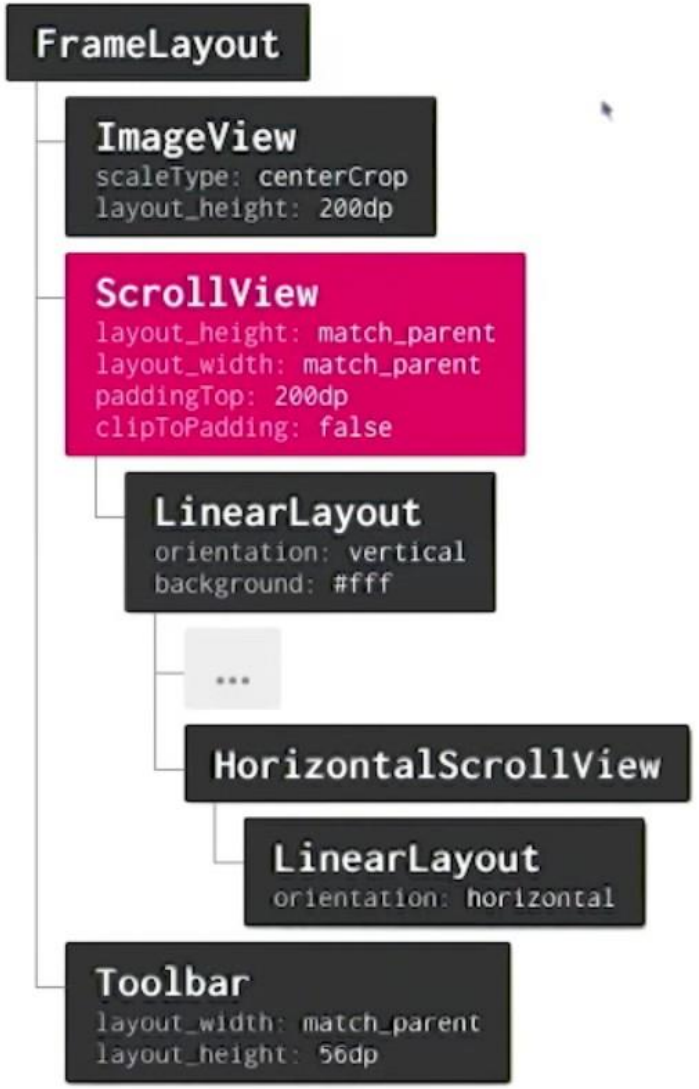




Source: www.udacity.com/course/

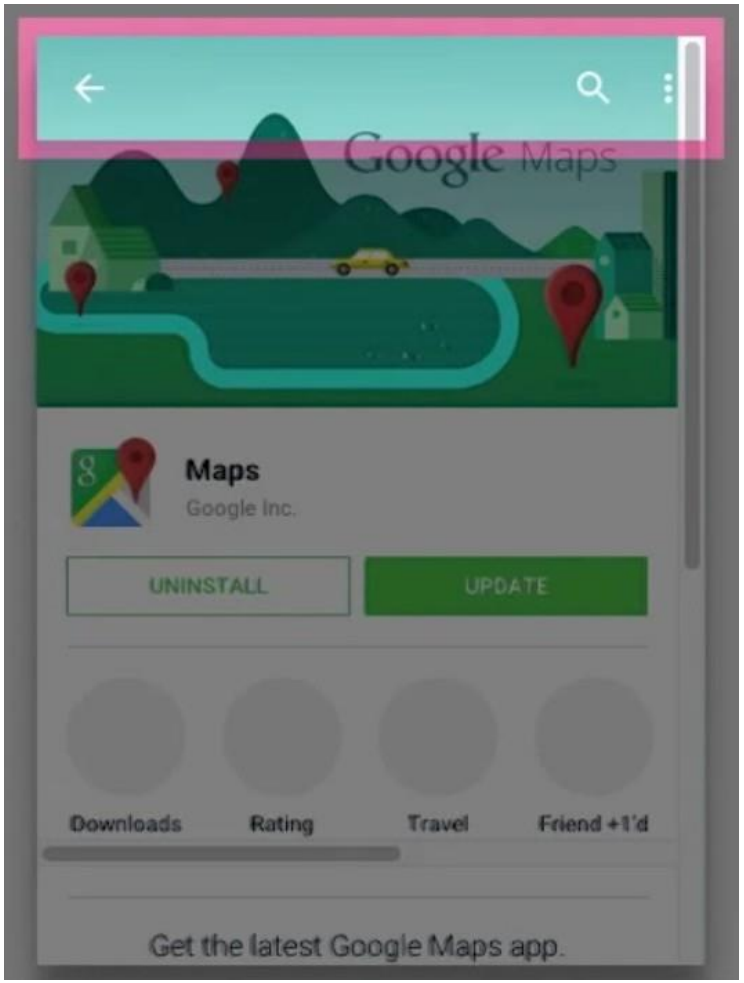
Layout Case Study Three

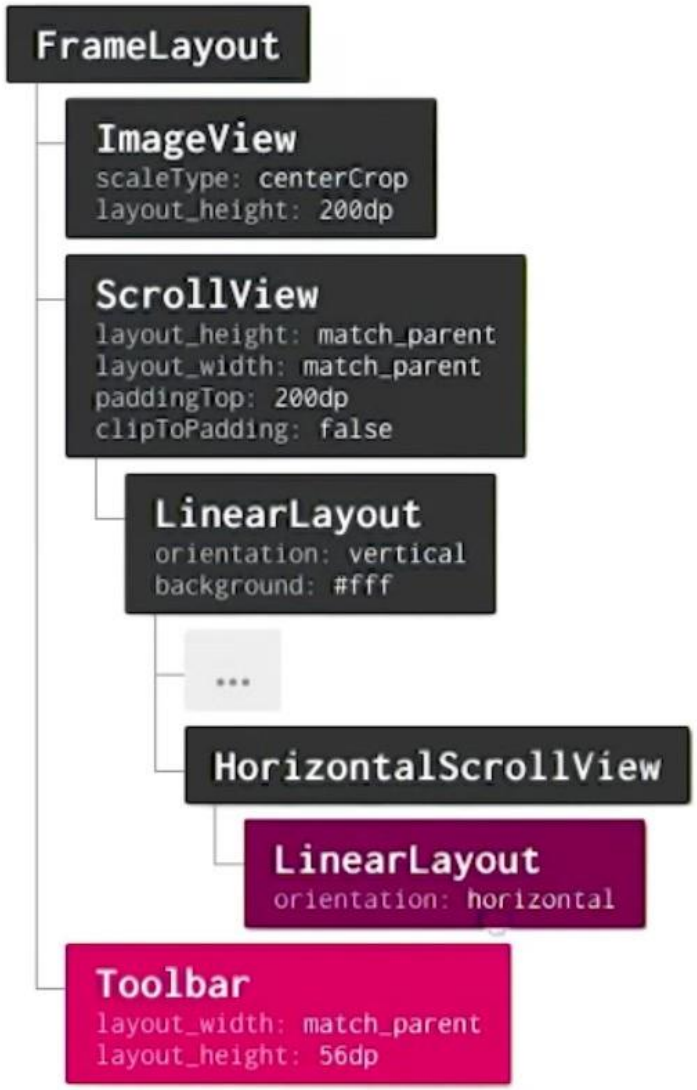




Source: www.udacity.com/course/

Layout Case Study Three





Source: www.udacity.com/course/

Common Interface Design Patterns

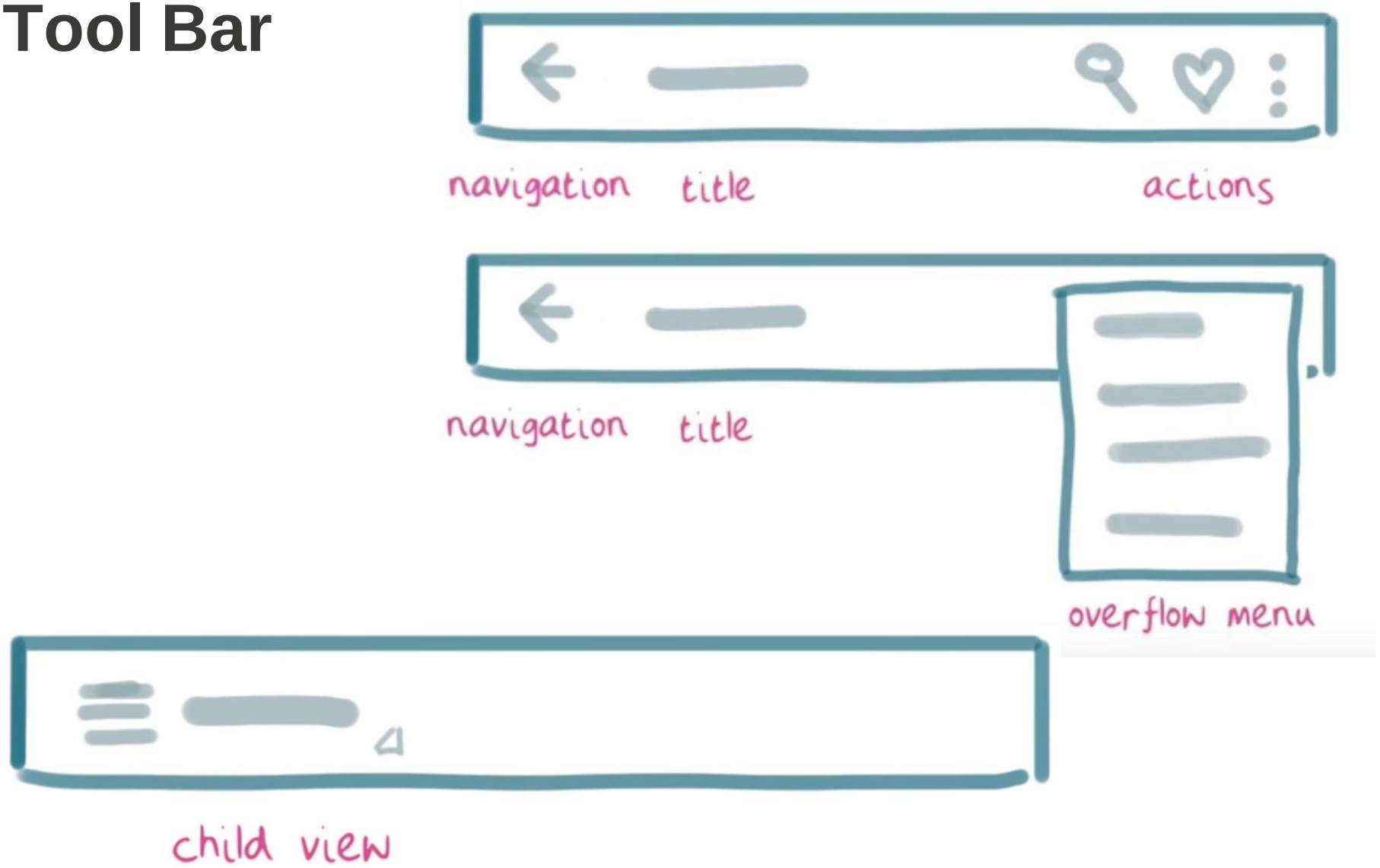
Common Interface Design Patterns

- Content, Padding and Margins
- Tool bar
- App bar and Tabs
- Navigation Drawer
- Scrolling and Paging
- List to Details

Content, Padding and Margins

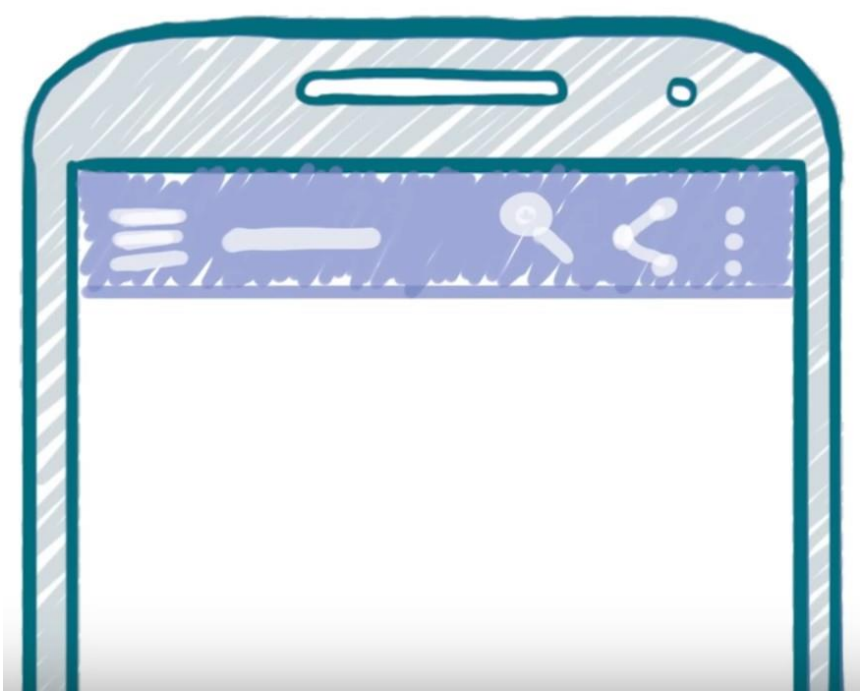
Source: www.udacity.com/course/

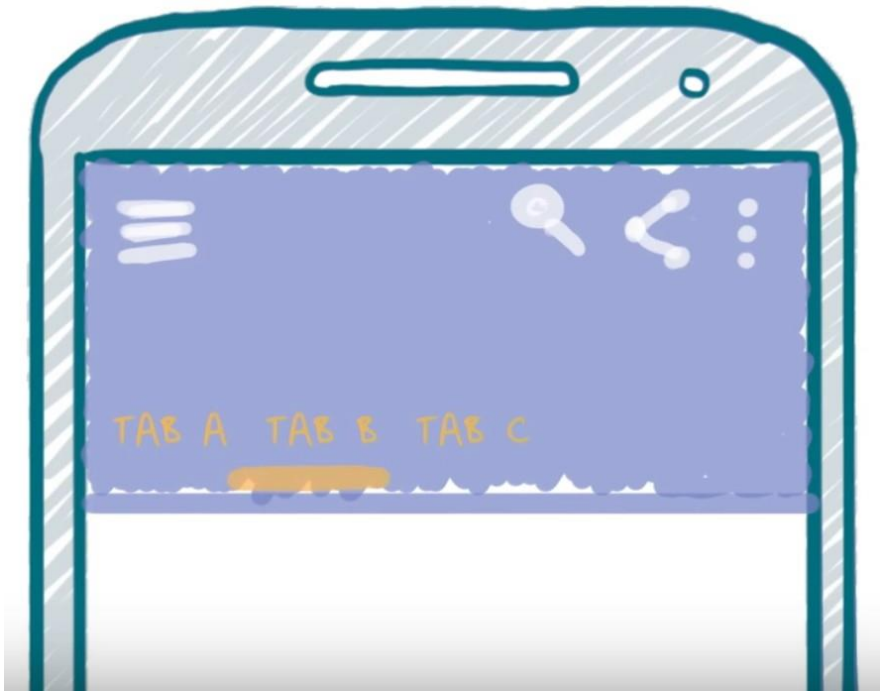
Tool Bar



Source: www.udacity.com/course/

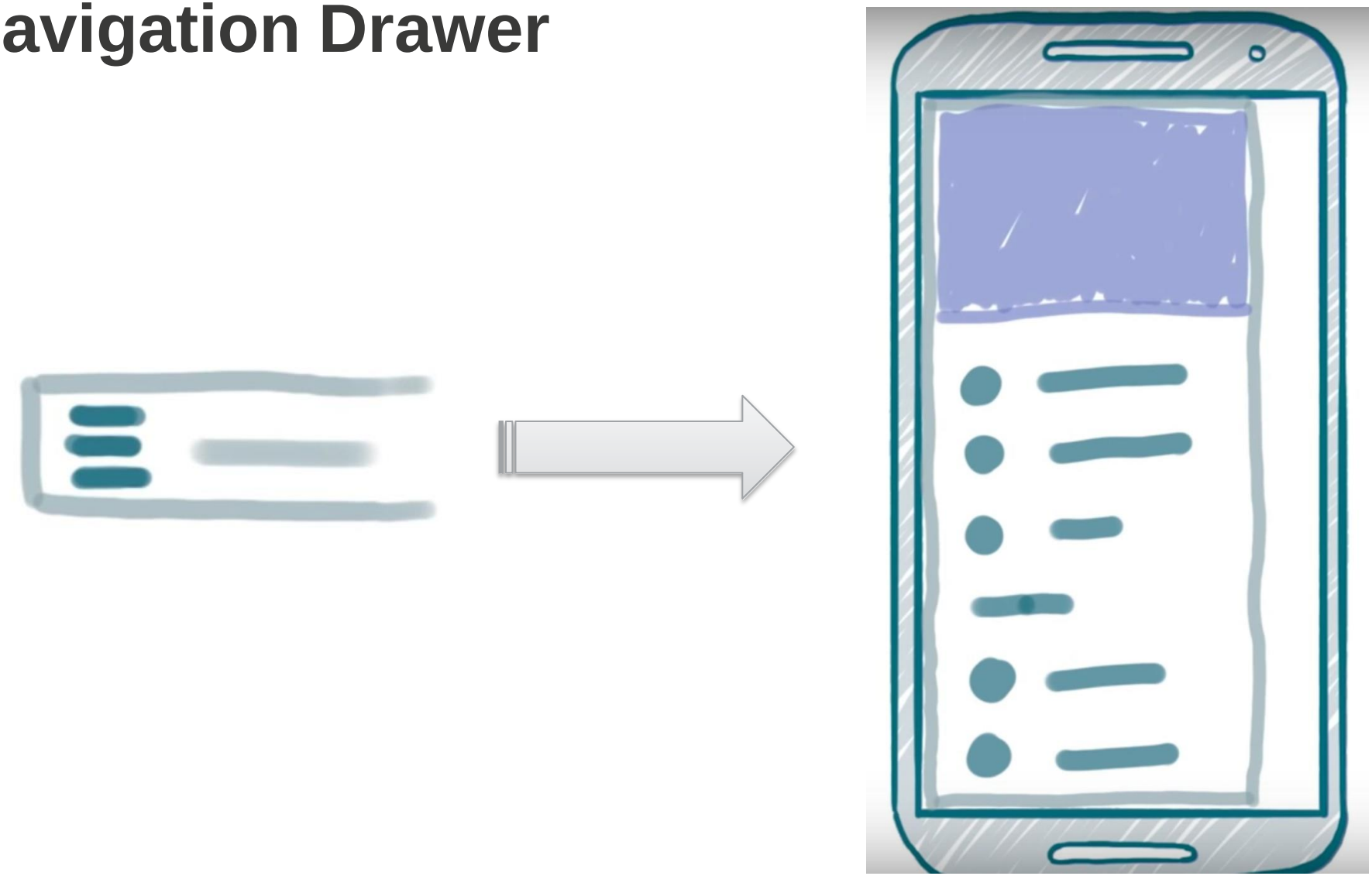
App Bar and Tabs





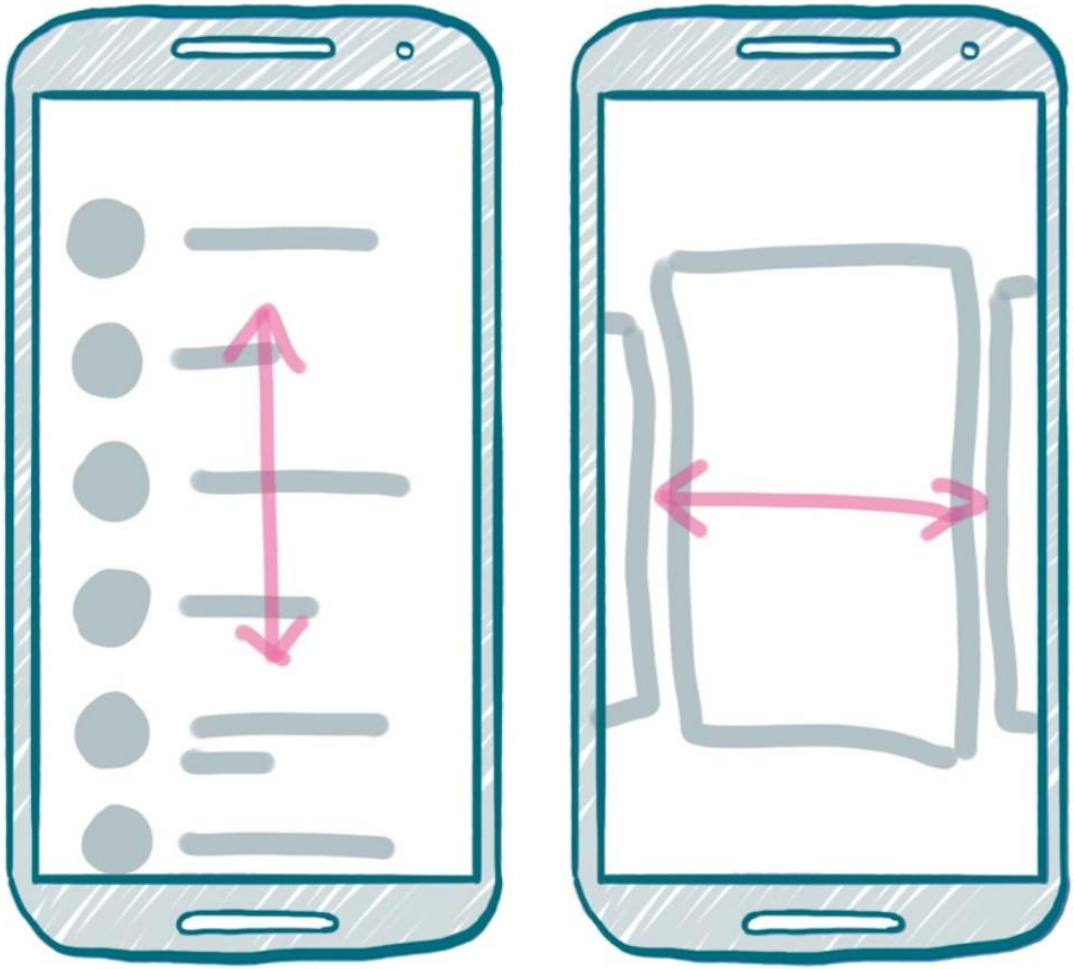
Source: www.udacity.com/course/

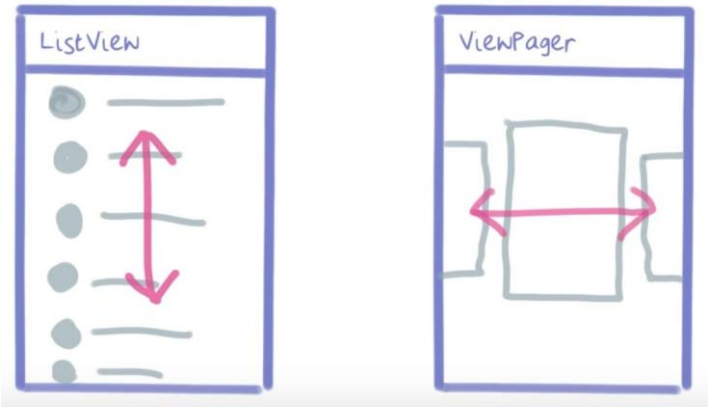
Navigation Drawer



Source: www.udacity.com/course/

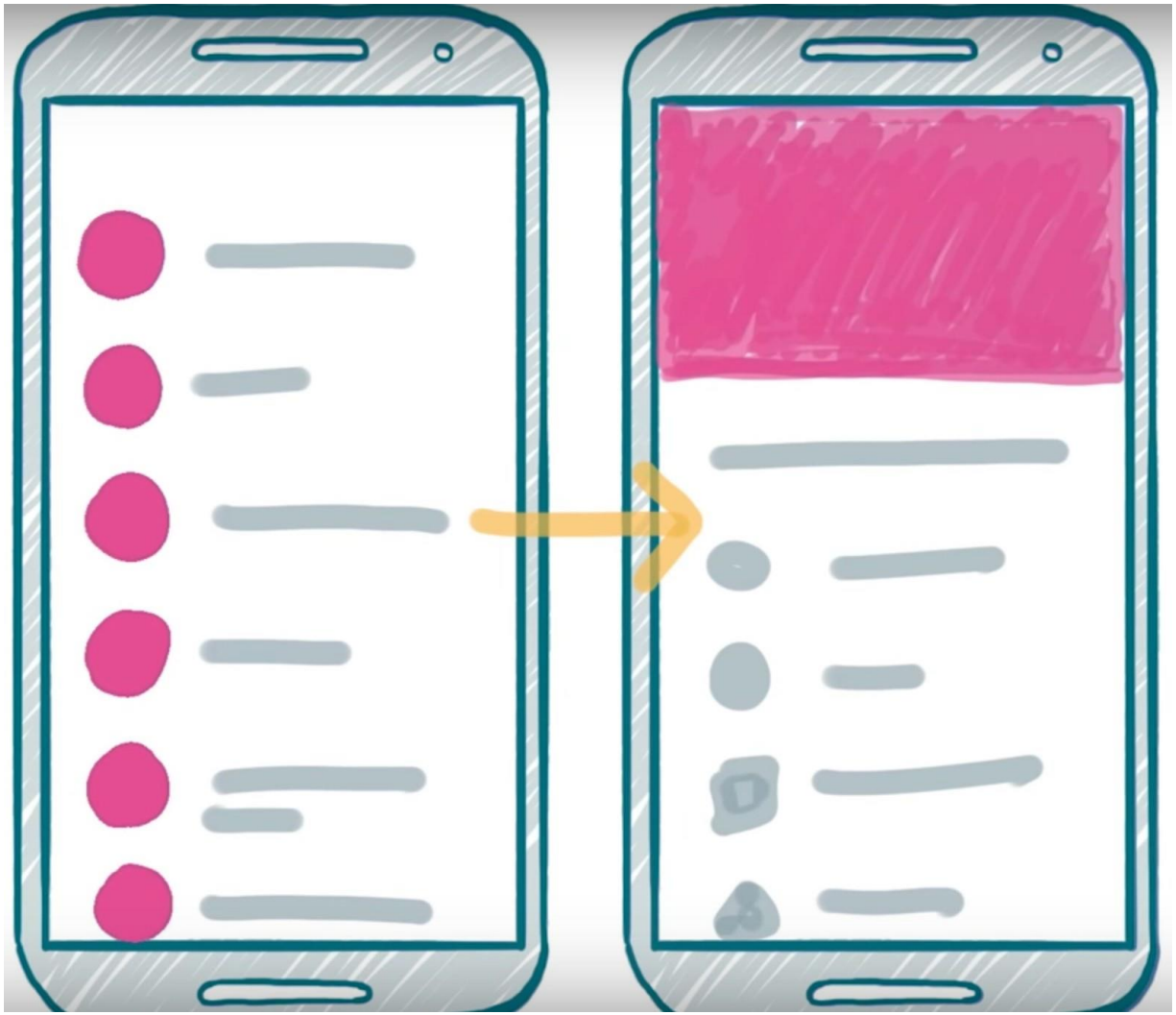
Scrolling and Paging





Source: www.udacity.com/course/

List to Details



Source: www.udacity.com/course/

References

- Android Design Principles: http://developer.android.com/design/get-started/principles.html
- Material Design for Android: http://developer.android.com/design/material/index.html
- Material Design for Android Developers: https://www.udacity.com/course/
- Google Design: https://design.google.com/
- Material Design Principles: http://www.google.com/design/spec/material-design/