# Mobile Design and Development

# Introduction

- In SA, mobiles dominate over laptops and desktops for internet usage and ownership
    - But data is slower and more costly
- Diversity in mobile devices (not just phones)
- Phones are more than just communication
- Mobile is context
    - Various contexts and ways to use phones
        - Use dependant on social, technology, environment and infrastructure

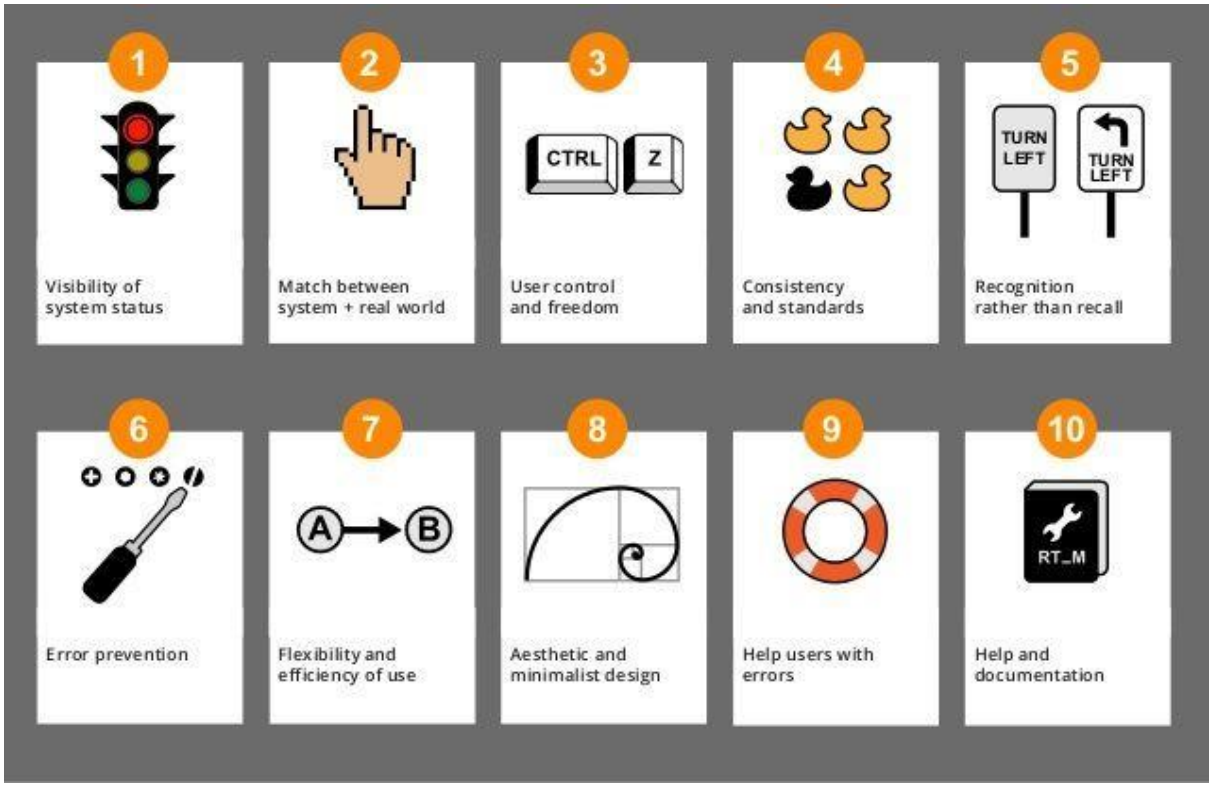
# User Centred Design

- Interaction Design: understanding the dynamics of how we interact with computing devices
    - Includes social dynamics, as well as high level (how to use the app)
    - The design of spaces for human communication and intersection (with computing devices and with one another)
- Human Computer Interaction – need interdisciplinary approach
    - Drawing upon other disciplines – like social science, psychology, to help inform our design principles and approaches

## GOALS


- At a high level – user interaction designers seek to build applications that are learnable, discoverable, effective, and not just usable, but pleasant to use.
- At the second order, however, have lots of different goals, from getting lots of likes and user retention to perhaps social good goals like improving how people use masks or understand their health.
    - Perhaps on the more sinister end, apps and services are designed and manipulated to foster addiction or affect public opinion and fears of foreign governments manipulating elections.

- How do we become good designers?
    - Need to follow written and unwritten design principles
    - Standards and guidelines that help us to predict whether someone will like a particular interface, and will help us to design something people will be able to use




### Visibility of System Status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

### Match Between System and the Real World

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

### User Control and Freedom

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

### Consistency and Standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

### Error Prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

# 10 Usability Heuristics

for User Interface Design

### Recognition Rather than Recall

Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

### Flexibility and Efficiency of Use

Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

### Aesthetic and Minimalist Design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

### Help Users Recognize, Diagnose, and Recover from Errors

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

### Help and Documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

by Jakob Nielsen

## Key Performance Indicators

- User retention
- More clicks/screen time
- More ad views
- Persuasion
- …

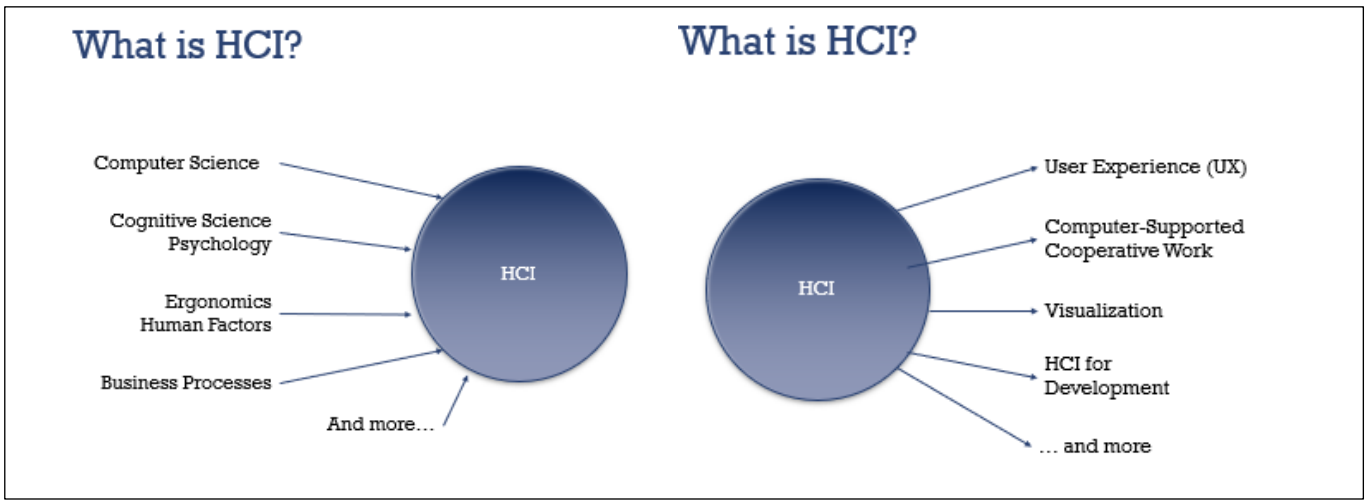
→ making more money!

→ political objectives

# The Design Cycle

- Needs between groups of users can vary - new interfaces may not have tried-and-true standards for implementation
- Iterative development coupled with customer or user engagement helps us to get to designs that ultimately work better for the actual real users of the systems.
    - It also happens to help us work better together.
- The design cycle is core to our approach to **doing design in a user-centered way**.
    - You start by evaluating what you know about the users – then based on that, come up with a design.
    - Finally, you actually build something.
    - This then starts the next cycle – an evaluation of what you've prototyped, analysis of the findings, resulting in a new design, followed by building the prototype. This continues as long as you have time until everyone is satisfied with the final product.

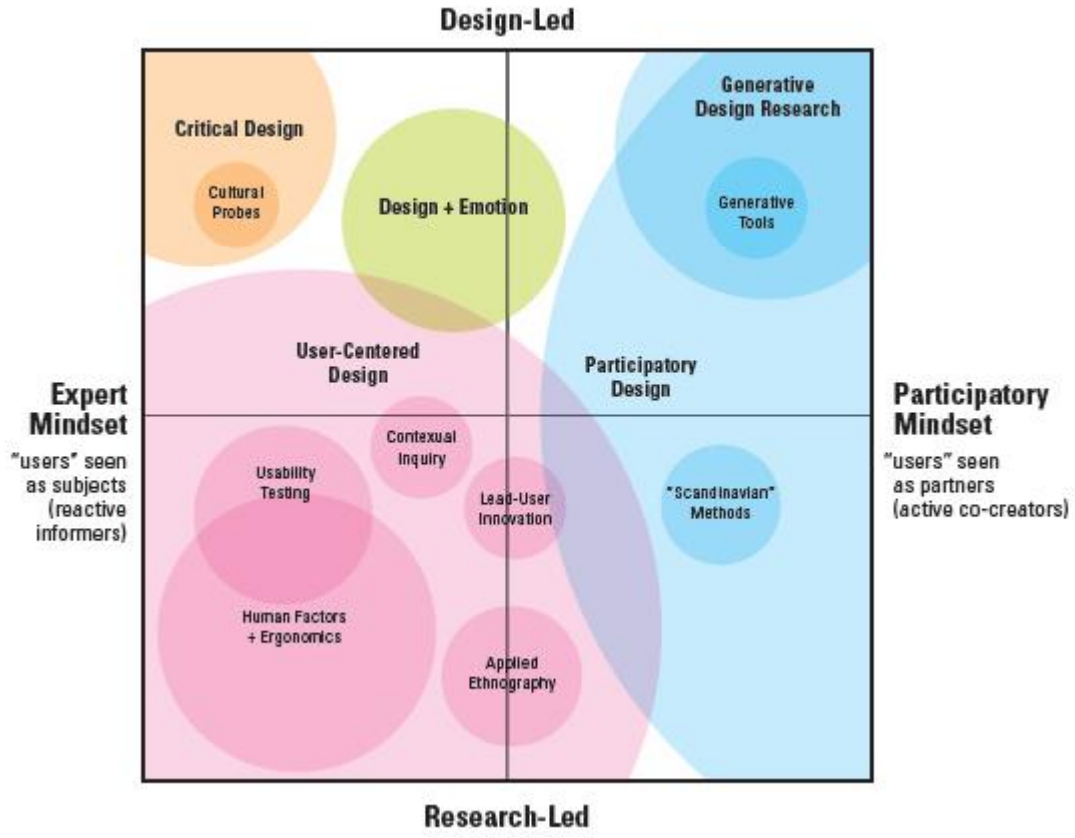
# Human Computer Interaction




## Spectrum of Design

- *Humans* play a significant role in whether and how computing systems work
- Human Computer Interaction (HCI):
    - Draws on many disciplines including cognitive psychology, anthropology, ergonomics, socio-technical design, and computer science.
    - HCI also has many sub-branches like UX (user experience)

## DESIGN




- This graph depicts some of the various approaches we take to design, organized by mindset towards design and the extent to which the practice is led by human designers or empirical outcomes
- Mindset axis is important to understanding design
    - Axis moves from an "expert mindset" in which the users are informants…
        - They can react to your interfaces and tell you what's wrong or what they want
        - Job of the designing is up to the "experts".
    - …to a participatory mindset in which users are invited to co-create and co-design their interactions.
- User plays a part in both sides of the axis - at the center of our activities
    - Whether they are participating in the design or not
- Tools such as interviews, observations, a/b testing are used to varying degree in all of these approaches, and most design involves some version of the design cycle

## HCI IMPORTANCE

- User Error vs Designer Error
- Devs and clients - often prioritize functionality over user experience, not understanding the critical role design plays in adoption, efficiency, and correct use of technology

# The Golden Rules

Shneiderman's Golden Rules of Design

1. Strive for consistency.
2. Cater to universal usability.
3. Offer informative feedback.
4. Design dialogs to yield closure.
5. Prevent errors.
6. Permit easy reversal of actions.
7. Support internal locus of control.
8. Reduce short-term memory load.

## STRIVE FOR CONSISTENCY

- **Consistent sequences of actions should be required in similar situations.**
- Identical terminology should be used in prompts, menus, and help screens
- Consistent colour, layout, capitalization, fonts, and so on, should be employed throughout
- Exceptions, such as required confirmation of the delete command or no echoing of passwords, should be comprehensible and limited in number
- Summary:
    - Have the same sequences of actions for similar situations, use same words, have consistent aesthetics and layout, limit the # of exceptions
- It allows people to transfer their knowledge of one system to another
    - why it can be difficult to move from iphone to android

## CATER TO UNIVERSAL USABILITY

- **Recognize the needs of diverse users and design for plasticity, facilitating transformation of content.**
- Novice to expert differences, age ranges, disabilities, international variations, and technological diversity each enrich the spectrum of requirements that guides design.
- Adding features for novices, such as explanations, and features for experts, such as shortcuts and faster pacing, enriches the interface design and improves perceived quality.
- Summary:
    - Put in features for novices and experts, and cater for age differences, disabilities and global variations i.e. make it such that all can use it
- Examples: supporting multiple languages, including alt-text for images, explanations for novices, shortcuts for experts etc

## OFFER INFORMATIVE FEEDBACK

- **For every user action, there should be an interface feedback.**
- For frequent and minor actions, the response can be modest, whereas for infrequent and major actions, the response should be more substantial.
- Visual presentation of the objects of interest provides a convenient environment for showing changes explicitly.
- Summary:

    - Always provide feedback – if action frequent, minimal feedback; for infrequent feedback, substantial feedback.
- Examples:
    - Confirmation as you submit an assignment

## DESIGN DIALOGS TO YIELD CLOSURE

- **Sequences of actions should be organized into groups with a beginning, middle, and end.**
- Informative feedback at the completion of a group of actions gives users the satisfaction of accomplishment, a sense of relief, a signal to drop contingency plans from their minds, and an indicator to prepare for the next group of actions.
- For example, e-commerce websites move users from selecting products to the checkout, ending with a clear confirmation page that completes the transaction.
- Clear sequence of events helps to clarify the interface
- Summary:
    - Have clear sequences of actions with beginning middle and end – do this to give users peace of mind

## PREVENT ERRORS

- **Design the interface so that users cannot make serious errors**
    - Examples: gray out menu items that are not appropriate and do not allow alphabetic characters in numeric entry fields.
- If users make an error, the interface should offer simple, constructive, and specific instructions for recovery.
    - Examples: users should not have to retype an entire name-address form if they enter an invalid zip code but rather should be guided to repair only the faulty part.
- Erroneous actions should leave the interface state unchanged, or the interface should give instructions about restoring the state
- Summary:
    - Don't allow users to make serious errors, inform them when they do make an error, don't let errors change the interface state

## PERMIT EASY REVERSAL OF ACTIONS

- **As much as possible, actions should be reversible.**
- This feature relieves anxiety, since users know that errors can be undone, and encourages exploration of unfamiliar options.
- The units of reversibility may be a single action, a data-entry task, or a complete group of actions, such as entry of a name-address block.
- Summary:
    - Allows actions to be reversed - allow undo!
- For example:
    - "Confirm submit" when submitting assignments prevents accidentally submitting when not ready
    - Allowing assignment resubmissions

## SUPPORT INTERNAL LOCUS OF CONTROL

- **Experienced users strongly desire the sense that they are in charge of the interface and that the interface responds to their actions.**
- They don't want surprises or changes in familiar behaviour, and they are annoyed by tedious data-entry sequences, difficulty in obtaining necessary information, and inability to produce their desired result.
- Summary:
    - Don't change things unnecessarily - inconsistency might lead to people feeling a loss of control

### REDUCE SHORT-TERM MEMORY LOAD

- Humans' limited capacity for information processing in short-term memory (the rule of thumb is that people can remember "seven plus or minus two chunks" of information) requires that designers **avoid interfaces in which users must remember information from one display and then use that information on another display.**
- It means that cellphones should not require re-entry of phone numbers, website locations should remain visible, and lengthy forms should be compacted to fit a single display.
- Summary:
    - Humans have short term memory, don't ask users to re-enter data
- Example:
    - Part of WhatsApp's ease of use is that they've done away entirely with passwords

## Applications

- Applying these rules in design is not necessarily easy
- It's expected that you will interpret and refine the rules based on context, and apply them in concert

# Standards vs Guidelines

## Design Principles


- Goal: provide designers with information and insights about the (potential) impact of their designs
    - Not a set of rules you must follow – too general and may not apply to an aspect of the interface
- Trade-offs: the more general the rule, the more chance it conflicts with another rule
- Rules are not exhaustive
- Vague distinction between:
    - **Guidelines**: vague, need to know theoretical underpinning to apply them well
        - General, have less authority
    - **Standards**: can be very specific (e.g. 3 button mouse)
        - Have more authority

## Standards

- Usually set by international committee
    - HW standards more specific than software
        - Because h-ware less likely to change
- Ideal: Large community following a standard leads to better technology
    - Don't have to focus on compatibility between technologies – can focus on other things instead
    - E.g. USB
- Unfortunately: not very useful for promoting usability
    - Too much specificity and standards

### Example: Android Standards

- Use the platform's built-in widgets and layouts whenever possible; these views provide accessibility support by default.
- Use the Options Menu as an alternative to touchscreen tasks
- Make sure the Back button correctly moves the user back one logical step in the task's back stack or the activity's back stack of fragments (when performing fragment transactions) as appropriate

# Guidelines

- Style guides published by Apple, Google, etc
- Tend to be generalizations – the more general, the earlier they should be in the design process
- Can range from:
    - Users must initiate all dialogs (Apple); to
    - Use white space between long groups of menu controls (like the Microsoft Office Ribbon)

## Example: Android Guidelines

- Use short names in the Options icon menu
- Place most frequently used operations first
- A Context menu should identify the selected item
- …to name a few!

# Mobiles

- Standards and guidelines for each mobile platform are important for making applications consistent within that platform, but they also have the effect of making the platform you are using recognizable at a distance.
- Don' evaluate apps/features by making a "checklist" and ticking off what guidelines the app/feature does follow
    - Tradeoffs as we try to meet those guidelines and the needs of the users
- Consider goals of devs, users and consider how guidelines conflict with one another as the app/feature is being designed
    - Find the overlaps and tradeoffs, as well as ways in which multiple guidelines can apply in a situation

## Summary Table

| Standards | Guidelines |
|---|---|
| ▪ Higher authority | ▪ Lower authority |
| ▪ Little overlap | ▪ Conflicts/overlap/trade-off |
| ▪ Limited application | ▪ Less focused |
| ▪ Minimal interpretation | ▪ Interpretation required<br>→ HCI background |

# Understanding users

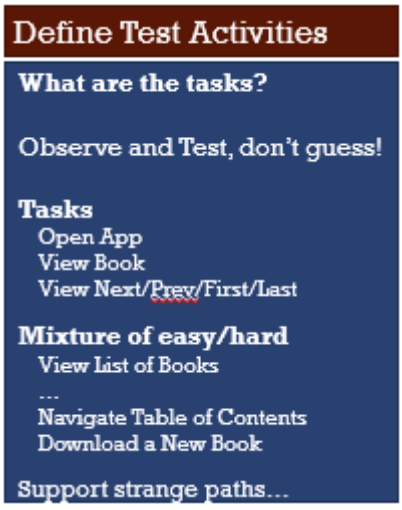
- Start design cycle by understanding users
- Importance of empathy in design thinking
- Design principles - can only take you so far in the design of your applications
    - . To design great applications, you need to understand users.

## PARTICIPANTS

- Who are your participants?
    - Who is the population of people you want to enquire from?
    - How do you know the people you choose represent the needs of all potential users of the app?
- Consider personas
    - Parents
    - Children
    - Teachers
    - Others?
- Consider context
    - Where and when will they use the app?
- Who can you get access to? How will you recruit them?
    - Need to consider these constraints especially in corona times

## TESTING

- What do you need to test?
    - Often, problems with the app aren't about whether buttons are obvious to people – rather, it's whether the concept itself is correct
- How do you test the flow and implementation of the app?
    - … vs. testing the layout
- Define test activities – i.e. what people will do as they test your app
    - Can define granularly (step by step) or broadly (e.g. "just explore the app")
- Example (for the book app):




## THE QUESTION

- While evaluating, think about what your question is.
    - This helps you to evaluate your methods to see if you question will be answered.
- Can have multiple questions that can be specific or general (or a mix)
- Some examples (for the book app):
    - Are buttons or swipe actions more natural for children using the app?
    - Does my approach to parental locks prevent children from accidentally exiting the app? Or turning the page? Or downloading a book?
    - What are user expectations around navigating a picture book on a mobile phone?
    - Is this app more "usable" than alternative app X?
    - Is this app "usable"?
    - Would you use the app?
- How to define usability?
    - Is it in terms of the # of mistakes people make when using the app?
    - Or the time it takes for them to progress through the app?

## A ROUGH APPROACH

1) **O**bservation
    a) Create an environment/a set of environments and methods with people that allow you to learn what's going on
2) **I**nquiry
3) **A**nalysis
    a) Analyse the data
4) **P**rototype
5) **R**inse and Repeat

Mnemonic: **O**nly **I**nteresting **A**nimals **P**eruse **R**eddit

## FOCUS THE STUDY

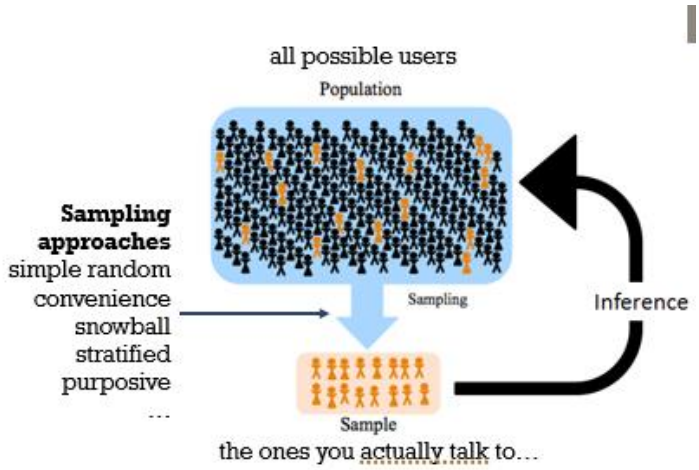
- Know the scope of what you're designing
    - How specific are the requirements, what constraints are in place?
- If you are less focused, you will leave room for inspiring design – looking for new ideas in a particular domain.
    - But a more focused study lets you gather information that will help you go in an authoritative direction.
- In earlier iterations of the cycle - look more for inspirations and broad ideas
    - As you progress, you may test specific interface elements or sequences
    - By the end you are testing for look and feel – and small things that can be changed without altering other parts of your application.
- Who are we interested in?
    - What is the profile (age, socio-economic background, skills, interests) of our users?
- What is the domain?

    - College students? Healthcare? Education? Leisure? Family reunions?
- What broad user-goals are we keen to support?
    - Collaboration? Diversion/fun?
- Are there any technology constraints?

# Recruiting Participants

- Some reasons why your friends and family might not be the best participants for understanding users:
    - Participant bias - won't necessarily give you their honest opinion; they won't take you seriously
    - Homogenity – the people around you tend to think like you
        - They may not give a good picture of what the general actual target population is like.
- It's not easy to
  1) find users and
  2) find users that will give you useful information.
- Your goal should be to find a representative sample – a group that typifies the larger group of your potential target users.

## REPRESENTATIVE SAMPLE




- Finding lots of people, selected in some principled way.
- The argument is that with a good sampling, the answers you get from a subset of the population can represent the answers of the whole population.
    - To make this work you need to have some percentage of the full target population.
        - More for less uniform populations.
- Can have different sampling approaches
    - Random samples vs convenience samples vs snowballing vs stratified sampling vs purposive sampling
        - Stratified = dividing population according to some features you're interested in (e.g. age group – have a few people from each grouping)
        - Purposive sampling – cherry pick people from a population to ensure that certain segments of that population are represented
    - Each approach has its strengths and weaknesses

## KEY INFORMANTS

- What to do when the sample size is small?

- Key informants can be used to recruit more participants or give info about the ways people in the population might respond
- **Field Guides** are people who know the lie of the land: individuals, groups and their dynamics, organizational politics.
    - Can act as intermediaries or be a literal guide into how people in the target population behave.
- **Liminals** are accepted by most people and move freely between groups (within the population). They are part of the action, but also can muse and reflect on what's going on.
    - Example (for the book app): a teacher who is also a librarian and parent
- **Corporate Informants** already spend a lot of time with your target users and provide insight as well as helping you find field guides and liminals.
    - Somebody who is good at finding participants and understanding what's going on in particular environments (professionally)
- **Exceptional Informants** are rare gurus that deliver "disruptive" insight into your topic
    - Experts in your domain of interest
    - Lots of experience- example (for the book app): someone who develops e-books for kids

### FINDING PEOPLE

- Off the street
- Advertising/recruiting
- Existing customers
- Colleagues
- Referral and Snowball Sampling

## Quantitative Approaches to Understanding Users

- Have set of users
    - How to generate feedback from them, which can be turned into design recommendations.

### QUANTITATIVE MEASUREMENTS

- Quantitative Measurements: Experiments to determine behavior in controlled settings
    - Often take place in lab/place where you can control environment to eliminate confounding variables.
- What are your metrics? Can't just measure "usability" – need "real" metrics
    - Time taken
    - Errors
    - Income generated
    - Data consumed
    - User Empowerment
    - …and more!
    - Important to understand *what* you're measuring and to have a tool that operationalizes that i.e. allows you to do a quantitative measurement of that metric
- What's your setting?
    - Are there any confounding variables?
- Do you have statistical significance?
    - Depends on variance and number of participants
        - Without the correct # of participants, may not be able to say anything definitive with quantitative measurements

    - Are your participants representative of the population?
- A/B Testing
- Likert Surveys
- Lab Experiments
- Data Logs

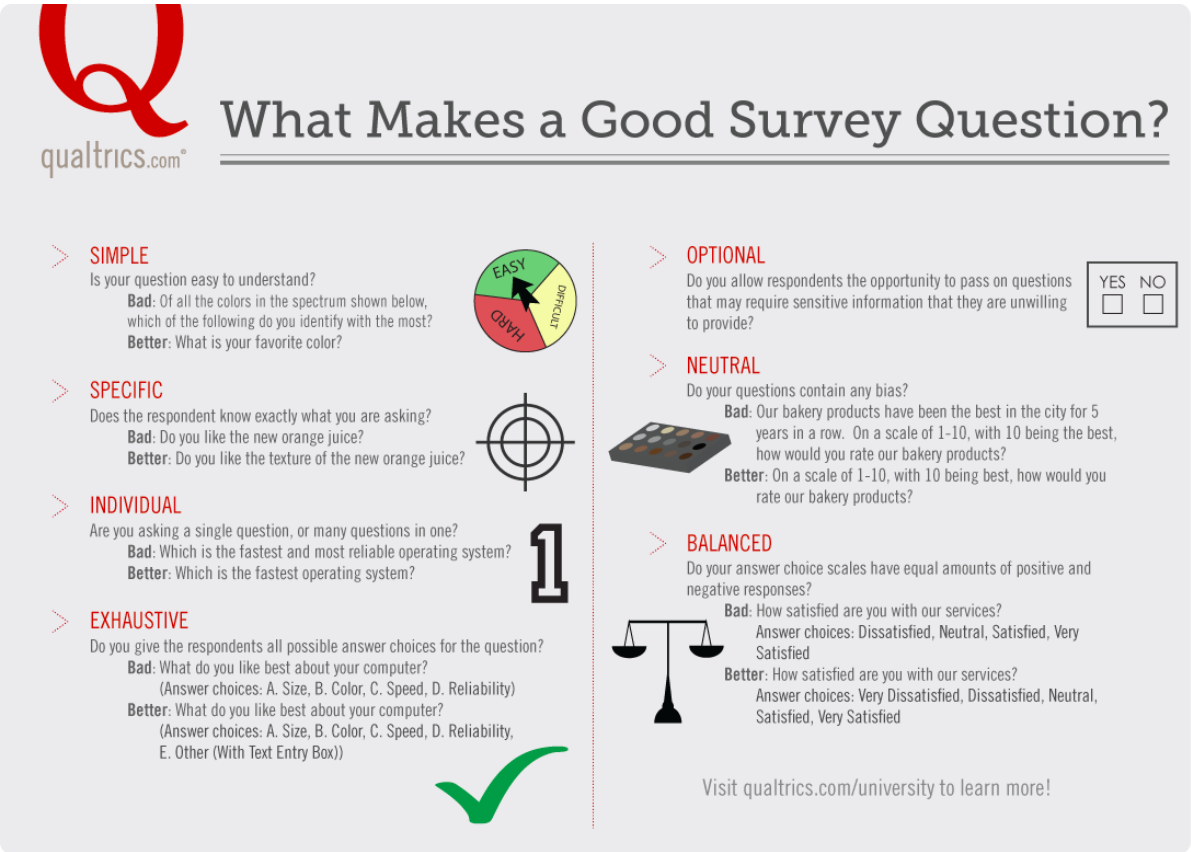
## QUANTITATIVE DATA

### Logging

- With mobile data collection, easy to do logging
- Instrument software to record interactions
    - Key presses
    - Mouse movements
    - Screen touches / clicks
    - Reading time
    - Eye tracking…
- Logs of interactions are time stamped
- Useful for
    - Web studies
    - Testing layouts, displays, interfaces
        - Can understand what users are paying attention to
    - Understanding reaction time
    - Measuring features most used, speed, performance
    - Measuring errors

### Surveys

- Numeric responses
    - How many times product was purchased?
    - How often one sends SMS?
- Likert scale responses
    - Odd (e.g., 3-point, 5-point, 7-point) allow neutrality
        - Like in Course evaluation forms
    - Even forces choice
    - Problem with Likert scale responses – how people view e.g. that 1 – 5 differs from person to person
        - Put labels on the scale to mitigate this
- Statistics can be calculated per-question and used in analysis
    - Mean, median, N, etc.
- Need to have **good** survey questions – bad ones mean you have no assurance that how the users have answered it is what they feel/what's actually going on
- Keep questions:
    - Simple
    - Specific
    - Individual – don't have two questions in one
    - Exhaustive – e.g. add "other" to an mcq
    - Optional – don't force users to pick an option if they don't agree with any of them
    - Neutral – the questions shouldn't drive people to a particular answer
    - Balanced – answer choices scales should have an equal amount of +ve and -ve options




## A/B testing

- Convenient – can use people already using your app
- A/B tests are ones in which users (typically website users) are presented with one of two versions of an interface.
    - Metrics such as click-through, purchases, time on page are then compared.
- Used by Facebook and others
- Example:
    - In this case, Performable wants to know what color button leads to more people clicking on the Get Started Now button.

## A/B test example: Button Color

**Green**
Green connotes ideas like "natural" and "environment," and given its wide use in traffic lights, suggests the idea of "Go" or forward movement. It also matches the company's logo color.

**Red**
Red is often thought to communicate excitement, passion, blood, and warning. It is also used as the color for stopping at traffic lights. Red is also known to be eye-catching. Red, in general, is not used as a button color nearly as often as green.


- Consider research design.
  - What is the independent variable? (In example, the colour of the button)
  - Dependent? (Click throughs)
  - Who are the participants, how are they selected?
- Research design ethics:
  - Have the participants been informed that they're part of the research study and how do they feel about it?

## The new quick and dirty..


**The red button outperformed the green button by 21%**

Over 2000 visits

"a few days" of traffic

- Results of A/B test (example):
  - Red outperforms green
  - In response, designers would change their buttons to red

### SAMPLE SIZE

- How many data points do you get for quantitative studies?
  - It depends on the distribution and variation of the effect you are measuring in the population.

- If you are very confident that the population is relatively homogenous, then fewer people are fine, but if there's a large amount of variation in the factors you are measuring, or even occasional outliers, then you will need more to get statistically significant results.
- You can estimate if you have enough participants using a power-law calculation.

### STATISTICALLY SIGNIFICANT ANALYSIS

- Once have data – analyse and interpret it
- Reporting descriptive stats like mean, median, mode and variance can be enough
    - To be more rigorous you will need to know more about your sample and to use statistical techniques to match your measures and your sample.
- Your outcome is only as good as your sample
    - If you have failed to get a representative sample, or even if you do something more sophisticated like a stratified or purposive sample, you can't use these stats the same way.
    - You can make statements about the subgroups you have selected against, and perhaps use more specific statistical techniques to analyze the results.
- Often use convenience samples and snowball samples.
    - Consider how well these represent the population.
    - There is biases inherent in most sampling techniques.
        - Need to account for this in your analysis and not just assume that because you have concrete numbers you can make declarative statements about the population as a whole.

## Qualitative Approaches

### QUALITATIVE OBSERVATIONS

- Think about the contexts in which you imagine your app will be used; and go and observe to see how people interact with their phones in those contexts.
    - Takes notes of what you observe, as well as what you're seeing/not seeing
- Explaining Phenomena and Thick Description
    - Thick description: provide description of their context, how they're feeling, how they're doing this work
        - Details help give your perspective to others
- Identify Values, Goals and Actions
- Understand the Context in which they're working and how it might affect use of a mobile app
- Track:
    - Tools
    - Documents that they consult/using to support their work
    - Emotional Setting in which they're using their mobiles
- Techniques:
    - Interviews
    - Contextual Inquiry
    - Cultural Probes
    - Focus Groups
    - Workshops

### INQUIRY

- Contextual Inquiry:

        - Ask them to do a particular activity so that you can observe them as they do it
    - The designer watches the participant begin an activity
    - The designer interrupts to question the reasons behind the activity
        - Try and better understand their reasons and why they're choosing the actions they're choosing
    - The participant provides commentary as they work, explaining while doing (think aloud)
    - Assume a Master-Apprentice model – the participant is the master of doing what they do and you're the apprentice trying to learn how to do what they're doing
- Diary Studies
    - Have participant record their activities at a regular interval
        - Through photos, written work etc.
    - Can be automated using software on phones, or interactive
    - Take all that data and analyse
- Interviews
    - Structured (survey) → Semi-structured (have set of questions, don't have to ask strictly those questions, more open-ended questions) → Unstructured (a conversation with a topic(s))
    - Prepare interview questions, a "guide", especially for semi-structured and unstructured interviews
- Cultural Probes
    - Capture aspects of daily lives or what was important to them
    - Capture aspects of culture that wouldn't have necessarily been revealed by traditional means (interviews/direct observation)

## FOCUS GROUPS AND WORKSHOPS

- Focus groups is more like a group interview
    - Workshops are more involved, with more activities
- Choose your participants
    - 3-7 for a focus group
    - more for workshops
- Prepare a guide
    - Outline questions and activities
- Choose your location: your place or mine?
    - Where will they feel comfortable?
- 30m-4h session led by a trained moderator
    - Workshops can be longer, from hours to days
- Follow up with analysis and written report
- Useful for brainstorming possibilities (vs. critiquing concepts)
- Can be a contained environment for presenting prototypes, or inviting potential users to sketch their own

### Focus Group Moderator

Some Tips

1. Observe/watch other moderators in action first!
2. Facilitate a natural conversation, make people comfortable, help them to share their insights
3. Dig deeper, ask open-ended questions
4. Direct participation, provide opportunities for everyone to have a voice, invite people to participate in the conversation

5. Moderate, don't participate!

- **Characteristics of a good moderator**
    - Prepared | Listens | Neutral
    - Flexible | Patient | Observant

## OBSERVATIONS

- Lots of different ways to make sense of your observations
- Affinity mapping – effectively sorting the ideas out into categories and subcategories.
    - Then you'll prioritize by taking those ideas and interpret them into an empathy map.

## ANALYSIS

- Process of analysis has four main steps
- Transcribing
    - Transcribing recordings yourself helps you to familiarize with what people say.
- Categorizing and organizing ("Coding")
    - Look through those transcripts noting, or coding, main ideas
    - E.g. highlighting and "tagging" notable quotes and responses
- Reflecting and interpreting
    - Start taking those codes, or categories, and you sort and prioritize them
- Making consumable ("Implications for Design")
- Analysis usually follows these steps
- Sometimes categories are emergent, based on the ideas and themes that occur frequently in the data.
- Or perhaps you have a particular theory or set of questions you are interested in, and you can define the categories and codes ahead of time.
- Noting frequency, variations on a theme, and priority of ideas helps you to systematically understand your data to identify phenomena in your research.
- As you uncover patterns, you'll find yourself better able to understand users and explain their approaches and reactions to interfaces.
- Rinse and repeat

# Prototyping

- Just building the app doesn't work
- One of the key concepts in HCI (user-centered design) is the design cycle
- Starting with an initial needs assessment and design, you can build successfully better prototypes, iteratively, based on the evaluation of the prior one.

## EVALUATION

### Understand your users

- But how do you bootstrap that first prototype?
- Start by spending some time understanding your users
- Try to move beyond sympathy, in which you appreciate their experiences at a distance, to empathy, where you can identify with their situation and feelings, that their pains are your problem too.

# Design

## CREATE PERSONAS


- Based on that initial evaluation you analyze these results and come up with implications for your design.
- One method of analysis is to develop personas that characterize perspectives of different users.
- A good persona:
    - Name (a fictitious one), so they are relatable, perhaps a catch phrase to help capture what makes them unique.
    - Need specific goals.
    - They have specific characteristics – what phone they use, how often they use it, how much data they use regularly, favorite apps, brands.
- More details helps you, and people you report to, better empathize with the projected user.
- Easier to generalize about fictional people
- General users have too many conflicting goals
    - Specific personas have clear, well-articulated goals.
- Why personas?
    - The general "user" is hard to identify with – and can be confusing – goals for expert and novice or infrequent users may be quite different.
    - Personas help to capture those differences and to clarify your design plan.

## DEFINE THE PROBLEM

- Focus on the problem
    - choose the appropriate framing
- In that first iteration you aren't actually evaluating an existing prototype – you're evaluating your problem statement
    - Use insights from your interactions with people to reframe that statement, so you can start to ideate.
- A good problem statement has the right framing
    - Mistake: people insert their preconceived solution into the problem.
        - To take a step back and capture their real need.

## Ideation

- Stretch mental muscles
- Prepare some materials
    - Get physical
        - Sketch
        - Make models
        - Act out
    - Capture interactions and relay thoughts.
- Capture ideas as they come, somewhere very visible, so everyone can see them and build on those ideas.
    - But also, stay focused on that problem statement you worked so hard to frame right.
- Key to brainstorming is that ALL ideas are good
    - Don't negate anything, but continue producing more
    - Aim for quantity of ideas – and defer judgement about whether ideas are realistic until later.
- Brainstorming is better in a group, with a trained moderator

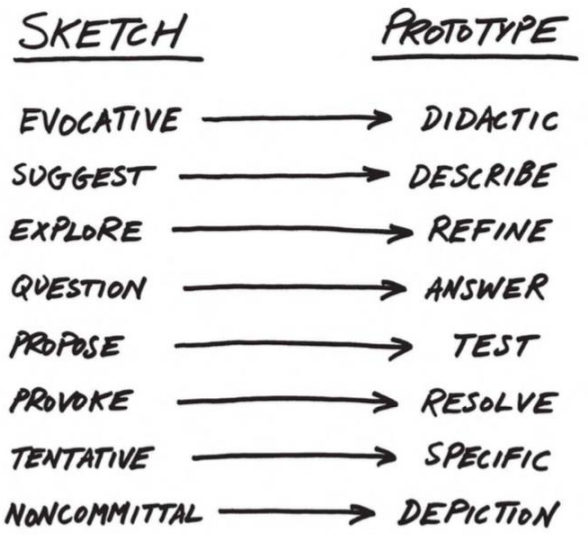
### SKETCHES

- Technique for brainstorming
- Clear vocabulary – lines extend through endpoints – clearly a sketch
- Needs to be just enough to communicate intended purpose/concept (not super high res)
    - Resolution of the rendering does not exceed the degree of refinement of the concept
        - Don't add in unnecessary details
- Sketches intended to be ambiguous – use to capture ideas that you haven't committed to.
- Take ideas for prototype and create quick sketches
    - Makes it easier to throw away – haven't committed a lot of resources to making them
    - Because it takes few resources – can create lots
- Have a series of sketches – as the design progresses, more detail might be added
    - Idea gets refined as more detail is added

### TIPS

- Don't stress about quality or if you can draw a straight line, stick figures are totally fine!
- Use squiggles instead of text, boxes instead of pictures
- Don't think, just draw (fast) – give yourself a time limit
- You don't have to finish every picture!

## Sketches vs Prototypes




- "The goal of sketches is to generate ideas and to explore, mostly for yourself or your team (which may include some user representatives if you are co-designing.) – in contrast prototypes capture and describe ideas that you want to test."
- "While it may feel like the two are interchangeable, especially for lo-fi prototypes – it's helpful to distinguish sketching and prototyping as two separate processes, two stages in the design cycle."

# Prototype

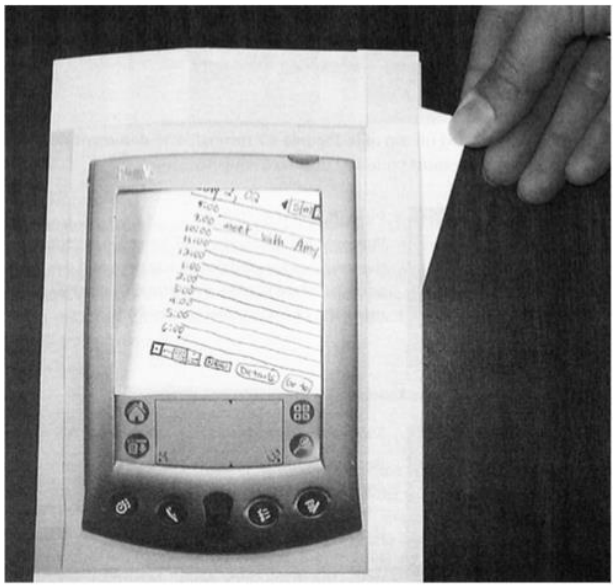
- Build something tangible so you can get **feedback**
- Experiment with alternative designs (cheaply!)
    - Can build lots of prototypes and get feedback on each one
    - Don't have to commit to any one as you progress in design cycle
- Fix problems with concept and flow before code is written
- Keep the design centered on the user!
    - Think about the user as you're developing
    - Prototyping allows you to get feedback from the user
        - Your mental model of what they need should match their mental model of what they need

## TYPES OF PROTOTYPES

- Lo-Fi to Hi-fi
    - Manual vs interactive

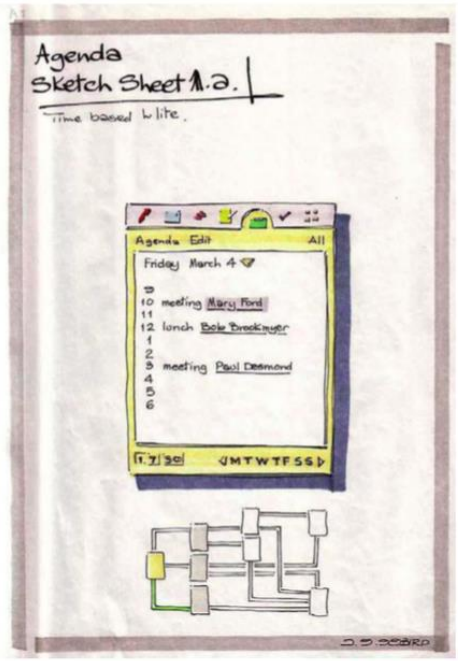
## Lofi

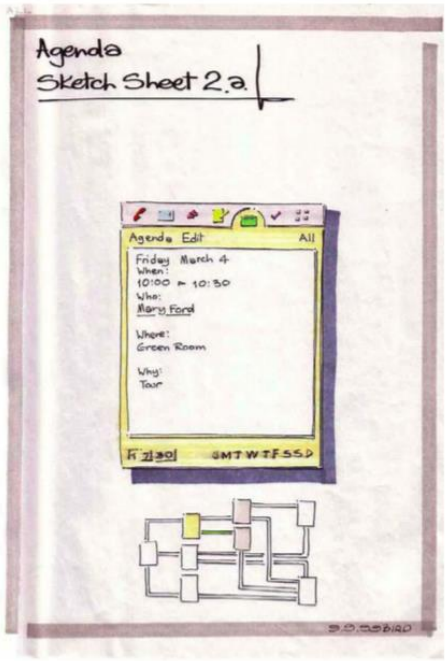
- Tracing/Overlaying
    - Technique for creating lofi prototypes
    - Take phone/screen and trace it
        - Able to present what you think it might look like



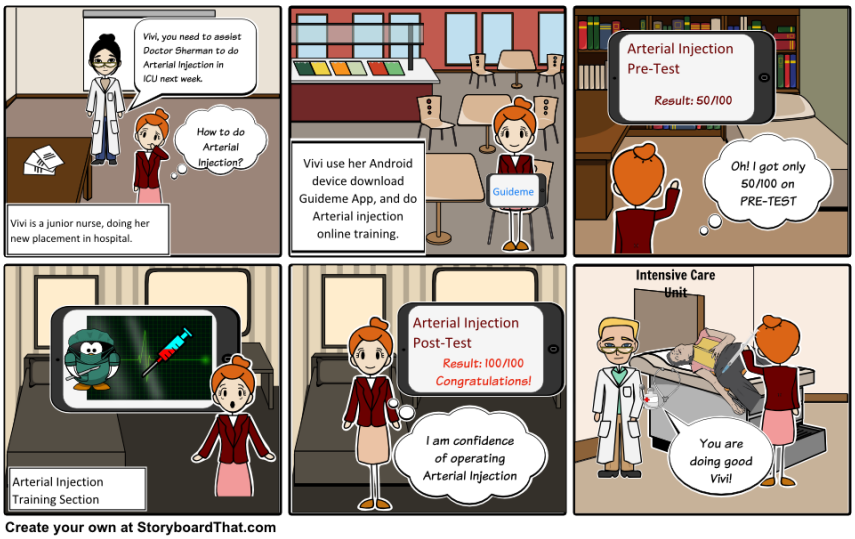
## State transition

- Have to show how people transition from one state to another
- Use transition diagrams/state diagrams helps communicate transitions i.e. the relationships between the different screens



## Storyboards




- Can place the app itself and the environment where it's being used.
- Can see how people are thinking and the context in which they approach the app, as well as how they're using it
    - Can take the empathy map and make it onto a storyboard


- Space between screens is important – it's where users might struggle
    - Users e.g. might press a button and have expectations of what is going to show up next (how the interface will react to that action)
        - That moment of encountering what they expect is what you want to observe
        - Useful to see how they're reacting to a screen and what they think individual things are – but the touchpoints are how they transition from one state to another within the app

## Tangible Materials: Lo-Fi

- Heavy white paper (large!)
- Index cards
- Post-it Notes
- Tape, gluestick, correction tape
- Pens & Markers
- ….and more!!

- Use it to capture ideas, which can then be conveyed to other people so that you can get feedback

### Digital Materials

- Digital materials can be printed and then physically cut and pasted, or you can use prototyping tools like invision or even powerpoint to create prototypes without coding.

### Constructing the Prototype

- Set a deadline
    - Don't think too long – build it!
- Draw a window frame on large paper
    - Draw larger – but keep the same aspect ratio
- Put different screen regions on cards
    - Anything that moves, changes appears/disappears
    - Use squiggles to indicate text if necessary
- Ready response for any user action
    - Pull-down menus
    - Screen transitions
- Use photocopier to make many versions
- Can also make interactive digital prototypes
    - InVision
    - Sketch
    - Proto.io
    - Powerpoint/Keynote
    - FluidUI
    - … and tons others


### Hifi

- Paper fosters collaboration and engagement with subjects, it's versatile.
    - Screens tend to be single user.
- Distort Perceptions of User
    - Formal representation indicates "finished" nature
    - People comment on color, fonts, alignment
        - Rather than on the actual flow or functionality
- Discourages major changes

    - Testers don’t want to change a “finished” design
        - The more “polished” it looks, the less willing people are to provide feedback about the underlying structure and features that have been included/excluded
            - Will rather comment on things they think can easily be changed
        - Participant bias
    - Designers don’t want to lose effort put into the design
- Lofi allows you as a dev to not commit to anything - your potential users can then provide feedback on lofi prototypes which would, in turn, help you to commit to an idea.
    - Users need to understand what kind of feedback they need to give you

# Evaluation

- Once you've developed a prototype – how to get feedback?
- Don't necessarily get it right on the first try
- Importance of UC (user-centered) design is that other people can help you find problems/identify things you may not have thought of

## GENRES OF ASSESSMENT

- **Automated:** Usability measures computed by software
  - Limited (in scope) in what they can actually measure
- **Empirical**: Usability assessed by testing with real users
  - Qualitative or quantitative
  - Setup a context in which you can gather findings from real users and analyse it
- **Formal:** Models and formulas used to calculate measures
  - Don't use real users
  - Build on prior experiment with real users and models that have been developed
  - Use formulas and models to see if that design has met that model and to evaluate whether/not you've met your objectives
- **Inspection:** Based on heuristics, skills, and experience of evaluators
  - Get another expert to come in and evaluate what you've developed
  - Based on their expertise and experience in their domain, they're able to give you useful feedback (which can be taken and used to iterate on your designs)

## QUANTITATIVE TESTING

- Costly in terms of time and money
- Have to setup whole experiment
- Have to setup a good survey
- User Compensation
- Time spent analyzing data
- If study is more sophisticated
  - Usability Expert Person-Hours
  - Specialized Equipment and space
    - E.g. eye tracking

## QUALITATIVE TESTING

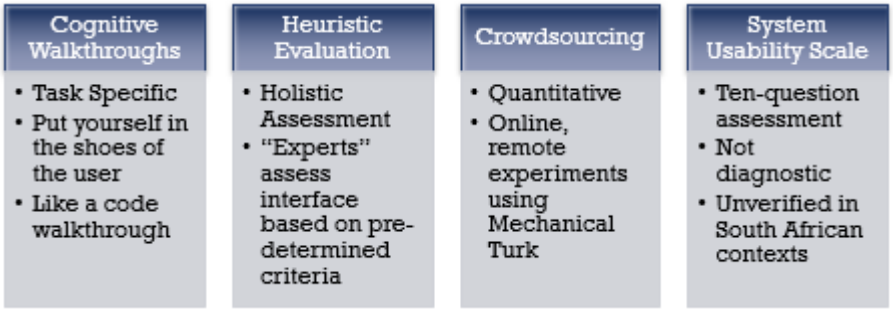
- Can be costly in terms of time and money
- Take lots of time but don't always lead to clear outcomes or implications for design.
  - Hard to do well.
- Ethnography
  - May take years to fully understand the context in which you're using things.
  - A branch of anthropology and the systematic study of individual cultures. In contrast with ethnology, ethnography explores cultural phenomena from the point of view of the subject of the study. (Wikipedia)
- Transcribe and Analyze (weeks)
- Need training (years) to be a good qualitative researcher
- Qualitative methods are valuable but hard for the initial stages of an app
  - Great for apps like MS Office – been around for years, previous versions of app have been integrated into people's lives (and don't have to worry about release date)

- All these stages need lots of time

## Discount Usability Techniques

- Often used in industry
  - "Quick and dirty" Usability Techniques
- How is "discount" usability different?
  - All methods simplified
  - Goal is not to get statistically significant data or a perfect picture of what's going on
  - Get like 80% there
- Scenarios, Simplified Thinking Aloud, Heuristic Evaluation
- Simpler methods are more likely to be used
- Pareto principle applies to research participants too!
  - Pareto principle: The Pareto principle states that for many outcomes roughly 80% of consequences come from 20% of the causes.
    - i.e. 80% work takes 20% of the time – last 20% of work takes 80% of the time
  - For studies – as you get more users, each will give you only a small amount of new info
- Has been determined you can do discount usability techniques with 3 – 5 users and get pretty decent results
- Cheap
  - No special labs or equipment needed (e.g. use notes instead of video)
  - The more careful you are, the better it gets
  - Focus on benefit-to-cost ratio
- Fast
  - Can often be done in one (1) day
  - Fewer "testers" involved
- Easy to use
  - Can be taught in 2-4 hours
- Focus:
  - Not on getting "research quality" data, but looking at the benefit-to-cost ratio
  - How much are you investing in the evaluation and how much benefit do you think you can get with it

### TECHNIQUES




- Cognitive Walkthrough
  - "Cognitive walkthroughs involve simulating a user's problem-solving process at each step in the human- computer dialog, checking to see if the user's goals and memory for actions can be assumed to lead to the next correct action." (Nielsen, 1992)
  - In discount usability scenario - get 3 to 5 people

    - See how they do things i.e. walkthrough your design
    - Have your user/yourself walkthrough how they would use the app
- Heuristic Evaluation
    - Take e.g. Golden rules and use it to explain what's going on with the app
    - Get 3 -5 experts to help – give evaluation using heuristics to help you explain their responses to the app
- Crowdsourcing
- System Usability scale
    - Weaknesses: if there's something wrong with the system, doesn't tell you what is wrong
        - Hasn't been verified in the SA context (English not everybody's first language, etc)
        - Nuances in 10 questions may not be picked up by evaluators
- Choose right discount usability technique for the context that you're working in and for the purpose of what you're doing

## Scenarios

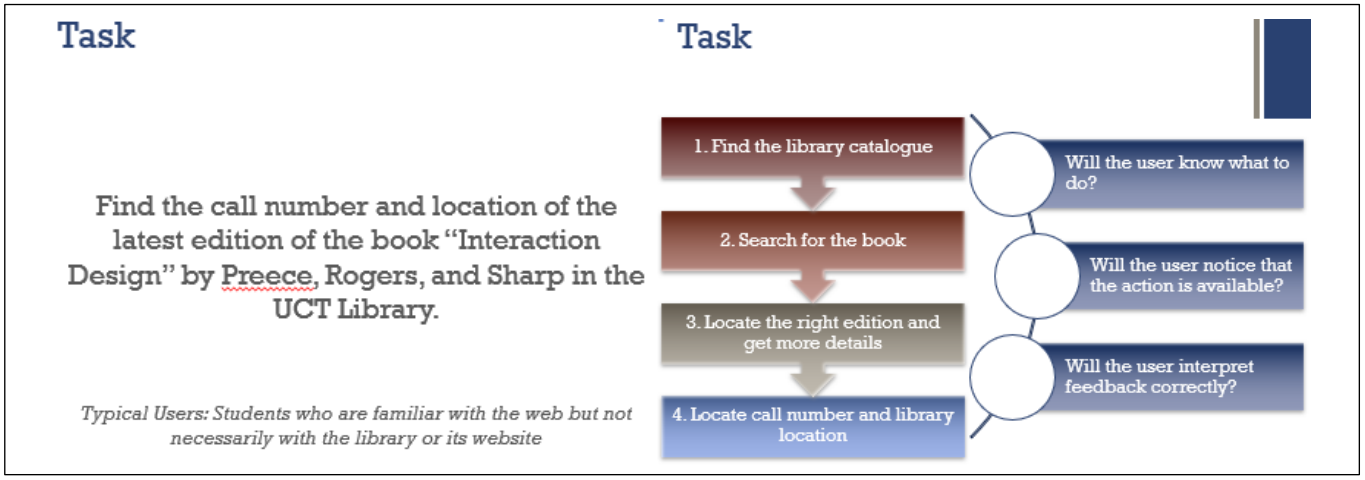
- Prototype around specific scenarios – not the whole system
    - Build prototype around specific scenarios – enough prototype so that users can enact those particular scenarios
- Reduce the scope of the evaluation to specific use cases
    - Easy to create and throw away
- But the user must stick to the planned path or it breaks! (weakness)
- Example:
    - Melanie has used an app to create alphabet book with photos of her family members and everyday objects corresponding to each of the letters. She wants to share her creation as an ebook for her 3-year old niece.
    - How to implement?

## Implementation

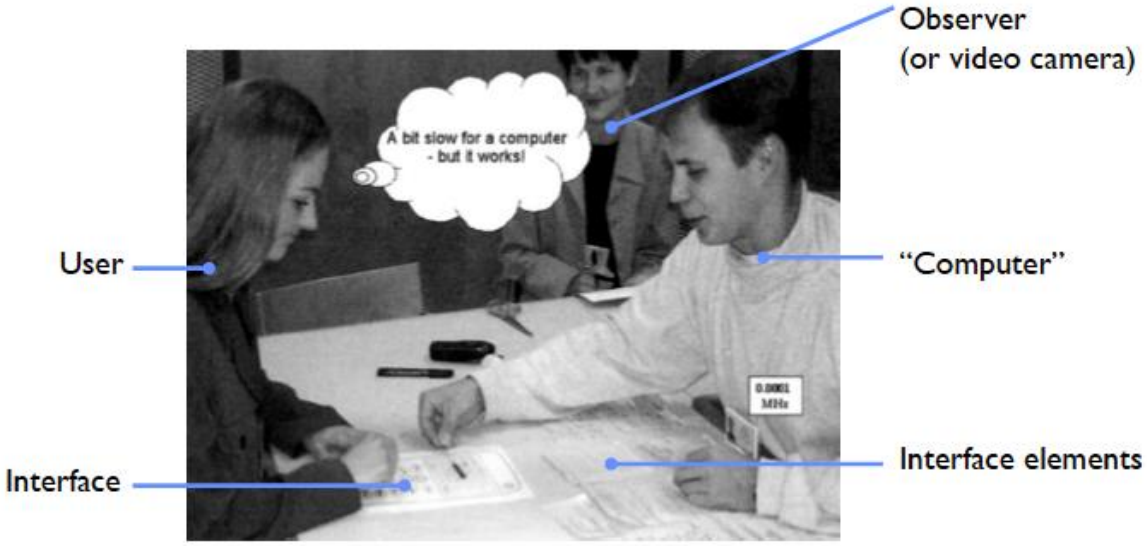
- Task
    - A detailed task with a concrete goal
    - Action sequences for completion of task
        - Need to implement enough of prototype for those action sequences to be completed
- Ask
    - Will the users know what to do?
    - Will the user notice that the correct action is available?
    - Will the user interpret the application feedback correctly?
- Record
    - What would cause problems?
    - Why?

- Give users a prototype and scenario, ask them as they go through the scenario to talk aloud, say what they're thinking and seeing, and how they're making a decision about what to do
    - Once they've completed an action – why they did that action?
    - Throw an error if necessary – see how they respond, are the errors being understood?

Example (existing interactive prototype)




## SE WITH PROTOTYPES




- Need to take "Wizard of Oz" approach with cognitive walkthroughs that have prototypes
- "Wizard of Oz" setup
    - Need at least 3 people – user interacting with prototype, person who acts as the computer, observer
    - Interface elements and interface

### Conducting a Woz Study

- Motivation: You have several scenarios you'd like to test, but you haven't yet implemented the backend.
    - Good for: **speech recognition, early prototypes, queries**

- Solution: a person reacts to user “input” by responding with the appropriate action in the prototype

1. Decide on a scenario and list of tasks
2. Implement a prototype that captures the scenario and possible branch points for the user
3. Designate the “computer” and the “observer”
4. Give the list of tasks to the “user”
5. Computer drives the user interface (can’t help people – let them be stuck), observer takes notes
    - Can’t give steps to user if they get stuck
6. Don’t forget to ask the user to THINK ALOUD

### Scenario and Task

- Provide only high-level instructions and scenario to the user:
    - This is an online shopping app. You already have an account with a credit card stored. Purchase some Prestik.
    - Don’t forget to think aloud!

  vs.
    - Enter “Prestik” into the search window. Select the Prestik and then purchase it by clicking the buy it now button.
- High level tells them what the goals are, but not what the expectations are in terms of the interactions with the UI
    - The other gives step-by-step instructions – thus unable to test whether/not they would intuitively know how to do certain things.
- In order to check whether UI design is working and will continue to work without giving explicit instructions to people, then don’t give them instructions as you’re doing the test.

## Outcome

- Each assessor prepares a list of issues
    - When did it occur?
    - What happened?
    - Severity
- Compile all reports into one
    - Note issues that occur frequently (might be more severe)
    - Prioritize the issues to be addressed
    - Give that feedback back to dev/yourself

# Usability Heuristics

- Similar to Golden Rules of Design
- Can be used as design principles
- Can be used to evaluate a design
- Pros
    - Easy & Inexpensive to do
    - Performed by experts – may be easier to access than users
    - No users required to do heuristic evaluation
    - Catch obvious design flaws
- Cons
    - Not just a checklist
        - Don't treat heuristics as a checklist – even if it has met all of the requirements, it may still not be quite "right".
        - Treating the heuristics as a checklist in order to deem an app usable is insufficient
    - HCI Experts might not know context in which app will be used
    - More difficult to do correctly than it seems!

## Heuristic Evaluation

- *Can be performed on working UI or on sketches*
- Small set (3-5) of evaluators (experts) examine UI:
    - check compliance with heuristics
    - different evaluators find different problems
    - evaluators can communicate afterwards to aggregate findings
    - designers use violations to redesign/fix problems


- Visibility of System status
    - Modality
    - E.g. if have an app that can be used both online/offline – need to then know when online/offline
- User Control and Freedom
    - Giving user's control over what they've done – being able to repair and go back and fix a mistake
- Aesthetic and minimal design
    - Don't show irrelevant info
    - The more you have on a screen, the harder it might be for the user to find things they need
    - Prioritise the features they're going to be using most of the time
- Flexibility and efficiency of use

    - Catering for diversity of users
    - Allow there to be multiple paths to complete the same task
    - Shortcuts can improve experience of using app if implemented obviously
- Help and documentation
    - Want to make help and documentation easy to find and search
- Match between system and real world
    - Let users be able to take what they know and have seen in other contexts and use that in the system as they navigate it
    - Caveat – people's worlds are different
        - Conventions familiar to one user may differ to others
- Error prevention
    - Prevent problems from occurring in first place
    - E.g. preventing typing numbers in a text only field.
- Consistency and standards
- Recognition rather than recall
    - Don't want users to have to think before they have to do – should be able to look at it and immediately recognise what the next step might be
- Help users recognise, diagnose and recover from errors

## CONDUCTING HEURISTIC EVALUATIONS

- (1) Pre-evaluation training
    - Provide evaluator with domain knowledge and context, if needed
        - Who are the target users, scenario etc
    - Provide a list of heuristics for reference
- (2) Evaluation
    - Experts evaluate interface as individuals
        - Give interfaces, background and task lists to those experts
        - Compare interface elements with heuristics
- Work in two passes:
    - 1. Get a feel for flow and scope
        - Walkthrough the parts of your interface and compare those with heuristics
    - 2. Focus on specific elements
        - As they run across problems
- Each evaluator produces a list of problems
    - NOT producing list of heuristics with problems next to them
    - Explain why this problem is a problem with reference to heuristic or other info
    - They're trying to meet a heuristic and they've counter balanced with a conflicting heuristic, but the balance isn't right
    - Be specific and list each problem separately
- (3) Severity Rating
    - Bring evaluators together
    - Establish a affordnacranking between problems
    - First rate individually, then as a group
        - Give each problem a severity rating
    - "How important is it that this be fixed?"
    - "Which problems should we fix first?"
- (4) Debriefing
    - Evaluators discuss outcomes with design team
    - Suggest potential solutions
    - Assess difficulty of fixing problems
    - Every problem then has a severity rating and a difficulty rating

## SEVERITY RANKING

### Example:

| | |
|---|---|
| 0 | Violates heuristic, but is not a usability problem at all |
| 1<br>cosmetic | Need not be fixed unless time is available |
| 2<br>minor | Minor usability problem: fixing this should be given low priority |
| 3<br>major | Major usability problem: important to fix, should be given high priority |
| 4<br>catastrophe | Usability catastrophe: imperative to fix this before product can be released |

- Used to allocate resources
- Estimates need for more usability efforts
- Combine frequency, impact, and persistence
- Should be calculated after all evaluations are in
- Should be done independently by evaluators

## DEBRIEFING

- General discussion with evaluation participants
- Brainstorm improvements to address major problems
- Development team rates difficulty of fixing problems
- Minimize criticism
    - Have clear, neutral point of view – aim is to improve interface and make it clear for everybody (not just devs)

# Pros and Cons

| Heuristic Evaluation | User testing |
|---|---|
| 1-2 hours per evaluator | Days or weeks of working with users to get useful feedback |
| Findings are straightforward | Requires interpretation of actions |
| Setup so that you have clear action items at the end of it | Far more accurate: takes real users and tasks into account<br>- But takes more resources – alternate quick and dirty evaluation with user testing<br>o Not doing full scale user testing on every iteration, but able to make incremental improvements before going back to users again |

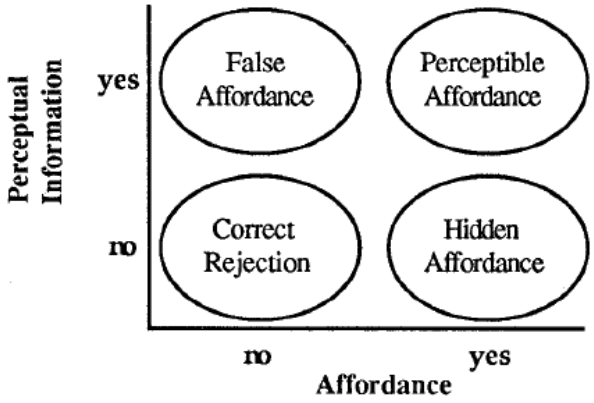
| | |
|---|---|
| May miss problems or find “false positives” (problems that aren’t real problems for the target audience) | |

# Everyday Design

## Affordances


- Being able to look at something and knowing that you'll be able to do something with that object.
    - For fidget cube: can pick up and hold in your hand, the functions it has.
    - The interpretation of what we see tells us what we can do with that object.
- Affordances are the potential actions and interactions that the environment offers
- Physical objects that have been designed specifically for us to interact with them
    - Includes mobile interfaces

### PERCEIVED AFFORDANCES




- Perceived affordances of an object are those properties of the object which give users clues as to how the device is used
- Perceptible affordance is the sweet spot – you can see it; you think you can do it and you can actually do it.
    - False affordance – you see it, you think you can do something with it, but it doesn't actually do what you want it to

        - Example: door that you think should be pushed but must actually be pulled
            - In mobile design – you get an iPad but don't know how to switch between apps; there is only one button on the tablet but it's unclear what it does
                - Button is overloaded (single press vs long press)
- Knowing what you can do and having that expectation there, is hidden until you know all the mappings and learned all the affordances and mappings of a particular device
    - The more affordances are hidden, the less learnable your app will be
- Have hidden affordances – things your app can do and features it supports
    - But it isn't obvious to the user how to do it
        - Disconnect: what's obvious to the designer may not be obvious to the user
        - Deliberate: it's actually an expert feature for people to use (can use a learned affordance from something else)
        - Easter egg
- Correct rejection:
    - They don't think that you can do it and you can't do it.
- For every feature that you have, you want how you enact that feature to be a perceptible performance
    - "How are they going to know how to interact with that feature?"
    - "How are they going to know how to do it?"
    - "How will they know what will happen when they interact with that feature?"

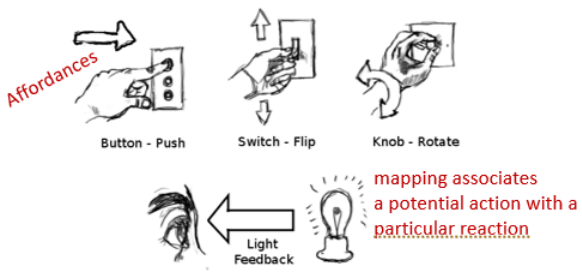

- Have the idea that you can combat and fix misunderstandings about perceived and hidden affordances by giving lots of instructions
    - But more instructions tell you that something isn't well designed
- Most apps don't come with user manuals – maybe have a short tutorial
    - Don't want user to be distracted by reading instructions instead of interacting with that app – that becomes an exit point
        - You lose the user
    - So, you design things that don't need instructions

## VISUAL AFFORDANCE


- Works by using familiar idioms and metaphors
- Leverage what's familiar to people so that when they encounter your designs, they'll be able to carry what they know (from the real world) into how they interact with their app
- Function over Form
    - You can take an idiom too far
        - Make it too much like the physical, real world object you're modelling it after, even though you're only working with a touchscreen.
    - Example:


- In example: you may be unsure whether some things are buttons or labels
    - Because it is possible, stuff gets overloaded – labels overloaded with actions and responses
    - Dialing with a mouse is not as easy as dialing with your fingers
- Visual Affordance Problems
    - Is it intuitive? Is it an affordance you would have picked up on, or is it hidden?

## Mapping




- Goes hand in hand with affordances
- Once you know *how* you may interact with a particular device, also want to know what is going to the be response once you take that action

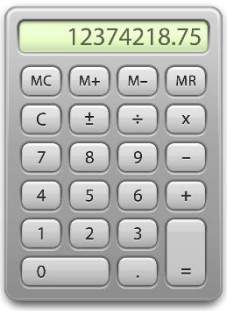
- Mapping: ensure a natural correlation between objects and the interface controlling them
    - Or between the actions and the interfaces
- In terms if affordances e.g. you can flip a switch
    - If you look at the objects, e.g. the switch, it will turn on a light
    - Immediate feedback (light turns on)
    - Helps create mapping in our heads between the switch and the light
- Affordance is the interface that is controlling the object and the mapping is the correlation/connection between the object and the interface controlling it
    - mapping associates a potential action with a particular reaction

## INSTRUCTIONS

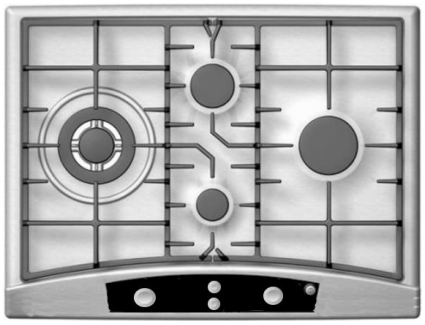

- Instructions are a sign of bad design
    - Weird intuitions and mappings that may come into how people respond to the designs we put before them
- Really need to understand how people are going to interact with your objects – hence the need for usability studies

# Objects vs Interface

- "Direct Relationship between artefact and interface controlling it"
    - Mappings capture not possible interactions (i.e. the interface) but the relationship between the interface and the artefact – the things that it does, or possible responses to your interactions.
- Example (calculator):
    - Have the affordance of pushing the buttons
    - The mapping tells you that pushing a button e.g. 1 , will display 1 on the screen




- Want recognition rather than recall with regards to mappings



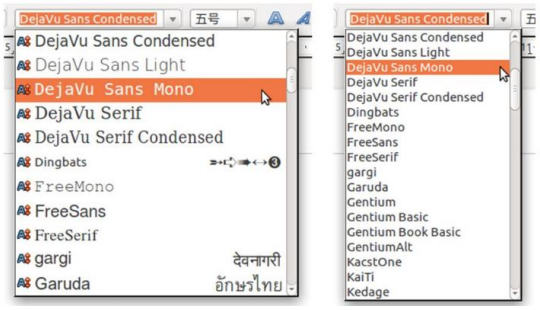
- How do you map afforded actions to expected reactions? Are these learned, cultural?
    - For a stove: turning the knob increases and decreases temperature
- Control Simplicity


- Having less than ideal experiences with an object because the design is too complicated for no good reason
- Knowing what will happen when you take a specific action, taking advantage of an apparent affordance, is effectively connecting that affordance to a consequence, or a mapping.
- Mapping and Modalities (e.g. mute button)
    - Modalities and the way icons change communicate modality to us, but also communicate state

# Design Principles

- Schneiderman's Golden rules of design – give an overview of important rules for design
    - Overlaps with Nielsen's 10 usability heuristics – work both as design principles and as a tool for doing heuristic evaluation

## Recognition Over Recall




    - Make objects, actions, options and directions visible
    - User should not have to remember information from one screen to another.
        - eg. Give input format, example and default
    - Example: One Time Pin (OTP)
        - Whatsapp reads you messages for you when you get that OTP sent to you
    - Users should not have to memorise a sequence of steps in order to navigate your interface
        - There should be correct affordances and mappings that allow them to recognise what the next right step is

### KNOWLEDGE & CHUNKING

- To improve on memory we tend to **chunk** actions
    - Group actions into a lump
    - Seek for meaningful relationships
    - 0796727796 vs 079 672 7796 vs 0 796 727 796
- To help people create meaningful relationships out of data they have to navigate (data may be random), we need to differentiate "knowledge in head" and "knowledge in world"
    - "knowledge in head" = memorising and trying to recall
    - "knowledge in world" = don't have to memorise action to go forward, can recognize patterns that are there and use info in the world to help you figure out the next step
    - Display-based action (e.g. response to touch) - "knowledge in world"
    - Recognition vs. Recall

## Pareto Law

- 20% of functionality will account for 80% of usage
- 80/20 Pareto Law
- The most important features should have the better real estate on the screen
- Difficulty comes in trying to guess what the 20% will be (as a designer)
    - Planning for the 20% and prioritizing it so that their experience is better 80% of the time, is important

## Principle of Least Effort

- Zipf's principle of least effort:
  - **Make frequent things easy, and unlikely things harder**
- Relates back to Pareto Law
- Similar to the simplicity idea, this manifests in the following ways:
  - Morse code's "E" is only one dot; apostrophe is 6 dots and dashes
  - Menus are organized with common things at top
  - "Dangerous" operations can be heavily nested or require many clicks and presses
    - Like deleting an app
- Designing to minimize errors

## Principle of Least Astonishment

- Reflect inner workings or user intuition?
- **Consistency** is a key goal in user interface design
  - How to measure how consistent you're being?
    - E.g. users should not be surprised by the action that results from clicking a button
  - Ensure that designs are reflecting user intuition about what those designs should look like (rather than internal implementation of that system)
- Applies to both **functionality** and **form**
- "People are part of the system. The design should match the user's experience, expectations, and mental models." (Saltzer, J. H.; Kaashoek, Frans (2009). Principles of computer system design: an introduction. Morgan Kaufmann. p. 85)

## Aesthetic Usability Effect

- Things that look better perform better in usability tests!
- The attractive product will be perceived as easier to use.
- Ease of use is often a criterion in purchase decisions – easy to use products require less training and support.
- So, by improving the attractiveness it increases the perceived ease of use – improving the chances of making a sale
  - Users will be more likely to develop positive feelings towards the attractive product.
  - This can lead to:
    Positive reviews – leading to more sales. They'll tell their friends – resulting in more sales leads
    They'll tolerate faults more – reducing support calls
    They'll be more patient with the app
- The attractive product will be perceived as of higher quality
- Most importantly: Customers may overlook feature deficiencies, so they get to use the more attractive product
- Apparent Usability vs Inherent Usability
  - Study showed that usability of layout varies based on the apparent aesthetic effect of the interface

# Modes

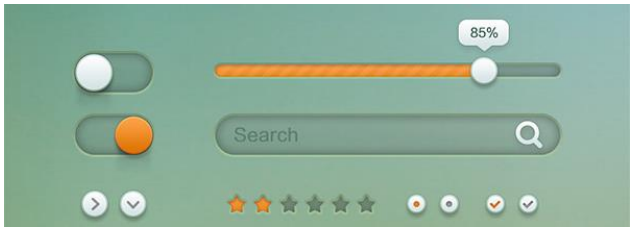
- Modes allow different behaviors from the same interface features (e.g. Caps Lock, Num Lock)
- Should be clear which view you're in so that when things start responding differently than what you're used to, you know it's because the mode is different
- Not necessarily bad – but sometimes really bad!
    - If mode change is not obvious – device appears non-deterministic because it's behaving differently in different ways
    - E.g. case-sensitive passwords and you accidentally have caps lock on
- Polya's principle of "Non-sufficient reason":
    - If there's no reason to believe things are different, they aren't different
    - Unless you're communicating clearly to the user that the mode is different, then you shouldn't expect them to believe that things are different
        - E.g. if the interface is going to behave differently when you're offline, then indicate to the user that they're offline

# Feedback

- Newton taught us that every action in nature is met with a reaction - this is not always the case in interfaces
- Every user action needs the interface to react so that the user knows the action is complete
    - This can be tricky in multi-tasking systems
- Especially important for displaying modal information
- E.g. go onto a website – difficult sometimes to tell whether it's continuing to load or if it's frozen
    - Need to change design so that the current status is communicated
- Progress bars, step-by-step updates etc, light on caps lock button

# Equal opportunity

- There should be no difference between input and output values (or known / unknown) - one can be substituted for the other.
    - This can clarify and simplify the interface
- Good examples include:
    - Spreadsheets - cells are neither input or output exclusively
    - Camera controls




- Example diagram above:
    - Control for it also communicates the immediate state – labels with 85%
    - Stars can be selected, and action confirmed
- Feedback incorporated into the affordance of the object

## Fitts's Law (1954)

- The time taken to acquire a target is a function of the distance to and size of the target
- For target of size S, a distance D from the pointer:

$$time = a + b \log_2 \left( \frac{D}{S} + 1 \right)$$

- Computationally modelling human reaction times and human ability to move from one place to another
    - Easier to hit bigger (than smaller) and closer (than further away) targets
- Time it takes to get to a particular interface option is a function of the object that you're trying to hit and the distance that it is from the starting point
- Implications for mobile devices and larger designs:
    - Can increase the size of the display to make it easier to interact with it
- Can choose a location for widgets, items and controls to make them closer and more convenient for the user or to deliberately make them harder for people to access

### FITTS'S LAW + PARETO PRINCIPLE

- Want to take the 20% of functionality that will be used 80% of the time and ensure that 20% is easier to access in terms of how big it is and how close it is.
- Example:
    - In a car with right hand drive, the power and volume button are what you will use the most and should be closest

## Hick's Law

- The time taken to reach a decision goes up as the number of choices increases
- The more complicated an interface, the harder it will be for people to use because they're having to spend more time deciding what to do
- Simplify your interfaces and reduce decision time for people – this improves their experience

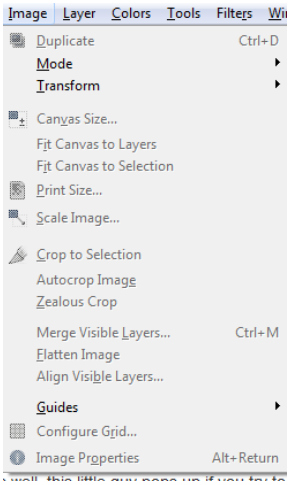
### SIGNAL TO NOISE RATIO

- Control vs. functionality
- Want info available in your interface to be greater than the noise in your interface
- Eliminate the unnecessary
- Too much density leads to degradation of viewer interest
    - Grouping items, putting them in hierarchies

# Examples: NextBus

## How easy is it to spot critical information?

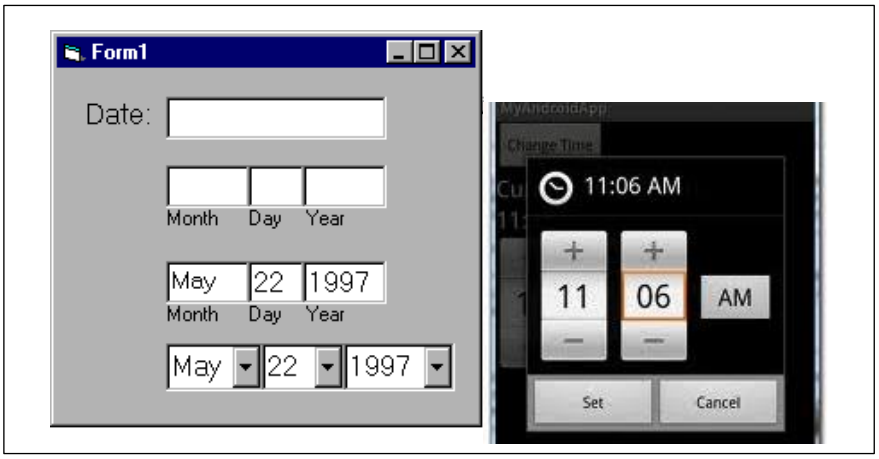
# Constraints and Characteristics




- Constraints: Constrain a design so that it can only be used the correct way
    - A particular kind of affordance that guides the user to a particular correct action
- Want to eliminate the possibility of a user committing an error – one of the ways this can be done is by implementing constraints
- Addition of constraints can open up other freedoms

## HIDDEN CONSTRAINTS

- Don't realise you'll be blocked from being able to do something just by looking at it
    - It's hidden until you try to access some particular action
- Example: entering the time
    - E.g. can't say it's 26 o'clock




- In the example, the blank (free text) field for the date would need a lot more error checking – don't know whether they enter the day/month/year first
    - Compare that to the select box at the bottom
    - Here using labels and constraints can help users pick the right thing and clarify their choices

# Transfer Effects


- People transfer expectations from known objects to similar new ones.
    - Previous experience conflicts with new situation ☹
    - Previous experience applies to new situation ☺
- Carry in the familiarity of a physical object to your app
    - Users are then able to recognize and react to it
- But what happens with people without that prior experience?

## LEARNED AFFORDANCES

- Relates to transfer effects
    - What you're transferring is what you've previously learned from somewhere else
- Idea of what you can or can't do or what the potential responses might be are carried forward from other experiences that you might have
- Examples:

    - Take this sorting cube. It's quite obvious to us that we have to match the shape to the hole. Yet when a child first encounters it, they aren't just struggling with motor coordination. They are learning that the shape of an object must match the shape of the whole.
    - The idea of a doorknob being able to turn isn't just a visual affordance – it's something we have learned through experience

## CULTURAL DEPENDENCIES

- Affordances and Mappings suggest how to use the object
    - Can be dependent on
        - Experience

        - Knowledge
        - Culture
- Example:

- It's a fire hydrant, but how do you know that? How do you know that you aren't supposed to park there?
- Conventions for designs change depending on where you are.
    - What we recognize and map to particular interfaces will change depending on where you are

## Obvious Affordances

- Origins: Affordances were necessarily visible
    - Push/pull door plates/handles
    - Glass can be seen through
    - Objects that you see and know how to interact with them
- **Discoverable:** But interaction reveals other (sequential) affordances
    - Knobs afford turning
    - Buttons afford pushing
    - One interaction will reveal another possible interaction
        - May not necessarily be a visual interaction, but rather a tactile interaction
    - In the same way, as you are designing apps, you may create a sequence of affordances (actions for people to follow) in order to get to a particular goal
- **Hidden:** Others are hidden
    - Automatic appliances – e.g. don't need to turn a handle to get a tap to turn on (motion sensor activated taps)
    - Sometimes unclear whether, when you try and accomplish a task, whether you're unable to do it because the app isn't working or you're doing something incorrect (you're not using the app in the correct way)
    - Puzzle boxes
    - Easter eggs
- **Learnable:** feedback → (leads to) discovery
    - Like with sequential discovery
    - As you interact, you can map that interaction to a possible consequence (if given feedback)
    - Glass breaks easily
    - Sending a message with Whatsapp – two ticks indicate the message has been sent and gone through
    - There needs to be an indication of status

        - Want affordances that are learnable, and that provide feedback that enables discovery
    - Puzzle pieces

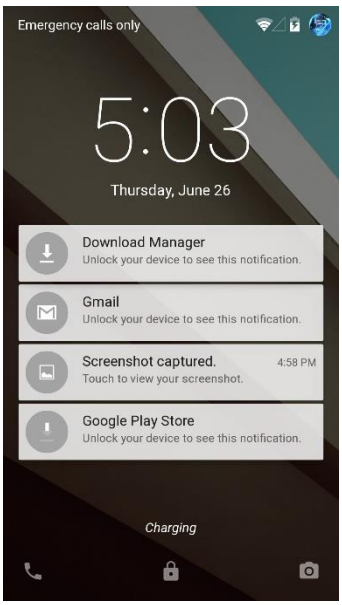
# Filtering

## SMALL SCREENS

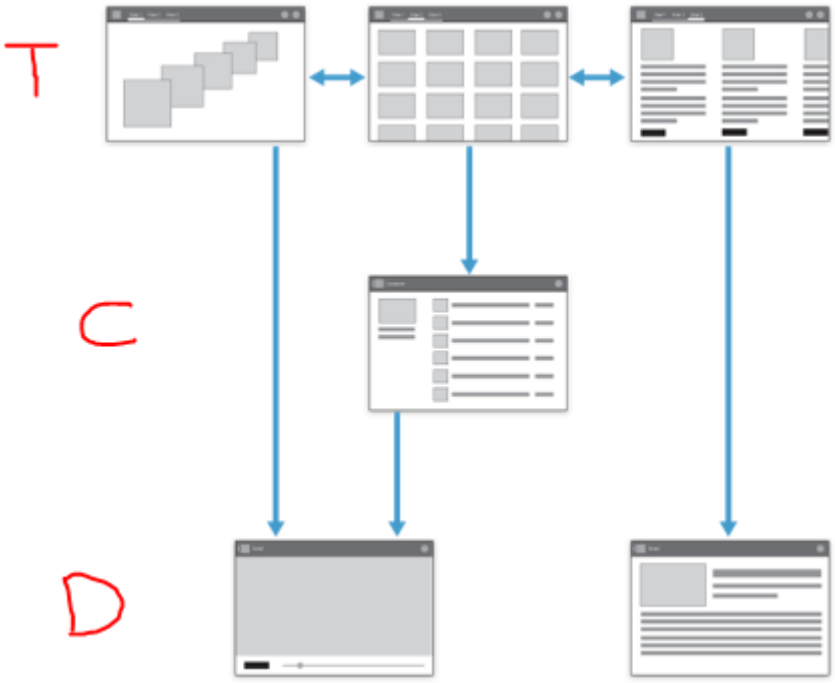
- Design consideration that's particular to mobile devices
- How do you design well for small screens, taking into account design principles?
    - Lots of Data – on a Small Screen
- Mantra of Mobile Design:  applies to text, audio, images…
    - overview
    - filter
    - details on demand
- Want to most effectively use a small screen to effectively show the things you want to convey

## OVERVIEW

- Start with interface that gives you an overview of the app (i.e. the entire experience, the primary features, etc)
- In that overview, it should give you access to sets of filters that will give you more and more detail as you go down the levels
    - As you decide how you want to interact, it gives you more details based on that – i.e. the ones directly relevant to the part of the overview that's of interest to you.
    - E.g. your phone's lockscreen – display s the day, time, and important app notifications
        - Helps you make the decision as to whether/not you're going to unlock your phone and give attention to it

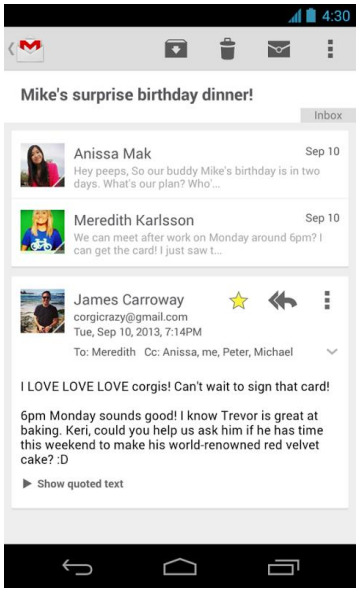


## Filter




- **Top Level views**
  - Top lvl of the app consists of the different views that your app supports
  - Views either show different representations of the same data or expose an altogether different functional facet of your app
- **Category views**
  - Category views allow you to drill deeper into your data **without having to see all of the data at once**
  - Example: with Whatsapp, have overview of all recent messages, can click on a person (chat) to get details of that person's messages to you
- **Detail/edit view**
  - The detail/edit view is where you consume or create data.

TEXT OVERVIEW




- Snippets that give you an idea of what the message is without having to open it
    - Helps you make an informed decision of what you want to read
- Priority inbox, importance flags, etc, helps you filter things that are most important to you (and ignore irrelevant ones)

## ORDERING

- Filtering is excluding irrelevant data
- Ordering clarifies exclusions
- Dynamic ordering helps identify which info is most important to you
    - Different types of orderings (e.g. time, date, type, and so forth) aids in this decision-making process
- Give user control over the filter and ordering helps them navigate the plethora of data available to them
    - E.g. what happens when you shop on Takealot
    - Filter at any time so that they can see the data they want – give users the power to make that choice
        - Remember the 80/20 rule

## RECOMMENDER SYSTEMS

- These dynamic categories emerge from crowdsourcing
    - Explicit: star ratings, reviews
    - Implicit: downloads, views
- Helps people identify the info they want, at a glance
- Example:
    - Use the stars and the colour of the stars as well as number of reviews helps the user scan and pick out the info that they want


## IMPLICIT FILTERING

- Location Based Services
    - Mobiles can know where they are geographically
    - Apps (given location data) can tailor results without having to filter location

## EXPLICIT FILTERING

- Use sliders and checkboxes to specify ranges and attributes
- Controls should be easy to do with your finger
    - E.g. trying to highlight and copy text with your finger on mobile device (little bubbles pop up to make the process easier)

GENERAL

- At a glance, being able to identify info that is most important/relevant to you is an important feature

# Mobile Modalities

- If you **must** enter text
  - Use auto-completion techniques, such as keyword in context, support voice.
- SMS and Messaging Services
  - Usefulness | Usability | User Experience
  - Why is SMS not dead?
    - Supported universally – anyone with a mobile phone can support an SMS
    - Cost
    - Low intrusion
    - Expressiveness and meaning – can express what you need to
    - Turn-taking – interface of status and feedback is immediate
      - Ticks on Whatsapp
    - Privacy – private communication in public spaces
    - Simple and Direct
  - All of the above factors into the UX
- Maps (mobile modalities)
  - E.g. Google Skymap
  - Can tell you where you are and how to get to your location
  - Location is an input and app giving update to location as output – mobile experience
  - Input and factor this into designs – how to do this effectively?
- Head up or Head down?
  - Concern – mobile phones cause a disconnect between people and what's around them
  - How to design apps that can be used to support in-person interactions and observing the world?

# Supplementary Materials

https://www.interaction-design.org/literature/topics/design-principles (main points)

- Design principles are widely applicable laws, guidelines, biases and design considerations which designers apply with discretion.
  - fundamental pieces of advice for you to make easy-to-use, pleasurable designs.
  - You apply them when you select, create and organize elements and features in your work.
- In user experience (UX) design, it's vital to minimize users' cognitive loads and decision-making time.
- Design principles should help designers find ways to improve usability, influence perception, increase appeal, teach users and make effective design decisions in projects.
- To apply design principles effectively, you need a strong grasp of users' problems and a good eye for how users will accept your solutions.
- You should use discretion whenever you apply design principles, to anticipate users' needs

## TYPES OF DESIGN PRINCIPLES

- Nielsen's commandments
- Additional guidelines:

1. **Don't interrupt or give users obstacles** – make obvious pathways which offer an easy ride.

2. **Offer few options** – don't hinder users with nice-to-haves; give them needed alternatives instead.

3. **Reduce distractions** – let users perform tasks consecutively, not simultaneously.

4. **Cluster related objects together.**

5. **Have an easy-to-scan visual hierarchy that reflects users' needs**, with commonly used items handily available.

6. **Make things easy to find.**

7. **Show users where they've come from** and where they're headed with signposts/cues.

8. **Provide context** – show how everything interconnects.

9. **Avoid jargon**.

10. **Make designs efficient and streamlined.**

11. **Use defaults wisely** – when you offer predetermined, well-considered options, you help minimize users' decisions and increase efficiency.

12. **Don't delay users** – ensure quick interface responses.

13. **Focus on emotion** – pleasure of use is as vital as ease of use; arouse users' passion to increase engagement.

14. **Use "less is more"** – make everything count in the design. If functional and aesthetic elements don't add to the user experience, forget them.

15. **Be consistent with navigational mechanisms**, organizational structure, etc., to make a stable, reliable and predictable design.

16. **Create a good first impression**.

17. **Be trustworthy and credible** – identify yourself through your design to assure users and eliminate uncertainty.

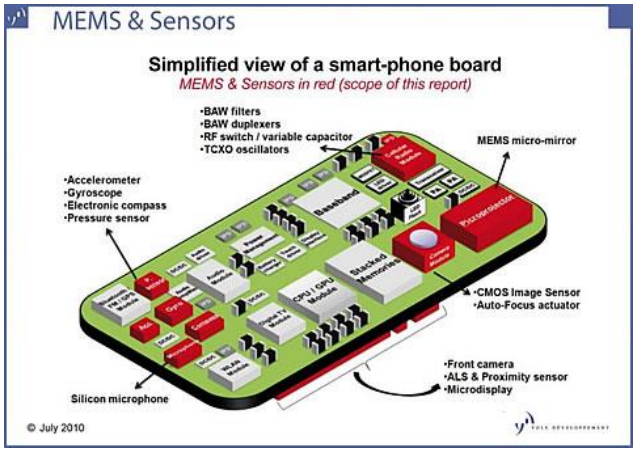
# Mobile Hardware

- Moore's law
- The evolution of mobile phones
  - Screen size and resolution changing
  - Computing power increasing
  - Power requirements increasing
  - All of these have different implications for computing

## Processor


- Intel missed the boat!
- ARM
  - RISC architecture licensed from ARM to many manufacturers
  - Optimized for low power consumption on low-end devices
  - Low-end routers, sub-100 computers
- Characteristics:
  - High speed (but fanless)
  - Low power consumption
  - High speed at odds with low power consumption
    - Designing to optimize for high speeds with lowest possible power consumptions
  - Multi-core
    - Because of the real-time interaction requirement
  - Handling inputs from a lot of different devices at once

### DEMANDS ON PERFORMANCE



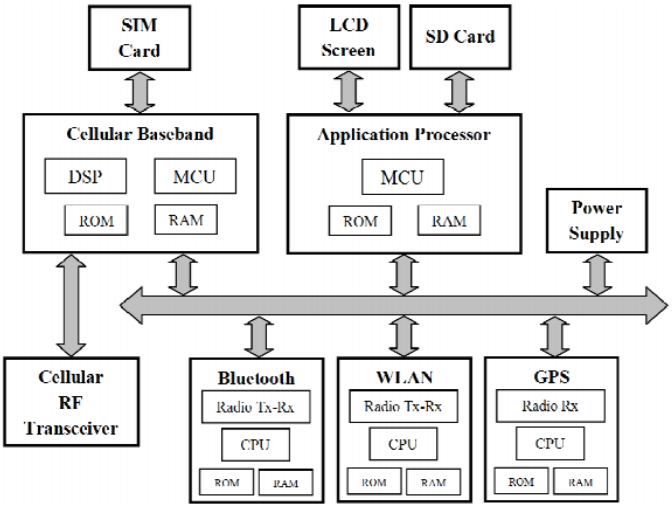

- Have # of sensors demanding performance from that CPU
    - Multi-core here makes a big difference

## MULTI-CORE PROCESSORS

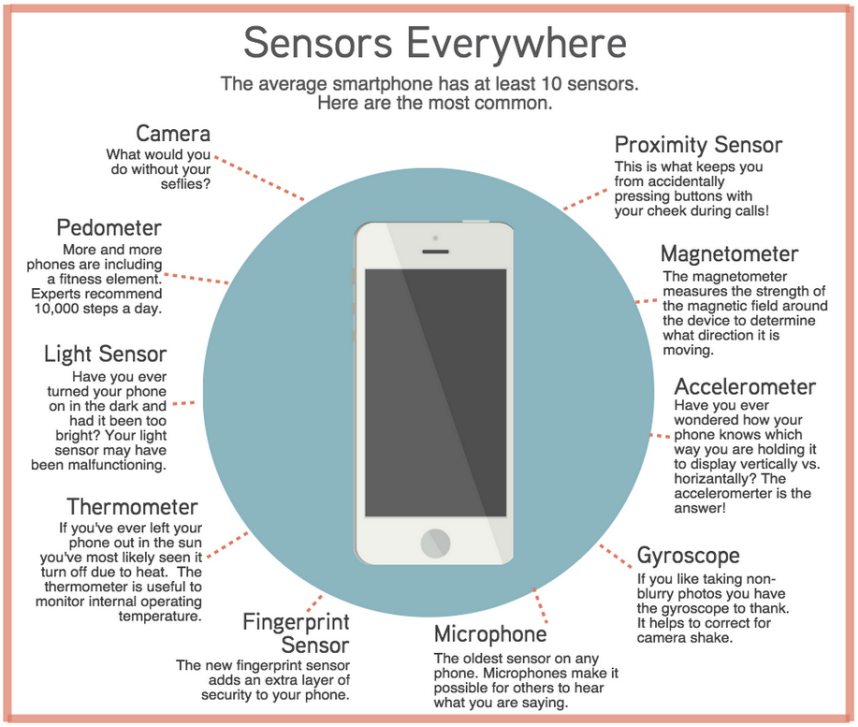
- Multi core makes a big difference
    - One core to run the radios
    - One core to run the OS and apps, etc
    - Especially important for multi-tasking and video
- Cores clock dynamically
    - More speed, more power consumption!

# Mobile Resources

## SMARTPHONE ARCHITECTURE




- Typically has radios, Bluetooth, wi-fi, GPS, controls screen, OS, manages battery charging




- Can use sensors to improve UX
    - Leverage sensors to make a more immersive/natural UX

## STORAGE AND RAM

- Limited RAM, Storage on a microSD card
- OS and base apps take up storage too

## BATTERY LIFE

- Apps consume data and battery life – constraint when designing apps
- Design apps such that they use less battery life

### DEVICE OWNERSHIP

- What phone are you developing for?
- Are you developing for a laptop, or for a smartphone?
- In SA – many have access to smartphones

## Communications and Networks

- GSM, CDMA, WiFi
    - GSM: have SIM chip with # tied to the chip (that gets inserted into phones)
    - CDMA: no SIM chip attached, register phone with provider and identity of phone is tied to phone number
- How people access the internet:
    - Mobile data, Wi-Fi
- Networks – how frequencies are encoded into bits and bytes on your phone
    - Mobile phone designed for one market may not work in an another because it's designed with a different physical radio inside

### TYPE OF MOBILE CONNECTIONS

- Where are people getting that data
- For the South African context:
    - Most people use prepaid

## Implications for Programmers

- Resource constraints (RAM, battery life etc) have implications for the programmers
- Errors have cost implications for users!
    - E.g. designing an app such that it overconsumes data

# Mobile OSes

- SW development
  - Mobile apps
  - As a dev, need to decide which platforms to support
  - Android vs iOS – this decision has implications for how sw development done
  - Developing apps for mobile is mainly done on a PC – using emulators, connecting to the cloud, installation and incompatibility and then debugging
  - Ongoing process that extends beyond single release
- People expect updates more frequently for mobile apps

## IOS

- iPod, iPad, iPhone OS
- Based on OSX – BSD Derivative
- Objective-C: needs a Mac
  - Need an Apple device to code for Apple devices
- Locked ecosystem
- Complex developer licensing
- Uniform target devices
  - When testing app, device used will be very similar to what users have

## ANDROID

- A Linux variant purchased by Google and maintained by the Android Open Source Project (AOSP)
- Java/Kotlin (Android Runtime Environment)
  - Run on android VM
  - OO language on app, but virtually interpreted and optimized on phone itself
- Reusable Components
- One OS, Many devices => difficulties hardware compatibility
- Low cost of entry

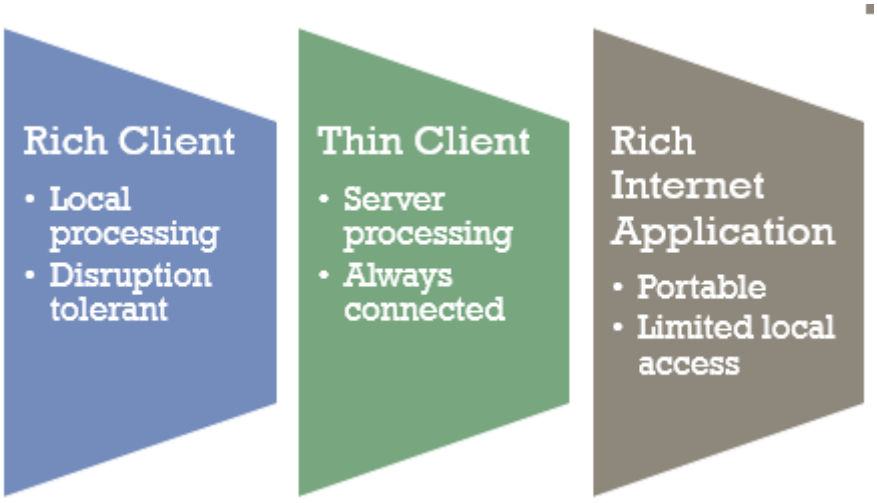
## J2ME

- Supported by most feature phone OSes
- Often device-specific libraries available
  - Apps don't always work across all devices
- Installation is cumbersome!
- No app store ☹

## OS Market shares by region

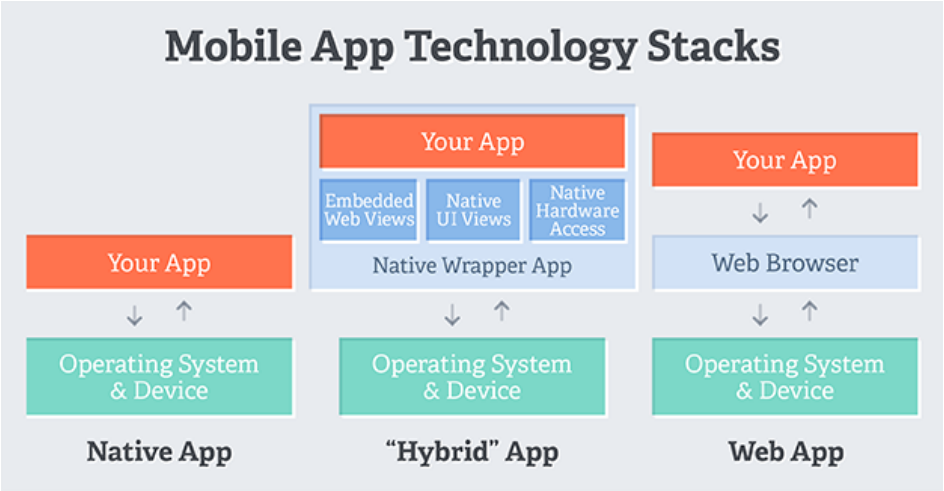
# Mobile Software Architecture

## High-level Architecture




- Different ways we could approach the design of mobile apps
  - One approach is to consider how much processing takes place in the phone vs remotely.
- Rich client:
  - Local processing
  - Disruption tolerant – if network connection lost, rich client will still work
  - Most apps, intelligence is on the phone (primary app logic on the phone)
- Thin client:
  - Server processing
  - Always connected – without connection to server, can't process anything => app won't work
  - E.g. Speech recognition clients are thin clients; web search
- Rich Internet Application
  - Portable – can write it once
    - Web based app
  - Limited local access – in terms of being able to access it when not connected to the internet
  - Often web/ajax/html interfaces, adopted by corporates (in order to limit amount of cross platform work they have to do), often not uniform with platform design guidelines (as a result of this write-once)

# Choosing a Framework




- Native vs Hybrid vs Web
  - Native speaks directly to OS – can be problematic, may end up building the same app for multiple platforms
  - Hybrid uses framework, use that framework's libraries to build native apps
  - Web – written in a language (like Java) that exists on the web (use a web browser); or the browser is compiled into your app
    - Develop interface using web-based framework

## COPING WITH DIVERSITY

- Desktop developers assume a homogeneity in hardware
  - Not true for mobile
  - Apps that you develop need to account for diversity of mobile devices
- Need to make consideration as you're choosing your arch and framework
  - Choosing not only arch, OS etc, but also ***range of devices*** that you're going to support
  - Choice that you make impacts who your potential users are, how many potential users you make have, the UX
- Diversity springs from
  - Different form factors
  - Different platforms
  - Different releases of platforms (i.e. versions) Win 8 vs win 7/NT/XP, Android Froyo/Jellybean/Kitkat
  - Manufacturers customising platforms

## HOLY GRAIL

- Write Once Run Everywhere
- Virtual Machines
  - J2ME, OneApp, Flash Builder
  - Can write piece of code that's not attached to hw, and then hw and OS on phone will have all the sw pieces that allow them to attach the app together
- Development paradigm
- Web Technologies
  - HTML5, CSS3

    - Appcelerator
    - PhoneGap
- The Web
    - Mobify
    - W3C
- Reality: Debug Everywhere (but getting there)

# Apps vs. Web

- How much of what an app does really needs an app?
- Why choose an app?
    - Will the user's experience of your company, or website etc, will be better because it's an app?
- Why is an app better than a website?
    - Apps may just replicate existing website, but they take advantage of the app affordances to make the experience better when people are on a phone
    - E.g. ticket booking
- The write once, run everywhere seems to have become, write everywhere or lose!
    - Can we shift to a model where we really write once and run everywhere?
    - Is our holy grail customized support for every platform?
- IF we can accomplish a write-once, work everywhere
    - How does that work with mobile phone design guidelines?
    - How to balance consistency across platforms for your app against consistency on the individual devices that people are using?
    - What's convenient for you may be carrying over experience with the website to the phone – but what about people who are mobile primary (and will never see your website)?
        - Then you want your app to be consistent with other apps on that phone, rather than the website (which they may never see)

# Mobile Web

- Mobile Web is an Improvement
- An entire web page is too much to take in!
- More space = more info = sometimes overwhelming
- With limited space of mobile screen, have to be more strategic about what they place in front of the user
    - Allows for a more focused experience

## LIMITATIONS

- In terms of Mobile web vs PC web
    - Small screen
    - Lack of windows
    - Navigation
    - No/limited Javascript or cookies on phone
        - Experience on mobile web can be different
    - Page types
    - Speed
    - Cost
    - Location

    - Social Impact
    - Connectivity to other social networks may be different depending on how things are implemented
- Limitations in terms of having a web-based approach to your UX

## Tradeoffs and Decisions

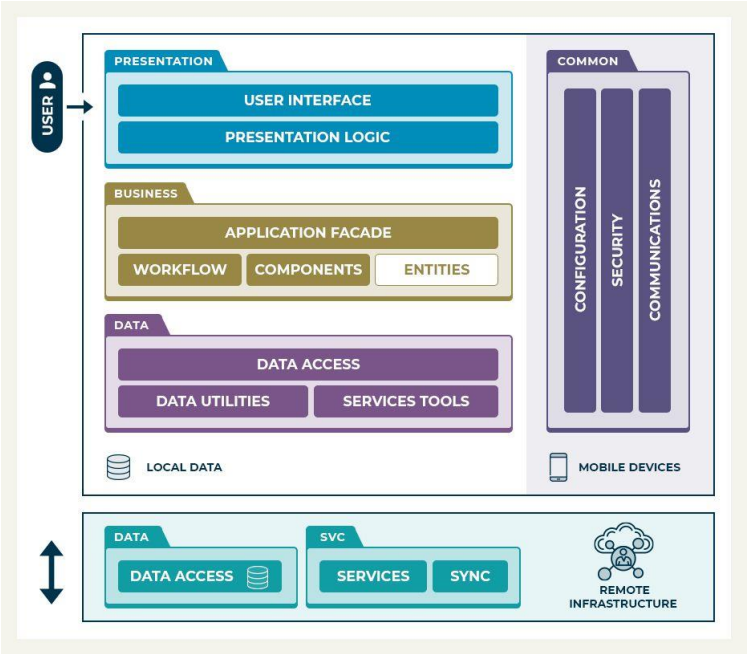
- Have to understand the requirements and pick the design and arch that matches what you want to do

| Feature | Native App | Hybrid App | Web App |
|---|---|---|---|
| Development Language | Native Only | Native and Web or Web Only | Web only |
| Code Portability and Optimization | None | High | High |
| Access Device-Specific Features | High | Medium | Low |
| Leverage Existing Knowledge | Low | High | High |
| Advanced Graphics | High | Medium | Medium |
| Upgrade Flexibility | Low<br>Always via app stores | Medium<br>Usually via app stores | High |
| Installation Experience | High<br>From app store | High<br>From app store | Medium<br>Via mobile browser |

- Also consider tradeoffs for
    - Data
    - Energy
    - Connectivity support
    - Location can be sent via web these days but what about personal info and other info?
    - Security and privacy – to what extent can you control them in these mobile apps?
- What are the motivations for going mobile?
- What are the non-functional requirements that are going to be influencing what the ”right approach” to this mobile app will be?
    - E.g. cross platform support, tighter integration with hw features, etc

# Component Architecture

- Have chosen framework and overall arch
- Have to now figure out how the little pieces fit together
- Need to design your app with a number of recognized components
    - Different architectural elements that are typical in mobile apps.
        - Log ins, settings window, where you're storing your data etc.
- Divide this into visuals, business logic and data.
    - But also – how will you support authentication, interaction with cloud services.
    - Where does shared data go?




- Mobile app is not just about the device itself, but how it will communicate with remote infrastructure
- Services:
    - Authentication services
    - Cloud services
- Have to figure out how to tie all components together

# Elements of Mobile Applications

- Authentication and Authorization
    - Consider using OpenID or existing login (eg. FB or Google)
- Configuration and Settings
    - Want to be consistent with how that's accessed across different apps
    - Use the cog icon, for example.
- Data Access (local and remote) and Synchronization
    - Architectural ways to approach this
    - Use XML to share structured data across the cloud (e.g. RSS)
        - Makes it easier for info to be processed by machines
        - Allows for consistent communications between client and server

    - Cloud hosting services (e.g. Firebase)
        - Pick which one to use in terms of functionality, speed, where the physical data will be located, cost etc
    - Caching (bandwidth constraints)
- Notifications
    - To what extent should your app be visible and accessible even while not in use?
    - How can you leverage notifications?
- Exception Management
    - Remember design principles
- Documentation
    - Should be easy to use and accessible
    - Need to plan for and build into the app's design
- Logging (only if necessary – why?)
    - Linked to security and privacy
    - Need lots of data to use AI
    - Need to communicate what you're doing to users of the app
- Security and Privacy
- How to organise it all?
    - Use sw design patterns
- Each of these is also often linked to specific user requirements - performance expectations, device support, etc.
- Don't try and make it all from scratch
    - Using patterns that are known to address particular needs and requirements, and that've been tried and tested

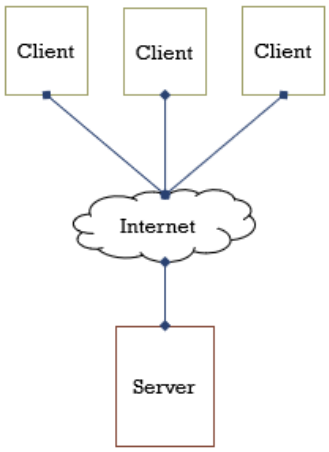
### ADDITIONAL DESIGN CONSIDERATIONS

- (1) What device types will you support? What peripherals are expected?
    - Don't just use sensors, but maybe additional devices like a card scanner (for transactions)
- (2) Consider bandwidth caps
    - Support for offline functions
    - Download on demand, based on connection type
    - Make data usage transparent
    - Weigh tradeoffs between caching and available storage
    - Avoid chatty or unnecessary network communications
- (3) Adhere to platform guidelines/consistency
- (4) Consider hardware resource constraints: battery life, memory, processor
- Remember ways in which design principles influences your choices, and that your architectural choice has implications for your UX

## Software Architecture

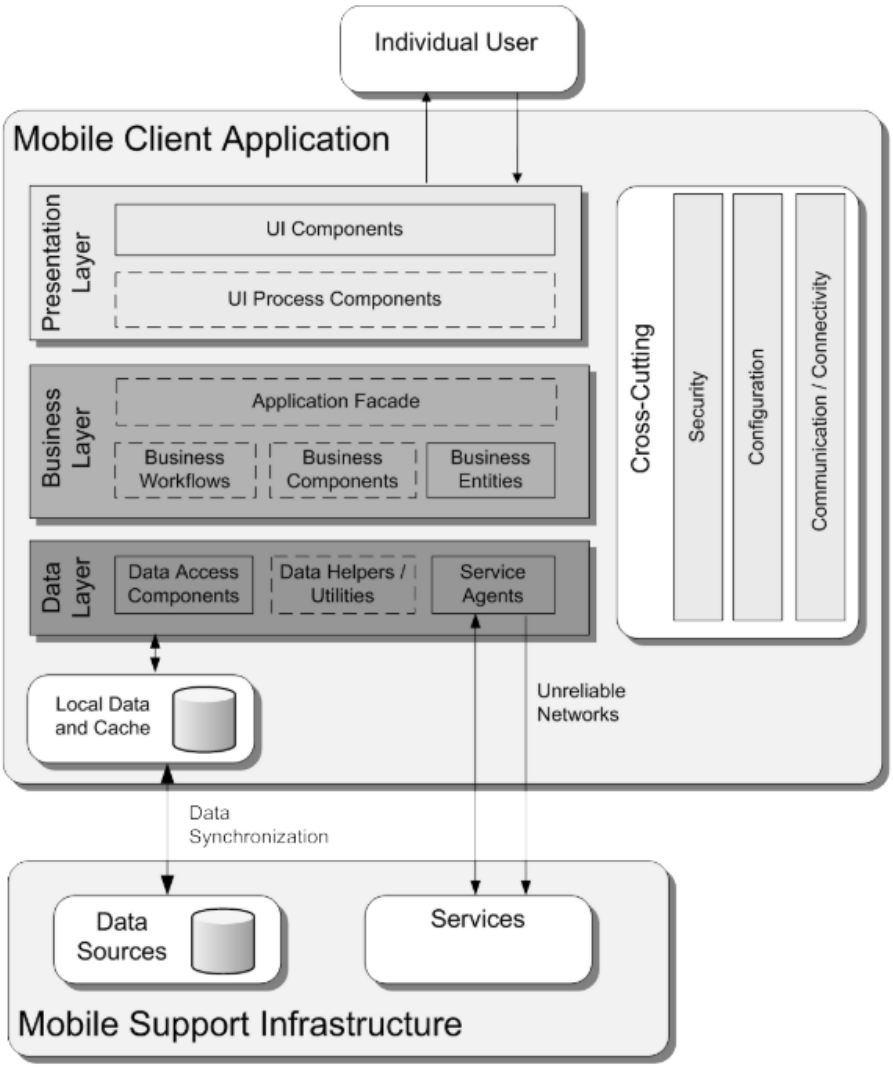
- Large-scale design patterns understood to address specific problems
- How all the parts of the code fit together
    - Use high level designs to explain this
    - Need high lvl arch to understand how to divide the work up, and how all the pieces will fit together well
- Adhering to known patterns makes code easier to understand, develop, work with other teams, debug, and read

## CLIENT-SERVER ARCHITECTURE



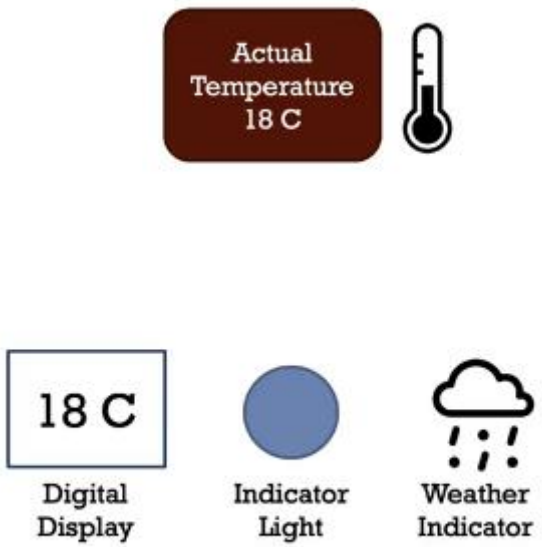

- In the case of mobiles, your application will use a protocol such as https or ssh to connect to a remote server to retrieve data, whether verifying a login or downloading data.
    - But this mostly tells you how to interact with data off of your device.
    - We must also consider how to build the application itself.
- *Problem: a shared database needs to be accessible from multiple locations*
    - Servers provide services
    - Clients (multiple) connect via computer network
- Examples: email, web, podcast, news readers
    - Anything where you're communicating with something off your mobile phone is leveraging client-server arch

## LAYERED ARCHITECTURE

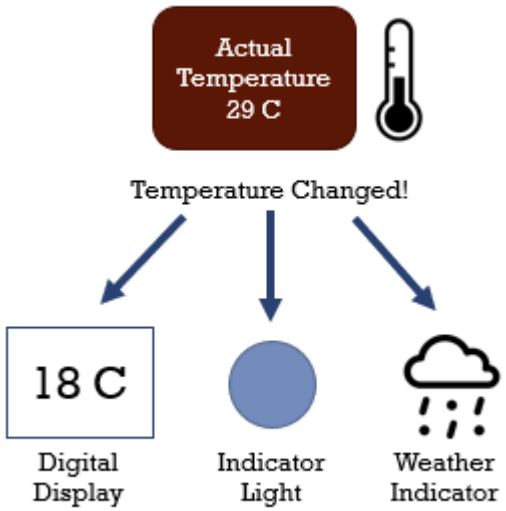



- The client software is organized into multiple layers
    - Similar to the overall android architecture
    - Presentation, business, data layers
- Each layer provides services to the layer above
- Having clean interfaces between each layer can prevent code from breaking
    - E.g. if change is made in business layer, it won't affect business layer IF you have a clear API between these two layers
- Cross-cutting services such as authentication are available to all layers
- Allows for independent development and upgrades within layers
- Apps don't have just one design pattern, diagram above have layered arch and within layers have client-server arch
    - Can compose many different design patterns together to make a whole sw arch

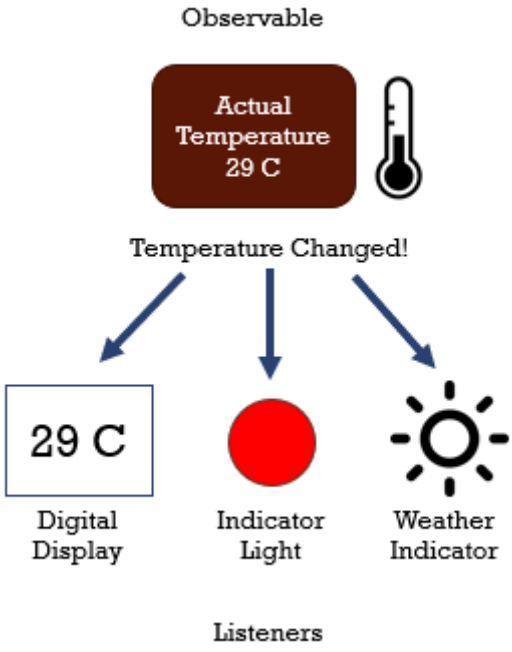
# Observable Design Pattern




- Within an app
- Defines a one-to-many dependency between objects
    - Not necessarily across a network, but within an app
- The Observable object notifies all listeners of changes to its state
- Especially common in Java-based user interfaces
- In Example Diagram:
    - Temperature sensor notifies all listeners



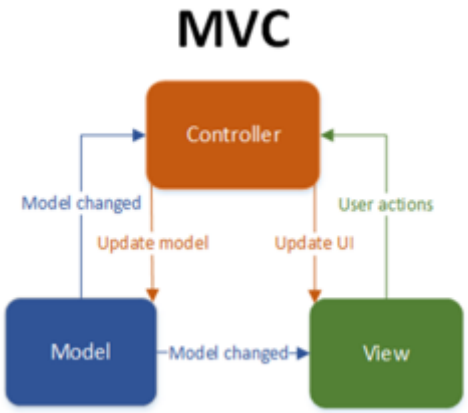

    - When it changes, it changes the interface – once the observers receive the notification
    - Observers change their state based on the new state of the observable object


Observable
Actual Temperature 29 C
Temperature Changed!
29 C
Digital Display
Indicator Light
Weather Indicator
Listeners

# Design Pattern

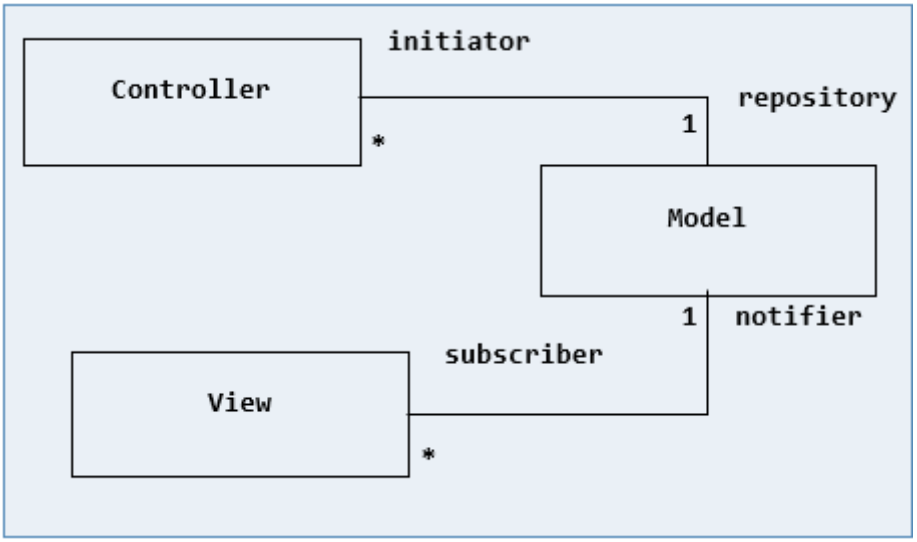
- Model View Controller/Presenter Architecture
- Software architecture design patterns for applications with graphical user interfaces.

## Model View Controller (MVC)




- Typically used for user interface development.
- Concerned with decoupling the graphical interface of an application from the code that does the work.
- Applicable to OOP languages
- Architecture for interactive applications.
    - A model (or application) is created independently of the user interface

### OVERVIEW




- Model: representation of the application
- View: way information is displayed
    - Relays the current state of the model

- Controller: mapping from user actions to operations on model
    - Changes the model
- UML diagram of MVC model
    - One model to many (relationship of) controllers and viewers

### Details

- Model/View/Controller
- Subsystems are classified into 3 different types
    - Model subsystem: Responsible for application domain knowledge
    - View subsystem: Responsible for displaying application domain objects to the user
    - Controller subsystem: Responsible for sequence of interactions with the user and notifying views of changes in the model.
- MVC is a special case of a repository architecture:
    - Model subsystem implements the central data structure, the Controller subsystem explicitly dictates the control flow
- Essentially, the pattern forces one to think of the application in terms of these three modules
    - Model
        - The core of the application.
        - This maintains the state and data that the application represents.
        - When significant changes occur in the model, it updates all of its views
    - Controller
        - The user interface presented to the user to manipulate the application.
    - View
        - The user interface which displays information about the model to the user.
        - Any object that needs information about the model needs to be a registered view with the model.
- This application architecture is very similar to a client/server model, except that all the components are bundled into one application.

## Components of the Pattern

### MODEL

- In MVC the model is the code that carries out some task.
- It is built with no necessary concern for how it will “look and feel” when presented to the user.
- It has a purely functional interface, meaning that it has a set of public functions that can be used to achieve all of its functionality.
    - Captures all of the business logic of your app
    - Represents underlying state and operations of your sys
- Some of the functions are query methods that permit a “user” to get information about the current state of the model.
    - Others are mutator methods that permit the state to be modified.
- However, a model must be able to “register” views and it must be able to “notify” all of its registered views when any of its functions cause its state to be changed.
    - Once again showing observable pattern
- In Java a Model consists of one or more classes that extend the class java.util.Observable.
    - This superclass will provide the register/notify infrastructure needed to support a set of views.

## VIEWS

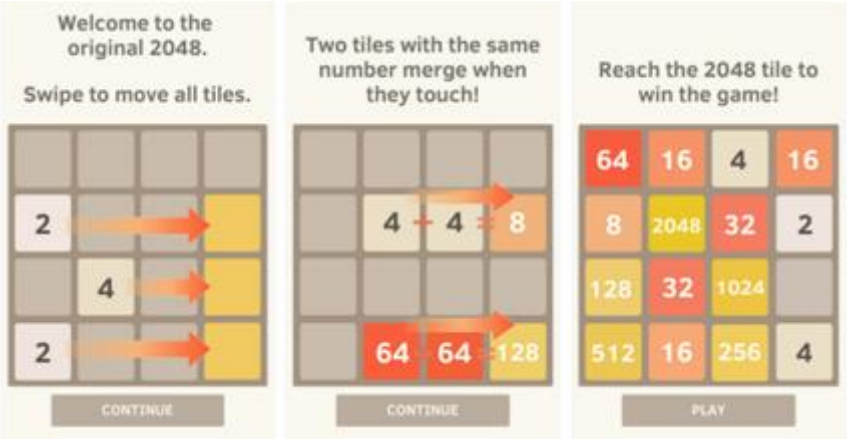
- A view provides graphical user interface (GUI) components for a model.
    - It gets the values that it displays by querying the model of which it is a view.
    - Text, labels, buttons, etc, that show what the underlying representation is
- When a user manipulates a view of a model, the view informs a controller of the desired change.
- In Android the views are built of components.
    - However, they must implement the java.util.Observer interface.
    - Done so that they can be registered as an observer on the model – so when the model changes, the view is able to respond to it
- Java Foundation Classes (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program.
    - Swing is the primary Java GUI widget toolkit - part of Java Foundation Classes (JFC)
    - Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform.

## CONTROLLERS

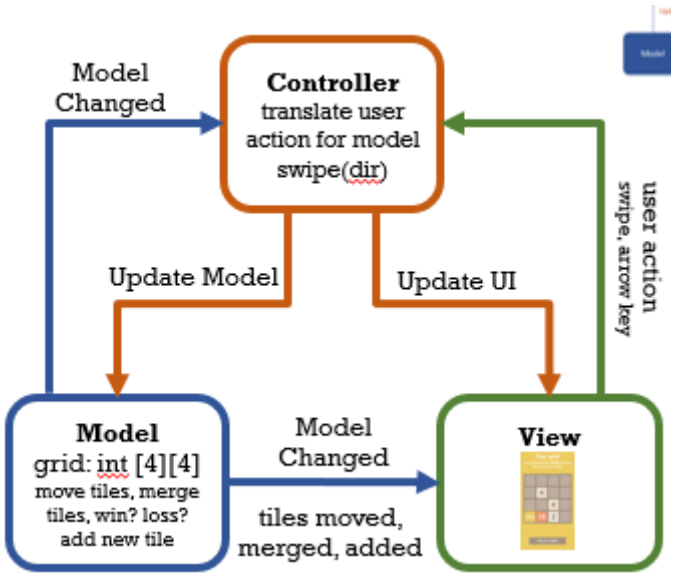
- Views in MVC are associated with many controllers that update the model as necessary when a user interacts with an associated view.
- The controller can call mutator methods of the model to get it to update its state.
    - Ensures consistency within the model
- Of course, then the model will notify ALL registered views that a change has been made and so they will update what they display to the user as appropriate.
- In Java the controllers are the listeners/observers in the Java event structure.

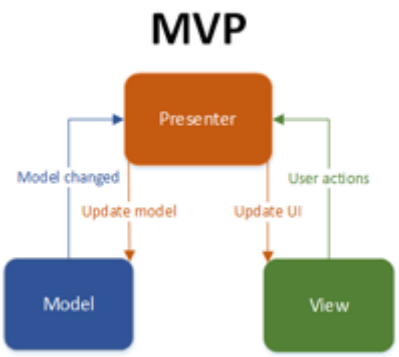
## Use Case

- Example: 2048




- Consider the 2048 game – how to implement using MVC?




- Adhering to MVP allows for a clearer code base
- Another version of MVC with same components

    - Rather than having view and controller be the observers for the model, you only have the controller be that observer
    - Have the observer-observable design pattern
- MVP does not allow the model to update the view, rather the update gets mediated through the presenter

## WHY MVC/MVP?

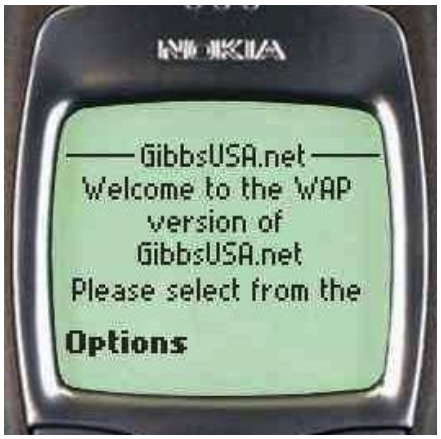
- **Separates presentation and interaction from the data**
- Effect of this: have this underlying model and only need to change the view and bits of pieces of the control layer to respond to the specific view
    - Can change view and controller while keeping the same model
    - Plethora of UI for the same function
- Allows data to change independently of its representation and vice versa – can easily change the UI without needing to change the model code
- Supports presentation of the data in different ways – consider A/B testing
- But!
    - Additional code and code complexity
    - Hard to implement in android (conflicting vocabulary between how android describes its architecture and the mvp paradigm)

# MVP on Android

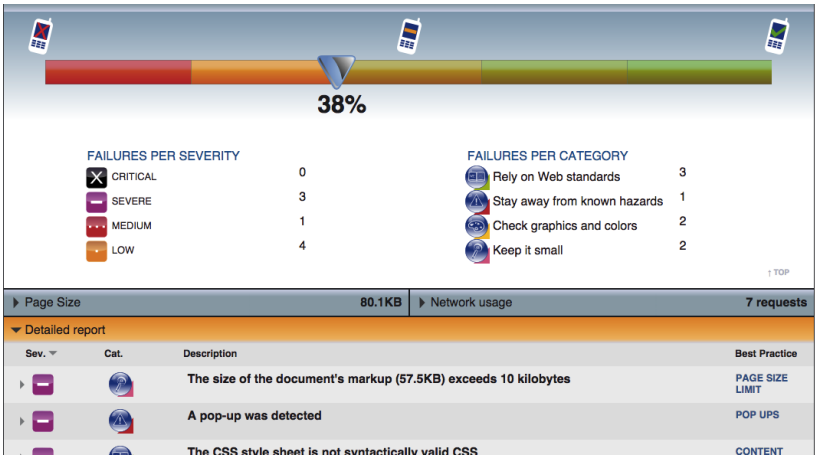
- MVP View ≠ Android View Component!
    - MVP View will manage multiple components
- View is essentially passive – the Presenter (controller) changes the UI upon Model updates.
- Use interfaces to define APIs between the Model, (architectural) View and Presenter, implemented using the Listener or Observable model
    - Presenter listens for UI events, and changes to the Model
    - View is updated by the Presenter once Model changes

# Mobile Web

- A design paradigm for mobile sw architectures
- If you have a web page – it should definitely support mobile browsing
    - Mobile devices are the dominant way that people access the web
- Approach to optimizing web experience
    - Opera Mini approach
    - Opera mini compresses pages by about 70%
    - Optimizes web performance
    - Reduces data compression
    - Reduces battery consumption
    - Caveat: not all websites will work; https (makes it harder to do the optimization)
- History of Mobile Web
    - # of mobile web mark-up languages
- Compromises




    - cHTML and WAP were created to enable mobile web
        - Required explicit reformatting – had to make a version of that webpage for the phone
        - WAP was terrible
        - cHTML was genius (DoCoMo/i-mode) => focused on providing services
    - Most browsers today are fully featured but still have small screens
- Not working with cHTML as much anymore, but have ajax, javascript etc be the same for all
    - Switch based on what the device is – can be done server side
    - Means there's lots of checking tools – e.g. is this webpage going to be compatible with mobile device? Does it meet web standards, etc?
    - In diagram, mobile version of gmail fails some checks




## WEBKIT

- Most smartphones use some variant of WebKit
- BSD License – so crops up in a lot of places
    - From Apple originally
- Incorporates
    - SVG, MathML, JavaScript, CSS
- Trend is towards HTML5
    - Adobe Edge based on HTML5, Javascript, CSS3, and jQuery
    - Microsoft Silverlight 5 – end of life in 2021
    - Google Web Designer (Beta)
    - Progressive Design

## SEPARATE URLS

- Strategy for supporting mobile web
- Explicitly drive users to the mobile site
- *.mobi | m.* | x.* | “Click for Mobile Site”
- These days websites will automatically respond to what device you have and displays things based on your screen size

# Approaches

- Reformatted Web Site
    - Focus on interface of website
    - Information Appliance
        - Focused interfaces for web sites
    - Sites are broken into structured tasks
    - Assumes people do not usually browse on mobile device (not strictly true)
- Familiar on desktop since 2005 (AJAX)
    - Combo of Javascript, CSS, DOM, XML, etc
    - Applications run in sand-boxed web browser
- Support for JavaScript in WebKit allows this
- Avoids installation process

## Emergence of HTML5

- Now only have to write one website


## RESPONSIVE WEB DESIGN

- Craft your website to create an optimal viewing experience across a wide range of devices
- Achieved by:
    - Fluid grid of components
    - Adaptive Layout based on proportion and size
    - Flexible images
    - Media queries

# Progressive Web Apps

- Uses responsive web design, and part of webpage stored and cached locally
- Want lightweight version of web app that appears on your phone but has all the advantages of normal web apps
- Reliable | Fast | Engaging
- **Progressive** - Work for every user, regardless of browser choice, because they are built with progressive enhancement as a core tenet.
- **Responsive** - Fit any form factor, desktop, mobile, tablet, or whatever is next.
- **Connectivity independent** - Enhanced with service workers to work offline or on low quality networks.
- **App-like** - Use the app-shell model to provide app-style navigation and interactions.
- **Fresh** - Always up-to-date thanks to the service worker update process.
- **Safe** - Served via HTTPS to prevent snooping and ensure content has not been tampered with.
- **Discoverable** - Are identifiable as “applications” thanks to W3C manifests and service worker registration scope allowing search engines to find them.
- **Re-engageable** - Make re-engagement easy through features like push notifications.
- **Installable** - Allow users to “keep” apps they find most useful on their home screen without the hassle of an app store.
- **Linkable** - Easily share via URL and not require complex installation.

# Google Accelerated Mobile Pages

- Featured in carousels as “AMP”
- Smaller pages
- Version of your content is hosted and served by Google AMP caches
    - Will improve search performance

- **Design Principles**
    - User Experience > Developer Experience > Ease of Implementation (developing a quality UX)
    - Only do things if they can be made fast
    - Don’t design for a hypothetical future browser
    - Prioritise things that improve the user experience – but compromise when needed
    - Don’t break the web
    - Solve problems on the right layer
        - E.g. not necessarily solving problems on the phone or the serving website, but rather deciding how much data to send to the phone can happen in the network itself
    - No whitelists
- Idea is that you support AMP, which makes the UX better because they’re getting web pages faster

# Mobile Input

- Is mobile interaction impoverished (means we have a compromised experience) or extraordinary?
    - Mobile design challenges how we look at HCI and provides opportunities for innovation
- Making text entry on phones easier
- Utility trumps usability
- Alternative keyboards: Laser projections, chording, mini-keyboards
- How people interact with keyboards is influenced by the affordances of how they're being held and the context in which they're using them

## KEYBOARD MODES


- Same user input (or system) produces distinct results depending on system state
    - Classic example: [CAPS] or [NUM] lock
    - Mobile affords a clearer way – if you press CAPS lock on a mobile keyboard, the keys change in response
    - Need clues about current mode and communicate it to the user
        - background/header colour
        - light indicators
        - toggle switches

# Smartphone Input


- Fewer physical keys means **greater ambiguity**

- Software screens allow for easier innovation
    - Virtual keyboards, handwriting recognition etc
- Keyboards and glossaries adapt according to language, personal vocabularies, letter order in words.
- But caveats – how well does this work for multi-lingual users, or in locales with different recognized names.
- Existing algorithms and dictionaries can arguably be considered biased against anyone who types words differently and hasn't had input into design of these systems.
- If text is a component of an app you are designing, think about how you can make that easiest for your users – from reducing the typing necessary to using the right keyboard options to make it work.

## SPEECH RECOGNITION

- Uses cloud service
- English recognition heavily biased towards American and UK accents
    - Bias decreases as they're able to build up more data for more accents
- Low-resource languages neglected – no gold standard for translating that recorded text into written text and there is no built up services to support this
- Privacy and social restrictions – can't use it in public if you don't want other people to hear what you want to be typed into the phone
    - Privacy concerns – your speech can be piped into a corporate database where people can listen to what you say
- Ambiguity – if it fails, can do so spectacularly or in an irritating way
    - Continued failure will cause people to stop using this feature
- One also needs to weigh the tradeoffs between implementation difficulty, and potentially conflicting user requirements
- There is some work that allows small vocab speech recognition within a device rather than remotely, but it's limited in application
- However importantly – for connected users, speech is a rapidly improving mechanism for mobile input.
    - Google Home, Amazon echo
- Pervasive speech recognition
    - Security considerations
- How to leverage speech recognition as an input?
    - The speech recognition device is still an interface and needs to follow rules of design and usability heuristics
    - Speaking needs to evoke specific responses
    - Learned affordance – talking to the device to learn how to use it
- These suggestions (emailed to the user) are effectively a user manual to bootstrap exploration and discovery.
- But they also highlight the ways in which home speech devices have failed to achieve the goal of natural user interaction.

## TOUCHSCREEN INPUT

- Core of non-text input in HCI: icons and menus
    - Things that you can point to and click on in order to evoke a response
- Using windows, menus and icon is essentially like learning language

# Direct Manipulation

- Key goal in design of interface is to support direct manipulation
    - Even more important for touchscreen devices
- The criteria for direct manipulation are the continuous representation of objects of interests and the fast, incremental, undoable actions, which have an immediate visual impact on the object itself. The goal is to allow the user to directly interact with the object.
- Key points
    - Fast and incremental allows for continuous representation.
        - Interface immediately responds to that action and it should be seen
        - Like resizing a window.
        - Zooming in on a map.
        - Scrolling a screen.
    - Incremental interactions affords learning – initiating an action that's undoable allows people to learn quickly what that action will achieve.
    - The immediate visual impact allows people to immediately know the effects of their action, making it seem like they are interacting with a real object, not just a virtual representation.
    - Undoable actions allows people to go back and forth with the manipulation
        - And gives them that freedom to explore

## TIMERS



- Consider various timer interfaces and whether they are easy to use.
- This one on the left – the visual timer is a great example of direct manipulation. Dragging and edge of the red bit around the circle is a direct manipulation of the interface that also tells you exactly what you're setting,
- Most apps will have you set the hms separately either by typing in the numbers or using various widgets to adjust the number up and down.
- This is also a form of direct manipulation - and akin to how you might have set an analog timer once upon a time.
    - But it is clunky and requires more interactions. And it misses Schneiderman's point about direct interaction with the object, as well as incremental interactions.

## CANONICAL TOUCH OPERATIONS


- multitouch → "stroking, not poking"
- Graphics: translate, rotate, and scale
- Text: cut, copy, paste
- Want to have interfaces that will leverage the canonical operations (zoom, scroll, tap)

## AUGMENTED REALITY

- As a form of mobile input
- Infinite Screens, Halos, and Peepholes, Augmented Reality
- Use sensors to interact with the environment around you and provide additional input to the phone
    - E.g. provide your location
    - Phone can interact with that additional info and provide feedback
- Train your phone on a particular object and it'll give you more info about that object
    - E.g. using the camera as a sensor to provide input to an app/service available n your phone

# Sensors and Attachments


- Camera, payments (Bluetooth/NFC), spirit level (gyroscope)
- Use sensors to provide feedback to system and to you

## CAMERA INPUT

- Not just for pictures – can also be used for making judgements about distance, length, barcode scanning etc.
- Leveraging the camera

## LOCATION INPUT

- Leveraging your location to better understand your environment

# Natural User Interface (NUIs)

- Interacting without having to think hard about how
- What is natural?
    - Interface that supports intuitive interactions
    - Don't have to think about how to do the interaction or what it should be
    - There is flexibility into what we classify as natural or not
- Kinect – but not Wii?
- Contrast with keyboard/mouse or indirect mappings
    - AR is a huge contrast to interacting with keyboard and mouse in order to achieve computing
- Manipulating photos – e.g. rotate
- Lower learning curves – easier to pick up
- Using sensors (pressure, etc) to interpret user requirements with more depth

## ON MOBILE INPUTS

- Small keyboards and display are constraining but **not limiting**
- Our aim is **direct manipulation** or a **natural user interface**
- Just because you can build something doesn't mean you should!

# Inside Android

- Android is a software stack for mobile devices that includes an operating system, middleware and key applications.
- The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language
- Android isn't just the sw on the phone – whole ecosystem/platform of people working to support and advance the sw

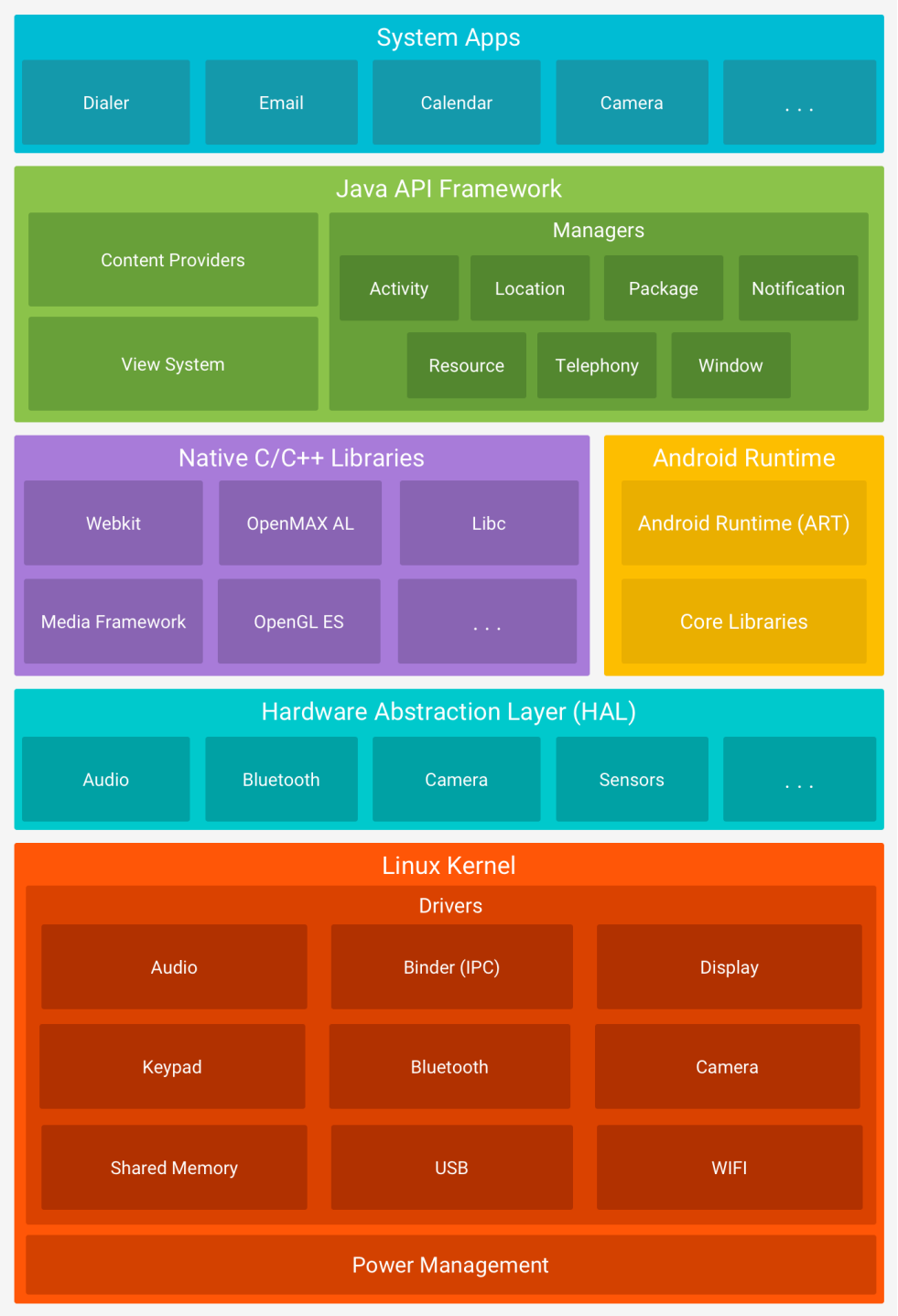
## The Android Open Source Project

### KEY AOSP FEATURES

- Layered Architecture
  - APIs between each layer – provides "hooks" for people to come in and implement new features, etc
  - This allows people to make contributions in one layer without disrupting another layer
- Apache License
- Open platform (open source), easy to adopt
  - Devs can leverage this
- Tied to the Google ecosystem (…or not?!)
- Developers can branch code for customization
  - For example, Amazon Fire, Google TV, Huawei
- Not just the code – also the people, and the principles for working together
- Open Handset Alliance helps make AOSP what it is

# Android Architecture

Architecture diagram

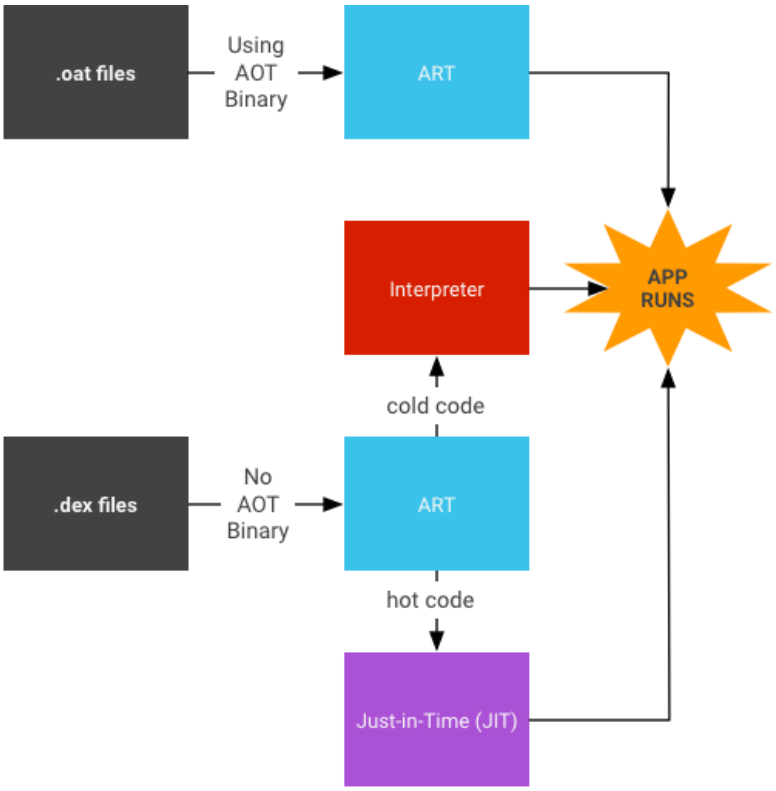


- **Linux Kernel**: manages threading and low-level memory management, well-known kernel
- **Hardware Abstraction Layer (HAL)**: api allows apps to be written independently of specific hardware models
    - E.g. takes all the different kinds of cameras that could be in a phone and makes them look the same
    - By doing this, don't have hw specific code in the upper lvl layers – all cameras look the same
- **Android Runtime**: where apps run
    - Includes set of core libraries and the run time environment which executes the byte code
- **Native Libraries**: C / C++, compiled for the hardware and installed by the vendor
    - Surface Manager: off screen bitmaps combined for rendering
    - Graphics: openGL
    - Media codecs: supports aac, avc, h.263, mp3, mpeg-4
    - Browser: webkit
- **Java API Framework:** set of libraries and packages that you can use to develop the apps
- **System Apps**: leverage API framework and providing UI and logic of whatever the app has to do (on top of the android OS)
- Android uses a layered architecture, with each layer exposing an API to the next, but also shielding the next layer from the implementation specifics required for that layer.
    - This allows an android app built and tested on one phone to effectively work on most other phones.
- Built on top of a linux kernel, with drivers for the standard devices on android phones
    - Use the adb command in the shell to interact with applications

## FOR THE DEVELOPER

- **Browser**: Integrated, based on the open source WebKit engine
- **Graphics**: Powered by custom 2D graphics library, with 3D graphics based on the OpenGL ES 1.0 spec
- **Data Storage**: SQLite for structured data
- **Media**: Support for common audio, video, and still image formats (Mpeg-4, H.264, mp3, aac, amr, jpg, png, gif)
- **Radio**: multiple frequencies, and GSM or CDMA
- **Wireless**: bluetooth, EDGE, 3G, 4G, 5G, and WiFi
- **Sensors and Devices**: camera, gps, compass, accelerometer
- **SDK**: Rich development environment, including a device emulator, debugging tools, memory and performance profiling

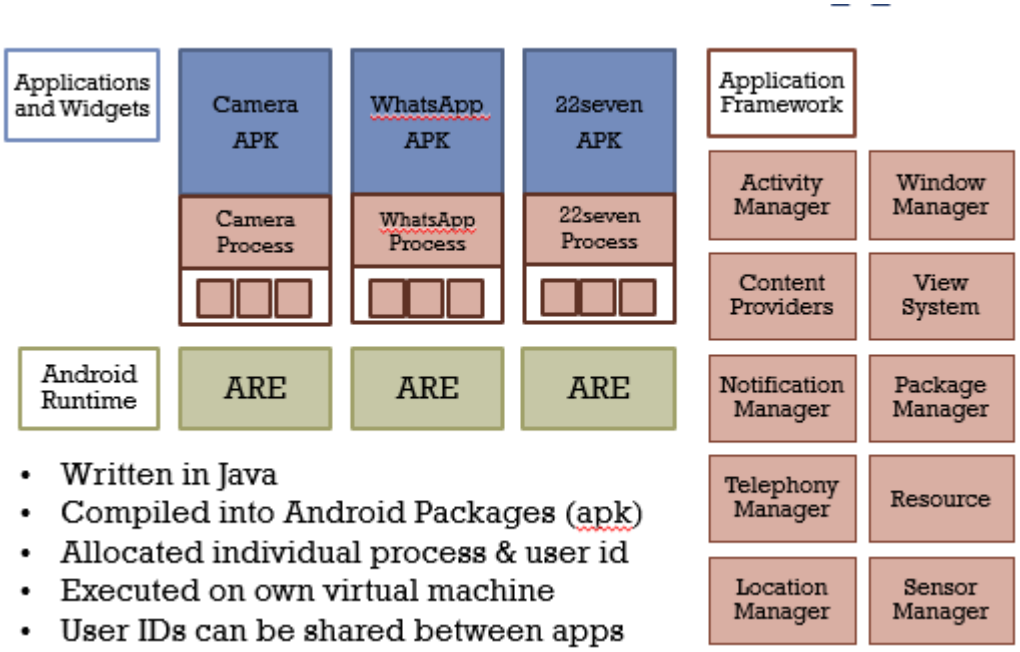
# Android Runtime Environment




- vs Dalvik Virtual Machine (ARE replaced this)
- Introduced with KitKat, official with Lollipop
- Nougat couples a JIT compiler with AoT compilation
- Have Java engine, write code, code gets compiled into byte code, gets interpreted on the system where it gets matched up to underlying hw so that it can run
    - Not building the executable, but it's being built on the fly with a JIT compiler (Just In Time compiler)
- ARE
    - Ahead of Time (AoT) Compilation
        - Optimisations put in so that compilation can be done on the device ahead of time
        - This works because once the app is on the device, the devices' hw won't change so won't need to do JIT
    - Apps compiled at install or boot
    - Faster to run
    - Independent VMs
        - Each app sits in its own VM so that apps can't accidentally run over each other in terms of using resources
    - Takes more space – but still a vast improvement
- DVM
    - Just in Time (JIT) Compilation
    - Faster to boot
    - Lower space requirements

# Nuts and Bolts of Android Apps

- Written in Java/Kotlin
- Compiled into Android Packages (apk)
- Allocated individual process & user id
- Executed on own virtual machine
- User IDs can be shared between apps
    - To share files/content
    - To share virtual machine
- Example:
    - Camera APK has process which runs in its *own* runtime environment
    - Each ARE is its own VM
- Apps leverage the app framework (basically a series of libraries on the system)
    - E.g. to show notifications, buttons, widgets etc




## Android App Development

- Android studio (IDE) and Android emulator

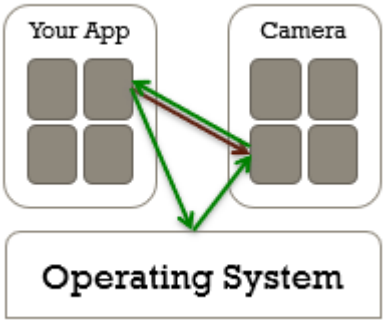
## KOTLIN VS JAVA

- As of 2019, Android development is increasingly Kotlin-first
    - Examples now provided on developer.android.com
- Interchangeable with Java - also runs on JVM
- More succinct code
- Fixes some issues with Java, but not always better
    - Null safe
    - No raw types
    - Arrays are invariant
    - Function Types
    - No checked Exceptions
    - Structured concurrency
- https://kotlinlang.org/docs/reference/comparison-to-java.html

### API

- This is an incredibly important decision – picking what API level you will support.
    - i.e. “How far back do I want to support?”
    - Drives what features you can support and who can use you app?
- Try and support as far back as you can

## Android Architecture Components




- Dividing apps into components saves resources
    - Allows the OS to better manage interactions between apps and resources of app
- Each component of an app is a point of entry for the operating system
- Your app can call parts of other apps (e.g. use the camera)

### TYPES OF COMPONENTS

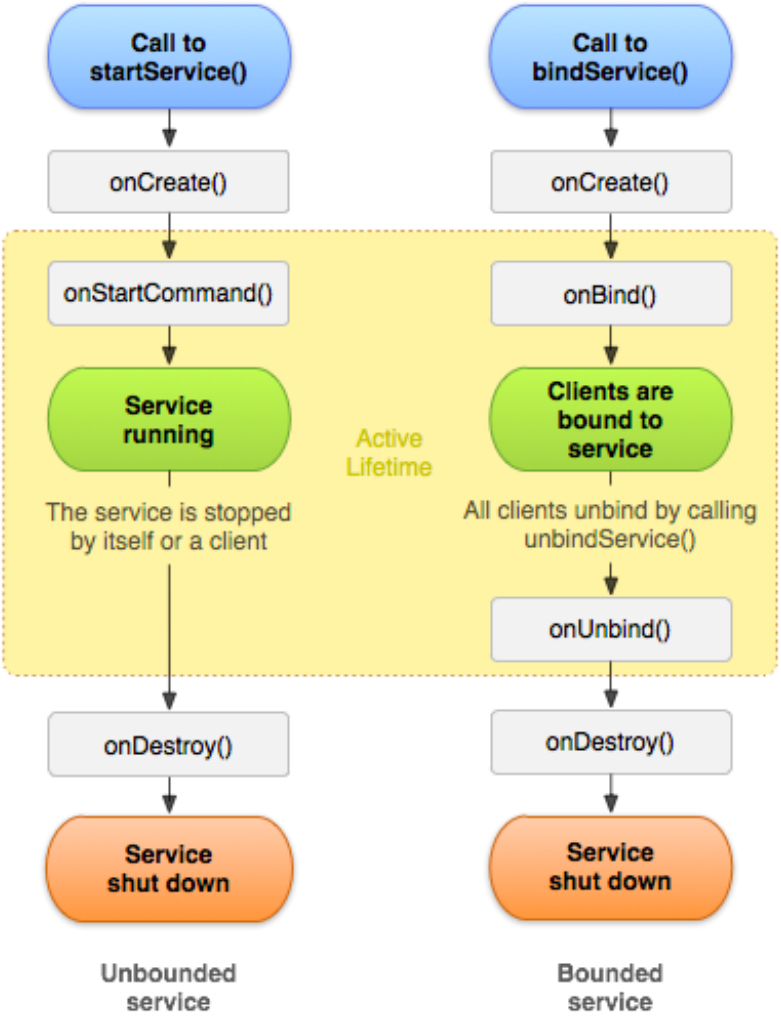
- **Activities**
    - User interface
    - Application Screens
    - example:  mail list, view message, compose message
- **Services**
    - Background tasks
    - Long-running operations
    - example: checking for new mail, sending mail
- **Content Providers**
    - Manages shared app data
    - Private data too
    - File system, SQLite, cloud
    - Query or modify data
- **Broadcast Receivers**
    - Responds to system-wide announcements (e.g. battery low, screen off)
    - Status bar notifications

## ACTIVITIES

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)

        fab.setOnClickListener { view ->
            Snackbar.make(view, text: "My action",
                    Snackbar.LENGTH_LONG)
                .setAction( text: "Action", listener: null).show()
        }
    }

}
```

- 1:1 correlation with screens
    - Screens => what you see on mobile device/what's displayed to you
    - One activity for each of the type of screens that can be displayed
- Must be declared in the manifest (xml)
    - Describes whole android app
- Use <intent-filter> to denote initial activity
    - Activity that gets launched when you click on the icon
- Activities are called by other Activities
    - If you need to e.g. change screen through business logic, activity would nee to launch that next activity
- Represent a 'unit of interaction' framed by a screen
- Can be composed of fragments (parts of activities)
    - May have several fragments within that activity that handle different interactions

## SERVICE




- Runs as part of the app
- A service can be bound, started, or both
    - onBind(): allows IPC, provides an API for interaction with app
    - onStartCommand(): long-running background service
- Multi-threaded!
    - Runs either in background/foreground
    - Runs operations that aren't necessarily bound to UI but can indicate when the UI needs to update
    - Make sure to start a thread for your work, or it will slow down the main activity!
        - Need multithreading so that we don't slow down the main activity
        - Want main activity and user display to be seamless (not interrupted by background processing)
    - Use IntentService to handle multiple calls at once
- Can send toast and statusbar notifications to the user
- Ongoing services (foreground) won't be killed if system is running low on memory
    - Examples: spotify playing your music (notification in the drawer that sticks)
- Components have hooks and behaviours that work with OS to help it manage that resource

## CONTENT PROVIDERS

- Shareable data stores
- Local or cloud: database, files, dropbox
- Store your own data
- Share your data with other apps
- Access other ContentProviders (e.g. contacts list) via ContentResolver

### BROADCAST RECEIVERS

- Receives system and application messages (Intents) and invokes response
- No UI – but can update status bar
    - E.g. listener parts of app that may react to other things that may be happening on your phone outside of the app
- Examples:
    - Battery low, message received, picture taken, incoming call

## Intents

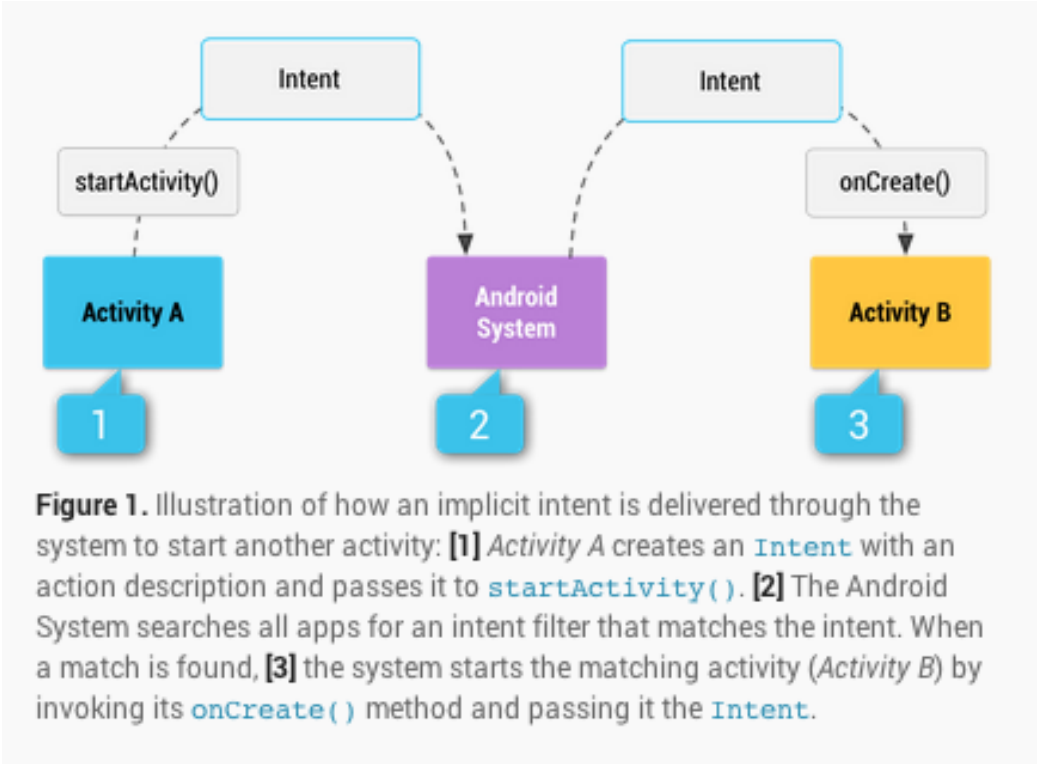



**Figure 1.** Illustration of how an implicit intent is delivered through the system to start another activity: **[1]** *Activity A* creates an `Intent` with an action description and passes it to `startActivity()`. **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its `onCreate()` method and passing it the `Intent`.

- How activities are communicating with underlying android system
- Activates services, activities, and receivers
- Can be targeted (want an app to start a particular activity) or broadcast (to everyone on system)
- Explicit or implicit (app chooser)
    - Explicit – opening a particular app to get the data from that app
    - Implicit – e.g. want to be able to send this message to any app that's willing to receive one
- Receive implicit intents using <intent-filters>

# Resource Management

- All this adds to the ways that android supports good resource management
- Responsiveness is maintained at all costs!
- Content providers automatically sleep (when not used)
- Activities and services can be shut down anytime
    - E.g. if you launch one app, then another, and there aren't enough resources for both, then the previous one gets shut down
- Process priority determines what gets killed (like *nix)
    - active, visible, service, background, empty
- Lower priority processes are killed when resources are tight

## RESOURCES

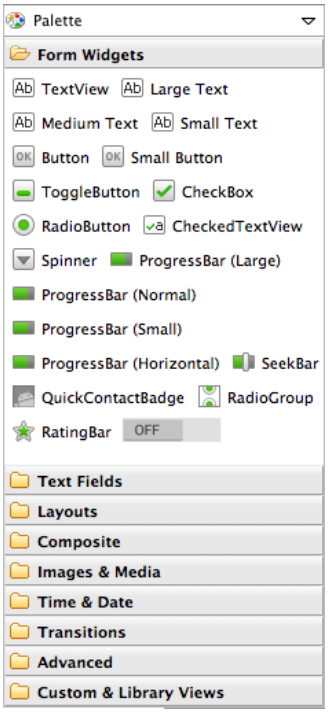

- Resources stored externally – don't declare Strings in code
    - Makes it easier to swap out another file in a different language
- Good for tailoring language, devices, and for re-use
- Can also include arbitrary xml, binary data
- Accessed in code through **R** instance
- In example:
    - Have icons, layouts, etc saved
    - All this gets compiled into the app itself and you can draw on these resources

## LAYOUTS AND VIEWS

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

- Declare all user interface elements in XML
- Activity will put up particular views and put them into layouts
- Example (diagram) is a linear layout

## MATERIAL COMPONENTS




- Each widget is the basis for a unit of action with the user.
- Custom widgets are derived from View

## APPLICATION MANIFEST

- Gives the details of the app
- Every .apk contains resources, code, and a **manifest file**
- Specifies the Java package for the app – a unique id
- All components are declared in AndroidManifest.xml
    - Permissions (e.g. internet access, read/write content)
    - Minimum API level (e.g. Eclair, 8)
    - Hardware and software features (e.g. camera, Bluetooth)
    - External APIs (e.g. Google Maps)
    - and more...

# Material Design

Assigned reading:

1. https://material.io/components
2. https://material.io/design/usability/accessibility.html#hierarchy
3. https://vula.uct.ac.za/access/content/group/57872ff9-5c75-4def-b501-80c784752dcf/mobile%20design%20and%20development/Shrine%20-%20Material%20Design.pdf

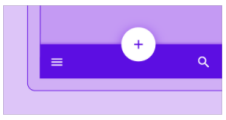
- Design principles for Android development
- By using consistent design principles, you can have a consistent experience across devices, different form factors, types of mobiles devices and different kinds of phones.
- Guideline on how android interface elements should appear and interact
- Key Principles
    - Material is Metaphor: inspired by physical world and textures
        - How we visualize and compose apps leverages our experience with the physical world and textures
    - Create layouts that are bold, graphic, intentional, planned: guided by print methods TYPOGRAPHY, grids, space, scale, color and imagery
        - Done so that we can appeal to people's sense of design
    - Motion provides meaning: use motion to focus attention and reorganize the environment with coherent transitions and subtle feedback
        - E.g. when we move from one activity to another, there's a motion

## MATERIAL PROPERTIES

- What can materials do?
- Characteristics:
    - Solid
    - Occupies Specific Unique Points in Space
    - Impenetrable
    - Mutable shape
    - Changes in size only along plane
    - Unbendable
    - Can join to other material
    - Can separate, split, heal
    - Can be created or destroyed
    - Move along any axis
- Physical Properties
    - Varying x & y dimensions
    - Uniform thickness (1dp)
    - Casts shadows naturally from relative position
    - Content is displayed on material, but does not add thickness
    - Content is independent but stays within bounds
    - Solid
- Transforming
    - Changes shape
    - But not bends or fold

    - Can join sheets
    - Can heal
- Movement
    - Spontaneous generation or destruction
    - Move on any axis
    - Z-motion on interaction with user

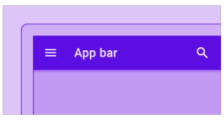
## MATERIAL COMPONENTS



App bars: bottom

A bottom app bar displays navigation and key actions at the bottom of mobile screens

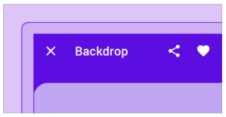
Android iOS Flutter




App bars: top

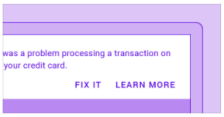
The top app bar displays information and actions relating to the current screen

Android iOS Web Flutter




Backdrop

A backdrop appears behind all other surfaces in an app, displaying contextual and actionable content



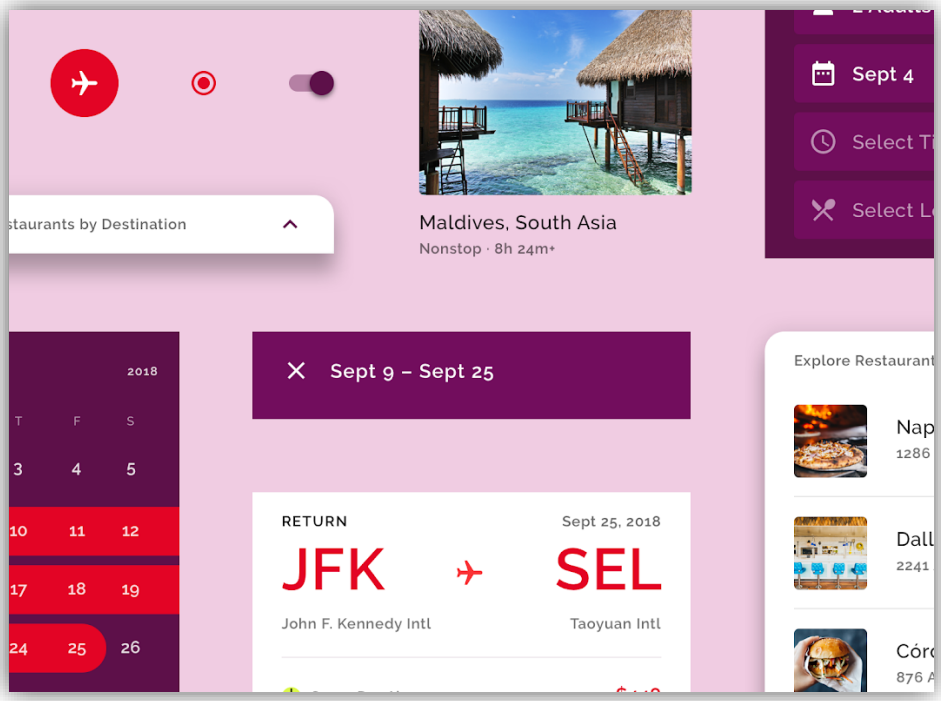

Banners

A banner displays a prominent message and related optional actions

iOS Flutter

- Material properties manifest in part as material components
- Interactive building blocks for creating an interface
- Material Design comes with a set of open source widgets that you can use to assemble the interface.
    - Use of these widgets increases consistency and learnability of an interface.
    - Can customize each of the building blocks – e.g. changing colours to match
- The guideline outlines how each component should be used.
- Read: Components (https://material.io/components)
- How to put all components together?

## CASE STUDY




- Material Case Study: Crane
  - Different components that're available have been assembled into the individual activities and screens
  - Supports content filtering
  - Simple high-level sections
  - Grid System – orderly and organizable
    - Facilitates responsive design for web apps – use grid to re-arrange items to fit better in the available space
  - Consistency across devices (tablets, PCs)
  - Use of color and shadow to create layering
    - Two colours – one primary, one secondary
    - Consistency of colour
  - Components put together in a series of layers


Maldives, South Asia
Machu Picchu, Peru
Sept 9 - Oct 6
October
Checkout
SEL
JFK
DEPART
RETURN
Find Trips
My Trips
New York, USA
New York, USA

# Navigation

- Schneiderman: filter, overview, details on demand

BARS


1 Status Bar
2 Navigation Bar
3 Combined Bar

- All these bars can be dimmed or hidden depending on what you want your application to look like.
    - Some apps make more sense in full screen!
- Combined bar used in web context
- Details on demand – fade away when you don't need them, come back when you do
- Actions that help direct user attention:
    - Lean Back – swipe to bring them back
    - Immersive – tap to bring them back
    - Lights out – automatically fade the bars after inactivity - deprecated

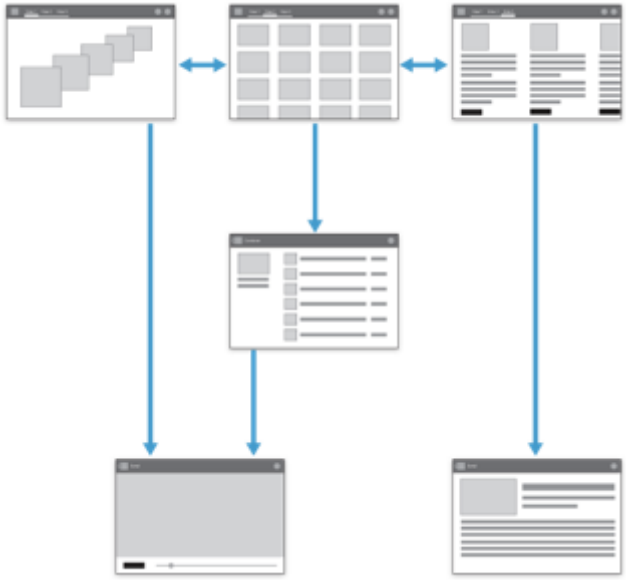
# Filtering

- Timeline interface: have stack of things you're looking at

- Grid: lots of thumbnails
    - Scroll through thumbnails to get to the next stage

- Catalogue view: scroll left and right to see summaries of each of these objects

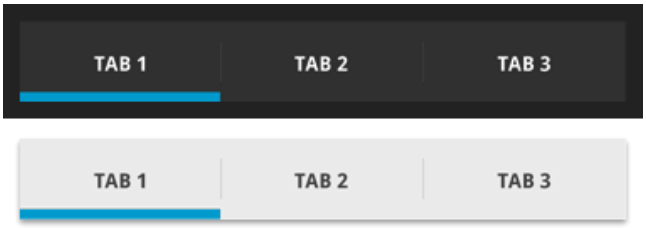
- With each of these top lvl views, if you tap on any of these material objects (interacting with them), will bring up another view
    - Could go into category views, detail/edit views



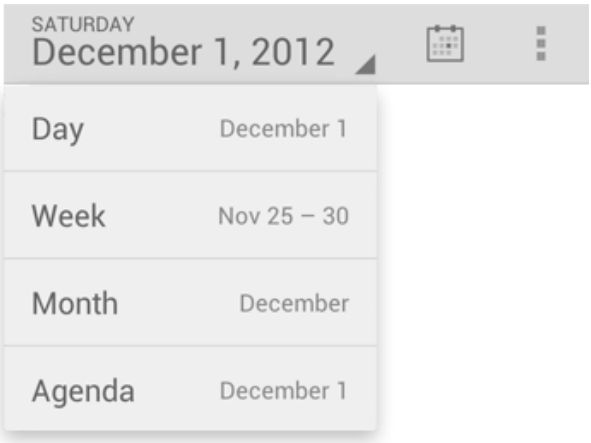
- Need to understand how to take lots of info and make it accessible on a small screen
    - Let user tell you what's important to them – demand detail when they need it

## TOP-LEVEL SWITCHING

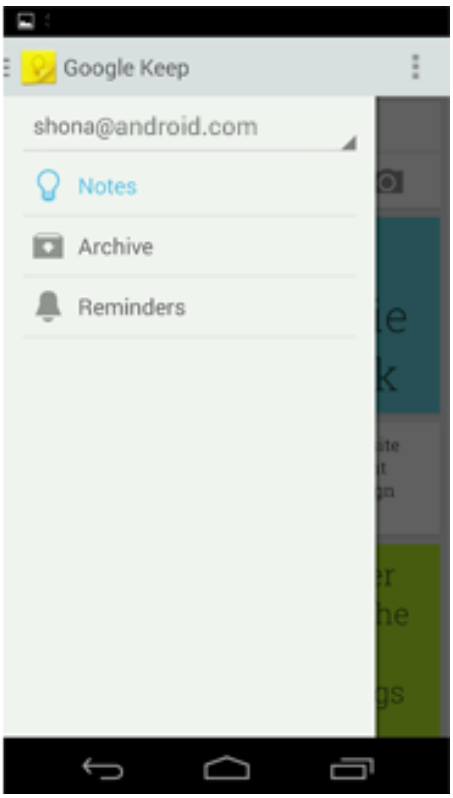
- Ways of supporting filtering within the app
- **Fixed Tabs**
    - You expect your app's users to switch views (modes) frequently.
    - You have a limited number of up to three top-level views.
    - You want the user to be highly aware of the alternate views.




- **Spinners**
  - You don't want to give up the vertical screen real estate for a dedicated tab bar.
  - The user is switching between views of the same data set or data sets of the same type.




- **Navigation Bar**
  - You have a large number of top-level views.
  - You want to provide direct access to screens on lower levels.
  - You have particularly deep navigation branches.



# Action Bar


1. App icon

The app icon establishes your app's identity. It can be replaced with a different logo or branding if you wish. Important: If the app is currently not displaying the top-level screen, be sure to display the Up caret to the left of the app icon, so the user can navigate up the hierarchy. For more discussion of Up navigation, see the Navigation pattern.


*App icon with and without "up" affordance.*

2. View control

If your app displays data in different views, this segment of the action bar allows users to switch views. Examples of view-switching controls are drop-down menus or tab controls. For more information on view-switching, see the App Structure pattern.

If your app doesn't support different views, you can also use this space to display non-interactive content, such as an app title or longer branding information.

3. Action buttons

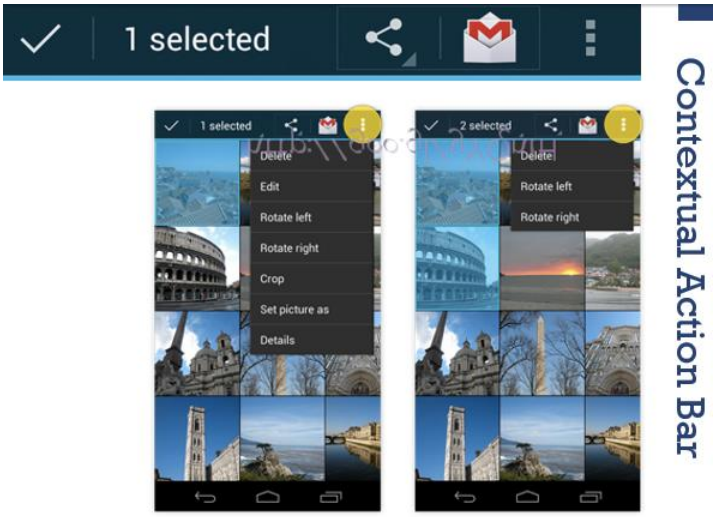
Show the most important actions of your app in the actions section. Actions that don't fit in the action bar are moved automatically to the action overflow. Long-press on an icon to view the action's name.

4. Action overflow

Move less often used actions to the action overflow.

1) App Icon
2) View Control – embodied as spinner in diagram
3) Action buttons – things that you can do with other things that're on the screen
4) Action overflow – any actions that are less used (i.e. don't fit in the simplified visual that's there) goes into that action overflow (app bar)

- App bars are flexible in the that you can make them contextual
  - App bar may have different contexts depending on what you're doing at any given time
- Instead of secondary/context menus, you should use the contextual action bar
- You can dynamically adjust the action bar menu items based on what is selected

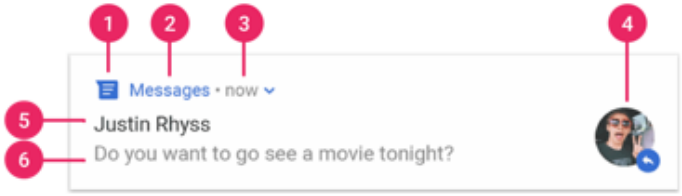


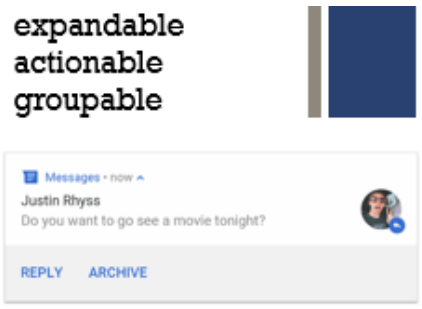

- Example:
  - On the left, they can do many things when one picture is selected
  - When they select two, anything that can't be applied to two pictures at once gets removed from that list

# Notifications

- Using notifications makes it easy to call people's attention back to the app without them having to launch the app themselves
- As users, don't have to open app to find info we want.
- Built into the design of these widgets are design principles
- Guidelines for Android notifications seek to increase the signal to noise ratio, and to show the most important information first.
- Use of mapping and clear affordances helps people to know what to do with a notification at a glance.
- And use of channels and importance levels gives control to the user over notifications.

## DRAWER NOTIFICATIONS



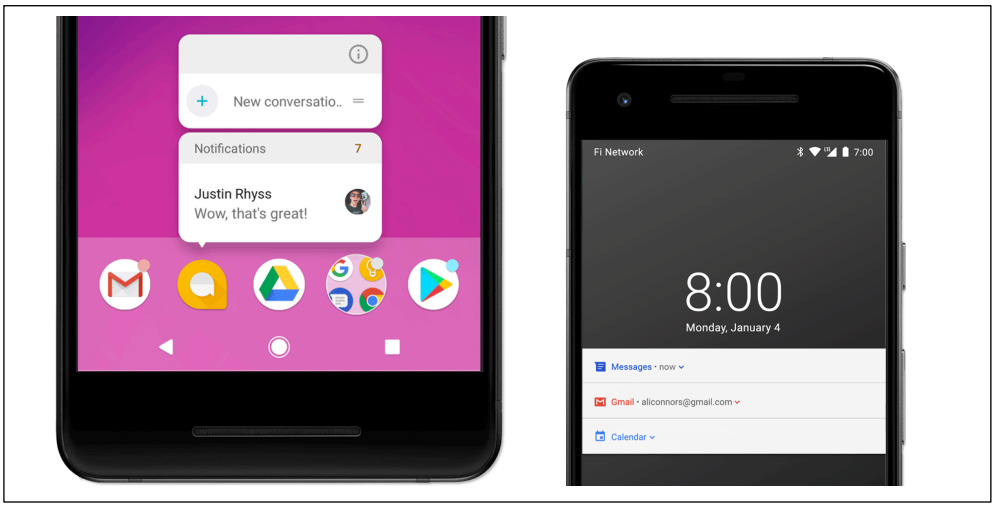




1. Small icon: This is required and set with `setSmallIcon()`.
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with `setWhen()` or hide it with `setShowWhen(false)`.
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set with `setLargeIcon()`.
5. Title: This is optional and set with `setContentTitle()`.
6. Text: This is optional and set with `setContentText()`.

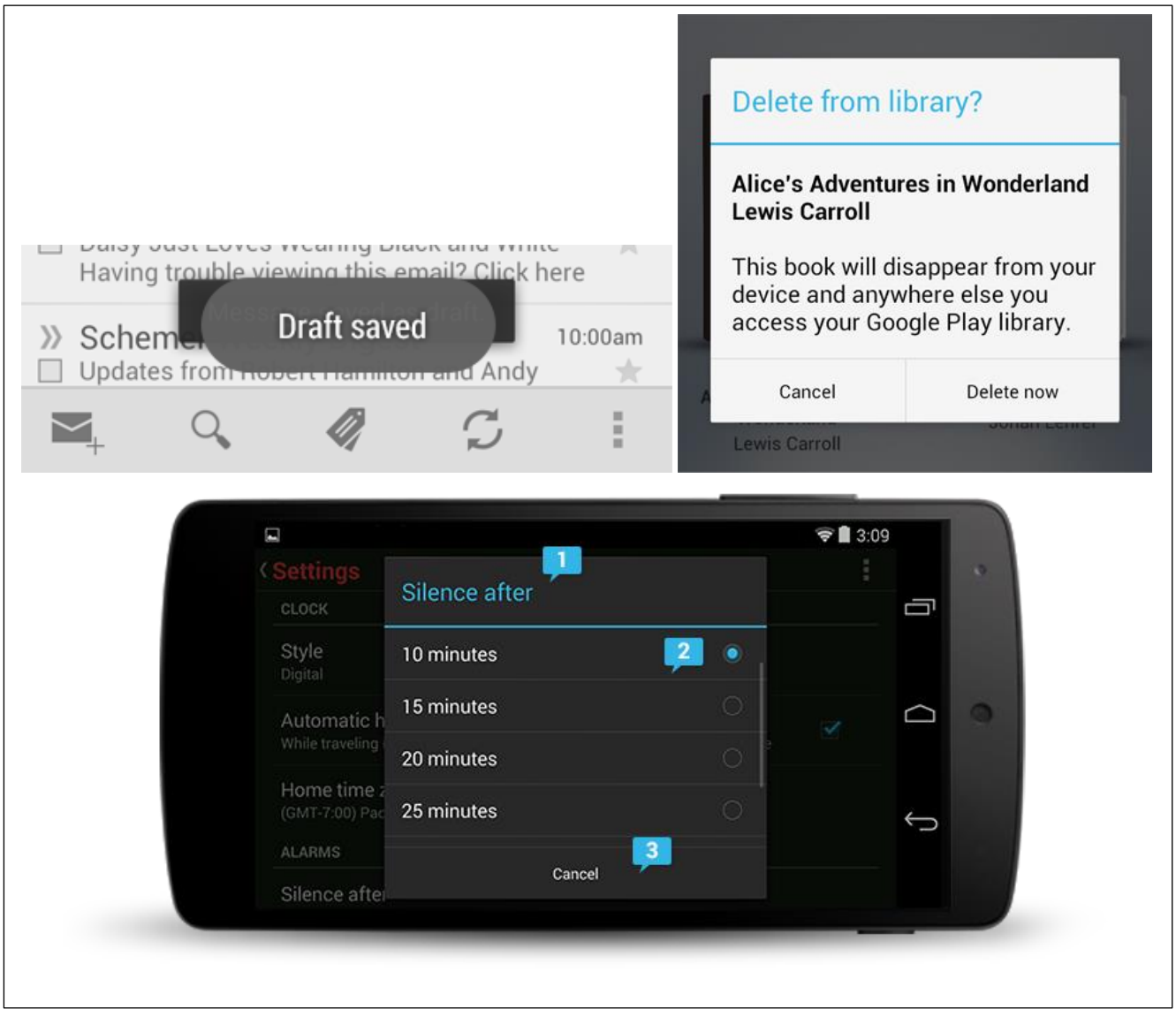
- Expandable – e.g. message in example diagram could show just sender or entire message
- Actionable – e.g. can interact with message without having to launch app
- Groupable – e.g. if you receive multiple messages from the same person, will be grouped as one person
- Drawer – chronologically & priority sorted, swipe to remove unless it is an ongoing notification
- Short: 1-2 lines, optional timestamp
- Time sensitive events - especially if they involve other people
- Use Stacked notifications rather than multiple notifications – expand to view more
- Mapping – e.g. created by small icon and app name
- Clear affordances – e.g. reply and archive invites touch
    - Both mapping and clear affordances help tell you what you can do with a msg notification at a glance

## OTHER NOTIFICATIONS

- Heads-up Notification – a floating window above the current app
- Lock-screen – can change what content appears or not for privacy
- App icon badge, and notification dot




## IN-APP NOTIFICATIONS



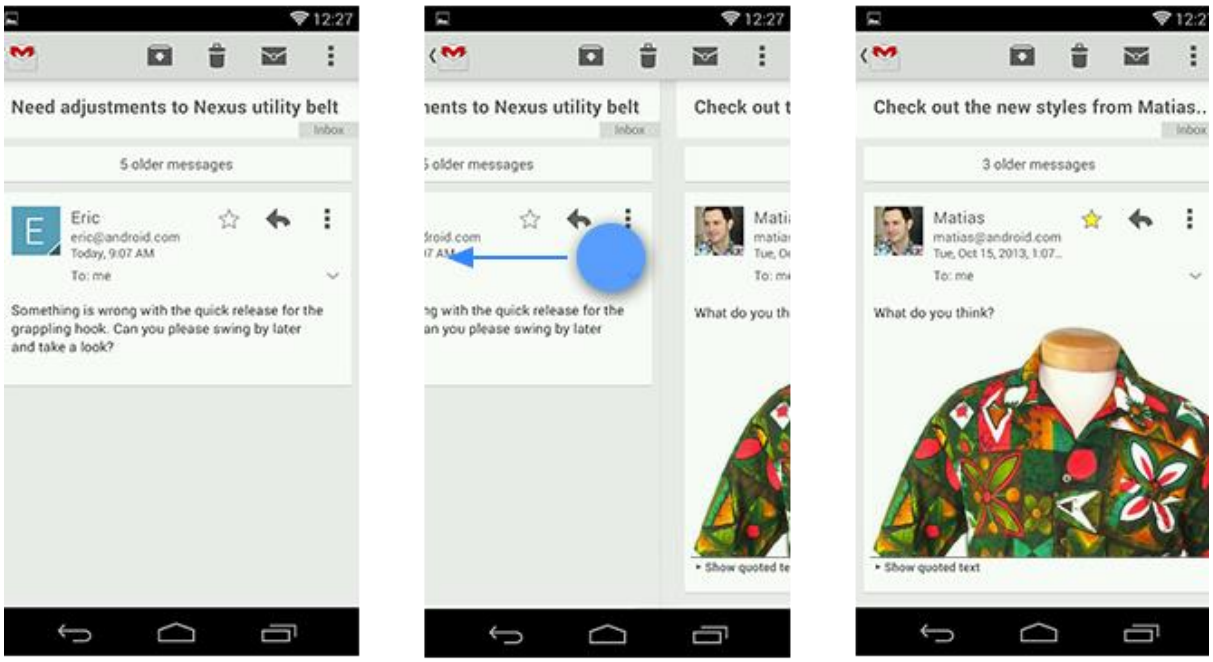

- **Alerts** for warnings and confirmations
- **Snackbars** and **Toasts** for lightweight feedback
- **Dialogs and Popups** for decisions and actions

- Choose interface element based on the right level of interaction that you want
    - E.g. if frequent, don't have dialog pop up everytime

## Gestures

- There are specifications for the design of the gesture

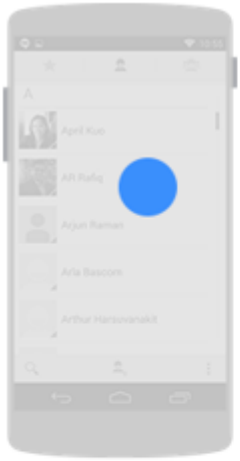
Swipe between detailed views




### OTHER GESTURES

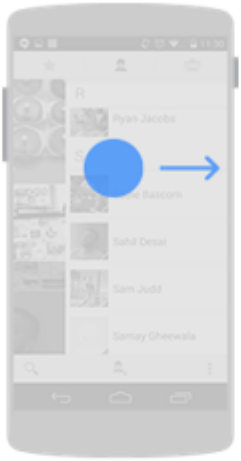
- Natural and intuitive interaction with objects on the screen
- Direct manipulation – continuous actions where it's reflecting back at you what you're doing as you're doing it

## Gestures



**Touch**
Triggers the default functionality for a given item.

**Action**
Press, lift



**Long press**
Enters data selection mode. Avoid using long press for showing contextual menus.

**Action**
Press, wait, lift

**Swipe or drag**

Scrolls overflowing content or navigates between views in the same hierarchy.

**Action**
Press, move, lift

## Gestures

**Long press drag**
Rearranges data within a view, or moves data into a container

**Action**
Long press, move, lift

**Double touch**
Scales up. Also used as a secondary gesture for text selection.

**Action**
Two touches in quick succession

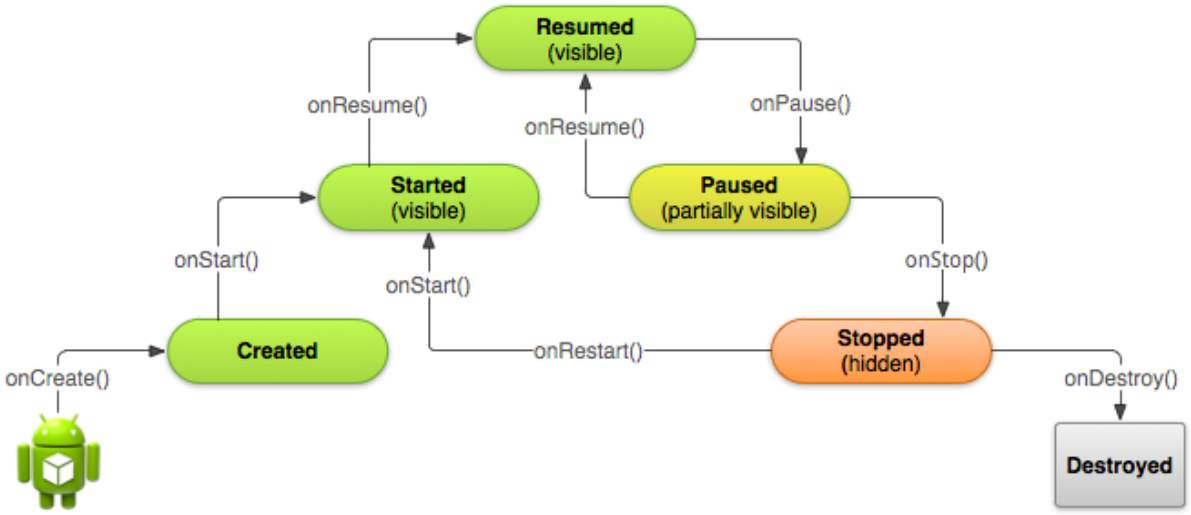
**Pinch**
Zooms into/out of content.

**Action**
2-finger press, move outwards/inwards, lift

# Activities

- Android apps and how all components fit together and communicate with os

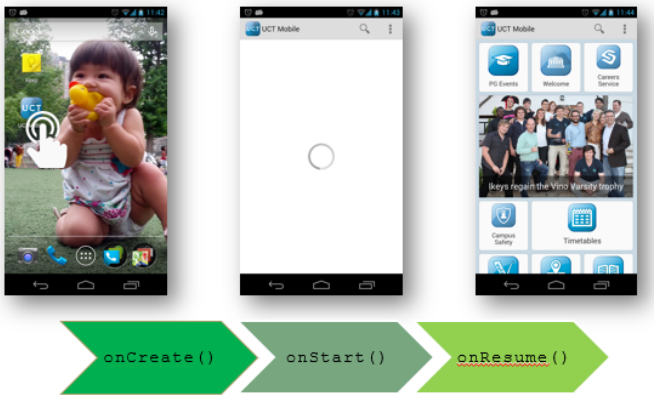
## Activity Lifecycle




- Different states and callbacks help the OS to manage resources and application interrupts
- Any given app goes through different aspects of activity life cycle
    - Resumed = app that's actively in use
    - Stopped = no longer visible but still running
- Callbacks are important – get called anytime a state transition is happening
    - Integrated into code to manage data in app, what's being communicated to user about state of app, what screens are being show etc
- App can go through the lifecycle many times before it gets destgroyed
- Different states support the interaction between OS and mobile apps to ensure resources are being properly managed

EXAMPLE

1. Initial Launch of Activity




- Click on app
    - onCreate() for that apps main activity gets called
    - OS goes and checks app manifest, figures out what class corresponds to the launcher activity
    - Opens up activity – onCreate() method on that activity gets called
    - After that onStart() gets called
    - Once app ready to be displayed, onResume() gets called

## Example: UCT Mobile

2. An alarm goes off, application is paused


- App itself is paused, onPause() and onStop() called in sequence
- Alarm has onCreate(), onStart() and on Resume() being called
- Alarm takes up the entire screen
    - Dismiss the alarm – series of methods called corresponding to the state changes that we expect activity to go through

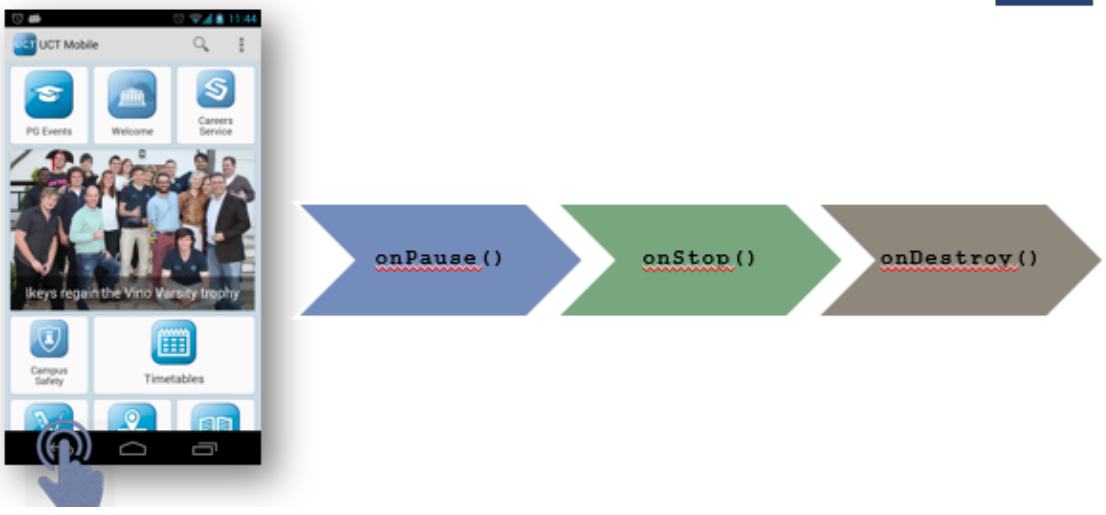
## Example: UCT Mobile

### 3. The alarm is dismissed, application resumes


- Want alarm to go away – currently resumed
- Needs to be paused, stopped and then destroyed
    - As these steps being taken, onPause(), onStop() and onDestroy() being called
- OS knows that previous app that was running is still the primary app that should be on the screen, automatically calls onRestart(), onStart() and onResume() to make app reappear on the screen
- What if back button pressed?
    - What is Android convention for when this happens?
    - Have to map back button to the idea of "back" – but for a given user, what is "back" isn't necessarily clear

## Example: UCT Mobile

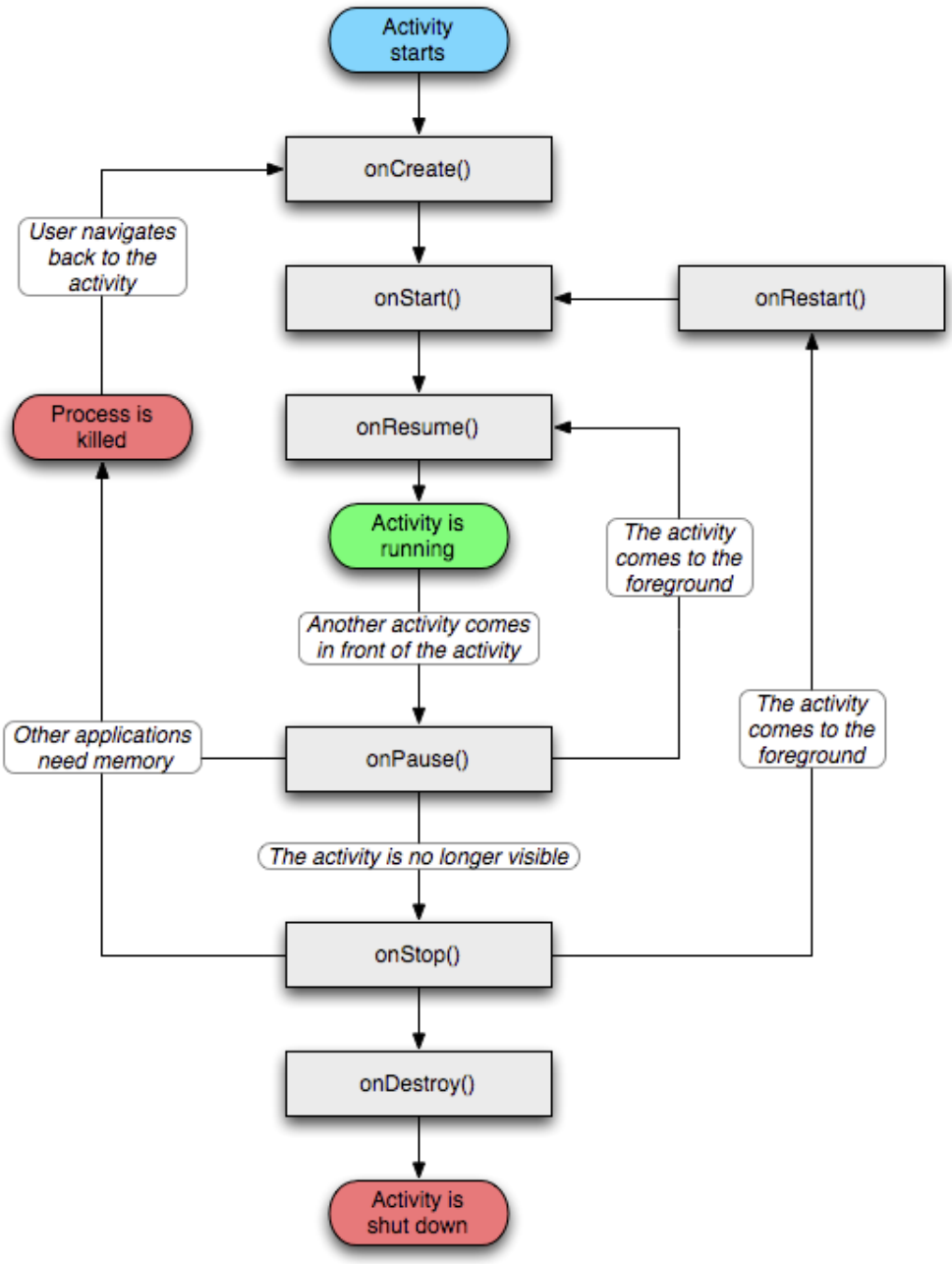
### 4. Back button is pressed




- Close app, or if several views in, it takes you back to previous view
- At top lvl view, back button closes app
    - onPause(), onStop() and onDestroy() gets called

# Interruptions

- Lifecycle management is especially important due to resource management constraints
    - Resources are e.g. what memory device has, speed of device etc
- OS manage multiple apps and ensuring each app is properly managing its state




## Implementation

- Be aware of the code of the activity and how it's represented in the app manifest file

# The Activity Class

- Functions need @Override
- Only implement functions that you need (where you need to take care of a particular behaviour)
- Call the super class method
- Pay attention to what you are building and destroying
- In what part of the lifecycle is your app?

## EXAMPLE

```
lateinit var textView: TextView

// some transient state for the activity instance
var gameState: String? = null

override fun onCreate(savedInstanceState: Bundle?) {
    // call the super class onCreate to complete the creation of activity like
    // the view hierarchy
    super.onCreate(savedInstanceState)

    // recovering the instance state
    gameState = savedInstanceState?.getString(GAME_STATE_KEY)

    // set the user interface layout for this activity
    // the layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity)

    // initialize member TextView so we can manipulate it later
    textView = findViewById(R.id.text_view)
}

// This callback is called only when there is a saved instance that is previously saved by using
// onSaveInstanceState(). We restore some state in onCreate(), while we can optionally restore
// other state here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in onCreate().
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    textView.text = savedInstanceState?.getString(TEXT_VIEW_KEY)
}

// invoked when the activity may be temporarily destroyed, save the instance state here
override fun onSaveInstanceState(outState: Bundle?) {
    outState?.run {
        putString(GAME_STATE_KEY, gameState)
        putString(TEXT_VIEW_KEY, textView.text.toString())
    }
    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState)
}
```

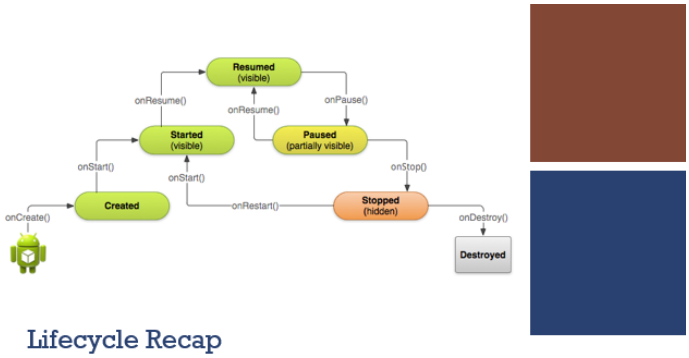
- onCreate() method – what should happen at point at which it's created
    - Try and retrieve saved instance state
    - Set content view to main activity
    - Text view assigned to a resource
- onRestoreInstanceState() – knows how to retrieve previous saved instance and load it into memory so that it can be used by the activity
- onSaveInstanceState() – takes current state and save it to a data bumdle

## User Navigation

- Handle **Home** and **Back** buttons gracefully
- It's important to distinguish between the navigation buttons
- *Most users don't know the difference – be gracious!*
- Home button
    - Calls onStop(), onStart()
    - The app is stopped but not destroyed
- Back button
    - Goes back to previous activity (unless you've overridden this)
    - Calls onStop(), onDestroy()
- Important when implementing multiple Activities

## Screen Rotation

- What happens to your app?
- Operating System Calls:
    - onPause()
    - onStop()
    - onDestroy()
    - onCreate()
    - onStart()
    - onResume()
- Your Activity will be destroyed and recreated each time the screen rotates
- Solution 1:
  Disable Rotations (can only view app in portrait mode)
  <activity …
  android:screenOrientation="portrait"
- Solution 2:
  Take current state and save it into a bundle, so that when screen starts in different orientation, able to restore that state and populate it into the activity again
  onSaveInstanceState()
  *called before onStop()*
  onRestoreInstanceState()
  *called after onCreate(), onStart()*




## Lifecycle Recap

- Activity lifecycle methods are called automatically depending on the OS needs and the stage they are in.
- Initiate the UI in onCreate()
- Use onSaveInstanceState() and onRestoreInstanceState() to preserve state

## Activities Recap

- 1:1 correlation with screens
- An app might have many Activities
- Use <intent-filter> to denote initial activity
- Activities are called by other Activities

```
public class YourActivity extends Activity
{
    public void onCreate(Bundle state) {...}
    public void onStart() {...}
    public void onRestart() {...}
    public void onResume() {...}
    public void onPause() {...}
    public void onStop() {...}
    public void onDestroy() {...}
}
```

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

# Intents and Bundles

- Idea of state and interactions between activities
- Activities do not have an explicit constructor – they use an indirect calling mechanism called an **intent**
    - Rather than you calling an activity, they're constructed by the OS, when the OS receives an intent for the activity to be run

## Intent

```
Intent intent = new Intent(this, SecondActivity.class);
startActivity(intent);
```

- Activity is associated with launch icon in AndroidMansifest.xml

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

## INTENTS

- A powerful mechanism for accessing and exposing Activities from/to other Activities
- Structure consists of an **action**, and a **data uri**
    - Can start an activity with that intent – OS will look through manifest file and know which app to start and give it the data that it needs to be able to access the data that's of interest.

```
Intent intent =
    new Intent(Intent.ACTION_DIAL,
                "content://contacts/people/1");
startActivity(intent);
```

- Known actions are **static final String**s in the Intent class
- URIs specify the data upon which the action should occur

## SHARING DATA

- Use the state **Bundle** to move data between activities
- onCreate(Bundle savedInstanceState) – pass instance state along to new intents
- Setting up data

```
Intent intent = new Intent(this, SecondActivity.class);
Bundle extras = new Bundle();
extras.putString("VALUE", value);
intent.putExtras(extras);
startActivity(intent);
```

- Receiving data

```
protected void onCreate(Bundle savedInstanceState)
{
    ...
    Bundle bundle = getIntent.getExtras();
    String value = bundle.getString("VALUE");
    ...
}
```

## RECEIVING DATA FROM ACTIVITIES

- Requesting Data: pick contact request

```
public class MyActivity extends Activity {
    ...

    static final int PICK_CONTACT_REQUEST = 0;

    protected boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
            // When the user center presses, let them pick a contact.
            startActivityForResult(
                new Intent(Intent.ACTION_PICK,
                new Uri("content://contacts")),
                PICK_CONTACT_REQUEST);
           return true;
        }
        return false;
    }

    protected void onActivityResult(int requestCode, int resultCode,
            Intent data) {
        if (requestCode == PICK_CONTACT_REQUEST) {
            if (resultCode == RESULT_OK) {
                // A contact was picked.  Here we will just display it
                // to the user.
                startActivity(new Intent(Intent.ACTION_VIEW, data));
            }
        }
    }
}
```

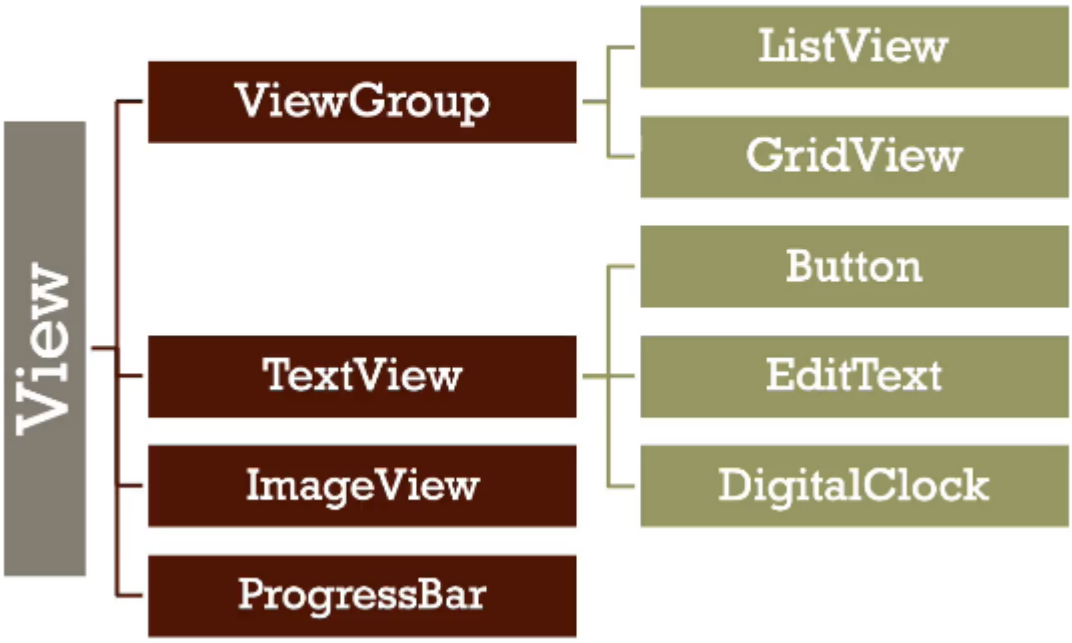
- Returning Data

```
Bundle bundle = new Bundle();
bundle.putString(key, value);
Intent intent = new Intent(action, data);
intent.putExtras(bundle);
setResult(RESULT_OK, intent);
finish();
```

- Cancelling Return

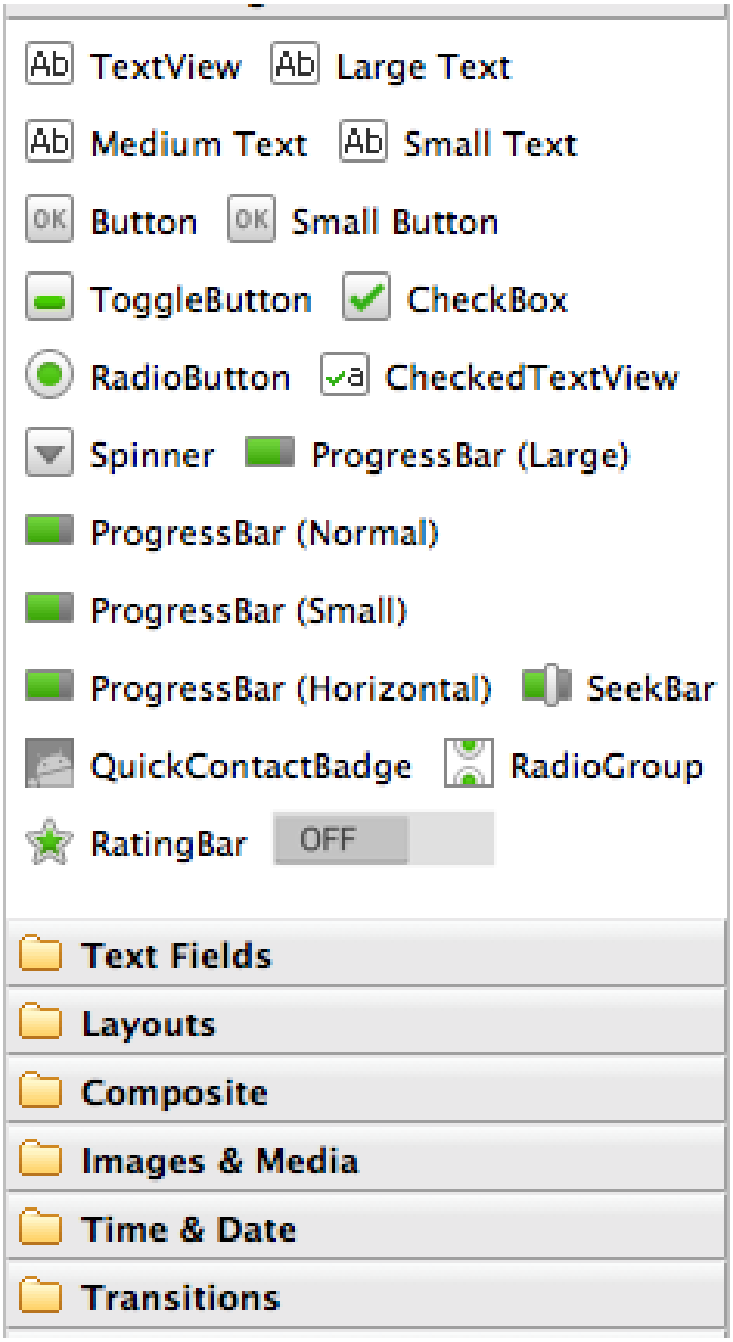
```
Intent intent = new Intent();
setResult(RESULT_CANCELLED, intent);
finish();
```

# UI Class Hierarchy




- Views are the basic building blocks for all UI components (like Canvas in Java applications)
- Essentially, it's a blank drawing board, with possible interactive capabilities
- Base class for all widgets
    - Class hierarchy represents viewgroup, textview etc as specific classes that're directly inheriting from view.
    - Some widgets will also have subclasses that that have the same interactions
- You can write custom views from scratch – chess board, game widgets
- Multiple views can be composed using viewgroups

## WIDGETS




- Every widget is an instance of a view
- Basic UI components with:
    - Attributes that control appearance
    - Event handlers to provide interactivity
        - Receive all gestures, typing etc
    - Dimensions
        - **wrap_content** dynamically sized; can also have fixed height and width
        - **fill_parent** expands to fill parent view

### Common widgets

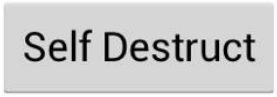
- Button

- Self-destruct button
- Important stuff coded in XML file as part of app manifest

```
<Button
    android:id="@+id/btn_self_destruct"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:text="@string/self_destruct"
    android:onClick="selfDestruct" />"

public void selfDestruct(View view)
{
    // kablooey
}
```


- Could do away with XML and put what happens with button explicitly in code
    - In example, what happens when button clicked (click listeners)

```
<Button
    android:id="@+id/btn_self_destruct"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:text="@string/self_destruct"
    android:onClick="selfDestruct" />"

Button button = (Button) findViewById(R.id.btn_self_destruct);
button.setOnClickListener(
    new View.OnClickListener() {
        public void onClick(View v)
        {
            selfDestruct();
        }
    });
```




## Edit Text

- Specify dimensions, hint, etc
- Can interact with text in widget

```
<EditText
    android:id="@+id/editText1"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:hint="@string/enter_msg"/>
```


```
EditText editText = (EditText)findViewById(R.id.editText1);
String msg = editText.getText().toString();
```

TextView

```
<TextView
    android:id="@+id/textView1"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:text="@string/msg"/>
```

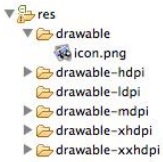
TextView

```
TextView textView = (TextView)findViewById(R.id.textView1);
textView.setText("Hello World!");
```

ImageView

- Can access drawables from within resources directory

```
<ImageView
    android:id="@+id/imageView1"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:src="@drawable/icon"/>
```



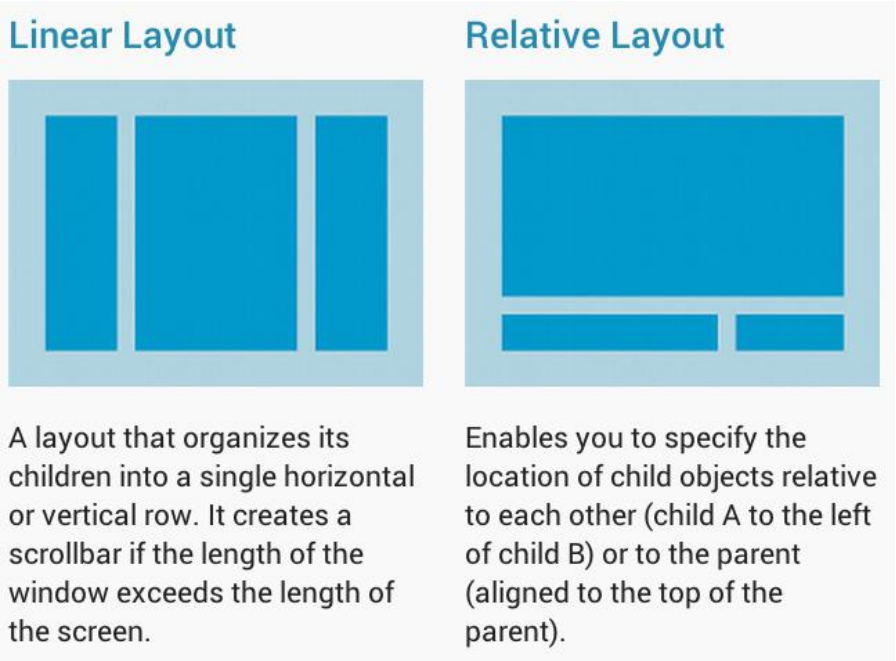

```
ImageView imageView = (ImageView)findViewById(R.id. imageView1);
imageView.setImageResource(R.drawable.icon);
```

# Layouts

- Android platform uses layouts to position objects together
    - Helps to position objs relative to one another, rather than on a specific x-y grid within a screen.

## ViewGroup and Layouts

- The **ViewGroup** class is the base class for layouts
    - Views composed into a group of views together
- Invisible containers that hold other Views and define their layout properties
- Can be **declared** in XML or **instantiated** at runtime
- Can be nested
- Common Layouts:




### RELATIVELAYOUT

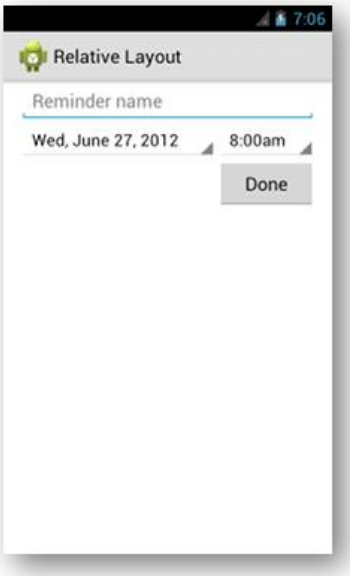
- Displays child views in relative positions
- Relative to:
    - Sibling elements, e.g. left of or below
    - Parent, e.g. bottom, left of center
    - Specify with an alignment

# RelativeLayouts: Alignment

- layout_alignParent[Top, Bottom, Left, Right]
  positions the widget at one of the edges of the parent window.
- layout_above
  positions the widget above another widget (using id)
- layout_below
  positions the widget below another widget (using id)
- layout_toLeftOf
  positions the widget to the left of another widget (using id)
- layout_toRightOf
  positions the widget to the right of another widget (using id)
- layout_align[Top, Bottom, Left, Right]
  positions the widget so that the edge is aligned with the corresponding edge of another widget (using id)
- layout_center[Vertical, Horizontal]
  positions the widget in the center (vert or horiz) in the parent
- layout_centerInParent
  centers the widget in the parent window

## XML Example

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <Spinner
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
    <Spinner
        android:id="@id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
    <Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```




CSC2002S, 2020

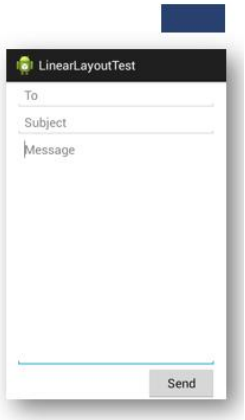
## LINEAR LAYOUT

- Aligns all children in a single direction, vertically or horizontally
- Specify the layout direction:
  - android:orientation="horizontal"
  - android:orientation="vertical"
- Use weights to adjust relative size of children

- android:weight="1"
- Can nest layouts within the layout

## XML Example

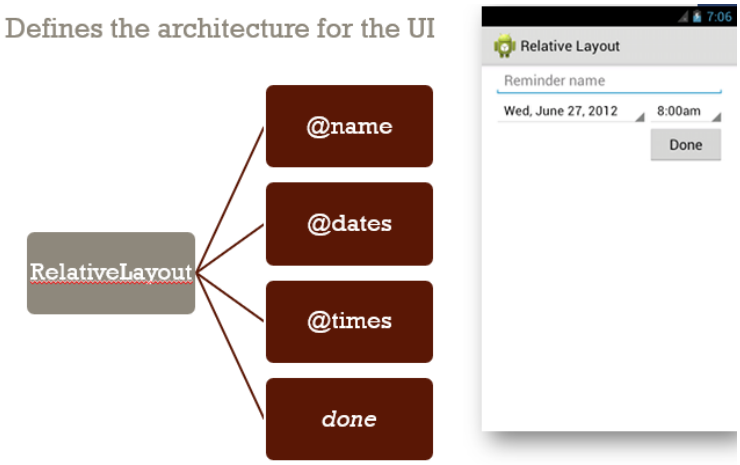
```
<LinearLayout android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />

    <EditText
        android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />

    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```




CSC2002S, 2020

## Relationship Chart


## LAYOUT RESOURCES




- Defines the architecture for the UI
- Specified as xml files in res/layout/*filename.xml*

- Compiled into apk, resolves to a **View**
- Resource reference:
    - In Java/Kotlin  R.layout.*filename*
    - In XML  @[*package*:]layout/*filename*
- android:layout_height and android:layout_width
    - **match_parent** sets the dimension to match parent element
    - **fill_parent** same as match_parent but deprecated
    - **wrap_content** sets the size to fit content

## Resource Identifiers

- android:id=@[*+*]id/*name_of_identifier*
- Ways to reference particular things in the resource files
- You can name elements in your resource files, and reference them later, both in XML and in Java/Kotlin
    - XML: android:id=@id/*name*
    - Java/Kotlin: findViewById(R.id.*name*);
- **@** denotes a named element
- **+** denotes the first use of the name (adding to namespace)
- Can also name all sorts of other resources in addition to ids:
    - String, color, drawable, layout, menu, style