FA1 Digital Solutions

Isabel Lifu

2022

Unit 1

Diagram

Description automatically generatedE

## Mind map of the solution

# Decomposing the problem – Part 1

# Decomposing the problem – Part 2

Graphical user interface, text

Description automatically generatedGraphical user interface, diagram, text, application

Description automatically generatedDiagram

Description automatically generatedDiagram

Description automatically generated

## Selected Proto persona

## Functional/Non-functional requirements

## Requirement Importance

## Overall breakdown

# **Development – UML**

## Use Case Diagram

Diagram

Description automatically generated

## Activity Diagram

Diagram

Description automatically generated

Diagram

Description automatically generated

Changes to initial idea: Update order, display queue, and bill calculation are all under the “Place order” use case, with the customer as the stakeholder.

## Class Diagram

Shape, polygon

Description automatically generatedShape

Description automatically generatedGraphical user interface, application

Description automatically generatedDiagram

Description automatically generated**Development – UI Sketches**

Functional req.:

Sends order to queue aka takes order

Functional req.:

Removes order when clicked on

Functional req.: Takes order when button is pressed

Accessibility: Large and easy to find buttons

Efficiency: multiple items displayed in one go reduces scrolling

Safety: familiar icons

Satisfaction: Success message lets user know order is in queue

Satisfaction: Greeting is friendly

Memorability: Once user is familiar with these buttons, it is easy to repeat the process

Accessibility: Large and easy to find buttons

Learnability: symbol has explanation

Safety: familiar help icon

Large buttons increase accessibility

**Development – Grid Layout Wireframes**

## Annotated interface design table for menu page

|  |  |  |
| --- | --- | --- |
| UI Sketch VS Kivy Output | Kivy Code | Annotations |
| Shape  Description automatically generatedUI SketchKivy output according to adjacent code | <MyGrid>  home: welcomePage  orders: orderList  GridLayout:  background\_color: 0,0,0,0  size: 100,50  pos: 50,250  cols: 3  Button:  text: "Home"  id: welcomePage  multiline: False  on\_press: root.pressed()  size: 50,50  pos: 50,250  size\_hint: 1, None  Button:  text: "Help"  id: support  multiline: False  on\_press: root.pressed()  size: 50,50  pos: 50,250  size\_hint: 1, None  Button:  text: "View order list"  id: orderList  multiline: False  on\_press: root.pressed()  size: 50,50  pos: 50,250  size\_hint: 1,None  meal: orderItem  help: support  GridLayout:  cols: 1  size: root.width - 100, root.height - 100  pos: 50,50  GridLayout:  Label:  text: "Menu"  pos: 350, root.height - 100  cols: 1  GridLayout:  background\_color: 0,0,0,0  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: root.width /2, root.height /2  pos: root.width/2,root.height/2  cols: 4  GridLayout:  background\_color: 0,0,0,0  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: root.width /2, root.height /2  pos: root.width/2,root.height/2  cols: 4  **Button:**  **text: "Add"**  **id: orderItem**  **multiline: False**  **on\_press: root.pressed()**  **size\_hint: None, None**  GridLayout:  Label:  text: "\n\n"  pos: 350, root.height - 100  cols: 1  #This code runs without error. | To develop the interfaces, Kivy code is used, which is a package of libraries that contain GUI objects for Python. For Kivy interfaces, both a main.py file and a my.kv file are required, as they are dependent on each other. For example, if the main.py file looks like the one below, then the kivy file will be named my.kv:  class MyGrid(GridLayout):  menu = ObjectProperty(None)  orders = ObjectProperty(None)  def pressed(this):  usertext = this.menu.text  orderstext = this.orders.text  print(menutext)  print(orderstext)    class MyApp(App):  def build(self):  return MyGrid()  if \_\_name\_\_=="\_\_main\_\_":  MyApp().run()  The purple-coloured code is repeated 8 times to draw 8 buttons for adding meals.  The blue code is added just to provide some space before the next row of buttons “Home”, “Help”, and “View order list”.  The green code is put last in the script so that it is drawn right after the empty space from the blue code.  Once again, the kivy output doesn’t entirely match the initial UI sketch, and that is because the buttons aren’t resizing as I intend them to. Consequently, the buttons for adding an order aren’t sitting neatly in the center of the screen. However, this interface is still learnable and functional for the customer to see the meals on menu and be able to tap on buttons. |

## **Development – Grid Layout Wireframes**

## Annotated interface design table for order list

|  |  |  |
| --- | --- | --- |
| UI Sketch VS Kivy Output | Kivy Code | Annotations |
| Graphical user interface, application  Description automatically generatedUI SketchKivy output according to adjacent code **Graphical user interface, application  Description automatically generated** | <MyGrid>  Label:  text: "Food 4"  pos: 50,100  background\_color: 0,0.5,1,0.5  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 100,75  pos: 50, 115  cols: 1  orders: orderList  GridLayout:  Label:  text: "Food description"  pos: 215,400  background\_color: 0,0,0,0  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 400,75  pos: 200, 415  cols: 1  Label:  text: "Food description"  pos: 215,300  background\_color: 0,0,0,0  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 400,75  pos: 200, 315  cols: 1  Label:  text: "Food description"  pos: 215,200  background\_color: 0,0,0,0  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 400,75  pos: 200, 215  cols: 1  Label:  text: "Food description"  pos: 215,100  background\_color: 0,0,0,0  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 400,75  pos: 200, 115  cols: 1  menu: customerMenu  update: updateOrder  remove: removeOrder  GridLayout:  cols: 1  size: root.width - 100, root.height - 100  pos: 50,50  GridLayout:  Label:  text: "Your orders"  id: orderList  pos: 50, 475  font\_size: 15  background\_color: 0,0.5,1,0.5  cols: 1  GridLayout:  background\_color: 0,0.5,1,0.5  size: 100,45  size\_hint: None, None  cols: 3  pos: 50,200  Button:  text: "Menu"  id: customerMenu  multiline: False  on\_press: root.pressed()  size\_hint: None, None  pos: 50,200  Button:  text: "Update order"  id: updateOrder  multiline: False  on\_press: root.pressed()  size\_hint: None, None  pos: 50,200  Button:  text: "Remove order"  id: removeOrder  multiline: False  on\_press: root.pressed()  size\_hint: None, None  pos: 50,200  GridLayout:  Label:  text: "Food 1"  pos: 50,400  background\_color: 0,0.5,1,0.5  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 100,75  pos: 50, 415  cols: 1  Label:  text: "Food 2"  pos: 50,300  background\_color: 0,0.5,1,0.5  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 100,75  pos: 50, 315  cols: 1  Label:  text: "Food 3"  pos: 50,200  background\_color: 0,0.5,1,0.5  canvas.before:  Color:  rgba: self.background\_color  Rectangle:  size: 100,75  pos: 50, 215  cols: 1 | The red section is an object declaration, which links the objects in kivy to python.  The yellow section is the main grid, which acts like a base for the rest of the content.  Light green is a title called “Your orders”, as seen in the kivy output in column 1.  The dark green and light blue sections are meal labels.  The dark blue section is the button group (Menu, Update order, Remove order).  Changes to interface design:  The menu and update order buttons are no longer at the top right corner, as initially planned. This is because buttons aren’t easily adjustable, unlike the images and adjacent food description labels.  Another change, following the same reasoning above, is the remove order button. Instead of having it sit next to every order, I changed it into a single button in the same grid layout as the Menu and Update order buttons.  Because it is now at the bottom of the page, it can be difficult to remove a meal. Therefore, the new interface can only support removing the entire order list.  This code runs without error. |

# **Development – Algorithms**

## Annotated Pseudocode to Python table for orderList()

|  |  |  |  |
| --- | --- | --- | --- |
| Pseudocode Adjustments | Coding Construct | Python Code | Code Annotations |
| Initial pseudocode for orderList():  BEGIN orderList()          CALL customerMenu          IF mealItem.selected=TRUE                  SET mealItem TO orderList                  DISPLAY successMsg                  CALL customerMenu          ELSEIF mealItem.selected = FALSE                   CALL customerMenu()                   DISPLAY cancelMsg                   CALL customerMenu          ENDIF  END  What changed:  mealItem.selected is an oversimplified version of the Python code for the algorithm. The mealItem seems like one object, therefore one selection, unlike in the code, mealItem is actually split into two components (type and name), meaning two selections are necessary already.  What remained the same:  Both the initial pseudocode and the Python code contain a confirmation selection. That is basically a yes or no question to confirm the order. It comes last out of the three inputs, further justified in the coding constructs column. | customerMenu is a local variable, can be updated by the restaurant, if necessary, but otherwise remains the same throughout the functions.  orderList is a list of dictionaries, each dictionary contains an unlimited number of mealItems.  customerMenu is called when customer initiates app, but also after a meal is selected until the orderList is confirmed complete.  cancelMsg and successMsg are all variables we can customise as we see fit  elif is used to avoid nested if statements  To increase accessibility, meals in the customerMenu dictionary comprise of both type and name variables rather than name only. This will narrow down the user’s menu based on the food type they want to see.  To provide safety for the user, a confirmation selection is contained at the end of every order. This ensures the user is making the right choice. Note that cancel is also an option for selection1 and selection2, additional safety. | def orderList():  totalOrder=[]  customerMenu = [  {"type":"dessert","name":"cake","price":int(7)},  {"type":"drink","name":"smoothie","price":int(8)},  {"type":"dessert","name":"icecream","price":int(5)},  {"type":"salad","name":"caeser","price":int(12)},  {"type":"drink","name":"lemonade","price":int(5)},  {"type":"burger","name":"beef","price":int(15)},  {"type":"dessert","name":"fudge","price":int(5)}  ]  selection1=str(input("Enter meal type:"))  if selection1=="cancel":  orderList()  print([row['name'] for row in customerMenu if row['type']==selection1])  selection2=str(input("Select a meal:"))  if selection2=="cancel":  orderList()  for row in customerMenu:  if selection2==row["name"] and selection1==row["type"]:  print(f"Order found || Type: {row['type']} || Name: {row['name']} || Price: {row['price']}")  orderItem=row["name"]  confirmation=str(input("Confirm order: [y/n] "))  if confirmation=="y":  print("Order successful!")  totalOrder.append(orderItem)  for item in totalOrder:  print(item)  if confirmation=="n":  print("Order discarded.")  orderList()  orderList() | Instead of using:  for row in customerMenu:     if row["type"]==selection1:              print(row['name'])  I used a one-line print:  print([row['name'] for row in customerMenu if row['type']==selection1])  The new output when executed becomes:  ['cake', 'icecream', 'fudge']  rather than:  cake  icecream  fudge  This makes the output more organised for the user.  The order in which input strings are called matter. For example, the input string confirmation must be defined after selection1 and selection2 have been successful, because the user cannot confirm on nothing selected.  The for loop is used for selection2 because unlike selection1, the print which comes after a meal type selection is more complicated. To reduce the complexity of the print, an f’string (fast string) is used.  totalOrder is a new dictionary where the orderItem is added.  This code runs without error. |

# **Development – Algorithms**

## Annotated Pseudocode to Python table for queueList()

|  |  |  |  |
| --- | --- | --- | --- |
| Pseudocode Adjustments | Coding Construct | Python Code | Code Annotations |
| Initial pseudocode for queueList():  BEGIN queueList()          CALL orderList          IF orderList !=NULL                  SET orderList TO queueList          ELSEIF clearQueue.selected=TRUE                  SET queueList TO NULL          ENDIF  END  What changed:  Because totalOrder is no longer an external text file, the whole algorithm needs to be rewritten.  BEGIN queueList()         CALL totalOrder         ADD totalOrder TO queueList         DISPLAY queueList  END  Added feature: Removable orders  BEGIN queueList()         CALL totalOrder         IF removeOrder == Yes                POP orderID         ENDIF  END | queueList is a transfer of data from the orderList to queueList  The queueList is a list of dictionaries. I chose to make it work like this because then it would be easy to refer to a dictionary within the queueList.  The order in which the orders enter the queueList matters, just like a line in queue. The first order that comes through can sit above the next, and so on.  To maximise efficiency of the popping, a new variable called “a” is introduced. “a” is any integer within the range of the number of order dictionaries in the list. That way, if the chef inputs 2, “a” will be 2, and hence order number 2 will be popped. | Initial python code:  def queueList():  queueList  for orderItem in totalOrder:  print(x)  clearOrder=str(input("clearOrder [y/n]"))  if clearOrder=="y":  orderList.write("")  if clearOrder=="n":  orderList.close()  queueList()  Updated python code:  def queueList():  totalOrder = [  {"type":"dessert","name":"cake","price":int(7)},  {"type":"burger","name":"beef","price":int(15)},  {"type":"drink","name":"lemonade","price":int(5)}  ]  queueList = [  {"primaryOrder"},  {"secondaryOrder"},  {"tertiaryOrder"}  ]  queueList.extend(totalOrder)  print(queueList)  selection1 = input("Would you like to remove an order? y/n")  if selection1 == "n":  queueList()  if selection1 == "y":  print("Alrightie")  selection2 = input("Select order to remove")  if selection2 == "a":  totalOrder.pop(a)  print(queueList)  elif selection2 =="Latest":  totalOrder.pop(-1)  print(queueList)  queueList() | A for loop is used to print all the lines of orders from the customer’s orderList.  After the ordered items are printed, the chef is presented with an option of clearing the queueList (yes or no).  Note that “primaryOrder” is also an order, just like totalOrder, except from another customer.  I used queueList.extend instead of queueList.append because it is more flexible with adding dictionaries. Unlike extend, queueList.append is more specific to adding one item rather than a whole dictionary.  This code runs without error. |

# **Development – Algorithms**

## Annotated Pseudocode to Python table for calcBill()

|  |  |  |  |
| --- | --- | --- | --- |
| Pseudocode Adjustments | Coding Construct | Python Code | Code Annotations |
| Initial pseudocode for calcBill():  BEGIN calcBill()          CALL orderList          IF clearQueue.selected=TRUE                 SET calcBill=sum(orderList.itemList.price)          ENDIF  END  What changed:  The initial pseudocode is spaghetti code, because it relies on the other algorithms, meaning debugging will be difficult. To resolve this, a new pseudocode is generated:  BEGIN calcBill()          CALL orderList          FOR item IN orderlist          SET price = orderItem.price          SET total = total + price  END | FOR is an iteration of events, used to calculate the price of an order in this case.  SET is used for assignments, where a variable is assigned a given value. | def calcBill():  totalOrder = [  {"type":"dessert","name":"cake","price":int(7)},  {"type":"burger","name":"beef","price":int(15)},  {"type":"drink","name":"lemonade","price":int(5)}  ]  total=0  for row in totalOrder:  price = row["price"]  total = total + price  print(total)  calcBill() | A new dictionary called totalOrder is created, as the algorithms aren’t linked yet.  A recursive for loop is used to calculate the total price of the order  This code runs without error. |

# **Evaluation of Solution and Prototype**

|  |  |  |
| --- | --- | --- |
| Criterion | Achieved | Justification |
| Prescribed |  |  |
| Take order | ✓ | In regards to the algorithm code, orders can be taken via selecting the meal type, followed by selecting the meal name. (Reference to orderList.py) This meets the functional requirement which must be completely (highest priority).  The interface developed using Kivy code is able to view the menu and add orders accordingly. On the menu page, meals can be added to order list; on the order list page, the list can be updated (sent to queue). However, due to limited kivy skills, I couldn’t get the remove order button to sit next to every meal item in the order list. This might mean only the whole order list can be removed, unlike what I previously thought. However, that is acceptable, because the prototype is useable and easier to learn for new customers. |
| Send order to queue | ✓ | There is a function called queueList.py which adds new orders into its list of orders. However, because this app is only at its prototype stage, the functions are not linked. Therefore, the orders taken from orderList.py are not the items added to the queue list. What’s being added to the list in queueList is a dummy order, which can be easily replaced with real orders once the application is finalised. Sending orders to queue is another high-priority functional requirement which has been met. |
| Remove from queue when checked | ✓ | Orders can be selectively removed from queue by entering the order number in the list. (Reference to queueList.py) However, there is no interface developed for this, because it is not something the user (customer) will need to look at. This is a relatively low-priority functional requirement, however it also must be completed according to the priority mind map. Consequently, this requirement is also met. |
| Self-determined |  |  |
| Calculate bill | ✓ | Through a recursive for loop in calcBill.py, the prices in each row (order item in list) are added together to calculate the bill. Once again, it is working on a dummy order, which can be easily replaced with real orders once the application is finalised. Currently, there is no interface developed for this. However, in future iterations, this could be added as another feature in the order list interface. |
| Welcome page and success page after order is updated | ✓/ X | Although these two interfaces were initially planned to exist, they were not high on the order of priority. They would be classified under the “should be completed”, but not a “must”, because it is not essential to the app. The user is still able to place orders, remove orders, and pay the right bill without the two interfaces. This is why the original two interfaces were removed from the documentation, but still annotated in the UI Sketches. The Kivy interfaces for them are available in the A4 code document. |
| Delete order after being checked | X | Since this solution is used in a restaurant setting, there will be many cycles for each table. In other words, once a customer has paid and left, another can come in and sit at the same table. However, once again because the algorithms haven’t been linked yet, it is impossible to determine when the order should be deleted. This criteria is probably one of the least important for the moment, because everything else needs to be working before there is an order to delete. In future iterations, this could be considered. |

# Code for the two removed interfaces:

## Welcome Page:

Diagram

Description automatically generated<MyGrid>

menu: customerMenu

orders: orderList

GridLayout:

cols: 1

size: root.width - 100, root.height - 100

pos: 50,50

GridLayout:

Label:

text: "Image"

Graphical user interface

Description automatically generated pos: 50, 400

background\_color: 0,0.5,1,0.5

canvas.before:

Color:

rgba: self.background\_color

Rectangle:

size: root.width - 100, root.height - 50

pos: 50, 400

cols: 1

GridLayout:

Label:

text: "Welcome to our restaurant!"

pos: 100, 325

cols: 1

GridLayout:

background\_color: 0,0.5,1,0.5

size: 100,50

pos: 50,200

cols: 1

Button:

text: "Menu"

id: customerMenu

multiline: False

on\_press: root.pressed()

size: 150,60

pos: 50,200

size\_hint: None, None

Button:

text: "Order List"

id: orderList

multiline: False

on\_press: root.pressed()

size: 150,60

pos: 50,200

size\_hint: None, None

GridLayout:

Label:

text: "Image"

pos: 250, 150

background\_color: 0,0.5,1,0.5

canvas.before:

Color:

rgba: self.background\_color

Rectangle:

size: root.width /3, root.height /3

pos: 250,150

cols: 1

GridLayout:

Label:

text: "Help"

pos: 550, 175

background\_color: 0,0.5,1,0.5

canvas.before:

Color:

rgba: self.background\_color

Rectangle:

size: root.width /4, root.height /4

pos: 550,175

cols: 1

Annotations:

The Kivy output matches pretty closely to the UI sketch, which is really good because I was worrying about the buttons not being in the right place.

I ensured the buttons to sit in the middle-leftregion to match the UI sketch by putting the buttons’ grid layout before the images’ grid layout. That way, it wouldn’t sit below and right at the bottom of the interface.

Another thing I noticed regarding to the buttons is that without adding “size\_hint: None, None”, it spans across the whole screen weirdly. That is why you’ll see size\_hint in almost all my buttons.

## Success message:

<MyGrid>

Shape, polygon

Description automatically generated take: takeAway

dine: dineIn

orders: orderList

GridLayout:

cols: 1

size: root.width - 100, root.height - 100

pos: 50,50

GridLayout:

Label:

text: "Image"

pos: 400,300

background\_color: 0,0.5,1,0.5

canvas.before:

Color:

rgba: self.background\_color

Rectangle:

size: 300,200

pos: 400, 300

cols: 1

GridLayout:

Label:

text: "Success!\n\nYour order has been updated!\n\nSelect one of the following options:"

pos: 150, 375

cols: 1

GridLayout:

background\_color: 0,0.5,1,0.5

size: 100,50

**Graphical user interface

Description automatically generated** pos: 50,250

cols: 3

Button:

text: "Take away"

id: takeAway

multiline: False

on\_press: root.pressed()

size: 100,50

pos: 50,250

Button:

text: "Dine in"

id: dineIn

multiline: False

on\_press: root.pressed()

size: 100,50

pos: 50,250

Button:

text: "Order list"

id: orderList

multiline: False

on\_press: root.pressed()

size: 100,50

pos: 50,250

**Annotations:**

In this interface, order does not matter with the image’s grid layout and success message’s grid layout, because they don’t depend on each other’s position. The buttons below that, on the other hand, need to be coded last because otherwise, it would appear at above the text and image.