CSC4001: Software Engineering

Deisgn Document for WeChat Miniprogram: Habiter — One for All

Jiarui Li
118020229

Ran Zhuo
118010475

Yuncong Cui
118010045

Zhixuan Liu
118010202

2020/2021 Semester 1

Table of Contents

Introduction

1.1 Background

Good habits and healthy lifestyle are crucial to everyone in the modern society. However, habit formation is not an easy task, which is an important part of behavior change interventions: to ensure an intervention has long-term effects, the new behavior has to turn into a habit and become automatic. This is sometimes difficult to achieve even for a self-disciplined person on his own, let alone the vast majority of people. Therefore, how to help people form and maintain habits has become a popular social topic [5].

Stawarz et al. [8] express the performance of smartphone applications on habit formation with positive effects but still some limitations, one of which is that a lot of people cannot continue to use those software applications because they are likely to lose their interests in the simple "check and remind" based platforms.

1.2 Design Idea of "Habiter"

1.2.1 Primary Idea

Our primary goal is to design a lite, user-friendly and private software to help users to develop good habits in certain directions. A qualified standard and expectation is to make our application have its own advantages to attract the users for a long term and exercise as an effective reminder with proper and humanized suggestions.

WeChat and QQ have brought the trend of online chatting. DouYin and KuaiShou have brought the trend of short videos. The ultimate goal of "Habiter" is the trend of calling for young people to cultivate good habits and have a healthy lifestyle steps by steps, basically from the first function, bookkeeping, in the system.

1.2.2 Flaws of Previous Software

According to the investigation to the current software, it is obvious to declare that the public needs a lite and convenient software to manage the cultivation of habits. All applications about payment has the functions of bookkeeping; however, they only record the transactions that use their system, which means user can hardly record all transactions by only one application. Moreover, most payment software put profit in the first place, so that many redundant functions and advertisements are included, which makes users find it difficult to stick on to use the application.

Figure 1-1: User Survey I

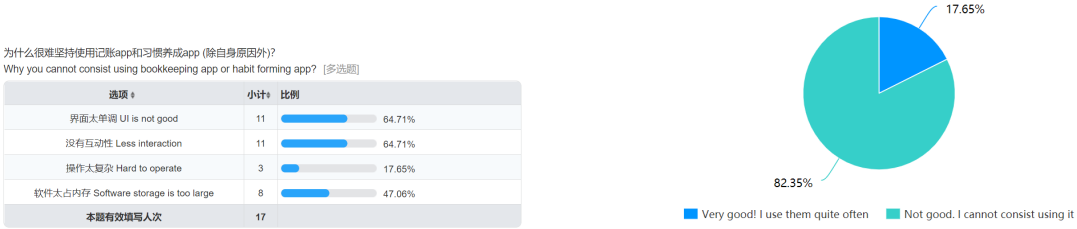

Figure 1-2: User Survey II

1.2.3 Improvement and Integration of Functions

The application our group design is called “Habiter”, which means a manager who help user to develop good habits. Based on improving the shorting comings of previous smartphone applications, our goal is to build an interesting and user-interacted software application, which provides multiple healthy lifestyle formation services including good habit check-in reminders, cost accounting statistic, daily life recommendation and so on. The highlight of our system is that “Habiter” is a chatrobot who not only does the cost analysis, recommendation, and reminding jobs, but also can perform daily chatting. This design solves the problems of previous products by greatly enhancing the user experience and the interest of the product.

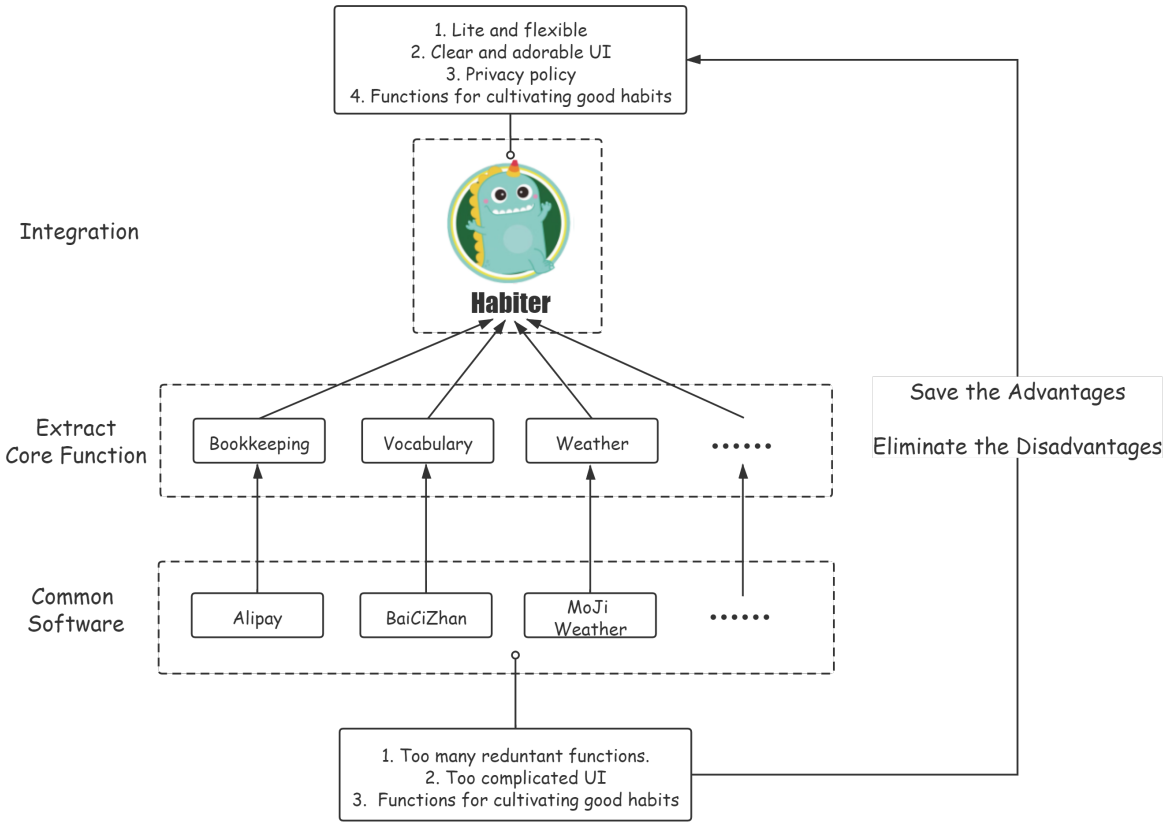

Figure 1-3: Design Idea

The application starts from the function of bookkeeping and plans to develop other functions about habits successively. Our team extract the core

idea of the main function in each software regarding to good habits (Figure 1-3). Each idea is implemented and encapsulated as independent module and connect them to the universal API in our application. The improvement and integration of functions, which interacts with users by the chatbot, is the dominated design.

Function Descriptions and Advantages

2.1 General Edges

2.1.1 Lightweight Structure

"Habiter" is a software about life-style, which has light and clean interface, simple operation, powerful functions, no advertisements, no software bundled. Our design of "Habiter" aims to provide an integration of life management software, focusing on helping good habit formations of users. It allows users to access easily to daily lite functions for good habits cultivation: bookkeeping, vocabulary memorizing, weather, daily recommendation and chatbot.

2.1.2 Utilize the advantages of WeChat MiniPrograms

2.1.2.1 Low Development Cost

"Habiter" is based on the framework of the WeChat Miniprogram. A WeChat Miniprogram is equivalent to a lightweight App, but it costs much less than developing a smartphone application. "Habiter" now only needs small several thousand yuan a year to rant the cloud server. Smartphone applications, on the other hand, require a lot of financial resources and take a long time.

2.1.2.2 A Larger Amount of Potential Users

WeChat Miniprogram is a big platform attached to WeChat billion traffic with quite a lot of access. Users can use the QR code, friend invitation, WeChat group forward, searching, WeChat chat box drop-down menu and many other

ways to enter the it. Many entrances brought by the user fission improve the advantages and attraction of “Habiter”.

2.1.2.3 Convenient and Flexible

WeChat Miniprogram is different from smartphone application which needs to be downloaded from software store. “Habiter” does not consume much memory space, which is very convenient and does not waste the time of users.

2.1.3 User-Friendly Interface

2.1.3.1 Clear Layout

All the visual elements of the interface in “Habiter” are clear and simple. Users can easily find the way to use different functions and do not have to deliberately think about the meaning of each element.

2.1.3.2 The Consistency

Keep the style of the entire interface consistent so that users can get used to the mode of operation. Although the “Habiter” combines modules with different functions, our team unify the styles of each pages to give users a comfortable using habit.

2.1.3.3 Anti-Dumbness Mechanism

A good user interface prevents users from accidentally making mistakes. All modules in “Habiter” have set certain safety cautions to avoid the misoperation from users.

2.1.3.4 High Efficiency

Good interfaces allow users to achieve desired "outputs" with minimal "inputs" and allow experienced users to operate more efficiently. Our team tries best to change as much input boxes as we can to be the selected boxes to reduce the work of users. Once the information has been recorded, the users can choose different modes to view the results without typing for multiple times.

2.1.4 Privacy Protection

Existing software acquires the access of users' personal information when users need to use their functions, which can lead to serious risks of information leak (Figure 2-1). "Habiter" can solve this by its own privacy protection strategy. Each user will be attributed a random and private ID to access data to the system. Personal information is not necessary for the using of functions.

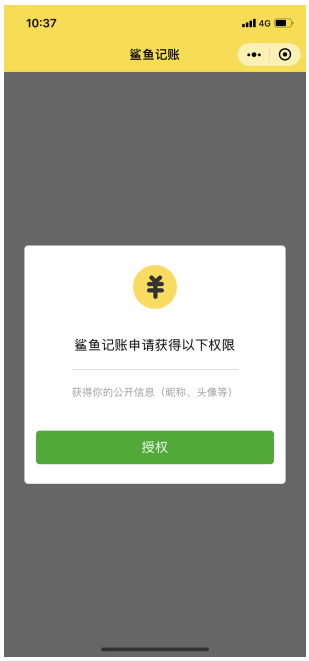

Figure 2-1: The Risk of Information Leak

2.2 Bookkeeping and Habit Cultivation

Bookkeeping is the main function module (Figure 2-2). Bookkeeping is a valuable and meaningful habit. With bookkeeping, user can understand financial situation, make sound financial planning, and maximize the efficient use of wealth. Moreover, user can clearly see their life by keeping an account.

Ultimate purpose of bookkeeping is to solve the financial problems of life, to help users achieve a variety of financial goals. Every process of our life can not be separated from money. When we are not clear about the monetary value of our corresponding life goals and our current financial situation, it is easy to cause money anxiety, which is a kind of instinctive anxiety for unknown things. Bookkeeping and financial planning can help us alleviate this situation very well. By planning and sorting out financial situation, users can see your income and expenditure flow clearly. Through scientific planning and adjustment, users can develop rational consumption habits.

Figure 2-2: Bookkeeping Module

2.3 Vocabulary

In the field of online education, the more lightweight and fragmented products are, the more they adapt to the habits of users on mobiles. On the contrary, the more complex and systematic educational products are, the more difficult they can be carried on mobiles. From this point of view, language learning products have a strong mobile nature because of their low knowledge difficulty, independence between knowledge and easier implementation of structuring. In such an environment, all kinds of words memory software emerge in endlessly, such as BaiCiZhan, etc. Therefore, we also add the vocabulary module in the "Habiter" in order to help the user to utilize the fragmentary time to memory words.

Figure 2-3: Vocabulary Module

However, most available words software are full of advertisements and redundant functions. Because of this, users have to waste time on trying different functions and their attraction can be easily dispersed. As a result, the learning efficiency can be greatly reduced due to irrelevant information in the interface. Thus, our team try to build the interaction interface as simple as we

can. The only information the user need to focus is the word and its meaning. All relevant information the software needs to deal with are hidden to give user a brief and clean environment (Figure 2-3).

2.4 Weather

With the rapid development of mobile Internet and the popularity of smartphones, the way for people to obtain weather information becomes more and more convenient and accurate. At first, users can only get general weather information in the next 24 or 48 hours through TV stations or newspapers. Now, users can get specific and accurate real-time weather through weather software, get the weather information for the next week or even two weeks, and facilitate the arrangement of the following travel, work and so on. Therefore, checking the weather everyday is a good habit to help people prepare for a new day.

2.5 Daily Recommendation

“Habiter” also provides users with recommendations for outfits and food which in a highly interacted method. This crabs users interest to continue using this app. According to the information obtained from our market research, “Habiter” is superior to most existing software in terms of user experience.

Moreover, suggestion of “Habiter” on outfit is related to weather condition because the different modules in “Habiter” can share the database, which few products on the market currently provide. Food recommendation of “Habiter” will be based on nutritious and healthy food. The recommendation of food will combined with habit cultivation part to improve healthy live style formation.

2.6 Chatbot

"Habiter" has a disruptive innovation in interaction. The entry point of "Habiter" is completely different from traditional products. It turns the reminding into chatting with chat-bot, which enhances users' willingness to keep their routine, so as to achieve the purpose of forming good habits.

Academically speaking, when we have knowledge or data, we can build a answering system based on it. "Habiter" does not need data from previous project of study. In our model, the data can be collected by the "Habiter" itself. With the database that records the bill of users, the "Habiter" can finish knowledge graph modeling and query without external data. On the other hand, the daily conversation is flexible and decided by favor of individuals in a large extend. The bill information can perfectly describe the life and customs of an individual.

It is known that it is always easier for human to carry on together than alone. "Habiter" can be a friend who goes along with you whenever you need, and an assistant who helps you to better manage your daily life and get rid of bad habits like spending money without a plan. In the process of punching the clock everyday, users gradually get used to regulating themselves and finally develop a sense of self-discipline. This is to say, "Habiter" works as a virtual companion and supervises users to persevere in forming good habits. As a result, users will become more self-disciplined and have a better control over their life.

Software Structure

3.1 Unified Modeling Language (UML) Diagram

3.1.1 User Case Diagram

A use case diagram shows a set of user cases, actors, and the relationships between them. Figure 3-1 is the user case diagram of “Habiter” in UML. Users can use all function modules, while administrator has the authority of database. The database is not local, but on the cloud server of WeChat Miniprogram.

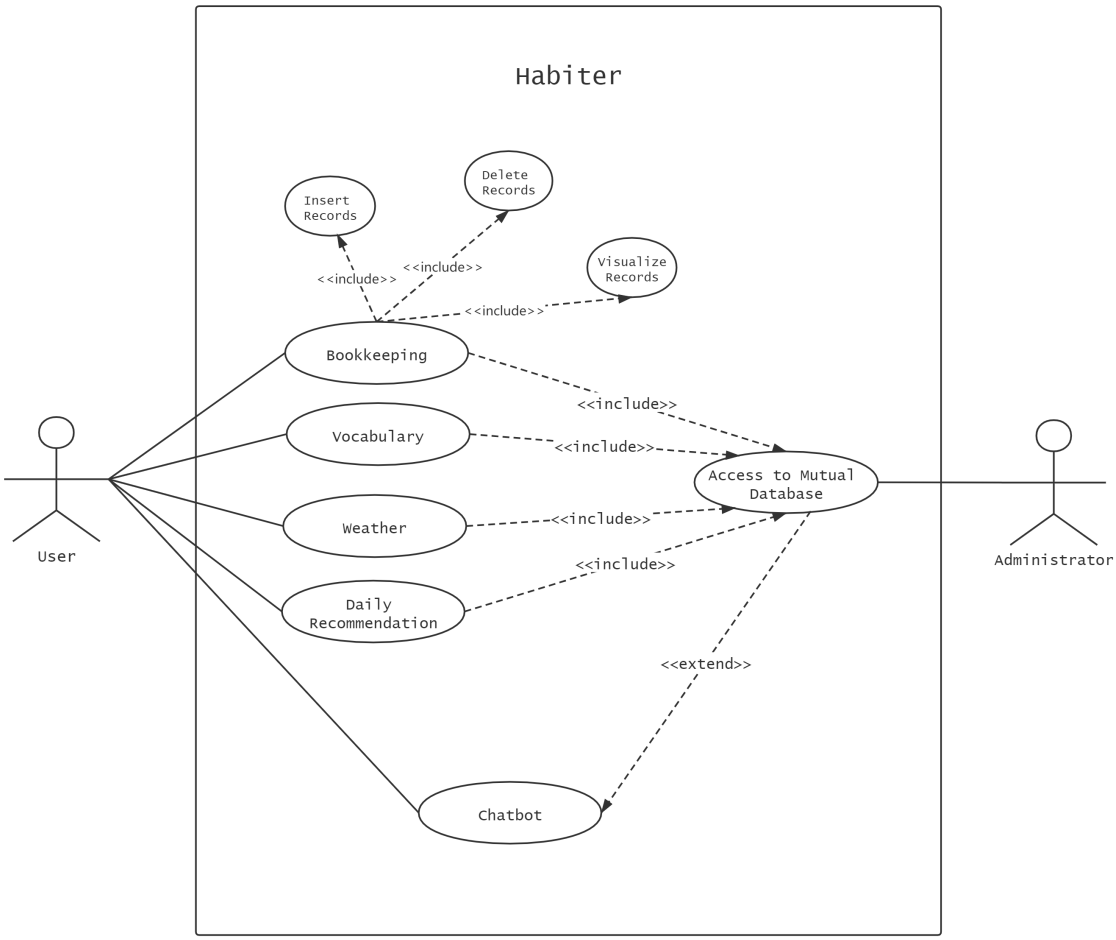

Figure 3-1: User Case Diagram of “Habiter” in UML

3.1.2 Class Diagram

Class diagrams show the classes of the system, their interrelationships (including inheritance, aggregation, and association), and the operations and attributes of the classes. Figure 3-2 is the class diagram of “Habiter” in UML. Function modules reserve certain data for the user; therefor, all function classes rely on the User class. Some function classes with complicated operations has subclasses.

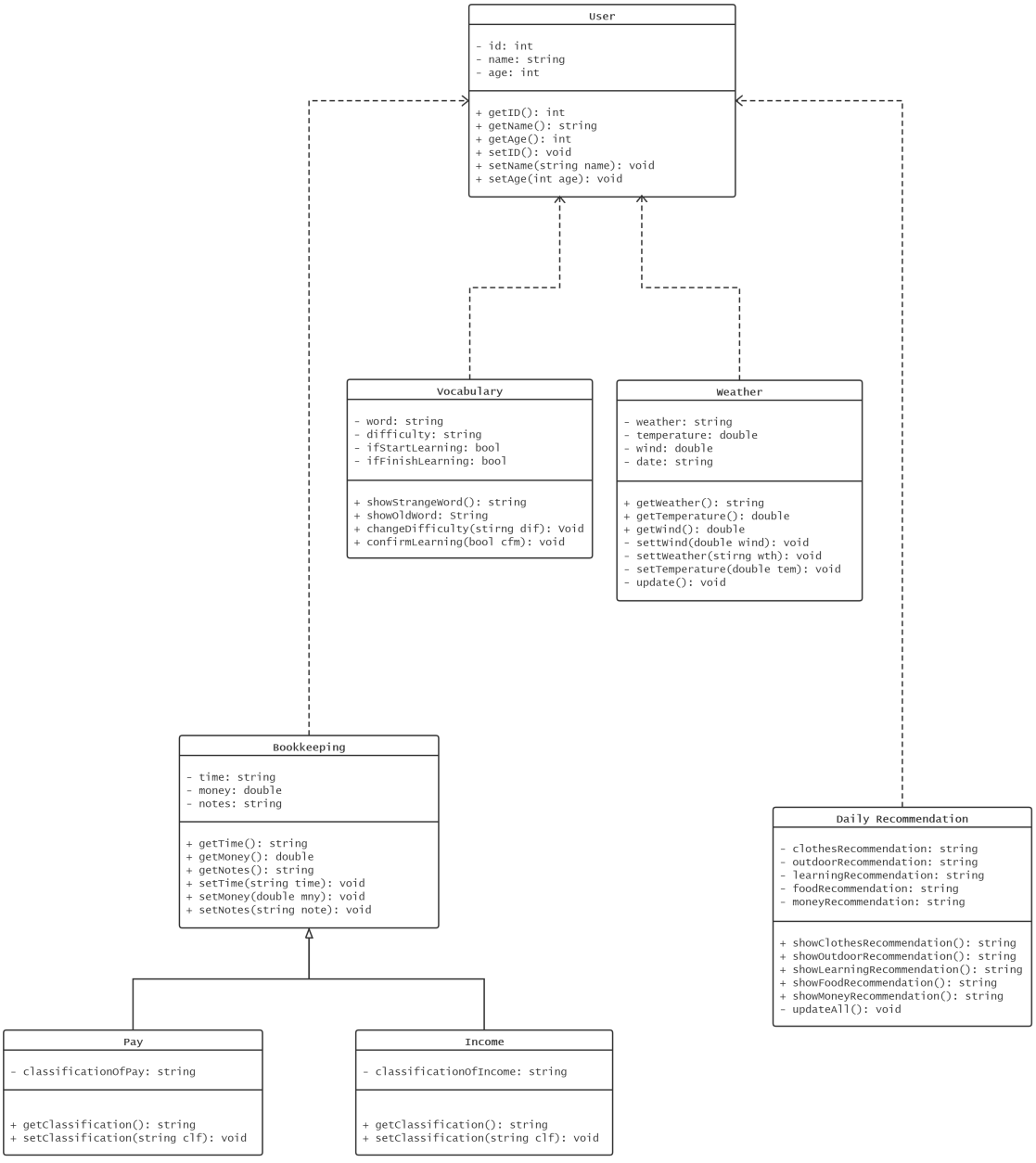

Figure 3-2: Class Diagram of “Habiter” in UML

3.1.3 Sequence Diagram

A sequence diagram is an interaction diagram that details how operations are carried out. Figure 3-3 is the sequence diagram of “Habiter” in UML. The diagram consists of five options to demonstrate function modules.

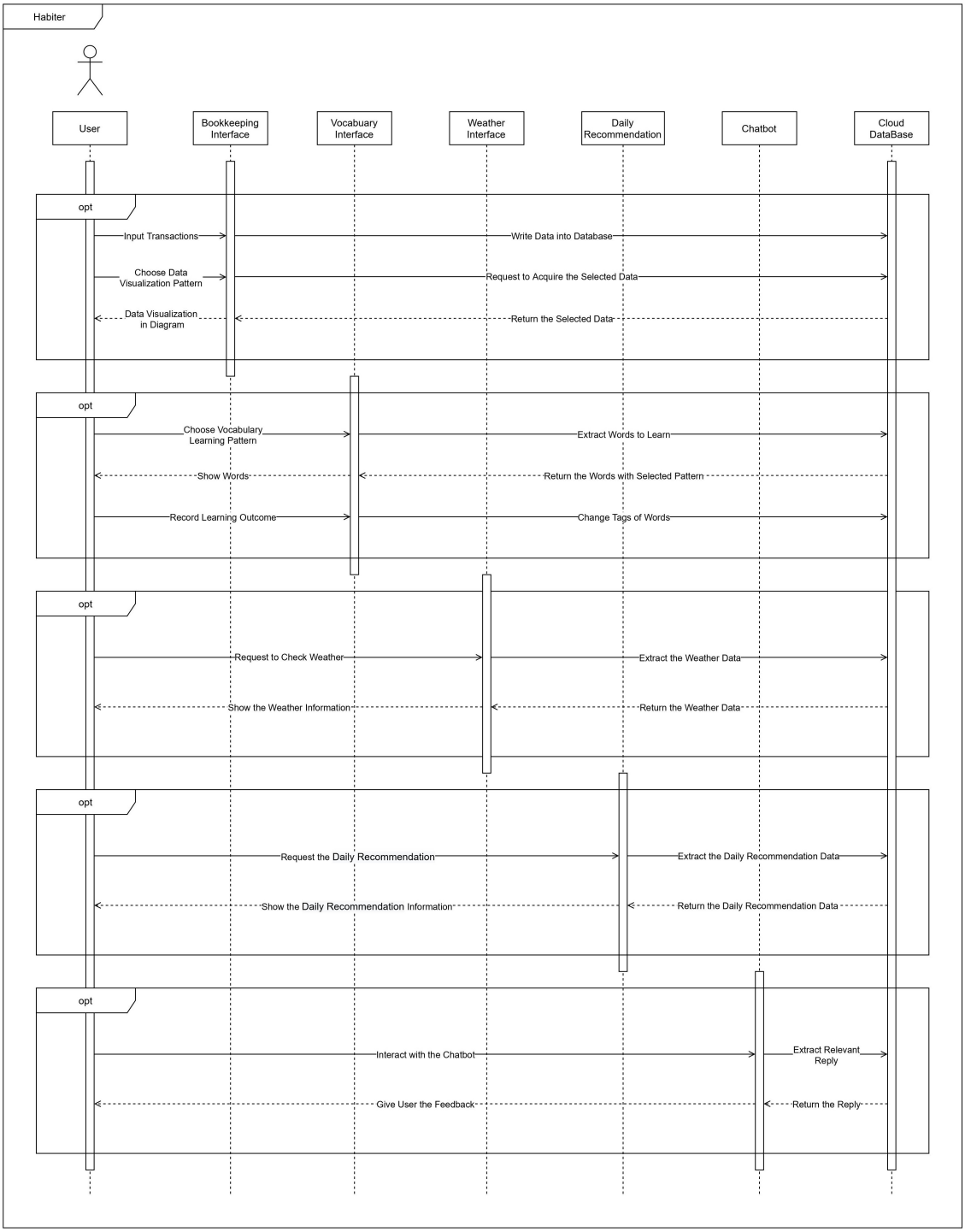

Figure 3-3: Sequence Diagram of “Habiter” in UML

3.1.4 Activity Diagram

An Activity diagram describes the workflow behaviour of a system. It focuses on condition of flow and the sequence in which it happens. We describes respective workflow in each module by the swimlane using an activity diagram. Figure 3-4 is the activity diagram of "Habiter" in UML.

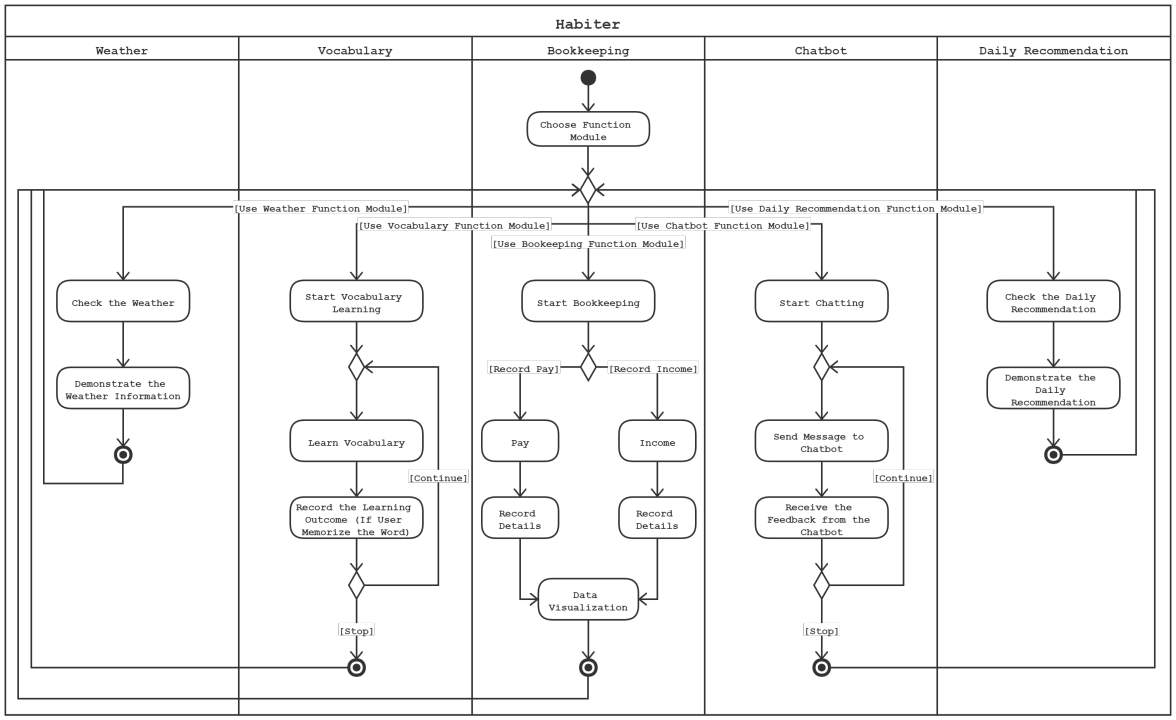

Figure 3-4: Activity Diagram of "Habiter" in UML

3.1.5 State Machine Diagram

A state machine diagram shows the possible states of the object and the transitions that cause a change in state. An object responds differently to the same event depending on what state it is in. Figure 3-5 is the state machine diagram of "Habiter" in UML.

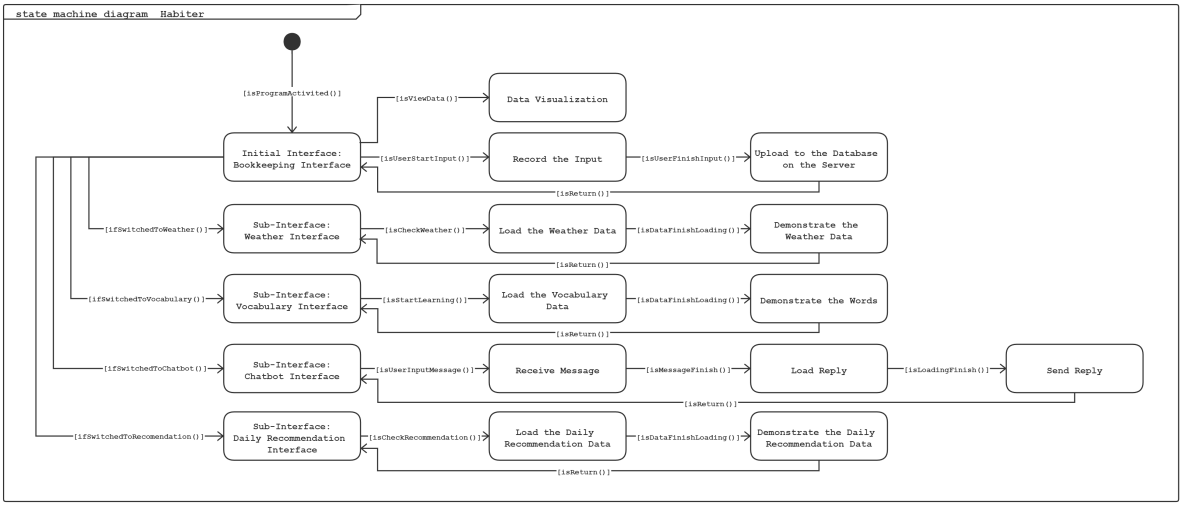

Figure 3-5: State Machine Diagram of “Habiter” in UML

Software Architecture

4.1 Architecture Description

Software architectures are critical artifacts in bridging the gap between the initial concept of a system and the system's implementation. Architecture should also achieve quality attributes that must be considered throughout design, implementation, and deployment.

Architectural patterns mainly target the non-functional requirements of the product, and provide an overall structure for the target system in order to address these requirements. The structure of "Habiter" is based on the traditional architecture patterns; however, it is not suitable to classify it as one single pattern due to its base of WeChat Miniprogram and the integration of different functions. Therefore, we use the "Microservice Architecture" to design the structure of "Habiter" around its business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data. The microservice architectural style is similar to the traditional "Distributed Component Architecture" because different function module has distributed systems to implement. The implementation of database on the cloud server is also a kind of distributed database. Moreover, "Habiter" has its own database on the cloud server, which leads to result that the data transfer module still need to use "Client-Server Architecture" (Figure 4-1).

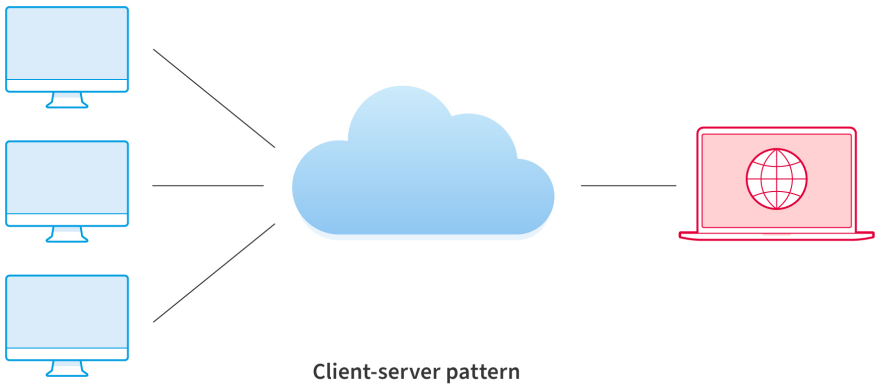

Figure 4-1: Client-Server Architecture

Based on the “Distributed Component Architecture” and “Client-Server Architecture”, “Microservice Architecture” is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms (HTTP resource API). These services are built around business capabilities and independently deployable by fully automated deployment machinery (Figure 4-2). Client can send a sequence of requests to ask the service on the Distributed components to respond. There is a bare minimum of centralized management of these services, which may use different data storage technologies.

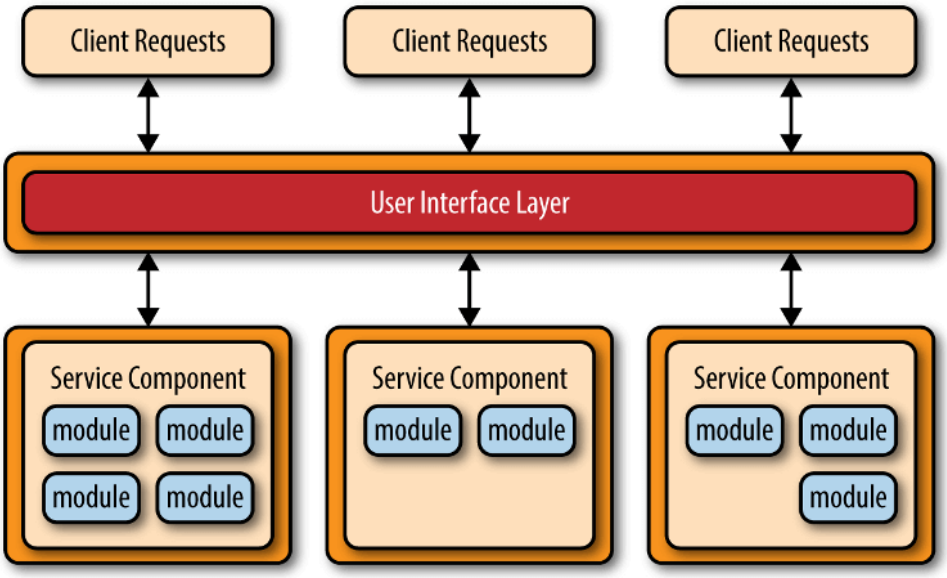

Figure 4-2: Microservice Architecture

4.2 General Suitability of Architecture Pattern

The primary ides of “Habiter” is to build a lightweight software, which is corresponding to the characteristic of the “Microservice Architecture”. The independent function module is implemented as an independent service in the architecture. The independence makes software easy to maintain. The general suitability of Microservice Architecture in “Habiter” can be concluded as four points:

- **Good scalability**: Low coupling between various services.
- **Easy to deploy:** “Habiter” can be broken down from a single deployable unit into multiple services, each of which is also a deployable unit.
- **Easy to develop:** Each component can be developed in a continuous integration style, deployed in real time, and continuously upgraded.
- **Easy to test:** Each service can be tested individually

Business stakeholders can understand the structure of “Habiter” easily, which also makes “Habiter” a good software architecture that can be usable over the long-term because user can easily figure out presented functions and explore new modules. Such architecture pattern is flexible, adaptable, and extensible. Our team can easily add features. Moreover, the system performance does not diminish due to this.

4.3 Availability of Architecture

Availability is a measure of how easy it is to use a product to perform a given task. It is concerned with system failure and its associated consequences. A system failure occurs when the system no longer delivers a service consistent with its specification.

"Habiter" collects the ideas of hot programs, such as Alipay and YouDao Dictionary. Such programs offer a variety of powerful capabilities, but require users to learn and remember dozens of obscure buttons to perform them. It can be said that such applications are of high practicability (they provide the user with the necessary functionality), but of low availability (the user must spend a lot of time and effort to learn and use them). In contrast, "Habiter" is a properly designed and simple application, which is easy to use with high availability. However, "Habiter" mainly focus on the most important and common functions, so that the high availability are still preserved.

Our team provides four main mechanisms to ensure the high availability in the development:

- **User-centric:** We focus on understanding users' needs in the design process. All functions in "Habiter" are designed to help the user to form good habits, which is a great need to most young people.
- **Integrated design:** All aspects of design are developed in parallel, not sequentially. Make the internal design of the product consistent with the needs of the user interface.
- **Test early and consistently:** By introducing availability testing throughout the development process of "Habiter", users have the opportunity to provide feedback on the design.
- **Repetitive design:** We keep modifying the design repeatedly throughout the testing process.

We have done the survey between 173 students from CUHK(SZ) to test its practical usage. Nearly 80% of them agree with the functional practicability. The result gives out a positive feedback about the high avalibility in "Habiter" (Figure 4-3).

You think Habiter is an interesting product and can help you form good habits:
你认为Habiter很有趣并且可以帮你形成好习惯 [单选题]

选项 ⇕	小计⇕	比例
Strong Disagree	0	0%
Disagree	19	10.98%
Neutral	19	10.98%
Agree	64	36.99%
Strongly Agree	68	39.31%
Others	3	1.73%
本题有效填写人次	173	

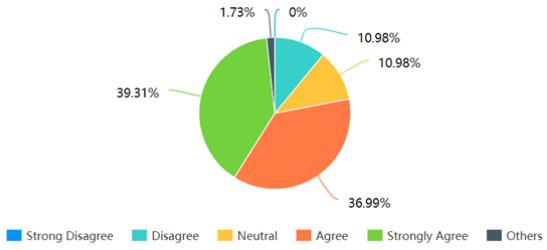

Figure 4-3: Availability Survey

4.4 Maintainability of Architecture

Software maintenance is the process of modifying software to correct errors or to meet new requirements after it has been delivered to service. We refer the book *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code* [9] to make rules for the "Habiter":

- **Write short code units.** "Habiter" requires that each function should be no longer than 15 lines long. Smaller methods are easier to understand and unit test. More than 15 lines means the method can be split.
- **Write simple code units.** "Habiter" limits the number of branch points per code unit to no more than 4. Simple code is easier to modify and test. Too many branching points means more test cases.
- **No duplicate code.** "Habiter" forbids the duplicate code, which is defined as a piece of code that has at least 6 lines in common. The change of duplicate code means a lot of rework and error prone.
- **Separate concerns between modules.** Classes in "Habiter" are required to be not too large or complex. The small size creates loose coupling between classes, which means that classes are more flexible to adapt to future changes.

- **Architecture components are loosely coupled.** "Habiter" uses the abstract factory design pattern, and returns value via methods of the factory class. This generic factory interface hides the process of creating specific products.

- **Automate development deployment and testing.** Tests include unit tests, integration tests, end-to-end tests, regression tests, and acceptance tests. Automated tests are repeatable and efficient. Asserts in automated tests can act as comment.

4.5 Extensibility of Architecture

Product extensibility describes how easy it is to expand a product's feature set. "Habiter" has been designed from its earliest stages for customization and enhancement. Extensible products are designed for ease in expanding your installation's feature set, enriching current features, and integrating with third-party software.

Maximizing extensibility has been our goal through all aspects of "Habiter" development. Core tasks such as bookkeeping are packaged as discrete modules. Except for the basic functions, user can expand features by installing modules in our next step of the development of this software. We have well-designed API to make sure user can easily add or delete subsequent functions by simply adding or deleting modules.

4.6 Understandability of Architecture

Software understandability plays a requisite role in measuring the reliability of the software, which means that without understanding the design of the system, the system structure will become progressively worse when

system modification happens, so the software understandability needs to be managed [1].

Our team try to decompose the system of “Habiter” into substructures by function modules, then measure the coupling and cohesion between substructures, where the cohesion is the strength of the functional relatedness within the same substructure and the coupling is the functional relatedness between other substructures [2].

Coupling	Bookkeeping	Weather	Vocabulary	Daily Recommendation	Chatbot
Bookkeeping	N/A	None	None	None	Medium
Weather	None	N/A	None	None	Medium
Vocaulary	None	None	N/A	Low	Medium
Daily Recommendation	None	None	Low	N/A	Medium
Chatbot	Medium	Medium	Medium	Medium	N/A

Table 4-1: The Coupling between Substructures

From above Table 4-1, the general coupling between modules is relative low due to the modules has been divide separately in “Habiter”. The structure of the code is easy for understanding. The software understandability is quite prominent.

4.7 Re-usability of Architecture

To systematically support the process for the ever growing complexity of software, higher levels of abstraction are needed. Kruchten [4] noted that "for a software reuse technique to be effective, it must reduce the cognitive distance between the initialconcept of a system and its final executable implementation".

Our team adopt the method of building two scenarios to measure the re-usability of architecture [6]. The scenarios analyse the objective result of architecture in "Habiter" in the hypothetical environment.

Scenario 1: The system of "Habiter" will be delivered with basic capabilities. New features for complex call processing will be incrementally introduced.

Architecture Impact: The "Habiter" is designed for connected more and more external function modules. The architecture supports incremental development because of the separation of concerns, decoupling of functionality. Re-usability is a special edge from the view of our architecture. However, the API is still need to specifically design for each module. The ideal model will have the automatic connection mechanism.

Scenario 2: A third party develops a new feature to interwork with the architecture of "Habiter".

Architecture Impact: If we want to a well-developed system to connected with feature from the third party automatically, proxies are needed to build to communicate with third party applications. Next step of our project is add the proxies in the UI to optimize this design. More explicit information on new features need to be identified for further analysis.

For an application, just a few scenarios were initially developed collaboratively with the architect. However, the weak coupling in the module and well API design of "Habiter" can still be useful in the future situation.

4.8 Performance of Architecture

Performance assessment of a software architecture determines whether the architecture will support the system's scalability and responsiveness requirements. For new development, it uncovers potential problems early in the development process when they are easier and less expensive to fix.

To value the performance of architecture, we adopt the PASA (Performance Assessment of Software Architectures) [10], a rigorous and systematic method for the performance assessment of software architectures, that consists of the following steps:

- **Identification of Critical Use Cases:** The externally visible behaviors of the software that are important to responsiveness or scalability are identified.
- **Selection of Key Performance Scenarios:** For each critical use case, the scenarios that are important to performance are identified.
- **Identification of Performance Objectives:** Precise, quantitative, measurable performance objectives are identified for each key scenario.
- **Architecture Clarification and Discussion:** Our team conducts a more detailed discussion of the architecture and the specific features that support the key performance scenarios.
- **Architectural Analysis:** The architecture is analyzed to determine whether it will support the performance objectives.
- **Identification of Alternatives:** If a problem is found, alternatives for meeting performance objectives are identified.
- **Economic Analysis:** The costs and benefits of the study and the resulting improvements.

Software Development

5.1 Agile Software Development

Industry practitioners and researchers emphasize rapid and flexible development. Software architecture, as a discipline, deals with modeling and managing a software system's structure, project blueprint, and communications among stakeholders, which are essential for achieving quality attributes such as usability and maintainability.

We adopt the agile software development in "Habiter", a set of frameworks and practices based on the values and principles expressed in the Manifesto for Agile Software Development and the 12 Principles [7] behind it. When a team approach software development in a particular manner, it's generally good to live by these values and principles and use them to help figure out the right things to do given their particular context.

5.2 Implementation

In this section, the implementation of our Habiter is described in details. It is important to emphasis that our implementation constantly keeps in track of our architecture design, the "Microservice Architecture". In general, the relationship and interaction between the cloud base and functional components are carefully designed to ensure the loose coupling and scalable microservice architecture. The rest of this section is composed by four parts. The technical stacks utilized is firstly introduced. The second and third parts are

mainly discussing about the work performed on the cloud base and functional components, which are followed by the last part, other modules.

5.2.1 Technical Stacks

Figure 5-1: wxml

Figure 5-2: wxss

Figure 5-3: js

Figure 5-4: json

Figure 5-5: cloud base

In the development, we mainly used the above technical stacks, wxml, wxss, javascripts (js), json and cloud base. The former four stacks (wxml, wxss, js and json) are familiar concepts. The wxml and wxss are just another name for the traditional html and css context. The underlying usage and syntax of them are basically the same. The json and js files are used for data transmission and dynamic feature and interaction feature of the miniprogram. The cloud base is an important module of our project. It act as a communication channel, resources pool and shared environment etc. In order for Habiter to run successfully the cloud base has to be setup carefully. The integrated development environment used is the *Wechat Devtools*, which integrated all above stacks, a simulated running window, test module, special running environment setting etc.

5.2.2 Cloud Base Setup

Before go into the construction of each functional components, the setup of the cloud base is particularly important. Firstly, our database is implemented on the cloud.

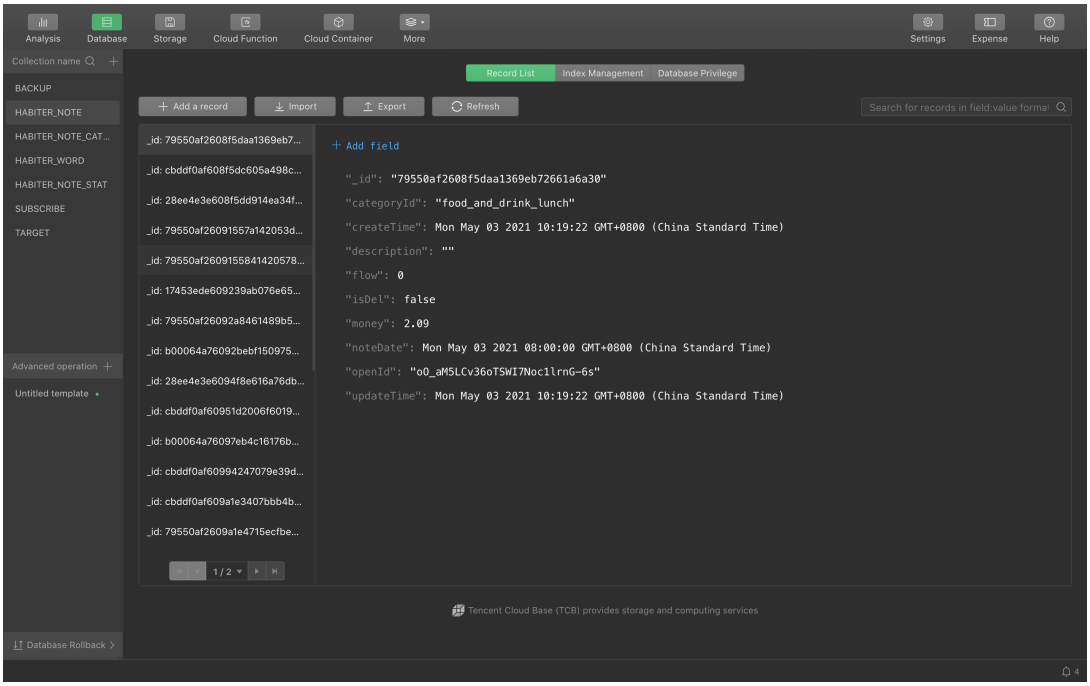

Figure 5-6: Cloud Database

The cloud base database, different from other commonly used relational database system such as MySQL, MariaDB etc, is a flexible database extremely suitable for our project architecture design. There is no key, no fix size limitation on each table, and even no constraint on the length of the entries in the same table.

The above feature of miniprogram cloud base performs as the bedstone of our ”Microservice Architecture” design, with highly extendable database. Moreover, in order to achieve the privacy goal, the local record and only open identity designs of the miniprogram is fully utilized, which helped us limited the number of database entities created and lowered the execution overhead. Some sensitive information is not sent to the cloud base, instead they are stored

locally or can be represented by the generated *openid*. Therefore, the risk of privacy leakage is minimized in our project context.

In order to support the currently designed functional components, bookkeeper, chat bot, personal center, vocabulary and weather, seven database entities are created. Included in them are BACKUP, HABITER_NOTE, HABITER_NOTE_CATEGORY, HABITER_NOTE_STAT, HABITER_WORD, SUBSCRIBE and TARGET. The BACKUP table is used to backup some important data, such as user's bookkeeping entries account name and password, periodically. HABITER_NOTE, HABITER_NOTE_CATEGORY, HABITER_NOTE_STAT entities are used to record all the information about the bookkeeping entries including the amount, time, location, description, labels etc. In the HABITER_WORD entity, the data related to the vocabulary component are kept. For example, the rate of forgetting, recited times, word list belonging to etc. The subscribe table is designed for upcoming Habiter membership. It will contain the information of membership purchasing, due date etc. The last entity, TARGET, corresponds to our target setting functionality, which allows the client to settle a saving goal to achieve. The table is mainly used to keep track with the progress of goal.

5.2.3 Functional Components

In this section the implementation of each functional units will be mainly decomposed into frontend programming and script codes. Interaction with the cloud and with some third party open platform will also be mentioned if they are used.

5.2.3.1 Bookkeeper

The first functional component to discuss is the bookkeeper module. "Habiter" allows users to keep records of their daily accounts. It can

automatically generate pie charts and trend charts according to users' book-keeping records, so as to more easily understand their own revenue and expenditure. Therefore, it will help users to manage and accumulate their property in a better way, and develop a sense in wealth management.

The main parts of this module are record inputting, entry querying and data visualization.

Figure 5-7: Entry Input

Figure 5-8: Calendar

Figure 5-9: Visualization

For the design of the frontend structured as the following. The three different pages above are actually implemented all together in one page while the Three circle buttons are floating tabs, which controls the showing and hiding of each single pages.

```
<view hidden="{{active !== 'list'}}">
  <list
    id="list"
    class="list"
    bind:switchTab="onSwitchTab"
    bind:reFetchBillList="onReFetchBillList"
```

```
    bind:editBill="onEditBill"
    bind:hideTab="onHideTab"
    tab="{{active}}"
    currentMonthData="{{currentMonthData}}"
  />
</view>
<view hidden="{{active !== 'index'}}">
  <index
    bind:reFetchBillList="onReFetchBillList"
    id="index"
    selectedCategory="{{$.selectedCategory}}"
    defaultCategoryList="{{$.defaultCategoryList}}"
    editBill="{{editBill}}"
    bind:hideTab="onHideTab"
  />
</view>
```

The conditional line of variable *hidden*, controls the showing and disappearing of the view component. Except for the basic background of these three pages, components such as the calendar, data charts, direction bars etc are all imported as page components.

```
{
  "component": true,
  "usingComponents": {
    "list": "../components/list/list",
    "index": "../components/index/index",
    "chart": "../components/chart/chart",
    "nav": "../components/nav/nav"
  }
}
```

The rendering of the frontend is defined in the wxss file. The following is an example of the gradient color background displayed in figure 5-7.

```
.banner-special {
  width: 100%;
  height: 410rpx;
  background: linear-gradient(40deg, #f7d14a, rgb(17, 196, 169));
  position: fixed;
  z-index: -1;
}
```

The backend script realized the dynamic feature and interacting function of the frontend. The transection recording function will be taken as an special example here, due to its involvement with the cloud base function.

```
<view class="btn-area">
  <button bindtap="submitForm" class="warn" wx:if="{{!loadingCreate}}">{{isEdit ?
      ’      ’ : ’      ’}}</button>
  <button class="warn" style="opacity: .5;" wx:else>{{isEdit ? ’      ’ :
      ’      ’}}   ...</button>
</view>
```

Above is the frontend tags that will detect the user movement.

```
wx.cloud.callFunction({
      name: ’account’,
      data: {
        mode: isEdit ? ’updateById’ : ’add’,
        money: transSum,
        categoryId: selectedCategory._id,
        noteDate: active_date_time,
        //
        description: note ? (showPayType && payType ? ‘${payType}-${note}‘ : note)
            : note,
        flow: active_tab,
        id: isEdit ? editBill._id : ’’,
      }
```

Above is part of the codes of function *submitForm* called by the frontend. This part of the function calls the function *account* which is previously configured in the cloud.

```
// entrance of the cloud function
exports.main = async (event) => {
  const wxContext = cloud.getWXContext();
  const {
    id, money, categoryId, noteDate, description, flow,
  } = event;
  cloud.updateConfig({
    env: wxContext.ENV === ’local’ ? ’release-wifo3’ : wxContext.ENV,
  })
  const db = cloud.database({
    env: wxContext.ENV === ’local’ ? ’release-wifo3’ : wxContext.ENV,
  });
```

The above function, which is part of the cloud function, will write the prompted inputs into the cloud. Therefore, the other modules such as data visualization and calendars, will be able to read the data and perform their jobs.

5.2.3.2 Vocabulary

The vocabulary component is completely implemented within our miniproogram. The long pressing navigation function is implemented as the following.

```
showIconName(event) {
  const { active } = event.currentTarget.dataset
  wx.vibrateShort()
  if (active === 'list') {
    wx.navigateTo({
      url: '/pages/danci/danci',
    })
  }
}
```

Figure 5-10: Vocabulary List Choosing

Figure 5-11: Vocabulary Reciting

For the above two pages, different from the *bookkeeper* component, are implemented in to pages instead of conditional hiding. The vocabulary meaning lines, which appears after tabbing 'unknown', instead are using the conditional hiding.

```
<view class = "cet">
    <view class = "content">
        <text class='word'>{{content}}</text>
        <view class="pron">
           <text class = "word-pron" bindtap='read'>/{{pron}}/</text>
        </view>
    </view>
    <text wx:if = "{{show}}" class = "word-definition">{{definition}}</text>
    <button class = "button miss" catchtap='show'>         </button>
    <button class = "button next" catchtap='next'>         </button>
</view>
```

The interaction and word displaying function is implemented with the following codes.

```
  onLoad: function () {
    var id = Math.floor(Math.random() * 499) + 1;
    var word = list.wordList[id];
    this.setData({
      content: word.content,
      pron: word.pron,
      definition: word.definition,
      audio: word.audio,
      show: false
    });
  }
```

5.2.3.3 Weather

The weather component is implemented in a completely different methodology. The weather module is a plug-in component, which is inserted into the program following a configuration-routing-call routine. Firstly, the Habiter need to configure and refer to the plug-in module globally.

```
  "permission": {
    "scope.userLocation": {
```

```
      "desc": "retrieve weather information according to the user's current location"
    }
  },
  "usingComponents": {
    "weather": "plugin://weather/weather"
  },
  "plugins": {
      "weather": {
        "version": "1.1.0",
        "provider": "wxc5af96529fe97d4b"
      }
    }
```

Since the weather is varying with the user's location, our Habiter has to ask for user's permission explicitly, "permission". The "usingComponents" identifies the address of the plug-in service. Lines in "plugins" names the subcomponent module and declare the version and provider information.

Next, in the page intended to insert the plug-in, declaring it beforehand in .json is needed.

```
{
  "component": true,
  "usingComponents": {
    "list": "../components/list/list",
    "index": "../components/index/index",
    "chart": "../components/chart/chart",
    "nav": "../components/nav/nav"
  }
}
```

In the end, the tag can be directly used in the frontend.

5.2.3.4 Daily Recommendation

Our daily recommendation function are implemented in two different parts. One is displayed with a popping window. The other is implemented together with the weather module.

Figure 5-12: Daily Recommendation Window

Figure 5-13: Daily Recommendation Page

Figure 5-14: Chat Bot

The recommendation inside the weather module is provided with the plug-in server. The popping window is implemented with querying the current temperature through the connected channel.

```
if (clickPigNum === 5) {
  wx.showToast({
    title: recommendMsgZ,
    icon: 'none',
    duration: 7000
  })
```

5.2.3.5 Chat Bot

The intelligent chat bot is implemented with the help of a third party open platform, ***OLAMI***. They provide free AI accessing APIs. The introduction to this component will also start with the frontend. The frontend outcome is shown above in figure 5-14.

The frontend is composed of a scroll view, upper bar and bottom box. Among them, the scroll view is the most tedious one to implement, due to its response to the user's message sending. Below is part of the code for scroll view implementation.

```
<scroll-view scroll-y style="height:{{scrollHeight}}rpx;background:#f7d14a"
    scroll-into-view="{{toView}}">
  <nav
  class="nav-instance"
  showIcons="{{['back']}}"
></nav>
  <view class="title">
    <text class="titleText">Habiter</text>
  </view>
  <view wx:for="{{messages}}" wx:for-item="i">
    <view id="{{i.id}}" class="padding clearfix" wx:if="{{i.me}}">
      <image class="right-Avatar padding_half" src="../../images/profile.jpg"></image>
      <view class="right-Triangle"></view>
      <view class="right-speech-message padding_half" wx:if="{{i.speech}}"
          data-filePath="{{i.filePath}}" bindtap="playSpeech">
        <view style="display:inline-block;">
          <image style="width:52rpx;height:42rpx;vertical-align:middle;margin:0
              6rpx;"
              src="{{playingSpeech==i.filePath?speechIcon:defaultSpeechIcon}}"></image>
              {{i.seconds}}
        </view>
...
```

The ***OLAMI***'s intelligent chat bot service is utilized with three steps, authentication, send request and receive feedback. The authentication information is stored globally.

```
globalData: {
  host: 'https://homolo.top',
  NLPAppkey: "5625eac76f9941d19cb478dad10a9042",
  NLPAppSecret: "c902d1eb8ef441fa874b51177b49a05c",
  NLPUrl: "https://cn.olami.ai/cloudservice/api",
  NLPCusid: "fe80::6692:1b74:43b8:66e7%utun1",
 emojisEn: ['bugaoxing', 'guai', 'qinqin', 'lengmo', 'qie', 'mianqiang', 'chijing',
     'tushe', 'hehe', 'hu', 'yi', 'haha', 'ku', 'pen', 'weiqu', 'kaixin', 'deyi',
     'nu', 'exin', 'jingku', 'jingya', 'han', 'huaji', 'kuanghan', 'shengqi',
     'yiwen', 'zhenbang', 'shuijue', 'xiaoyan', 'mengmengda', 'bishi', 'yinxian',
     'heixian']
```

```
  },
```

In the chat box page's javascript file, the following part will send request to the given global authenticated platform.

```
    wx.request({
      url: nliUrl,
      data: {
        appkey: appkey,
        api: api,
        timestamp: timestamp,
        sign: sign,
        rq: rq,
        cusid: cusid,
      },
      header: { 'content-type': 'application/x-www-form-urlencoded' },
      method: 'POST',
      success: function (res) {
        var resData = res.data;
        console.log("[Console log]:NLIRequest() success...");
        var nli = JSON.stringify(resData);
        typeof arg.success == "function" && arg.success(nli);
      },
      fail: function (res) {
        console.log("[Console log]:NLIRequest() failed...");
        console.error("[Console log]:Error Message:" + res.errMsg);
        typeof arg.fail == "function" && arg.fail();
      },
      complete: function () {
        console.log("[Console log]:NLIRequest() complete...");
        typeof arg.complete == "function" && arg.complete();
      }
    })
```

In the end, the returned message will be parsed.

```
var answer = nliArray[0].desc_obj.result;
let msg = answer.replace(/         /g, "Habiter")
let contents = util.getContents(msg)
let id = 'id_' + Date.parse(new Date()) / 1000;
wx.getStorageSync('userInfo').avatarUrl, speech: false } // this one checks if the
    user has authorized the miniprogram with his/her profile
let data = { id: id, contents: contents, me: false, avatar:
    "../../images/mascot.png", speech: false }
```

Conclusion

Since 2017, when the WeChat Mini Program come into being, the development of the WeChat Mini Program has contributed to a new generation of mobile Internet industry is developing toward the development of “micro, light, and small”, like the WeChat Mini Program, this revolutionary lightweight “APP” that does not require downloading, runs out, and simultaneously has information publishing, advertising, and service functions, will become the best choice to replace the mobile client APP [3].

The usage paradigm of WeChat Mini Program, “direct search and open wihtout installation”, “use with wechat account” and “quit at anytime” are extremely suitable for software carrying daily light weight functions, including bookkeeping, habits cultivation, daily dressing or diet recommendation. In addition, the AI subsystem, Habiter chat bot is one significant component, it will advance the appeal and profitability of the program.

The most exciting thing is that “Habiter” is actually a chatbot. “Habiter” will interact with user in an artificial chatting manner. This method greatly increases the interest of the program and enhances the user’s interactive experience. By doing this, the problem that users often feel bored when using previous habit formation smartphone application will be solved efficiently.

In conclusion, our miniprogram Habiter, with “Microservice Architecture”, multiple loose coupled functional components, light weight package and interaction oriented design, successfully provide each user a considerable, intelligence and simply accessible “housekeep“ Habiter, to help clients form their habits and accompany their life in a light weight manner.

References

[1] Mohammed Akour et al. “Software Architecture Understandability of Open Source Applications”. In: *International Journal of Computer Science and Information Security* 14.10 (2016), p. 65.

[2] Philippe Dugerdil and Mihnea Niculescu. “Visualizing software structure understandability”. In: *2014 23rd Australian Software Engineering Conference*. IEEE. 2014, pp. 110–119.

[3] Lei Hao et al. “Analysis of the Development of WeChat Mini Program”. In: *Journal of Physics: Conference Series* 1087 (Sept. 2018), p. 062040. DOI: 10.1088/1742-6596/1087/6/062040. URL: https://doi.org/10.1088/1742-6596/1087/6/062040.

[4] Charles W Krueger. “Software reuse”. In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 131–183.

[5] Phillippa Lally and Benjamin Gardner. “Promoting habit formation”. In: *Health psychology review* 7.sup1 (2013), S137–S158.

[6] Chung-Horng Lung et al. “An approach to software architecture analysis for evolution and reusability”. In: *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. 1997, p. 15.

[7] Agile Manifesto. “Principles behind the agile manifesto”. In: *Retrieved July* 23 (2001), p. 2005.

[8] Katarzyna Stawarz, Anna L Cox, and Ann Blandford. “Beyond self-tracking and reminders: designing smartphone apps that support habit

formation". In: *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 2015, pp. 2653–2662.

[9] Joost Visser et al. *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code*. " O'Reilly Media, Inc.", 2016.

[10] Lloyd G Williams and Connie U Smith. "PASASM: a method for the performance assessment of software architectures". In: *Proceedings of the 3rd International Workshop on Software and Performance*. 2002, pp. 179–189.