An approach to auxiliary code generation for mobile environment: a case study of thrift-based codes conversion

Binyang Qiu*

School of Computer Engineering and Science,
Shanghai University,
Shanghai, China
Email: qby98@shu.edu.cn
*Corresponding author

Qiming Zou

Computing Center,
Shanghai University,
Shanghai, China
Email: kim@shu.edu.cn

Abstract: In the development of mobile applications based on location information, system development for multisegment path planning and distant monitoring sites is complex. Developers are required to manually write inbound and outbound trigger functions for each site after multiple integrations of the planning results for a single route. In addition, each time that the route is changed, the developer needs to re-plan and re-bind the trigger function. This paper proposes an auxiliary code generation method based on the thrift application. Based on this method, a system for automatically generating code through visual design is designed, and an example of generating auxiliary development code for an application (APP) is implemented. Thrift provides multilanguage code generation services and implements a predefined point of interest (POI) recommendation service for real-time returned location information. This approach improves the efficiency of application (APP) development and also provides the ability to respond to changing process requirements.

Keywords: mobile environment; process customisation; thrift; code generation.

Reference to this paper should be made as follows: Qiu, B. and Zou, Q. (2019) 'An approach to auxiliary code generation for mobile environment: a case study of thrift-based codes conversion', *Int. J. Intelligent Internet of Things Computing*, Vol. 1, No. 1, pp.53–73.

Biographical notes: Binyang Qiu is currently an undergraduate student at the School of Computer Engineering and Science, Shanghai University, China. His main research areas are software engineering and machine learning.

Qiming Zou received his PhD in Machine Manufacturing from the Shanghai University, Shanghai, China, in 2015. He is currently an Assistant Professor at the Shanghai University, China. His research interests include cloud computing and grid computing computer aided manufacturing.

1 Introduction

At present, various mobile environment-based applications (APPs) are emerging, including Dazhong Dianping, Meituan, Baidu map, etc. However, in mobile environments, developers need to manually bind trigger functions for sites to monitor mobile devices and provide specific point of interest (POI) recommendation services. The development of this kind of APP code is complex, and the path often needs to be divided into multiple destinations and multisegment path planning. Therefore, it is necessary to integrate each route into code after planning. If the route needs to be changed, the changes are large. It is not appropriate to use traditional development methods. Furthermore, due to changing business needs, constant code modification is not efficient. Model-driven development technology improves the development efficiency and features the ability to solve the ever-changing business process customisation problem. Business process customisation is the act of defining the whole workflow from the start to the finish in software, thereby allowing enterprises to customise their workflows using different software to adapt the software to the companies' existing workflows (Wikipedia, 2013). Traditional process customisation is designed by the designer and developed by the developer according to the documentation. It takes much time and effort to communicate and discuss the requirements, and the efficiency is low. In addition, when facing changing needs or multiple needs, this approach will often choose to abandon many up-front efforts or repeat many of the same developments. Thus, we seek a method that combines design and development, generates code from requirements, and combines interactive code generation technology to mass-produce the same pattern of code to improve software efficiency. In 2001, after the Object Management Group proposed the concept of the model driven architecture (MDA), the technology of automatically generating code through the initial design of the model was supported by many people. The idea is to analyse the model at a higher level of abstraction, use the MDA tool to identify the read model information, and automatically convert it to code in languages such as C++ and Java through different standard mapping methods. Therefore, when the high-level needs change, only the abstract model needs to be modified. The MDA program will regenerate the code based on the mapping.

This paper proposes an APP auxiliary code generation method for the mobile environment. The generated code can be directly inserted into the development code of the APP. The method mainly adopts the concept of model-driven development. After the developer completes the design through the visual interactive interface, the developer fills in the custom function code, and finally, the system is integrated to generate executable code. The code that is generated by design is run in the basic functional framework in the form of a configuration file. The code extracts and models

the process-related semantics that are generated by the business and business data so that the process logic is completely separated from the APP logic, which enables the development to adapt to the rapidly changing business processes and also allows the various steps of the design phase to be returned. Thrift is an interface description language (IDL) and binary communication protocol that was developed by Facebook for large-scale cross-language service development. Thrift is used to define and create cross-language services. To enable users to customise some of the function code, the method in this article uses thrift for the code frame generation, which provides an interface for the user to write custom code and, finally, combines this code into the overall generated code.

The case scenario that is studied in this paper is the mobile APP environment. In recent years, maps have become increasingly more important in life, and APPs based on web map location services are becoming increasingly more popular. APIs are widely used in programmers' development as a general network programming interface. The Maps API allows third-party websites to use the information and functionality in the Maps Services website database using API programming. Through its direct embedding of iframe, as well as various kinds of interfaces such as HTTP+XML and web service, the Maps API can satisfy the service APPs of various C/S and B/S architectures. According to incomplete statistics, the number of websites using the Maps API has exceeded one million, and the number of developers using the API has reached more than two million. The technology in this article will be illustrated using a thrift-based multilanguage code custom generation system for mobile environments and an implementation case for using the system.

The final planning results will be provided in the available code and XML data storage. The system does not monitor the location information of the terminal in real-time. The system sets the site location of the inbound and outbound stations in advance and the corresponding trigger function. When the mobile device arrives at the site and conducts an inbound or outbound behavior, the corresponding function is triggered. A signal is sent to the server, the server reacts, and the corresponding data are returned. The data that are returned here contain the POI that the system predefined. Considering location-based changes, it is necessary to increase the recommended service based on location. Thrift is responsible for the remote procedure call (RPC) communication between the mobile terminal and the server. The entire service is completely independent and only has coupled data with other modules of the APP so that the generated code can be directly inserted into the APP.

The rest of the paper is organised as follows. In Section 2, related technology research in the field of process customisation and code generation are introduced. In Section 3, the entire process and functions of the system are designed and illustrated. In Section 4, the implementation method of the visual interaction between the system and the user is developed, and the back-end implementation of each function in the system flow is detailed in Section 5. In Section 6, the communication model of the system in the mobile environment is built. In Section 7, the code generation technology that is adopted by the system is explained, and the error detection mechanism and the generated code are addressed. Finally, in Section 8, the article is summarised and future research plans are given.

2 Related work

2.1 Process-oriented service dynamic configuration

The key to solving business process changes and allowing the corresponding examples to change is to separate the process logic from the APP logic. Only when the abstract business process changes can we only adjust the process logic without having to consider changing the APP logic of the implementation (Pentland, 2003). In academia, research has proposed some architectures for separating process logic, such as the four-layer process-driven architecture model that was developed by Strnadl (2006), which clearly addresses current business and IT issues. Meanwhile, this research demonstrates the applicability of the model at the theoretical descriptive and prescriptive levels. Humphrey (1992) proposes a process MDA, which is also divided into four layers: the technology integration layer, the service layer, the information layer, and the process layer.

However, these architectures only model the surface representation of the business, and no process and process-related semantics that are generated by the business data are extracted and modelled. Therefore, these structures do not achieve a real separation of the process logic and APP logic, and the business process changes based on these models still require changes to the implementation code. Jain and Schmidt introduced a service dynamic configuration model that separates the implementation of services from their configuration times. This mode increases the APP's flexibility and scalability by configuring its constituent services at any point in time (Jain and Schmidt, 1997). Kwon and Park (2018) proposed a system architecture for dividing the roles of users accessing web services, managing user rights based on each role, and providing users with user rights and appropriate service resources. They modularise the functions that they want to provide, and then open the corresponding service portfolio based on permissions, which is also a new idea. That is, all services are prepared in advance, and the process is combined according to the configuration requirements at the beginning of the business. Dong gives a semantic analysis method based on OWL-S for the dynamic configuration of web services. The strategy based on the inference model solves the problem of dynamic reconstruction. That is, the system can be controlled at the meta level without changing its underlying implementation, which is also a typical implementation case that abstracts the business process logic (Dong, 2008). The author gives the refactoring constraints in the XML description and abstracts the strategy to obtain flexible business processes. Liu et al. (2006) proposed a new remote dynamic component configuration method based on JMX technology that modifies the component configuration to take effect immediately at runtime, and the information system can provide uninterrupted operating capabilities. Wang et al. applied the mature evolution and adaptation of internetware to service-oriented architecture (SOA), and this allowed perceived changes to drive the changes in SOA-based APPs. The work performed by Wang et al. (2014) guides these changes at the top or business level rather than guiding these changes at the coding level.

The system in this paper only needs to re-design the route file using the system and replace the original route file instead of changing the code after generating the code. This system can extract the process design entirely from the specific implementation to solve the changing business process modelling problem.

2.2 Code generation technology

With the development of the MDA, code generation research has gradually shifted from the compilation field to the high-level language development field. Focusing on the use of relevant models for the automatic conversion of executable code promotes the development of software scale industrialisation. Syriani et al. (2018) surveyed template-based code generation technology and stated that MDA is already a mature technology. Albert et al. introduced the automatic code generation engine that addressed model driving and model conversion. The design method and ideas were implemented in the automatic code generation that was based on the unified modelling language (UML) class diagram, and the rules and the strategies of the model conversion were introduced in detail (Albert et al., 2010). Gaedke and Rehse (2000) introduced an automated code generation method based on a template and reusable component detection strategy for the webcomposition repository, which is an important tool for retrieving and classifying large component sets. Imam et al. (2014) introduced a rule-based code generation strategy. Bak et al. (2006) introduced a method for generating code using annotations. Gomes and Baunach (2019) proposed an automated RTOS portability framework that used the interaction between modelling software and hardware to generate the underlying code. Hu et al. proposed a code generation method based on model rules. This method addresses the layout problem of the Android embedded APP interface and defines the constraint rules (Hu and Zhang, 2014).

There are no implementation details for UML, a single UML model is not enough to complete code generation, or some model elements may not be directly converted to source code. Viswanathan and Samuel (2016) focused on workflow modelling and automation and developed an algorithm, Am_To_Prototype, to generate code from a combined activity model and sequence diagrams.

After the code is generated, the modification of the error cannot be solved by regenerating code each time. In recent years, there have been many solutions to this problem. Possatto and Lucrédio (2015) developed a mechanism to semi-automatically detect and propagate changes from reference code to templates that keeps them in sync and reduces workloads. Gusarovs et al. (2017) proposed a method for generating simplified LISP code from a double hemisphere model that focused on the dynamic aspects of the system.

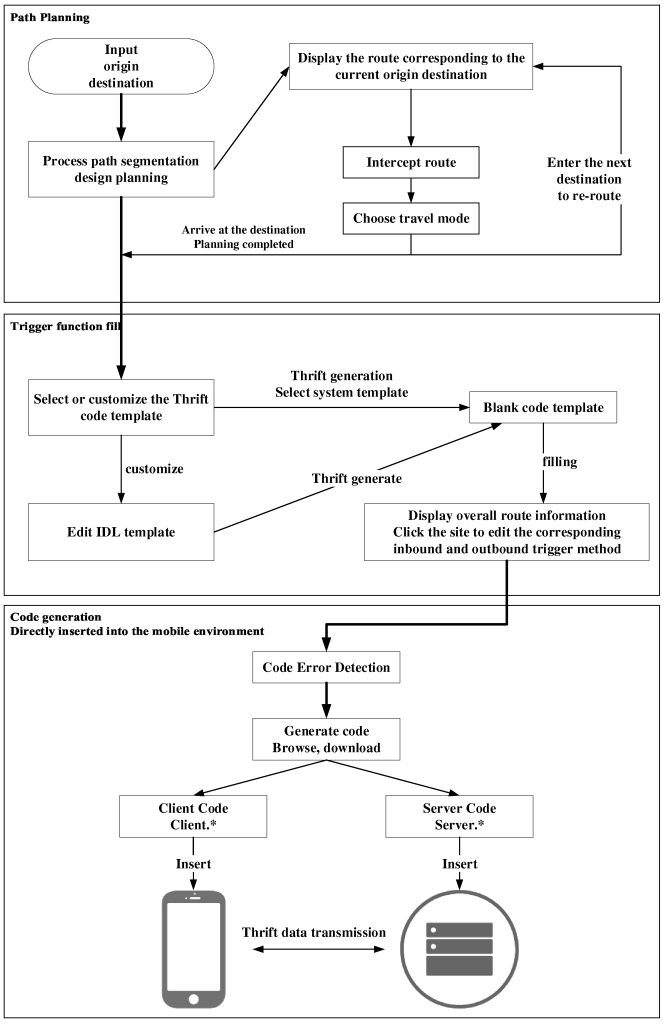
In recent years, there has been much research in the field of code generation combined with machine learning, which is also the research direction that the technology proposed in this paper follows. Malik et al. (2018) proposed a code generation framework for template machine learning algorithms to minimise the manual intervention in developing machine learning solutions. Szydlo et al. (2018) proposed the source code generation concept of the machine learning model and the generation algorithm of common machine learning methods. Yavuz et al. (2016) and others used code generation technology to build a GeNN using a flexible and extensible interface.

3 Process overview

The system that is implemented using the technology that is proposed here needs to realise the following functions: a multisegment path starting from the origin, a plurality of driving modes that can be freely planned, and a trigger function that is written for

each station in the route. The trigger function is written in multilanguage support and allows users to customise the code template. Based on the thrift implementation of the multilanguage trigger function writing function and the framework of the fixed basic service model, the method interface is applied for the user to fill the custom code. The user's freedom is improved while ensuring the characteristics of MDA abstract modelling, and the characteristics of thrift are utilised to construct an efficient RPC communication service in the mobile environment. The overall flow chart of the system is shown in Figure 1.

Figure 1 Process overview



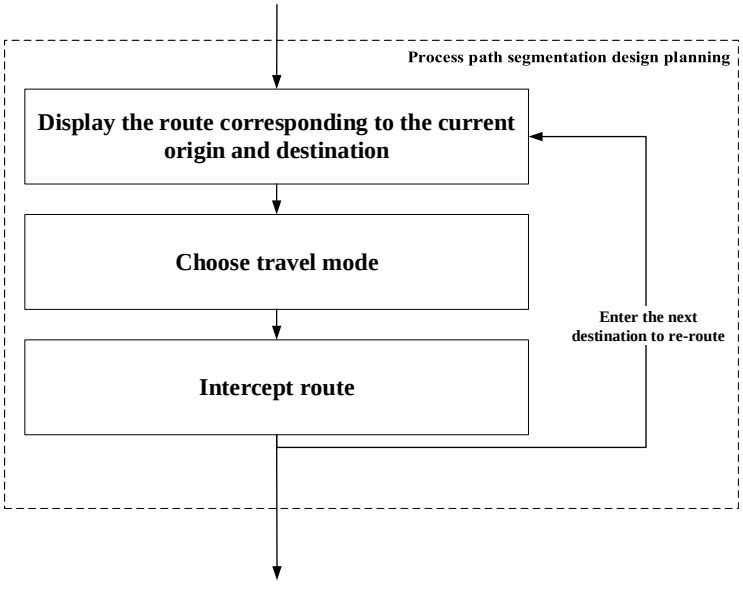
The whole process is divided into three parts: path planning, trigger function filling, and code generation and insertion into the mobile environment. First, in the path planning part, the user plans a route through multiple destinations under the interactive guidance of the system. After the route design is completed, the system will set a trigger function

for each route site to monitor the movement of devices in the mobile environment and push the POI at a fixed point. Therefore, in the trigger function writing part, the method of sending location information to the server and the POI push method of the server to the terminal device in the mobile environment need to be filled in the inbound and outbound trigger function. In the last part, after the function is filled, the system packages all the code for download. Users can insert code directly into the APP environment to run it.

3.1 Process path segmentation planning

The system completes the route planning of a single origin and a single destination using the AutoNavi Map API. The AutoNavi Map API is the interface of a map solution that is provided by AutoNavi. As a well-known navigation and map service provider in China, Google also uses AutoNavi's map data in China due to its high data credibility and stable service. The specific process is shown in Figure 2.

Figure 2 Process path segmentation design planning process



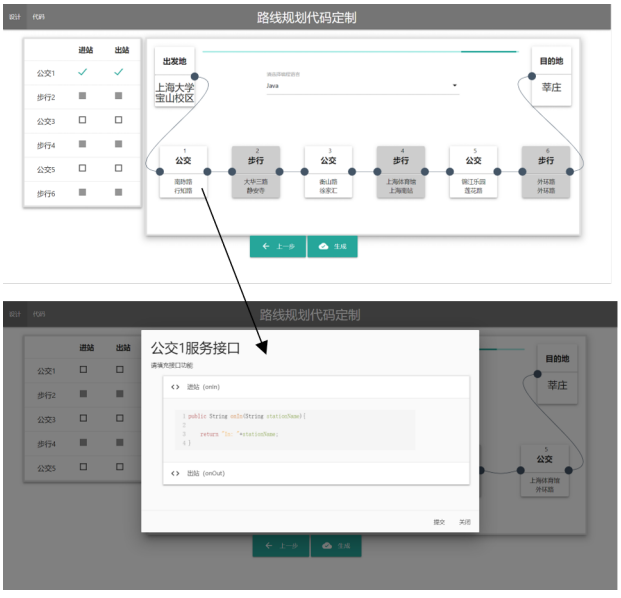
In the route planning stage, the system implements single route planning many times to freely design each segment route by asking for the next destination and mode of travel.

3.2 Multilanguage trigger function

Through thrift, the trigger function can be written in many languages. Apache Thrift is an efficient, extensible cross-language communication framework that supports multiple programming languages. Apache Thrift makes generating the target code easy and convenient.

Before writing, the user needs to design a code template for the trigger function. The default two IDL code templates can be selected in the system or a custom template can be used. After selecting the programming language and submitting the thrift template, the system will automatically generate the corresponding interface method, and the user needs to fill the method in the system. If there is no way to enter and leave the station on foot, it cannot be filled. The user needs to set the in and out of station methods for buses, trains, ferries and other forms of public transport. The page is shown in Figure 3.

Figure 3 Trigger function code writing page (see online version for colours)



After completion, the submit button can be clicked to make a submission to the server while the front-end is blocked and is waiting for the server to check the code for errors. After receiving the code, the server generates the corresponding interface implementation class and compiles and runs the code. Eventually it returns as a success if no error occurs. If there is an error, the error message is captured, and the vital information is returned to the front end. Only if the code is correct is the submission is successful, and the successful submission is checked using the table on the left.

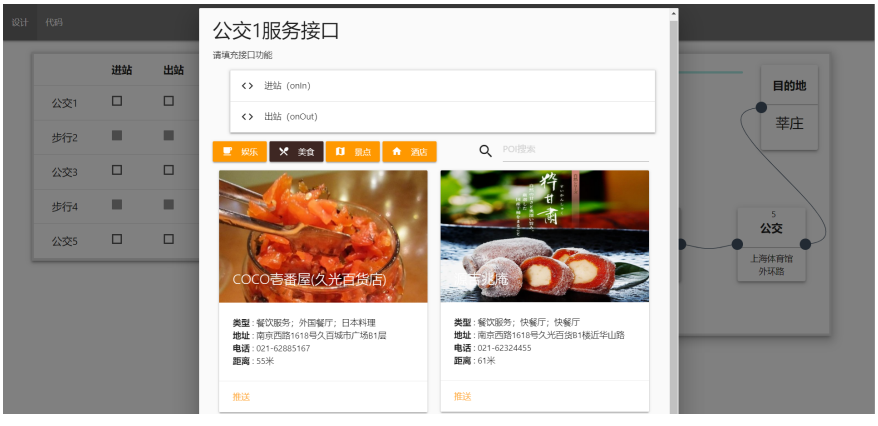
3.3 POI recommendation

POI refers to all geographical entities that can be abstracted into points. Each place, including houses, companies, communities, schools, parks, hospitals, shopping malls, and many more, is a POI. When the interface code is written, the system provides the POI recommendation function, which is shown in Figure 4. When the user opens the POI recommendation function, he can select the relevant POI through classification or a keyword search when writing the code.

For convenience, the POI's search range is set to be less than 100 metres, and a maximum of ten POIs are selected according to the ranking of the scores. The provided

POI information includes the name, type, address, telephone, and distance. This article provides four categories. If the user needs to query other categories, he can conduct a keyword search.

Figure 4 POI recommendation (see online version for colours)



The last saved number is the unique identification number of the POI. The AutoNavi Map API has a unique identification number for more than 75 million POIs in China. When the client runs the trigger, it only needs to search the POI information from the AutoNavi Map API using this identification number.

3.4 Auxiliary code communication and execution mode

After the interface method is edited, click generate to generate the code in the background, and then one can browse and download the code online. All downloaded files are shown in Table 1. * represents the different suffixes of the files in different languages.

Table 1 Download file content

File function	*File name*
Route stored in XML	Route.xml
XML route read operation tool class	XMLutils.*
Thrift template code	Template.thrift
Thrift server code	Server.*
Thrift client code	Client.*
Thrift generated function implementation code	ServiceImp.*

Route.xml is the planned route file, which contains the total distance, the toll, the passing site, the binding name of the inbound and outbound trigger functions, and the ID of the POI, and the file can be downloaded separately. XMLutils.* is a packaged XML operation function that can be directly called. Template.thrift is an IDL template file for the thrift generated code that is used to design some information about the generated

code. Users can use this file to conduct local testing using thrift. Server.* is the server code of thrift, which can be run directly to open the server and wait to receive the client Client.* message. The prepared trigger function code is inserted in Client.*, and the operation of Client.* relies on various methods in ServiceImp.*.

The downloaded code can be inserted directly into the APP code. Because of the high independence of the code, data coupling can be built into any source code as a complete module. After inserting the thrift client code and service implementation code into the mobile APP, the server code is placed on the server. After the first planned route is provided for the mobile terminal, the feedback from the client can be given. The mobile terminal triggers the thrift client to provide feedback information to the server in each entry and exit station, which allows the server to monitor the mobile terminal location information and also pushes the merchant information from the server to the client for the POI recommendation according to the set push information. Thrift is responsible for the data transmission in the whole process and transfers data into a binary data stream with high speed and efficiency.

4 Visual interaction design

The front-end and back-end functions of the system are closely related. Therefore, the visual interaction design of the system adopts an interactive design. This design gradually guides users to complete the whole design through the continuous inquiry and intuitive presentation of the resulting route. There is no actual code implementation until the final padding of the trigger function code. Each step allows the user to go back to the previous step to redesign the method before the build is exported. Then, the system simply regenerates the configuration file via dynamic configuration and regenerates the code.

4.1 Route storage and display

Each route is stored in the queue of the front-end cookie. Each time the fixed origin and destination are passed to the backend, a JSON dictionary is returned. The key of the dictionary is the paragraph code that is based on the transfer segment. The value is the specific information dictionary of the segment, where ‘busline’ indicates the name of the vehicle for this paragraph, and ‘spots’ indicates the array of sites for the route. An example of the return value is as follows.

```
{
    "seg1": {
        "busline": ["Subway Line 7(Meilan Lake--Huamu Road)"],
        "spots": ["Nanchen Road", "Shangda Road", "Changzhong Road", "Dachang
            Town", "Xingzhi Road", "No.3 Dahua Road", "Xincun Road", "Lan'gao
             Road", "Zhenping Road", "Changshou Road", "Changping Road", "
            Jing'an Temple"]
    },
    "seg2": {
        "busline": ["Subway Line 1(FUjin Road--Xinzhuang)"],
        "spots": ["Hengshan Road", "Xujiahui", "Shanghai Indoor Stadium", "
            Caobao Road", "Shanghai South Railway Station", "Jinjiang Park",
            "Lianhua Road", "Waihuanlu"]
    },
    "seg3": {
        "busline": ["Walk"],
        "spots": “”[-]
    }
}
```

The front-end route data processing is shown in Figure 5. The front end links the spots of each segment into a string array route in sequence and establishes an integer array len to store the number of sites of each segment. Then, a two-row table on the page is created to show the current route. The first row shows the route array, and the second row shows the vehicle name based on the len array to control the length of the cell. A slider that is supported by the noUiSlider framework is added at the top of the table for the user to obtain the selection. After the user drags and selects, they click on the switch to switch between the bus and walking.

Figure 5 Front-end route data processing

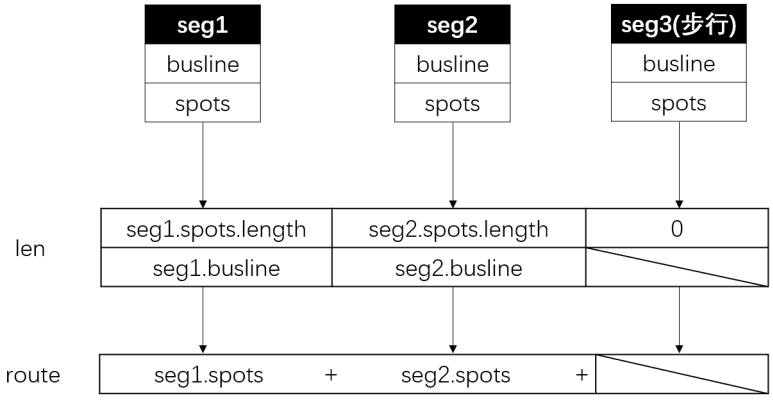
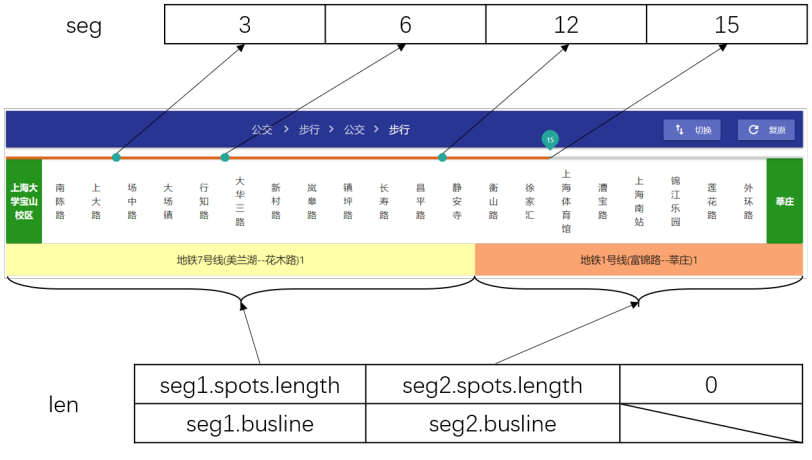


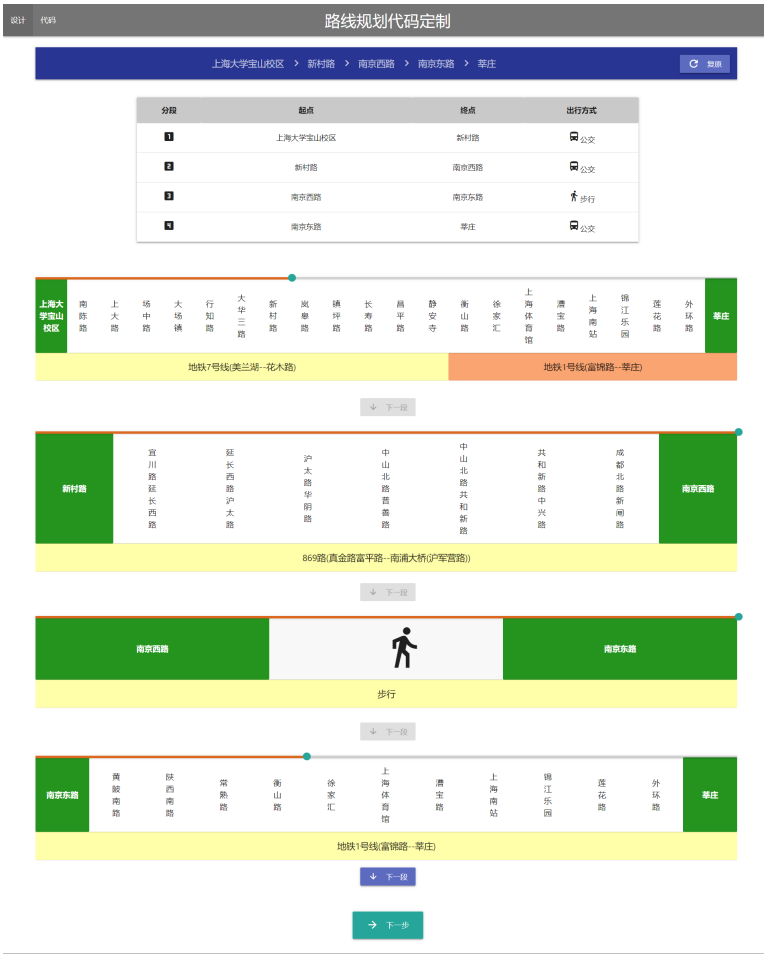
Figure 6 Display and interception processing (see online version for colours)



The record value that is obtained each time is the position integer of the current site in the route array. In a route, this record is stored in the integer array seg. The obtained display and storage method is shown in Figure 6. The display of the route consists of two levels of tables with all the sites on the upper level and the vehicles on the lower floors. Since each site has the same width, the length of the lower layer can be

determined by calculating the number of spots. That is, seg1.spots.length in Figure 6 is calculated as the number of spots.

Figure 7 Route segmentation design example (see online version for colours)



After the design completes the current route, the front end will package the array name and value into a dictionary using the route, len, and seg arrays, and store them in the cookie's queue records. After choosing the means of transportation, proceed to the design of the next route. The final segmentation design results are shown in the example in Figure 7.

4.2 Design trigger function

The user needs to select the site where the function is to be written. Therefore, it is necessary to visually display the sites where the user has entered and exited the entire route. Based on the resulting cookie data in the design process, Algorithm 1 describes the process of extracting the inbound and outbound sites.

Algorithm 1 Inbound and outbound site extraction

Input:
The sites queue in cookie, $Records$;
Output:
Site name sequential string array that needs to set inbound and outbound trigger functions, $Passed$;

```
for cur = 0 to Records.length do
  seg ⇐ Records[cur].seg
  route ⇐ Records[cur].route
  len ⇐ Records[cur].len
  temp ⇐ 0
  for pos = 0 to seg.length do
    if seg[pos] > len[temp] and seg[pos − 1] < len[temp] then
      passed ⇐ passed + route[len[temp]]
      passed ⇐ passed + route[seg[pos]]
      temp ⇐ temp + 1
    else
      passed ⇐ passed + route[seg[pos]]
    end if
  end for
end for
```

The input of Algorithm 1 is the queue records that are stored in the cookie in the previous part of the path plan. The queues containing all the site arrays, the array of the number of sites of each vehicle, and the destination site array seg that are generated after the route is formed are obtained. The output is the site name string array that was passed, which needs to set the inbound and outbound trigger functions. The algorithm sequentially extracts the queue headers in the cookie. For each team head element, the corresponding site name is removed from the route according to the value in seg and placed in the site record of the passed array. If there is a transfer in the process, you also need to record the transfer site in the passed array.

5 Realisation of back-end technology

5.1 Data processing

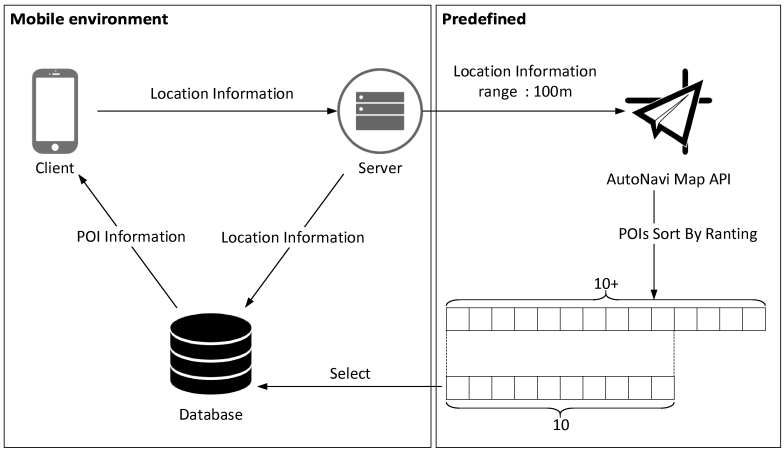
The parameters that are accessed by the path planning interface of the AutoNavi Map API are the two latitudes and longitudes of the origin and destination and not the location names. The longitude and latitude of the location are searched using the geocoding interface of the AutoNavi Map API and then passed to the path planning interface for use. The return value format is shown in Table 2.

The system issues an HTTP GET request using the splicing URL and sets the return type to XML. This article uses dom4j to process the XML and directly determine the best route in the first transit so that the root directory directly locates the first transit of the segments for operations.

Table 2 API return value format

Field name	*Field meaning*
status	Return status
info	Returned status information
count	Number of public exchange programs
route	Public exchange information list
origin	Starting point coordinates
destination	End point coordinates
distance	Walking distance from the start and end points
taxi_cost	Taxi fare
transits	Public exchange plan list
transit	Public exchange program
cost	This transfer plan price
duration	Expected time of this transfer plan
nightflag	Is it a night bus
walking_distance	Total walking distance of this program
segments	Transfer section list
walking	This section of walking information
bus	Bus navigation information for this section
entrance	Subway entrance
exit	Subway exit
railway	Information on the train

Figure 8 POI push operation mechanism



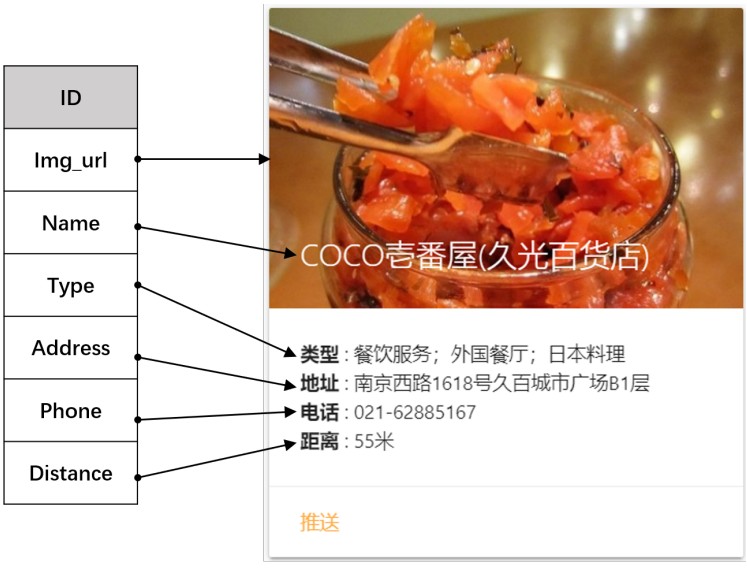
5.2 POI recommended data processing and transmission

The research scenario of this paper is the mobile environment. During the whole location monitoring process, the mobile terminal location information will be continuously returned. We can use this information to push the surrounding POI information, and the pushed content is predefined when writing the trigger function. The push location points are marked in the client's route, but the specific push content is stored in the

server's database. Considering that the business information may change at any time and reduce the code framework that is inserted by the client, the data are loaded on the transmission, and each time the server pushes the POI information that the client needs to display. The overall push architecture is shown in Figure 8.

The specific information data structure of the push is shown in Figure 9. The ID is a unique identifier of the POI, which is processes as a mark on the route, and one location point allows multiple IDs or multiple POI recommendation points. All display images are saved in the webserver. Img_url is the network address of the displayed image. Each time only an Img_url string data are transmitted, the client can request the network to display the image, thus reducing the transmission burden.

Figure 9 POI push information data structure (see online version for colours)

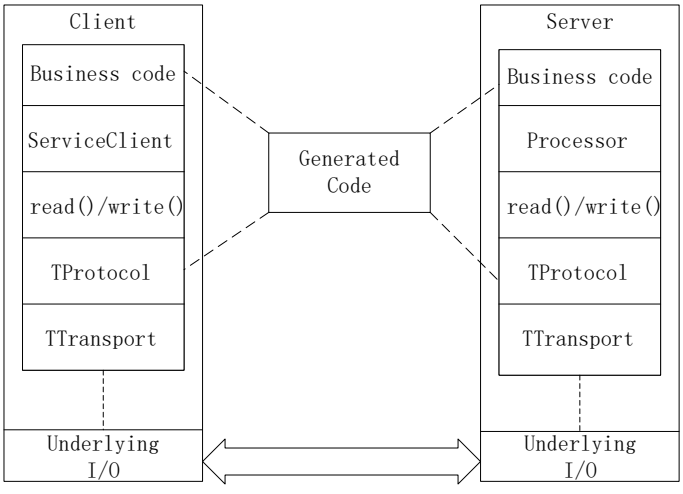


6 Communication model in mobile environment

Thrift implements the C/S mode, which uses the code generation tool to generate the server-side and client-side code from the IDL template code, thus enabling cross-language support between the server and the client. The user declares his own service in the IDL file. After compiling, the service will generate the code file of the corresponding language, the client will call the service, and the server will provide the service. Figure 10 shows the overall architecture of thrift.

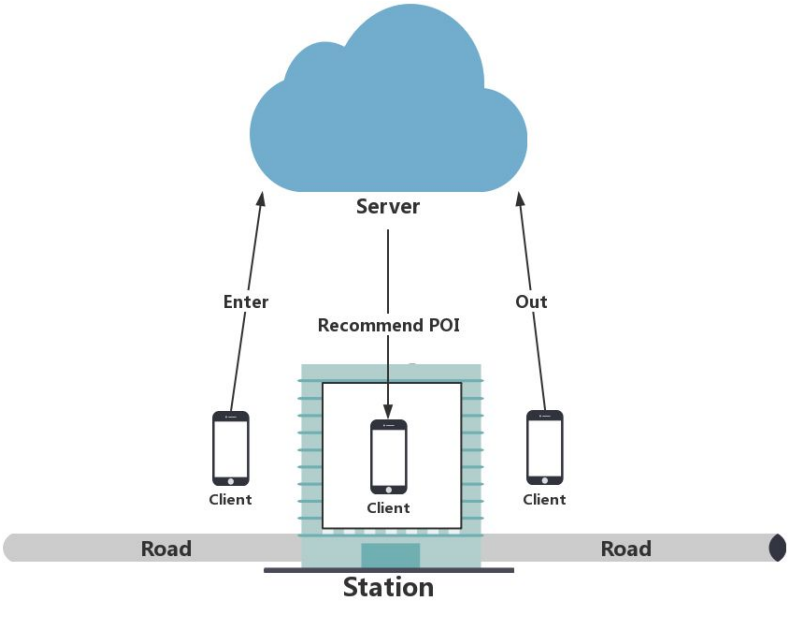
TTransport is the transport layer, which is responsible for receiving and sending message bodies with the underlying I/O in a byte stream, regardless of the data type of the message body. This way decouples the TTransport underlying thrift from the rest of the system. The protocol abstraction layer defines a mechanism for mapping the in-memory data structures into transportable formats, which are responsible for data type parsing. The processor encapsulates the operation of reading data from and writing data to the data stream, which directly responds to client requests.

Figure 10 Thrift architecture



As shown in Figure 11, the RPC communication model of the system in the mobile environment.

Figure 11 Generated code for RPC communication model in mobile environment (see online version for colours)



Here, the client's inbound and outbound signals and the server's recommended POI information are transferred to binary data for transmission using thrift. After testing, it is found that thrift is more efficient than the method that was implemented using HTTP protocol.

7 Code generation phase

7.1 IDL template definition

Thrift defines the data types and services that are transmitted using interface definition language (IDL). The thrift code compiler automatically generates the code for the target language from the IDL file and implements the RPC protocol layer and transport layer using the generated code.

To reduce the coupling, the function that is generated by the interface code of the system encapsulates a web service on the server using Axis2. After editing the submission on the webpage, the background sends the template code to the encapsulated server interface. If the syntax is correct, the server will return the generated code. A default interface template is shown.

```
service MsgService{
   string onIn(1:string stationName)
   string onOut(1:string stationName)
}
```

Each time the inbound triggers onIn(), the outbound triggers onOut(), and it is combined with the hardware, GPS location information and/or other information. The interface code can automatically trigger the user to provide accurate service.

7.2 Code error detection

The code troubleshooting in this article uses the language compiler to check for errors. Individually, after the completion of the commit code generation, the packaged program is run in the command line environment on the server. In addition, the error information of the console is captured and returned to the front end.

This paper use Java's runtime class's exec(cmd) method to execute the commands on the command line. However, it should be noted that if you directly execute the method and then use BufferedReader to read the output stream of the buffer, you will find that if there is an output statement in the program, the program will block it in the waitfor() function because when executing the exec(cmd) of the runtime object, the JVM starts a child process that establishes three pipe connections with the JVM process: the standard input, standard output, and standard error streams. The standard buffer size of the Java run window is fixed. If the standard output stream and the standard error stream are always output and the JVM does not read them, the buffer cannot be written until the buffer is full, thus blocking it in the waitfor() function. The solution in this article is to open two threads to read the standard output stream and the standard error stream separately.

7.3 Generating code overview

7.3.1 Thrift client code Client.py

This article uses python as an example to more intuitively expose the trigger function that needs to be filled for the user to write. When filling in the visual interface, you can

only see the code framework below. When generating the code, insert this code directly into the client request code file Client.py. The code that needs to be populated based on the IDL template is as follows.

(see online version for colours)

```
# coding=utf-8
def onIn(stationName):
    # Your Code
    return 'Inbound'

def onOut(stationName):
    # Your Code
    return 'Outbound'
```

Thrift also generates a ServiceImp.py file based on the IDL template, which stores the various function codes of the thrift client. All functions and object configurations are generated according to the definitions in the IDL file. The aforementioned Client.py implements data transfer by calling various functions in ServiceImp.py. The code for Client.py is as follows.

(see online version for colours)

```
from thrift.transport import TSocket
from thrift.transport import TTransport
from thrift.protocol import TBinaryProtocol
from com.thrift.gen import MsgService

if __name__ == '__main__':
    transport = TSocket.TSocket('localhost', 8899)
    transport = TTransport.TBufferedTransport(transport)
    protocol = TBinaryProtocol.TBinaryProtocol(transport)
    client = MsgService.Client(protocol)

    # connect
    transport.open()

    result = onIn('Shanghai University station')
    result = onOut('Jingan Temple')

    # close
    transport.close()
```

7.3.2 *Thrift server code Server.java*

This article uses Java to build the thrift server. First, we need to create a TTransport object. Secondly, we need to create input and output protocols for the TTransport object. We must create a processor based on the input and output protocols, wait for the connection request, and give them to the processor. The final build code is as follows.

(see online version for colours)

```
public class ThriftServer {
    public static void main(String[] args) {
        try {
            System.out.println("Server is running...");
            TProcessor tprocessor = new MsgService.Processor<MsgService.Iface>(new
                  MsgServiceImpModal());
            TServerSocket serverTransport = new TServerSocket(8899);
            TServer.Args tArgs = new TServer.Args(serverTransport);
            tArgs.processor(tprocessor);
            tArgs.protocolFactory(new TBinaryProtocol.Factory());
            TServer server = new TSimpleServer(tArgs);

            server.serve();
        } catch (TTransportException e) {
            e.printStackTrace();
        }
    }
}
```

7.3.3 Route Route.xml code

The function attribute value in the tag is the name of the binding interface function, and the POI attribute value is the unique identification number of the bound POI. At runtime, the POI information will be queried through the identification number. If there are multiple POIs, we must separate the attribute values with a comma. Sites with attributes will recognise post-bound listeners through the functions in the XML manipulation tool class.

(see online version for colours)

```
<?xml version="1.0" encoding="GBK" ?>
<route>
    <origin>Shanghai University</origin>
    <destination>Xinzhuang</destination>
    <distance>45.5</distance>
    <cost>6.0</cost>
    <segs type="list">
        <seg>
            <busline>Subway Line 7(Meilan Lake--Huamu Road)</busline>
            <spots type="list">
                <spot function="in1">Nanchen Road</spot>
                <spot>Shangda Road</spot>
                <spot function="out1">Changzhong Road</spot>
                <spot>Dachang Town</spot>
                <spot>Xingzhi Road</spot>
                <spot>No.3 Dahua Road</spot>
                <spot>Xuncun Road</spot>
                <spot function="in2">Langao Road</spot>
                <spot>Zhenping Road</spot>
                <spot>Changshou Road</spot>
                <spot>Changping Road</spot>
                <spot function="out2" POI="B00155LV8J">Jingan Temple</spot>
            </spots>
        </seg>

        <seg>
            <busline>Subway Line 1(Fujing Road--Xinzhuang)</busline>
            <spots type="list">
                <spot function="in3">Hengshan Road</spot>
                <spot>Xujiahui</spot>
                <spot>Shanghai Indoor Stadium</spot>
                <spot>Caobao Road</spot>
                <spot>Shanghai South Railway Station</spot>
                <spot>Jinjiang Park</spot>
                <spot>Lianhua Road</spot>
                <spot function="out3">Waihuan Road</spot>
            </spots>
        </seg>

        <seg>
            <busline>Walk</busline>
            <spots/>
        </seg>
    </segs>
</route>
```

8 Conclusions

This paper studies APP-assisted development code generation technology for the mobile environment. Using this technology, a thrift-based visual path planning code generation system is implemented and a code generation case is detailed. The system extracts all process designs into a configuration file, which solves the problem of the high coupling of traditional MDA process logic and APP logic that does not allow them to adapt to the rapid changes of business processes. The system uses thrift to generate a code-writing framework that provides a self-programmable framework in the generated code to increase user freedom.

The extracted process logic proposed in this paper is mainly the business entity of the business logic layer and the workflow. In fact, the business process also has the possibility to change in many other aspects. The next step will focus on a more specific classification of the process logic. We will use a random process to generate different business process samples and classify the changes in the samples. Furthermore, we will summarise more diverse process logic extraction for these categories, thereby making the model more adaptable to changes in actual business needs.

Acknowledgements

This work is supported by the CERNET Innovation Project under Grant No. NGII20170513.

References

Albert, M., Cabot, J., Gómez, C. and Pelechano, V. (2010) 'Automatic generation of basic behavior schemas from uml class diagrams', *Software & Systems Modeling*, Vol. 9, No. 1, pp.47–67.

Bak, T., Sakowicz, B. and Napieralski, A. (2006) 'Development of advanced J2EE solutions based on lightweight containers on the example of 'e-department' application', in *Proceedings of the International Conference Mixed Design of Integrated Circuits and System, MIXDES 2006*, IEEE, pp.779–782.

Dong, W. (2008) 'Dynamic reconfiguration method for web service based on policy', in *2008 International Symposium on Electronic Commerce and Security*, IEEE, pp.61–65.

Gaedke, M. and Rehse, J. (2000) 'Supporting compositional reuse in component-based web engineering', in *SAC*, Citeseer, No. 2, pp.927–933.

Gomes, R.M. and Baunach, M. (2019) 'Code generation from formal models for automatic rtos portability', in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, pp.271–272..

Gusarovs, K., Nikiforova, O. and Giurca, A. (2017) 'Simplified LISP code generation from the two-hemisphere model', *Procedia Computer Science*, Vol. 104, pp.329–337.

Hu, W. and Zhang, K. (2014) 'Research and implementation of android embedded code generation method based on rule model', *International Journal of Multimedia and Ubiquitous Engineering*, Vol. 9, No. 11, pp.273–282.

Humphrey, W.S. (1992) 'Toward a discipline for software engineering', *Sei Conference on Software Engineering Education*, Springer-Verlag.

Imam, A.T., Rousan, T. and Aljawarneh, S. (2014) 'An expert code generator using rule-based and frames knowledge representation techniques', in *2014 5th International Conference on Information and Communication Systems (ICICS)*, IEEE, pp.1–6.

Jain, P. and Schmidt, D.C. (1997) 'Service configurator: a pattern for dynamic configuration of services', in *COOTS*, pp.209–220.

Kwon, Y. and Park, Y.B. (2018) 'A study on dynamic role-based user service authority control and real-time service configuration', in *2018 International Conference on Platform Technology and Service (PlatCon)*, IEEE, pp.1–6.

Liu, L., Li, Z. and Li, R. (2006) 'Improving information system flexibility through remote dynamic component configuration', in *2006 International Conference on Service Systems and Service Management*, IEEE, Vol. 1, pp.461–466.

Malik, M.Z., Nawaz, M., Mustafa, N. and Siddiqui, J.H. (2018) *Search based Code Generation for Machine Learning Programs*, arXiv preprint arXiv:1801.09373.

Pentland, B.T. (2003) 'Sequential variety in work processes', *Organization Science*, Vol. 14, No. 5, pp.528–540.

Possatto, M.A. and Lucrédio, D. (2015) 'Automatically propagating changes from reference implementations to code generation templates', *Information and Software Technology*, Vol. 67, No. C, pp.65–78.

Strnadl, C.F. (2006) 'Aligning business and it: the process-driven architecture model', *Information Systems Management*, Vol. 23, No. 4, pp.67–77.

Syriani, E., Luhunu, L. and Sahraoui, H. (2018) 'Systematic mapping study of template-based code generation', *Computer Languages, Systems & Structures*, Vol. 52, No. C, pp.43–62.

Szydlo, T., Sendorek, J. and Brzoza-Woch, R. (20118) 'Enabling machine learning on resource constrained devices by source code generation of the learned models', in *International Conference on Computational Science*, Springer, pp.682–694.

Viswanathan, S.E. and Samuel, P. (2016) 'Automatic code generation using unified modeling language activity and sequence models', *IET Software*, Vol. 10, No. 6, pp.164–172.

Wang, J., Peng, Q. and Hu, X. (2014) 'A modeling: internetware-based dynamic architecture evolution applying to SOA', in *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, IEEE, pp.100–105.

Wikipedia (2013) *Business Process Customization* [online] https://en.wikipedia.org/wiki/Business_process_customization (accessed 2019).

Yavuz, E., Turner, J. and Nowotny, T. (2016) 'GeNN: a code generation framework for accelerated brain simulations', *Scientific Reports*, Vol. 6, p.18854.