The object of research is the implementation of relations between software classes. It is shown that when implementing the aggregation relationship between classes, errors may occur if more than one client class is found. Class interaction errors can be caused by management of resource class attributes by one of the client classes in a way that is unacceptable to another client class due to invalid attribute values, state changes, method blocking, etc. To solve the problem, a special organization of the queue for client classes is proposed. A feature of the queue is the use of models of client classes and resource class. The model of a resource class provides an idea about its resources (attributes and methods) and how they are used. The client class model shows how much of these resources will be used by the client and how this will be done. This organization of the queue makes it possible to provide resources to the next client class only after checking its compatibility with active client classes. In general, client classes have different types, and this complicates the organization of the queue. Therefore, it is proposed to make them derived from the base class, which defines the interface for the queue. Similarly, the problem of the interaction of the class-resource with the queue is solved. The proposed base class for the resource class also provides the necessary queue interface.

Software was developed that automates the process of converting classes: analysis of a resource class, determination of resource needs from client classes, construction of base classes. After the conversion is completed, the queue functions are supported. The study results verification showed a reduction in the time for converting classes by about three times, and the waiting time for access to resources during the work of the queue – at least two times

Keywords: aggregation relationship, class-client, class-resource, mathematical model, queue of class objects, class conversion, software

UDC 004.415

DOI: 10.15587/1729-4061.2024.301011

IMPLEMENTATION OF CLASS INTERACTION UNDER AGGREGATION CONDITIONS

Oleksii Kungurtsev
PhD, Professor
Department of Software Engineering
Odesa Polytechnic National University
Shevchenko ave., 1, Odessa, Ukraine, 65044
Nataliia Komleva
Corresponding author
PhD, Associate Professor
Department of Software Engineering
Odesa Polytechnic National University
Shevchenko ave., 1, Odessa, Ukraine, 65044
E-mail: komleva@op.edu.ua

Received date 16.01.2024
Accepted date 25.03.2024
Published date 30.04.2024

***How to Cite**: Kungurtsev, O., Komleva, N. (2024). Implementation of class interaction under aggregation conditions. Eastern-European Journal of Enterprise Technologies, 2 (2 (128)), 20–30. https://doi.org/10.15587/1729-4061.2024.301011*

1. Introduction

The object-oriented approach to the construction of software (SW) offers the implementation of SW in the form of a set of interacting classes [1, 2]. Interaction of classes is implemented owing to connections, the main of which are inheritance, aggregation, and composition. Aggregation and composition in programming involves the use in some class of attributes of the type of another class [3]. When introducing the concept of the main class and the resource class, the composition assumes that the main class creates the resource class and fully controls it, while in the case of aggregation, the object of the resource class is not necessarily created by the main class. The main class can receive a reference to the object of the resource class [4] and, therefore, there is no full control over its attributes. In this case, there is a need to share resources, that is, a transition to the task of mass service [5] and the need to organize a queue for the main classes, which, according to their new role, can be termed client classes.

For a better idea of the possible situation as a resource class, consider a cruise ship that arrived at an intermediate port. Excursions are provided for passengers. Let the travel company T1 (client class) be the first in line. T1 organizes tours to the city's museums. Then T1 selects all willing cruise ship passengers. Another tourist company T2 (also a client class) organizes tours around the city. If T2 is second in line, it cannot become active until T1 is disabled. Let the company TP1 (class-client) supply fuel to the ship and stand third in line. Since TP1 is not using the vessel resources used by T1, it can become active. Similarly, firm PR1 (customer class), supplying products and occupying the fourth place in the queue, can also become active.

Fig. 1 shows the interaction scheme of client classes with resources of the resource class. Let at time t_1 Class-client1 called Method1 of class-resource, which uses Attribute1 and Attribute2. Next, at time $t_2>t_1$, Client Class 2 called Method 2 of the resource class, which modifies Attribute 1. After that, at time t3>t2, Client Class 1 called Method 3, which uses Attribute 1 and Attribute 3. As you can see, when trying to use Attribute 1, a conflict (error) occurs, because the value of this attribute was changed by Client Class 2, which was not known to Client Class 1.

The resource approach is a management method in which decisions about the type and features of the system are made on the basis of available resources [6]. When defining a queue, the resource approach considers a class-resource, which contains an informational part (information about its attributes and the state of the object) and a functional part (the possibility of providing methods at the disposal of the class-client). customers are not decided. First of all, one needs to clarify the concept of "getting the resource at the disposal of the client". Secondly, the concept of "client" needs to be clarified, since different clients may need different resources.

It follows from the above that there is an actual task of implementing class aggregation, both from the point of view of determining the capabilities of the class-resource, and from the point of view of determining the necessary resources and the conditions for obtaining them by the class-client.

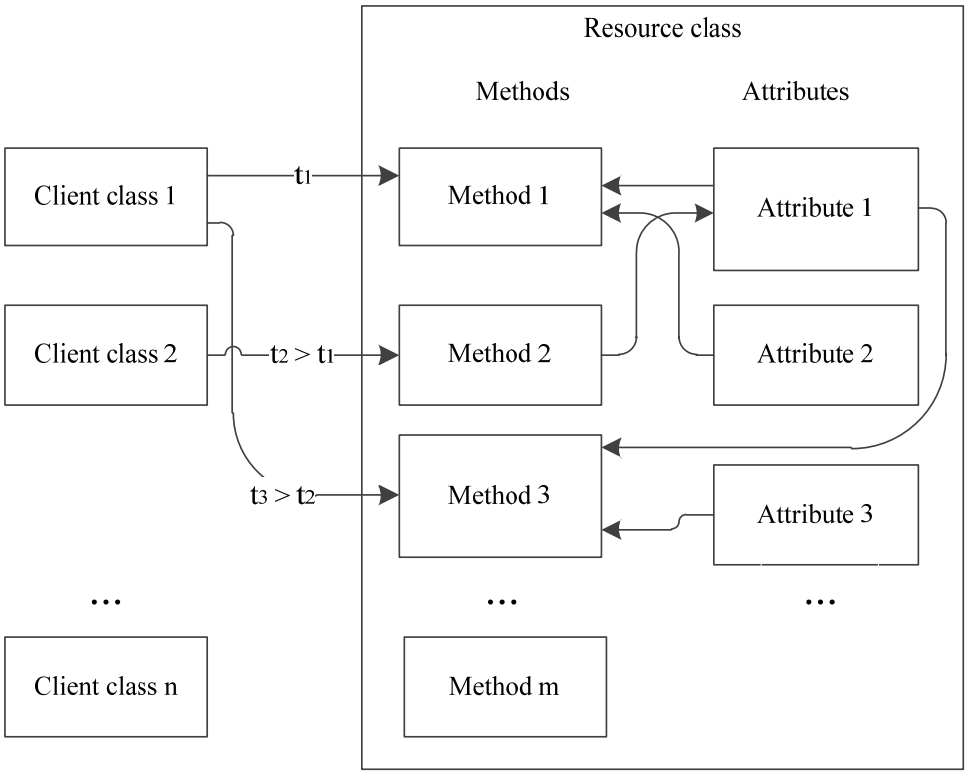

Fig. 1. Resource sharing conflicts

2. Literature review and problem statement

Work [4] deals with the issue of identification and analysis of the effectiveness of class attributes. However, the research is only concerned with attributes whose selection is consistent with the purpose of creating the class, while with "re-use" the purpose of applying the class may be slightly altered.

Paper [7] revealed the problem of unbalanced use of classes when implementing aggregation. However, a similar problem occurs not only at the level of aggregated classes but also at the level of attributes and methods of the same class.

Work [8] shows the comparative effectiveness of aggregation when during development there is a need to add data and functions to the class that do not correspond to its initial concept. The considered problem to a greater extent concerns the choice of the resource class, and not the interaction of classes during the implementation of aggregation.

Paper [9] outlines the benefits of aggregation and demonstrates that poor design of aggregation as part of development leads to significant problems in the design process of the entire software. However, the authors do not consider the problem of class-resource sharing.

A well-known approach to creating and improving the efficiency of a mass service system is to determine the impact of queues on various aspects of the organization in which they are implemented.

Paper [10] examines the role of mass service theory in the banking sector and the probability of the influence of queues on various types of banking activity. The authors conducted a number of studies with congested queues in some selected banks but did not make specific recommendations for solving the problem and announced the continuation of the study.

Study [11] examines the contribution and application of mass service theory in the applied field of health care management. This review proposes a system of classification of health care areas for which mass service models are being built. The authors describe major trends in the application of mass service theory and models available to health care decision makers, but the references end in 2011 and should not be considered definitive.

Paper [12] shows that mass service models are an attractive and highly efficient alternative for quantifying the efficiency of traffic flows compared to simulation. The authors reported in-depth studies on the analysis of non-stationary queues with non-stationary application receipt processes using an approximation approach. However, a limitation of the study is that traffic flows have their own unique characteristics, and the study cannot be projected onto an arbitrary subject area.

In works [13, 14], a classic model of a mass service system with one server is considered. It independently handles incoming requests, which does not make it possible to achieve the efficiency of providing services of a clinical hospital. This leads to patients waiting too long in the queue for medical care and potentially many irreversible problems. Parallel processing of requests by several servers would make it possible to distribute the load more evenly and speed up customer service. However, in the case of interaction of classes, it should be taken into account that not all operations can be performed in parallel (for example, initial diagnosis and discharge of patients). At the same time, operations that are not related to changing attributes can be parallelized.

Consider the possibility of using known types of queues to implement aggregation. Paper [15] emphasizes the importance of analyzing several related processes, including queue

entry, waiting in a queue (in effect, a storage process) and service by a server (or servers) at the beginning of the queue. However, the authors consider only the same type of queue elements, while the served customers can be completely different in nature and structure.

In addition to the classic FIFO (First-In-First-Out) model, a queue with priorities is often used. In this type of queue, each item is assigned a specific priority, and items are processed in order of decreasing priority. This allows for more important requests or customers with the highest priority to be served first. In [16] it is proposed to manage the arrival of client requests based on the superposition of two processes: independent and correlated, but the authors do not consider the features of the resources used by clients.

The authors of [17] developed an optimized version of the priority queue customer service algorithm by combining parallel operations, which significantly increases performance. However, the study is limited to only client requests that are a priori ready to be served. At the same time, in the problem considered in this paper, it is necessary to check this readiness.

Paper [18] presents a practical approach to estimating the waiting time for the service of several classes in a queue with priorities. The disadvantage of the study is that the proposed approach can be applied only to certain types of emergency departments when serving patients with different levels of severity of the problem.

Selective Service Queue allows the system to select certain requests for service depending on their characteristics or priority. In paper [19], the authors proposed a mechanism for servicing only requests of a certain type. However, the subject area of research is limited to industrial control systems and has no recommendations for other areas. But if, for the problem being studied, the resources needed for the client class are considered as an informational component of the request, then the model of a queue with selective service can be considered as a prototype of the solution to the problem of class interaction.

A limited-capacity queue makes it possible to limit the number of elements that can be contained in it. When the queue reaches its maximum capacity, new items can be rejected or processed in a special way. Paper [20] shows that bounded queues are used to manage the load and prevent overflow in systems where resources are limited. An area with a fixed maximum number of waiting customers is usually easier to control than one where customers are allowed to come and join a line of arbitrary length. However, the paper does not address the issue of the possibility of preliminary assessment of the successful service of a potential client, provided that the current filling of the queue and the service time of active customers in it are known.

The authors of paper [21] showed that a queue can include several links with different throughput. However, the question of the nature of the capacity of the queue has not been investigated. Within the framework of the problem under study, the throughput of the queue may depend on the degree of communication of client classes and the resources they require.

To increase system reliability and ensure uninterrupted service, one can use a queue with backup channels (Backup Queue). In such queues, requests may be routed to a backup channel or otherwise handled if the primary channel is unavailable or congested. In paper [22], it is proposed to use a backup server in the case of absence (due to staff vacations, technical breakdowns, or other reasons) of the main server. However, the authors do not evaluate the cost of implementing a backup server and the feasibility of its use.

In work [23], it is proposed to distribute customers to the main and reserve service channels in order to maximize the overall performance indicator. However, the work does not provide specific recommendations regarding the number and possible loads of backup channels when serving customers.

The authors of work [24] proposed an approach to resource sharing using the example of virtualization of network functions and a shared protection mechanism, which allows several functions to share backup resources. However, the paper did not sufficiently consider the potential risks and limitations of the proposed approach, as well as ways to minimize or overcome them.

Depending on the type and characteristics of queues, various risks may arise that affect the quality of service and customer satisfaction. In [25, 26], the emphasis is on the risks associated with connecting to long queues of customers sensitive to delays, as well as on the uneven distribution of the load in such queues. However, in these works there is no formalization of the conditions for the occurrence of risks, which creates gaps in providing a full-fledged analysis of potential dangers and limitations that may arise when working with queues.

Our analysis of known solutions revealed that the interaction of classes under the conditions of aggregation has not been sufficiently studied. The queue type, as a means of organizing such interaction, should be largely determined by the resource class that provides the means for sharing. However, the client class also needs a special definition from the point of view of the nomenclature and the nature of resource use.

3. The aim and objectives of the study

The purpose of our study is to build a mechanism for the interaction of client classes with a resource class, which could reduce the time of waiting in a queue and the time of conversion of classes to implement the aggregation relationship.

To achieve the goal, the following tasks are to be solved:

– to determine the features of the queue for the implementation of aggregation;

– tobuild a class-resource model;

– toconstruct a model of the class-client;

– to devise a method for providing services to client classes under conditions of aggregation;

– to develop software for the implementation of the method for providing services to client classes.

4. The study materials and methods

The object of research is the implementation of relations between software classes. The subject of our study is the model and method for implementing the aggregation relationship between software classes. The main hypothesis of the study assumes that when two or more customer classes interact with one resource class, conflicts may arise, to eliminate which it is necessary to define compatible customer classes. Research methods in the work are:

– the basics of the theory of mass service for determining the properties of the queue of client classes;

– methods of object-oriented and syntactic analysis to determine the elements of classes and their interaction;

– provisions of set theory for building resource class and client class models.

To implement the method for providing services to classes-clients, the Class Aggregation Facilitator software system was developed in the Python programming language. At the same time, the PyCharm 2022.3.2 integrated development environment from the JetBrains company and a modern basic computer that supports PyCharm were used.

5. Research results on the construction of a mechanism for the interaction of client classes with a resource class in the context of aggregation

5. 1. Organization of the queue

Queue requirements are determined by the fact that both the client and the resource are classes and must interact according to the rules of use and interaction of classes. Based on this provision, the following properties of the queue are formulated:

– access to work with the object is guaranteed to the client who occupies the first position. Queuing clients can access the object if their activity does not interfere with previously connected clients;

– the client can leave the queue at his/her discretion;

– for the possible automation of processes, customers can be allowed to order the time of working with the resource (moment of connection, minimum and/or maximum time of use, maximum waiting time for starting work with the resource, etc.).

The peculiarities of the queue for the implementation of aggregation are considered. Table 1 gives in a formalized form the main actions of the person who makes the decision to organize the queue according to certain requirements for providing services to customer classes and potential risks.

Fig. 2 offers a working scheme of the Queue class. The Client communicates with the Resource through the Queue. In order to connect to the Queue, the Client must transfer his/her model to it. The presence of the Customer model indicates that the Customer is in the Queue.

The queue can transfer the Client to the "active" state. Then the Client methods that use the Resource start working.

The following options are possible to terminate the Client's work with the Resource:

– The client transmits a message about disconnection from the Queue;

– the t_r time ordered by the Client has expired (setting t_r is optional);

– the maximum t_rMax time allotted for work with one Client has expired (setting t_rMax is also optional).

Table 1

Managing queue features to implement aggregation

Requirement	Action	Risk
Setting the queue type	setType (queue, type), type={FIFO, SelectiveServiceQueue, QueuewithMarkingofUrgentRequests, ...}	if <queue type is incorrect> then <long service waiting times> and/or <data loss> and/or <complexity of reengineering client classes and resources>
Setting the queue size	setMaxLen (queue, maxLength)	if <peak load> then <client class denied service>
Organization of the reserve queue	setReserve (queue, reserveQueue)	if <peak load> then <service start time unknown to client class>
Setting a threshold value for waiting time in the queue	setMaxWaiting (queue, maxTimeWaiting)	if <maxTimeWaiting is too large> then <the system reacts late to load changes> if <maxTimeWaiting is too small> then <generation of false positives of abnormal load messages>
Setting a threshold value for the duration of ownership of a resource for all or some client classes	setMaxHolding (queue, maxTimeHolding) setMaxClassHolding (queue, class, maxClassTimeHolding)	if <maxTimeHolding/maxClassTimeHolding is too large>then <system performance decreases> if <maxTimeHolding/maxClassTimeHolding is too small> then <class-client is not fully served>
Setting a threshold value for the length of stay for all or some customer classes in the queue	setMaxLifetime (queue, maxTimeHolding) setMaxClassLifetime (queue, class, maxClassTimeHolding)	if <maxTimeHolding/maxClassTimeHolding is too large>then <queue overflow> and/or <data obsolescence> if <maxTimeHolding/ maxClassTimeHolding is too small> then <client-class data loss>
Management of priorities	setClassPriority(queue, class, priority)	if <priority too low> then <long service waiting time> if <priority for N classes is too large and N is large> then <queue overload and service slowdown>
Configuring redirect strategies	setRedirectionClass (queueFrom, queueTo, class)	if <high-priority client-class requests are forwarded to low-priority queue> then <client-class data loss> if <client class requests are often redirected from one queue to another without real need> then <queue/queue performance degradation>

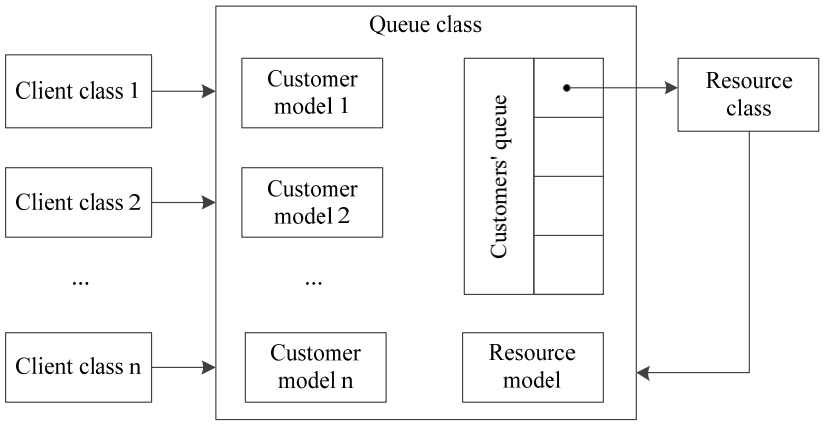

Fig. 2. Scheme of operation of the queue class

5. 2. Class-resource model

A class is usually represented by a name, attributes, and methods. From a modeling perspective, there are no special name requirements for the class used for aggregation. As for attributes, it is not enough to know their name, type, and purpose. When using attributes together, they must be assigned to the following groups:

– attributes that do not change their value after the creation of the object (stable attributes);

– attributes whose values cannot block the execution of methods (non-controlling attribute);

– other attributes (other attributes).

Representing class methods also requires additional classification:

– methods that do not change attribute values;

– methods that change attribute values, which, in turn, can block the execution of class methods;

– other methods.

Based on the above requirements, the resource class is represented by a tuple:

$$c = \langle cHead, mMeth, mAttr \rangle, \quad (1)$$

where *cHead* – class header,

mMeth – the set of functions (methods) of the class,

mAttr – the set of attributes of a class.

The class header is also represented as a tuple:

$$cHead = \langle cName, cBaseName \rangle, \quad (2)$$

where *cName* – class name;

cBaseName – the name of the parent class for *cName*(can be empty).

Each attribute from the *mAttr* set is represented as:

$$Attr = \langle attrName, attrType, attrCateg \rangle, \quad (3)$$

where *attrName* – attribute identifier;

attrType – attribute type;

attrCateg – attribute category.

An attribute category can take the following values:

– *attrStable* – an attribute that does not change its value after the object is created;

– *attrNonControl* – an attribute whose values cannot block the execution of class methods;

– *attrOther* – other attribute.

The set of methods is represented by a tuple:

$$mMeth = \langle mFunc, mConstr, destr \rangle, \quad (4)$$

where *mFunc* – a set of ordinary class methods;

mConstr – the set of class constructors;

destr – class destructor (usage depends on the programming language).

Any element from *mFunc* looks like this:

$$mMeth_i = \langle fName_i, mArgs_i, retType_i, mOperator_i, methCategor \rangle, \quad (5)$$

where *fName* – method name;

mArgs – the set of arguments of the method;

retype – the type of return value (empty for constructors and destructor);

mOperator – the set of method operators;

methCategor – the category of the method.

The method category can take the following values:

– *methNChange* – a method that does not change the value of attributes;

– *methChange* – a method that changes the values of attributes;

– *methSetX* is a method that sets the value of a single attribute *x*;

– *methOther* – other methods.

5. 3. Client class model

A client class is any class that uses an attribute class object. The model is needed to determine which attribute class resources will be used by a particular client class. Model *Cl* is represented by a tuple:

$$Cl = \left\langle \begin{matrix} clName, cName, objName, mClAttr, \\ mClMeth, clPriority, clActive \end{matrix} \right\rangle, \quad (6)$$

where *clName* – the name of the client (can be his/her number in the queue);

cName – the name of the class that represents the client class;

objName – the name of the object with which the client works (within the queue – one object for all clients);

mClAttr is the set of attributes of the resource class that the client uses.

Each attribute is represented by a tuple:

$$attr = \left\langle \begin{array}{l} immutability, mValideValue \vee , \\ (lowerLimit, upperLimit) \vee \\ \vee mInvalidValue \vee \\ \vee AnyValue \vee mClMeth \vee , \\ clPriority \vee clChangePriority \vee \\ \vee (clActive, clDefaultActive) \end{array} \right\rangle, \quad (7)$$

where *immutability* – the requirement not to change the value of the attribute by another client (whether true or *false*):

mValideValue – the set of valid attribute values from the client's point of view;

lowerLimit, upperLimit – lower and upper bounds of attribute values;

mInvalidValue – the set of values that are unacceptable from the client's point of view;

anyValue – the client is satisfied with any value;

mClMeth – the set of methods of the *cName* class that the client wants to use;

clPriority – the priority of the client when sharing the *cName* class is set automatically when the client is added to the queue;

clChangePriority – checkbox for the possibility of changing the client's priority by an expert (one can change it – true, one can't change it – false);

clActive – checkbox of the client's activity (uses services – true, does not use – false);

clDefaultActive is the initial value of the client activity checkbox.

5. 4. The method for providing services to the client class

The *cName* client class can be in two states – working and waiting. A client is in working state if it can perform all actions according to its model and use all attributes with values acceptable to it. Otherwise, the client is in a waiting state:

Stage 1. Implementation of models of client classes and resource class.

The list of client classes and the resource class should be defined in advance. Then the resource class model is formed. Based on the resource class model, client class models are formed, and client classes are transformed. The resource class model is included in the queue class.

Stage 2. Initial formation of the queue of customers.

For queue management, the number of customers in the queue *cn* and the number of active customers in the queue *ca* are entered. Initially, $cn=0$ and $ca=0$.

When a $cName_i$ client registers in a queue, the following actions are performed:

$$cn = cn + 1,$$

$$clName_i = cn. \quad (8)$$

Stage 3. Providing services to the client.

We considered the case when the client is the first in line – $clName_i=1$. Here, two options are possible:

a) the object of the resource class has not yet been created. A class object is created. The values of the attributes included in the $mClAttr_i$ set are set according to the requirements for them (7). The client is transferred to the "active" state:

$$clActive_i = true,$$

$$ca=1; \quad (9)$$

b) the object of the resource class was created earlier. The attribute values of the resource class are compared with the corresponding requirements for the client (7). If the value of the attributes of the resource class does not meet the requirement, then the attribute is adjusted by calling the *methSetX*() method. The client is transferred to the "active" state (9).

We considered the case when the client is not the first in line – $clName_i=n>1$. The following options are provided here:

a) a set of attributes used by the client is analyzed. If the *anyValue* property is set for all attributes from the $mClAttr_i$ set (any value can be used) and all $mClMeth_i$ methods belong to the *methNChange* category (do not change attribute values), then the client is transferred to the "active" state:

$$clActive_i = true,$$

$$ca:=ca+1; \quad (10)$$

b) a set of attributes used by active clients is formed:

$$mAttr\Sigma = \bigcup_{j}^{j=1,ca} mClAttr_j. \quad (11)$$

If $mAttr\Sigma \cap mClAttr_i=\emptyset$, then the client is transferred to the "active" state and actions (10) are performed;

c) if $mAttr\Sigma \cap mClAttr_i \neq \emptyset$, then $cName_i$ client attributes used by active clients are determined:

$$mClAttr' = mAttr\Sigma \cap mClAttr_i \quad (12)$$

and checks are performed:

c1) if the $cName_i$ client for all attributes from *mClAttr'* allows any values, has not set the *immutability* requirement and its $mClMeth_i' \in mClMeth_i$ methods, which use attributes from the *mClAttr'* set, belong to the *methNChange* category, then it becomes "active" (10);

c2) if all the conditions of point B1) are met for the $cName_i$ client, except for the requirement of *immutability* for some *mClAttr''* ($mClAttr'' \in mClAttr'$) attributes, but for the same attributes there is an *immutability* requirement for active clients, then it becomes "active" (10);

c3) if a change of *mClAttr''* ($mClAttr'' \in mClAttr'$) attributes is provided for the $cName_i$ client, but there is no *immutability* requirement for the same attributes and the *anyValue* requirement is set, then it becomes "active" (10);

c4) if all checks for the $cName_i$ client failed to make it "active", and there is another client in the queue for connecting to resources, then the actions of stage 2 are repeated for this client.

Stage 4. Exit of the client from the queue.

The decision to remove the client from the queue can be made in one of the following ways:

– by the client himself/herself, and then s/he must form a request $set(clActive_i)=false$;

– by thequeue if the client has exceeded the set time limit for using resources $timeUsed(Cl_i)>timeLimit(Cl_i)$.

At the same time, two cases are possible – an active client leaves the queue, and an inactive client leaves the queue.

Let the client's number be *n*. Then, in the first case, the following actions are performed:

$$ca = ca - 1,\ cn = cn - 1, \quad (13)$$

that is, for all customers with numbers greater than n, the number is reduced by 1, and for the first inactive customer in the queue, the actions of stage 3 (providing services to the customer) are performed.

In the second case, the following actions are performed:

$$cn = cn - 1, \quad (14)$$

that is, for all customers with numbers greater than n, the number is reduced by 1.

5. 5. Development of software for the implementation of the method for providing services to client classes

When developing software for conducting experimental research, the following circumstances were taken into account:

– elements of the customer queue class are objects of different classes;

– objects of client classes interact with the queue through a limited set of messages that solve the same tasks;

– the implementation of the queue depends on the resource class; however, the same tasks are solved here, regardless of the specific class.

Our analysis of these factors led to the decision to implement the class structure shown in Fig. 3.

The BasicClient base class contains virtual methods for interacting with the queue. The registration(...) method is used to register a client in a queue, which involves creating a client model and fixing its position in the queue. The setAccess(...) method is intended to grant a client access to a resource based on the created client module and the current state of the resource. The shutdown(...) method is responsible for exiting the client from the queue.

In real client classes Client1, Client2, etc. a reference to inheritance is made. In these classes, virtual methods registration(...), setAccess(...), and shutdown(...) are specified.

A queue is also represented by a class hierarchy. The BasicQueue base class defines virtual methods for interacting with clients and the resource. Methods are specified in generated classes for a specific resource object and client object.

The proposed class structure makes it possible:

– to formally create a queue for elements of the same type of the basic class BasicClient;

– to define in advance the set and partly the algorithm of the necessary methods for the client and queue classes.

Fig. 4 shows the service provision technology for the ClientsCollection client class collection.

The operation of forming a client involves creating a descendant class Client[i] from the input class located in the ClientCollection and the basic class BasicClient.

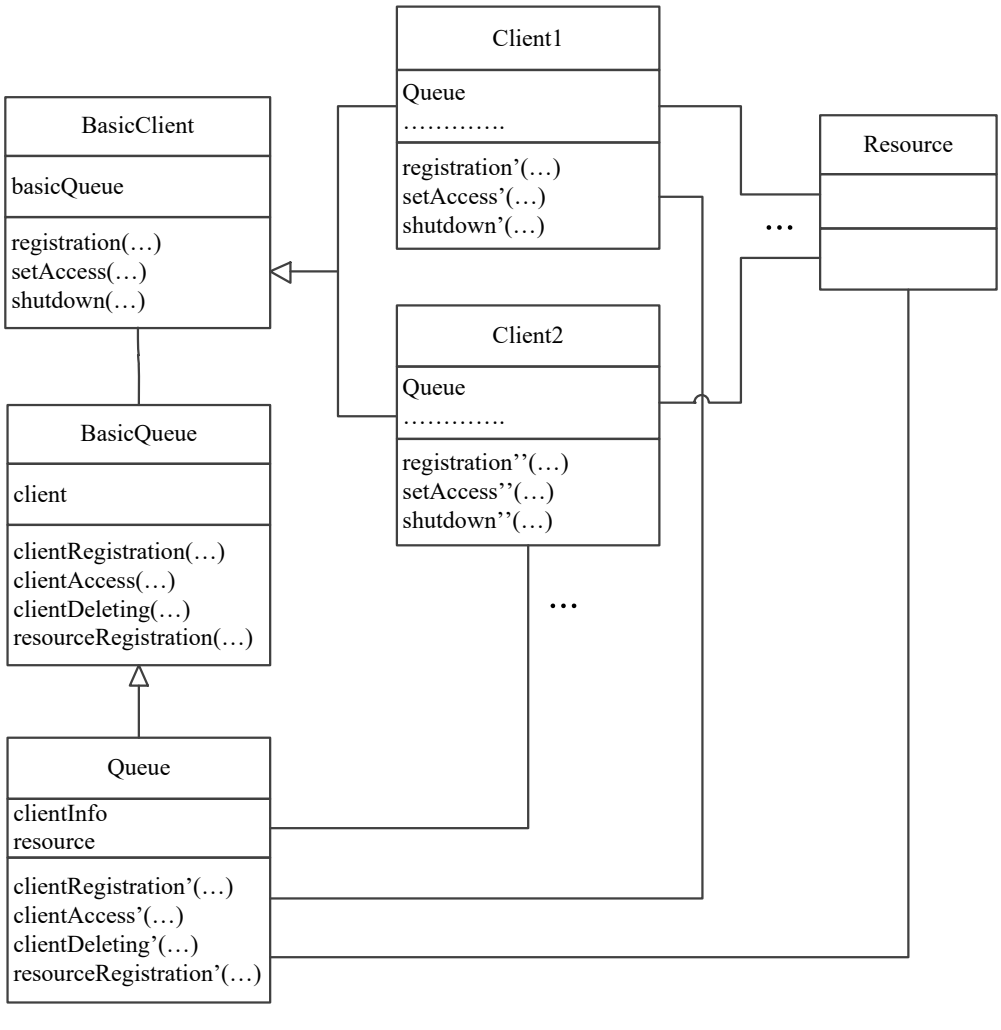

Fig. 3. Structure of program classes for software implementation

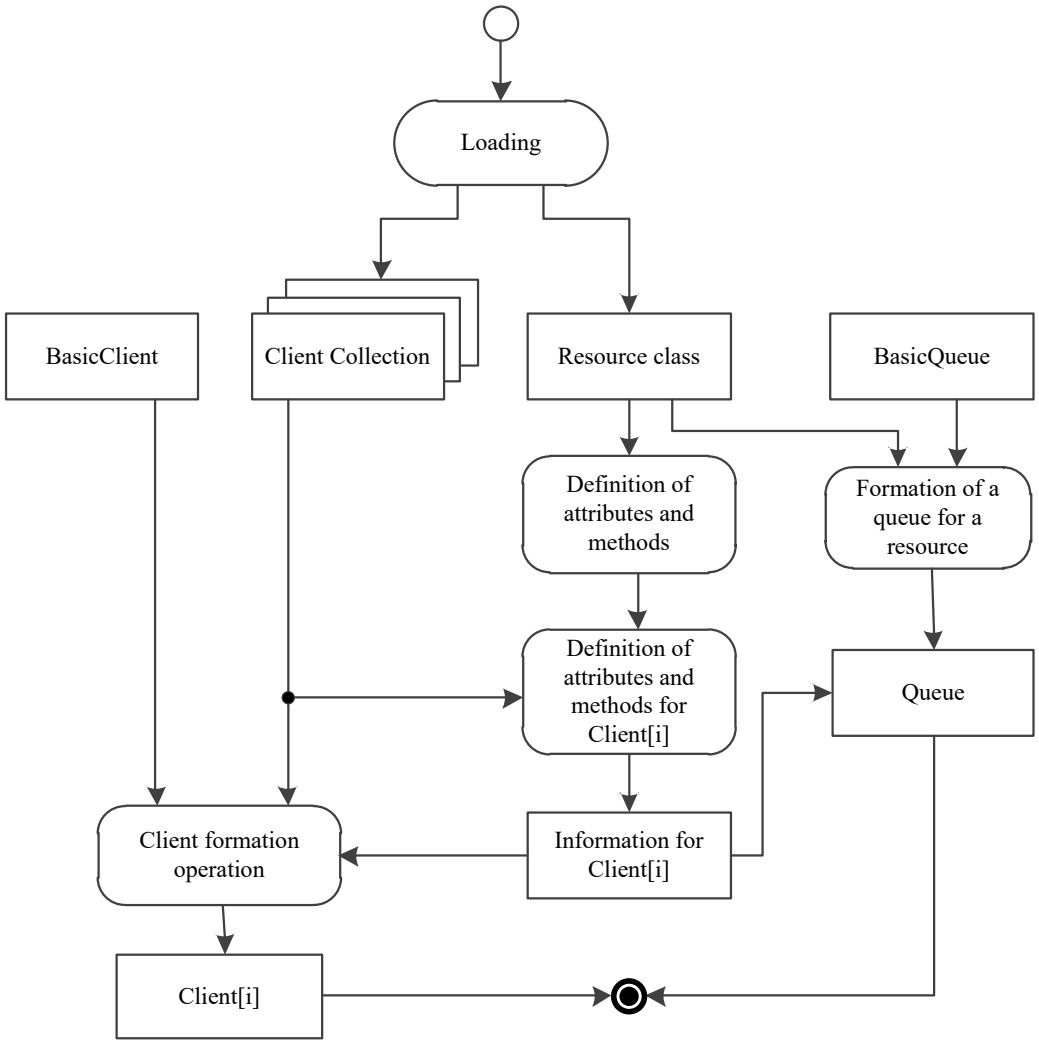

Fig. 4. Technology of preparing classes for the use of aggregation

Analysis of the resource class makes it possible to identify its attributes and methods. For each client class, the necessary attributes, methods, and conditions of their use are defined. The received information is passed to the client class and the queue.

The Class Aggregation Facilitator software system was developed, which implements the proposed technology.

The software form for loading client classes is an interface that makes it possible to select a programming language and client classes from the list (Fig. 5). One can view the code for the selected class. The "Class parsing" option is intended for highlighting attributes and methods. The "Load new client class" option makes it possible to load new client classes into the system.

In Fig. 6 the program form for working with the resource class is given. The form contains a list of resource classes present in the system, from which the user can choose the desired one. The results of parsing attributes and methods are displayed for the selected class. The user can select the required methods and attributes, as well as specify requirements for the characteristics of the attributes.

In the course of working with the Class Aggregation Facilitator system, 17 potential customer classes and 6 potential resource classes were downloaded. Experiments were conducted to prepare 10 of these classes for the implementation of aggregation under "manual mode" and automated using the developed software. The time required to transform a resource class and build a queue largely depends on the complexity of the resource class, which in turn depends on the number of attributes. Small classes with 7–10 attributes were used for the experiments. For such classes, the preparation time under the manual mode was an average of 4.8 hours, and under the automated mode – 1.8 hours. Thus, the use of Class Aggregation Facilitator reduced the time compared to the manual mode by 2.7 times. When using more complex classes, one should expect much better results.

Experiments have been conducted on sharing 4 client classes for a common resource class. No errors were detected in the operation of Class Aggregation Facilitator.

To evaluate the effectiveness of the developed method, we shall evaluate the access time to resources for two modes of operation: when the aggregation is organized in a traditional way (a simple queue) and when using the proposed queue.

Let there be n customers who need time t_1, t_2,..., t_n for work. Then, in the case of using a normal queue, the waiting time for access to the resource for the n-client will be $T^0 = \sum_1^{n-1} t_i$. In the case of using the proposed queue, the resource can be accessed by K clients at the same time. If we assume that the average number of clients using the resource at the same time is K_s, then the waiting time for an n-client is $T^1 = T^0/K_s$.

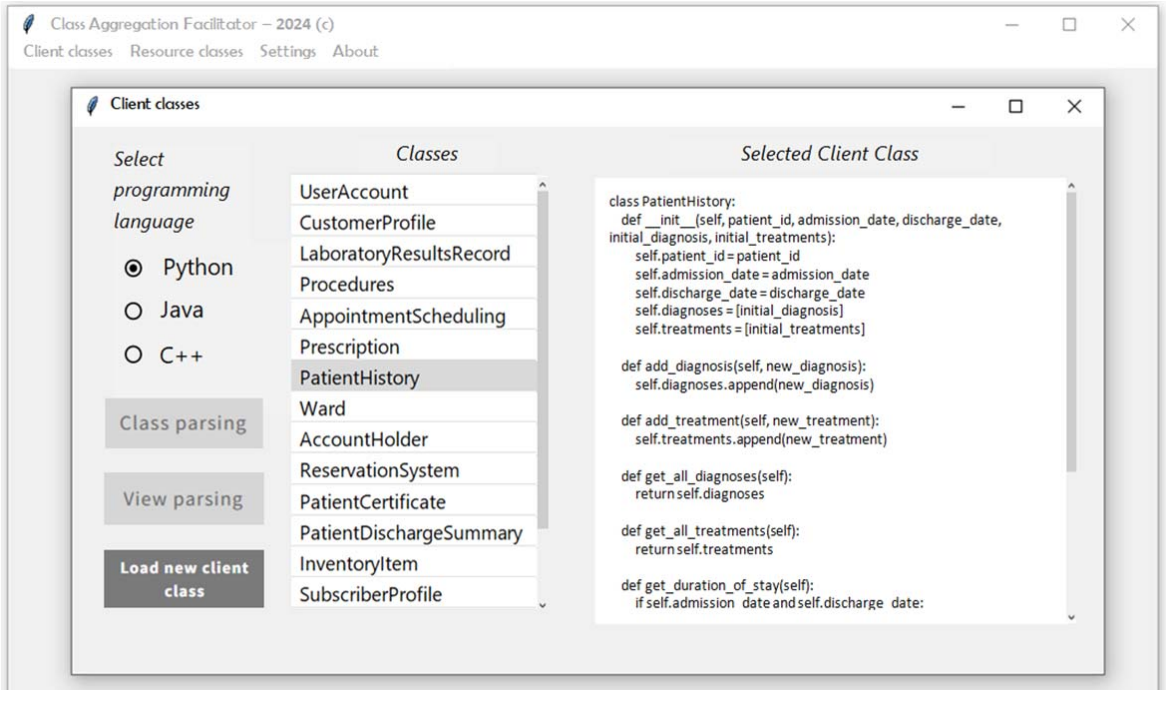

Fig. 5. A software form for uploading client classes

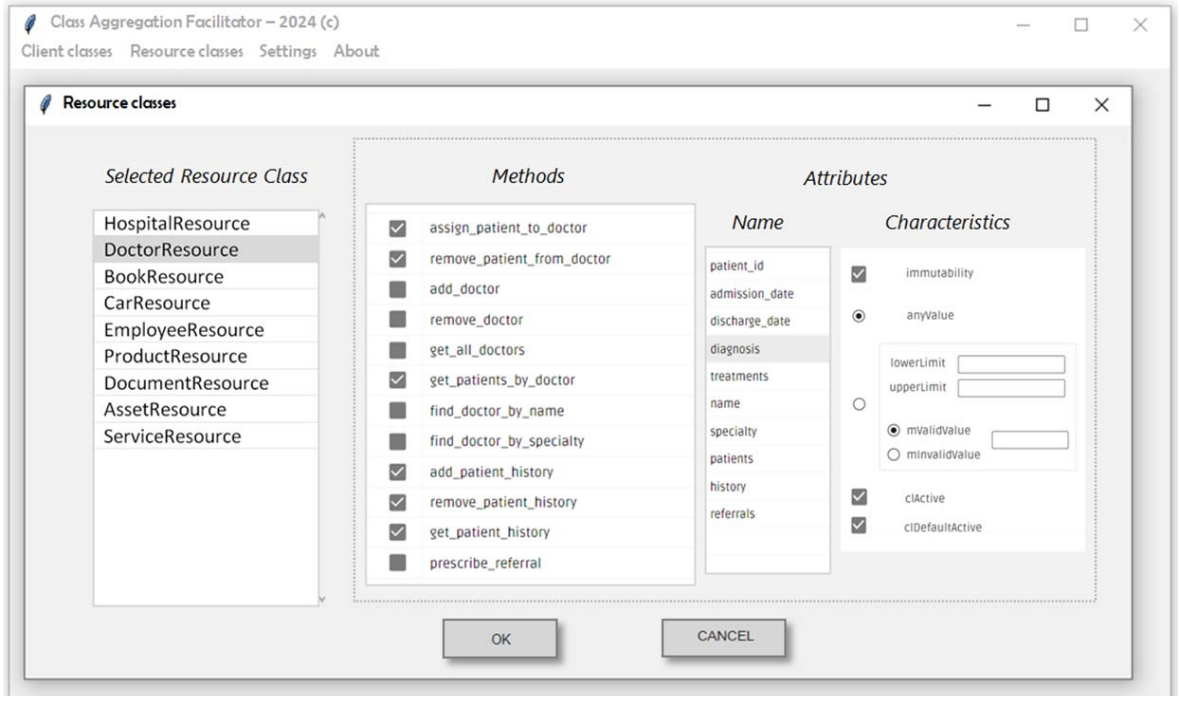

Fig. 6. A software form for working with a resource class

Without a specific resource class and client classes, it is impossible to give an accurate estimate of K_s. But even with the simultaneous operation of two client classes, the resource waiting time is halved.

6. Discussion of results of investigating the mechanism of interaction of classes under the conditions of aggregation

The proposed organization of the queue (Fig. 2) takes into account the peculiarities of the aggregation relationship between software classes. A peculiarity of the queue, in contrast to known solutions [4, 7–9], is the use of the model of the resource class and customer classes. This makes it possible to manage the provision of services to client classes in the absence of conflicts.

The proposed model of the resource class (1) makes it possible to determine its capabilities from the point of view of the expected needs of client classes. A feature of the model, in contrast to known solutions [2], is the introduction of the classification of attributes and methods from the point of view of their interaction when performing a certain func-

tionality. The model is the basis for the compatible use of class resources.

The client-class model (6) makes it possible to determine what part of the resources and under what conditions the client-class is going to use. The peculiarity of this model, in contrast to known solutions [3], is the formulation of the conditions for using resources of another class. Having a class-client model, the queue has the ability to define the conditions for providing services to the client without disrupting the work of already active clients. This makes it possible to provide resources to clients "over the head" of clients who were earlier in the queue, which can significantly increase the efficiency of using the resource class.

The method of providing services to the client class offers a comprehensive sequence of actions for preparing and implementing class aggregation. The method determines the state of the client class (11), conditions for client activation, organizes the client's exit from the queue. Unlike known solutions for queue management [20, 21, 24], the method involves not only management but also preliminary transformation of classes.

The developed software solves two problems. First, it makes it possible to automate the process of building class models and transforming classes. Fig. 3 shows the proposed class structure, and Fig. 4 – technology of their transformation Here you can estimate a certain gain in time compared to "manual" operations.

Second, the software implements the queue. Here, the gain in time completely depends on the specific class-resource and classes-clients. But this gain will be in any case when there are two or more compatible client classes.

This study did not focus on any specific object-oriented programming language. The used principles of encapsulation, imitation, and polymorphism are inherent in all developed languages of this type.

The limitations of the study include the lack of a class transformation mechanism under conditions where the resource class or the client class is generated by a class, and the programming language does not allow multiple inheritance. This problem is planned to be solved in the future.

The work does not define the conditions and circumstances of creating an instance of the resource class. Various procedures are possible here, which are planned to be developed in the future.

The work did not consider the execution time of the class-client order. Further research is expected to provide a clearer picture of the performance of class-resource methods, as well as the possibility of their parallel execution.

7. Conclusions

1. It was established that a queue of client class objects under aggregation conditions can provide services to several classes at the same time. To eliminate conflicts between clients, models of the resource class and client classes are included in the queue. Each time before providing services to a client class, its compatibility with active classes is checked. This makes it possible to reduce the waiting time of the resource by half, even with the simultaneous operation of two client classes.

2. A class-resource model has been developed, which, unlike existing class models, provides detailed information about attributes and methods from the point of view of the possibility of their use by other classes. The model makes it possible to define the ability of a resource class to provide services to a certain client class within the queue.

3. A model of the client class has been built, which, unlike existing class models, shows what services this class needs from the resource class (the methods, attributes, and their values involved). The model makes it possible to determine, within the framework of the queue, whether the resource class has the ability to provide services to a certain client class.

4. A method for providing services to classes-clients under conditions of aggregation has been devised. The method provides a complete sequence of actions for the organization and use of the queue, starting with the formation of models of the resource class, client classes, and the queue to the execution of client registration operations, connecting it to the resource, and removing the client from the queue. Unlike existing technologies, the method allows automating the processes of class conversion and queue management.

5. The Class Aggregation Facilitator software has been developed, which makes it possible to automate the analysis of the resource class, determine the desired services of client classes, create a queue, and carry out the transformation of the resource class and client classes. The use of the developed software showed a 2.7-fold reduction in class conversion time, and a 2.7-fold reduction in waiting time for access to resources during queue operation, compared to existing technologies.

Conflicts of interest

The authors declare that they have no conflicts of interest in relation to the current study, including financial, personal, authorship, or any other, that could affect the study and the results reported in this paper.

Funding

The study was conducted without financial support.

Data availability

All data are available in the main text of the manuscript.

Use of artificial intelligence

The authors confirm that they did not use artificial intelligence technologies when creating the current work.

References

1. Bontchev, B., Milanova, E. (2020). On the Usability of Object-Oriented Design Patterns for a Better Software Quality. Cybernetics and Information Technologies, 20 (4), 36–54. https://doi.org/10.2478/cait-2020-0046

2. Kungurtsev, O., Novikova, N., Reshetnyak, M., Cherepinina, Y., Gromaszek, K., Jarykbassov, D. (2019). Method for defining conceptual classes in the description of use cases. Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2019. https://doi.org/10.1117/12.2537070
3. Kungurtsev, O. B., Novikova, N. O. (2020). Identification of class models imperfection. Herald of Advanced Information Technology, 3 (2), 13–22. https://doi.org/10.15276/hait.02.2020.1
4. Rashidi, H., Parand, F. A. (2019). On Attributes of Objects in Object-Oriented Software Analysis. International Journal of Industrial Engineering & Production Research, 30 (3), 341–352. https://doi.org/10.22068/ijiepr.30.3.341
5. Ürler, Ü., Berk, E. (2016). Queueing Theory. Decision Sciences, 287–348. https://doi.org/10.1201/9781315183176-7
6. Komleva, N., Liubchenko, V., Zinovatna, S. (2020). Improvement of teaching quality in the view of a resource-based approach. CEUR Workshop Proceedings, 2740, 262–277. Available at: http://ceur-ws.org/Vol-2740/20200262.pdf
7. Pang, X., Wang, Z., He, Z., Sun, P., Luo, M., Ren, J., Ren, K. (2023). Towards Class-Balanced Privacy Preserving Heterogeneous Model Aggregation. IEEE Transactions on Dependable and Secure Computing, 20 (3), 2421–2432. https://doi.org/10.1109/tdsc.2022.3183170
8. Otu, G. A., Usman, S. A., Ugbe, R. U., Iheagwara, S. E., Okafor, A. C., Okonkwo, F. I. et al. (2023). Comparative analysis of aggregation and inheritance strategies in incremental program development. Fudma Journal Of Sciences, 7 (2), 57–64. https://doi.org/10.33003/fjs-2023-0702-1710
9. Zhang, S. G. (2021). An In-Depth Understanding of Aggregation in Domain-Driven Design. Available at: https://www.alibabacloud.com/blog/an-in-depth-understanding-of-aggregation-in-domain-driven-design_598034
10. Afolalu, S. A., Babaremu, K. O., Ongbali, S. O., Abioye, A. A., Abdulkareem, A., Adejuyigbe, S. B. (2019). Overview Impact Of Application Of Queuing Theory Model On Productivity Performance In A Banking Sector. Journal of Physics: Conference Series, 1378 (3), 032033. https://doi.org/10.1088/1742-6596/1378/3/032033
11. Lakshmi, C., Appa Iyer, S. (2013). Application of queueing theory in health care: A literature review. Operations Research for Health Care, 2 (1-2), 25–39. https://doi.org/10.1016/j.orhc.2013.03.002
12. Wang, N., Roongnat, C., Rosenberger, J. M., Menon, P. K., Subbarao, K., Sengupta, P., Tandale, M. D. (2018). Study of time-dependent queuing models of the national airspace system. Computers & Industrial Engineering, 117, 108–120. https://doi.org/10.1016/j.cie.2018.01.014
13. Adeniran, Dr. A., Sani Burodo, M., Suleiman, Dr. S. (2022). Application of Queuing Theory and Management of Waiting Time Using Multiple Server Model: Empirical Evidence From Ahmadu Bello University Teaching Hospital, Zaria, Kaduna State, Nigeria. International Journal of Scientific and Management Research, 05 (04), 159–174. https://doi.org/10.37502/ijsmr.2022.5412
14. Nor, A. H. A., Binti, N. S. H. (2018). Application of Queuing Theory Model and Simulation to Patient Flow at the Outpatient Department. Proceedings of the International Conference on Industrial Engineering and Operations Management Bandung, 3016–3028. Available at: https://ieomsociety.org/ieom2018/papers/694.pdf
15. Kumar, R. (2020). Queueing system. Chap. 4. Modeling and Simulation Concepts. Available at: https://www.researchgate.net/publication/346721926_Book_Chapter_-_queueing_system
16. De Clercq, S., Walraevens, J. (2020). Delay analysis of a two-class priority queue with external arrivals and correlated arrivals from another node. Annals of Operations Research, 293 (1), 57–72. https://doi.org/10.1007/s10479-020-03548-1
17. Walulya, I., Chatterjee, B., Datta, A. K., Niyolia, R., Tsigas, P. (2018). Concurrent Lock-Free Unbounded Priority Queue with Mutable Priorities. Stabilization, Safety, and Security of Distributed Systems, 365–380. https://doi.org/10.1007/978-3-030-03232-6_24
18. Hou, J., Zhao, X. (2019). Using a priority queuing approach to improve emergency department performance. Journal of Management Analytics, 7 (1), 28–43. https://doi.org/10.1080/23270012.2019.1691945
19. Ferrari, P., Sisinni, E., Saifullah, A., Machado, R. C. S., De Sa, A. O., Felser, M. (2020). Work-in-Progress: Compromising Security of Real-time Ethernet Devices by means of Selective Queue Saturation Attack. 2020 16th IEEE International Conference on Factory Communication Systems (WFCS). https://doi.org/10.1109/wfcs47810.2020.9114505
20. Hernandez-Gonzalez, S., Hernandez Ripalda, M. (2018). Systems With Limited Capacity. IGI Global, 172–211. https://doi.org/10.4018/978-1-5225-5264-2.ch006
21. Larrain, H., Muñoz, J. C. (2020). The danger zone of express services: When increasing frequencies can deteriorate the level of service. Transportation Research Part C: Emerging Technologies, 113, 213–227. https://doi.org/10.1016/j.trc.2019.05.013
22. Chakravarthy, S. R., Shruti, Kulshrestha, R. (2020). A queueing model with server breakdowns, repairs, vacations, and backup server. Operations Research Perspectives, 7, 100131. https://doi.org/10.1016/j.orp.2019.100131
23. Ahmadi-Javid, A., Hoseinpour, P. (2019). Service system design for managing interruption risks: A backup-service risk-mitigation strategy. European Journal of Operational Research, 274 (2), 417–431. https://doi.org/10.1016/j.ejor.2018.03.028
24. He, F., Oki, E. (2021). Unavailability-Aware Shared Virtual Backup Allocation for Middleboxes: A Queueing Approach. IEEE Transactions on Network and Service Management, 18 (2), 2388–2404. https://doi.org/10.1109/tnsm.2020.3026218
25. Sunar, N., Tu, Y., Ziya, S. (2021). Pooled vs. Dedicated Queues when Customers Are Delay-Sensitive. Management Science, 67 (6), 3785–3802. https://doi.org/10.1287/mnsc.2020.3663
26. He, B., Li, T. Z. (2021). An Offloading Scheduling Strategy with Minimized Power Overhead for Internet of Vehicles Based on Mobile Edge Computing. Journal of Information Processing Systems, 17 (3), 489–504. https://doi.org/10.3745/JIPS.01.0077