UDC 519.85:004.42

AN ARCHITECTURAL APPROACH FOR QUALITY IMPROVING OF ANDROID APPLICATIONS DEVELOPMENT WHICH IMPLEMENTED TO COMMUNICATION APPLICATION FOR MECHATRONICS ROBOT LABORATORY ONAFT

V. Makarenko[1], O.Olshevska[2], Yu. Kornienko[3]

[1,2,3]Odessa National Academy of Food Technologies, Odessa, Ukraine
ORCID: [2]0000-0002-4512-3915, [3]0000-0002-3806-4494
Scopus ID: [2]57192687506, [3]6701449522
E-mail: [1]makarenkovaleria3@gmail.com, [2]olshevska.olga@gmail.com, [3]yurikkorn@gmail.com

DOI: 10.15673/atbp.v9i3.714

Abstract: Developing a proper system architecture is a critical factor for the success of the project. After the analysis phase is complete, system design begins. For an effective solution developing it is very important that it will be flexible and scalable. During the system design, its component composition and development tools are determined. The system design phase is an opportunity to maximize the speed and effectiveness of subsequent development.

There are quite a lot of architectural approaches for building systems. Despite their small differences, they have much in common. They all define ways of splitting the application into separate layers. At the same time, in each system, at least, there is a layer containing the business logic of the application, a layer of data interaction and a layer for displaying data.

The "Clean Architecture" approach has been analyzed and adapted to the communication application for mechatronics robot laboratory developing. This approach allows to solve all the problems while building the application architecture: it makes the code modular, tested and easily readable, and also positively affects the quality of development.

New architectural components which was introduced by Google in 2017 was considered. The analysis showed that the Architecture Components fit well into the concept and will interact with the "Clean Architecture" approach. Dagger 2 framework was applied for a complete abstraction and simplify testing. Also, it is planned to implement the RxJava library.

Keywords: android, architecture, clean architecture, architecture components, model-view-presenter, architecture layers, dependency rule, dependency injection.

1. Introduction

Android operating system provides a powerful foundation for developing applications that work on a variety of devices and form-factors. Android applications have a more complex structure unlike desktop programs that have a single entry point and run as a single monolithic process. A typical android application consists of various components: activities, fragments, services, content providers and broadcast receivers, that complicate the architecture design.

Qualitative architecture makes the process of developing and maintaining the program more simple and effective. The program must solve a set of tasks and perform its functions clearly and under different conditions. This includes such characteristics as reliability, security, performance and scalability. Any application has to change over time - requirements change, new ones are added. The faster and more convenient changes, which made by developers to the existing functional bring the less problems and errors this will cause. The architecture should parallelize the development process so that many people can work on the program at the same time. Also the project should be clearly structured, does not contain duplication, has well-designed code and documentation. The system should use standard, common and familiar solutions for developers, if possible. [4]

2. Theoretical part

Google did not provide recommendations for creating an application architecture, developer's had problems with a code "swelling" in activities, testing, scaling and supporting applications. Long enough Android developers had been solving all

these problems on their own, and a "Clean Architecture" concept gradually arose. It is desirable to keep in mind when developing any application.

The “Clean Architecture” is based on such basic rules:

- Architecture should not depend on various libraries and frameworks.
- Business rules should be tested without UI, database, server, etc.
- The UI should change easily, without changing the rest of the system.
- The database should be easily replaced, business rules should not be based on the database structure.
- Business rules do not need to know anything about the outside world. [1]

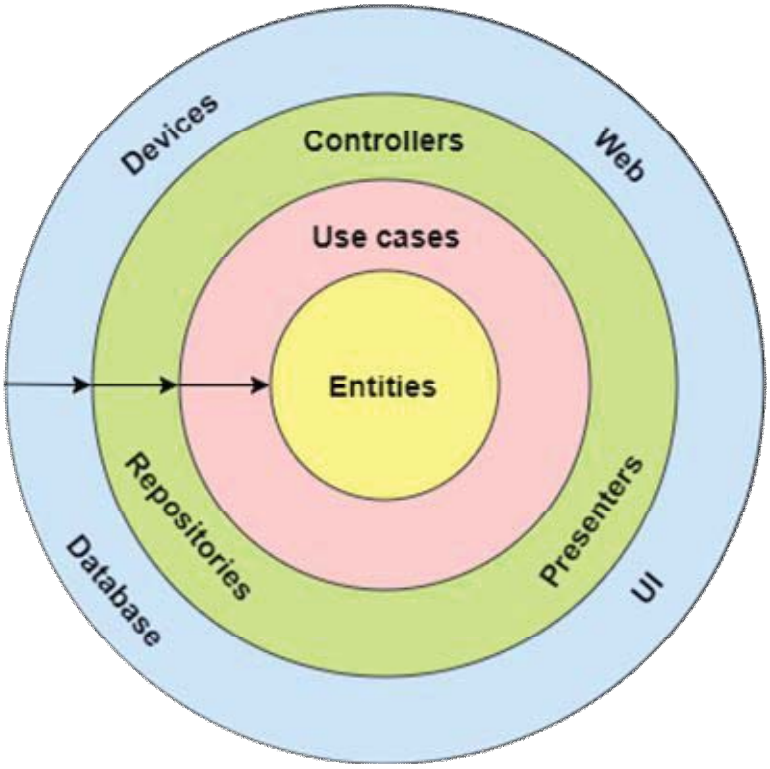

Fig. 1 – “Clean Architecture” layers

Circles represent different areas of software (Fig.1). In general, the further in you work, the higher level the software becomes. External circles are android-frameworks, internal - pure java-code without android-dependencies. It is important to consider The Dependency Rule: the code must have dependencies only in inner circles and should not know about what is happening in external circles. [1]

A glossary of terms to better understand this approach:

Entities: the business logic of the application. These are functions or objects with methods that implement business logic common to many applications, which are necessary for work and transition between layers.

Use cases: this layer contains the application's specific business rules. It encapsulates and implements all methods in the system. They are also called interactors - objects that implement the scenarios using entities.

Interface adapters: this set of adapters converts data from a format convenient for use cases and entities to a format convenient for UI-frameworks. These adapters include Presenter/Controllers.

Frameworks and drivers: user interface, various tools and frameworks, databases, etc. [5]

The goal of "Clean Architecture" is to separate tasks in such way as to keep the business rules not knowing anything at all about the outside world, thus, they can be tested without any dependency to any external element.

To achieve this, it is proposed to break the project into 3 layers (Fig.2), each of which has its own goal and can work independently of the others. It should be noted that each layer uses its own data model. Thus, the necessary independence and abstraction can be achieved. This approach helps to ensure that the layers do not overlap completely.

Presentation Layer. The application logic is associated with Views. There is nothing more than a Model-View-Presenter pattern, but it is possible to use any other pattern like Model-View-Controller or Model-View-ViewModel. This layer has only UI logic. Presenters communicate with the interactors, which involves working on a new thread, and passing through callbacks information that will be displayed in the View.

Domain Layer. It contains the entities and the use cases. This layer is a module on a pure java, without any android dependencies. All external components use interfaces to communicate with business objects.

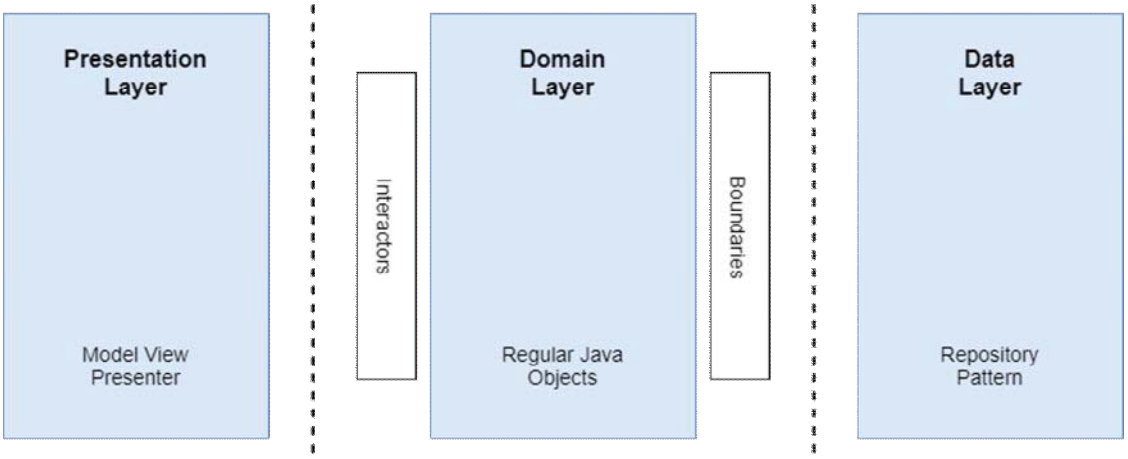

Fig. 2 – Project layers

Data Layer. All data required for the application is supplied from this layer through the implementation of the Repository. It uses the Repository pattern with a Strategy pattern that through the Factory pattern selects a various data sources depending on certain conditions.[5]

The approaches of "Clean architecture" are good, except that it is inappropriate to break logic into UI and business in an application. Many developers use an architecture, which can be represented by the following scheme (Fig.3):

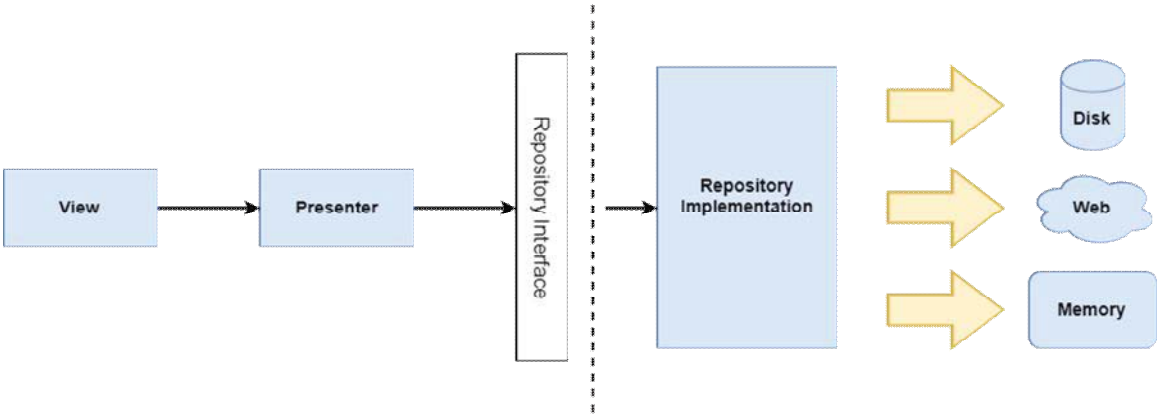

Fig. 3 – Simplified "Clean architecture" scheme

The main problem is that the problems of the lifecycle of android components aren't taken into account. Developers solve these problems on their own.

However, in 2017, Google presented their vision of architecture and offered their Architectural Components. The main idea of the new architecture from Google is the maximum logic removal from activities and fragments. This structure is quite similar to the previous one. Google has taken into account the previous ideas and efforts of developers (Fig.4).

New architecture components provide tools for binding the application kernel to lifecycle events and saving it from explicit dependencies. The Observer pattern was used for this earlier. This sounds simple enough, but often there are several simultaneous asynchronous calls and they all handle the lifecycles of their components. Some boundary cases can be easily overlooked and new components can help in this. One of the main problems in Android is the need to constantly subscribe/unsubscribe from some objects when calling lifecycle methods. And because only the Activity and Fragment have lifecycle methods, and objects like GoogleApiClient, LocationManager, SensorManager and others must be located only inside them, and this leads to a large number of lines of code in these files.[6] Google proposed using the LiveData class to solve this and other problems. LiveData is an observable container for data. It allows application components to monitor LiveData for changes without creating explicit dependencies between them. LiveData also takes into account the state of the lifecycle of application components (activities, fragments, services) and does everything possible to prevent object leakage. The Lifecycle component is used to ensure that LiveData considers the lifecycle, but also there is an ability to use it without reference to the lifecycle. The LiveData class is an abstract generic class which encapsulates the logic of the component.[2] Exactly, the same model can be used to implement server requests. The method of obtaining data does not change at all, but data is delegated to LiveData, which is essentially a binding for View.

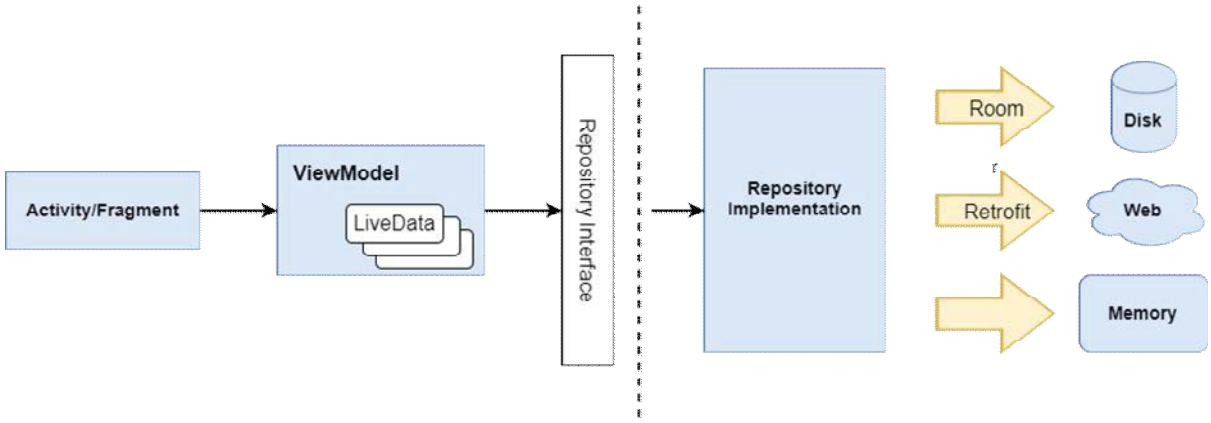

Fig. 4 – "Clean architecture" scheme ft. Google

How to deal with the problem of the lifecycle? Google has given a component which is going through the Activity recreation - ViewModel. ViewModel provides data for a specific component of the user interface, such as Activity or Fragment, and communicates with the business part of data processing, for example, calling other components to loading data or redirecting user changes.[2] That is, it can act as a Presenter/ViewModel in the presentation layer of "Clean Architecture". ViewModel does not know about the View and does not fall under the configuration changing, for example, screen rotation. Since ViewModel is going through the Activity recreation, LiveData will be created only once and the server request will be executed only once, that is, the main problems are solved.[6] In general, the ViewModel component can be described as a singleton with a collection of LiveData instances. It ensures that it will not be destroyed while there is an active activity exists. It is also worth noting that any amount of ViewModel can be bound to the Activity. In addition, Google released a new library for working with the database - Room, which will perfectly fit into the Repository implementation. Room abstracts some of the basic details of the implementation of working with raw SQL tables and queries. It also allows to monitor changes of database data using a LiveData object. [2]

The most important thing that needs to pay attention to is the division of responsibility in the application. Writing all code into an Activity or a Fragment is a big mistake. Any code that does not handle interaction with the user interface or an operating system should not be in these classes. Using them as independent as possible allows to avoid many problems related to the lifecycle.

The second important principle is that the user interface control must come from a preferably constant model. "Viability" is ideal for two reasons: users do not lose data even if the OS destroys the application to free resources and the application will continue to work even when the network connection is unstable or not connected. Keeping the user interface code simple and free from application logic simplifies management and maintaining. Basing on models with a clearly defined responsibility for data management make it easily testable and suitable for requirements.

3. Practical part

The "Clean Architecture" approach has been applied to communication application for mechatronics robot laboratory onaft in order to speed up and simplify development, quickly add new functionality and easily navigate and understand the existing code. Model-View-Controller pattern has been chosen for presentation layer of application. Domain and data layer were combined into the one interactor. Interactor observes data changes in the database. Retrofit and Realm library are used for server and database communication accordingly.

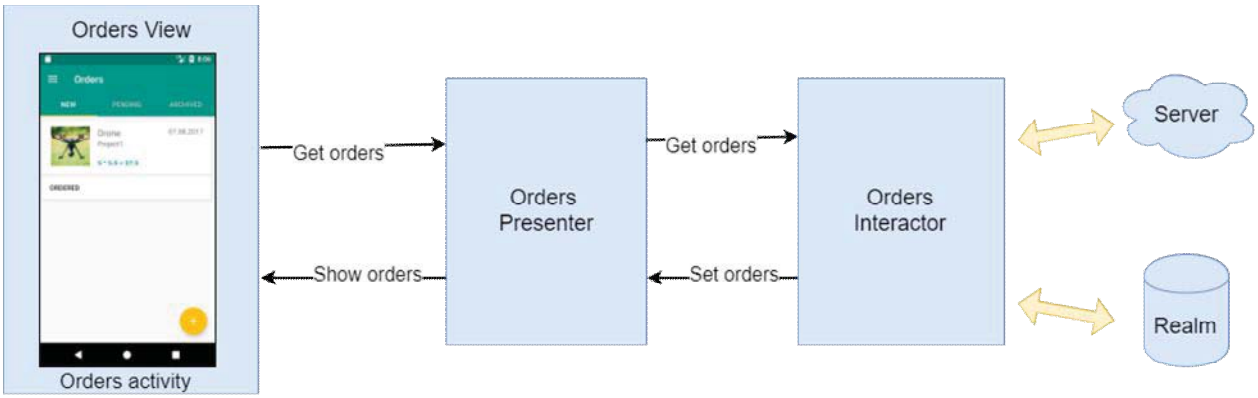

Fig. 5 – Orders section scheme

To facilitate testing and add new functions, the Dagger 2 framework has been used, which makes the components interchangeable. Dagger is a fully static, compile-time dependency injection framework for both Java and Android. It relies purely on using Java annotation processors and compile-time checks to analyze and verify dependencies. It is considered to be one of the most efficient dependency injection frameworks built to date.

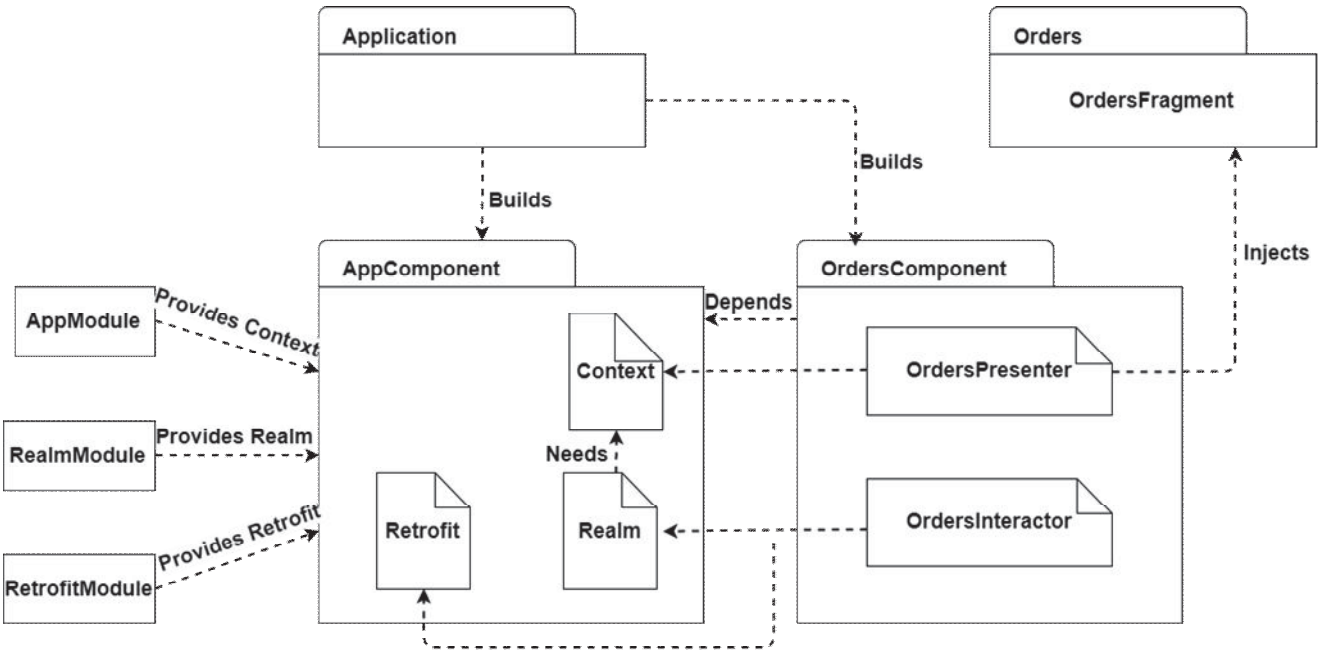

Fig. 6 – Dagger 2 scheme

Conclusions

Once the architecture components will have transferred from the alpha version to the release, they will be implemented immediately without affecting the application structure and development time, since the layers are practically independent of each other. Architecture components will take care of the configuration changes, that will make application even more stable and secure. In addition, RXJava library will be used to facilitate the work with the network, threads and simplify the code inside the interactor.

RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences. It extends the observer pattern to support sequences of data/events and adds operators that allow to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety and concurrent data structures.

Referenses

[1] "The Clean Architecture | 8th Light." [Online]. Available: https://www.bing.com/cr?IG=A77A11C07412436A919999694235260 0&CID=14E277DC4AEF68B206607C9E4BE96937&rd=1&h=KR-P8v9qhPgPNZiwLXzWAshNWwC7ffeOzuB6bK0wwmU&v=1&r=https%3a%2f%2f8thlight.com%2fblog%2funcle-bob%2f2012%2f08%2f13%2fthe-clean-architecture.html&p=DevEx,5068.1. [Accessed: 12-Sep-2017].

[2] "Guide to App Architecture," *Android Developers*, 06-Nov-2017. [Online]. Available: https://developer.android.com/topic/libraries/architecture/guide.html. [Accessed: 12-Sep-2017].

[3] "Zabluzhdenyya Clean Architecture / *Bloh kompanyy MobileUp* ..." [Online]. Available: https://www.bing.com/cr?IG=8059D7167490467799419ACEAD4603B8&CID=212704257BF4606227DD0F677AF26195&rd=1&h=nkXj-PiUGgOJHr6ZWv_31M38S4Ax0TxTd4jDWK40DGI&v=1&r=https%3a%2f%2fhabrahabr.ru%2fcompany%2fmobileup%2fblog%2f335382%2f&p=DevEx,5066.1. [Accessed: 12-Sep-2017].

[4] "Sozdanye arkhytekturы prohrammы yly kak proektyrovat' ..." [Online]. Available: https://www.bing.com/cr?IG=DDC03E5C2FBB4B44AC478CCE3AD652D7&CID=2067CB488D196BE01E9CC00A8C1F6AF9&rd=1&h=6_UUqlJ2HMip-7qVaWakR-zNiJA131Y3aSpePQOI4KY&v=1&r=https%3a%2f%2fhabrahabr.ru%2fpost%2f276593%2f&p=DevEx,5066.1. [Accessed: 22-Nov-2017].

[5] F. Cejas, "Architecting Android...The clean way?," *Fernando Cejas*, 02-Sep-2014. [Online]. Available: https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/. [Accessed: 12-Sep-2017].

[6] "Razbyraemsya s novыmy arkhytekturnыmy komponentamy v Android," / *Bloh kompanyy Google / Khabrakhabr*. [Online]. Available: https://habrahabr.ru/company/google/blog/330208/. [Accessed: 22-Nov-2017].

Література

[1] "The Clean Architecture," *8th Light*. [Online]. Available: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html. [Accessed: 12-Sep-2017].
[2] "Guide to App Architecture," *Android Developers*, 26-May-2017. [Online]. Available: https://developer.android.com/topic/libraries/architecture/guide.html. [Accessed: 12-Sep-2017].
[3] "Заблуждения Clean Architecture," / *Блогкомпании MobileUp / Хабрахабр*. [Online]. Available: https://habrahabr.ru/company/mobileup/blog/335382/. [Accessed: 12-Sep-2017].
[4] "Создание архитектуры программы или как проектировать табуретку," / *Хабрахабр*. [Online]. Available: https://habrahabr.ru/post/276593/. [Accessed: 12-Sep-2017].
[5] "Architecting Android...The clean way?," *Fernando Cejas*, 02-Sep-2014. [Online]. Available: https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/. [Accessed: 12-Sep-2017].
[6] "Разбираемся с новыми архитектурными компонентами в Android," / *Блог компании Google / Хабрахабр*. [Online]. Available: https://habrahabr.ru/company/google/blog/330208/. [Accessed: 12-Sep-2017].

UDC 697

THE MODEL FOR POWER EFFICIENCY ASSESSMENT OF CONDENSATION HEATING INSTALLATIONS

D. Kovalchuk[1], **A. Mazur**[2], **S. Hudz**[3]
[1,2,3]Odessa National Academy of food technologies
[1]Postgraduate, [2]Assistant professor, [3]Master student
ORCID: [1] 0000-0003-0549-5244
E-mail: [1]radiolomaster@gmail.com, [2]mazur.av.ua@gmail.com, [3]sergogudz@gmail.com

DOI: 10.15673/atbp.v9i3.715

Abstract: The main part of heating systems and domestic hot water systems are based on the natural gas boilers. For increasing the overall performance of such heating system the condensation gas boilers was developed and are used. However even such type of boilers don't use all energy which is released from a fuel combustion. The main factors influencing the lowering of overall performance of condensation gas boilers in case of operation in real conditions are considered. The structure of the developed mathematical model allowing estimating the overall performance of condensation gas boilers (CGB) in the conditions of real operation is considered. Performace evaluation computer experiments of such CGB during a heating season for real weather conditions of two regions of Ukraine was made. Graphic dependences of temperature conditions and heating system effectiveness change throughout a heating season are given. It was proved that normal CGB does not completely use all calorific value of fuel, thus, it isn't effective. It was also proved that the efficiency of such boilers significantly changes during a heating season depending on weather conditions and doesn't reach the greatest possible value. The possibility of increasing the efficiency of CGB due to hydraulic division of heating and condensation sections and use of the vapor-compression heat pump for deeper cooling of combustion gases and removing of the highest possible amount of thermal energy from them are considered. The scheme of heat pump connection to the heating system with a convenient gas boiler and the separate condensation economizer allowing to cool combustion gases deeply below a dew point and to warm up the return heat carrier before a boiler input is provided. The technological diagram of the year-round use of the heat pump for hot water heating after the end of heating season, without gas use is offered.