DOI: 10.20998/2079-0023.2025.02.08
UDC 004.41

D. E. DVUKHHLAVOV, Candidate of Technical Sciences (PhD), Docent, Associate Professor at the Department of Department of Software Engineering and Management Intelligent Technologies, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine; e mail: dmytro.dvukhhlavov@khpi.edu.ua; ORCID: https://orcid.org/0000-0002-3361-3212

O. S. PELYPETS, Master Student, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine; e-mail: pelipets.olga.developer@gmail.com; ORCID: https://orcid.org/0009-0005-5974-9045

A. S. DVUKHHLAVOVA, Senior Lecturer at the Department of Software Engineering and Management Intelligent Technologies, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: alona.dvukhhlavova@khpi.edu.ua; ORCID: https://orcid.org/0000-0002-0111-3010

ANDROID APPLICATION MODULARIZATION ESTIMATING MODEL

The relevance of the research, the results of which are presented, is determined by the fact that mobile applications have evolved into complex software systems with growing code bases, which complicates development, testing and support. It is shown that improving the maintainability and scalability of Android applications projects is possible by moving from a monolithic architecture to a modular architecture, based either on the list of functions that the application should perform, or on the architectural features of creating the application. To select a modularization option, a classification of approaches to modularization implementing is proposed. Regardless of which direction of modularization implementing is chosen, it is aimed at reducing the impact of changes in one module on the need to make changes to others. Such a dependence between modules can be assessed by determining the cohesion and coherence of the project and individual modules. To quantitatively assess the advantages of modularization, a mathematical model has been developed that takes into account the balance between the cohesion of modules and the integrity of the project in whole. The model proposes to take into account the number of modules into which the monolithic architecture will be divided, the level of interaction between the modules that will be selected, as well as the level of their dependence on each other. Expressions are presented for automating calculations of division options into modules. The results of the assessment of the modularization of the Android application project for e-commerce based on different approaches to modularization implementing are presented. The obtained evaluation data allowed us to demonstrate the potential of modularization in reducing project assembly time, minimizing conflicts, and increasing project flexibility, offering a scalable solution for modern mobile development.

Keywords: classification of approaches to Android application modularization implementing, modularization model, evaluation indicators for Android application modularization options, project cohesion and coherence.

Introduction. Mobile applications perform the majority of informational tasks required by people in everyday life. Over the past decade, mobile development has undergone significant evolution: from simple applications with basic functions, such as calculators or notepads, it has advanced to complex software systems, such as social networks, governmental programs, navigation systems, banking applications, and business software. As the power and functionality of mobile devices increase, so do user demands. Project sizes are continuously expanding, and their complexity is rising exponentially. Modern mobile applications often contain hundreds of thousands of lines of code and numerous integrations with external services, making their development, testing, and maintenance increasingly challenging tasks. The main typical problem faced by large projects is the complex structure of the code, which makes it difficult to detect errors. The complex structure of monolithic code complicates both manual and automated testing, makes it labor-intensive and resource-intensive, and flexibility, scalability and ease of support suffer.

Purpose of the work. The briefly described features of modern mobile application development determine the relevance of research that considers the issue of structuring the code of large-scale mobile applications, since unstructured code can lead to development delays, increased costs and reduced product quality. This article presents the results of one of the studies in this scientific and practical direction, the object of which is the process of creating mobile applications for the Android platform with a large amount of code. The subject of this study is approaches to organizing development and principles of code structuring to ensure maintainability and scalability of the mobile application project.

The purpose of the research was to reduce the time for testing and compiling a version of the mobile application.

The purpose of the article, which is presented at the discretion of specialists, is to reveal the concept of modularization of a mobile application project, present a classification of approaches to implementing modularization and a model for assessing the effectiveness of modularization, as well as present the results of assessing the effectiveness of modularization for different approaches using the example of one of the projects of a specific development company.

Research results.

1. Analysis of problem areas in the development of large-scale mobile applications projects and known approaches to solving them. Studying sources [1] and [2] allows to create a vision of the mobile application development process. The development process of a mobile application often begins with a single-module "Hello World!" project and continues by adding new code, creating function after function.

Developers divide the code mobile application into meaningful parts based on the app’s features. Popular examples of such features are Authorization, Registration, Onboarding, Home Screen, Search, Product Catalog, Product Details Page, Account, Payment, Favorites, etc [3].

Developers in same time also attempt to divide the code according to the principles of Clean Architecture and SOLID [4]. That's why every project usually contains separate packages with code for business logic (domain layer), code for retrieving data from the network, files, or database (data layer), and code for presenting the interface (UI layer). In all modern projects Dependency Injection [5] frameworks are commonly used to create instances of classes according to their scope.

Staying within a single module, classes have access to all other classes because of their internal visibility. That's why they are often misused, contrary to the principles of single responsibility, interface segregation, and the open-closed principle. As result a team create typical single-module monolithic giant projects.

A typical large project is developed and maintained by a big developer team. This means that all developers regularly encounter merge conflicts, the resolving of which is a complex technical task. These conflicts can lead to new bugs that are difficult to prevent.

Additionally, a large monolithic project on each developer's local machine has a certain compilation time. Changing any line of code triggers a long, full rebuild of the project, reducing work productivity.

Moreover, large projects use automated processes for code testing, build assembly and distribution. Typically, each merge request on a remote server is checked by lint and ktlint; the code is compiled into some build variants, and JUnit tests and integration tests are run. Also checks of code quality, code security, codebase size, and content uniqueness may also be executed [6]. Depending on the project size and the number of checks and tests, the entire process for a single merge request can take from a couple of minutes to an hour or more of expensive server machine time. Also the productivity of developers decreases while they wait for the remote verification to complete.

Over time any project grows and requires changes. Each change can introduce errors into the existing code, which either go unnoticed and turn into crashes in production or are identified by costly manual or automated regression testing.

Since the obvious cause of the listed problems is the large size of programs, the evident solution is to divide projects into smaller parts that can be developed, tested, and compiled separately. A common approach is to extract parts of the code into separate projects, which are then included in the main project as pre-compiled JAR or AAR files or as external dependencies of the code assembler that bring pre-compiled code into the project [7]. This approach is commonly used by companies that develop multiple software products built on reusable internal libraries. The main benefits of this approach are reduced code conflicts, faster compilation times, and faster testing of the main project. The effect is achieved because writing the code in a separate, small repository eliminates merge conflicts and improves code quality. Only public classes are visible externally, while the internal logic remains encapsulated. The code in such a library is covered by tests and checks that are only run when the library itself changes. Compilation occurs before the library is deployed, not in the main program.

However, the approach also has disadvantages related to architectural limitations. Not every fragment of code can be isolated and reused for a long time without modifications. If functionality undergoes frequent changes, it must be updated, tested, and published before being used in the main project. Additionally, if this functionality depends on another internal library that also requires changes, maintaining and developing such projects becomes a complex technical challenge related to versioning. Therefore, this approach is typically not applied to projects that do not require code reuse, and therefore there is a need to find other approaches to structuring the mobile application code.

2. The essence of a modularization and classification of approaches to implementing modularization a mobile application project.

2.1. The essence of modularization and basic assessment of the effectiveness of its implementation. The concept of "modularization of a mobile application project" should be understood as the distribution of project code into files to ensure ease of development, flexibility to changes in requirements, maintainability and scalability. The term is compiled based on a similar term in robotics [8].

Assuming that the compilation time of a monolith project depends on the number of files n and the complexity of dependencies between them $\mathrm{O}\left(f(n)\right)$:

$$T_{\text{mono}} = \mathrm{O}\left(f(n)\right).$$

In a modular approach, where the project is divided into m independent modules, each containing *n/m* files, the compilation time becomes:

$$T_{\text{modular}} = \mathrm{O}\left(\frac{f(n)}{m}\right).$$

To build projects, the development company on which the research was conducted uses Gradle, an Android application build system that provides flexibility, automation, and support for numerous plugins. It is the standard in modern Android development due to its adaptability and efficiency [9]. Dividing a project into independent Gradle modules allows you to build and cache them separately. Gradle supports parallel and incremental builds.

Incremental building means that Gradle rebuilds only the modified files, leaving the rest of the project untouched. At the same time, separate parts are built simultaneously in parallel processes, significantly speeding up the build. The Gradle Build Cache configuration allows reuse of results from previous builds, avoiding duplicated effort.

Time of incremental compilation:

$$T_{\text{build}} = \mathrm{O}\left(f(\Delta n)\right).$$

where Δn is the number of modified files.

In a monolith project $\Delta n \approx n$, whereas in a modular project $\Delta n \approx n/m$.

Thus, even splitting a single-module project in half into two modules can approximately halve the build time if changes are made to only one module. This is highly beneficial for large projects, where every second of saved time matters.

At the same time the number of merge conflicts depends on the number of modified files *F* and number of developers *D*:

$$C_{\text{mono}} = \mathrm{O}\left(F \cdot D\right).$$

If the code is divided into modules and developers work independently then:

$$C_{\text{mono}} = \mathrm{O}\left(\frac{F \cdot D}{m}\right).$$

This means that the more modules there are, the fewer code conflicts occur.

In the course of further research, results were obtained that allow implementing modularization and assessing the feasibility of its use.

2.2. Formal model of modularization. The first step was to create a formal model of the Android application.

Android application may be represented as a set of all its components $\boldsymbol{S}$ (e.g., classes, functions, resources, etc.) Then modularization may be considered as a division the set $\boldsymbol{S}$ into subsets (modules) that meet certain criteria (functionality, independence, reusability).

The set of all components of the application:

$$\boldsymbol{S} = \{c_1, c_2, ..., c_N\}.$$

Modularization $\boldsymbol{M}$ may be considered as a family of subsets, where each module is subset of $\boldsymbol{S}$:

$$\boldsymbol{M} = \{\boldsymbol{M}_1, \boldsymbol{M}_2, ..., \boldsymbol{M}_i, ..., \boldsymbol{M}_K\}, \boldsymbol{M}_i \subseteq \boldsymbol{S}.$$

The union of all modules covers the entire application:

$$\bigcup_{i=1}^{K} \boldsymbol{M}_i = \boldsymbol{S}.$$

In the ideal case modules do not intersect, though in reality, there may be weak dependencies:

$$\boldsymbol{M}_i \cap \boldsymbol{M}_i = \varnothing, i \neq j.$$

Thus, modularization is the portioning of the set *S* into non-overlaping (or minimally overlapping) subsets:

$$\boldsymbol{S} = \boldsymbol{M}_1 \cup \boldsymbol{M}_2 \cup ... \cup \boldsymbol{M}_k;$$
$$\boldsymbol{M}_i \cap \boldsymbol{M}_j = \varnothing, i \neq j.$$

The division of a project into modules can be done in various ways [10], which are discussed below.

2.3. Classification of approaches to implementing modularization.

2.3.1. Horizontal Modularization. The horizontal approach is based on dividing the code into layers in accordance with the principles of Clean Architecture [11], where the project is split into data, domain, and UI layers:

$$\boldsymbol{M} = \{\boldsymbol{M}_{\text{data}}, \boldsymbol{M}_{\text{domain}}, \boldsymbol{M}_{\text{UI}}\}.$$

Compilation time of monolith project is:

$$T_{\text{mono}} = T_{\text{data}} + T_{\text{domain}} + T_{\text{UI}},$$

where T_{data}, T_{domain} and T_{UI} are compilation time of code of appropriate parts.

For a modular approach where work is done in parallel:

$$T_{\text{hor.modular}} = \max(T_{\text{data}}, T_{\text{domain}}, T_{\text{UI}}).$$

It is recommended to begin the dividing by identifying business entities (in the terminology of the Kotlin language, used for Android development, these are data classes). Business entities are managed by usecases, which provide data for screen models. Usecases retrieve data from repositories, which are hidden behind interfaces. Entities, usecases and repository interfaces are extracted into a separate domain module, which has no external dependencies and does not depend on the type of software product, as it defines only business logic.

Next, the implementation of repositories and all logic for retrieving data from the network, files, and databases are moved to a separate data module. The data module operates with its own internal entities (data classes), which are serialized for storage and transmission and deserialized and transformed into domain entities for delivering data to the domain. Thus, the data module depends on the domain module and on external libraries for data handling (Retrofit, Socket, Room, Preferences, Data Storage).

The remaining main module contains the code for the UI presentation, which depends on the domain module, from where it retrieves data. The Dependency Injection framework (Hilt, Dagger) creates entities, keeping the data logic hidden from the UI presentation (look at Fig. 1).

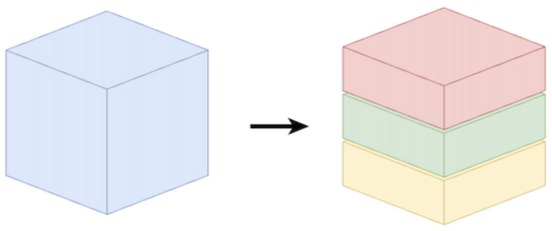
Fig. 1. Horizontal Modularization

Thus, any project can be divided into three parts, each of which is compiled and tested separately.

2.3.2. Vertical Modularization. The strategy of vertical module splitting is applicable when a project can be divided into features that are independent of one another. This, for example, works well for projects with a plain client-server architecture, where each individual screen (or a group of screens of the same feature) can be isolated. In this approach, one base module is identified, which manages navigation to the screens of other modules via interfaces. For instance, in a typical E-commerce project, these could be modules for product, cart, order, and user profile (see Fig. 2). The feature modules do not depend on each other and know nothing about one another, their logic is encapsulated. Only the base module is aware of the public interfaces of the other modules. In formal view division may be presented as

$$\boldsymbol{M} = \{\boldsymbol{M}_{\text{base}}, \boldsymbol{M}_{\text{product}}, \boldsymbol{M}_{\text{order}}, ..., \boldsymbol{M}_{\text{profile}}\}.$$

The compilation time in the vertical approach is

$$T_{\text{vert.modular}} = \sum_{i=1}^{K} \mathrm{O}\left(f(n_i)\right),$$

where n_i is the size of every feature module.

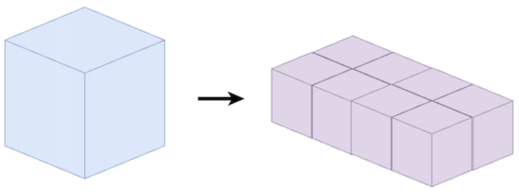

Fig. 2. Vertical Modularization

Thus, this approach resolves issues related to compilation time, code conflicts and test execution.

2.3.3. Combined Modularization. In small to medium-sized projects, applying only vertical or horizontal modularization may suffice. However, in large enterprise projects with distributed development teams, the effectiveness of modularization may remain low if only one approach is used.

In these situations, a combined approach is applied, integrating both vertical and horizontal modularization [12].

The code is split into modules both vertically and horizontally. Each functionality is isolated into separate data, domain, and UI modules (see Fig. 3).

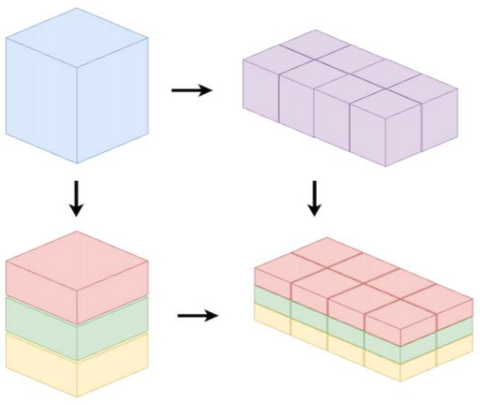

Fig. 3. Combined Modularization

Domain modules may depend only on each other. Each data module depends only on its corresponding domain module and remains hidden from other data and UI modules. The code for each feature screens is encapsulated in separate UI modules, which depend on their respective domain modules and expose only navigation interfaces externally. A single base module retrieves all entities from Dependency Injection framework, launches the mobile application and manages navigation.

In formal view division may be presented as

$$\boldsymbol{M} = \left\{ \begin{matrix} \boldsymbol{M}_{\text{data1}}, \boldsymbol{M}_{\text{domain1}}, \boldsymbol{M}_{\text{UI1}} \\ \dots \\ \boldsymbol{M}_{\text{dataK}}, \boldsymbol{M}_{\text{domainK}}, \boldsymbol{M}_{\text{UIK}} \end{matrix} \right\}.$$

The total compilation time for a fully divided project:

$$T_{\text{comb.modular}} = \max_i \mathrm{O}\left(\frac{f(n_i)}{m}\right),$$

This approach minimizes merge conflicts and significantly reduces the time required for building and testing the project, as Gradle efficiently uses caching, and tests are executed only for the modules that have been modified. The number of tests remains minimal since only small, isolated modules are tested.

A drawback of this approach is the complexity of creating the modules. Therefore, while combined modularization is the best solution for large-scale projects, determining the optimal number of modules remains an open question. To address this, a mathematical model is required to calculate the benefits of modularization.

3. Mathematical model of modularity estimating. To assess the benefits of modularizing native Android applications, a mathematical model is proposed that accounts for the internal cohesion of modules and their external dependencies.

The model is based on the following elements.

Let it created a set of modules ***M***:

$$\boldsymbol{M} = \{\boldsymbol{M}_1, \boldsymbol{M}_2, \dots, \boldsymbol{M}_K\}.$$

where $\boldsymbol{M}_i$ is an individual module.

A directed graph of connections may be described as

$$\boldsymbol{G} = (\boldsymbol{M}, \boldsymbol{E}),$$

where $\boldsymbol{E} \subseteq \boldsymbol{M} \times \boldsymbol{M}$ represents dependencies between modules (it is the set of directed edges that describe the connections between modules).

For each edge $e_{ij} \in \boldsymbol{E}$ the communication cost between modules $\boldsymbol{M}_i$ and $\boldsymbol{M}_j$ can be defined as

$$C_{ij} = f(L_{ij}, D_{ij}),$$

where L_{ij} is the connection latency and D_{ij} is the volume of data transmitted.

The key idea lies in balancing two characteristics: the cohesion of a module $\mathrm{Coh}(\boldsymbol{M}_i)$ and the coherence of the system $\mathrm{Cohes}(\boldsymbol{M})$. Cohesion $\mathrm{Coh}(\boldsymbol{M}_i)$ measures the internal consistency of a module (in conditional units from 0 to 100), while coherence $\mathrm{Cohes}(\boldsymbol{M})$ is defined as the number of connections between different modules $|\boldsymbol{E}|$.

The objective function for the benefit of multi-modularity is given as follows:

$$V(\boldsymbol{M}) = \boldsymbol{\alpha} \cdot \sum_{i=1}^{N} \mathrm{Coh}(\boldsymbol{M}_i) - \boldsymbol{\beta} \cdot \mathrm{Cohes}(\boldsymbol{M}),$$

where $\boldsymbol{\alpha}$ is the weight coefficient for cohesion, and $\boldsymbol{\beta}$ is the weight coefficient for coherence.

To evaluating modularization more realistically, the objective function have been extended by adding a saturation term for the intrinsic benefit of modularization and a fragmentation cost that grows with the number of modules:

$$V(\boldsymbol{M}) = \boldsymbol{\alpha} \cdot \sum_{i=1}^{N} \mathrm{Coh}(\boldsymbol{M}_i) - \boldsymbol{\beta} \cdot \mathrm{Cohes}_{\mathrm{PR}} + \\ + B_{\max} \cdot (1 - e^{-kN}) - C_{\mathrm{cpl}} N^q,$$

where $\mathrm{Cohes}_{\mathrm{PR}}$ – the level project coherence between modules (to distinguish it from per-module cohesion); $B_{\max} \cdot (1-e^{-kN})$ – the saturation benefit of having more modules; $B_{\max}$ – the maximum attainable bonus, $k > 0$ is the saturation rate; $C_{\mathrm{cpl}} N^q$ – the fragmentation (coordination) cost that grows with the number of modules; $C_{\mathrm{cpl}} > 0$ – the amplitude of that cost; q is a dimensionless scaling exponent that controls the nonlinearity of the fragmentation cost.

In the subsequent calculations the following empirically derived coefficient values was used: $\boldsymbol{\alpha} = 0.5$, $\boldsymbol{\beta} = 0.35$, $B_{\max} = 900$, $k = 0.06$, $C_{\mathrm{cpl}} = 0.6$, $q = 1.8$.

The goal of the model is to maximize $V(\boldsymbol{M})$, which is achieved by increasing the internal consistency of modules while minimizing their external dependencies.

4. Practical calculations. To demonstrate the model's functionality, let's consider an example of an E-Commerce application on Android, written in Kotlin, with a total codebase of 200 000 lines. The application includes eight functional areas: authorization, product pages, product categories, search, bag, checkout, user profile (payment details, shipping address), and order history. Is was calculated $V(\boldsymbol{M})$ for three architectural variants: a single-module project, a multi-module project with 8 modules, and a multi-module project with 24 modules based on Clean Architecture principles.

4.1. Single-Module Project. In the first variant, the entire application code (200 000 lines) resides in a single module $\boldsymbol{M}_1$, encompassing all functions – from authorization to order history. The number of modules is $n=1$ and the size $|\boldsymbol{M}_1| = 200\ 000$ significantly exceeds $S_{\max}$, which is permissible for this baseline case. Cohesion $\mathrm{Coh}(\boldsymbol{M}_1) = 50$ as the mixing of eight diverse functions (e.g., payment logic with category UI) reduces internal consistency. There are no external connections, so $\mathrm{Cohes}_{\mathrm{PR}} = 0$. Calculation:

$$V(\boldsymbol{M}_1) = 0.5 \cdot 50 - 0.35 \cdot 0 + \\ + 900 \cdot (1 - e^{-0.06 \cdot 1}) - 0.6 \cdot 1^{1.8} = 76.81.$$

The value $V(\boldsymbol{M}_1) = 76.81$ reflects low benefit due to weak cohesion, although the absence of inter-module dependencies eliminates any coherence penalty. This approach is typical for monolithic applications, where maintenance and scaling are challenging.

4.2. Multi-Module Project (8 Modules). In the second variant, the application is divided into 8 feature-modules, each responsible for a separate function.

The number of modules is *n=8*, and the code is distributed as follows: authorization – 15 000 lines; categories – 30 000; search – 20 000; cart – 25 000; payment – 30 000; profile – 25 000; history – 15 000.

Summa of lines in modules equals 200 000.

Module cohesion is high due to functional isolation:

- $\mathrm{Coh}(\boldsymbol{M}_1) = 90$ (authorization);
- $\mathrm{Coh}(\boldsymbol{M}_2) = 85$ (product pages);
- $\mathrm{Coh}(\boldsymbol{M}_3) = 80$ (categories);
- $\mathrm{Coh}(\boldsymbol{M}_4) = 85$ (search);
- $\mathrm{Coh}(\boldsymbol{M}_5) = 80$ (bag);
- $\mathrm{Coh}(\boldsymbol{M}_6) = 85$ (checkout);
- $\mathrm{Coh}(\boldsymbol{M}_7) = 80$ (profile);
- $\mathrm{Coh}(\boldsymbol{M}_8) = 90$ (order history).

Sum of cohesions:

$$\sum_{i=1}^{8} \mathrm{Coh}(\boldsymbol{M}_i) = \\ = 90 + 85 + 80 + 85 + 80 + 85 + 80 + 90 = 675.$$

Connections between modules include:

- authorization → profile;
- authorization → cart;
- product pages → bag;
- categories → product pages;
- search → product pages;
- bag → checkout; checkout → order history;
- profile → checkout;
- order history → product pages.

Total $\mathrm{Cohes}_{\mathrm{PR}} = 9$.

Calculation:

$$V(\boldsymbol{M}_8) = 0.5 \cdot 675 - 0.35 \cdot 9 + \\ +900 \cdot (1 - e^{-0.06 \cdot 8}) - 0.6 \cdot 8^{1.8} = 652.11.$$

The value $V(\boldsymbol{M}_8) = 652.11$ demonstrates a significant increase in benefit compared to the single-module variant $V(\boldsymbol{M}_1) = 76.81$ due to high cohesion (average $\mathrm{Coh}(\boldsymbol{M}_\mathrm{i}) \approx 84.4$) and moderate coherence, confirming the effectiveness of modularization for medium and large applications.

4.3. Multi-Module Project with Clean Architecture (24 Modules). In the third variant, each of the 8 feature-modules is split into three layers according to Clean Architecture principles: data layer (data handling), domain layer (business logic), and UI layer (interface). This results in $n = 8 \cdot 3 = 24$ modules. The code has been divided into several parts, the description of which is presented below.

Feature Authorization: data 4 000 lines, domain 3 000, UI 8 000 lines of code (15 000).

Feature Product page: data 10 000, domain 8 000, UI 22 000 (40 000).

Feature Categories: data 8 000, domain 6 000, UI 16 000 (30 000).

Feature Search: data 5 000, domain 4 000, UI 11 000 (20 000).

Feature Bag: data 6 000, domain 5 000, UI 14 000 (25 000).

Feature Checkout: data 8 000, domain 6 000, UI 16 000 (30 000).

Feature Profile: data 6 000, domain 5 000, UI 14 000 (25 000).

Feature Order History: data 4 000, domain 3 000, UI 8 000 (15 000).

Total: 200 000 lines of code.

Layer cohesion is high due to strict isolation: $\mathrm{Coh}(\boldsymbol{M}_{\mathrm{data}i}) = 90$ (data logic); $\mathrm{Coh}(\boldsymbol{M}_{\mathrm{domain}i}) = 95$ (use cases); $\mathrm{Coh}(\boldsymbol{M}_{\mathrm{UI}i}) = 80$ (because UI depends on domain).

Calculation of cohesions: $8 \cdot 90 = 720$ (data); $8 \cdot 95 = 760$ (domain); $8 \cdot 80 = 640$ (UI).

Sum of cohesions:

$$\sum_{i=1}^{24} \mathrm{Coh}(\boldsymbol{M}_i) = 720 + 760 + 640 = 2120.$$

Calculation:

$$V(\boldsymbol{M}_{24}) = 0.5 \cdot 2120 - 0.35 \cdot 25 + \\ + 900 \cdot (1 - e^{-0.06 \cdot 24}) \quad - 0.6 \cdot 24^{1.8} = 1554.98.$$

The value $V(\boldsymbol{M}_{24}) = 1554.98$ demonstrates maximum benefit due to high cohesion (average $\mathrm{Coh}(\boldsymbol{M}_i) \approx 88.3$), despite the increase in coherence.

4.4. Variants 4, 12 and 50 modules. In the same way calculations for 4 modules, 12 modules and 50 modules project were done to expand data for diagram.

4-Module Variant ($V(\boldsymbol{M}_4) = 348.36$) shows a significant improvement over the single-module variant due to better functional isolation, but its benefit is limited by lower cohesion (broader feature groups) and moderate coherence. It is a practical starting point for smaller projects or teams transitioning from a monolithic architecture.

12-Module Variant ($V(\boldsymbol{M}_{12}) = 901.06$) offers a higher benefit than the 8-module variant due to finer granularity and higher cohesion, but it is penalized by significantly higher coherence (38 edges vs. 9 in the 8-module variant) due to the addition of utility modules. This approach is suitable for medium-to-large projects where shared utilities are beneficial but not yet requiring full Clean Architecture.

50-Module Variant ($V(\boldsymbol{M}_{50}) = 1427.27$) shows a lower benefit than the 24-module optimum, indicating a smooth decline beyond the peak due to diminishing cohesion gains and increasing fragmentation/coordination costs from over-modularization.

Results are present below in table 1 and Fig. 4.

Table 1 – Results of calculation of modularity benefits

Variant	n	$\sum_{i=1}^{N} \mathrm{Coh}(\boldsymbol{M}_i)$	$\mathrm{Cohes}_{\mathrm{PR}}$	$V(\boldsymbol{M})$
Single-Module	1	50	0	76.81
Multi-Module 1	4	330	4	348.36
Multi-Module 2	8	675	9	652.11
Multi-Module 3	12	1010	38	901.06
Multi-Module 4	24	2120	25	1554.98
Multi-Module 5	50	2600	120	1427.23

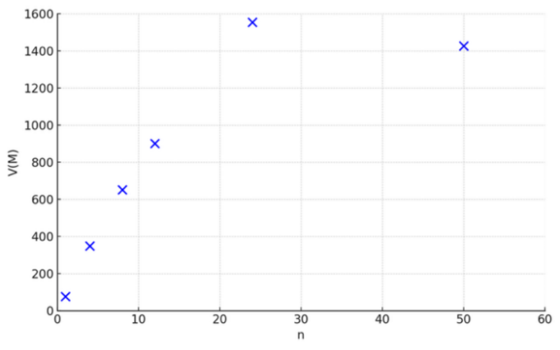

Fig. 4. Dependency of Modularization Benefit $V(\boldsymbol{M})$ on Number of Modules n

4.5. Analysis. Comparing the three variants reveals: the single-module project ($V(\boldsymbol{M}_1) = 76.81$) offers low benefit due to weak cohesion; the 8-module variant ($V(\boldsymbol{M}_8) = 652.11$) improves the result through functional isolation; the 24-module variant with Clean Architecture ($V(\boldsymbol{M}_{24}) = 1554.98$) maximizes $V(\boldsymbol{M})$ due to strict layer separation. For Android applications on Kotlin, this underscores the value of modularization, especially $\mathrm{Cohes}(\boldsymbol{M})$ with tools like Hilt to minimize dependencies. The increase in from 0 to 25 is offset by the rise in $\sum \mathrm{Coh}(\boldsymbol{M}_i)$ from 50 to 2120, making the 24-module approach optimal for large projects.

Conclusions. Presented results of the study demonstrates that modularization significantly enhances the maintainability and scalability of Android applications, addressing the limitations of monolithic architectures. By applying horizontal, vertical, and combined modularization strategies rooted in SOLID principles and Clean Architecture, developers can reduce merge conflicts, accelerate build times, and streamline testing processes.

The proposed mathematical model provides a quantitative framework to evaluate these benefits, revealing that a 24-module structure, integrating feature and layer-based separation, offers the greatest advantage for large-scale projects (benefit score: 1554.98) compared to single-module (76.81) or 8-module (652.11) designs. While the complexity of managing numerous modules poses challenges, tools like Gradle and Dependency Injection frameworks (e.g., Hilt) mitigate these drawbacks. Future research could refine the model by incorporating dynamic factors such as team size, development velocity, or real-world performance metrics, further optimizing modularization strategies for enterprise mobile applications across platforms.

References

1. *Android Developers. Guide to App Modularization.* URL: https://developer.android.com/topic/modularization (accessed 11.11.2025).
2. Gorin M. *Modular Architecture: The Key to Efficient Mobile App Development. Medium.* URL: https://maxim-gorin.medium.com/modular-architecture-the-key-to-efficient-mobile-app-development-8c0640edfff4 (accessed 11.11.2025).
3. *ACA Group. The Benefits of a Modular Architecture in Mobile Development.* URL: https://acagroup.be/en/blog/the-benefits-of-a-

modular-architecture-in-mobile-development/ (accessed 11.11.2025).

4. Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Upper Saddle River, NJ: Prentice Hall, 2017. 432 p.
5. *Android Developers. Dependency injection in Android.* URL: https://developer.android.com/training/dependency-injection (accessed 11.11.2025).
6. Chow J. *Software Architecture with Kotlin: Combine various architectural styles to create sustainable and scalable software solutions.* Packt Publishing, 2024. 462 p.
7. Wangereka, H. (2024) *Mastering Kotlin for Android 14: Build powerful Android apps from scratch using Jetpack libraries and Jetpack Compose.* Packt Publishing, 2024. 370 p.
8. ISO. (2021) ISO 22166-1:2021 Robotics – Modularity for service robots – Part 1: General requirements. Edition 1. Geneva: International Organization for Standardization. 69 p.
9. *Gradle. Build Tool.* URL: https://gradle.org/ (accessed 11.11.2025).
10. Pelypets O. S., Dvukhhlavov D. E. Strategies of Modularization for Android Applications. *Інформаційні технології: наука, техніка, технологія, освіта, здоров'я. Тези доповідей міжнародної науково-практичної конференції MicroCAD-2025 (14–17 травня 2025 р., м.Харків).* Харків: НТУ «ХПІ», 2025. С. 1395.
11. Aigner S., Elizarov R., Isakova S., Jemerov D. *Kotlin in Action, Second Edition.* Manning Publications, 2024. 560 p.
12. Campos, E., Kulesza U., Coelho R., Bonifácio R., Mariano L. Unveiling the Architecture and Design of Android Applications. *An Exploratory Study. Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS).* P. 201–211. DOI: 10.5220/0005398902010211.

References (transliterated)

1. *Android Developers. Guide to App Modularization.* Available at: https://developer.android.com/topic/modularization (accessed 11.11.2025).
2. Gorin M. *Modular Architecture: The Key to Efficient Mobile App Development. Medium.* Available at: https://maxim-gorin.medium.com/modular-architecture-the-key-to-efficient-mobile-app-development-8c0640edfff4 (accessed 11.11.2025).
3. *ACA Group. The Benefits of a Modular Architecture in Mobile Development.* URL: https://acagroup.be/en/blog/the-benefits-of-a-modular-architecture-in-mobile-development/ (accessed 11.11.2025).
4. Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Upper Saddle River, NJ: Prentice Hall, 2017. 432 p.
5. *Android Developers. Dependency injection in Android.* Available at: https://developer.android.com/training/dependency-injection (accessed 11.11.2025).
6. Chow J. *Software Architecture with Kotlin: Combine various architectural styles to create sustainable and scalable software solutions.* Packt Publishing, 2024. 462 p.
7. Wangereka, H. (2024) *Mastering Kotlin for Android 14: Build powerful Android apps from scratch using Jetpack libraries and Jetpack Compose.* Packt Publishing, 2024. 370 p.
8. ISO. (2021) ISO 22166-1:2021 Robotics – Modularity for service robots – Part 1: General requirements. Edition 1. Geneva: International Organization for Standardization. 6 pp.
9. *Gradle. Build Tool.* Available at: https://gradle.org/ (accessed 11.11.2025).
10. Pelypets O. S., Dvukhhlavov D. E. Strategies of Modularization for Android Applications. *Informatsiini tekhnolohii: nauka, tekhnika, tekhnolohiia, osvita, zdorovia. Tezy dopovidei mizhnarodnoi naukovo-praktychnoi konferentsii MicroCAD-2025 (14–17 travnia 2025 r., m.Kharkiv).* [Information Technologies: Science, Engineering, Technology, Education, Health: Proceedings of the International Sci.-Pract. Conf. MicroCAD-2025 (14–17 May 2025)]. Kharkiv: NTU "KhPI", 2025, p.1395.
11. Aigner S., Elizarov R., Isakova S., Jemerov D. *Kotlin in Action, Second Edition.* Manning Publications, 2024. 560 p.
12. Campos, E., Kulesza U., Coelho R., Bonifácio R., Mariano L. Unveiling the Architecture and Design of Android Applications. *An Exploratory Study. Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS)*, pp. 201–211. DOI: 10.5220/0005398902010211.

Received 18.11.2025

УДК 004.41

Д. Е. ДВУХГЛАВОВ, кандидат технічних наук (PhD), доцент, Національний технічний університет «Харківський політехнічний інститут», доцент кафедри програмної інженерії та інтелектуальних технологій управління, м. Харків, Україна; e-mail: dmytro.dvukhhlavov@khpi.edu.ua; ORCID: https://orcid.org/0000-0002-3361-3212
О. С. ПЕЛИПЕЦЬ, Національний технічний університет «Харківський політехнічний інститут», магістрантка, м. Харків, Україна; e-mail: pelipets.olga.developer@gmail.com; ORCID: https://orcid.org/0009-0005-5974-9045
А. С. ДВУХГЛАВОВА, Національний технічний університет «Харківський політехнічний інститут», старша викладачка програмної інженерії та інтелектуальних технологій управління, м. Харків, Україна; e-mail: alona.dvukhhlavova@khpi.edu.ua; ORCID: https://orcid.org/0000-0002-0111-3010

МОДЕЛЬ ОЦІНЮВАННЯ МОДУЛЯРИЗАЦІЇ ANDROID-ЗАСТОСУНКІВ

Актуальність дослідження, результати якого представлені, визначається тим, що мобільні застосунки еволюціонували в складні програмні системи зі зростаючими кодовими базами, що ускладнює розробку, тестування та підтримку. Показано, що покращення супроводжуваності та масштабованості проєктів Android-застосунків можливе шляхом переходу від монолітної архітектури до модульної архітектури, виходячи або з переліку функцій, які має виконувати застосунок, або з архітектурних особливостей створення застосунку. Для вибору варіанта модуляризації запропоновано класифікацію підходів до модуляризації. Незалежно від того, який напрямок реалізації модуляризації обрано, він спрямований на зменшення впливу змін в одному модулі на необхідність внесення змін до інших. Таку залежність між модулями можна оцінити, визначивши зв'язність та узгодженість проекту та окремих модулів. Для кількісної оцінки переваг модульності розроблено математичну модель, яка враховує баланс між зв'язністю модулів та цілісністю проєкту в цілому. Модель пропонує враховувати кількість модулів, на які буде розділена монолітна архітектура, рівень взаємодії між модулями, що будуть обрані, а також рівень їх залежності один від одного. Представлено вирази для автоматизації розрахунків варіантів поділу на модулі. Представлено результати оцінки модуляризації проекту Android-застосунку для електронної комерції на основі різних підходів до реалізації модуляризації. Отримані оцінки дозволили продемонструвати потенціал модуляризації у скороченні часу збирання проекту, мінімізації конфліктів та підвищенні гнучкості проекту, пропонуючи масштабоване рішення для сучасної мобільної розробки.

Ключові слова: класифікація підходів до реалізації модуляризації Android-застосунків, модель модуляризації, показники оцінки варіантів модуляризації Android-додатків, зв'язність та узгодженість проекту.

Повні імена авторів / Author's full names

Автор 1 / Author 1: Двухглавов Дмитро Едуардович / Dvukhhlavov Dmytro Eduardovych
Автор 2 / Author 2: Пелипець Ольга Сергіївна / Pelypets Olha Serhiivna
Автор 3 / Author 3: Двухглавова Альона Сергіївна / Dvukhhlavova Alona Serhiivna