Software Engineering Fundamentals to Design Application for Modern Game Engines

Viacheslav Bezditnyi[1], Olena Chebanyuk[2]

1 Igor Sikorsky Kyiv Polytechnic Institute, Beresteiskyi Ave, 37, Kyiv, 03056, Ukraine
2 National Aviation University, 1, Liubomyra Huzara ave., 03058, Kyiv, Ukraine

Abstract
The article outlines a methodology for designing multimedia applications for game engines using a component-oriented architectural style. It highlights the importance of selecting the appropriate architectural pattern, considering the features of modern game engines such as Unity, Unreal Engine, and Godot Engine. Key challenges and issues in designing flexible and scalable architectures are addressed.
The life cycle model of the user interaction session with the application is described, comprising three main stages: Bootstrap, GameLoop, and Dispose. The article emphasizes the significance of managing transient processes between life cycle states to ensure the application's proper functioning. To better organize service interactions, the mathematical framework of category theory and set theory is applied, allowing for clear definition and management of dependencies and relationships between components.
The benefits of using category theory for modeling and managing data flows and dependencies in the system, particularly through functors and monads, are discussed. These methods facilitate the creation of adaptive and scalable systems that are easy to maintain and extend.
The article provides practical recommendations for designing the architecture of game applications, considering the specifics of component-oriented and service-oriented architectural styles. It underscores the need for regular review and updates of the project architecture in response to changing requirements and operating conditions. A proposed methodology serves as a theoretical foundation for developing flexible architectures of multimedia applications, accounting for the specifics of component-oriented and service-oriented architectural styles and the functioning of game engines.
The work includes examples of practical applications of the proposed theoretical approaches in real projects, demonstrating their effectiveness and applicability in various game development contexts. These examples feature concrete implementations of patterns, state management using the State Machine, and optimization of component interactions through Service Locator and Dependency Injection.

Keywords
Game Engine, Design Patterns, Unity, Service-Oriented Architecture, Theory of Categories, Component-Oriented Architectural Style, Service locator

1. Introduction

Earlier the approaches to design graphical applications used the same designing principles as designing of web and other types of applications [1].

Designing of multimedia applications require special approaches for interface designing. Classical Model-Driven Development methodologies needed to be adopted for development of applications that need component-oriented [2], service-oriented [3, 4], agent-oriented [5], and other architectural styles [6.7].

Also modern multimedia applications support complex UI, and usual software designing approaches based on UML diagrams cannot effectively reflect all the details of multimedia applications.

Special approaches, considering the peculiarities of different GameObject structures, various types of game scenes (2D, 2.5D, and 3D scenes), and the assessment of script quality attached to

14th International Scientific and Practical Conference from Programming UkrPROG'2024, May 14-15, 2024, Kyiv, Ukraine
Corresponding author.
† These authors contributed equally.
vyachbezd@lll.kpi.ua (V. Bezditnyi); chebanyuk.olena@gmail.com (O. Chebanyuk)
0009-0002-7319-964X (V. Bezditnyi); 0000-0002-9873-6010 (O. Chebanyuk)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-3806/S_46_Bezditnyi_Chebanyuk.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

GameObjects, may be adopted for designing of multimedia applications [8]. The same problem is actual for the designing of VR? AR, and mixed reality applications [9].

The modern game industry is constantly developing, offering more and more complex and innovative solutions in the creation of game applications. One of the key aspects of effective game development is choosing an architecture that provides flexibility, scalability, and ease of code maintenance. The basis of a game application is a game engine, which is an environment for developing and prototyping game applications with its own features and programming language. Among the popular game engines, Unity Engine, Unreal Engine, and Godot Engine are most often singled out (Fig. 1) [10].

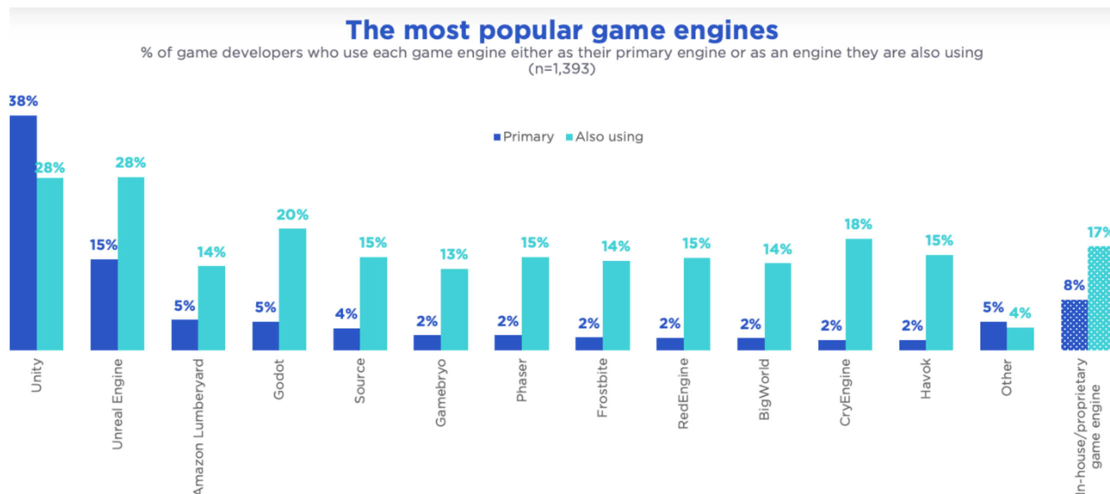

Figure 1: The most popular game engines in the world. Figure is taken from [1].

2. First Challenges and Issues in Designing Flexible Architectures for Modern Game Engines

Designing the architecture of any application can pose certain challenges and problems, in particular:

1. Choosing an appropriate architectural pattern: Game engines do not impose strict restrictions on the architecture, which can lead to difficulties in choosing the optimal pattern. Common patterns such as MVC (Model-View-Controller), MVVM (Model-View-ViewModel) or ECS (Entity Component System) have their advantages and disadvantages depending on the project [11].
2. Initial architectural decisions can make it difficult to scale the project in the future. The wrong choice can lead to the need to rewrite the code when adding new functions [12].
3. Using the available functionality of game engines without a designed architecture and without using best practices leads to the creation of tightly coupled systems, which complicates their testing and refactoring [13]. Using SOLID principles and design patterns such as Dependency Injection, Service Locator can help manage these dependencies.
4. An architecture that complicates testing can significantly increase development time. Automated testing such as unit tests can be difficult to implement in Unity due to game engine dependencies [6].
5. Some architectural patterns can negatively affect performance, especially in projects with a large number of objects and components. Optimization and proper pattern selection are key to maintaining high performance.

Integration with external systems such as databases or web services can be difficult depending on the **chosen** architecture. It is important to plan these aspects in the early stages of development.

Different developers may have different approaches to architecture, which can make team collaboration difficult. It is important to establish clear standards and principles of architecture at the beginning of the project.

To solve these problems, it is important to spend time planning the architecture, regularly reviewing and updating it according to changes in the design, and taking into account the experiences and best practices of other developers.

Today, there are a large number of patterns and implementations of their combinations. In order to choose the most suitable architecture, it is suggested to consider what challenges developers have to deal with.

The first is getting dependencies from another class. This is usually solved by setting up relationships between components in the development environment, using the Singleton pattern, static fields, or events [14]. The disadvantages of this approach become noticeable in the process of filling the project with objects and logic. An alternative is Dependency Injection (DI), which allows you to pass a reference to an object through a dependency injection mechanism [15].

Another challenge is maintaining the structure of the project, minimizing problems with the initialization order. For this, you need to clearly control the life cycle of the application, as well as game objects. The State pattern is suitable for defining life cycle stages, since each stage is a separate state of the application that we can enter and exit.

In addition to dependencies, developers often face the question of how to use the same functionality in different parts of the project? If the logic is global for all users and 100 users make requests to one service at the same time, then at best only a few will get the result. This problem is solved by the Service Locator pattern in combination with the State pattern to ensure sequential execution of requests in the order determined by the state [10, 11, 6].

Let's take a closer look at the states and description of the game cycle of the application.

3. Technologies Enhancing Gaming Application Functionality

For any application, such as an educational app or a game, the session life cycle typically follows this structure (see Figure 2):

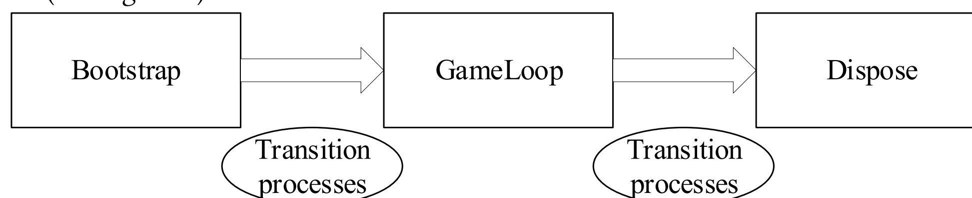

Figure 2: Model of the user interaction session life cycle with the application.

Bootstrap is an entry point, a place where services, dependencies are initialized, resources necessary for launch are loaded.

GameLoop is directly the game process, interaction with the application, in which all game or business logic takes place.

Dispose is the state of the application before exiting, unloading resources, services, etc.

The most critical points in this scheme (Fig. 2) are between the states of the life cycle and can be called "transitional processes". This means, for example, the order of initialization of services, verification of obtaining dependencies, the order of unloading resources from memory. If these processes are left unchecked, we get incorrect program execution or even errors that block the execution process.

For the correct operation of the application, we need to control these states, have a convenient mechanism for transition between them, have an initial entry point in which the necessary dependencies are initialized in a controlled manner using services. And access to the services themselves is implemented using Dependency Injection.

4. A formal model of the gaming process, utilizing discrete mathematics principles

In order to interpret the game cycle into a mathematical model, it is possible to use the Theory of Category. Consider the main analogies:

- Objects and Morphisms: In category theory, systems and their interactions can be modeled using objects (components, services) and morphisms (functions that describe interactions between these objects). In the context of Unity, this can be used to define the relationships between various game components, such as the Service Locator, DI, Factory, and State Machine.
- Functors and Monads: Functors (structures that map objects and morphisms of one category to another) and monads (a type of functor that allows sequentially combining operations) can be used to model and manage data flows and dependencies in a system.
- Sets and Subsets: Game elements (objects, components) can be represented as sets, and their properties and characteristics as subsets. This allows you to clearly define and manage dependencies and relationships between components.
- Operations on Sets: Operations such as union, intersection, and difference of sets can be used to describe interactions between components. For example, combining sets can represent the integration of different services into one system.
  Application in Unity:
- Service Locator: This can be represented as a centralized system responsible for tracking and providing access to various services (objects), using the principles of category theory to manage dependencies.
- Dependency Injection: This can be interpreted through set theory, where dependencies between objects are represented as relations between sets.
- Factory Pattern: Used to create objects, which can be viewed through the lens of category theory, where a factory is a functor that maps parameters to objects.
- State Machine: This can be modeled using set theory, where states are represented as sets and transitions are represented as relationships between them.
- Extended Application and Benefits:
- Scene Management: Category theory can manage scenes and transitions in Unity, treating each scene as an object and transitions as morphisms.
- AI Systems: AI behaviors and decision-making processes can be modeled using functors and monads to manage state and behavior transitions.
- Performance Optimization: By formalizing interactions and dependencies, category theory can help identify and optimize performance bottlenecks.
- Scalability and Modularity: These theoretical approaches promote modularity and scalability, making it easier to update, test, and maintain the system.
- Real-World Examples: For instance, consider a Unity project where the service locator pattern is used to manage audio, saving/loading, and networking services. Each service is an object, and interactions between services are morphisms. Using dependency injection, the audio service can be easily replaced without modifying the networking code.

Challenges and Considerations:

Complexity: Implementing these concepts can be complex and may require a steep learning curve for developers unfamiliar with category theory.

Tools and Libraries: Utilize libraries and tools that support functional programming and dependency injection to facilitate the implementation of these concepts.

Integration: Ensure these theoretical approaches integrate well with Unity's existing architecture and components.

By combining the theoretical approaches listed above, it is possible to develop a flexible and scalable system for Unity that optimizes the management of dependencies and data flows in complex projects. The formalism provided by category theory and set theory can help in designing robust and maintainable game architectures, enhancing both development and performance.

5. Model of Game Application Functionality

States and state management are typically implemented using the Game State Machine pattern.

A "State Machine" is a mathematical model used to describe the behavior of a system. It consists of a set of states, transitions and actions. A state represents a specific behavior or condition of a system, while a transition defines movement from one state to another. Actions associated with states or transitions represent the logic to be performed upon entering, exiting, or staying in a state [9].

Figure 3 shows the sequence of launching a software application (in particular, a game).

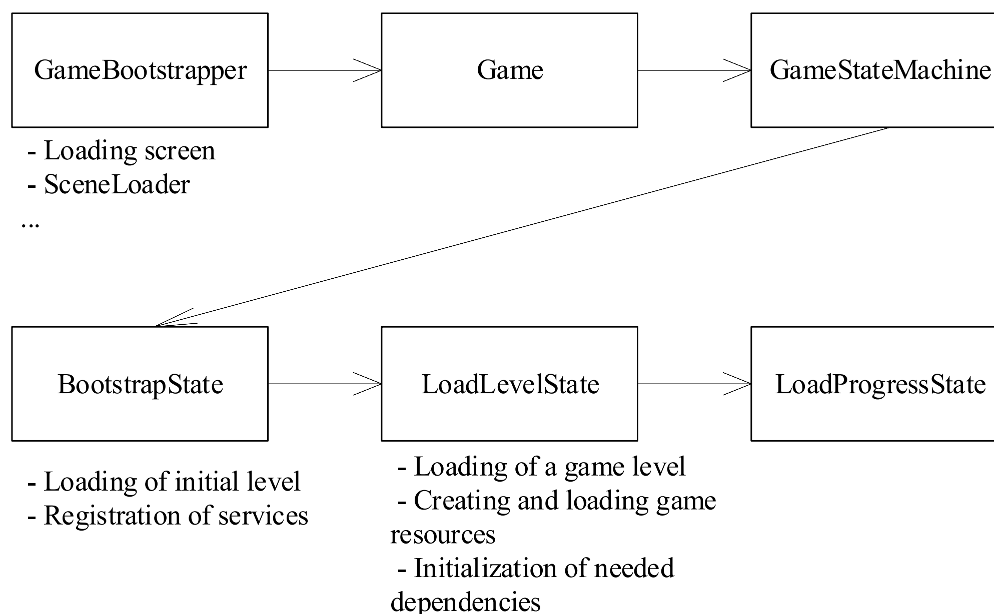

Figure 3. Sequence of Actions to Launch an Application.

The first step in implementing the State Machine is to define the states. Each state will be represented by a separate script (logic) that inherits from the IState interface, which declares 2 main methods: entering a certain state and exiting it.

The next step is to create a GameStateMachine class to manage state transitions and perform appropriate actions.

The GameStateMachine class can have different states (for example: \`BootstrapState\`, \`LoadLevelState\`, \`LoadProgressState\`, \`GameLoopState\`), which can be treated as objects in a category. The transition between states through the Enter() and ChangeState() methods can be analogous to morphisms that transform one object (state) into another.

One of the key aspects of category theory is the composition of morphisms, which in this case can be represented as a series of transitions between states. For example, going from

`BootstrapState` to `LoadLevelState` and then to `GameLoopState`. The composition of these transitions (morphisms) creates a "path" through different states of the game.

In category theory, functors are mappings between categories. One can consider the interaction between different components of the system (for example, between `GameStateMachine` and specific states) as a functor-like relationship, where the behavior of one component determines the behavior of another.

A system of states can be considered as a category where states are objects and transitions between them are morphisms. Within a larger system (for example, the entire game), such a state system can be considered a subcategory.

In category theory, each object has an identical morphism that reflects the object in itself. This can be implemented as the ability of a state to remain unchanged if there is no transition to another state.

Consider the following formula (1), which describes the transition from BootstrapState to GameLoopState via LoadLevelState:

$$LLS = BS --> LPS --> GLS\,, \tag{1}$$

This equation can be interpreted as a composition of two morphisms:

1. BootstrapState->LoadProgressState. This morphism corresponds to the transition from BootstrapState to LoadProgressState.
2. LoadProgressState->GameLoopState This morphism corresponds to the transition from LoadProgressState to GameLoopState.

The composition of these two morphisms gives the BootstrapState -> GameLoopState morphism, which corresponds to a direct transition from BootstrapState to GameLoopState.

This example demonstrates how category theory can be used to describe and understand transitions between states in a system of game states.

This mathematical model is a simplified representation of the system of game states. A real system can be much more complex, with more states, transitions, and interactions.

Category theory provides a powerful toolkit for modeling and analyzing systems of states, but understanding and applying its concepts requires some knowledge of abstract mathematics.

6. Architecting Game Applications with Category Theory

Designing the structure of a service-oriented application (SOA, Service-Oriented Architecture) in Unity requires an understanding of both the basic principles of SOA and the specifics of Unity as a game engine. The main idea of SOA is to create modular, independent services that can be easily replaced, updated, or modified without affecting other parts of the system [10].

SOA defines a way to make software components reusable and interoperable through service interfaces. Services use common interface standards and an architectural pattern so that they can be quickly integrated into new applications. This relieves the task of the application developer who previously redesigned or duplicated existing functionality or had to know how to connect or ensure compatibility with existing functionality.

Each service in an SOA contains the code and data needed to perform a complete, discrete business function (such as checking a customer's creditworthiness, calculating a monthly loan payment, or processing a mortgage application). Service interfaces provide free communication, that is, they can be called almost without knowing how the service under them is implemented, which reduces the dependency between applications.

Below are the key considerations for SOA design in Unity:

- Definition of Services: Functional elements of the application can be separated as independent services. These can be, for example, the game save system, sound management, network interactions, advertising management, etc.
- Service Interfaces: It is necessary to clearly define the interfaces for each service. This will allow replacing service implementations without the need to change the code that uses these services.
- Dependency Checking and Dependency Injection: Using the Dependency Injection pattern will help manage dependencies between different services and components. This can be implemented through constructors, setters, or through special frameworks (Zenject [15], VContainer [16]).
- Projecting to the Service Manager: To manage services, it is necessary to create a central service manager (Service Locator), which will be responsible for initialization, storage, and access to various services in the application.
- Design of Modular and Flexible Architecture: Each service must be designed so that it is self-sufficient and can function independently of other parts of the system. This will provide high flexibility and simplify testing and project development.
- Deployment and Updating of Services: Methods of deployment and updating of individual services should be able to update services without stopping the entire system.
- Testing: Each service is designed taking into account the possibility of its testing. Automated testing is a key element in maintaining high code quality and system stability.
- In Unity, SOA can be implemented both with the help of built-in tools (for example, through MonoBehaviour components) and with the help of external libraries or frameworks. The main thing is to keep the focus on clearly defining the roles and responsibilities of each service, as well as on the flexibility and extensibility of the architecture.

Additional Considerations for SOA Design in Unity:

- Performance Optimization: It's crucial to consider performance optimization when designing SOA in Unity. Profiling tools like Unity Profiler can help identify bottlenecks and ensure efficient memory and resource management.
- Security: Secure communication between services, especially for network interactions, is essential. Implementing proper authentication and authorization mechanisms will help protect sensitive data.
- Real-World Examples: Practical examples of SOA in Unity, such as case studies of successful implementations, can provide valuable insights and guidance. For instance, discussing a game that effectively uses SOA principles can illustrate the benefits and challenges faced during development.
- Cloud Integration: Leveraging cloud services like AWS or Azure can enhance the scalability and flexibility of Unity applications. Cloud services can be used for hosting, data storage, and processing, enabling seamless scaling of services.
- Service Versioning: Maintaining backward compatibility and managing updates through service versioning is vital to ensure that new updates do not disrupt existing functionality.
- By incorporating these additional considerations, you can create a more comprehensive and robust guide for designing service-oriented applications in Unity.

7. Creating test app based on recommended approach

First of all, we need to create two scenes: InitialScene – where we will manage the loading of the app, resolving dependencies, taking saves, and other preparing actions; MainScene (it can be level, menu, lobby scene whatever that can be after loading scene).

The next step is to set up the DI container. In our case for Unity, we need to import the Zenject package. After importing we can set up Project Context, and Installers for every scene. This allows as then bind needed components and inject dependencies where we need them. Also, we need to add a SceneContext component for every scene and create and assign installers to this context. Inside these installers we can bind the Service locator to have access from any place, only need to define it with the [Inject] attribute.

Based on the previous explanations about categories Scenes are also objects and switching between them is a morphism. Switching contains a few steps (states) that we manage by StateMachine. That is why the state itself needs to contain two main methods (Fig. 4) Enter() and Exit() something can happen when we go to the next state and if we exit we first need to clear memory, unsubscribe, and dispose. Sometimes we need additionally to have the ability to loop or update something inside the script for these purposes we can use other interfaces to extend our functionality.

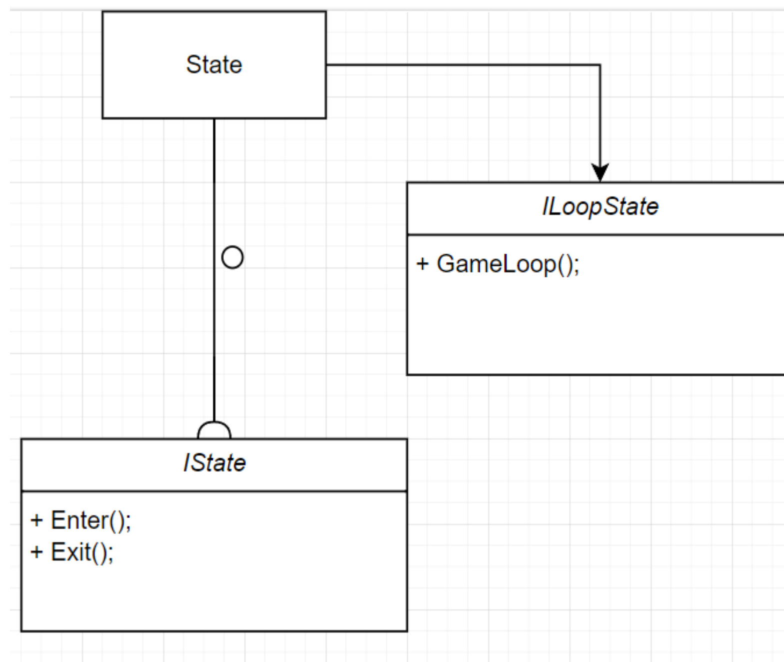

Figure 4. UML Class Diagram to represent States.

We defined states [17] and separated needed logic for loading and play sessions (Fig. 5). In the Bootstrap state we register all our services. Then go to the next scene through LoadLevelState. Usually, we have some stored data on the client or server, for this purpose, we can use LoadProgressState and load if exists or create new data if needed. To store data locally or on the server we can create a SaveData service and register it in the BootstrapState. Then every needed logic we as a service and use it from any place.

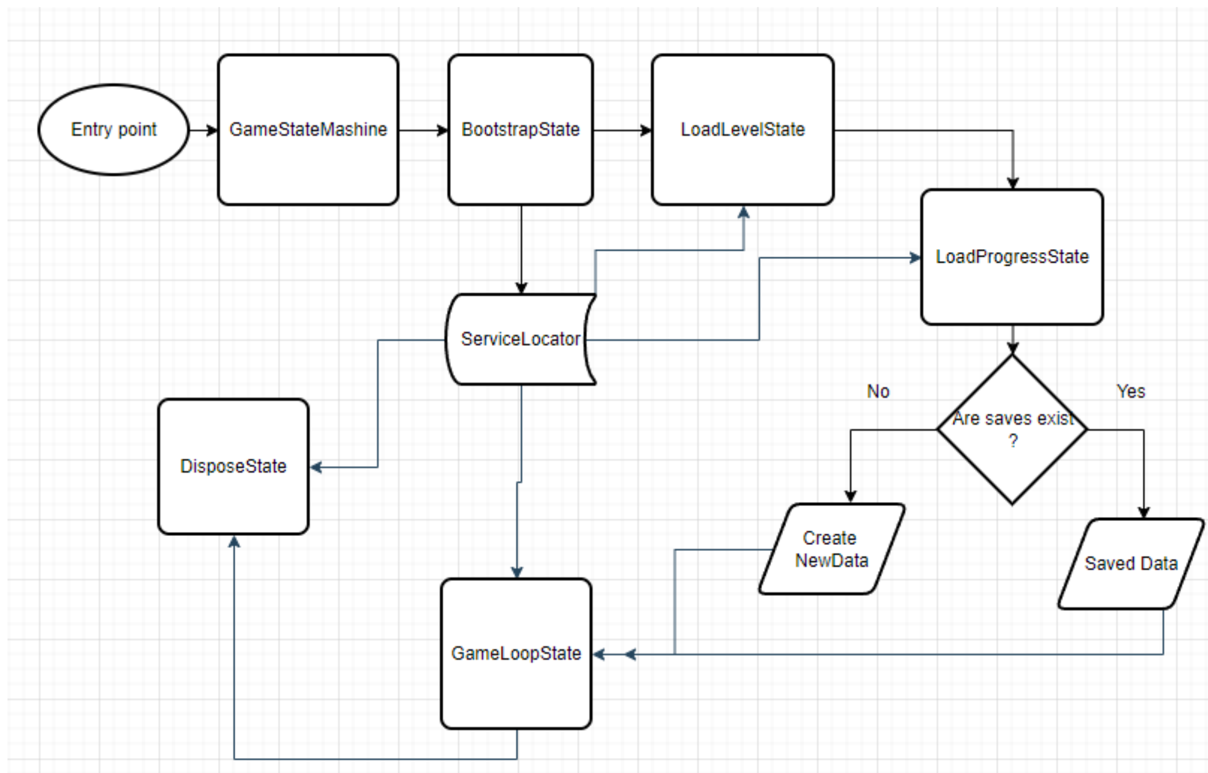

Figure 5. Diagram of applications' states.

8. Conclusions

In the course of the study, the key provisions of the theory of categories were analyzed, which became the theoretical basis for the development of the application design methodology.

It was proposed to interpret the life cycle of the application as a set of states of the program, which go from one to another with the help of the "State Machine". The composition of these states can be represented as a composition of morphisms, a mapping of one state into another, which follows from the theory of categories.

The described interpretation enables a controlled scaling of the project architecture and improves the understanding of the execution of processes and logic in the application. New logic we can represent as a service and need only register it to use somewhere. This type of architecture allows us in a comfortable and faster way extend or change our logic, only by replacing the service. The next step for improvement will be to manage the realization of the services interface for example by Strategy pattern.

The proposed technique is a theoretical foundation for the development of flexible architectures of multimedia applications. It takes into account the specifics of component-oriented and service-oriented architectural styles, as well as the peculiarities of the functioning of game engines.

References

[1] Mavrevski, R., Traykov, M. and Trenchev, I., 2019. Finding the shortest path in a graph and its visualization using C# and WPF. International Journal of Electrical and Computer Engineering (IJECE), 10(2), pp.2054-2059. DOI: 10.11591/ijece.v10i2.pp2054-2059.

[2] Litvin, A., Velychko, V. and Kaverinsky, V., 2020. Method of information obtaining from ontology on the basis of a natural language phrase analysis, in: CEUR Workshop Proceedings, CEUR-WS, Kyiv, Ukraine, 2020: pp. 323–330.
[3] Kulibaba, S., Popereshnyak, S., Shcheblanin, Y., Kurchenko, O., Mazur, N., 2022. Advanced communication model with the voice control and the increased security level. In: Cybersecurity Providing in Information and Telecommunication Systems, October 13, 2022, (CPITS-2022) Kyiv, Ukraine. CEUR Workshop Proceedings, 3288, pp.64-72.
[4] IBM, no date. What is service-oriented architecture (SOA)?. Available from: https://www.ibm.com/topics/soa [Viewed 3 April 2024].
[5] Popereshnyak, S., Yurchuk, I., 2021. Social Networks: Analysis, Algorithms and Their Implementation. In: 5th International Conference on Computational Linguistics and Intelligent Systems, April 22–23, 2021, Kharkiv, Ukraine. CEUR Workshop Proceedings, 2870, pp.811-821.
[6] 2023. Mobile game development using Unity engine. Methodologies and intelligent systems for technology enhanced learning. In: Workshops - 13th International Conference, pp.129-138. Available from: https://www.researchgate.net/publication/373475694_Mobile_Game_Development_Using_Unity_Engine [Accessed 2 April 2024].
[7] Kaverinsky, V., & Malakhov, K. 2023. Natural Language-Driven Dialogue Systems for Support in Physical Medicine and Rehabilitation. South African Computer Journal, 35(2). https://doi.org/10.18489/sacj.v35i2.17444
[8] Chebanyuk, O., 2023. Approach to reuse of Unity game scenes. In: 2023 International Conference on Advanced Enterprise Information System (AEIS), London, United Kingdom, 2023. pp.11-15. ISBN: 979-8-3503-5926-8. DOI: 10.1109/AEIS61544.2023.00009.
[9] Kolev, M., Trenchev, I., Traykov, M., Mavrevski, R. and Ivanov, I., 2023. The impact of virtual and augmented reality on the development of motor skills and coordination in children with special educational needs. In: Zlateva, T. and Tuparov, G. (eds) 19th EAI International Conference on Computer Science and Education in Computer Science, CSECS 2023, Boston, United States, June 28-29, 2023. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (LNICST), vol. 514. Springer Nature Switzerland, pp.1-13.
[10] Kishor, K., Rani, R. and Rai, A.K., 2023. 3D application development using Unity real-time platform. Proceedings of Fourth Doctoral Symposium on Computational Intelligence, pp.665-675.
[11] Mosler, P. et al., 2023. Using the game engine Unity efficiently in teaching: Development of a fully-automated webserver-based build pipeline. In: eCAADe 2023: Digital Design Reconsidered, Graz, Austria, 20-22 September 2023. Available from: https://doi.org/10.52842/conf.ecaade.2023.2.883 [Accessed 5 April 2024].
[12] A. Litvin, V. Velychko, and V. Kaverinsky. A New Approach to Automatic Ontology Generation from the Natural Language Texts with Complex Inflection Structures in the Dialogue Systems Development in: CEUR Workshop Proceedings, CEUR-WS, Kyiv, Ukraine, 2022: pp. 172-185
[13] Boiarskyi, O., Popereshnyak, S., 2022. Automated System and Domain-Specific Language for Medical Data Collection and Processing. In: Babichev, S., Lytvynenko, V. (eds) Lecture Notes in Computational Intelligence and Decision Making. ISDMCI 2021. Lecture Notes on Data Engineering and Communications Technologies, vol 77. Springer, Cham. https://doi.org/10.1007/978-3-030-82014-5_25
[14] Uzayr, S. bin, 2022. Mastering Unity: A Beginner's Guide (Mastering Computer Science). 1st ed. CRC Press.

[15] GitHub, no date. GitHub - modesttree/Zenject: Dependency injection framework for Unity3D. Available from: https://github.com/modesttree/Zenject [Accessed 3 April 2024].

[16] VContainer, no date. About | VContainer. Available from: https://vcontainer.hadashikick.jp/ [Accessed 2 April 2024].

[17] ServiceLocatorArchitecturePresentation, no date. Available from: https://youtu.be/nGHrLzqlVWg [Accessed 3 April 2024].