Abstract. The present article deals with the main methods and models of automatic software systems development. The main types of implementations of automata abstract models are determined and analyzed. A new option of creating a software system module that is responsible for the behavior of an object, which can change depending on its current state, is proposed. The diagram and main components of this module are presented.

Keywords: automata-based programming, state machine, state transition, event manager, output signal, design pattern.

Introductions. In today's world, most software products are highly loaded, distributed, scalable systems. The development of any program, whether a small procedure for information processing or a comprehensive software product, consists of several stages, the proper implementation of which is a prerequisite for achieving a successful result.

In software development, the fundamental stage is design - complete planning of what will have to be developed, in what timeframe, with what source data and the expected result.

A feature of modern software projects is the use during the development of the refactoring process [1] - regular restructuring of the source code of the program (without changing the functionality) to facilitate support for changes and improve
understanding of the code. A large number of patterns and techniques are used to perform refactoring, which describe ways to improve the code.

**Aim.** The aim of this work is to analyze software solutions, algorithms, design patterns based on finite state machines to identify existing shortcomings, and their modernization to optimize the design and implementation of software systems.

**Materials and methods.** The following methods were used in the work:
- methods of automata theory [2];
- methods of discrete mathematics;
- design patterns of object-oriented programs;
- methods of construction and analysis of algorithms [3];
- abstraction (highlighting only the basic properties of the software system; abstraction from hardware interaction when performing I / O operations);
- formalization (representation of a finite state machine in the form of a set of elements).

The "State" design pattern was used as an object-oriented representation of the finite state machine [4].

**Results and discussion.** Abstract finite state machine can be represented as a mathematical scheme $G$, characterized by six elements: a finite set of $I$ input signals; a finite set $O$ of output signals; finite set $S$ of internal states; initial state $s_0$; transition function $x(s, i)$; output function $y(s, i)$ [5].

The operation of the finite state machine is as follows: on each $n$-th cycle on the input of the machine, which is in state $S(n)$, a signal $I(n)$ is given, to which the machine responds by switching to $(n + 1)$-th cycle in new state $S(n + 1)$ and the issuance of some output signal $O(n)$. In general, the operation of a finite state machine can be described by the following formulas [6]:

$$
\begin{align*}
S(n + 1) &= x[S(n), I(n)], n = 1, 2, \ldots; \\
O(n) &= y[S(n), I(n)], n = 1, 2, \ldots;
\end{align*}
$$

An example of a finite state machine graph is shown in the figure 1.

The projection of a finite state machine in the world of object-oriented design is the pattern “State” [7], which allows objects to change their behavior depending on their state. From the outside it seems that the class of the object has changed. This
The “State” pattern is used when designing modules (classes) that have a certain set of internal states and, depending on the current state, perform different operations while providing the same interaction interface [8].

But like any other, this pattern has some drawbacks:

− division of the implementation of states by different classes leads to the distribution of the logic of transitions between them;
− the independence of the classes that implement the states from each other is not ensured;
− creating a hierarchy of state classes and their reuse is difficult.

In order to optimize the use of this pattern, it must be modified, which will get rid of the described shortcomings and solve the following problems:

− impossibility of reusing state code;
− decentralization of the logic of transitions between the states of the object;
− dependence of object-oriented classes of states on each other.

The main idea of the developed modification of the pattern "State" is to separate the class responsible for transitions (change) between states (state manager can be a completely separate class or part of the interface representing the main class of the
CURRENT ISSUES AND PROSPECTS FOR THE DEVELOPMENT OF SCIENTIFIC RESEARCH

object) and classes that represent specific states, to achieve their full autonomy and independence [9].

To change the current state of an object, state classes pass event objects to the state manager.

The class diagram of the modified version of the “State” pattern is shown in the figure 2 [10].

![Modified pattern class diagram](image)

Fig. 2. Modified pattern class diagram

One of the differences between this modification and the "State" pattern is the way to change the active state. If in the initial pattern the state class is responsible for specifying the next state, then in this modification this class only sends the event object to the state manager, which is making the necessary conclusions about their possible state change.

The advantage of this modification is that the classes that implement the states may know nothing about each other, because the choice of the next state is made by the state manager depending on the current conditions.

Conclusions. In the process of research, a design pattern was developed that
allows objects to change their behavior depending on their condition, while eliminating the main shortcomings of the existing pattern "State".

The practical significance of the work is that the results simplify the development and maintenance of software systems and can be used in practice when devising and designing their behavior.

The proposed pattern allows you to design those modules of the system that have a state and can change it during operation, while ensuring the reuse of state classes, independence between them and the centralization of the logic of transitions between states.

References: