Event-Driven Programming as a Way to Unify Control Statements

Alexander Prutzkow

***Abstract*—Event-driven programming (EDP) is a programming paradigm in which actions are performed after notification of an event. The paradigm can be used to unify control statements (as in Markov normal algorithms). Unification consists in using only a linear sequence of commands when programming events and their handlers. We introduce a new look at EDP. Event-driven architecture (EDA) for our look includes a framework, events and their handlers. The framework consists of an event queue and its dispatcher, event loop handlers as threads. EDA has the following features: one input event is transformed into one output event, one event is processed by one handler and vice versa, subscription to an event by its type, handler searching by event parameters, queue with a broker, data exchange between handlers through events. The event is characterized by a type, input data, a status, the parent event, a processing result. The handler processes the event, filters processed events, and provides a reference to the subscribed event type. The resolving handler transforms the initiated event into a handled one. The transformation handler initiates a new event in place of the handled one. EDA is thread-safe due to storing state only in events, stateless handlers, using a thread-safe queue, sequentially processing events one after another. EDA was programmed in Java. The program includes 3 loop statements (including 1 in the thread pool) and 4 conditional statements. All these statements are in the framework. Only a linear sequence of commands is used in events and handlers.**

***Keywords*—Event, event-driven programming, event-driven architecture, event handler, event loop, Markov normal algorithms, Java, multithreading.**

I. Could Control Statements Be Unified? (Introduction & Motivation)

Markov normal algorithm [1] is an algorithmic model. The normal algorithm is executed on a word R consisting of alphabet A symbols in the following order. The normal scheme of the algorithm is scanned from top to bottom, and a substitution is selected from it, the left part P of which is contained in the word R. If the substitution is found, then the first occurrence of the substring P in the word R is replaced by the substring Q, and the scan of the scheme begins again.

A substitution is a single statement of the Markov normal algorithm. Conditions and loops are taken out in the order of execution.

Could imperative control statements be unified?

Manuscript received January 15, 2024.

A. Prutzkow is with the Ryazan State Radio Engineering University, 390005, Gagarin str., 59/1, Ryazan, Russia, and with Lipetsk State Pedagogical University, 398020, Lenin str., 42, Lipetsk, Russia (e-mail: mail@prutzkow.com).

II. The Purpose of the Study

The purpose of the study is to simplify the program structure by unifying the control statements used. At this stage of the study, we realistically consider the use of only a linear sequence of commands as unification of statements.

We'll use event-driven programming (EDP) to achieve the purpose.

We state the study in original manner. The section (except this one) headers are questions, the section texts are answers.

III. What Is Event-Driven Programming?

A. What Are Types of Message?

Class objects, program modules, and network nodes interact through messages of three types (table 1, adapted from [2]):

- commands – requests to perform operations;
- events – notifications about actions that have occurred;
- queries – requests for data.

Messages are specific within a particular system or domain.

Table 1. Message types and their features

Message Type	Behavior / state change	Response
Command	Requested to happen	Result or status
Event	Just happened	Never
Query	None	Data

B. How to Define Event-Driven Programming?

EDP is a programming paradigm in which one or more software components execute in response to receiving an event notifications (adapted from [3]).

EDP can be considered as an architectural style [3]. An architectural style is a specialization of element and relation types, together with a set of constraints on how they can be used for a family of architectural instances (combined from [4, 5]).

C. Is There Event-Driven Architecture?

EDP is a base of event-driven architecture (EDA). Depending on the task being solved, EDA has variations [6, 7, 8, 9, 10].

In addition to the event, there are mandatory elements in any variation:

- event source;
- event transmission channel / event listener;
- event handler.

A source initiates an event. The event is transmitted over a channel or the source notifies a listener of the event. The transmission channel is the event queue. The transmission channel or the listener passes the event to the handler that

processes it. Receiving events from sources and passing them to handlers constitutes the event loop.

EDA has the following attributes (adapted from [11]):

- the state of the software system changing after event processing;
- implicit sequence of command to solve the problem (also in [12]);
- loosely coupling (also in [3, 13, 14]); sources and handlers are only associated with the event transmission channel / event listener; sources and handlers are not coupled to each other; sources and handlers can be added independently of other elements [15];
- decentralized control; multiple sources can trigger events (also in [13]).

EDA is discussed in detail in [3, 14, 16].

D. When is Event-Driven Programming Used?

There are the following use cases of EDP:

- processing user actions in the graphical interface [17, 18];
- microservice architecture [19, 20];
- implementation of the Observer pattern [21] (and the MVC pattern [18]);
- software libraries: parsing XML files using Simple API for XML (SAX) [22], data exchange over the network using Twisted [23].

The reasons for using EDP are the following (adapted from [3]):

- the system already has events;
- the operation of the system can be reduced to processing a sequence of events in a natural way (for example, processing requests by a web server);
- the system must be expandable, scalable, with weak coupling of modules.

E. Where Did Event Driven Programming Come from?

We didn't identify a origin for EDP clearly. There are three versions of its origin (table 2).

TABLE 2. VERSIONS OF THE ORIGIN OF EVENT-DRIVEN PROGRAMMING

Author(s) of version	First publication on event-driven programming
Faison T. [16]	Krasner G. and Pope S., 1988 [24]
Ferg S. [25]	Yourdon E. and Constantine L., 1978 [26]
Tucker A. and Noonan R. [18]	Stein L.A., 1999 [27]

F. What Else Do We Know about Event-Driven Programming?

Concepts of EDP could be defined by temporal logic [14, 28]. The standard implementation with callbacks can be explained in temporal sense $\Diamond A$ as $\overline{\Box \overline{A}}$ [28].

EDP can be combined with genetic programming [29]. Both events and genetic functions are labeled with evolvable tags. When an event arises, the function with the closest matching tag is triggered.

Analysis of event-based style variations showed that middleware infrastructures (not necessarily event-based) implicitly define architectural (sub)styles [15].

Event handlers (called event procedures) have very low reusability [30]. There is no simple way to pass data between handlers. The diagram of the relationships between events, handlers, and the general procedures they call does not provide a clear understanding of the solution to the problem. Instead of an event loop, the callback or inverse control programming paradigm could be used, as well as Java language synchronization [31].

The TaskJava backward-compatible extension to Java transforms program fragments into a switch statement. Each case of this statement is an event handler [32]. The scope of the extension is a method, not the entire program. The EDP language Jadescript is designed to implement agents in multi-agent systems [33].

Events are used in programming languages with multithreading. Tame introduces four related abstractions for handling concurrency [34]. Event is one of them. Eve is a programming language with shared memory within the event loop [35]. AmbientTalk is a programming language to compose objects as services in mobile networks [36]. Events are messages. The WS-BPEL language in service-oriented architecture was extended by EDA concepts [37]. SUNNY is a model-based programming paradigm for designing and developing interactive event-driven systems [38]. The paradigm is structured around models of data, network, event, etc. Protothread is a programming abstraction to write an event-driven program in thread-like manner [39]. An extension of OpenMP combines asynchronization and parallelization [40]. The extension facilitates the development of event-driven programs, especially for GUI applications, to achieve better responsiveness and event handling acceleration.

P is a domain-specific language for writing asynchronous event-driven programs [41]. State machines declared in a program communicate with each other asynchronously through events. Depending on the state of the machine, events that transit the machine to another state are ignored, or are not removed from the event queue. P was a tool to design the USB Hub in Windows 8.

In the Rhapsody tool events change the state of class objects [42]. Objects are linked. When the state changes, a step is executed. A step is composed of microsteps. There is an IDE to visualize objects, their states and relationships with other objects.

EDP can be used for packet processing in network devices and in software-defined networking. EDP is an approach to write correct and efficient programs for distributed systems [43]. All network algorithms are event-driven [44].

Testing event-driven programs requires specific approaches to writing unit tests [45].

EDP is criticized. Threads are no worse than for events for high concurrency servers [46]. Threads, when programmed correctly, have good performance and don't limit control flow, are not difficult to synchronize.

The results of these studies will be used in our new look at EDP.

IV. WHAT DOES A NEW LOOK AT EVENT DRIVEN PROGRAMMING CONSIST IN?

We introduce a new look at EDP.

EDA has following structure (fig. 1).

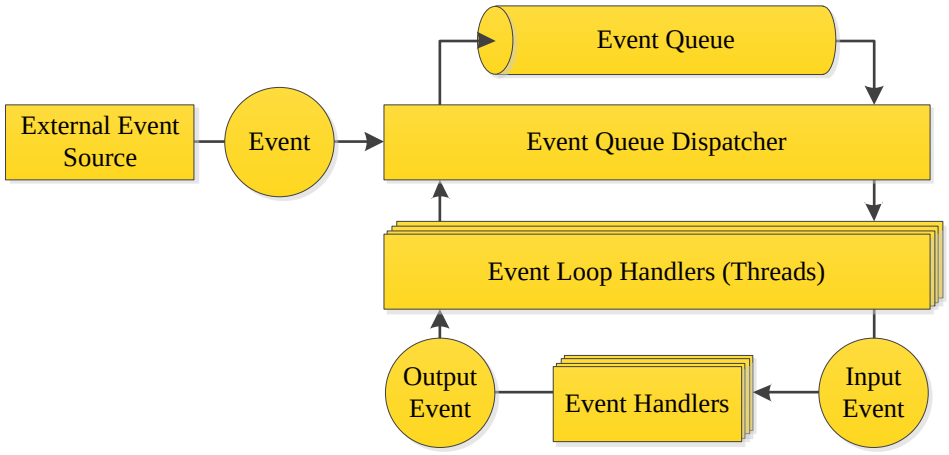

Fig. 1. A structure of the new look event-driven architecture

Events are stored in a queue. Events are pushed and pulled by a queue dispatcher. The event loop is implemented in event loop handlers. Event loop handlers are threads.

EDA consists of:

- framework (the event queue, the event queue dispatcher, the event loop handlers, the abstract event and the abstract event handler);
- events and their handlers.

The sources of events fall into two types:

(1) external sources;
(2) transformation event handlers (defined further).

Events are processed by handlers.

External sources interact with the event queue dispatcher. Events from external sources may require an adapter to cast the event to a known type.

The look has the following features:

- one input event is transformed into one output event;
- one event is processed by one handler and vice versa;
- subscription to an event by its type;
- searching a handler by event parameters;
- queue with a broker (according to the classification in [47]);
- data exchange between handlers via events.

V. What is an Event?

A. What characterizes an event?

The event has the following properties:

- type;
- input data (processing data);
- status of processing;
- parent event;
- result of processing.

B. Why is the Event Type Introduced?

The event type is required to find its handler.

C. When is the Event Status Utilized?

Handlers are founded depending on the event status. If the event is initiated, a resolving handler is founded (see Section VI). If the event is handled, a transformation handler is founded.

D. Why is the Event Processing Result Introduced?

In a sequence of events, the result of processing one event influences the following events. The influence manifests itself in two forms:

- depending on the result, a new event is initiated;
- the result of the event is used as input for the next event.

E. Are the Parent Event and Event Hierarchy Necessary?

The sequence of events can be duplicated, for example $ABCABC$. Changing the sequence entails modification of all its duplicates that increases time complexity of maintenance. Duplication can be eliminated by introducing composite events. A composite event is a sequence of events. A composite event can be part of another event. Events form an inclusion hierarchy: from simple events to composite ones. The hierarchy is implemented through a reference to the parent event.

The top-level event of the hierarchy has no a parent event. In this case, the reference to the parent event is NULL_EVENT.

F. Why is the Event Phase at All?

Let a composite event is a sequence of events $ABAC$. To determine what event should be initiated after the event A is processed, an event phase is introduced. Let the sequence of events AB form phase 1, and the sequence of events AC form phase 2. Then in phase 1, the event B is initiated after the event A, and in phase 2, the event C is initiated after the event A.

A phase isn't a state of an event. A composite event could have the same state in different phases. Phases are sequential. An event is in a phase only once.

VI. Who Does Process an Event?

Handlers are in charge of event processing.

Handler is either:

- resolving – processing events and convert the event from the Initiated status to the Handled status (fig. 2) or;
- transformation – creating and initiating a new event based on the results of event processing (fig. 3).

Resolving handlers subscribe to the event itself, while transformation handlers subscribe to the parent event. This reduces the number of subscribers for lower-level events in the hierarchy.

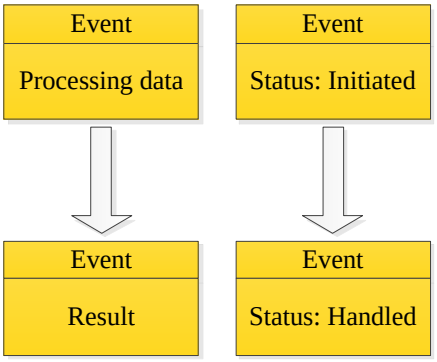

Fig. 2. Changing
event properties by the resolving handler

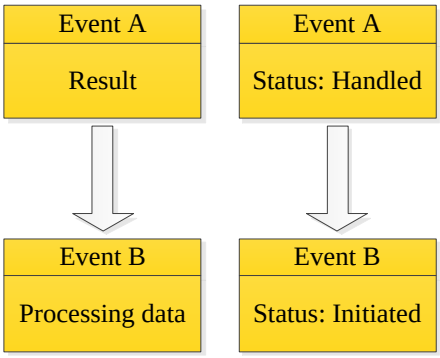

Fig. 3. Creation of an event by the transformation handler
from a handled event

VII. What Event Properties Are Exhaustive for an Handler?

A handler has a filter for the events that it can process. The filter uses the following event properties:

- event type;
- type of parent event;
- event phase;
- event status;
- result of event processing.

VIII. Are Event-Driven Programming and Multithreading Compatible?

Events can be processed multithreaded. Threads implement the event loop.

The introduced look at EDP is thread-safe for the following reasons:

- using a thread-safe queue (*java.util.concurrent.LinkedBlockingQueue* of the Java programming language) as the event queue;
- storing the state of the problem solution in events; at each moment of the program's operation, the event is processed by one handler that eliminates the need for synchronization;
- handlers are stateless;
- the sequence of events is processed one after another; this excludes the processing of the next event before the previous one;
- events are immutable; if an event is processed, a copy of it is created with the Handled status; when the phase of an event changes, a new phase is created, rather than the current one being changed (as in [48]).

These are enough reasons not to use synchronization.

IX. Could the Look at Event-Driven Programming Be Implemented?

It is feasible to add up numbers from 1 to N with our look at EDP. The circles (fig. 4) represent events processed by the resolving handlers. The arrows between the events represent the work of the transformation handlers.

The problem was programmed in Java. A project can be downloaded from the author's website http://prutzkow.com [49].

X. How is the Look Programmed?

A. How Is an Event Mapped in a Program?

An event is an abstract data class and has the following members:

- fields of processing status (listing 1, line 6), a reference to the parent event (line 7), and the processing result (line 8);
- methods for getting field values (omitted from the listing), an factory method for duplicating an event as handled one (lines 36-48) reflectionally with an overriding constructor (lines 18-23);
- method for switching the event phase with the reconstruction of the phase object (lines 32-34).

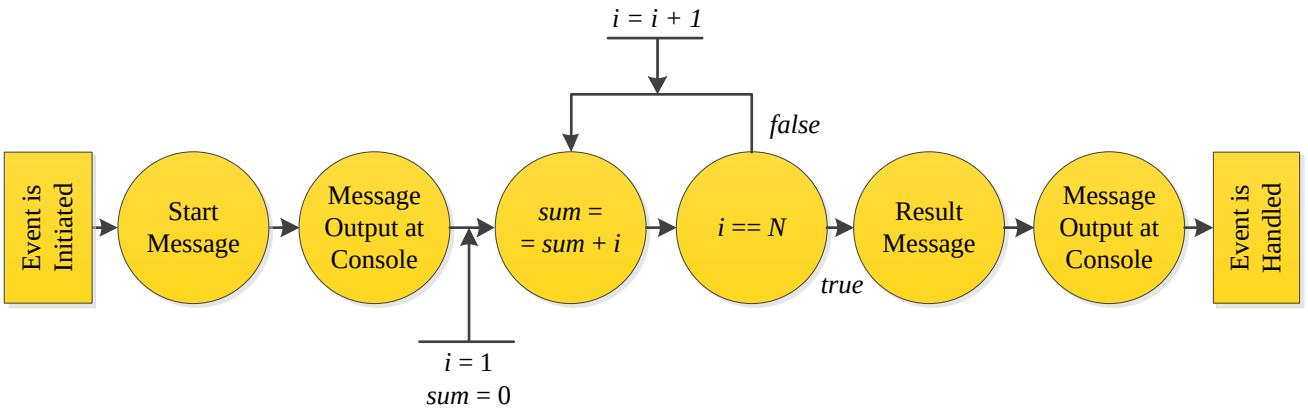

Fig. 4. Event flow for summing numbers from 1 to *N*

Listing 1. The AbstractEvent superclass

```
1  public abstract class AbstractEvent {
2      private static final String DEFALT_PHASE = "Default Phase";
3
4      public static final AbstractEvent NULL_EVENT = new
               AbstractEvent(AbstractEvent.NULL_EVENT) {};
5
6      private final EventHandleStatus eventHandleStatus;
7      private final AbstractEvent parentEvent;
8      private final EventResult eventResult;
9
10     private EventPhases eventPhases = new
               EventPhases(List.of(AbstractEvent.DEFALT_PHASE));
11
12     public AbstractEvent(AbstractEvent parentEvent) {
13        this.eventHandleStatus = EventHandleStatus.INITIATED;
14        this.parentEvent = parentEvent;
15        this.eventResult = EventResult.NULL_RESULT;
16     }
17
18     public AbstractEvent(AbstractEvent sourceAbstractEvent,
               EventResult eventResult) {
19        this.eventHandleStatus = EventHandleStatus.HANDLED;
20        this.parentEvent = sourceAbstractEvent.parentEvent;
21        this.eventResult = eventResult;
22        this.eventPhases = sourceAbstractEvent.eventPhases;
23     }
24
25     protected AbstractEvent(AbstractEvent parentEvent, List<String>
               eventPhasesAsString) {
26        this.eventHandleStatus = EventHandleStatus.INITIATED;
27        this.parentEvent = parentEvent;
28        this.eventResult = EventResult.NULL_RESULT;
29        this.eventPhases = new EventPhases(eventPhasesAsString);
30     }
31
32     public void switchPhase() {
33        this.eventPhases = this.eventPhases.switchPhase();
34     }
35
36     public AbstractEvent
               getHandledEvent(EventResult eventResult) {
37        Constructor<? extends AbstractEvent>  constructor = null;
38        try { constructor = this.getClass().
               getConstructor(this.getClass(), EventResult.class);
39        } catch (NoSuchMethodException | SecurityException e) {
40           System.err.printf("Can't get constructor reflectionally for
               class %s to create a handled event. Check
               %s(sourceEvent, eventResult) existence and publicity\n",
               this.getClass().getCanonicalName(),
               this.getClass().getSimpleName());
41           return AbstractEvent.NULL_EVENT;
42        }
43        try { return (AbstractEvent) constructor.newInstance(this,
               eventResult);
44        } catch (InstantiationException | IllegalAccessException |
               IllegalArgumentException | InvocationTargetException e)
               {
45           System.err.printf("Can't create a handled event reflectionally
               for class %s\n", this.getClass().getCanonicalName());
46        }
47        return AbstractEvent.NULL_EVENT;
48     }
49
50     // field getters
51 }
```

The class has no methods for setting field values, so its instances are immutable. The class has three constructors. The first constructor (lines 12-16) is used for single-phase initiated events.

The event class has no a separate type field. The event type is determined from its class by the getClass method of the Object superclass.

The abstract class declares an empty event NULL_EVENT (line 4).

Subclasses that are concrete events have only fields, constructors, and getters.

B. How Is an Event Handler Programmed?

An event handler is a utility class implementing the EventHandler interface with only final fields and therefore no state. The interface (listing 2) declares three methods and an empty handler NULL_EVENT_HANDLER.

The interface methods define the basic actions of the handler:

(1) the canHandle method (line 2) filters for the processing event;
(2) the handle method (line 4) processes the input event and initiates an output event (fig. 2 and fig. 3);
(3) the getSubscribedEvent method (line 6) returns the class of the event that the handler should subscribe; this method simplifies the implementation of the subscription.

The empty NULL_EVENT_HANDLER handler (lines 8–10) is required as the negative result of the event handler searching.

Listing 2. The EventHandler interface

```
1  public interface EventHandler {
2     boolean canHandle(AbstractEvent inputEvent);
3
4     AbstractEvent handle(AbstractEvent inputEvent);
5
6     Class<? extends AbstractEvent> getSubscribedEvent();
7
8     EventHandler NULL_EVENT_HANDLER = new EventHandler() {
9        // overridden method implementation
10    };
11 }
```

C. How Is the Event Loop Implemented in a Program?

The event loop is implemented as follows. The handler can be subscribed to the event itself (for a resolving handler) or to the parent event (for a transformation handler). In the event loop, there are two handler searches: by event (listing 3, line 1) and by parent event (line 7). If the event handler is not found (line 2) and the parent event is empty (line 4), the event is not processed. The result of processing the input event is the output event (line 9, as in [19, p. 79]). The output event is pushed into the event queue (line 10).

Listing 3. The event loop

```
1   EventHandler eventHandler =
                this.eventSubscriberRegister.getEventHandler(event);
2   if (eventHandler == EventHandler.NULL_EVENT_HANDLER) {
3      AbstractEvent parentEvent = event.getParentEvent();
4      if (parentEvent == AbstractEvent.NULL_EVENT) {
5         return;
6      }
7      eventHandler = this.eventSubscriberRegister.
                getEventHandler(event, parentEvent);
8   }
9   AbstractEvent outputEvent = eventHandler.handle(event);
10  this.eventQueueHandler.addEvent(outputEvent);
↳
```

XI. What are the Results?

We inventory the control statements (table 4, table 3) in 26 classes. All the conditional and loop statements are in the EDA framework. There are neither events nor event handlers with the conditional or loop statements.

There are no tricks to avoid the conditional and loop statements (Java Stream API, ternary operator, etc.).

Table 3. Conditional Statement Occurrences

Class	Condition
EventLoopHandler	Equality of the found handler and EventHandler. NULL_EVENT_HANDLER (listing 3, line 2)
EventLoopHandler	Equality of the parent event and AbstractEvent.NULL_EVENT (listing 3, line 4)
EventSubscriberRegister	Capability of the handler for event processing
EventPhases	Existence of the next phase

Table 4. Loop Statement Occurrences

Class	Description
EventLoopHandler	Event loop
EventSubscriberRegister	Search a handler by event
ThreadPool	Add threads to a pool and start them

XII. What Is in the Issue? (Conclusion)

We conclude the following:

(a) The problem of unification of control statements is posed, reduced to the use of only a linear sequence of commands.

(b) EDP is a programming paradigm suitable for unification of control statements. EDP is a base for EDA. EDA includes an event source, an event transmission channel or an event listener, and an event handler. The main attributes of the paradigm are low coupling and decentralized control.

(c) We introduced a new look at EDP. EDA includes framework (an event queue, an event queue dispatcher, event loop handlers, an abstract event and an abstract event handlers), events and their handlers. The event is characterized by a type, input data, status, parent event, and the result of event processing. The event handler processes the event and returns the event it is subscribed to. The look at EDP is thread-safe.

(d) The conditional and loop statements are in the EDA framework, but not in events and event handlers. The control statements have been unified, as in Markov normal algorithms. Unification of the control statements will simplify its development and testing.

There are use cases of the results of the study:

- languages of simplified programming;
- solutions of various event-driven problems (for example, in software for devices).

The following issues need to be addressed in further study:

- the complexity of writing event handlers and the events themselves;
- difficulty in debugging errors related to the sequence of events and handler calls.

To address these issues, the study will continue as follows:

- development of a visual program development environment (as in [42]);
- search for errors in the event-driven program: lack of a handler for an event; lack of conversion of the results of one event to another;
- issuing recommendations on the program;
- identification of types of tasks that cannot be implemented by EDP, and their characteristic features.

We used EDP and EDA in the author's website [49].

University students [50] and high school students [51] learn the EDP while developing a graphical user interface.

As you can see from the references, the actual interest in EDP is supported only by microservices [10, 19, 20] and the Kafka system [2, 8].

References

[1] Markov A.A., Nagorny N.M. The Theory of Algorithms. Kluwer Academic Publishers, 1988.
[2] Stopford B. Designing Event-Driven Systems. Concepts and Patterns for Streaming Services with Apache Kafka. O'Reilly, 2018.
[3] Etzion O., Niblett P. Event Processing in Action. Manning, 2011.
[4] Clements P. et al. Documenting Software Architectures, 2nd ed. Addison-Wesley, 2010.
[5] Mens T., Demeyer S. (eds) Software Evolution. Springer, 2008.
[6] Fairbanks G. Just Enough Software Architecture. A Risk-Driven Approach. Marshall & Brainerd, 2010.
[7] Ford N. et al. Building Evolutionary Architectures. O'Reilly, 2017.
[8] Koutanov E. Effective Kafka. A Hands-On Guide to Building Robust and Scalable Event-Driven Applications with Code Examples in Java. Leanpub, 2021.
[9] Meyer B. Object-Oriented Software Construction, 2nd ed. Prentice Hall, 1997.
[10] Percival H., Gregory B. Architecture Patterns with Python Enabling. Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. O'Reilly, 2020.
[11] Hansen S., Fossum T.V. Event Based Programming. In Kenosha WI, 2010.
[12] Richards M., Ford N. Fundamentals of Software Architecture. An Engineering Approach. O'Reilly, 2020.
[13] Bansal A. Introduction to Programming Languages. CRC Press, 2014.
[14] Mühl G. et al. Distributed Event-Based Systems. Springer, 2006.
[15] Carzaniga A. et al. Issues in Supporting Event-Based Architectural Styles. In 3rd International Workshop on Software Architecture, 1998:17–20.
[16] Faison T. Event-Based Programming. Taking Events to the Limit. Apress, 2006.
[17] Liang Y. Introduction to Java Programming and Data Structures. Comprehensive Version, 12th ed. Pearson, 2019.
[18] Tucker A., Noonan R. Programming Languages: Principles and Paradigms, 2nd ed. McGraw-Hill, 2007.
[19] Bellemare A. Building Event-Driven Microservices. O'Reilly, 2020.
[20] Carnell J., Sánchez I.H. Spring Microservices in Action, 2nd ed. Manning, 2021.
[21] Robillard M. Introduction to Software Design with Java. Springer, 2019.

[22] Friesen J. Java XML and JSON. Document Processing for Java SE, 2nd ed. Apress, 2019.
[23] Williams M. et al. Expert Twisted. Event-Driven and Asynchronous Programming with Python. Apress, 2019.
[24] Krasner G., Pope S. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In Journal of Object-Oriented Programming, 1988.
[25] Ferg S. Event-Driven Programming: Introduction, Tutorial, History. 2006.
[26] Yourdon E., Constantine L. Structured Design. Fundamentals of a Discipline of Computer Program and Systems Design, 2nd ed. Yourdon Press, 1978.
[27] Stein L.A. Challenging the Computational Metaphor: Implications for How We Think. In Cybernetics and Systems, 1999, 30(6).
[28] Paykin J. et al. The Essence of Event-Driven Programming. In Leibniz International Proceedings in Informatics, 2016.
[29] Lalejini A. et al. Evolving Event-Driven Programs with SignalGP. In Genetic and Evolutionary Computation Conference, 2018:1135–1142. DOI: 10.1145/3205455.3205523.
[30] Philip G. Software Design Guidelines for Event-Driven Programming. In Journal of Systems and Software, 1998, 41:79–91.
[31] Petitpierre C. An Event-Driven Programming Paradigm Compatible with OO-Programming. In OOPSLA, 1998.
[32] Fischer J. et al. Tasks: Language Support for Event-Driven Programming. In ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 2007:134–143. DOI: 10.1145/1244381.1244403.
[33] Petrosino G. et al. Imperative and Event-Driven Programming of Interoperable Software Agents. In International Workshop on Engineering Multi-Agent Systems, 2023:23–40.
[34] Krohn M.N. et al. Events Can Make Sense. In USENIX Annual Technical Conference, 2007:87–100.
[35] Fonseca A. et al. Eve: A Parallel Event-Driven Programming Language. In Euro-Par 2014 Workshops, Part II, 2014:170–181.
[36] Van Cutsem T. et al. AmbientTalk: Object-Oriented Event-Driven Programming in Mobile Ad hoc Networks. In 26th International Conference of the Chilean Computer Science Society, 2007.
[37] Srbljić S. et al. Programming Language Design for Event-Driven Service Composition. In Automatika, 2010, 51(4):374–386.
[38] Milicevic A. et al. Model-Based, Event-Driven Programming Paradigm for Interactive Web Applications. In ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'13), 2013:17–36. DOI: 10.1145/2509578.2509588.
[39] Dunkels A. et al. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In SenSys, 2006.
[40] Fan X. et al. Towards an Event-Driven Programming Model for OpenMP. In 45th International Conference on Parallel Processing Workshops, 2016:240–249. DOI: 10.1109/ICPPW.2016.44.
[41] Desai A. et al. P: Safe Asynchronous Event-Driven Programming. In 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013.
[42] Harel D., Kugler H. The Rhapsody Semantics of Statecharts (or, on the Executable Core of the UML) – Preliminary Version. In SoftSpez Final Report, Lecture Notes in Computer Science, 2004, 3147:3250354.
[43] McClurg J. et al. Event-Driven Network Programming. In PLDI, 2016:369–385. DOI: 10.1145/2908080.2908097.
[44] Ibanez S. et al. Event-Driven Packet Processing. In 18th ACM Workshop on Hot Topics in Networks, 2019:133–140. DOI: 10.1145/3365609.3365848.
[45] Hosobe H. Testing Event-Driven Programs in Processing. In ESSE, 2020. DOI: 10.1145/3393822.3432338.
[46] von Behren J.R. et al. Why Events are a Bad Idea (for Highconcurrency Servers). In HotOS, 2003:19–24.
[47] Jaworski M., Ziade T. Expert Python Programming, 4th ed. Packt, 2021.
[48] Noback M. Object Design Style Guide. Manning, 2019.
[49] Prutzkow A.V. Internet-Resurs dlja Razmeschenija Rezultatov Nauchnoj i Obrazovatelnoj Dejatelnosti [Internet-Resource for Scientific and Educational Work Result Publishing]. In Vestnik of the RSREU, 2018, 63:84–89. [in Rus]. DOI: 10.21667/1995-4565-2018-63-1-84-89.
[50] Bruce K. et al. Event-Driven Programming Facilitates Learning Standard Programming Concepts. In OOPSLA, 2004.
[51] Lang R., Saacks-Giguette M. Introducing High School Students to Event-Driven Programming. In 29th ASEE/IEEE Frontiers in Education Conference, 1999.