Reactor Design Pattern

Marko Mijač¹, Antonio García-Cabot², Vjeran Strahonja¹

¹University of Zagreb, Faculty of Organization and Informatics, Varaždin, Croatia

²University of Alcalá, Computer Science Department, Alcalá de Henares, Spain

Abstract – In object-oriented (OO) applications, objects collaborate through message passing, which results in these objects being coupled and mutually dependent. These dependencies can be *reactive*, i.e. such that, for example, the state change of one object, requires automatic reaction in all dependent objects. Examples of such reactive dependencies can be found in various software systems, including rich graphical interfaces, spreadsheet systems, animation, robotics, etc. Unlike the reactive paradigm which natively provides abstractions and mechanisms for the management of reactive dependencies, the OO paradigm lacks proper support. Object-oriented applications developers often resort to the use of ad-hoc solutions or design patterns such as the well-known Observer pattern, which are not suitable for managing more complex scenarios. In this paper we offer a novel design pattern (REACTOR), which utilizes a graph data structure to improve the management of reactive dependencies in OO applications.

Keywords – design patterns, design science, object-oriented paradigm, reactive paradigm

1. Introduction

The Object-oriented (OO) paradigm dominates in the modeling and development of large and complex software systems. The central abstraction of this

paradigm is "object", which encapsulates state (attributes) and associated behavior (methods) of concepts in the problem domain. Each object has its responsibility and is expected to contribute to the overall goal of the application. In this process, objects collaborate in a form of message passing, i.e. invoking other objects' behavior, or fetching their state. This results in objects being coupled and mutually dependent.

Dependencies between objects can sometimes be *reactive*, i.e. such that for example the state change in one object, automatically triggers an update of state in all dependent objects. Dependent objects need to recognize the occurrence of something interesting happening and *react* accordingly. Examples of such reactive dependencies can be found in several software systems, including rich graphical user interfaces, spreadsheets, animation, modeling, etc.

OO paradigm lacks native support for managing reactive dependencies. Instead, the well-known *Observer* design pattern (or some variant of it) is mostly used as a solution. Observer's overall idea is to "*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*" [1]. This coincides with the idea of reactive dependencies, however, the use of Observer has several shortcomings that made it heavily criticized as unfit for complex use (e.g. in [2], [3] and [4]). Indeed, when managing a large number of mutually interconnected reactive dependencies, the following issues may arise: (1) failing to react and update dependent objects when there is a need for it; (2) reacting and updating defensively and redundantly when there is no need for it; (3) updating in the wrong order and thus causing inconsistencies (so-called *glitches*); (4) forming circular dependencies which cause infinite loops, etc. In order to address these issues, we propose a novel design pattern, which uses well-known graph structures to express reactive dependencies and support related operations.

The rest of the paper starts with presenting related work (section 2), followed by a description of the chosen research paradigm and methodological framework (section 3). Outcomes of activities and steps prescribed by the methodological framework are contained in section 4. Finally, contributions are discussed and conclusions offered in section 5.

DOI: 10.18421/TEM101-03

<https://doi.org/10.18421/TEM101-03>

Corresponding author: Marko Mijač,

Department of Information Systems Development, Faculty of Organization and Informatics, Pavlinska 2, Varaždin, Croatia.

Email: mmijac@foi.hr

Received: 10 November 2020.

Revised: 09 December 2020.

Accepted: 16 December 2020.

Published: 27 February 2021.

© 2021 Marko Mijač, Antonio García-Cabot & Vjeran Strahonja; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at www.temjournal.com

2. Related Work

The concept of reactive dependencies described in the previous section has been recognized by both academy and industry and is related to several fields of research. Reactive programming, event-based programming, and aspect-oriented programming are examples of relevant research fields. Solutions offered in these fields differ in approach and scale and take different forms, ranging from design patterns, libraries and frameworks, specialized programming languages, and language extensions.

Bainomugisha et al. [5] in their extensive survey define *reactive programming* as a paradigm centered around the issues of *continuous time-varying values* and *propagation of change*. These issues are tackled by providing abstractions able to express and automate the flow of time, as well as abstractions for expressing data and computational dependencies. Thus, they identify two distinguishing abstractions offered by reactive programming languages: (1) *behaviors or signals* and (2) *events*. Based on these built-in abstractions some solutions from the fields of reactive programming provide a mechanism for automated management of reactive dependencies.

Most of the research in the field of reactive programming stems from representatives of functional-reactive languages such as Fran [6], which is why reactive programming has a strong functional-declarative flavor as opposed to the imperative style of OO programming. Recent trends, however, acknowledge the advantages and disadvantages of both reactive and object-oriented paradigm and aim at reconciling them. Boix et al. [7] identify two approaches for this: the first taking reactive programming, and the second taking OO programming as a starting point. Their ROAM framework implemented in *AmbientTalk* programming language is an example of the second approach. Conversely, REScala is an example of the first approach [2].

Event-driven programming is a paradigm specifically suited for the development of event-based systems. According to Faison [8], these are the systems whose elements primarily interact through event notifications. Similarly, Hinze et al. [9] describe them as systems in which observed events cause reactions. Most of today's software systems, especially the ones with graphical user interfaces, are event-based.

At the core of the event-based paradigm is the notion of *event*. It can be defined as a "*detectable condition that can trigger notification*" [8], or "*significant change in the state of the universe*" [9]. It is important to see that only detectable and significant conditions qualify as events. A related concept is a *notification* which can be defined as an

"*event-triggered signal sent to a run-time recipient*" [8]. This signal is usually sent either by *transferring data* or *transferring execution control*.

Faison [8] describes two principal roles essential in event-based interaction. The first role is commonly referred to as *event publisher/sender* and it describes an entity able to produce an event and send a notification. The second role, commonly referred to as *event subscriber/receiver*, describes an entity able to receive notification and react to it. The act of sending the notification is often called *firing the event*, while the act of establishing the link between sender and receiver is called *subscription* or *registration* [8]. Similarly, Hinze et al. [9] classify parts of the event-based system into (1) *monitoring component*, (2) *transmission mechanism*, and (3) *reactive component*. While monitoring and reactive components correspond to event sender and event receiver roles respectively, transmission mechanisms describe how notifications are sent and received. If notifications are exchanged directly between sender and receiver, we talk about *decentralized, point-to-point* systems. However, often a separate, third component is introduced to decouple senders and receivers and route notifications throughout the system. Such an approach is called a *centralized or middleware* approach.

At the core of event-based programming is the idea of *implicit invocation*, which according to Steimann et al. [10] can be considered as both an architectural style and programming paradigm. Since being introduced by Garlan and Notkin [11], it is frequently used to implement inversion of control and achieve loose integration of objects and components. While in traditional systems the components usually interact by explicitly invoking each other's methods, implicit invocation relies on events instead. A component, acting as a sender, can publish one or more events as a means of informing other components that some *significant change* happened. Other components, acting as receivers, can express their interest in these events by subscribing to it. Whenever the sender component triggers an event, all subscribed components invoke their event handling methods (thus the name *implicit*). Now, instead of sender component being required to know its receivers, receivers are required to know the sender (thus the name *inversion of control*). According to Notkin et al. [12], implicit invocation makes it easier to add, modify, and integrate new components, without the need to change existing components. Garlan and Shaw [13] in their seminal work also recognize support for reuse and evolution of software as the two most important benefits of implicit invocation.

Implicit invocation and event-driven programming are based on imperative implementations of design patterns such as the well-known Observer pattern

[11]. The stated intent and overall idea of Observer is to “*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*” [1]. The original Observer pattern assumes objects taking the role of *Subject* (Observable) - an object whose state change can be observed, or *Observer* - an object who observes Subject's state and updates its state accordingly. That said, implicit invocation implemented through design patterns is at least at the conceptual level a natural fit for managing reactive dependencies. However, solutions based on the Observer pattern are often criticized as inadequate, so similar, but “more powerful” design patterns, have been proposed over time.

Observer pattern revisited [14] for example improves the original Observer pattern by introducing the concept of *ObserverManager* as a central entity in charge of managing dependencies between *Observer* and *Observable*. This is along the lines of *ChangeManager* in a more advanced version of Observer pattern proposed by Gamma et al. [1]. In the *Event-notification* pattern [15], on the other hand, dependencies are distributed across individual *Subjects* in dedicated *StateChange* objects, which together with *EventStub* objects allow finer event granulation (i.e. specifying multiple events per *Subject* and multiple update methods per *Observer*). *Propagator* pattern [16] also manages dependencies in distributed manner, however it discusses mechanisms for handling acyclic and cyclic dependency graphs.

Mijač et al. [4] analyzed these patterns and compared their ability to handle complex cases of event propagation and dependencies between objects. They concluded that while individual design patterns exhibit some useful features and ideas, no single design pattern is suitable for handling complex dependency graphs. This is especially the case with a traditional Observer pattern, which is mostly mentioned in the context of decoupling user interface implementation and underlying data. E.g. in a well-known MVC architectural pattern, an Observer pattern is used to decouple and synchronize Model, View, and Controller [17]. To address the shortcomings of current Observer pattern implementations in handling complex dependency networks, Mijač et al. [4] identified several useful features of existing design patterns and proposed they should be combined.

Aspect-oriented programming (AOP) is another field which offered its take on handling dependencies. It is a paradigm first described by Kiczales et al. [18] as an approach to implement design decisions that crosscut the system's basic functionalities, so-called crosscutting concerns. One of these crosscutting concerns is managing

dependencies and data synchronization between objects through design patterns such as Observer [19]. However, proponents of AOP point out several drawbacks of traditional Observer pattern implementations. Eales [14], for example, claims that in traditional Observer pattern implementations concrete classes must implement Subject/Observer interfaces or inherit abstract classes, while in fact, they are not real specializations of Subject or Observer. Not only does this disrupt natural inheritance hierarchy, but in existing systems with already established inheritance hierarchy, it can be very challenging to introduce this kind of Observer pattern implementation. Similar is stated by Jicheng et al. [20] as they claim that Observer pattern features are tangled with the object's core features, obscuring their primary concern. This results in the core system's functionality being more difficult to develop, understand, and maintain. The same thing happens also to code in charge of managing dependencies. On one hand, it is being scattered all over the system, and on the other, it is tangled with the code implementing core functionalities.

The solution to these problems is seen in using aspects. Noda and Kishi [21] use the concept of separation of concerns to implement design patterns more flexibly. They argue that design patterns and core application functionality are different concerns and need to be separated. In his paper Tennyson [19] presents a novel, “*aspectized*” variant of Observer pattern with the same intent as the original design pattern, but with eliminated or reduced crosscutting concerns. Several similar variants of Observer pattern based on aspects are proposed and demonstrated in AspectJ programming language: [20],[22],[23]. All these implementations recognize Observer pattern related code as a crosscutting concern. They extract it from core classes and put it in aspect, keeping core classes not only clean but completely oblivious of their role as Subject or Observer.

3. Methodology

This paper addresses a research problem that is not only recognized and relevant in an academy setting but is also frequently dealt with by practitioners. Similarly, the type of solution this paper proposes, i.e. design pattern, is the subject of many research papers, but even more importantly, it is directly applied by software architects and developers across the world. To systematically guide our efforts in transitioning from problem to solution, we need a suitable problem-solving scientific paradigm. This is offered by *design science* [24] - a pragmatic paradigm that solves real problems by creating innovative *artifacts* (constructs, methods, models,

and instances). While it has long been accepted in engineering disciplines [25], recently we are witnessing an increase in the popularity of *design science* also in fields of information systems and software engineering [24]. To conduct *design science* research rigorously, we follow the 5-activity methodological framework [26]:

Figure 1. Design science methodological framework

In (1) *Explicate Problem* activity, the problem of managing reactive dependencies in OO applications is first defined and then placed into context. We then proceed to show the problem as relevant to both practitioners and researchers. Finally, to be able to design the solution, we investigate possible causes of the problem. For this activity, a detailed literature review is conducted and shown in the previous section.

With the problem being explicated, we enter the (2) *Define requirements* activity which aims at defining requirements for the solution (artifact). We first outline the future artifact by stating its most important general characteristics. After that, detailed requirements are specified which artifact needs to meet to be able to eliminate or mitigate the causes of the problem. The main sources for requirements will be a literature review, the researcher's own experience, and the use of prototyping.

In (3) *Design and Develop artifact* activity according to stated requirements, the artifact is designed and developed. This activity is characterized by inner iterations in which we collect the design and implementation ideas, assess them,

select the promising ones, build them into the artifact, and reflect on the results.

Finally, (4) *Demonstrate artifact* and (5) *Evaluate artifact* activities are in this paper going to be treated as one joint "Evaluation" activity because we deal with a small-scale artifact. Evaluation in design science is essential as it adds a scientific component to design. To achieve the necessary rigor, we take into consideration the guidelines proposed by Mijač [27]. As suitable *properties* for evaluation, we chose *technical feasibility* and *efficacy*, which according to Prat et al. [28] are among top three most frequently evaluated properties in design science research. Technical feasibility shows us that it is possible to build an artifact from a technical perspective. Efficacy, on the other hand, shows that the artifact is working and has the potential to solve the stated problem. As for the evaluation method, we chose *demonstration* - one of the most frequently used evaluation methods in design science [27].

4. Creating Solution

In this section, we go through the previously defined activities of the methodological framework and document the efforts and outcomes associated with them.

a. Explicate problem

The activity of problem explication is hard to constrain within one part of the design process. In this paper, a lot of the problem explication efforts are already presented in the introduction and literature review sections. This subsection will reiterate and supplement already given information.

We consider dependency between two objects to be *reactive* if, for example, the state change in one object requires the other (dependent) object to automatically update its state. In cases when there is a lot of mutually dependent object states, reactive dependencies may form large and complex graph structures. The core problem this paper addresses is the difficulty and error-proneness of manual management of such graph structures. Even in a very simple graph, the developer must carefully analyze the graph to understand it. The situation becomes increasingly difficult as this graph structure becomes larger and more intertwined. At some point, it simply becomes unsustainable to keep manually determining what object states should be updated after a change, and in which order should this be done. This then leads to several critical problems, including (1) failing to perform necessary state updates, (2) performing unnecessary state updates (defensive updates), (3) perform state updates redundantly, (4) perform state updates in incorrect order (causing glitches and incorrect results), (5) infinite loops caused by circular dependencies, etc.

Managing reactive dependencies is a practical problem, pervasive in the development of OO applications. Indeed, one of the most well-known OO design patterns is *Observer* pattern [1]. Alternative design patterns that address this problem also exist. Some programming languages have also provided built-in features that try to mimic the idea of Observer pattern. For example, in .NET languages [29] as well as in Java [30] this is achieved using *events* and *event handlers* (*event listeners* in Java). In addition to its practicality, in section 2 we also saw that this problem regularly occurs in scientific research related to OO design patterns, reactive programming, event-driven programming, and aspect-oriented programming. All this demonstrates that managing reactive dependencies in OO applications as a research problem holds both practical and scientific relevance, which is one of the requirements set by *design science*.

There are multiple reasons why managing reactive dependencies in OO applications is being difficult and error-prone. The fundamental one is the fact that the graphs formed by reactive dependencies are inherently complex structures, which, due to our natural cognitive limitations we have a hard time comprehending and understanding. When faced with such limitations, we resort to using different tools and aids. However, in this context, they are difficult to build because there is a lack of proper abstractions and mechanisms for expressing reactive dependencies, graphs they form, and the overall handling of the objects' state update process.

b. Define requirements

In the previous section, we dealt with the research problem, however, in this section we make the first steps towards a solution. We start by outlining the fundamental aspects of the proposed artifact, which is followed by specifying a concrete list of requirements.

While our long-term goal is to provide a framework with integrated tools for managing reactive dependencies in OO applications, in this paper we address the lack of proper abstractions and mechanisms for expressing reactive dependencies, dependency graphs, and handling the update process. One of the software reuse techniques that are fit to represent this kind of artifacts is *design pattern* [1]. Thus, the main resulting *artifact* of design science process described in this paper is going to be a novel design pattern called REACTOR. As design pattern, REACTOR can from the perspective of design science be characterized as *model* type of artifact.

To outline the REACTOR as an artifact let us examine its position in the software reuse landscape. Being a design pattern, the REACTOR is at the pure design reuse from the perspective of *abstraction*.

Regarding its *size*, design patterns usually offer smaller-scale reuse (as opposed to e.g. architectures) by targeting specific, micro-architectural issues.

Gama et al. [1] suggest individual design patterns to be cataloged depending on their *purpose* (what does it do?) and *scope* (does it apply to classes or objects?). In this regard, the REACTOR can be characterized as *behavioral* (deals with how classes and objects interact and distribute responsibility) and *object* (deals with relationships between objects).

The natural starting point in eliciting requirements was understanding the research problem and its causes, which we documented in the previous section. This was followed by analyzing existing literature in the search for other design patterns presented as an alternative to Observer pattern. Significant input for this we obtained from our previous research [4]. After that, a series of try-outs and prototyping were performed to identify desirable characteristics of design pattern artifact. These were then transformed into a consistent list of requirements (see *Table 1.*).

Model artifacts are built from constructs related to each other [26]. Since the REACTOR will be based on graph structure, we will borrow terminology from graph theory to form a vocabulary of constructs necessary to express requirements.

- **Reactive node** n is an entity that encapsulates the state/behavior of an object designated to participate in reactive dependencies.
- **Predecessor node** is a reactive node p which acts as a *predecessor* to some other node s in reactive dependency d . In a graph, it is represented as a starting node of an edge e .
- **Successor node** is a reactive node s which acts as a successor to some other node p in reactive dependency d . In a graph, it is represented as an ending node of an edge e .
- **Reactive dependency** is an ordered pair $d=(p, s)$, where p acts as a predecessor node and s acts as a successor node. It is represented by an edge e in a graph.
- **Dependency graph** is a structure expressed as an ordered pair $G = (N,D)$, where N is a set of reactive nodes and D is a set of reactive dependencies. Each reactive dependency d from D associates a pair of reactive nodes (p,s) from N , where p is a predecessor and s is a successor.
- **Update process** is the process of updating reactive nodes from dependency graph after e.g. a change in graph structure occurred or there was a change in an object state represented by reactive node.

Table 1. List of specified requirements

#	Requirement
REQ1	Multiple reactive nodes can be defined for one target object, allowing finer granularity.
REQ2	Reactive node can simultaneously play different roles in different reactive dependencies.
REQ3	Reactive dependency is identified by the ordered pair of predecessor and successor node.
REQ4	Reactive nodes and reactive dependencies are part of a larger structure – dependency graph.
REQ5	Dependency graph can be changed at runtime by adding and removing nodes and dependencies.
REQ6	Multiple dependency graphs can coexist independently in one software application.
REQ7	Dependency graph can detect cycles.
REQ8	Arbitrary method can be assigned to perform update of individual reactive node.
REQ9	During update process all nodes that are affected by the change are updated.
REQ10	During update process only nodes that are affected by the change are updated.
REQ11	During update process, nodes are updated in correct order which prevents glitches and state inconsistencies.
REQ12	Information on the status of update process can be obtained.

c. Design and develop artifact

Now that we have the list of requirements our artifact must fulfill; we can proceed with activities in charge of concretizing these requirements by creating design pattern artifact. Activities in this section boil down to describing available design options, assessing these options, trying them out, and accepting or abandoning them.

In order to be able to discuss design of reactive dependencies, we first have to consider some design options for their constituent parts – *reactive nodes*. The usual interpretations in Observer and related patterns describe reactive dependencies as a means to synchronize state of objects. In practice, this is most often realized by representing the entire object's state with one reactive node (*object-level* granulation). The one exception with finer granulation we found in *Event notification* [15] pattern, which allows multiple reactive nodes per object to be defined in a form of special *StateChange* (predecessor) and *EventStub* (successor) objects (see Table 2.). Rather than reactive node representing the entire state of an object (as in Observer pattern), with the REACTOR we want to enable finer granulation, i.e. make it possible for reactive node to represent individual state member of an object. This will allow for more precise management of reactive dependencies.

Reactive dependency can be described as an ordered pair of two reactive nodes playing the roles of *predecessor* and *successor*. Except for Propagator pattern, in existing design patterns these roles are treated as distinct concepts, implemented by separate abstractions (see Table 2.). Regarding the implementation mechanism, reactive node's roles are usually played by inheriting abstract classes, realizing interfaces, or a combination of the two options. *Event notification pattern* is an exception and uses composition.

For reactive dependencies to be able to form acyclic graphs, individual reactive nodes must be able to play both predecessor and successor role. We cannot make separate roles happen using inheritance since in most OO languages multiple inheritance is not possible. We could circumvent this problem by using interfaces, however, this would seriously limit future implementation reuse. Therefore, we opt for using a single abstraction, i.e. reactive node encompassing both roles. Since even in this case inheritance would impair our desired granularity, and possibly disrupt existing class hierarchies, we decide to use composition as an implementation mechanism.

Table 2. Reactive node's roles in different design patterns

Pattern	Predecessor	Successor	Mechanism
Observer (simple) [1]	Subject	Observer	Inheritance
Observer (advanced) [1]	Subject	Observer	Inheritance
Observer revisited [14]	Observable	Observer	Inheritance
Event notification [15]	StateChange	EventStub	Composition
Propagator [16]	Propagator		Inheritance

One additional design point related to reactive dependencies is whether we want to express them as *implicit* or *explicit* concepts. In an implicit option, we express reactive dependency as a mere association of two reactive nodes and therefore need only reactive node abstraction. In case of an explicit option, we must create additional abstraction for reactive dependencies themselves, however, this also allows us to capture additional data (e.g. weight, priority, etc.). If we look at the related patterns and language features, we can see that reactive dependencies are expressed implicitly. Since in the REACTOR we also do not consider associating additional data with individual reactive dependencies, we will also opt for implicit implementation.

In addition to expressing individual reactive dependencies, another design point worth consideration is how do we express dependency

graph as a set of reactive nodes interrelated with reactive dependencies. Again, a question is whether we need dependency graph as an implicit or an explicit concept. Part of design patterns and built-in features propose making predecessor nodes in charge of constructing and storing reactive dependencies they are involved in. Instead of graph-like abstraction that manages reactive dependencies centrally, reactive dependencies are scattered across individual predecessors. On the other hand, Observer pattern (advanced) and Observer pattern (revisited) do prescribe a separate abstraction called *Manager*, which takes on the responsibility of dependency graph. Contrary to reactive dependencies, with dependency graph we advocate the use of explicit abstraction. One of the reasons is that we want to ensure multiple dependency graphs can coexist in the application. Also, there is whole range of state and behavior related to constructing, storing, fetching, and updating reactive dependencies that naturally fit into this abstraction.

Graphs in general are commonly represented by two data structures, namely: adjacency matrix and adjacency list [31], although incidence matrix is also mentioned in this context in developers community forums. Since matrix structures (adjacency and incidence matrix) would in our case be inefficient in terms of memory space, adding and removing nodes, search and traversal operations, we will use adjacency list to express dependency graph.

The main goal of handling reactive dependencies is keeping dependency graph and its reactive nodes consistent, i.e. ensuring that all state members are up to date. The inconsistencies arise because of change in dependency graph structure (adding or removing reactive nodes and dependencies), or because of a change in the object's state represented by reactive node. When we determine the cause of the inconsistency, we must determine which reactive nodes are going to be updated, and in which order. Since dependency graph is a directed-acyclic graph (DAG) structure, this means we must find topological sorting of the graph (or the part of the graph). Topological sorting will give us a linear ordering of reactive nodes, such that for every reactive dependency $d = (p, s)$ from predecessor node p to successor node s , p comes before s in update process. Commonly used algorithms for topological sorting are *Kahn's algorithm* [32] and *Tarjan's algorithm* [33], which are based on *breadth-first search* (BFS) and *depth-first search* graph traversal algorithms respectively. We found both algorithms perfectly suitable in our context.

To describe the REACTOR pattern in a systematic and standardized way we will use the template suggested by Gamma et al. [1] in their famous book.

Pattern name:

REACTOR

Classification:

Purpose: behavioral; *Scope:* object.

Intent:

Manages dependencies between objects which are characterized as *reactive*, i.e. such that for example the change in object's state automatically updates all dependent states in the same or other objects.

Also known as / similar with:

Observer, Event-notification, Propagator.

Motivation:

Manually managing reactive dependencies in OO applications is difficult and error-prone. This is due to potential size and inherent complexity of dependency graph which reactive dependencies tend to form. This leads to several issues such as: failing to update dependent nodes; defensive and redundant updates; update inconsistencies (so-called *glitches*); circular dependencies, etc.

Figure 2. shows simple example where fields/attributes of three classes are mutually dependent. For example, we can see that the value of attribute *E* (class *Gama*) is determined by some function *f* of attributes *A* and *B* (class *Alfa*). If *A* or *B* change their value, *E* must be updated to be valid, thus becoming dependent on them. Put together, multiple of such dependencies can form a dependency graph structure.

Figure 2. Example of reactive dependencies

In Figure 2. we can see that even when reactive dependencies form very simple graph, developer must carefully consider what reactive nodes should be updated after a change, and also in what order should this be done. For example, if node *A* is changed, this eventually affects nodes *B*, *D*, *E* and *F*. However, node *B* must be updated before *E* if we want *E* to be correct. Also, node *F* must be updated last.

The situation becomes more difficult as the size of dependency graph increases. At some point, it simply becomes unsustainable to manually determine which reactive nodes should be updated in what order. This is where the REACTOR pattern comes into play.

Applicability:

Reactor pattern is meant to be applied instead of the traditional Observer pattern in a more complex cases of dependencies between objects. Some of the indicators that should make you think about applying the REACTOR are:

- dependencies between objects form acyclic dependency graph,
- centralized approach in managing dependencies is needed,
- an object needs to behave as both subject and observer,
- finer granulation is needed than the one provided by Observer,
- It is undesirable or even not possible to integrate Subject and Observer roles into existing class hierarchy,
- Cycles in dependencies must be prevented,
- Update process should be monitored.

Structure

Figure 3. Reactor pattern structure

Participants

- *ReactiveNode* – encapsulates member of an owner object designated to participate in reactive dependencies. It is an identifiable node in dependency graph which can serve as both predecessor and successor in reactive dependencies. It provides an *Update* method which is invoked during the overall graph update process.
- *Reactor* – is responsible for providing data structure implementation capable of storing dependency graph, as well as operations for constructing graph. It also provides interface for invoking graph's update process, and mechanisms for monitoring its progress.
- *Client* – uses Reactor to construct dependency graph and invoke update process.

Collaborations

There are two main lines of collaborations that can be expected when using the REACTOR pattern. The first one deals with constructing or altering dependency graph. Here the client code makes new instances of reactive nodes and adds them to *Reactor* (*AddNode*), or fetches existing reactive nodes from the Reactor (*GetNode*). Once it has required reactive nodes, client code can establish reactive dependencies between them (*CreateDependency*). Besides adding nodes and dependencies, client code can also remove nodes and dependencies from *Reactor*.

The second line of collaborations deals with performing graph update process (see *Figure 4*). Again, client code can use *Reactor* to invoke update process of entire dependency graph, or only part of it. After determining which reactive nodes must be updated and the order in which this should be done, *Reactor* requests from each of these nodes to update themselves.

Figure 4. Example of Client invoking update process

Consequences

REACTOR pattern allows developers to treat management of reactive dependencies as a crosscutting concern, separate from core responsibility of client classes. When using the REACTOR, we do not have to introduce artificial hierarchies in our application's code, nor do we have to change the structure of our classes.

As with traditional Observer pattern, objects and members represented by reactive nodes are decoupled and entirely unaware of their roles in reactive dependencies. While this individual "blindness" may be desirable from the perspective of decoupling, there are situations when it is beneficial, or even necessary to have complete picture. With the REACTOR we can access entire graph structure, not only to perform update process correctly but also for implementing potential optimizations, graph analyses and visualizations. Indeed, one of the greatest benefits of the Reactor is that it can serve as a basis for developing more powerful options for managing reactive dependencies.

Implementation

While the general structure of the Reactor pattern is mostly technology-neutral, when it comes to concrete implementation, we must use some particular technology. Our implementation will be based on C# .NET technology (see Figure 5.), although we will in parallel also mention possible Java implementation.

As we previously discussed, *ReactiveNode* points to an individual *Member* of an object. We will refer to a member by their name (string) since it is more general and technology-agnostic option. However, one can also use meta-data member representations, which are available using *reflection* in many programming languages (including .NET and Java). The owner object is also going to be stored as an Object supertype, which exists in .NET, Java, and many other OO languages.

With regard to a reference to an *Owner* object, one issue that should be mentioned is object deallocation. If we would reference an owner object in *ReactiveNode* in a conventional manner, we would create a strong reference to an object. This means that the garbage collector would not be able to collect the object even if there were no references to an object in application code. One option to prevent this would be to use the so-called *weak references*. Their defining characteristic is that they provide access to an object, but they do not prevent garbage collector from collecting the object. Weak references are supported in both .NET and Java.

Another aspect of reactive nodes which is essential for any operation *Reactor* pattern supports, is being able to uniquely identify them. Here, the usual comparison by reference will not do, because we could have multiple reactive nodes pointing at the same *object-member* pair, and thus representing the same node. Therefore, we should assign reactive node with explicit and unique identifier, which can be automatically generated, for example, as a hash value combination of *Owner* object and *Member* name.

To be able to perform overall update process on a graph level, each reactive node also must be able to invoke arbitrary behavior on owner object. We do this by providing reactive node with *Update* method, which is just a proxy method that calls the owner object's method. The owner object's method can be referenced using constructs such as *delegates* (.NET) which are types that can hold a reference to a method. Although a bit less straightforward, this can also be achieved in Java using interfaces and *Delegate* class. Alternatively, we can turn to a *reflection* and *meta-programming* capabilities to reference and invoke a method.

Figure 5. Potential Reactor pattern implementation

If we want to represent different types of class members (e.g. fields, properties, methods) with reactive nodes, possible implementation differences can be extracted and dealt with in subclasses (see

Figure 5.). For example, *Update* method for a *PropertyReactiveNode* invokes explicitly defined method which reevaluates the property. On the other hand, in *MethodReactiveNode* we just need to invoke method specified by the *object-member* pair.

As previously discussed, we did not introduce explicit abstraction to model reactive dependencies. They are rather expressed as an association of two reactive nodes – one serving as a predecessor and one as a successor. If the need arises to assign additional information to reactive dependencies, then an explicit abstraction should be introduced.

While the *Reactor* does hold the central list of created reactive nodes, reactive dependencies are distributed across individual reactive nodes. This corresponds to adjacency list implementation of graph structure. Each reactive node uses two separate lists (*Predecessors* and *Successors*) to keep a record of reactive nodes which it forms reactive dependency with. This approach does require more memory space, but it is beneficial in terms of performance as it allows us to directly access all immediate neighbors of reactive node without graph traversal.

Regarding overall update process, two *PerformUpdate* method overloads allow us to update entire dependency graph, or only the part of it which depends on provided initial reactive node. These methods perform topological sorting using either Kahn's or Tarjan's algorithm. After obtaining list of

sorted nodes, all that it remains is to invoke *Update* method of each of the reactive nodes from the list.

In addition to performing update process, the Reactor also reports that the update process started, finished successfully, or failed. One of the ways to implement this in .NET would be to define *UpdateStarted*, *UpdateSuccessful* and *UpdateFailed* events, which client code can subscribe to and be notified when the events occur. Similar can be achieved in Java using *listeners*.

Sample Code

In this subsection we bring code fragments of the REACTOR pattern implementation in C# programming language.

ReactiveNode's constructor is used to provide mandatory parameters and generate unique identifier.

```
public ReactiveNode(string member, object ownerObject)
{
    Validate(member, ownerObject);

    Member = member;
    Owner = ownerObject;

    GenerateIdentifier();
}
```

Figure 6. Constructor for *ReactiveNode* class

PropertyNode implements *Update* method which uses reflection to invoke a method with name "*Update_{property name}*" on owner object. Method name format is in this case hardcoded, however, with small alterations we could allow developer to pass method name through e.g. *PropertyNode* constructor.

```
public override void Update()
{
    Type type = Owner.GetType();
    MethodInfo method = type.GetMethod($"Update_{Member}");
    if (method != null)
    {
        method.Invoke(Owner, null);
    }
}
```

Figure 7. *PropertyNode* Update method

In case of *MethodNode*, *Update* method just calls the method which is represented by the node.

```
public override void Update()
{
    Type type = Owner.GetType();
    MethodInfo method = type.GetMethod(Member);
    if (method != null)
    {
        method.Invoke(Owner, null);
    }
}
```

Figure 8. *MethodNode* Update method

When creating reactive dependency, predecessor and successor nodes are automatically added to graph in case they are not present. Predecessor and successor are then added each other's list of successors and predecessors, respectively.

```
public void CreateDependency(IReactiveNode predecessor,
    IReactiveNode successor)
{
    ValidateDependency(predecessor, successor);
    if (Nodes.Contains(predecessor) == false)
    {
        Nodes.Add(predecessor);
    }
    if (Nodes.Contains(successor) == false)
    {
        Nodes.Add(successor);
    }
    predecessor.AddSuccessor(successor);
    successor.AddPredecessor(predecessor);
}
```

Figure 9. Creating reactive dependencies

Performing update process includes sorting (topological) reactive nodes and then invoking *Update* method of each reactive node. Events *UpdateStarted*, *UpdateSuccessful* and *UpdateFailed* are fired at convenient places during the update process.

```
public void PerformUpdate()
{
    try
    {
        OnUpdateStarted();
        var sortedNodes = Sort(Nodes);
        foreach (var node in sortedNodes)
        {
            node.Update();
        }
        OnUpdateSuccessful();
    }
    catch (Exception) { OnUpdateFailed(); }
}
```

Figure 10. Reactor's *PerformUpdate* method

Known Uses

Since REACTOR is proposal for a novel design pattern, it does not yet have reported uses. However, in this paper we showed that it can be used as an alternative for well-known Observer pattern. This means that it can be used for synchronization in graphical user interfaces. However, more appropriate use would be in cases where dependencies form complex dependency graphs. Examples may include different calculation models and procedures such that can be found in spreadsheet systems.

Related Patterns

Depending on the complexity of chosen *Reactive pattern* implementation, various other design patterns can be applied together with the *Reactor*. These include:

- *Mediator pattern* - *Reactor* class can be seen as a *mediator* between individual reactive nodes.
- *Singleton pattern* – can be used to ensure only one *Reactor* instance to exists.
- *Registry pattern* – can be used to make available multiple instances of *Reactor*.
- *Factory patterns* – can be used to isolate instantiation logic of concrete *ReactiveNode*'s.
- *Strategy pattern* – can be used to make sorting algorithms interchangeable.

d. Evaluation

As we announced in methodology section, evaluation of the REACTOR design pattern (i.e. design science model artifact) will be done by *demonstrating* its *technical feasibility* and *efficacy*. According to March et al. [34], by building instantiation artifact we operationalize its underlying model artifacts, thus demonstrating their feasibility. In order to do this, we created C# implementation (*instantiation*) of the proposed design pattern (*model*). While fragments of implementation are presented in previous section, entire implementation can be found at pattern's GitHub repository (<https://git.io/JTHdC>).

Figure 11. Passing unit tests

To demonstrate design pattern's efficacy, we used *unit testing*. Testing in general has two goals: (1) demonstrate that software meets the requirements, and (2) discover incorrect or undesirable behavior [35]. Both goals align well with efficacy evaluation property. Additionally, since design patterns are operationalized by writing programming code, each unit test in fact simulates the usage of pattern, and thus shows that the pattern works.

Figure 12. Unit testing code coverage

To test the REACTOR design pattern implementation, a total of 46 unit tests (see Figure 11.) were written using MSTest unit testing framework. According to Visual Studio IDE, these tests provide 100% code coverage (see Figure 12.). Tests themselves were based on an illustrative scenario shown in Figure 2., and are available at the aforementioned GitHub repository.

5. Conclusion

In this paper we conducted design science research to address the problem of managing reactive dependencies in OO applications. The solution was offered in a form of a novel design pattern named the REACTOR. This design pattern is an improvement over the existing patterns with similar intent, and can be used instead of, for example, the well-known Observer pattern. The most notable REACTOR's benefits include: (1) handles acyclic graphs of any size and complexity, (2) detects cycles in graph, (3) each node can play both roles of subject and observer, (4) nodes can be defined at a finer granulation level (multiple nodes per object), (5) does not disrupt existing class hierarchies, (6) allows update process to be monitored, (7) can serve as a basis for more complex dependency management solutions.

This paper's contributions hold both practical and theoretical significance. Practitioners can utilize the pattern to express mutually dependent parts of object state as an acyclic graph structure, and then automate the update process. Since the REACTOR does not impose changes to client class structure (e.g. by inheritance), it can be more easily introduced into existing applications. Also, the REACTOR can provide a strong starting point for developing more complex and featureful solutions (e.g. frameworks).

Apart from practical importance, this paper also contributes to a scientific community. An extensive literature review shows the very problem we addressed is deeply rooted in the research on OO design patterns, reactive programming, event-driven

programming, aspect-oriented programming, etc. By offering novel design pattern, this paper directly contributes to these fields.

For the future research we plan to design a software framework based on REACTIVE design pattern. The framework will not only promote a much larger-scale design reuse, but it will also offer implementation reuse. For example, the framework will offer accompanying tools for dependency graph analysis and visualization. Also, it will address the issue of large amount of boiler-plate code resulted from specifying reactive nodes and dependencies.

References

- [1]. Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). Elements of reusable object-oriented software. *Reading: Addison-Wesley*.
- [2]. Salvaneschi, G., & Mezini, M. (2013, March). Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proceedings of the 12th annual international conference on Aspect-oriented software development* (pp. 37-48).
- [3]. Maier, I., Rompf, T., & Odersky, M. (2010). *Deprecating the observer pattern* (No. REP_WORK).
- [4]. Mijač, M., Kermek, D., & Stapić, Z. (2014). Complex Propagation of Events: Design Patterns Comparison. In V. Strahonja, N. Vrček., D. Plantak Vukovac, C. Barry, M. Lang, H. Linger, & C. Schneider (Eds.), *Information Systems Development: Transforming Organisations and Society through Information Systems (ISD2014 Proceedings)*. Varaždin, Croatia: Faculty of Organization and Informatics. ISBN: 978-953-6071-43-2.
- [5]. Bainomugisha, E., Carreton, A. L., Cutsem, T. V., Mostinckx, S., & Meuter, W. D. (2013). A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4), 1-34.
- [6]. Elliott, C., & Hudak, P. (1997, August). Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming* (pp. 263-273).
- [7]. Boix, E. G., Pinte, K., Van de Water, S., & De Meuter, W. (2013). Object-oriented reactive programming is not reactive object-oriented programming. *REM*, 13.
- [8]. Faison, T. (2006). *Event-Based Programming*. Berkeley, CA, USA: Apress,
- [9]. Hinze, A., Sachs, K., & Buchmann, A. (2009, July). Event-based applications and enabling technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems* (pp. 1-15).
- [10]. Steimann, F., Pawlitzki, T., Apel, S., & Kästner, C. (2010). Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1), 1-43.
- [11]. Garlan, D., & Notkin, D. (1991, October). Formalizing design spaces: Implicit invocation mechanisms. In *International Symposium of VDM Europe* (pp. 31-44). Springer, Berlin, Heidelberg.
- [12]. Notkin, D., Garland, D., Griswold, W. G., & Sullivan, K. (1993, November). Adding implicit invocation to languages: Three approaches. In *International Symposium on Object Technologies for Advanced Software* (pp. 489-510). Springer, Berlin, Heidelberg.
- [13]. Garlan, D., & Shaw, M. (1993). An introduction to software architecture. In *Advances in software engineering and knowledge engineering* (pp. 1-39).
- [14]. Eales, A. (2005). The observer pattern revisited. *Educating, Innovating & Transforming: Educators in IT: Concise paper*.
- [15]. Riehle, D. (1996). The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation. *TAPOS*, 2(1), 43-52.
- [16]. Feiler, P. H., & Tichy, W. F. (1997, August). Propagator: A family of patterns. In *Proceedings of TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages* (pp. 355-366). IEEE.
- [17]. Avgeriou, P., & Zdun, U. (2005). Architectural patterns revisited-a pattern language. *Proc. 10th European Conf. Pattern Languages of Programs (EuroPLOP)*, pp. 431-470
- [18]. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997, June). Aspect-oriented programming. In *European conference on object-oriented programming* (pp. 220-242). Springer, Berlin, Heidelberg.
- [19]. Tennyson, M. F. (2010, October). A Study of the data synchronization concern in the observer design pattern. In *2010 2nd International Conference on Software Technology and Engineering* (Vol. 1, pp. V1-71). IEEE.
- [20]. Jicheng, L., Hui, Y., & Yabo, W. (2010, July). A novel implementation of observer pattern by aspect based on Java annotation. In *2010 3rd International Conference on Computer Science and Information Technology* (Vol. 1, pp. 284-288). IEEE.
- [21]. Noda, N., & Kishi, T. (2001, October). Implementing design patterns using advanced separation of concerns. In *Workshop on Advanced Separation of Concerns at OOPSLA2001*.
- [22]. Piveta, E. K., & Zancanella, L. C. (2003). Observer pattern using aspect-oriented programming. *Scientific Literature Digital Library*.
- [23]. Borella, J. (2003). The observer pattern using aspect oriented programming. *Proceedings of the Viking Pattern Languages of Programs, Viking PLOP*.
- [24]. Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, 75-105.
- [25]. Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, 24(3), 45-77.

- [26]. Johannesson, P., & Perjons, E. (2014). *An introduction to design science*. Springer.
- [27]. Mijač, M. (2019). Evaluation of Design Science instantiation artifacts in Software engineering research. In *Central European Conference on Information and Intelligent Systems* (pp. 313-321). Faculty of Organization and Informatics Varazdin.
- [28]. Prat, N., Comyn-Wattiau, I., & Akoka, J. (2015). A taxonomy of evaluation methods for information systems artifacts. *Journal of Management Information Systems*, 32(3), 229-267.
- [29]. Microsoft. (2019). .NET Guide. Handling and Raising events. Retrieved from: <https://docs.microsoft.com/en-us/dotnet/standard/events/> [accessed: 15 June 2020].
- [30]. Oracle. (2020). Java Documentation. Introduction to Event Listeners. Retrieved from: <https://docs.oracle.com/javase/tutorial/uiswing/events/intro.html> [accessed: 15 June 2020].
- [31]. Goodrich, M. T., & Tamassia, R. (2015). *Algorithm design and applications* (p. 349). Hoboken: Wiley.
- [32]. Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, 5(11), 558-562.
- [33]. Tarjan, R. E. (1976). Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2), 171-185.
- [34]. March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, 15(4), 251-266.
- [35]. Sommerville, I., & Alfonso Galipienso, M. (2011). Ingeniería de Software (Novena edición ed.). *MI Alfonso Galipienso, A. Batía Martínez, F. Mora Lizán, & JP Trigueros Jover, Trads.* Ciudad de México, México: Pearson Educación, SA.