Complex propagation of events: Design patterns comparison

Marko Mijač *marko.mijac@foi.hr*
University of Zagreb, Faculty of Organization and Informatics
Varaždin, Croatia

Dragutin Kermek *dragutin.kermek@foi.hr*
University of Zagreb, Faculty of Organization and Informatics
Varaždin, Croatia

Zlatko Stapić *zlatko.stapic@foi.hr*
University of Zagreb, Faculty of Organization and Informatics
Varaždin, Croatia

Abstract

Today's applications require high level of interactivity and the applications are expected to update their state, and provide users with immediate correct results. In high complex applications this usually means updating states of a number of objects that are part of complex dependency network. To decouple these dependencies such systems usually employ implicit invocation mechanisms such as well-known Observer pattern. However, building efficient propagation system that will keep object states up-to-date is a challenging task, since a number of issues arise. In this paper we identified 18 qualitative criteria and compared 5 design patterns that can be used to build propagation system on top of applications' business objects. The exhaustive comparison results are given along with discussion and remarks that should be taken in consideration when dealing with this challenging task.
Keywords: Design patterns, Observer, Mediator, Events, Propagation, Publish-subscribe, Change manager, Implicit invocation, Reactive systems.

1. Introduction

Object oriented paradigm has been acknowledged as a dominant paradigm for developing complex applications for a quite some time. These applications can have large number of objects that are greatly interdependent. That means that object's state and computations are dependent on one or more other objects. Such dependencies can form a large and complex dependency network and such application usually requires high level of interactivity which also adds to its overall complexity. Moreover, users expect to receive immediate results whenever they change any of input parameters. In such system, the main issue around implementing interactivity is in handling dependencies between related objects and keeping the states of these objects synchronized and continuously up-to-date. Building propagation system that will efficiently traverse dependency network, and update required objects is a challenging task, since a number of issues arise. Maier et al. [13] claim that contrary to traditional batch mode programs, "interactive applications require a considerable amount of engineering to deal with continuous user input and output".

There are several design patterns (first of them introduced by Gamma et al. [9]) that are designed for building interactive systems with propagation. They all share the common idea of *implicit invocation* and *asynchronous communication* between dependent objects. Implicit invocation, according to Avgeriou and Zdun [1] and Eugster et al. [7], offers several advantages (such as loose coupling, dynamic adding and removing of dependent entities and components during runtime etc.) over a point-to-point and synchronous communication which

leads to rigid and static applications. In their paper, Maier et al. [13] state Observer pattern to be predominant approach for managing state changes, while Szallies [18] discusses that the Observer pattern introduces an additional level of indirection and blurs state dependencies between objects, which increases flexibility but decreases the understandability and performance of the code.

In this paper we examined and compared design patterns suitable in solving the issues of event propagation and management of dependencies between objects in complex systems. Second section describes the methodology that was used in our research, in the third and fourth section we present and discuss the results respectively, while in final section we drew conclusions.

2. Methodology

In order to address the issues of event propagation and management of dependencies between objects, our intent was to compare existing, explicitly documented design patterns. The first step was to identify design patterns which are intended to deal with *propagation*, *events*, *reactive* and *interactive behavior*, and *implicit invocation*. We performed exhaustive search in several iterations on the following scientific databases: IEEE Xplore, Google Scholar, ACM Digital Library, SCOPUS. Initially, for defining search keywords well-known patterns such as Observer and Publish/Subscribe have been taken as a reference point, so an initial set of search keywords was as follows: *design pattern, subject, observer, event, publish, subscribe*. Since conducted search did not result in a desired number of patterns, other search iterations were performed. The set of keywords expanded in each iteration, so in the final iteration in addition to already mentioned keywords, it contained a significant number of other keywords (*propagation, event notification, reactive, interactive, dependency network, implicit invocation, inversion of control, state changes*), which we combined when constructing search queries.

Despite quite broad and thorough search, we managed to identify only 5 explicitly described design patterns/variants of design patterns: Observer Pattern (simple) [9], Observer Pattern (advanced) [9], Observer Pattern revisited [6], Event-notification pattern [16] and Propagator pattern [8].

Since our research is focused on design patterns in imperative object-oriented paradigm, we did limit our results to design patterns in this paradigm. That said we excluded the results from functional, declarative, aspect-oriented and reactive paradigms. Also the patterns focused on distributed environments (e.g. Publish/Subscribe [7], [1], Event Notifier [11], CORBA notification/event service) were excluded.

Table 1. Qualitative comparison criteria

Criteria		
General attributes	Intent	
	Also known as	
	Related patterns	
	Structure	
Event propagation related characteristics	Object roles: event emitter/receiver	Event granulation
	Dependency network location	Order of propagation
	Dependency network structure	Direction of propagation
	Response to event	Cut-off propagation
	Event data	Acyclic graph handling
	Coupling	Cyclic graph handling
	Subject and Observer relationship	Events composition and filtering

In order to qualitatively compare identified design patterns, we formed a set of 18 qualitative criteria as presented in Table 1. Among considered criteria the first four are proposed by Gamma et al. [9], and are traditionally used to describe design patterns. These will allow us to compare general ideas behind design patterns, to highlight structural differences between them, and to show if and how they relate to other design patterns. Other 14 criteria were chosen in order to differentiate design patterns according to characteristics closely related to the issues of event propagation. They aim to enhance overall understanding of the event propagation problem, and to differentiate design patterns according to their capabilities in this matter. The 14 criteria were carefully extracted from papers which described identified design patterns. Also, papers from aforementioned other paradigms (functional, declarative, reactive programming…) were taken into consideration when forming and justifying this set of criteria.

3. Results

Following the stated methodology, after obtaining the literature review results, the five design patterns and the papers in which they were initially presented were analyzed in detail. The data about these patterns has been extracted according to the set of previously identified comparison criteria and the results are presented in Table 2. Columns contain data regarding characteristics of particular design pattern, while rows present particular comparison characteristics across different design patterns.

Table 2. Results of qualitative comparison

	Observer pattern (Simple) [9]	**Observer pattern (Advanced)** [9]	**Observer pattern revisited** [6]	**Event-notification pattern** [16]	**Propagator pattern** [8]
Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.			Manage update dependencies between objects by introducing an event notification mechanism.	Define a network of dependent objects so that when one object changes state, all direct and indirect dependents are updated.
Also known as	Dependents, Publish-subscribe	Dependents, Publish-subscribe	-	Implicit Invocation Mechanism	Cascaded Update
Related patterns	-	Mediator, Singleton	Mediator	Observer, Mediator	Composite, Observer, Mediator, Strategy
Structure	Subject and Observer	Subject, ChangeManager and Observer	Observable, ObserverManager, Observer	Observer, Subject, EventStub, StateChange.	Propagator object
Object roles: event emitter / receiver	Depends on implementation (Abstract class or interfaces)	Depends on implementation (Abstract class or interfaces)	Claimed. Both Subject and Observer are expressed as Java Type object.	Depends on implementation (Abstract class or interfaces)	Yes (Propagator)
Dependency network location	Distributed. Subscriptions contained in each Subject.	Centralized. Stored and managed by ChangeManager object.	Centralized. Stored and managed by ObserverManager.	Distributed. Stored in Subject's StateChange objects.	Distributed. Propagator object can contain lists of both predecessors and dependents.
Dependency network structure	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Response to event	Observer's update method.	Observer's update method.	-	One or more arbitrary methods in Observer, as EventStub objects.	Propagator's update method.
Event data	No additional data (Update method has no parameters)	Update method can contain parameters for defining Subject and aspects.	Parameters. Combines push and pull model.	Possible parameters that indicate Subject, event identification, changed data...	Update method can contain parameter indicating predecessor object.
Coupling	Subject and Observer are loosely coupled (to interfaces or abstract classes)	Subject and Observer are loosely coupled (to interfaces or abstract classes)	Decoupled. Observable and Observer are stored in ObserverManager	Subject and Observer decoupled by using EventStub and StateChange objects.	Propagators are loosely coupled (to abstract class).

			as instances of base Java Object.		
Subject and Observer relationship	One-to-many	One-to-many (SimpleChangeManager), Many-to-many (DAGChangeManager)	Many-to-many	Many-to-many	Many-to-many
Event granulation	One event per Subject, one update method per Observer.	Suggested use of aspects to achieve higher granulation.	One event per Observable, one update method per Observer.	Multiple events per Subject, multiple update methods per Observer.	One event per Propagator, one update method per Propagator.
Order of propagation	Depth-first	Depth-first	Depth-first	Depth-first	Depth-first
Direction of propagation	Forward	Forward	Forward	Forward	Forward/Backward
Cut-off propagation	Possible	Possible	Possible	Possible	Yes
Acyclic graph handling	No	Yes, topological sorting or breadth first order of propagation.	Claimed, by maintaining the list of visited objects.	No	Yes, topological sorting, smart propagation.
Cyclic graph handling	No	No	Claimed, by maintaining the list of visited objects.	No	Yes, topological sorting, smart propagation, graph marking.
Events composition and filtering	No	No	No	No	No

4. Discussion

4.1. Pattern description and general attributes

All examined patterns explicitly state their purpose as managing dependencies between objects, and therefore share the same intent and idea. They do however differ in various implementation and design aspects. The first three patterns can be considered variants of Observer pattern, while Event-notification and Propagator pattern evolved more or less independently and can be considered as separate patterns. However, Riehle mentions three variants of Observer pattern which include Propagator and Event-notification Pattern as well [15].

Mixture of names used to denote these patterns shows a mess in the nomenclature. A lot of design pattern names are used interchangeably to describe different design patterns or different terms. For example Riehle [16] states Event-notification pattern to be also known as Implicit invocation mechanism, which should rather be considered an architectural style than design pattern. Also, some design patterns had synonyms that are now deprecated.

Our focused patterns are implementing inversion of control mechanisms, which are according to Gasiunas et al. "essential for improving stability and reusability of software systems" [10]. They also state implicit invocation to be a major technique for inversion of control, which is again usually implemented by the Observer pattern. If we take a look at the structure of compared design patterns (see Fig. **1** to Fig. 5), we can see that they follow a general idea of Observer pattern. We can identify two basic roles: entities that emit events and entities that receive these events, and react to them. For a purpose of clarity in the following table we show terms denoting these roles in different patterns:

Table 3 Roles in design patterns

Pattern	**Event emitter role**	**Event receiver role**
Observer (Simple)	Subject	Observer
Observer (Advanced)	Subject	Observer
Observer revisited	Observable	Observer

Event-notifier	Subject	Observer
Propagator	Propagator	Propagator
Other terms:	*Publisher, Notifier, Broadcaster, Producer,*	*Subscriber, Reactor, Dependent, Listener, Consumer*

By looking at the structure of Simple Observer pattern (see Fig. **1**) described by Gamma et al. [9], we can see that two above mentioned roles dominate. Subject represents an object which emits events (notifications) about its state changes, knows whom to notify about these changes, and provides an interface to attach and detach Observers. Observers are objects which are interested in state changes that occur in Subject. They have a responsibility to subscribe to Subject's events and react on them. To achieve higher level of decoupling the roles of Subject and Observer are in this pattern implemented as base classes, which are then inherited by ConcreteSubject and ConcreteObserver. Base class implementation is very common, but implementation with interfaces, composition or combination of these concepts is also possible.

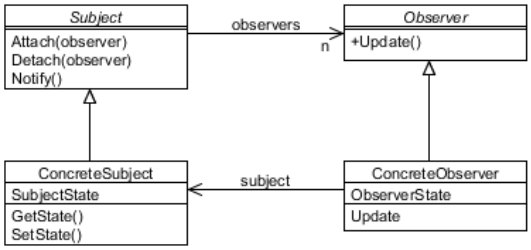

Fig. 1 Simple Observer Pattern [9]

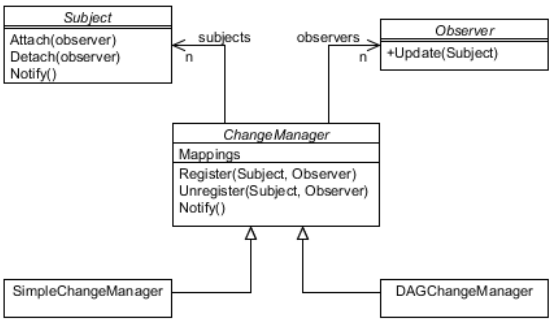

Fig. 2 Advanced Observer Pattern **[9]**

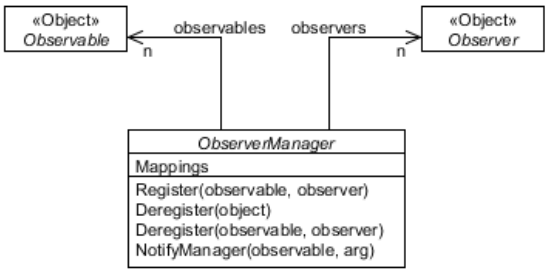

Fig. 3 Observer pattern revisited **[6]**

Gamma et al. were aware of the limitations of simple version of Observer pattern, so for complex use of this pattern they proposed some improvements and modifications (see Fig. 2). They introduced a third role in Observer pattern – change manager – a central object for managing dependencies between Subjects and Observers. Subject and Observer role are still existent, but this time dependencies between them are not kept in each Subject, but in one central place – hash table inside ChangeManager object. This allows a global management of dependencies and possibility to deliver optimizations and overall improvements by implementing propagation strategies.

In order to improve the Observer implementation, Eales [6] proposed ObserverManager – a central object for controlling a whole lifecycle of Observer-Observable relationships, ensuring no dangling references occur. Also, it manages update process, avoiding multiple redundant updates and cycles. Observer and Observable objects are expressed as Java base type Object.

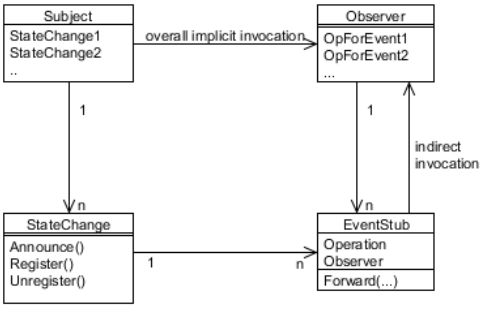

Fig. 4 Event notification pattern [16]

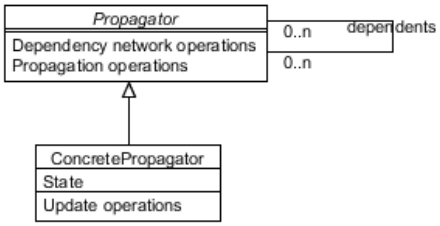

Fig. 5 Propagator pattern **[8]**

Explicit modeling of abstract state and dependencies, increased decoupling of Subject and Observer and selective registration possibility are some novel concepts that are presented in Event-notification pattern (see Fig. **4**) [16]. Subject continues to have responsibility to manage its state, and to publicly expose state changes as StateChange objects. It is now the responsibility of these StateChange objects to register, unregister and notify interested Observers through EventStub objects about state changes. EventStubs are Observer's first class objects who know which operation of Observer should be called in case of StateChange invocation. This structure has some important consequences in addressing the issue of event granulation.

Propagator pattern described by Feiler and Tichy [8] actually represents a family of patterns, consisting of: Strict propagator, Strict propagator with failure, Lazy propagator, and Adaptive propagator. Different variants share common structure, but differ in propagation strategy. Feiler and Tichy [8] see interconnected objects in application as nodes in dependency network. To be a part of dependency network, an object must inherit Propagator class (see Fig. 5), which handles all dependency network related operations and propagation operations as well. Propagator class encompasses at the same time roles of Subject, Observer, and change manager, making the structure of Propagator pattern much simpler than the structure of Observer pattern. However, authors still leave the possibility to implement global change manager as a separate entity using Mediator pattern.

Other design patterns usually participate only in more complex implementations of here considered design patterns. Such is the case with implementations containing change manager role. According to Gamma et al. [9] by encapsulating complex semantics change manager acts as mediator between Subject and Observer, so here we can utilize Mediator pattern. Same authors also propose use of Singleton pattern in change manager implementation, to make it globally accessible and ensure existence of only one manager instance. Such claims are also supported by Eales [6], Riehle [16], Feiler and Tichy [8]. In addition to Mediator and Singleton pattern, according to [8], with Propagator pattern also Composite pattern and Strategy pattern can be associated.

4.2. Event propagation related characteristics

The structure of dependency network consists of a number of objects that are interdependent, so it is a common situation for an object to depend on other objects, but also to influence other objects. Therefore, an object can at the same time encompass both: the **roles of event emitter and event receiver**. This is easily realized in Propagator pattern, since its Propagator role naturally encompasses roles of both event emitter and receiver. In other design patterns, these two roles are clearly separated as Subject (Observable) role and Observer role. However, by applying different implementation approaches, one can accomplish to merge these roles into one object. For example, in Observer revisited pattern as claimed, Observable and Observer objects are implemented as base Java objects, so they can play both roles at the same time. In other design patterns this could be accomplished by implementing at least one of the roles as class interfaces, since multiple-inheritance is rarely supported in modern OO

programming languages. We believe that in complex systems, where dependency networks are formed, it is not convenient to threat event emitter and receiver as separate roles, so a solution like in a Propagator pattern would be more appropriate.

The **location of dependency network** states where the subscriptions to events are located. Regarding to this we identified two types of dependency network: *distributed* and *centralized*. Distributed dependency network implies that subscriptions to events of particular event emitter are contained within that very emitter. On the other hand, centralized dependency network assumes that subscriptions to all events are held in central change manager object in some kind of hash table. This characteristic has some important implications regarding propagation performance and optimization. For example, centralized option with Change manager implementation is more advisable when dealing with complex and large dependency networks, because it keeps all dependencies between objects at one place, which makes it easier to create propagation strategies and apply optimizations. However, compared with distributed option, it additionally blurs dependencies between objects. Advanced variant of Observer pattern and Observer pattern revisited natively implement centralized dependency network, while other design patterns implement distributed option. However, the authors of Propagator pattern and Event-notification pattern also claim change manager implementation as possible.

In dependency network, dependencies between objects are created or destroyed dynamically (at runtime), so it is imperative for design pattern to support **dynamic structure of dependency network**. In patterns implementing the role of change manager this is usually done centrally in change manager itself, while in others it is responsibility of event emitter (e.g. Subject) to maintain a list of its receivers (e.g. Observer). However in both cases event receivers have the role to initiate the process of subscribing and unsubscribing from event.

Responding to event is the responsibility of event receiver, which appoints one of its methods to react to event. In all patterns except Event-notification pattern this is method is hardcoded in design (Update method), while in Event-notification pattern arbitrary method can be assigned to one or more of Subject's events. Hardcoding a method that will be responsible for reacting to event greatly reduces the flexibility, because we react to possible several different events with the same method. So, although it increases complexity of implementation, Event-notification pattern has the advantage in this case. However, instead of implementation with EventStubs and StateChange objects, one could preferably go with simpler implementations with delegates.

In order to appropriately react to event, receiver often needs to **retrieve additional data** from event emitter. Here we can differentiate two models: push model and pull model [5]. With push model, emitter sends additional data to receiver along with event notification. Receiver receives this data whether it needs it or not, usually as parameters of update method. On the other hand, with pull model receiver only receives event notification, and if additional data is required it is pulled from emitter by receiver. Both models have their own advantages and disadvantages, and the best model to choose depends on particular situation. A question of when to use parameterization of event notification (push model) and when not to (pull model), is also discussed by Riehle [15].

Loose coupling is generally recognized as a factor influencing quality characteristics of software (such as reusability and maintainability). Achieving loose coupling between interconnected objects and components is one of the reasons for utilizing Observer and related patterns. Here, this is done by employing principles of implicit invocation and coupling object on interface or abstract class level. For example, all that Subject needs to know about object it notifies is that it implements Observer interface. On the other hand, the relationship between Observer and Subject can be stronger, especially if Observer needs to pull some specific data from Subject. Although all compared design patterns by default help in achieving loose coupling between dependent objects, Event-notification pattern goes a bit further by introducing EventStub and StateChange objects as an additional level between event emitters and receivers, thus lowering coupling even more.

All focused patterns imply the ability of handling one-to-many **relationship** between event emitter (i.e. Subject) and event receivers (i.e. Observers). More complex scenarios often

require Observer to have the ability to listen more than one Subject, i.e. many-to-many relationship. Although it can be achieved in all considered patterns, this should be approached with caution. Having objects which depend on possibly many other objects, and at the same time influence multiple other objects can result in quite complex network of dependencies. If not properly handled propagation of events in such cases can lead to acyclic behavior, redundant updates and glitches. Gamma et al. [9] also suggest caution here, and suggest the use of their DAGChangeManager instead of SimpleChangeManager.

All considered patterns except of Event notification pattern support one event per Subject (**event granulation**), and one Update method per Observer. Event notification pattern [16] supports explicitly exposing more than one event per Subject through its StateChange objects. Similarly, through EventStub objects allows definition of more Update methods. Gamma et al. [9] also recognize this requirement, and they propose a workaround by using aspects to achieve higher granulation. In this case Subject and Observer still have only one event and one update method, but the user can specify an aspect in which it is interested. Although the original version of Propagator pattern does not support multiple events per object, Boeker [3] in his web article describes implementation of Propagator pattern as an alternative to Observer pattern, and improves it by introducing state change objects, similar to ones in Event notification pattern. Ability to offer several different events per object is a feature which increases flexibility. The examples of such objects with multiple available events can be found in most modern OO frameworks (e.g. Java and .NET frameworks).

An order in which the changes in dependency network are propagated (**order of propagation**) has a significant impact on performance and the correctness of propagation. Feiler and Tichy [8] identify two ways in which changes can propagate through dependency network: Depth-first and Breadth-first propagation. These are analog to well-known search algorithms which are thoroughly analyzed in the literature, for example in [4]. In Depth-first propagation method a changed node always notifies the first direct dependent, which then notifies its first direct dependent, and so on until leaf nodes are reached. Only then initially changed node notifies the second and the other direct dependents. Alternatively, in Breadth-first propagation, a changed node first notifies all direct dependents and only then passes to another level. Depth-first propagation is by default employed by all considered design patterns. However, this method can result in some significant issues such as redundant updates, poor performance, incorrect results, inconsistent data (also called *glitches* [2]). This can happen for example when a node has more than one predecessor, which is quite common situation in dependency networks. Therefore it is essential to appropriately address these issues. Feiler and Tichy [8] and Maier et al. [13] propose topological sorting of dependency network when implementing Depth-first method. Alternatively, implementation of Breadth-first propagation method is going to have the same effect as topological sorting.

Besides order of propagation a **direction of propagation** should also be considered. Feiler and Tichy [8] identify two types of propagation depending on direction: forward (immediate, eager) propagation and backward (on demand) propagation. They also propose different variants of Propagator pattern implementing these approaches: Strict Propagator and Lazy Propagator. Except for the Propagator pattern which supports both approaches, other considered design patterns employ only forward propagation.

In a large and complex dependency network propagation of changes can be quite computationally demanding, so wherever possible optimization steps should be applied. **Cut-off propagation** denotes a simple and obvious optimization step of comparing new value of object's state with current. If both current and the new state are the same, then we can stop (cut-off) the propagation. This optimization step is explicitly supported in Propagator pattern, referred as "smart propagation" [8] and "intelligent adjustment of states" [14]. As a means of implementing cut-off, Feiler and Tichy [8] involve the use of Memento pattern; however, contrary to complex Memento pattern, simple comparison between attribute's current value and the upcoming new value is often good enough. Although not explicitly stated, other considered design patterns can also easily implement this optimization, which we strongly encourage. This is a simple optimization step; however, in complex scenarios more sophisticated optimizations could be required, such as the possibilities to create smart and

adaptable propagation strategies. This poses a great challenge, since the structure of dependency networks is highly dynamic (object can be added or removed from dependency network at any time, the same is with relationships between objects).

Interdependent objects in dependency networks can often form acyclic or even cyclic graphs. Multiple redundant updates, "gliches", inconsistencies, infinite loops are some of the issues that indicate the existence of such circular graphs in one's application. As already mentioned, it is essential to address these issues in order for dependency network to function properly.

Some of the authors recognized this problem, and included recommendations on how to avoid these issues in particular design patterns. Gamma et al. [9] recommended using DAGChangeManager implementation of Observer pattern (instead of Simple observer pattern and SimpleChangeManager implementation) when dealing with possible acyclic dependency network, aiming at prevention of multiple redundant updates. The order of propagation is crucial in avoiding inconsistent state which arises from acyclic nature of dependency network. Therefore proper measures should be assured, such as aforementioned topological sorting of dependency graph, or employing a Breadth-first method of propagation. Eales [6] claims his Observer pattern implementation avoids multiple updates and cycles by "viewing the update process as graph traversal which maintains a list of visited objects", although no implementation specifics are provided. Feiler and Tichy [8] in their Propagator pattern propose topological sorting and "smart propagation" as a means of dealing with acyclic graphs. In contrast, Riehle [16] in his Event Notification Pattern doesn't specify particular strategies to deal with acyclic graphs.

Furthermore, dependency network can contain **cyclic graphs**, which are causing infinite loops during propagation. To prevent this behavior, infinite loop must be manually stopped as soon as dependency network reaches consistent and correct state, and perhaps some additional specific condition is met. Determining a point is safe to be stopped is challenging, since we need to monitor a progress of propagation at all time, and be aware of all objects and dependencies involved in particular propagation instance. Eales [6] states his Observer pattern revisited avoids circular behavior by maintaining a list of visited objects, while Feiler and Tichy [8] propose topological sorting, "smart propagation" and graph marking when dealing with cyclic graphs.

In their original form, none of the examined design patterns supports **composition and filtering of events**. However, in complex environments, the abilities to compose new events from existing ones and to fire or react to events only under certain circumstances (filtering) would be highly desirable. Such features were already pointed out in declarative and reactive approaches ([13], [2], and [10]), and would also be useful in imperative programming.

4.3. Utilizing design patterns

Since none of the considered patterns possess all useful features, it is probable that features from different patterns will have to be combined in order to construct satisfactory solution. Therefore, as it can be seen in Table 4, we identified advanced features and we give guidelines for implementing them by using one or combining more design patterns.

Table 4 Guidelines for implementation of advanced features

Feature	Guidelines
Dynamic dependency network	All five patterns apply.
Object can both emit and receive events	See Propagator role in Propagator pattern [8], or consider implementing emitter/receiver as class interfaces.
Objects' dependencies are held centrally	See implementation of ChangeManager [9] and ObserverManager [6].
Event can be handled by arbitrary method	See implementation of EventStub objects in [16].
Expose multiple events by the same emitter	See implementation of StateChange objects in [16].
Emitter send additional data to receiver	Push additional data to receiver as parameters, or let receiver pull required data from emitter. A combination is also possible.
Loosely coupled dependencies between objects	All five patterns apply. For additional level of decoupling see implementation of EventStub and StateChange objects in [16].

Receiver depends upon multiple emitters	All patterns except for simple Observer pattern apply. However, be sure to address the possible acyclic behavior.
Dependency network forms acyclic graph	Watch for the order of propagation. Be sure to apply breadth-first order of propagation, or a topological sorting of dependency network in case of depth-first order. See DAGChangeManager [9]
Dependency network forms cyclic graph	Apply advanced techniques such as graph marking (maintaining the list of visited objects), topological sorting, and "smart propagation" to break up the loop and avoid redundant updates.
Propagation performance optimization	Apply cut-off propagation step. Keep objects' dependencies in one central location (hash map) by implementing change manager object. Also, make sure possible acyclic and cyclic behaviors are properly handled, so no redundant updates occur. For most complex scenarios consider developing "smart" and adaptable propagation strategies.
Composition and filtering of events	None of the considered patterns provides such capabilities, however some implementation guidelines and ideas can be taken from declarative and reactive approaches (e.g. [1], [15], and [11]).

The table summarizes the presented and discussed comparison results and gives guidelines on how to implement the features ranging from those supported by all five design patterns to those that require special attention in combination and update of compared patterns.

5. Conclusion

Conducted comparison showed a great similarity between considered design patterns, especially in their overall idea and intent. All design patterns assume the existence of event emitter and event receiver roles, either in separate or in a single object. In a number of considered criteria mentioned design patterns are quite uniform. Such is the case with the ability to dynamically change dependency network, order and direction of propagation, ability to cut-off propagation etc. However, significant differences can be seen in their structure, possibilities and implementations.

Simple Observer pattern is quite modest in its capabilities, and we find it not suited for handling complex propagation cases. Others do satisfy a number of criteria and employ some advanced features, which are common to more design patterns or unique to particular pattern. It appears however, that no design pattern has all desired features.

By comparing these design patterns we managed to recognize several features that should be considered when dealing with complex propagation scenarios: An object should be able to be both Subject and Observer at the same time; It should be possible to choose the appropriate level of event granulation (e.g. one or more events per object, one or more events per component); An object should be able to have zero or more observing objects, while at the same time being able to observe zero or more other objects; It should be possible to selectively register to only some of available events; To additionally decouple Subject and Observer, implementation of events as objects (possibly first-class objects) should be considered; Events should be able to pass certain data (parameters) to the listening object; Adaptable update strategies that allow consistent, non-redundant, fast and scalable update should exist; In a case where one strategy is not sufficient, multiple strategies should be able to coexist; Update strategies should be able to deal with acyclic and cyclic dependency graphs and provide a way to break update loop; Central mediator object (change manager) that manages dependencies between objects and update strategies should be considered; Composition and filtering of events should be considered; Possibility to adjust settings of update process, such as commencing update manually or automatically, updating dependency network partially or entirely; Possibility to perform updates utilizing multithreading options; and Possibility to properly handle update failure should be considered.

A number of individual features can be rather easily implemented and handled. However, building a system which will be characterized by the ability to update complex network of interdependent objects in a fast, flexible, consistent, and scalable manner is a much harder task with different emerging issues that are mentioned in the discussion part of the paper.

Resolving aforementioned issues presents a possibility to expand this topic and conduct further research, which should result in building adequate model and framework for events

propagation. Approaches covered in this paper are based on imperative programming paradigm, so a comparison with approaches developed in other paradigms would be valuable. This is especially the case with solutions presented in the area of reactive programming and reactive systems based on declarative/functional programming (e.g. [10], [17]), aspect-oriented programming (e.g. [12]) and hybrid approaches. Although omitted from this study, propagation of events in distributed environment brings even more challenges and also possibilities for future research.

References

1. Avgeriou, P., Zdun, U.: Architectural Patterns Revisited – A Pattern Language. In: Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005). , Irsee, Germany (2005)
2. Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A Survey on Reactive Programming. ACM Comput. Surv. 45 (4), (2013)
3. Boeker, M.: Propagator in C# - An Alternative to the Observer Design Pattern, *CodeProject*, http://www.codeproject.com/Articles/38108/Propagator-in-C-An-Alternative-to-the-Observer-Des, Accessed: July 10, 2013, (2009)
4. Bratko, I.: Prolog Programming for Artificial Intelligence. Addison Wesley (2000)
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented software architecture: System of patterns. John Wiley & Sons, Chichester (1996)
6. Eales, A.: The Observer Pattern Revisited. Presented at the Educating, Innovating & Transforming: Educators in IT, (2005)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The Many Faces of Publish/Subscribe. ACM Comput. Surv. 35 (2), 114–131 (2003)
8. Feiler, P., Tichy, W.: Propagator: A family of patterns. In: Technology of Object-Oriented Languages and Systems. IEEE (1997)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
10. Gasiunas, V., Satabin, L., Mezini, M., Nunez, A., Noye, J.: Declarative Events for Object-Oriented Programming. Technische Hochschule, Darmstadt (2010)
11. Gupta, S., Hartkopf, J., Ramaswamy, S.: Event Notifier, a Pattern for Event Notification. Java Rep. 3 (7), (1998)
12. Jicheng, L., Hui, Y., Yabo, W.: A novel implementation of observer pattern by aspect based on Java annotation. In: Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on. pp. 284–288. IEEE, Chengdu (2010)
13. Maier, I., Rompf, T., Odersky, M.: Deprecating the observer pattern. Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland (2010)
14. Rege, K.: Design patterns for component-oriented software development. In: 25th Proceedings of EUROMICRO conference. pp. 220–228. IEEE Computer Society, Milan (1999)
15. Riehle, D.: Describing and composing patterns using role diagrams. Presented at the WOON, (1996)
16. Riehle, D.: The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation. Presented at the TAPOS 2.1, (1996)
17. Salvaneschi, G., Mezini, M.: Reactive Behavior in Object-oriented Applications: An Analysis and a Research Roadmap. Presented at the AOSD, Fukuoka, Japan (2013)
18. Szallies, C.: On Using the Observer Design Pattern. (1997)