Pattern Language for User Information feed-back

By Christophe Addinquy (Christophe.Addinquy@valtech.fr)

Abstract

The User Information Feedback pattern language gives patterns for event (error and information) reporting and management. This pattern language describes the importance of separating error signalling, context memorization, event production and event publication on any media.
On one hand, this pattern language is an extension of Neil Harrison's "Patterns for logging diagnostic messages" [Harrison98]. It suggest some ways for message identification, and their dispatching and filtering on different media.
On the other hand, this pattern language links to Ward Cunningam's "Checks Pattern Language" [Cunningham95].

Pattern language overview

Context

These patterns are appropriate for software that gives continuous information about operation completion, warning or dysfunction. This information may be reports of high level events, or may be low level, such as program traces.

Problem

Interactive software as well as transaction oriented software must give back information about operation success or failure. Even if this kind of feedback is not the application's main work, it helps to make the user more confident in using the application, and more capable of dealing with errors.
Interactive software as well as transaction oriented software must give back information about operation success or failure. Even if this kind of feedback is not the application's main work, it helps to make the user more confident in using the application, and more capable of dealing with errors.

Presentation

Important note: This pattern language uses widely UML with, occasional minor extensions, as well as some other notations. Please, refer to the "Notations" appendix at the end of the paper for the needed explanations.

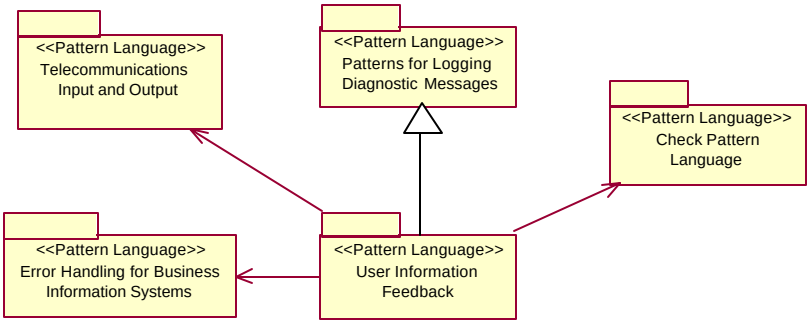


Figure 1:Relationships between User Information Feedback Pattern Language and connected Pattern Languages

This pattern language is built on the same way than Neil Harrison's Patterns for Logging Diagnostic Messages. It may be considered as a specialization of this former Pattern language to create identifiable and elaborated messages.

The pattern language is also close to Ward Cunningham's Check Pattern Language, which essentially takes care of input validations. User Information Feedback extends this behavior.
The Telecommunication Input and Output Pattern Language[Hanmer+98] deals with a number of the same problems faced here. The concrete solutions may differ, but the approaches to the solutions are similar.

Language map

This Pattern language presents five linked patterns.

- **Event Type** is the Pattern Language cornerstone. It explains how to build identifiable events. Once identifiable events are created, events can be handled or filtered in several different ways.
- **Media Dispatcher** pattern is the way by which the Event types can be presented, filtered and processed.
- To be presented clearly and efficiently on various media, events must be expressed as **Focused Messages**. This pattern offers a guideline for event messages redaction. But events are not just standard messages. To really add value, events should contain details about event occurrence.
- The **Part-Made Context** pattern address this constrains: it helps the application to remember volatile information, which may become "clues" about what happened when information or errors are reported.

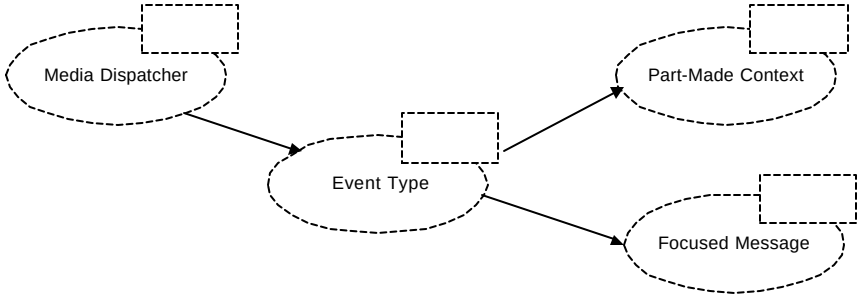


Figure 2: Relationships between the User Information Feedback Patterns

Pattern Language boundaries

The pattern language domain can be considered as an information flow management. So, we can point out two main limits are an upstream limit, where information is created, and a downstream limit where information is concretely recorded or displayed

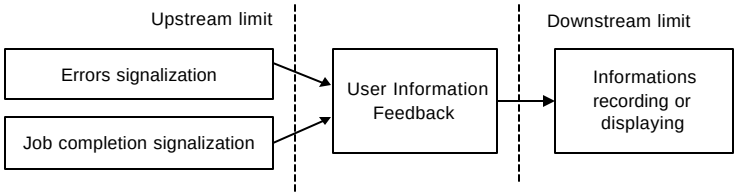


Figure 3: Relationships between the pattern language and its boundaries

Downstream limit

The "what is done with information feedback" problem is not addressed by this pattern language. Things like "how events are serialized on file logging?" or "how events are displayed " are beyond the scope of this pattern language. Our purpose is make it possible to display and record meaningful event information. So, for example, we make the data available in a form that in can be displayed or serially recorded.

Upstream limit

Here, we consider the question of the events origin, especially for errors. It's often possible to make the error production and the event production the same thing, and it's sometime done. At this point, we must consider two problems:

1. Errors are directly used as reported events.
2. Error production point is the same than event production point.

As suspected, the first point is more problematic than the second. Let's take a look at these two points.

Why errors must not be directly used as reported events.

Take, for example, the Model-View-Controller architecture used in many OO systems. We believe that a user view may introduce visualization differences from the original model. The same is true with errors: users are not interested in a display of the internal behavior of the system. By considering this kind of information feedback as a view on errors, we respect the separation of a user oriented view built upon model oriented data.

Why we must consider different points for error production point and for event production point.

Even if Information feedback is separated from error production, you may choose the same point report errors and to produce information events. By doing so, you take several risks:

- A huge number of event production points will probably appear. It will make the code cluttered by event production points, with probably a lot of production points for the same events.
- You will probably have a great number of events produced, with several events for each error. At worst, these events may produce different and unrelated messages for a given error.
- Libraries and third party subsystems can't deal with our information feedback system. So, anyway, you must consider different point for errors production and information feedback creation.

Therefore, consider different points for errors production points and for information feedback creation. Limit as much as possible the number of these information feedback creations points, with several error paths leading to these points. Such a point become a strategic point for problem detection, ad may be good choice for debugger breakpoints.

Resulting connection between errors and user events.

Now, we have a clear separation between error management and information feedback management, from an information-handled viewpoint as well as from a production point consideration.

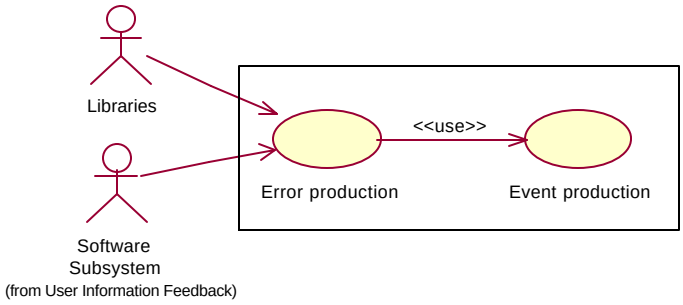


Figure 4: Transition between Error handling and the corresponding event generation.

The former diagram summarizes the situation: Libraries as well as our software subsystems generate errors when a fault is detected ([Renze99], p. 7). These errors are propagated backward up the calling stack ([Renzel99], p. 21). When convenient, the errors may be cached, and a corresponding information event is created.

Upstream limit related Patterns

- **Exceptional value** [Cunningham95]: This alternative Pattern suggest to leave a place to exceptional value. We can use this pattern to make a value "tagged" as error.
- **George Washington is Still Dead** [Hanmer+98]: Deals with error reporting redundancies. The alternative solution proposed is to create message only on the state changes.
- **The Bottom Line** [Hanmer+98]: This pattern takes care about number of messages with the same origin. Solution differs by grouping these messages instead of creating messages at the origin.

Five Minutes of No Escalation Messages [Hanmer+98]: This pattern also deals with output pollution, especially in abnormal situations. It proposes a policy to reduce the message number, in order to avoid user panic.

Event Type

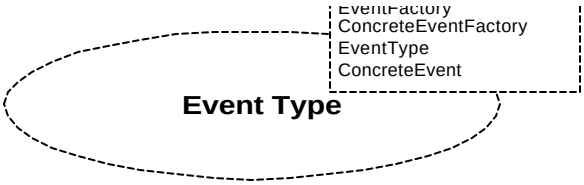


Figure 5: The Parameterized Collaboration representation for the Event Type Pattern

Example

Imagine a fire station alarm management system. Such a system records emergency calls, launches alarms, assigns firemen and vehicles to these alarms, and helps to follow the damage evolution. Reliability and ability to process alarms and related events are important characteristics of our alarm management system.
On the one hand, we can obviously consider alarm events as a continuous flow of timestamped information, but they must not be considered as equally important: alarm launching is more important than vehicle assignment, which is more important than alarm summary reports.
On the other hand, in order to make the system more reliable, we must consider the technical devices behavior from a timestamped event viewpoint. For example, if established communication between the firemen headquarters and a vehicle is unexpectively broken, we must generate a device error feedback message.
In short, we need to manage a large amount of heterogeneous information,which has several different levels of importance and different purposes. Different kinds of workers look at these events, and are interested in different kinds of problems, so they need to be able to focus on certain events, or to filter less important events. So we need a kind of "smart events", rather than the usual raw text-based events.

Problem

We want to give more intelligence in various messages, such as information or errors, in order to enhance this message management. For example, message filtering, representation, alerts or statistics can take advantage of this intelligence.

Context

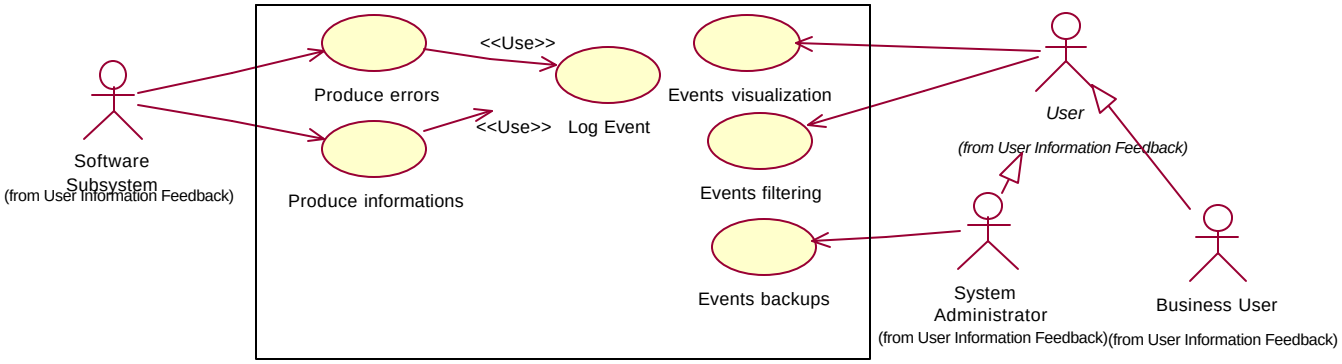


Figure 6: Context for Event Type usage

A software information feedback message may be considered as a piece of information or collection of events. These pieces of information must be considered from several viewpoints:

- **The software subsystems** : produce the feedback while running. The feedback gives account of normal processing such as job completion or invocation, as well as all kinds of errors. All subsystems can produce such information, at any time, everywhere. But once produced, this information will be logged, shown and classified, so it must be recognized.
- **Users** are consumers of information feedback. They often want to see messages, but depending on their role, they also want to focus on particular information categories and filter unrelated ones. For example, technical staff will probably be interested in backups of events, in order to replay a previous situations.

Forces

- You want to identify unambiguously each kind of information to log (information, warning or error), across your entire system …

…But you want this identification be something easy to create (even if the team is split in several places), otherwise team members will bypass event creation.

- You must display a short sentence for each event notified…
  …But you don't want your software be dependant of localization or even of sentence minor changes, to display a more pertinent sentence, for instance.
- You want to take in account the evolution of software in future release, which may cause new events or make old events obsolete…
  … But you may want these changes to be compliant with old releases, in order to reload event logs backup, for instance.
- You want to be able to filter events according criteria such as the logical group it belongs to or severity level…
  … But you don't want to worry about each kind of event when changing filter criteria. Instead, we prefer to deal only with logical groups and severity levels.
- You want to be able to serialize events on a persistent media…
  … But you don't want this serialization to be dependent of a fixed number of events, but instead be compliant with future releases where new kind of events will be created.

Solution

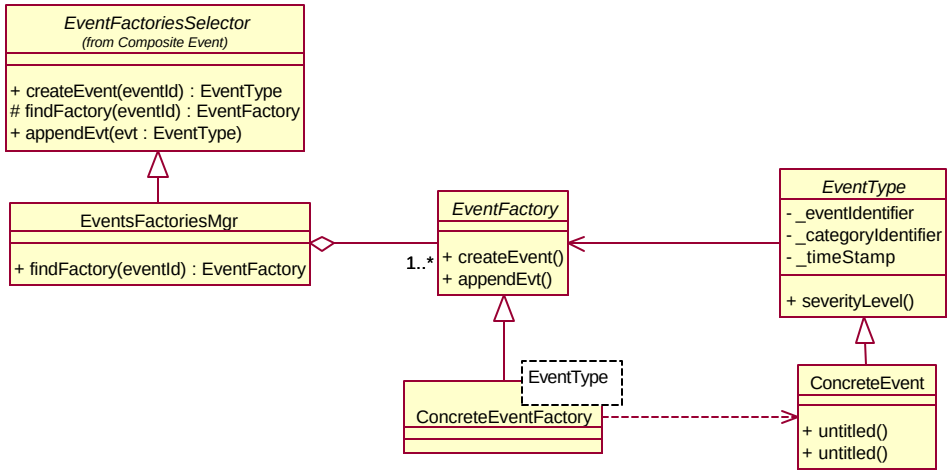


Figure 7: The Event Type solution class diagram

The main idea is to represent each kind of event as a class with a unique identifier named Event Type. This class representation allows events identification and the unique identifier facilitates serialization. For each new kind of event, create a new Event Type. An event may be information about operation completion or failure, warning or error you want to point out to user. Each Event Type must contain:

Event identifier

This identifier may be a simple number for small applications, or a complex one, split in several parts, for large system. For instance, a system built upon components and libraries can use an event identifier composed of a major number which identifies the component or the library, and a minor number which identifies an event type within the component or library.

An event identifier must never be reused. This prevents identifier clashes and the need to know the corresponding release when you reload old log-events backup.

Category identifier

Event Types may be shared in several categories depending on your system. Make an event type associated with a category is helpful in event filtering and representation.

Severity level

Some error levels must be defined to handle the importance of the event. Be careful to not define too many levels. Severity levels are useful for filtering events, especially for selecting a convenient media for the event.

We want the EventsFactoriesMgr be independent of the EventType inheritance tree, because this tree may change and evolve. By associating the same template factory (ConcreteEventFactory) with each ConcreteEvent, we help the EventFactoriesMgr to deal uniformly with different concrete classes. These factories are responsible for event instantiation as well as loading and storage of localized event descriptions. Event type objects doesn't store these descriptions, so they can be as lightweight as possible, and also take advantage of message localization and of further evolutions of these messages. These factories follow the Design Pattern Factory Method. In so doing, the EventFactory can be managed by the EventFactoryMgr through a map (where the key is the event identifier), while the ConcreteEventFactory is responsible for ConcreteEvent instantiation.

Participants

Class: EventType	
Superclass:	
Subclasses: ConcreteEvent	
Responsibilities	**Collaborations**
Events identification	
provides a standard message	EventFactory

Class: ConcreteEventFactory	
Superclass: EventFactory	
Subclasses: none	
Responsibilities	**Collaborations**
Installation of a concrete event	ConcreteEvent

Class: ConcreteEvent	
Superclass: EventType	
Subclasses: none	
Responsibilities	**Collaborations**
Identify a specific event	
store event specific details	

Class: EventFactoriesMgr	
Superclass: none	
Subclasses: none	
Responsibilities	**Collaborations**
Allow instantiation of any kind of event	
Select a factory for event instantiation	EventFactory

Class: EventFactory	
Superclass: none	
Subclasses: ConcreteEventFactory	
Responsibilities	**Collaborations**
Abstraction for creation of any kind of event.	EventFactoriesMgr
Load and storage of event localized messages	

Collaboration

1. A client asks the EventsFactoriesMgr to create an event, using the `createEvent` method. It specifies the Event Type required by its corresponding unique identifier (shown as eventId, here).
2. The EventsFactoriesMgr uses the evntId as a key to find the right EventFactory in its owned EventFactory collection, by using a private `findFactory` method.
3. The EventsFactoriesMgr calls the `createEvent` method on the selected EventFactory. At this step, the eventId is no longer needed.
4. The `createEvent` method is called on the EventFactory abstract class, but realized on the ConcreteEventFactory concrete class. This concrete realization instantiates a corresponding ConcreteEvent object. This instantiation may give the ConcreteEvent standard contextual information like a timestamp and a reference to its corresponding factory.
5. Once instantiated, and in order to stay as small as possible, ConcreteEvent does not contain any standard message, but only classification data and contextual information. Then, when display is required, The EventType ask the EventFactory for this message by using the `getMessage` method.

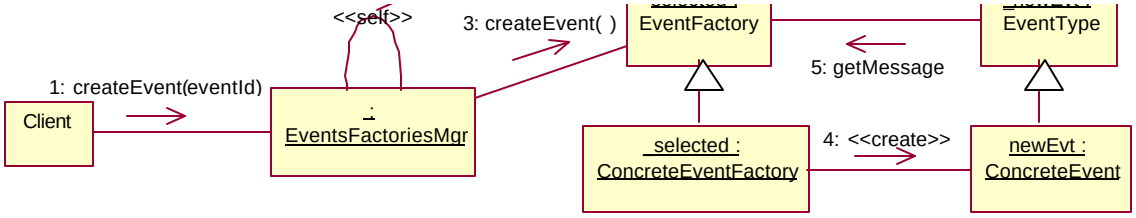


Figure 8: The Event Type collaboration diagram for event instantiation

Implementation

- Event Identifier may be implemented following a broad range of ways. You can use strings as well as numeric values. In both cases, complex identifiers are possible. When using strings, a pathname-like technique seems obvious, while a shifted-values technique may be used for numeric identifiers. aA shifted-values technique is used in Microsoft COM, for HRESULT type ([Box98] p. 42).

- The ConcreteEventFactory may be implemented as a generic class. Such a generic class only needs to redefine the `createEvent` method using the generic parameter. A drawback of this option is that there is no room for variation in the ConcreteEvent constructor parameter list. It may be a problem if different ConcreteEvent classes need a different number of detailed information parameters. The subsequent Part-Made Context Pattern avoids this drawback.
- Operating Systems often allow localized messages. For example, on Windows NT operating system, such messages are available as resources stored separate from program data, and identified with a numeric value. The API function `FormatMessage` loads messages depending on localization settings. The EventFactory can do both the message identifier storage and the message loading.
- The EventFactoriesMgr can be a Singleton ([Gamma+95] p. 127). In doing so, the unique instance of this class can be accessed from everywhere, in order to make easy to instantiate events. On the other hand, it's probably a good idea to eliminate a direct dependency on the EventsFactoriesMgr from every subsystems. This goal can be achieved by using a design pattern Façade ([Gamma+95] p. 185)

Rationale

- This pattern ensures unique identification of each kind of event: The Unique Identifier ensures a non-ambiguous identification through space (the system scope) and time (from release to release). For large systems, a split identifier allows a non-centralized identifier management.
- It allows localized and evolvable messages associated with events: EventFactory assumes a localized load and stores the standard message for each kind of event. These messages are not stored in events themselves, and they are managed separately as resources.
- It takes into account release to release compatibility: it doesn't reuse old event identifiers for new events, even if they are no longer used in your system ([Fowler97] p. 89).
- It allows group filtering by category identifier and severity identifier.
- It provides for serialization and reloading of events: The explicit unique identifier must be written ahead the record. This makes future rebuilds of corresponding EventType objects easier.

Consequences

☺

- Each event generated is more than a message; it's information which belongs to an EventType, allowing manipulations and classifications unthinkable with simple messages.
- Event identifier allows an out of process life for the events, like storage on persistent media or transmission to an administration tool.

☹

- Because there is a class for each kind of event, the number of classes increases. Moreover, there is a new subsystem especially for events managing
- Eliminating dependencies between subsystems and identifiers require some work.

Related Patterns

- **Factory Method** [Gamma+95] p. 107: Event creation follows this pattern, with EventFactory as Creator, ConcreteEventFactory as ConcreteCreator, EventType as Product and ConcreteEvent as ConcreteProduct.

- **Singleton** [Gamma+95] p. 127: Is a possible alternative for EventsFactoriesMgr, to make event instantiation possible from anywhere. See also [Grand98], p.127.
- **Façade** [Gamma+95] p. 185: Another possible alternative to make the event instantiation service available without direct exposition of EventsFactoriesMgr. See also [Grand98], p. 205.
- **Typed Diagnostics** [Harrison98]: EventType and ConcreteEvent are variations of this pattern. Typed Diagnostics mainly focuses on message handling. Unlike this, Event Types focuses on event identification and classification. The message is considered volatile information.
- **Capsule Pattern** [Martin97]: Allows error category identification using a specific interface for each category. This pattern eliminates the problem of a subsystem strongly coupled to a unique error definition [Lakos96].
- **Part-Made Context**: May help ConcreteEvent to append additional contextual information about the event.
- **Media Dispatcher**: In order to be shown, logged, etc… Event Type must be handled. Media Dispatcher achieves this goal.
- **IO Triage** [Hanmer+98]: Handles messages of differing importance; messages with high importance are expedited.
- **Timestamp** [Hanmer+98]: Interprets the order of event occurrence as important information. This information can be conveyed by Event Type if events are able to be output in a different order.
- **Error Handler** [Renzel99]: Can be an alternative for error management. Like Event Type, this pattern separates client code and the error management subsystems. But rather than creating user oriented event objects, it deals with error objects themselves. This pattern can't deal with non-error information.
- **Naming** [Mowbray+97], p. 213: The CORBA naming space looks like our unique identifiers.
- **Flyweight** [Gamma+95], p. 195: The way by which localized messages are stored by the EventFactory, and shared by several event types address the same problem.

Known uses

- Windows NT event logging [Murray98] uses this kind of definition for events stored within the logging service, and displayed with event viewer. The EVENTLOGRECORD stores Event Identifier, Category identifier, and Severity Level.
- Ilog Solver ([Ilog98]) uses such unique identifier called `IlcErrorType` (an enumerate), in order to identify kind of errors that may happen.
- SOS: Uses this pattern to log on alerts information: first call, first firemen departure, first aid arrival, end of the alert.
- Acropole: This gas management system uses this pattern to show operations done on the gas network as well as operations done on the information system (like database updates).
- DCE had popularized the concept of unique identifier, namely UUID [Rosenberry92]. UUID were reused later in Microsoft COM, to provide unique identifiers for class or interfaces [Box98].

Part-Made context

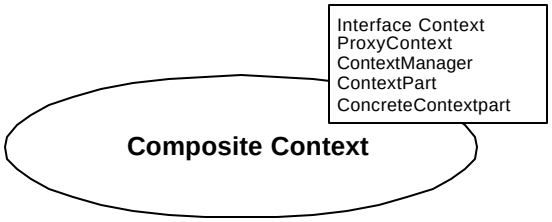


Figure 9: The Parameterized Collaboration representation for the Part-Made Context Pattern

Example

By now, our firemen alarm management system looks fine: it's able to produce events, even complex ones. Nevertheless, we need to make these events more accurate by giving details related to the current context.

On the one hand, the global system context is something complex, with current alarm state, vehicles, peoples and all kind of devices. A particular event doesn't need all this information. For example, the "vehicle arrived on alarm spot" needs only this vehicle's details (number of firemen, equipment, etc.).
On the other hand, the system context can't be considered as a single chunk of information. The overall current context is made of several parts: a current alarm, a current communication link, a current vehicle "processed", etc.So we need to divide the overall current context in several parts.

Problem

Pointing out a problem or an operation completion is, in most of cases, not sufficient by itself. It's often useful to have more than a standard message. Because some jobs require other operations to be done first, or some data values have no sense, or devices can be out of order, smart messages including contextual information must be given.

Context

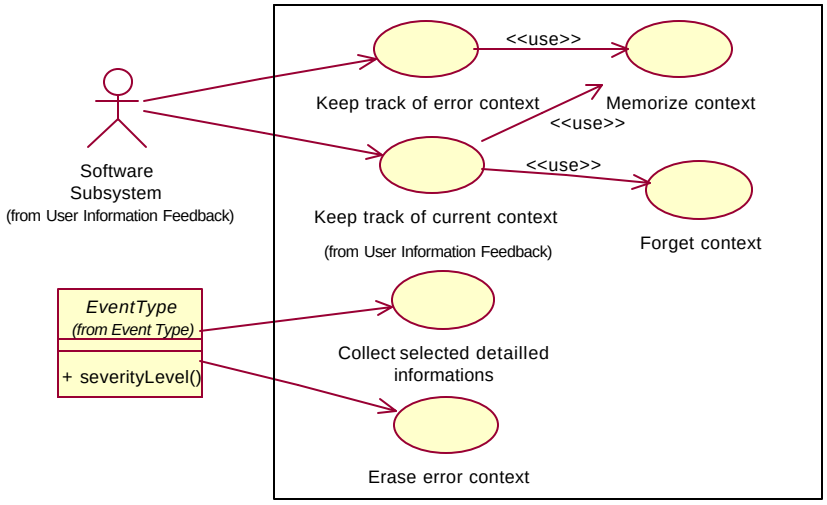


Figure10: Context for Part-Made Context usage.

Event Types are standard events produced by the software. But they are produced in specific situations, which is an important information to the produced events.
The software subsystems: Are responsible for keeping track of current context changes. When entered, these new contexts must be remembered, and forgotten when exited. When an error is produced, the context surrounding the error is memorized as well.
Event Types: Are clients of these contexts. They are interested in pieces of information related to their nature. For example, errors related to database access would be interested in the current database reference, or the table currently used.

Forces

- We want events be able to retrieve and store some contextual information, in order to give the user relevant detailed information...
  … But we don't want to worry about this collecting work when we produce the event. Moreover the pertinent information is not always structurally accessible from the event production point.
- We want to give user a detailed information…
  … But we want each EventType to be able to select its own pieces of atomic details.
- We want to keep track of general context information…
  … But we want this work be as slim as possible. In particularly we don't want to worry about deleting context information when it is no longer needed.
- We want to keep track of exceptional contexts such as error context…
  … But we also want this information to be deleted once the corresponding error event is built.

Solution

We need to consider the overall system context, and divide it in several parts. A context-part is a chunk of information that can be updated as a whole and that can be used by EventTypes to give focused details. The whole context, with all its parts, is built around a ContextManager. This ContextManager brings together atomic pieces of information as ContextPart. Each context part may store its own information in any way, but must be able to be translated into string. When an EventType requires detailed information, it calls ContextManager as many times as necessary through `getCtxInformation` method, using a different key each time. This method fetches the right ContextPart object, then requests translated information using the `convertToText` method. When built, EventType also calls ContextManager's `flush` method in order to erase potential error information.

On the other side, subsystems set up contextual information using proxies. There are three reasons to use proxy on the application side:

- Dependencies: As proxy does not reside in the ContextManager's package, it eliminates crucial dependencies. Moreover, a multi-threaded application may lead to multiple ContextManagers, as discussed in the implementation section. Using a proxy hides such complexity.
- The application side only needs an interface to register and unregister information. Only event management must have access to other operations available on ContextManager.
- Specialized proxies can be designed separately to store general contextual information or exceptional error information.

  StackableCtxProxy is designed for general contextual information, with information registration performed in constructor and unregistration performed by the destructor. When used as automatic instance, only the declaration is needed, everything else is done behind the curtain including unregistration which is done while stack is unwinding.

ErrorCtxProxy is designed for exceptional error information. This proxy only registers information but never unregisters it. Error info erasing is performed when the next EventType is built.

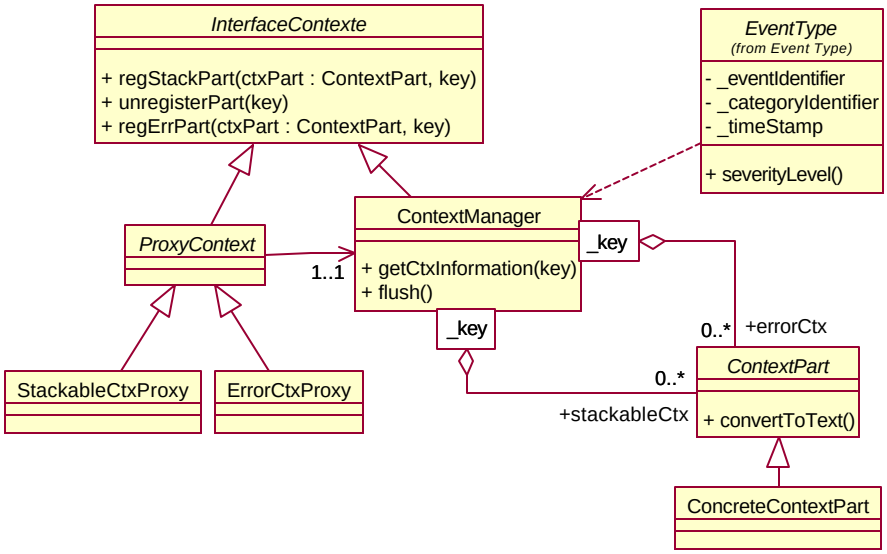


Figure 11: The Part-Made Context solution class diagram.

Participants

This section does not describe EventType, which comes from the EvenType pattern.

Class: InterfaceContexte	
Superclass: none	
Subclasses: ContextManager, ProxyContext	
Responsibilities	**Collaboration**
Provides an interface for registration and unregistration services	

Class: ProxyContext	
Superclass: InterfaceContext	
Subclasses: StackableCtxProxy, ErrorCtxProxy	
Responsibilities	**Collaborations**
Register and unregister contextual information on subsystems side.	
Send contextual information to a central manager	ContextManager

Class: ContextManager	
Superclass: InterfaceContext	
Subclasses: none	
Responsibilities	**Collaborations**
Store atomic information	ContextPart
Provide atomic information	EventType

Class: ContextPart	
Superclass: none	
Subclasses: ConcreteContextPart	
Responsibilities	**Collaborations**
Provide interface to be managed by ContextManager	ContextManager
Provide an interface for text-based exportation	

Class: StackableCtxProxy	
Superclass: ProxyContext	
Subclasses: none	
Responsibilities	**Collaborations**
Hide context registration as error	
Build specific contextual information	ConcreteContextPart

Class: ConcreteContextPart	
Superclass: ContextPart	
Subclasses: none	
Responsibilities	**Collaborations**
Store or aggregate a specific context information in a native form	
Implement the convertToText method to convert native data into text.	

Class: ErrorCtxProxy	
Superclass: ProxyContext	
Subclasses: none	
Responsibilities	**Collaborations**
Hide contextual information unregistration	
Build specific contextual information	ConcreteContextPart

Collaborations

The following collaboration describes a ContextPart update.

1. A ProxyContext is declared as a local variable of a method. When entering the method, the ProxyContext constructor is invoked. Contextual data are passed as constructor parameters.
2. The Constructor instantiates a new ConcreteContextPart, using its parameters.
3. Then the ProxyContext calls its own `regStackPart` method, passing the new ConcreteContextPart and a key as parameters.

The ProxyContext `regStackPart` method calls the ContextManager `regStackPart` method. This last method stores the ConcreteContextPart.

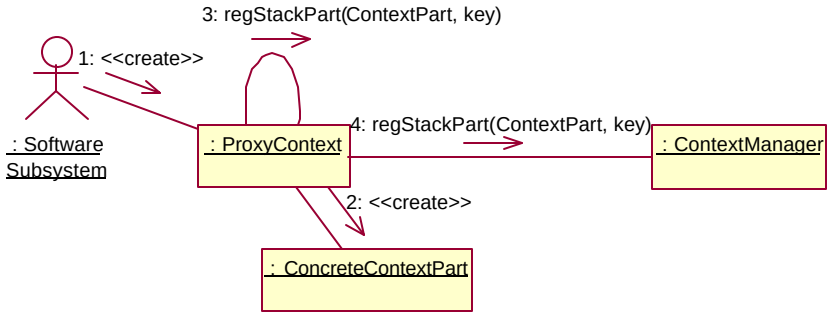


Figure 12: A context-Part creation using a proxy.

When execution path gets out of the context-part scope, the related information is removed as follows:

1. Upon exiting the method, the ProxyContext local variable is destroyed by calling the ProxyContext destructor.
2. The ProxyContext destructor calls its `unregStackPart` method in order to invalidate the corresponding ContextPart, passing a key as a parameter.
3. The ProxyContext `unregStackPart` method calls the equivalent method on the ContextManager.
4. The ProxyContext `unregStackPart` fetches the ContextPart corresponding to the key. When found, the ContextPart is deleted.

Implementation

- StackableProxy is a kind of "automatic proxy", which registers information when built, and unregister this information when destroyed. The StackableProxy constructor must :

1. Fetch a valid reference to the ContextManager it needs.
2. Build a ConcreteContextPart.
3. Call ContextManager `regStackPart` method.

   Step (2) brings to light the need for a specific StackableCtxProxy corresponding to each ContextPart. In C++, this goal can be achieved with StackableCtxProxy as a generic class. In order to be feasible, all ConcretePart constructors must require the same number of parameters. In C++, we can force StackableCtxProxy to be an automatic instance by declaring `operator new` and `operator delete` as private ([Meyers96] p. 157).

- Like StackableProxy, ErrorCtxProxy can also be implemented as a generic class in order to achieve the same goal.
- For most simple applications, there will probably be only one ContextManager instance. In this case, ContextManager may be declared as a **Singleton** ([Gamma+95] p. 127), or as an EventsFactoriesMgr's aggregate (look ahead to EventType pattern). But for multi-threaded applications, there may be several ContextManager. In this case:
  - Some information belongs to the main thread, and is shared by all secondary threads as read-only information.
  - Some other information belongs specifically to each secondary thread.

  This kind of applications will provide a per-thread ContextManager, which owns the thread specific contextual information. This ContextManager will forward request involving shared information to a "main" ContextManager.
- Sometimes, subsystems are built upon frameworks, and these frameworks provide a kind of ConcreteContextPart class. A good idea is to reuse it directly as a true ConcreteContextPart. It can be done by using the **External Polymorphism** design pattern ([Cleeland+96])

Rationale

- Ease of detailed information retrieval: Events only needs to tell the ContextManager which atomic information is required, no matter of where did the information comes from. Moreover, this information collecting is encapsulated inside the EventType itself, so there is no more special work to do when producing events.
- Smart selection of detailed information: Detailed information is available separately on the ContextManager. Only each concerned EventType knows how they fit together to produce aggregate information.
- Ease of general context memorization: The StackableCtxProxy allows easy information memorization, using a simple declaration.
- Error context erasing: The error contexts are separately managed and registered. On one hand, the ErrorCtxProxy does not unregister the corresponding ContextPart. One the other hand, these ContextPart are explicitly managed as error contexts, and erased each time an event is produced.

Consequences

☺

- With a little cooperation of subsystems, we continuously keep track of current runtime context.

- Clear separation between context management and events generation. As EventTypes and context management are both designed in the same subsystem or in closed subsystems, event types can easily obtain required detailed information.
- Context recording and event production are asynchronous. When applied in multi-threaded environment, a per-thread ContextManager allows collection of thread specific context information. Moreover, as this ContextManager is accessible through a proxy, event notification may evolve from a single threaded to a multi-threaded application with little changes.

☹

- Every subsystem which is involved in context changes must pay a little by manually signaling these changes.
- Runtime overhead: Each context change notification involves an extra execution time. Designers must take care to only signal significant context changes. For example, notification of all method calls is surely an exaggeration. Context change notification is a price to pay, even if we don't take advantage of this information within a produced event.
- Design complexity increases because of additional classes for context management. On the other hand, event code complexity decreases, because information collecting becomes easy.

Related Patterns

- **Shopper** [Doble96]: Is an alternative way to collect information from various objects. If applied here, EventType becomes Consumer, ContextManager become Provider, ContextPart become Item and Key become ItemRequested.
- **Proxy** [Gamma+95] p. 207: The register / unregister side of the Part-Made Context pattern is built upon this pattern. See also [Grand98], p. 79.
- **Diagnostic Contexts** [Harrison96]: Is an alternative to manage context as a piece of information, which can be exchanged each time a new particular context is entered.
- **External Polymorphism** [Cleeland+96]: This pattern can be used in order to reuse preexistent but unrelated classes, which carry contextual information.
- **Event Type**: This pattern is an obvious client of Part-Made Context. EventType needs Part-Made Context to obtain detailed information.

Known uses

- Part-Made Context was used on Acropole, a gas management system.

Focused Messages

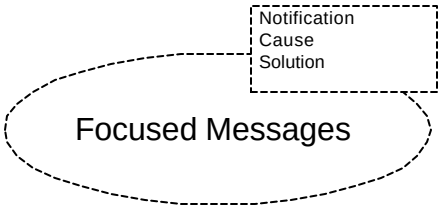


Figure 13: The Parameterized Collaboration representation for the Focused Messages Pattern.

Example

A fire station alarm management system is mainly used by firemen, but officiers, system administrators, and software support may use the system too. That's why, even if events are identified and if we have captured detailed information, we must make our events readable by each kind of user. It seems to be a perfect goal, but the way to reach it is less than obvious.
First, we have different kind of information: corporate information, legal information and technical information, for example. These different kind of information deal with different kind of users: firemen, officers, or technical staff (software support and system administrators). These different users use different vocabularies and focus on different point of view.

Second, we must think about how we can make messages efficient, especially when messages are errors. For example, when a fire station don't answer to an alarm call, this problem must be reported to the headquarters. But this reporting is not enough by itself, it's probably important to relate why the problem occurred, and who should be called about this kind of malfunction. In addition, it might suggest some alternative to correct or bypass the problem.
In short, we need some rules and advice about how to write information and error messages.

Problem

Who needs what kind of information?
What kind of information may be provided beyond the notification, especially for errors?

Context

Events must be shown in an understandable way. Event identifiers don't achieve this goal. Moreover, different kind of users has different needs. For example:
The **Manager** focuses on risk management and employees overall performances.
A **Business user** wants to be helped when error occurs or when jobs can't be completed, even when he caused the problem. He also wants to be informed when required jobs are completed.
System administrator, when involved, needs to be informed of technical problems with their precise technical context. He also wants alarms on problem occurrence, even if they are minor ones, in order to keep track of resources problems like network load.
Hot-line Attendant must have solutions to end users problems. Sometimes, this solution is to suggest an alternative action to the user. Therefore, the hot-line attendant must have clues about missing elements (data or actions) as well as access to some internal data.

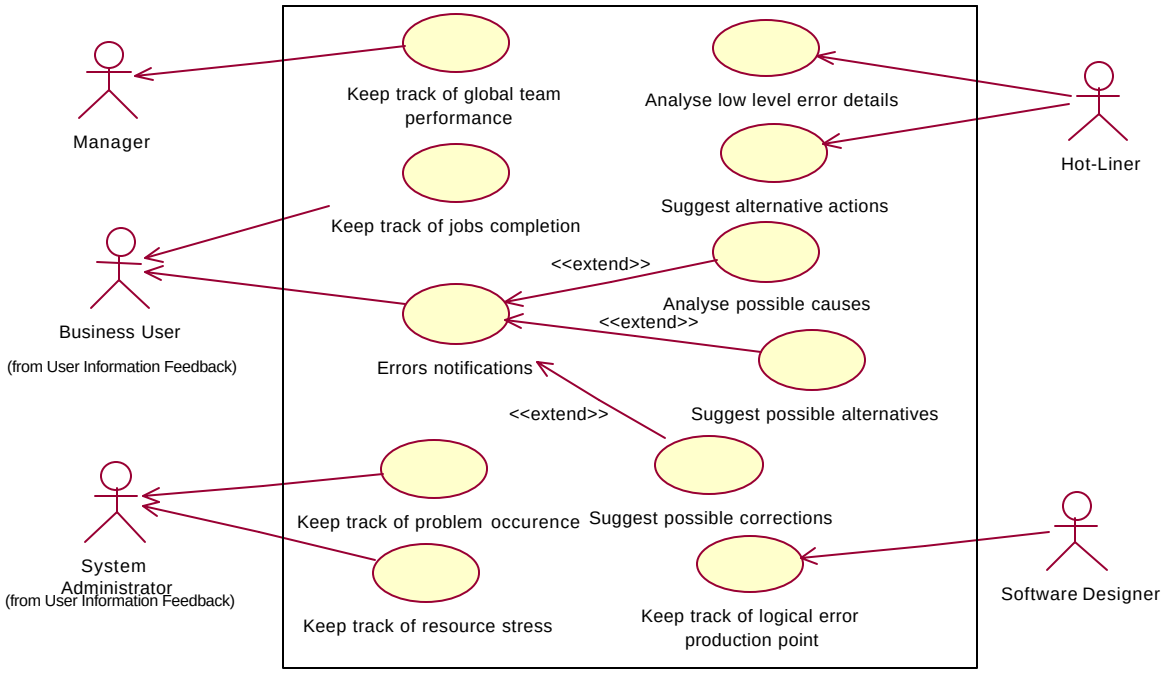


Figure 14: Context for Focused Message usage

The **Software designer** is involved when potential bugs are outlined. So he needs to know where the problem originated, if possible.

Forces

- Different users often needs different kind of information or different level of details…
  … But we don't want to produce events several times.
- Different users are not interested in the same events…

… But, as all kind of users may have a view on the system at the same time, all events must be generated.

- End users want to be notified, especially for errors, in their business words …
  … But they also want to know why problem happen and how it can be corrected.

Solution

Rather than simply writing error message with technical words, take a user viewpoint and write it in a way which helps users to understand what happened and to correct the problem. This involves not only to stating the problem with the user's own vocabulary, but also explaining why the problem occurred and how to solve it.

- Because software engineers are not the best qualified to write error messages, give this work, if possible, to documentation writers. Elsewhere, let the messages be reviewed by users. This point is already addressed, in most part, by the **Mercenary Analyst** pattern
- Error messages must be written using user's words. Avoid abbreviated format using technical word.
- Make the error messages an explicit software quality evaluation criteria.
- Don't make messages too lengthy. Remember that error messages break user's work rhythm. In the worst case, the user won't read the notifications if it takes more than a couple of seconds.

End users often require help from hot-liner attendants or system administrators. In order to make their intervention as efficient as possible, give them some information.

- Some internal information may help to find a problem, for example: a record identifier in a database, the name of a temporary file, or the line number involved when a text file is parsed. Because this kind of information is not of interest to end users, they must be reserved to technical staff.
- Hot-liner attendants need "hot spots" in the software. Sometime, these hot spots get the form of special or misunderstandable messages or even of low-level traces with the method called name. A better approach is to have special tags, only visible by support engineers, with a map that gives the corresponding meaning of these tags.

Whoever is the target of error messages needs three levels of information:

Notification: "what happened?"
Cause: "Why did the problem happen?"
Solution: "How can I get by?"

The following table describes what each of the former three points should contain, in order to be valuable for each kind of user.

Users	Cause	Problem	Solution
Manager	• Gives a localization of the problem by means of software deployment and / or user involved.	• Point out impacts of the problem. • Draw a map of security break if needed	• Point out who must be contacted (users, system administrator, ...)
Business user	• Point out business data involved. • If technical devises are involved, ask about them with a user point of view • Checks the user interface manipulation, which may leads to the problem. • If commands are recorded [Sommerlad96], point out incompatible command used.	• Point out actions that cant be completed • Prevent unrecoverable errors by displaying a message, which outlines bad consequences.	• Recall works that must be previously done before. • Give a way to check or correct data. • If devices are involved give concrete ways to check them (wire connections, power supply, paper...). • Gives links to online help.
System administrator	• Recall resources and user involved. • Gives precise timestamp of the event. • Gives a link to related events and error if concerned	• Point out resources contentions. • Point out system inefficiencies (network frames lost, timeouts).	• List tools, which may help to solve the problem, like analyzers or performance meters. • Gives a link to a bug report base or a

			knowledge base.
Hot liner	• Give a dump of related context.	• Point out "hot spot" involved	• Give links to a bug report base. • Give links to a "cookbook" reference that gives ways to bypass the problems.
Software designer	• Give traces which may reconstitute the call stack as close as possible.	• Gives implementation oriented messages including methods and variables names.	• Give links to a debug oriented cookbook.

Rationale

- Event types may store detailed information as their own, because unlike standard messages, detailed information is context dependent.
- Message selection depending on the user role is not addressed here, but in the following **Media Dispatcher** Pattern.
- Follows the general advice to write error messages: use understandable words, review the message, write three-part messages.

Consequences

☺

- Information provided for each kind of user better match their needs.
- End users have more than notifications: they can take advantage of experience because they know why some errors happen. On the other hand, corrections suggested make the users more independent and more productive because they require less help from a specialist.
- Hot-line workload decreases.
- By becoming user-friendly, users become well impressed by the software, even if all requirements are not developed as expected.

☹

- Possible errors must be well analyzed, in order to keep track of possible causes and suggest possible corrections.
- Writing error messages become time consuming, because:
  - We must take time to express messages using understandable words (from a user viewpoint).
  - Instead of one notification message, we have to fill three sections.
  - The work process is impacted because of message quality evaluations.

Related Patterns

- **Command Processor** [Sommerlad96]: As the command processor stores a list of commands, it may help to find a possible incompatible previous command, when problem occurs.
- **EventType**: Event Types may store the different kind of detailed information, messages (notifications) and solutions.
- **Part-Made Context**: Is a way to give some information for event's details.
- **Diagnostic Contexts** [Harrison96]: Is an alternative way to obtain detailed information.
- **Deferred Validation** [Cunningham95]: Assumes values check when an action is requested. Doing so, you can expose potential problems and deduce the cause.
- **Diagnostic query**[Cunningham95]: Allows problem tracing for hot-line attendants as well as software developers.
- **MML** [Hanmer+98]: Advice to use a consistent language in order to be understood by users.
- **Who Asked?** [Hanmer+98]: Outlines the existence of "classes of workers", who don't need to same degree of information.

- **Mercenary Analyst** [Coplien95b]: Adress the problem of choose a technical writer who is also proficient in the user domain. Our pattern uses this one.

Known uses

- This pattern was widely used by Ben Ezzell's ErrorMessage dynamic library ([Ezzell98]).

Separation of message and detailed information is used within Windows NT event logging ([Murray98]).

Media dispatcher

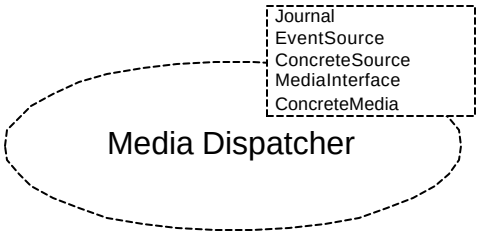


Figure 15: The Parameterized Collaboration representation for Media Dispatcher Pattern

Example

With event identification, context memorization and message creation, the event reporting for our fire station alarm management system is nearly complete. It needs only one thing, last but not least: the reporting itself. For legal duty, some events must be reported on printers, like emergency calls, first vehicle departure or first arrival at the scene. Important errors must be displayed to the user via dialog boxes, such as a major communication malfunction or database crash. Most events are displayed on screen as a message list, but even here, firemen look at alarm management events, and system administrators look at devices and communications malfunctions. In short, we need to able to use several media for event reporting, to select or change these media dynamically, and to create events regardless where they will be displayed.

Problem

Once we have events generated, we want to manifest them in different ways. These manifestations may be unrelated. Some possible manifestations are:

- Error panels.
- A console or messages window.
- Status bar.
- Printed log.
- File (or database) based log.
- Vocal messages.
- Mail.

Each of these media may be used differently by each kind of user. A medium can be used for all or selected events, selection can be based on error-level or event-category, and this selection may be modified dynamically at runtime.

We need a solution that allows publication on any media, no matter how they are different.

Context

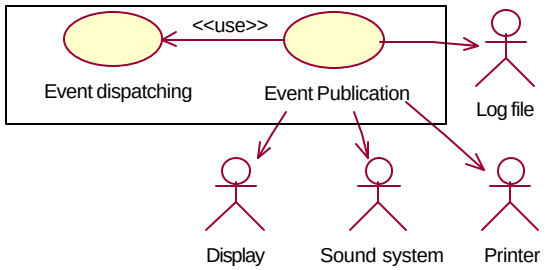


Figure 16: Context for Media Dispatcher usage.

Events can have many different uses. These uses may involve many different media, as stated in the former section. We need an agent, which takes the responsibility of event publication, in order to decouple the representation (on log file, printer, sound system, display, etc…) from the event itself.

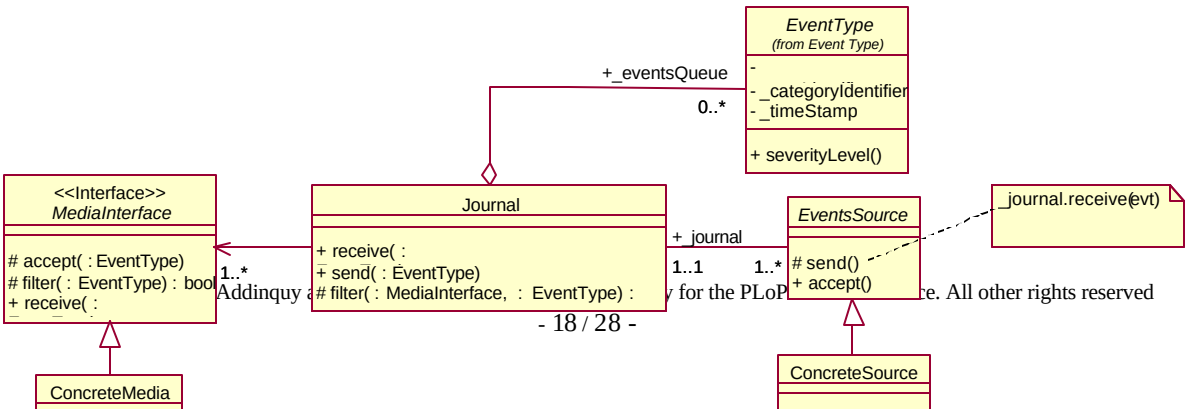
Forces

- We want to make the events appear…
  … But we don't have yet a good idea of the definitive display.
- We want a lot of information to be available about operation completions, warning and errors…
  … But we don't want experienced users be drowned in a large number of informative messages.
- We want users be informed of overall processing information …
  … But we also want to make these users focus on a category of information.
- We want to be informed of software processing events from within the software …
  … But we also want to be informed from a specialized administrative tool, on the same computer or on a network accessible computer. Moreover, we may want to be informed using a general tool such as mail.

Solution

The main idea is to decouple the collection of new events from redistribution to the concerned media. Therefore, we can improve event manifestations through different media, separately from events creation. This pattern makes the Journal the link between one or several event sources and one or several media. Each potential event source subclasses EventSource. Doing so, it can take advantage of the `send` method, which forwards events to the Journal. The ConcreteSource class sends all events generated to the Journal without any filtering. The Journal class accept events using its `receive` method and may store these events in the _eventsQueue association. Then the Journal class looks for all media recorded as MediaInterface if the event publication is required, using the `filter` method. The filtering criteria may be:

- Is the event able to be published on the medium? For example, to be published on sound device, the event must own corresponding sound resource.
- Is the ConcreteMedia owner allowed to watch this event? This ConcreteMedia may be a proxy on a distant media, and this distant medium may be owned by another application run by another user.
- Is the medium selected for this error level or for this event category?



Of course, this is an open list. The way this filtering is done inside the filter method is an open question too. A **Strategy** Pattern ([Gamma+95] p. 315) seems to be a good approach if dynamic filtering strategy modification is required.

For media selected, `receive` methods are called on MediaInterface. This method acts as a template method: it calls the `filter` method, then, if the event is accepted, it calls the `accept` abstract method. Each ConcreteMedia must redefine this `accept` method, by which the events are definitively published.

Figure 17: The Media Dispatcher solution class diagram.

Participants

This section does not describe the EventType class, which is already defined in the first pattern.

Class: Journal	
Superclass: none	
Subclasses: none	
Responsibilities	**Collaboration**
Accept events generated elsewhere.	EventSource
Store events received.	EventType
Dispatch events.	MediaInterface

Class: EventSource	
Superclass: none	
Subclasses: ConcreteSource	
Responsibilities	**Collaboration**
Defines a connection with Journal, in order to feed it with events.	Journal
Provide a send events service for subclasses.	

Class: MediaInterface	
Superclass: none	
Subclasses: ConcreteMedia	
Responsibilities	**Collaboration**
Receive events to be published.	Journal
Defines an interface for event publication.	

Class: ConcreteMedia	
Superclass: MediaInterface	
Subclasses: none	
Responsibilities	**Collaboration**
Realize the event publication on specified media.	MediaInterface

Class: ConcreteSource	
Superclass: EventSource	
Subclasses: none	
Responsibilities	**Collaboration**
Provides events that must be published.	EventType

Collaboration

The following collaboration shows how created events are dispatched to the medium.

1. A `ConcreteSource` can be any kind of event producer. For instance, it can be an EventFactory. The Client object is shown here just for the example convenience.
2. The `send` method is called on the EventSource, passing the newly created event as parameter.
3. The EventsSource `send` method calls the Journal `receive` method, in order to make the event registered.
4. When received on Journal, the Journal's `send` method is called. This method iterates on each registered media.
5. For each registered medium, initial event filtering can be done by calling the Journal's `filter` method.
6. If the filter method is "passing", then the `receive` method can be called on the specified media, throught its MediaInterface base class.

7. When received on the media, the EventType can be filtered once again using the ConcreteMedia `filter` method.
8. If this last filtering is "passing", then the ConcreteMedia `accept` method is called. This method must be implemented on each medium in order to display, print or play the event message as needed.

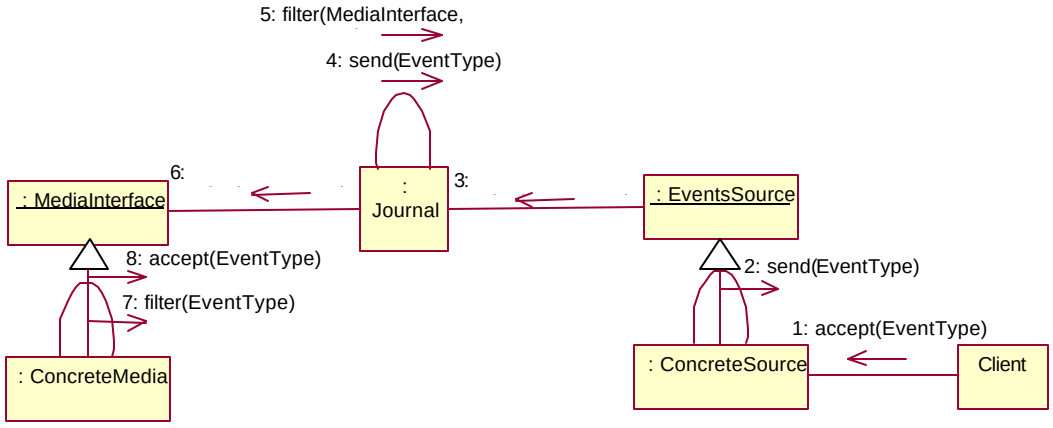


Figure 18: Event dispatching collaboration.

Implementation

Some issues must be considered when implementing the media dispatcher pattern:

- EventSource: The EventsFactoriesMgr from EventType pattern may also play the role of EventSource.
- Asynchronous reception / publication: Except for severe errors, event publication can be done asynchronously. Thus, the application does not need to wait for publication on every medium. This issue may be especially important if the concerned application is critical, like a server. Then, the application thread can be extended to the Journal event feeding, but not to the publication. Then, the Journal's `receive` method only stores received events in the event queue. The publication thread picks off events from event queue (which is a FIFO queue). In this context, the event queue must be protected against concurrent access, using a mutex object, for example.
- Do you want the ConcreteMedia be able to connect or disconnect dynamically? From within an application, the only reason to want a ConcreteMedia be like that is to support device or library "hot-plug". Elsewhere, the dynamic connection / disconnection is useful if distributed events must be supported. If only a static connection is needed, MediaInterface can register itself from within its constructor.
- Do you want the event source to be connectable dynamically too? The same reasons can leads to a dynamic connection for EventSource. From a distributed object point of view, dynamic connection may be useful for administrative tools, which collect events from outside applications. EventSource needs not to be registered inside the Journal, but the EventSource must have a reference on the Journal. For a static connection, the Journal cans create the EventSource at the very beginning, giving to this EventSource a reference to itself.
- For a dynamic connection, EventSources can be instantiated at any time, so they need access to Journal reference. To achieve this goal, the Journal can be a Singleton, for example.

Rationale

- Event generation for undefined media: Now, events can be generated without regard to future media definition. Once EventTypes define what kind of information they deal with (text, sound, icons…), publication problem can be handled later. The only hot spot for ConcreteMedia is to pay attention about the exact information format.
- Event filtering configuration: Event filtering can be done on two levels: on the Journal, or on the MediaInterface. Filtering of error level messages is better done on the Journal.
- Focus on information category: We can have a by media subtle filtering, using the MediaInterface filter method. For a specific media (a console window, for instance) we may have a low error level

filtering using the Journal filter method. But using the MediaInterface filter method, we can select only the event categories of interest.

- Dispatching and publication across process boundaries and / or across the network: A CORBA or DCOM based object may be a ConcreteMedia as well as a ConcreteSource. Doing so, complexity of event transmission, marshalling and reception is localized to the distributed objects, but it's the common goal of these distributed objects. The dispatching logic stays localized to the pattern with no difference from other media. It's a clear separation of concerns.

Consequences

☺

- We don't have to worry too soon about the way events are published.
- Event display or recording is clearly separated from their generation. This separation of concerns keeps the door open for changes.
- Introduction of the concept of media is much less restrictive than only "display" or "record" events. It handles manipulations of data screen media and file media unifomly, and leads to thinking about new media.
- It's now easy to send events across process boundaries or even across the network without additional complexity. For example, a CORBA distributed object may be a kind of medium.
- This pattern can be applied outside the scope of the User Information Feedback Pattern Language, in order to transmit pieces of information from one or several sources to one or several destinations.

☹

- Each new medium may need a corresponding mesaage format for each event.
- One increases the design complexity for nothing if only one medium is needed. But, most often, we have at least two of the three more common media, namely, error panels, console windows and log files.

Related patterns

- **EventType**: In order to be dispatched using smart criterias, events must include additional information. This pattern achieves this goal.
- **Client-Dispatcher-Server** [Buschmann+96] p. 323: This pattern is better used in a distributed environment, but the three-part structure is close to this pattern. Nevertheless, the intent is to connect a Client with a Server using a dispatcher. The Media Dispatcher is event transmission oriented; however it always separates the event source and output medium.
- **Publisher-Subscriber** [Buschmann+96] p. 339: This other alternative allows event dispatching between a publisher and a subscriber. There is no way to consider several publishers as different sources of same thing.
- **Strategy** [Gamma+95] p.315: Can be used within this pattern if dynamic filtering strategy changes are required. See also [Grand98], p. 371.
- **Template Method** [Gamma+95] p. 325: The MediaInterface `receive` method is implemented like that, in order to transmit to the `accept` method only the events accepted by the `filter` method.
- **Singleton** [Gamma+95] p. 127: For dynamic EventSource connection, provide an access point to the Journal reference. See also [Grand98], p. 127.
- **Mind Your Own Business** [Hanmer+98]: Deals with message filtering and helping the user focus on selected messages categories.
- **IO Triage** [Hanmer+98]: Focus on message filtering following priorities.
- **Who Asked?** [Hanmer+98]: Split the event flow across several channels, but not all. This pattern is close to this one, but no specific design is proposed.
- **Beltline Terminal** [Hanmer+98]: Deals with large workplaces. It proposes to redirect or broadcast messages to several terminals through this workspace.
- **Alarm Grid** [Hanmer+98]: Proposes a specific medium, an alarm grid, for emergency events. Such a medium can be used as a ConcreteMedia by this pattern.

- **Raw I/O** [Hanmer+98]: Proposes another kind of medium, a low level text medium, in order to give feedback even if all other media are out of order. Like the previous one, it can be used as an ordinary medium.
- **Private interface** [Newkirk97] : The Media Dispatcher pattern can take advantage of this pattern for the communications between EventSource and Journal, or between Journal and MediaInterface.
- **Ephemeral feedback** [Grand99], p. 137: Deals with how to keep the user informed about processing without breaking its workflow.
- **Error Dialog** [Renzel99], p. 69: This pattern proposes a direct error display using a dialog box. Even if it deals with the same problem, forces and solution are very different from this Media Dispatcher pattern. This pattern only handle errors and don't abstract the way by which display is realized. No unification is done with logging. In fact, it looks like a lightweight display variant.
- **Centralized Error Logging** [Renzel99], p. 62: Proposes a way by which several applications running on different computers can log errors on the same host. This pattern deals with one kind of media, files. On the one hand, the centralized logging problem solved here is powerful but out of the scope of our Media Dispatcher. On the other hand, this pattern can't be applied to other media.
- **Agent** [Mowbray+97], p. 202: This pattern achieves a part of our pattern goals: It allows uniform access to diparate services.

Known uses

- This pattern was applied on the Acropole project.

Pattern Language consequences

☺

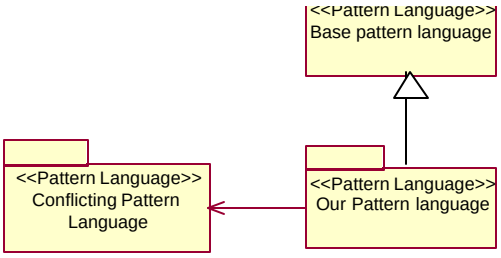
- You can build a framework upon this pattern language. When reused over applications, only specific parts must be redesigned:
  - Concrete Events.
  - Concrete Context Parts and their related proxies.
  - Messages.
  - Concrete Media, when needed.
- It helps us to think about flat messages. First, we should think about event categorization. During software building, we look for data that are context sensitive. We also improve the way by which events are expressed.
- Information and error management are clearly separated in uncoupled parts. We produce errors without any event management in mind. Then we instantiate events, without having to worry about how they will be represented. Then we show these events without any idea about how and where they are produced.

☹

- Event creation requires more work. Rather than a simple display a la printf, we have to create a new class, sometime to create new context parts and to memorize these contexts and to think hardly about the messages themselves.
- We make all subsystems depending on the event management subsystem. Instead of having merely independent messages display, all subsystems involved in event instantiation must know the event management framework.

Acknowledgements

I would like to thanks Neil B. Harrison for his help in shepherding. Neil didn't read a just part of the paper, but the whole pattern language, and appeared to be very involved in the pattern language goals. Thanks to Frédéric Paulin (Ilog), who helped me to enhance the Part-Made Context pattern and gave me links to Ilog products error management, and to Laurent Sarrazin (Société Générale) who carrefully read this paper and produced interesting remarks.



Appendices

Notations

Problem

We want to improve our pattern expression by using graphical representation as often as possible. We would prefer standard representations to exotic diagrams because, as a common language, they will be better understood. The UML seems to achieve this goal. Anyway, this language is very new and some concepts are not well known. Moreover, some existing concepts must be specialized and some needed concepts are not covered by the UML. We need to make the UML used notation understandable, and the alternative notations explained.

Context

We are writing a Pattern Language, a set of cooperative patterns that can be applied to a particular domain.

Forces

- We want to illustrate each section of each pattern as well as the pattern language relationships…
  … But we want to avoid exotic representations in order to makes the patterns understandable as widely as possible.
- We want to use the widely adopted UML notation with its pattern-related concepts…
  … But we must take in account that this notation is rather new and some of its particular concepts are not well known yet.
- We want to take advantage of UML diagrams…
  … But as we use these diagrams not for software development documentation, but for a pattern description, we must keep in mind that these diagram may be misunderstood.
- The UML not exactly meets the pattern description needs…
  … But we don't want to modify the UML notation so far that its initial meaning will be perverted.

Solution

We use UML notation ([Booch98] and [Rumbaugh99]) as widely as possible through this pattern language. So, this usage needs more explanations.

Pattern Language dependencies

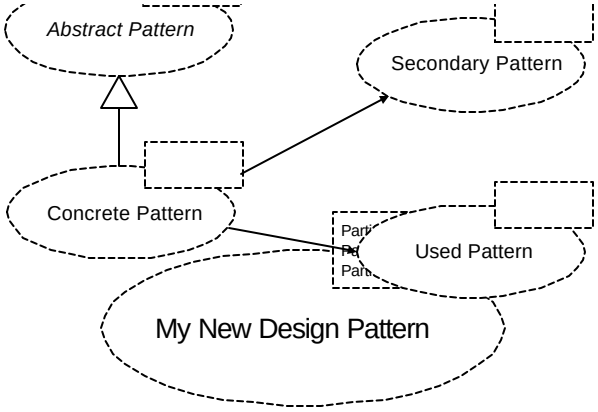
Patterns Languages come from the architecture ([Alexander+77]). However, evens in the original Alexander work, the introduction show how all patterns can fit together. Now, we must take into account the existence of several pattern languages, and show how the pattern language depends on others.
The Pattern Languages dependencies diagram (figure 1) **looks like UML class diagram** as represented in [Booch+98] p. 87 & p. 176, [Rumbaugh+99] p. 380 and in [Jacobson+99] p.202. Pattern Languages are represented as stereotyped packages because, following the definition ([Rumbaugh+99] p. 378) a package is "a grouping of model elements and diagrams" and "a general-purpose namespace that can own any kind of model element". The "Pattern Language" stereotype is from my own, but seems convenient to make these packages.

Figure 19: Pattern Language dependencies example.

We represent here two kinds of relationships:

9. Dependency relationships: Our pattern language conflicts with the other (it offers a different solution to a same problem), or it uses the other as an upstream or downstream domain.
10. Generalization relationships: Our pattern language is built upon an existing one.

Pattern representation



At the head of each pattern, we summarize the following description by using the standard UML representation for design patterns: the parameterized collaboration. Such a modeling element is represented as follows:

Figure 20: A parameterized Collaboration representation example.

The parameterized collaboration shape is composed of two shapes:

- A dashed ellipse, which shows the Collaboration. A collaboration is a "general arrangement of objects and links that interact within a context to implement a behavior" ([Rumbaugh+99], p. 195). The name in the center of the ellipse is the pattern name.
- A dashed rectangle on the upper right corner. This dashed rectangle is the general UML representation for parameterized elements. Names in this rectangle are the pattern participants.

Patterns relationship representation.

Because there is nothing to represent collaboration relationships in UML diagrams, my own representation is out of normalization. Anyway, in these diagrams, patterns are still represented as UML parameterized collaborations, with the same shape, formed with a dashed ellipse and a dashed rectangle. In order to keep these diagrams clear, parameters in the dashed rectangle are omitted. Such a diagram looks like figure 21.
In this pattern language, figure 2 takes advantage of this pattern relationship representation. The former example is a summary of our notation.

- The Concrete Pattern specializes the Abstract Pattern. In patterns vocabulary, we also say that the Concrete Pattern is a particular variant of the Abstract Pattern.
  If needed, an abstract pattern may be represented with an italic name. We don't have abstract pattern in our pattern language but, in short, an abstract pattern is a pattern that can't be used directly, but only through its variants. Generally, an abstract pattern is a "solutionless" pattern. For example, **Proxy** ([Gamma+95, p. 207]) can be considered as an abstract pattern. Only the differents variants are concrete patterns (remote proxy, virtual proxy, protection proxy…).

The Secondary Pattern and the Used Pattern are both patterns that came before the Concrete Pattern. The Concrete Pattern "knows" the Secondary Pattern and Used Pattern, but the reverse is not true. There are several way to verify such assertion: The Concrete Pattern may use others as "building blocs" or may be connected to these one to make a larger whole. This last explanation can be applied to our use of the relations shown by arrows.

Figure 21:Patterns relationships diagram example.

Context use case representation

Context sections are illustrated with Use Cases, as described in [Schneider98] and [Jacobson+99]. Use case diagrams contains two kinds of elements:
Use cases: Are represented as ellipses. A use case is a specification of a behavior that a system can perform by interacting with outside actors ([Rumbaugh+99], p. 488).
Actors: Are represented as stick figures. Actors are an abstraction for entities outside a system that interact directly with the system. Actors can be humans as well as other systems, subsystems or devices. Anyway, even if they are software or hardware, actors are represented with human-like stick figure, which may lead to confusion.
A use case diagram looks like this:

Behaviors shared between several use cases may be separated in used use cases. "Save deal" and "load deal" are such use cases.
Optional behaviors can be separated to extend use cases. "Select valuation formula" is this kind of use case.

The overall system is surrounded by a rectangle, so that we can see clearly that actors are outside the system.

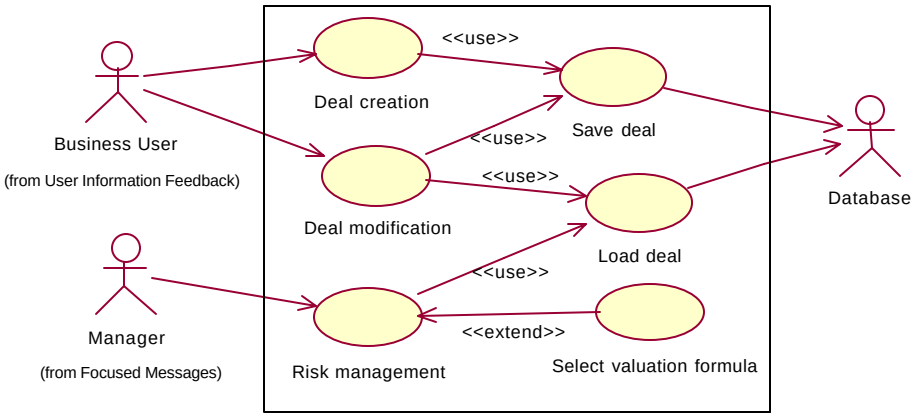


Figure 22: Use case diagram example.

Structural diagram

UML structural diagrams are well known, so it's useless to describe it again. We use it to illustrate the solution section.

Collaboration diagram

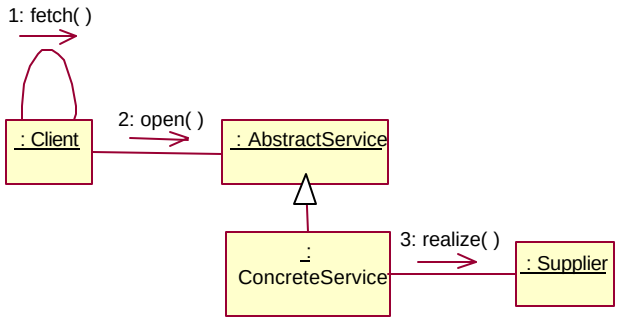


Figure 23: A collaboration diagram example.

We can use two kind of object diagrams that describe the pattern behavior:

1. Sequence diagram: Is used when time ordering seems important. It is also a clearer diagram when only a little number of objects interact, with a great number of messages exchanged.
2. Collaboration diagram: Is used when space ordering seems more important than time ordering. It is also a clearer diagram when a lot of objects interact with a reduced number of messages exchanged. Since UML 1.3, collaboration diagrams show another feature: they can show inheritance, as the former diagram does.

CRC cards

Participants are captured as CRC cards ([Bellin+97]), a widely used technique in the pattern community ([Buschmann+96]).

Typographic representation

In the former pattern language, we have used the following conventions:
Participant: We underline the participant names within each pattern.
Pattern: Each pattern name is presented with a bolded and underlined typography. The only one exception is the pattern described itself, for which we don't use any special typography.
`Method and code`: We use a courier font for these pieces of implementation.

Rationale

- When UML diagrams can be applied, we avoid mixing with non-UML elements. In fact, we use non-UML representations with CRC cards, but all other diagrams are UML compliant. By doing so, we avoid confusion, but we don't necessarily avoid misunderstanding.
- In order to avoid misunderstanding, we explain UML used concepts and their usage in the pattern context. This is the goal of this Notation Pattern.
- To avoid confusion we gave explanations in their pattern usage context.
- Where UML don't exactly match, we use stereotypes ([Rumbaugh+99], p. 449). When UML don't match the need, we use non-UML but standard notation. That's the case of CRC cards.

Consequences

☺

- We now have a strong graphical representation for our patterns as well as for the pattern language itself. Nearly all sections can be graphically illustrated.
- Because we use a standard notation with a rigorous semantic, misunderstanding risks become less important.

☹

- The large number of diagrams make the patterns lengthy.
- The UML standard notation is not always natural, and we use it powerfully. If needed, readers must improve their UML skill in order to fully understand our diagrams.

Related representations

- The example illustrations were introduced by [Buschmann+96].
- "Notation, Notation, Notation"[Vlissides98] explains how concrete collaborations can be represented. Two alternatives to UML collaborations are shown: Venn diagram style and the Gamma's Pattern:role annotations.
- James Coplien Proposes an alternative structural representation based on the pattern geometric properties as well as a representation for pattern relationships ([Coplien95], [Coplien98], [Coplien99a] and [Coplien99b]).
- Different kind of relationships between design patterns are addressed by [Zimmer97].

Bibliography

[Alexander+77]: Christopher Alexander, Sara Ishikawa, Murray Silverstein - *A Pattern Language, Towns, Buildings, Construction* - Oxford University Press 1977.

[Booch+98]: Grady Booch, James Rumbaugh & Ivar Jacobson - *The Unified Modeling Language User Guide* - Addison Wesley 1998.

[Box98]: Don Box - *Essential COM* - Addison Wesley 1998.

[Bellin+97]: David Bellin & Susan Suchman Simone - *The CRC card book* - Addison Wesley 1997.

[Buschmann+96]: Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal - *Pattern-Oriented Software Architecture, a system of patterns* - John Wiley & sons 1996.

[Cleeland+96]: Chris Cleeland, Douglas C. Schmidt & Tim Harrison - « External Polymorphism », in *Pattern Languages of Program Design vol. 3* - R. Martin, Dirk Riehle & Frank Buschmann edt. - Addison Wesley 1996 - pp. 377 - 390

[Coplien95]: James Coplien - « Patterns and Idioms in circles, complex ellipses, and real bridges », in *C++ Report My 95 vol. 7 / n° 4*, pp. 54 - 59, 74.

[Coplien95b]: James Coplien – « Mercenary Analyst », in Pattern Languages of Program Design – James O. Coplien & Douglas C. Schmidt edt. – Addison Wesley 1995 – pp. 213- 214.

[Coplien98]: James Coplien - « The Geometry of C++ Objects », in *C++ Report Oct 98 vol. 10 / n° 9*, pp. 40 - 44.

[Coplien99a]: James Coplien - « More on the Geometry of C++ Objects, Part 1 », in *C++ Report Jan. 99 vol. 11 / n° 1*, pp. 53 - 57.

[Coplien99b]: James Coplien - « More on the Geometry of C++ Objects, Part 2 », in *C++ Report Mar. 99 vol. 11 / n° 3*, pp. 52 - 58.

[Cunningham95]: Ward Cunningham - « The CHECKS Pattern language of information integrity », in *Pattern Language of Program Design* - James O. Coplien, Douglas C. Schmidt edt - Addison Wesley 1995 - pp. 145-155.
[Doble96]: Jim Doble - « Shopper », in *Pattern language of Program Design vol.2* - J. Vlissides, J. Coplien & N. Kerth edt. - Addison Wesley 1996 - pp. 143 - 154
[Ezzell98]: Ben Ezzell - *Developing Windows Error Messages* - O'Reilly & Associates 1998.
[Fowler97]: Martin Fowler - *Analysis Patterns, Reusable Object Models* - Addison Wesley 1997.
[Gamma+95]: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides - *Design Patterns, elements of reusable Object-Oriented Software* - Addison Wesley 1995.
[Grand98]: Mark Grand - *Patterns in Java, vol. 1* - John Wiley & sons 1998.
[Grand99]: Mark Grand - *Patterns in Java, vol. 2* - John Wiley & sons 1999.
[Hanmer+98]: Robert Hanmer & Greg Stymfal - « Telecommunications Input and Output Pattern Language », in *proceedings of PLOP 98*
[Harrison98]: Neil B. Harrison - « Patterns for logging diagnostic messages », in *Pattern Language of Program Design vol. 3* - R. Martin, D. Riehle and F. Buschmann edt. - Addison Wesley 1998 - pp. 277-289
[Ilog98]: ILOG Solver 4.3 Reference Manual, June 1998.
[Jacobson+99]: Ivar Jacobson, Grady Booch & James Rumbaugh - *The Unified Software Development Process* - Addison Wesley 1999.
[Lakos96]: John Lakos - « Large scale C++ software design », in *C++ Report June 1996 vol. 8 / n° 6*, p.27.
[Martin97]: Robert C. Martin - « Cross-Casting: The Capsule Pattern », in *C++ Report June 1997 vol.9 / n°6*, p.47.
[Meyers96]: Scott Meyers - *More effective C++, 35 new ways to improve your programs and designs* - Addison Wesley 1996.
[Mowbray+97]: Thomas J. Mowbray & Raphael C. Malveau - CORBA Design Patterns - John Wiley & sons 1997.
[Murray98]: James D. Murray - *Windows NT Event logging* - O'Reilly & Associates 1998.
[Musser+97]: David R. Musser & Atul Saini - *STL Tutorial and reference guide* - Addison Wesley 1997.
[Newkirk97]: James Newkirk - « Private interface », in *PLOP'97 proceedings.*
[Renzel99]: Klaus Renzel - Error Handling for Business Information Systems, A Pattern Language - Sd&m GmbH &Co, http://www.sdm.de/g/arcus/cookbook/
[Rosenberry+92]: Ward Rosenberry, David Kenney & Gerry Fischer - *Understanding DCE, OSF Distributed Computing Environment* - O'Reilly & Associates 1992.
[Rumbaugh+99]: James Rumbaugh, Ivar Jacobson & Grady Booch - *The Unified Modeling Language Reference Manual* - Addison Wesley 1999.
[Vlissides97]: John Vlissides - « Multicast - Observer = Typed message », *in C++ Report Nov-Dec 97 vol. 9 / n° 10* pp. 48-52.
[Vlissides98]: John Vlissides - « Notation, Notation, Notation », in *C++ Report Apr 98 vol. 10 / n° 4* pp. 48 - 51.
[Sommerlad96]: Peter Sommerlad - « Command Processor », *in Pattern Language of Program Design vol. 2* - J. Vlissides, J. Coplien & N. Kerth edt. - Addison Wesley 1996 - pp. 63 - 74
[Stroustrup94]: Bjarne Stroustrup - *The Design and Evolution of C++* - Addison Wesley 1994
[Zimmer96]: Walter Zimmer - « Relationships Between Design Patterns », *in Pattern Language of Program Design vol. 2* - J. Vlissides, J. Coplien & N. Kerth edt. - Addison Wesley 1996 - pp. 345 - 364

Figure List