Warning Message Accumulator

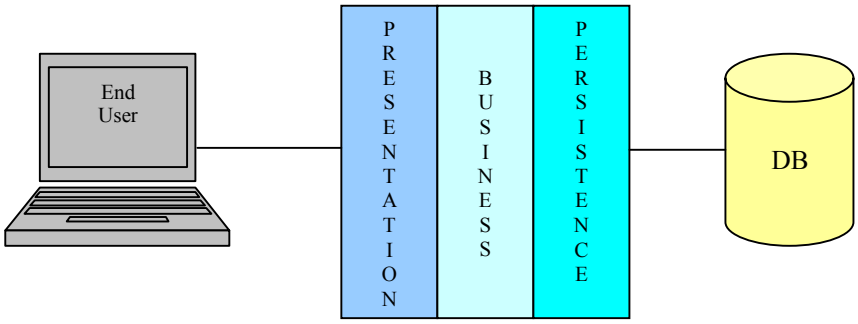
Author

Kanwardeep Singh Ahluwalia (kanwardeep.ahluwalia@wipro.com or kanwardeepa@yahoo.co.uk).

Address:
81-A, Punjabi Bagh,
Patiala – 147001
India
Phone: +91 9811016337
Fax: +91 124 4014355

Abstract. This paper presents the pattern "*Warning Message Accumulator*". The *Warning Message Accumulator* pattern applies to software systems which need to accumulate all the warnings generated during execution and display them to the end user. The pattern addresses this problem by consolidating all the warnings generated without impacting execution.

Example Consider a typical three-tier system. End-users interact with the system using a presentation layer, typically a GUI. The middle-tier, comprising the business logic, interacts with the persistence layer. The persistence layer further interacts with the database. Let us assume that on a particular event from the end-user, the system is supposed to start a long and expensive task, which requires a lot of space on the hard disk (e.g., to create a huge file) as well as the database (e.g., to store a lot of information).



Typically the validation for availability of the required free space in the file system is done in the business layer and the validation for free database space is done in the persistence layer. Consider a case when the system finds that there is not enough space in the file system or in the database to carry out the intended expensive task. An exception is thrown to the end-user from the business layer after doing a check for free file system space, even before the persistence layer validation is done. Next time the user, after making sure that there is enough free space available in the file system again starts the same task. This time the business layer does not give any warning and it executes the expensive and long task. Unfortunately, this time the end-user gets the message from the persistence layer regarding the shortage of database space.

The end-user had to wait a long time for the business layer to process the task before the persistence layer informs him about insufficient database space. The ideal solution to the above problem is to show both the messages from the business and the persistence layers to the end-user during the first try only. However, since exceptions are used for error handling, it is tough to do both the checks and show the consolidated messages.

Context Systems that have a range of error situations, including non-critical ones that should be reported to the user but which shouldn't stop execution.

Problem Error handling is an important aspect of all software systems. When certain validations fail in the system, errors are generated and execution is stopped. However, there are certain situations which are not so critical that the execution should be stopped, but still the end-user should be informed. Usually exceptions are used to handle error conditions in systems that make use of Object Oriented technologies. However, in case there are multiple warning conditions to be checked by various validations in the system during execution, then exceptions are not the best choice. Exceptions break the execution as soon as the first validation fails. This leads to the hiding of other

warnings which may arise as a result of failure of later validations. The end-user has to wait until he fixes the first warning condition and executes the same action again to be shown messages from later validations.

The problem is that the end-user should be shown all warning messages generated by validations at various stages of execution in the system at once. To address this problem, the solution must resolve the following *forces*.

- *Availability of all warnings.* The end-user should be able to see all of the warning messages at once.
- *Continued execution.* The system should not stop the execution if a warning condition is reached. Instead it should collect all such warnings encountered during execution and show them to the end-user after execution is over.
- *Flexible.* The solution should be simple and flexible. It should fit for simple as well as complex systems making use of threads, etc.
- *Extensible.* The solution should be able to accumulate all forms of warnings in the system, whether these warnings are strings, alarms or any other form, which may be even persisted.

Solution Store the messages generated while executing a task and pass the control of execution to the next level without stopping the task. Introduce a Message Accumulator, which accumulates all the warnings generated by validations at different stages of execution. The Message Accumulator will keep track of the messages for a particular task.

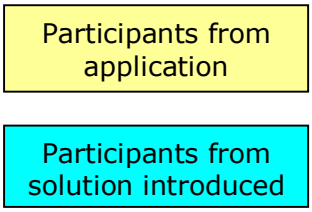
When validations encounter a warning situation, they will ask the Message Accumulator to collect the generated warning information and continue execution. After the execution is completed, all accumulated messages from the Message Accumulator can be shown to the end-user.

Structure The following participants form the structure of the Warning Message Accumulator pattern.

- A *Message* represents the warning information. It is a part of the solution introduced by the pattern.
- A *Message Accumulator* stores all the Messages generated during the execution of a task (a unit of work that involves multiple participants). It is a part of the solution introduced by the pattern.
- A *Validator* checks to see if a critical situation is reached during execution. It is a part of the application code.
- A *Task Handler* that manages the execution of a task in the system. It is a part of the application code. The basic function of the task handler is to receive the requests from the end-user and delegate them to the lower layers. After the execution is complete, the task handler responds back to the end-user. In case of multithreaded applications, task handler has more work to do. Since there are multiple tasks being executed in the system at any given time, so the task handler also keeps mapping of a task with respect to the thread which is executing it. This is not required in the case of single threaded applications because there is only one active task being executed in the system at any given time. Here, the task handler does not keep a mapping of a task

with respect to the thread. Instead, it simply receives the request from the end-user and delegates it further to lower layers.

The following describes the color scheme adopted for displaying various participants.



The following CRC cards describe the responsibility and collaborations of the participants.

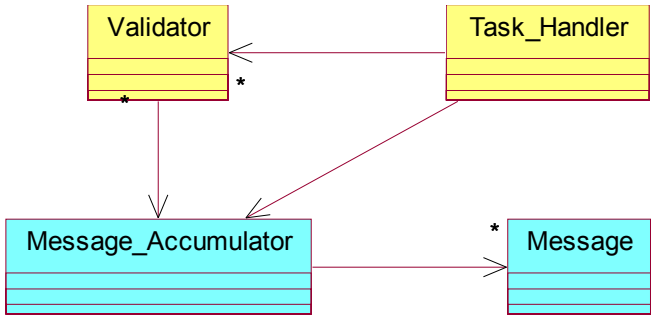
CLASS Message	*Collaborator*
Responsibility • Stores the warning information	

CLASS Task_Handler	*Collaborator* • Validator • Message Accumulator
Responsibility • Receives request for task execution • Delegates execution to other participants • Collects response from participants	

CLASS Message_Accumulator	*Collaborator* • Message
Responsibility • Maps the Message wrt a key that is unique for task being executed	

CLASS Validator	*Collaborator* • Message Accumulator
Responsibility • Checks for warning situations during execution of a task	

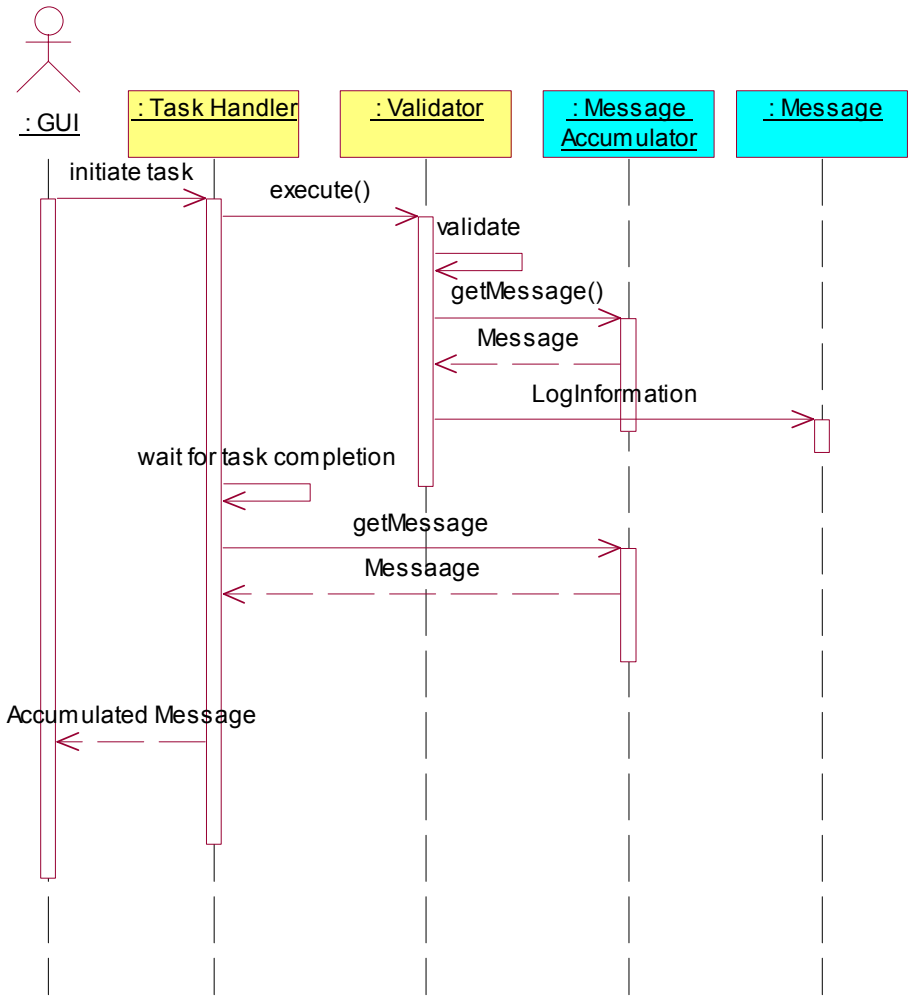
The following class diagram illustrates the structure of the Warning Message Accumulator pattern.



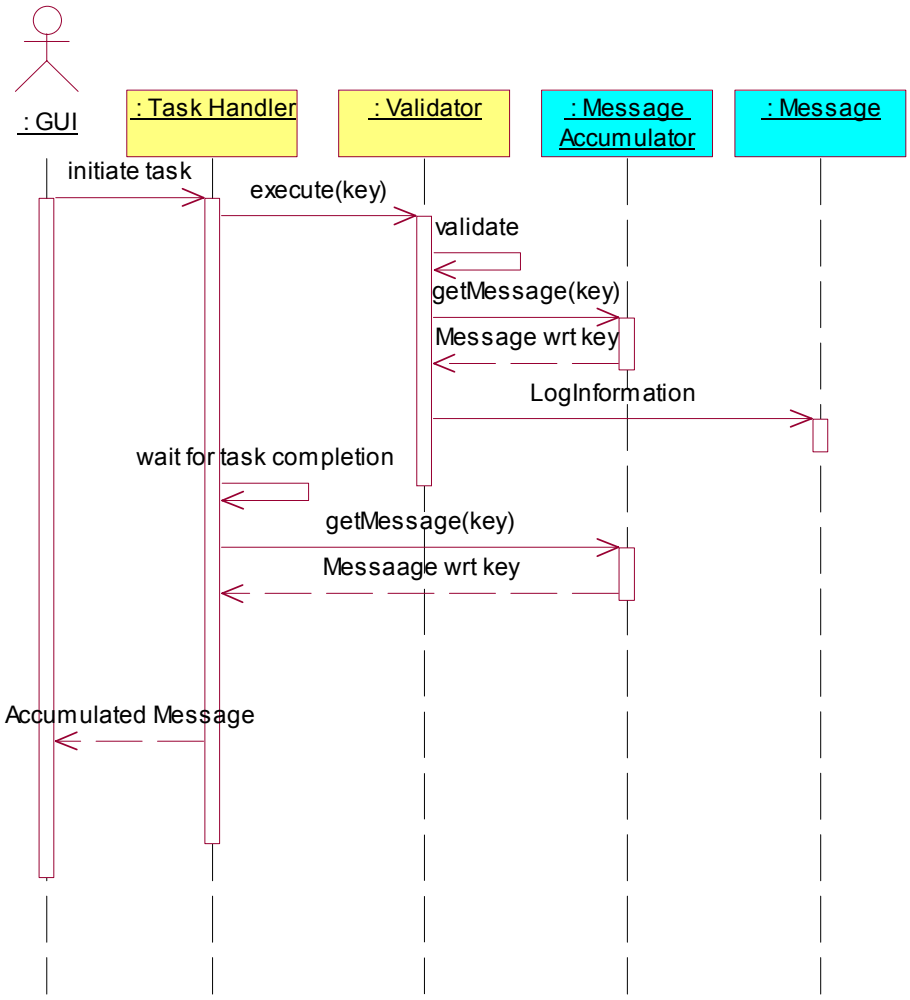
Dynamics *For Single threaded applications:*

In single threaded applications, the solution is fairly simple as there is only one active task in the system at any given time. The task handler receives a request to initiate a task from an actor, e.g., GUI. It delegates the execution of the task to the validator to see if some critical situation is reached. If yes, then it asks the message

accumulator to provide an instance of message where the information about that particular situation can be logged. If the message instance does not exist, then the message accumulator creates an instance of message and reuses it when requested by validator in the future.



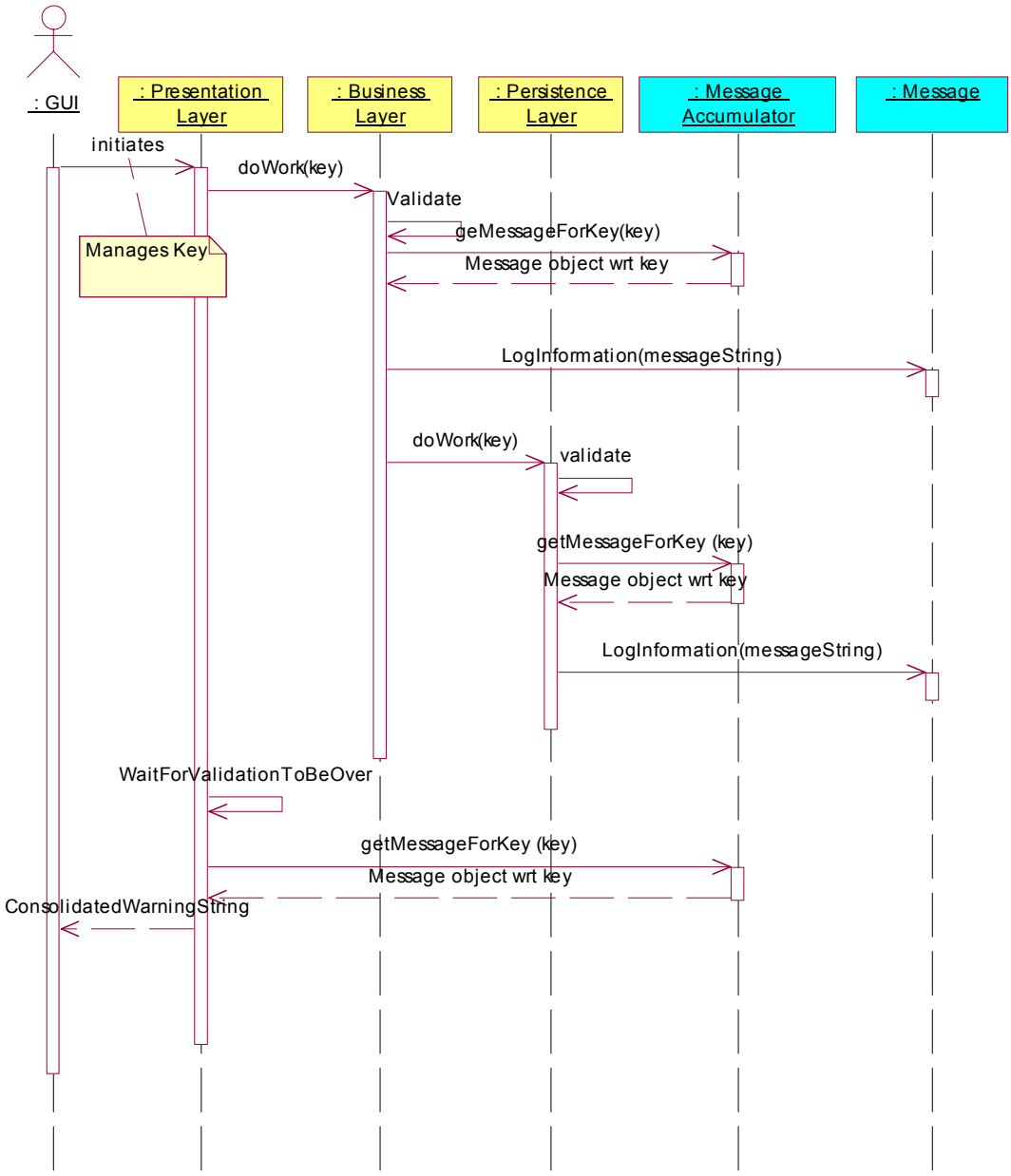
For multithreaded applications:
In multithreaded application, there can be multiple tasks being executed in the system at any given time. The message accumulator keeps a mapping of messages with respect to tasks. The task handler receives a request to initiate a task from an actor, e.g., the GUI. In addition to this, task handler usually identifies a unique key for that particular task (e.g., thread id of a thread assigned to a task for complete duration of its execution). Then it delegates the execution of task to the validator, along with the unique key to see if some critical situation is reached. If yes, then it asks the message accumulator to provide an instance of the message where the information about that particular situation can be logged. The message accumulator further checks if there is an existing instance of the message for that particular task. If not then it creates an instance of message for that particular task and reuses it when requested by the same task in future.



Logging of message can be done by multiple validators in the system. The message accumulator will accumulate the messages generated by multiple validators.

Upon completion of the task, task handler asks the message accumulator to provide all the accumulated messages, if any, for the completed task. The task handler will then forward these messages to the actor who initiated the execution of task.

Example Resolved Consider the example in which the end-user wants to have information about the shortage of file system space and database space during the initial execution. In this case the layer that interacts with the user interface can act as task handler. The business and persistence layers can act as validators because these layers perform available space validations. Assuming that the task handler allocates a particular thread for the execution of the task, the thread id can be used as a unique key for the task. The following interaction diagram shows how the example can be resolved using the Warning Message Accumulator pattern.



Consequences There are several **benefits** of using the Warning Message Accumulator pattern:

- *Availability of all warnings.* The Warning Message Accumulator pattern allows the end-user to see all the consolidated warning messages encountered during a particular execution. The message accumulator collects all the messages, which are later shown to the end-user.
- *Continued execution.* The Warning Message Accumulator pattern allows continued execution if a warning is encountered. The execution is not stopped as it is a warning condition and not an error condition. The warning messages are collected by the message accumulator and the execution continues.
- *Versatile.* The Warning Message Accumulator pattern is flexible. It can be used for simple single threaded application as well as multithreaded execution. In multithreaded applications, since there are multiple tasks being executed at any given point of time, message accumulator keeps a mapping of messages with respect to tasks that generate those messages.
- *Extensible.* Using Messages, any type of information can be stored. Since it does not define any data type, messages can be strings, application specific objects like alarms, etc. Further, warning messages can be persisted easily by mapping the message class to a database table or simply a flat file. This will allow the messages to be available for reference in the case system goes down without any prior notification, e.g., in case of a crash.

There are some **liabilities** using the Message Accumulator pattern:

- *Execution constraints.* In case the system generates warnings that do not impact the flow of execution, i.e., flow is not required to be diverted when non-critical situations are encountered, then the Warning Message Accumulator pattern can be easily used. However, when a system faces critical situation and still wants to continue but with a modified flow, then care has to be taken to divert the flow to the next level where further validations can be done.
- *Memory.* Since all the messages are being accumulated by message accumulator, system may run short of memory if the rate of generation of messages is too high and/or the message size is significant. This problem suggests the conception of other patterns. These patterns may be addressing this problem by showing accumulated messages to the user even before the task is completed and destroying them so that the memory is released.

Known Uses

Compiler. A typical compiler compiles the source code and simultaneously displays the warnings encountered during compilation. It does not stop compilation when warnings are encountered. The warning messages can be displayed to the end-user either in some file/console or some other storage media while still keeping the compilation going.

Scheduler application. A typical scheduler while starting a scheduled task does some validations that may result in warning conditions. These situations are not so critical that the intended task can not be started. For instance, if the file path for log generation specified by the user does not exist, then the scheduler will still execute the task, but generates the log in some other default path. Here it collects the warning message but still continues execution.

This warning message is shown to the end-user at the time of scheduling the task.

Order Management System. In an Order Management System, usually a *Wizard*-like UI is provided for order creation. In the *wizard* form, the user can enter the information in parts on each *page* and then press the *Next* button to reach the next page. After the information is added to the last page, the user likes to validate the information entered in all pages before executing the order creation. Here, validation by the system gives the user a detailed report of all possible warnings/errors in one attempt rather than one at a time.

Separation of an employee from employer. The Warning Message Accumulator has a real-world known use case. When an employee resigns from a company, he is supposed to get a 'No objection certificate' from various departments. Once this process is started by the human resource department, the required form goes to all concerned departments like administration, library, finance, etc for their certification, which states that employee does not owe anything to their department. If a department finds that the employee owes something to them, the proper message is filled-in on the form and the form is passed on to other departments for their verification. Once it is routed through all the concerned departments, human resource hands over the form to the employee. The employee looks at all the information filled in by various departments and acts accordingly. Here departments are acting as validators and human resource department as task handler.

See Also Warning Message Accumulator allows continued execution in situations arising out of not so critical or meaningful errors. The pattern Meaningless Behavior in CHECKS Pattern Language [1] also mentions about ignoring situations which does not have any business sense and continue the execution. Both these patterns allow continued execution in case of non-critical errors.

While the Warning Message Accumulator is more suitable for systems executing tasks in transactional modes or for enterprise applications, its usage can be extended to real time systems having continued execution by the application of the patterns George Washington is still Dead, The Bottom Line, Five Minutes of No Escalation Messages and IO Gatekeeper from Telecom IO pattern language [2]. The application of these patterns along with Warning Message Accumulator will make the display of accumulated warning messages more meaningful to the end-user.

Credits I would like to thank Robert S. Hanmer who was my shepherd for PLoP'05 for his valuable review comments.

I would also like to thank the authors of the book POSA Volume 3 [3] – Michael Kircher and Prashant Jain for their guidance and valuable suggestions while writing this pattern.

Credit also goes to the participants of writer's workshop at PLoP'05 who gave very useful comments.

References [1] 'CHECKS Pattern Language of Information Integrity' by Ward Cunningham (PLoPD1).
[2] An Input and Output Pattern Language (Telecom IO) by Robert S. Hanmer and Greg Stymfal.
[3] 'Pattern-Oriented Software Architecture (POSA) volume 3' – Michael Kircher and Prashant Jain.