A Software Fault Injection Pattern System ([1])

Nelson G. M. Leme
nelsongm@ic.unicamp.br

Eliane Martins
eliane@ic.unicamp.br

Cecília M. F. Rubira
cmrubira@ic.unicamp.br

State University of Campinas (UNICAMP), Brazil
Computing Institute (IC)

I. Introduction

Fault Injection. Nowadays, the Fault Injection technique has been considered very useful to evaluate the behavior of computing systems in the presence of faults. This happens because the technique tries to produce or simulate faults during an execution of the system under test, and then the behavior of the system is observed.

Problem. Among the various methods to perform Fault Injection, the technique of *Software Fault Injection* is getting more popular. In this technique, a special piece of code, associated to the system under test, tries to simulate faults. Generally, Fault Injection testing can be done by using a Fault Injection tool, and there is a number of them. However, there are no tools that work under each and every computing environment. Also, new kinds of systems, that work under different conditions, are created by developers. Therefore, although there are a number of fault injection programs, there is a need for more of them.

Proposed solution. *Patterns* may ease the development of new fault injection programs. By creating a pattern system for Fault Injection tools, that will allow developers to develop tools for computing environments that currently do not have a Fault Injection tool.

II. Architectural Pattern: *Fault Injector*

Example

Already many tools that use Software Fault Injection have been developed, and they employ an architecture in which is possible to inject faults; to monitor the system under test; to activate the system; control the whole process; and inform the user about the test results, as well to receive his or her requisitions. The architecture for these tools should allow the realization of all these activities.

Context

A developer is trying to architect a Software Fault Injection tool.

Problem

A program or a tool that performs Software Fault Injection (for now on, it will be just refered as “tool”) should do the following activities, in order to perform fault injection reasonably:

(i) In the beggining, the tool should activate the system under test; if the system is idle, the ocurrence of a fault will have no consequences. Only when it is having to do some task is that the presence of a fault may cause some unexpected behavior of the target system.

(ii) After that is done, the tool may inject faults inside the target sytems, as the user has specified.

(iii) Following the injection, the monitoring of target system should begin, in order to verify if the system behaves as expected.

(iv) The activities of fault injection, monitoring and activation should be properly controled and coordinated.

(v) There should be an user interface in which the user of the tool can specify the faults that he or she intends to inject, and also where he or she can get the results of the experiment.

Therefore, it is necessary an architecture to the fault injection tool that allows the execution of all these activities. Also, it is required that the following *forces* be balanced:

- It should be possible to change the way in which some of the activities above are done, without by that interfering in the other activities.
- Only the modules that really need to communicate with the system under test should be allowed to do so. It should be avoided that control and user interface activities communicate with the target system, because those activities do not deal with aspects of the execution of the system under test.
- There should be a coordinate way to the modules that perform the injection, monitoring and activation activities to communicate between themselves, in order to avoid that they change redundant messages.
- The tool shoud be easily rewritten to work in a different computing environment; it should not be restricted to just one environment, as many Fault Injection tools are.
- The architecture exposed in this pattern should be designed in a way that the Fault Injection tool would not be restricted to test just one specific type of application.

Solution

The fault injection tool shoud be structured in the following way. It will be defined five subsystems:

- *Activator:* it activates the target system, allowing it to be tested in its normal conditions.
- *Injector:* it does the injection of the faults inside the system under test.
- *Monitor:* it monitors the target system, in order to verify if it is operating as expected.
- *Controller:* it controls the subsystems above, so they do their activities coodinately.
- *User interface:* it receives the specifications from the user for the execution of the experiment and it gives back the results.

Of the subsystems above, only *injector*, *monitor* and *activator* communicate with the system under test. They should exchange messages through the *controller* subsystem, which makes the coordination between them. The *controller* will receive experiment specifications from the *user interface*, which it will use to define the parameters that will be passed to the subsystems. These, afterwards, will send back to the *controller* the data obtained in the execution of the experiment, data that will be stored and passed back to the *user interface*.

Beyond those subsystems, there are other two. These latter subsystems help the execution of the subsystems above, acting as data repositories:

- *Fault manager:* this data repository stores the faults to be injected.
- *Monitored data manager:* this repository stores the data orginated from the monitoring of the system under test.

Each one of the above subsystems communicates in a already specified and high level way, so they can be replaced, without interfering in the others.

Structure

The following diagram shows the structure of the solution proposed on the previous section.

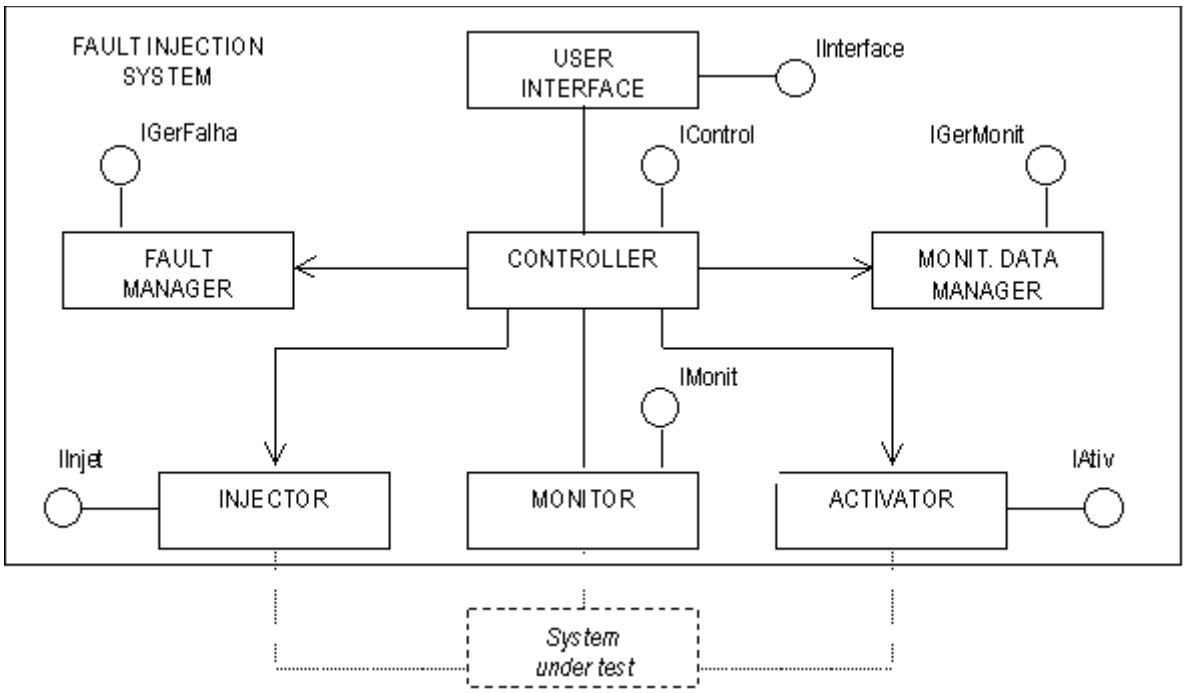

Fig. 1: "Fault Injector" architectural pattern structure.

The rectangles correspond to the subsystems described in the "Solution" section, and the circles represent the interfaces for each subsystem.

Dynamics

A possible execution scenario of a system that follows the present pattern could be the following:

- **(i)** The execution starts at the *controller* subsystem.
- **(ii)** The *controller* asks the *interface* to demand from the user the data about the experiment. The *interface* get that data and it gives back the data to the *controller*, which by its time stores the data in the *fault manager*.
- **(iii)** The *controller* asks the *activator* to start the system under test. The *activator* tries to do so. In case of failure, the experiment ends, and the *controller* asks the *interface* to show an error message.
- **(iv)** In case of success of the *activator*, the *controller* asks the *fault manager* data about a fault (or faults) to be injected. This data is then passed to the *injector*, which starts the fault injection, and it informs the *controller* about the progress of the injection.
- **(v)** The *controller* starts the *monitor*, which begins to monitor the system under test. The data obtained in that process is passed to the *controller*, which stores it in the *monitored data manager*.
- **(vi)** After all the faults have been injected and gathered all the data from the target system, the *controller* finishes the experiment, deactivating the *monitor*.
- **(vii)** The data gathered may be passed to the *interface*, through the *controller*.

Implementation

The following steps indicate a method to implement the "Fault Injector" pattern.

(1) *Specify the target system for the tool.* The target system for the tool should be cleared defined.
(2) *Determine the form of fault injection.* The next step is to establish how the faults will be injected in the system under test. The developer should choose the most suitable technique to the kind of system that the tool will deal with.
(3) *Specifiy the form of the monitoring process.* The developer should choose in which way the tool will monitor the target system.
(4) *Design the* injector *and* monitor *subsystems*. The developer should create these subsystems, taking into account what has been established in the steps (2) and (3). In this moment, the developer should not care about how these subsystems will interact. In order to build these components, the patterns "Injector" and "Monitor", also described in this pattern system, may be used.
(5) *Specify how the target system will be activated.* It is necessary to design some way to start the execution of the target system. After that, the *activator* subsystem may be build.
(6) *Build the* controller *subsystem*. After the subsystems of the steps (4) and (5) are defined, the developer should design the subsystem that will make the communication between them and that will coordinate their activities.
(7) *Design the data repositories (*fault manager *and* monitored data manager*)*. Once the basic operation form of the tool is defined, it may be specified how it will store its data. In this step, a structure to store the specification of the faults will be designed (this structure will be stored in the *fault manager*). Also, it will be developed a structure to store the data obtained from the monitoring of the target system. (structure which will stay at the *monitored data manager*).
(8) *Build the* user interface *subsystem.* Accordingly with what has been defined in the step (6), it will be elaborated an user interface, that will receive from the user the data needed by the subsystems of the tool, and that will also show the results of the fault injection experiment.

Consequences

The adoption of the pattern "Fault Injector" brings the following *benefits*:

- It allows that the way in which the tasks of fault injection, monitoring, activation, coordination, display and storage of data are done be changed individually without affecting the other activities. Therefore, if some subsystem needs to be changed, it is possible to reuse the code for the other subsystems.
- The pattern centralizes the communications between subsystems in the *controller*, that will then coordinate the messages that should be exchanged. By doing so, it avoids the communication of redundant messages.
- Only the subsystems that need to communicate with the target system do so. This is done so because it causes the lowest level of disturbance as possible.

But, the architectural pattern presents the following *problems*, too:

- It is not possible to make a quick and simple communication between the *injector*, *monitor* and *activator* subsystems without passing by the *controller*. This may slow the injection process, what increases the level of disturbance of the target system.
- In case that in the midst of injection process the *injector* or the *monitor* subsystem needs some extra piece of information from the user, they have to send the request to the *controller*, which by its time will send the request to the *user interface*. This may delay the execution of the target system, causing an increase in the disturbance of it.

Known Uses

JACA Fault Injection Tool [LMR01]**.** This tool is being developed at the State University of Campinas (UNICAMP), Brazil, and implements the architecture just described above. It is written in Java, and it gives an example of how to implement the pattern. It also proves the validity of the pattern's benefits.

FIRE (Fault Injection using a REflective architecture) [Ros98]**.** This is another tool built at the State University of Campinas (UNICAMP), Brazil. It proved the usefulness of the technique of Fault Injection through the use of a Meta-Object Protocol (MOP). Also, it was one of the best structured Fault Injection tools, and contributed a lot in writing this pattern.

FlexFI [BRR99]**.** A tool developed by the Polytechnic Institute of Torino, Italy, is another highly structured tool. Here too, the developers of the tool acknowledges the need for building Fault Injection tools that could be more easily adapted, and the way to do so is by better structuring the tools. It also contributed a lot to the creation of this pattern.

FIESTA (Fault Injection for Embedded System Target Application) [KJA98]**.** One of the first tools that stated the need for making more adaptable Fault Injection Tools. However, the approach the developers chose here was unique. They realized that the modules *activator*, *monitor* and *injector* already exist for many systems, in the form of *debuggers*. Debuggers, like GNU's *gdb*, can change values inside a system, observe many of its aspects and can start the execution of the system. Therefore, it was just a matter of writing the other subsystems of the architecture and connecting them with a debugger. Since there are debuggers for almost any computer system, the tool could be easily changed to run in different systems.

Related Patterns

The following patterns may help the implementation of a system which is based in the architectural pattern "Fault Injector":

- The design patterns "Injector" and "Monitor" describe, each one of them, how to structure the *injector* and *monitor* subsystems of this pattern.

III. Design Pattern: *Injector*

Example

In the development of a Fault Injection tool, the first problem is to find a structure that makes the fault injection as it is, namely, a structure that will simulate the presence of faults inside the system under test.

Context

Inside an architecture for a Software Fault Injection tool, the developer wants to create a structure that will produce or simulate the presence of faults in the system under test.

Problem

Inside the tool or program that makes fault injection, there should be a module or component that is the one which effectively simulates the occurence of faults inside the system under test. This

component is in close contact with the target system, therefore it should work in the same environment and in the same conditions of the system. It should change the behavior of some element inside the system under test, in such way that it seems to have a fault. It should also manage the fault injection considering the fact that they may be permanent, transient, or intermitent.

Beyond these activities, the structure that will make the fault injection should allow the balance of the following *forces*:

- The structure that makes the fault injection as it is should be easily adaptable to work under a different computing environment.
- The disturbance of the target system should be as low as possible. In order to perform the test in the closest to real conditions, the injection component should interfere the minimum in the execution of the system under test.
- It should be possible to use the same structure either in multi-thread environments or single-thread environments.
- The injector component should be easily extensible, in the meaning that it can work with new kinds of faults, added after the design of the tool has already been done.

Solution

In this section it is described a structure to inject faults. This structure will have three components, listed below:

- *Injection manager:* this component will control the injection process in itself. It will instantiate the *injectors* (described in the following item) accordingly to the kind of fault to be injected. Also, the manager will activate them in accordance with the timing of the fault. And yet the manager will be able to activate the injectors simultaneously, in case it will be working under a multi-thread environment. Finally, it will verify if the injectors managed to do their jobs, when it will return a message of success.
- *Injector:* it is instantiated by the *injection manager* to take care of the injection of some specific kind of fault. It should be a subclass of an abstract base class for the injectors, in which each subclass is written to deal with some specific kind of fault. However, the *injector* does not communicate directly with the system under test: it does so through a *physical injector* (described in the following item). When an *injector* is instantiated, this one automatically instantiates an associated *physical injector*.
- *Physical injector:* it is the component that communicates directly with the system under test, and therefore it is the lowest level component. It presents a standart interface, which contains primitive operations to communicate with the target system, accordingly with the environment of the latter one. If it is required to make the fault injection in a new environment, it is expected that only the *physical injector* has to be rewritten.

Each component should be the most cohesive and weakly linked as possible. By doing so, it may be possible to change some of these components withoud interfering in the others.

Structure

The following diagram shows the structure discussed in the previous section.

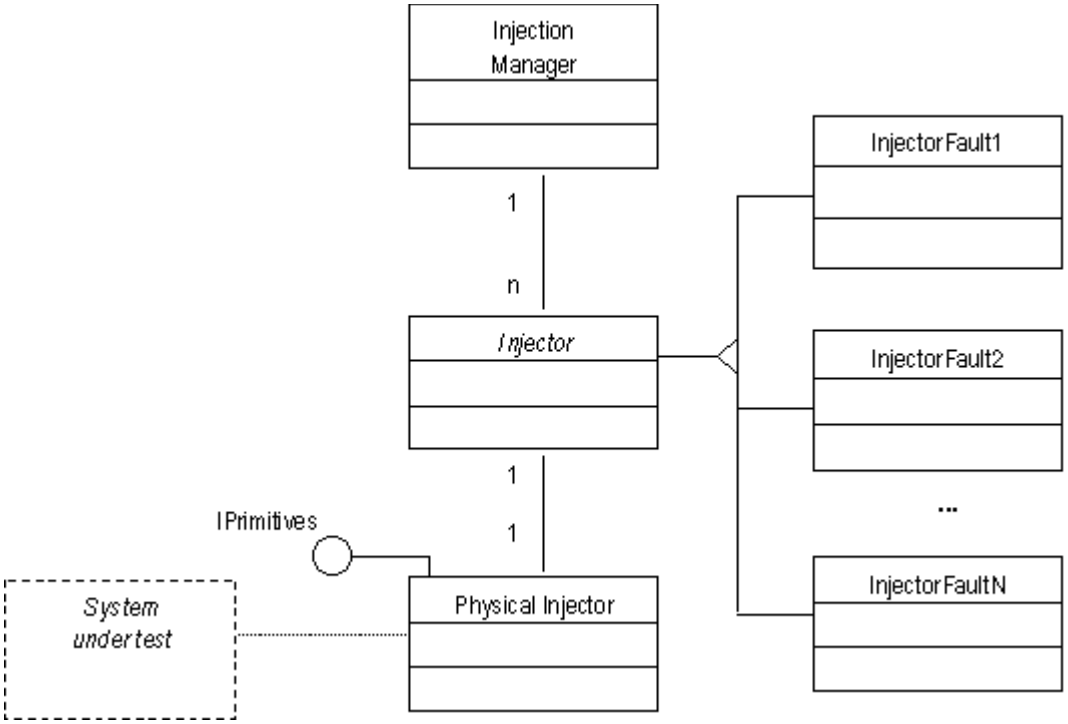

Fig. 2: "Injector" design pattern structure.

The interface *IPrimitives* refers to the set of the primitive operations for communication with the target system.

Dynamics

A scenario for the execution of the structure previously exposed could be the following:

- **(i)** The *injection manager* receives the specification of a fault (or faults) to be injected in the system under test.
- **(ii)** It instantiates the *injectors* that take care of each kind of fault received.
- **(iii)** By its time, each *injector* instiates a *physical injector* to communicate with the target system.
- **(iv)** The *injection manager* activates each of the *injectors*, so they make the fault injection. If it is working under a multi-thread system, the *injectors* can be activated simultaneously.
- **(v)** Each *injector* performs the fault injection through the *physical injectors*.
- **(vi)** If any of the faults to be injected in the target system presents a repetition pattern, the *injection manager* activates its *injector* again, following steps (iv) e (v).
- **(vii)** Each *injector* returns the result of its operation.
- **(viii)** The *injection manager* receives the results of all *injectors* and returns if it was successful or not to inject the specified faults.

Implementation

A possible way to implement the "Injector" pattern is indicated below:

- **(1)** *Specify if the program that will do the fault injection can be multi-thread or not.* The fact that the tool is multi-thread or not may change the way it works. Namely, if it will be able to inject several faults simultaneously (if it is multi-thread) or not.
- **(2)** *Establish a fault model.* The developer should elaborate a fault model with which he or she will work. This is, he or she should specify the types of faults that will be injected.

(3) *Elaborate a set of primitive operations for communicating with the target system.* In this moment, it should be elaborated a set of primitive operations that allows the communication with the target system.

(4) *Implement the* physical injectors. Once step (3) is done, the *physical injectors* should be implemented. These will have an interface that is the set of primitive operations established in step (3). In this step the developer will have the closest contact with caracteristics of the environment.

(5) *Implement an abstract base class for* injector. It should be created an abstract base class for the *injectors*. This class will establish a common interface for all *injector* and it will also have some default methods that the *injectors* may reuse.

(6) *Implement the* injectors *for each kind of fault.* Using the abstract base class defined in step (5), the *injectors* for each kind of fault are implemented, accordingly with the fault model established in step (2). In this process, it should be take into account the set of primitive operations for communication with the target system defined in step (3).

(7) *Implement the* injection manager. It should be determined how the *injection manager*, based on a fault specification received, will instantiate the correct *injectors* and activate them in the right time. In this step, it should be taken into account whether the system is multi-thread or not, what has been defined in step (1).

Consequences

The adoption of the "Injector" design pattern brings the following *benefits*:

- It makes easier to rewrite the injection component to work in a new environment. All the code that has to deal with specific caracteristics of the environment is concentrated in the *physical injector*. If it is required to change the environment, the changes in the code will be restricted to the *physical injector*.
- It is easily extensible. In case it is needed to include new faults in the fault model, the developer will have only to create a new subclass of the abstract base class for the *injectors*. Also, he or she will have to modify the *injection manager* to recognize a fault specification for this new kind of fault.
- The same object structure can be used for multi- or single-thread environments. In case the *injection manager* receives the specification of several faults to be injected, it will instantiate several *injectors*. If it is a multi-thread environment, they can be activated simultaneously; if not, they can be activated sequenceally. As an additional advantage, this means that only the *injection manager* has to be modified to change from multi-thread to single-thread, or vice-versa.
- The structure needs a minimum of communication with other modules in the fault injection program. As it can be seen, the communication with other modules is restricted to a fault specification, in the beginning, and a operation result in the end.

However, the present pattern causes the following *problems*:

- The design pattern puts a level of indirection between the *injector* and the system under test, in the form of the *physical injector*. This indirection results in a overhead, which may cause a disturbance in its execution.

Known Uses

SOFIT (Software Object-oriented Fault Injection Tool). In the paper describing the Fault Injection tool SOFIT [AvT95], Avresky et al. first recognize the need for some common structures for performing Fault Injection, in order to avoid that developers would recreate the same structures each time a new Fault Injection tool was designed. Clearly, Avresky et al. were looking for a pattern, but at that time, patterns were not as widespread as today. For performing the injection itself, Avresky et al. suggest a three level structure, just as described in this pattern, and they use it in their tool, named SOFIT.

FIRE. Aiming at building a highly structured Fault Injection tool, the developers of FIRE did the same for the injection as it is, making a carefully designed structure for that task. The level of detail in the design of FIRE for that structure was very helpful in the creation of this pattern.

JACA Fault Injection Tool. In that tool, one can find the most closely implementation of this pattern. That way, it may serve as an example of how to use the structure described here for injection as it is in a real project.

Related Patterns

The following pattern is related to "Injector" design pattern:

- The "Fault Injector" architectural pattern provides an architecture for a program that performs fault injection. Therefore, the "Fault Injector" architectural pattern gives an architecture where the "Injector" design pattern could fix in.

IV. Design Pattern: *Monitor*

Example

There are many techniques to perform the fault injection as it is. However, the program that test systems using fault injection has also to *monitor* the target system in order to observe its behavior. The developer of a fault injection program or tool must define a structure that allows the program to inspect several aspects of the behavior of the system under test.

Context

Inside the architecture of a Software Fault Injection tool, the developer wants to create a structure that allows the monitoring of a system undergoing fault injection testing.

Problem

A fault injection tool or program tries to generate or simulate faults inside the system under test. But, it is also required to monitor the system so it can be verified how it will react in the presence of the faults. There should be a structure that contains and controls a number of *sensors* in contact with the target system. It is important to remember here that what is being looked after is a structure to monitor a system *undergoing fault injection testing.* It is not an objective of this pattern to propose a general architecture to monitor a system under any circunstances and for different purposes.

Such structure for monitoring should balance the following *forces*:

- The structure that does the monitoring works in a given computing environment. In the case that this environment change, the work required to adapt the monitoring structure should be as small as possible, and it should use the maximum of code already written.
- The monitoring of the system under test should cause as low intrusivity as possible in the execution of the target system. It is desirable that the monitoring process do not modify the conditions under which the target system works with.
- It is desirable that the same monitoring structure should work in a multi- or single-thread environment as well.
- The monitoring structure should allow that new aspects of the target system to be observed be added even after the structure is already defined and implemented.

Solution

The structure for monitoring a system undergoing Fault Injection testing should be totally made of software, and henceforth it would depend of shared resources with the target system. This approach has been chosen because it is considered cheaper and more versatile. The monitoring structure would have three components:

- *Monitoring manager:* this component would control and coordinate the monitoring process. To each different aspect of the target system, it would instantiate a specific *sensor* (see following item). The component would receive the data of each of the *sensors* and it would put them in data objects that would be sent to an external component.
- *Sensor:* this object is in charge of monitoring some specific aspect of the system under test. It is defined an abstract base class for all sensors. This class would specify a common interface and some default methods. From this class, all the other *sensors* would be derived, each *sensor* for each different aspect. The *sensors* communicate with the target system through *physical sensors* (see below).
- *Physical sensor:* the objects of this class makes the communication between *sensors* and the system under test. They have as interface a set of primitive operations that allows the communication with the target system without having to specify details relative to the environment. Therefore, only this component has to deal with those details.

The components above, again, should be the most cohesive and most weakly linked as possible. This way, if it is necessary to change some of them, the others may remain unmodified.

Structure

The components listed in the previous section should be structured in the following way:

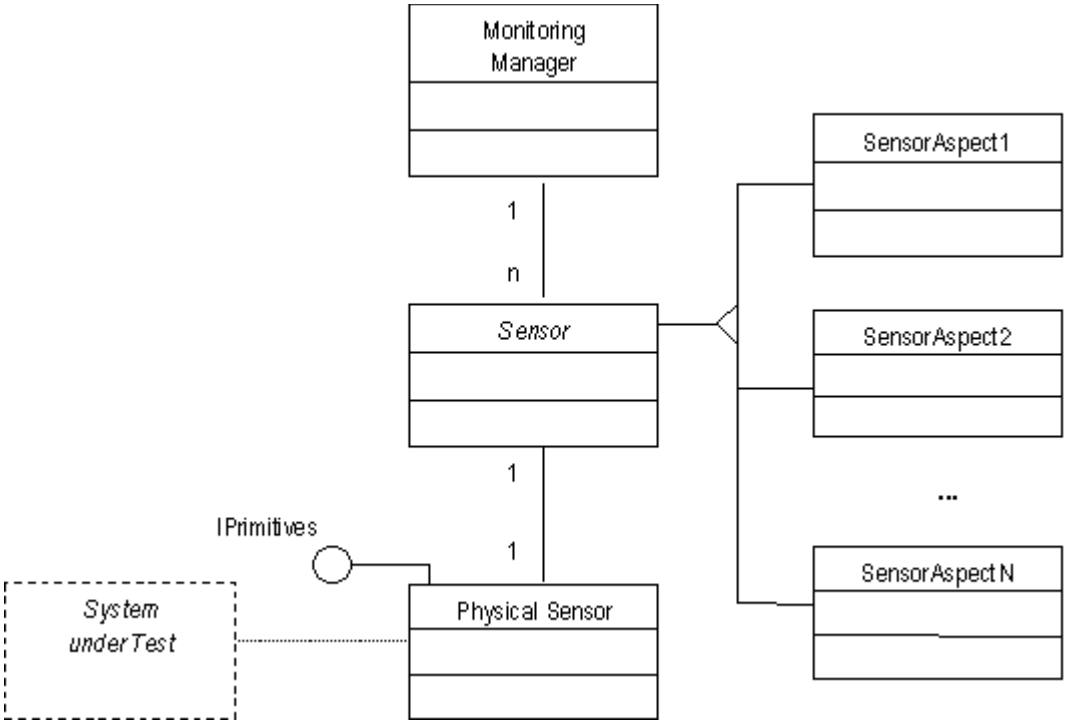

Fig. 3: "Monitor" design pattern structure.

The interface *IPrimitives* designates the set of primitive operations that communicates with the target system.

One may realize that the above structure is very much similar to the structure described in the "Injector" pattern. That happens because of the following fact. By studying monitoring theory, one can

see that a “monitor” is a process that waits for some event, and when that event happens, it may trigger an action. In fact, an “injector” is in fact a specialized case of a “monitor”, in which the injector waits for an event (the start of execution of the system under test, for example) and that will trigger an action, which is the simulation of the presence of a fault.

Dynamics

The following scenario for a execution of a component based on the pattern can be elaborated:

(i) The *monitoring manager* receives a requirement for monitoring the system under test.
(ii) The *monitoring manager* instantiates *sensors* to monitor each different aspect of the system.
(iii) Each *sensor* instantiates its associated *physical sensor*.
(iv) The *monitoring manager* activates the *sensors*. If it is a multi-thread environment, they are activated simultaneously. Otherwise, they may be activated sequenceally.
(v) The *sensor* communicates with the *physical sensor*, through the primitive operations for communication, to start to gather data from the target system.
(vi) The *physical sensor* communicates with the system under test and get the required data. It passes it to the *sensor*.
(vii) The *sensor* gives the data back to the *monitoring manager*.
(viii) The *monitoring manager* packs the data in a data object and it sends the object to the external component which has required the monitored data.
(ix) The process is repeated from step (v), while it is necessary to gather data from the target system.

Implementation

Below it is displayed the steps for the implementation process of the monitoring structure:

(1) *Specify if the system will work in a multi- or single-thread environment.* One of the key points in monitoring is to know whether the monitor will work in a multi- or single-thread environment.
(2) *Describe what data from the target system is required.* It should be determined what aspects of the target system will be monitored, and what data will be gathered from them.
(3) *Formulate a set of primitive operations to communicate with the target system.* A set of primitive operations that allows the communication with the target system should be created. These operations should have not to specify caracteristics of the environment.
(4) *Implement the* physical sensors. Based on the operations created in step (3), the *physical sensors* should be designed, with that set of operations as their interface. Only in this step is that the developer has to take care of the given caracteristics of the environment.
(5) *Implement the* sensors. An abstract base class for the *sensors* should be elaborated, in order to specify a common interface and also some default methods that they may reuse. After that, taking care of the aspects defined in step (2), the *sensors* are built.
(6) *Implement the* monitoring manager. The *monitoring manager* should be implemented. A way for instantiating and activating the *sensors* should be specified, taking into account what was defined in step (1). Also in the present step, it will be defined how the *monitoring manager* will pack the data in objects.

Consequences

The adoption of the “Monitor” design pattern brings the following *benefits*:

- It makes easier to adapt the monitoring structure to work in a new environment. If it is required to work in a new environment, it is expected that only the *physical sensor* has to be rewritten.
- It can be easily extended. In the case that new aspects of the target system have to be monitored, what is required is the creation of a new subclass of the abstract base classe for the sensors, that will monitor the new aspect, and a modification of the *monitoring manager* so it can instantiate this new class of *sensors*.
- The same structure works in multi- or single-thread environments. The multi-thread environment is preferable. But, if this is not possible, a scheme such as polling could be used, and the structure would be the same.
- The monitoring structure requires a minimum of external communication. In the beginning it receives a solicitation for monitoring the target system from some external component. While it is monitoring, the structure sends back data objects. This helps keeping the level of intrusiviness to a minimum.

However, the "Monitor" pattern causes the following *problems*:

- The design pattern puts a level of indirection between the *sensor* and the system under test, which is the *physical sensor*. This may increase the level of intrusiviness.
- It may also increase the intrusiviness because it shares computing resources with the target system, which inevitably brings an overhead to the system.

Known Uses

SOFIT (Software Object-oriented Fault Injection Tool). For monitoring, SOFIT uses the same three layer structure that it uses for injection, and the paper describing the tool [AvT95] describes as a general structure for Fault Injection tools, just as in this pattern.

FIRE (Fault Injection using a REflective architecture) [Ros98]**.** The Fault Injection tool, FIRE, as in the whole system, uses also for monitoring a carefully designed structure, which is described in detail. Again, it was very helpful in designing this pattern.

JACA Fault Injection tool. For monitoring, too, the JACA Fault Injection tool gives the most closely implementation of this pattern, illustrating the similarities between the structures for fault injection, as it is, and monitoring.

Related Patterns

The following pattern may be related to the "Monitor" design pattern:

- The "Fault Injector" architectural pattern intends to solve the problem of how to architect a program that does fault injection. The design pattern "Monitor" fixes in the architecture proposed by that pattern.

V. Conclusion

The pattern system shown here consolidates the knowledge about how to do software fault injection. It gives a base on knowledge from which other researchers may extend fault injection, creating new ways of performing this technique. The pattern system also gives to the developers many resources in order to design a fault injection program. By doing so, it allows the development of more fault injection programs. This may make the software fault injection technique even more popular.

VI. References

[AvT95] Avresky, D. R.; Tapadiya, P. K. “A Method for Developing a Software Based Fault Injection Tool”. *Texas A&M University, Department of Computer Science, Technical Report 95-021.* Texas, USA, 1995.

[BRR99] Benso, Alfredo; Rebaudengo, Maurizio; Reorda, Matteo Sonza. “FlexFi: A Flexible Fault Injection Environment for Microprocessor-Based Systems”. *SAFECOMP 1999.* Springer-Verlag, Heidelberg, Germany, 1999, pages 323-335.

[KJA98] Krishnamurthy, N.; Jhaveri, V.; Abraham, J. “A Design Methodology for Software Fault Injection in Embedded Systems”. *Proc of the 1998 IFIP International Workshop on Dependable Computing and its Applications*. Johannesburg, South Africa, Jan. 12-14, 1998, pages 237-248.

[LMR01] Leme, Nelson G. M.; Martins, Eliane; Rubira, Cecília M. F. “A Software Fault Injection Pattern System”. *Proceedings of the IX Brazilian Symposium on Fault-Tolerant Computing.* Florianópolis, SC, Brazil, March 5th-7th, 2001, pages 99-113.

[Ros98] Rosa, Amanda. *Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos.* Mastership thesis, UNICAMP, Campinas, Brazil, 1998.