

## WINDOW TO THE WORLD

KRISTIAN ELOF SØRENSEN <ELOF@ELOF.DK> HTTP://ELOF.DK

**Version.** Conference draft for EuroPLoP 2003

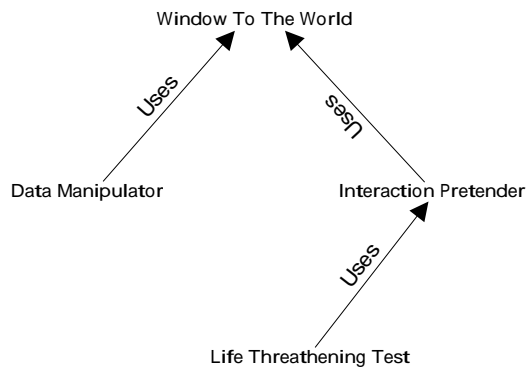
**Abstract.** You are building a system that interacts with a remote system. You intent to test this interaction.

**Window to the World** increases the likelihood that tests made with test versions of the remote system or of the communication mechanisms are good predictors for behavior when running against the real remote system. It also makes it easy to switch in between testing against various test implementations and the real remote system.

**Data Manipulator** deals with the problem of testing your systems handling of interactions that strays away from the well behaved documented way of behaving and into the land of the faulty communications.

**Interaction Pretender** allows you to test your systems interaction with other systems that are still on the drawing board when you perform your tests, or are unavailable for test for other reasons.

**Life Threatening Test** uses **Interaction Pretender** and therefore also **Window to the World** to solve the problem of testing a system where bugs can lead to destructions in the physical world up to and including killing humans.



# Window To The World

**Non software example.** You are standing inside a house beholding the view that comes to your eyes through the windows. How can you know that you are looking at the world outside this house and not a simulation presented on high definition screens cleverly camouflaged as windows?

**Context.** You are building a system that interacts with other systems, be that local sub-systems of the same program or other programs running on the same or other computers and you intend to test the interaction with these systems.

**Problem.** You want your system to be capable of interacting with test versions of the remote system and of the interaction mechanisms, just as easily as when interacting with the plain remote system.

## **Forces.**

- Switching from interacting with one system to another must be swift and painless so it does not become a hindrance to perform as much testing as possible.
- If your system can tell when it is interacting with the real remote system and possibly even tell the various test versions apart, then the likelihood that its behavior when faced with one of these will be exactly the same as when faced with another one is diminished, potentially to a point where it compromises the trust worthiness of the tests.
- Being able to switch between what to interact with during runtime can lower overall testing time because system restarts and possible recompilations in between test runs are avoided, however the ability to switch while running, means that it is necessary to be able to end all pending interactions gracefully literally “at the press of a button”.

**Solution.** Design the interaction in between your system and the remote system as a selection of operations that can be specified as remote or local method calls on an interface, including definitions of necessary data carry objects and exceptions. Include methods to signal that all pending interactions should be terminated gracefully. Make sure that all parts of the interaction do pass through an implementation of this.

## **Resulting Context.**

*Benefits.* There are nice well defined boundaries around your interaction code so you can write several implementations that each are either a strait interaction with the real remote system, a **Data Manipulator** or an **Interaction Pretender** implementation. You can plug any instance of these into your system without changing anything else.

If the mechanism to switch interaction mechanism while running is fully implemented you can even create an **Interaction Pretender** that steps your system through a selection of pretended interactions designed to test as many aspects of your systems behavior as is possible from the interaction interface. This can be used right after system start up during ordinary use of the system after deployment to the end users, as a part of a system self test. After this test has succeeded the interaction is swapped for one that interacts with the real remote system.

If you need logging of the interaction with the remote system, and have made a **Data Manipulator** that does not alter the data transmitted but merely logs it, you can switch logging on and off by swapping the logging implementation for one that does nothing but interacting.

*Liabilities.* Having to make all interaction communication pass through an implementation of interface etc. that defines the interaction can lead to a performance loss compared to tightly coupling the two parts. This loss can be minimized and in many cases entirely avoided with careful software design and implementation work. Therefore if performance is a potential problem, you have to assign good people to make the interactions.

# Data Manipulator

**Non software example.** A human interpreter with a vested interest in the matter being discussed, will pass most of the messages on un-altered merely translated, but make certain omissions and alterations to the messages now and then to further his own goals.

**Context.** You intend to test a system that interacts with another system. Maybe you have considered Interaction Pretender, but you are in a situation where you can interact with the remote system in the tests for most tests.

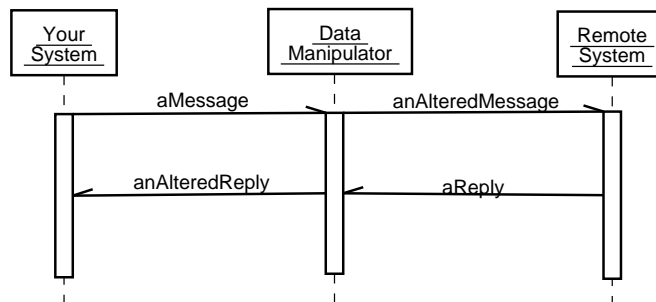
**Problem.** You want to test scenarios that are close to the ordinary execution of interactions in between your system and the remote system, but with differences that makes it hard or impossible to test against the remote system without making some modifications somewhere along the interaction flow.

**Forces.**

- You intend to test your systems reaction to interactions that takes longer than usual, but the remote system will send data as soon as they are available and has no means of delaying them.
- You intend to test your systems reaction to interactions that goes faster than usual, but the remote system has no way of operating faster than its usual maximum speed.
- You want to test with mangled data, but the remote system can only send perfect data. This simulated defective data that has sneaked into a database, bugs in code implementing data structure or communication etc.
- You intend to test your systems reactions toward interactions where the sequence of events are different than usual, but the remote system cannot change the sequence away from the customary one on command. In a real production situation a change in sequence is likely to be due to something like a retransmission on a network, which is hard to provoke easily and reliably enough for use in a test case.

**Solution.** Make implementations of **Window to the World** that passes on data in between your system and the remote system but makes certain changes to the data or to the sequence of individual parts of the data.

You can delay data, change data and alter the sequence of data this way, but you cannot speed up interactions. Therefore you will need to implement an **Interaction Pretender** in order to test your systems reaction to faster than normal interactions.



### **Resulting Context.**

*Benefits.* **Data Manipulator** lets you test behavior in the face of several kinds of hard to provoke in the real world scenarios such as out of sequence interactions, slow interactions and errors in the interaction data.

As an additional benefit you can use a **Data Manipulator** to log interactions, by making an implementation that passes the interaction data on unaltered, but logs them.

*Liabilities.* Not all scenarios can be handled using a **Data Manipulator** implementation, use an **Interaction Pretender** for such cases.

Switching in between different interaction implementations by means of **Window to the World** gives you the liabilities of that pattern.

# Interaction Pretender

**Non software example.** The “Print to file” functionality in many office applications can be used to test the final outcome of the print without waiting for the printer and without consuming paper and ink or laser toner, if the format that is printed to can be viewed on screen immediately and looks the same as the print. This is the case when “print to file” generates the same Postscript that will be feed to the printer when “print” is pressed.

**Context.** You intend to test a system that interacts with another system. Maybe you have considered **Data Manipulator** but you still have unsolved testing problems or maybe you are facing the problem of **Life Threatening Test** and are seeking a solution.

**Problem.** You want to test scenarios that the remote system cannot take part in or where you do not want the remote system to take part.

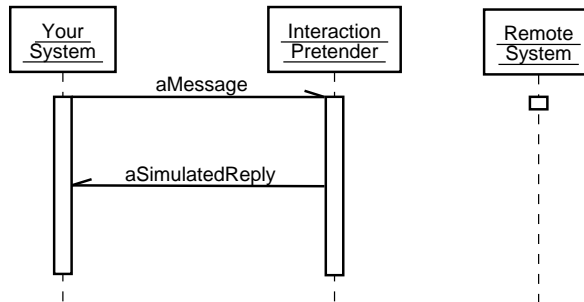
## Forces.

- It can be the case that the remote system is not finished yet when you want to test your systems interaction with it or it can be the case that it is not available for your use as often as you need it.
- You want to test an interaction with the remote system, but it might very well not be built to take part in such a test. An example is when you want to repeat a small part of an interaction over and over with slight changes to the data and timing, but the system can only act out the full interaction consisting of many steps, and it only gives you very indirect ways of deciding the data and timings to be used in the part of the interaction you are interested in.
- You want to test an interaction with the remote system, but it could not possibly be built to take part in such a test. An example is when you want to ensure that your system can handle faster interactions than what the current implementation of the remote system is capable of.
- You might or might not have the remote system available for testing, but you cannot use it because every use comes at a price that prevents you from testing as much and as often as you would like. Some examples are wear and tear on a machine, missile launches or financial transactions with a fee per transaction.
- You want to perform tests where the consequences of an error in your code or the remote system can be excessive wear or destruction of machinery or even physical harm to humans. Examples are when your system controls a remote system that is a piece of hardware such as a factory machine, an escalator, the brakes or steering in a car or a missile. In such a case you need to have your system and its interaction with the remote system thoroughly tested before you connect it to the real remote system.

**Solution.** Create a software simulation of as little of the remote system as is necessary for it to act the part of the remote system in interactions with your system during testing. Make the implementation be a **Window to the World** so you can use its specification of the operations that can take place during the interactions,

and so you are able to switch between interacting with the simulation and with the real system.

This is a tiny simulation. It can be as simple as an implementation of **Window to the World** where method calls on the interface results in a proper response being looked up in a table of request, response sets and returned.



### Resulting Context.

*Benefits.* The tiny simulation can be a lot faster than interacting with the real remote system since it has to do so little work and can be local so there is no communication channel overhead. Therefore you are more likely to test often and test a lot.

There are no longer a difference between test scenarios that are tedious or time consuming to try out and test scenarios that are not, so this is no need longer a reason to test some scenarios less than desired.

You can test scenarios that would not otherwise be testable.

You can build software that controls potentially dangerous machinery without risking calamity by operating the machinery with the earliest versions of the control software.

An interaction pretender can be used as a part of a system self test for your system when it is deployed. So having such a system self test does not lead to further complications of your system design, since you are reusing aspects of the design put in place for the benefit of testing.

*Liabilities.* The more differences there are in between talking to the real remote system and to an **Interaction Pretender** the less value will a test against an **Interaction Pretender** have. To remedy this, implement **Interaction Pretender** as a **Window to the World** and be careful that the various **Window to the World** implementations are truly interchangeable.

It is easy to be misled into thinking “all the tests passed, so my software must be perfect”. This is especially dangerous when you are drawing conclusions on your softwares behavior based on testing it against a simple simulation such as an **Interaction Pretender**. The simulation is not the real system, and no matter how through you were in implementing the simulation to simulate all aspects of the behavior of the real remote system, you can only simulate what you know about its behavior. So you have a more or less perfect simulation of the behavior of the remote system as it is described in its documentation. Your simulation is not simulating bugs in the remote system that you do not know about, and it is not simulating details of the behavior that are not part of the documentation for the remote system.

# Life Threatening Test

**Non software example.** You are developing protective clothes for forest workers designed to prevent a chain saw from cutting into the limbs of the wearer. Are you going to test the first sample of the clothes by putting it on yourself and starting up your chain saw to see if it works as expected?

**Context.** You intend to test a system that controls a piece of machinery or in some other way has an influence on the physical world.

**Problem.** How can you test a piece of software when revealing bugs during testing might kill somebody?

**Forces.**

- You do have the remote system available for testing, but you do not want to use it for testing, because errors in your software or the remote systems interaction with in can harm or destroy the machinery that is the remote system or even physically harm or kill humans.
- Because of these grave potential consequences of errors, you want to thoroughly test your system and its interaction with the remote system before connecting it to the real remote system.
- At some point you will have to let your system control the real remote system and accept the risks involved, however you want to postpone this until both your system and its interaction with the remote system has passed extensive testing.

**Solution.** Create an **Interaction Pretender** that simulates the remote systems behavior toward your system and use it for as thorough a test program as possible before you connect your system to the real remote system for the first time.

**Resulting Context.**

*Benefits.* With this solution you can test as much as you want without putting machinery or humans at risk. Such tests can give you a good enough confidence in your systems correctness, that you dare connect it to the real remote system.

*Liabilities.* When you connect to the real remote system you do run a risk of being bitten by bugs not uncovered by the tests against simulations, because the simulations differed from the real remote systems in possibly small and subtle but nevertheless important ways. This can be because of undocumented details of the remote systems behavior and bugs in its behavior, which are not duplicated in your simulation because you had no way of knowing about them until you ran into them when running your system against the remote system. This pattern does not give a solution to this problem.

## Known uses for the whole language

The Danish credit card handling company PBS provides a software simulator that's a plug&play substitute for the real interface to their system. They charge for every transaction against the real system, and they only allow transactions with real money and real credit cards against their actual systems so this simulator is necessary for all development. PSIP[1]

The Nordic credit card clearing system DIBS provides the possibility to switch to a simulation mode where you can invoke all operations without transferring real money. You can also force all documented success and failure scenarios by using specific credit card numbers. You cannot force network errors or protocol errors. DIBS[1]

For the sea sparrow missile system used by many navies around the world there exists a simulation system for on-ship training of personnel. During training the personnel on board the sea sparrow equipped warship can go through the whole process of firing a missile from planning to impact, and it all feels just as if an actual missile had been launched and detonated. In reality the actual launch, flight, interception and detonation of the missile as well as the target itself are computer simulations plugged into the ships system in the **Window to the World** fashion LEDIN[2]. See LEDIN[1] and DDJ[1].

## Acknowledgments

This pattern language evolved from the single pattern of the same title that was work shopped at EuroPLoP 2001. The help and support I received from Christa Schwanninger who was my shepherd and from the participants at the writers workshop has helped me a lot.

Juha Parssinen was a very dedicated shepherd for the EuroPLoP 2003 version of this paper. His help with everything from illustrations and patterns form, to the writing and the technical content itself lead to substantial improvements to the paper.

Thanks also goes to Linda Rising for her "bread machine" non software examples for the GoF patterns, which along with Christopher Alexanders "A Pattern language", was the inspiration for the non software examples in this language.

## References

- DDJ[1] Dr. Dobb's Journal May 2001 page 83-92.
- LEDIN[1] <http://www.ledin.com>
- LEDIN[2] private email exchange with Jim Ledin
- DIBS[1] <http://payment.architrade.com/n2butiks.htm>
- PSIP[1] <http://www.eos.dk/training/psip/emul.html>