"A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and body can connect with the universe and move bits of it about."
Douglas Adams[1]

Patterns for Interface Design

Amir Raveh
amirr@netvision.net.il
Copyright Amir Raveh © 2004

Introduction

Interfaces provide means for the system to interact with users and other systems. Designing the interfaces for the system is part of system design.

The patterns here are part of a work in progress on the topic of interface design.

The patterns listed here provide a few solutions to problems encountered in interface design.

'Kinder Egg' provides a solution regarding reducing risks that might be associated with the use of an interface.

'Fire Doors' provide a solution regarding reducing the risk of a ripple effect or avalanche, when a problem associated with an interface affects interacting systems.

'Simulatable Interface' provides a solution for problem isolation and testing of systems across interfaces.

Background and Terminology

The patterns are described here from the perspective of the systems engineering discipline. To help those who are not familiar with systems engineering or with systems engineering terminology, a short introduction is provided in this section.

A widely accepted definition for **system** is: a set of interrelated **components** which interact with one another in an organized fashion towards a common purpose[2]. These **components** include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements[3].

Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. Systems engineering brings a disciplined focus on the end product, its enabling products and services, and its internal and external operational environment[3].

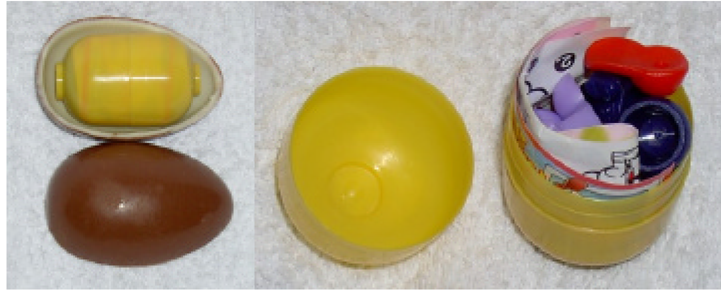
One of the roles of a systems engineer is system designer - owner of “system” product / chief engineer / system architect / developer of design architecture / specialty engineer (some, such as human-computer interface designers) / “keepers of the holy vision”[4][5].

Under this role, a systems engineer is in charge of designing the system's external interfaces, component breakdown and internal interfaces among components. The interfaces might be human-machine interfaces, mechanical, electrical, hydraulic and an ever increasing number of software interfaces – from TCP based protocols and services such as HTTP to APIs.

Name

Kinder Egg

(Safety Cap*)



Context

You are designing an interface for a system. You have some assumptions about the background, training and knowledge necessary for using this interface.

Problem

If these assumptions are violated, some of the interfaces or the potential use of the interfaces may present a risk for the people using, misusing or abusing the interface, to other people or to other systems.

Forces

- As system designers we must protect the user, other people, the system and interacting systems from potential errors that may occur in use, misuse or abuse.
- We must also design the system so that activities can be carried out in a manner that does not hinder usage.
- People may unintentionally misuse an interface through lack of knowledge or attention. Interacting systems may misuse an interface through erroneous data or programming errors.

Solution

Evaluate the potential risks associated with each interface. For those interfaces where risks can severely impact the system under design, interfacing systems or people - **introduce an artificial constraint to restrict the use of the interface in a way that filters out inappropriate usage or reduces the ability of users to harm themselves, other people, the system under design or interfacing systems.** The constraint might enforce using the interface only in a manner that does not invoke a risk.

Resulting Context

The introduced constraint allows reducing or mitigating the risk associated with the use of the interface. The constraint should be designed so as not to overburden the user or reduce the effectiveness of the interface – as this may motivate the user to find ways to bypass the constraint.

This can be done by matching the harmful consequences with the imposed extra effort, and the usage frequency. The "Are You Sure (Y/N)?" message loses its magic protection from accidental erasure, if you fall into the habit of automatically hitting the Y key, followed by Enter.

---

* We offer this alternate name, since it seems FDA regulations[9] prohibit importing Kinder Eggs into the USA. Kinder Eggs might also not be a household name in every culture.

Examples of overdose effects can be found in childproof caps on medicines – which can frustrate adults, and in cases where factory workers bypass security measures in order to keep up to the work pace required of them.

And never, ever underestimate the ingenuity of users – there are reported cases of users finding ways to bypass constraints ranging from users forcing miniature USB plugs in the wrong direction[6] (destroying connectors in the process) to a nuclear technician dying of lethal exposure to radiation after disabling all safety devices[7]

Known Uses

- A Kinder egg™ (pronounced "keender") by Ferrero[8][9] cannot be opened by 3 year olds – it takes the force and problem understanding of an older person to open it. This constraint helps protect children from ingesting the small parts contained inside the Kinder egg ™, by making sure an adult is there to open it. The picture at the top of this pattern shows the chocolate egg, the plastic container and the disassembled toy contained in it.
- Similarly, a "childproof" safety cap is designed for bottles containing drugs so that children will not be able to gain access to potentially lethal drugs, by requiring reading and coordination skills an older person has.
- Hydraulic presses and guillotines used in the metal forming industry can only be operated by pressing two switches, ensuring both hands of the operator are out of danger - to protect the operator from losing limbs.
- A chainsaw will not engage its cutting chain unless both hands of the operator are used to hold it, to reduce the risk of accidents.
- Safety locks on the entrance to protective covers and cages used in industrial automation disconnect all power to the system when they are opened in order to service a system component.
- A microwave oven will not operate with an open door, to prevent radiation injuries.
- Garbage cans in national parks in the USA are equipped with "animal proof" covers, to prevent wild animals from feeding on garbage disposed by park visitors.
- Constraints in SQL help enforce data integrity on database objects, each time they are created, updated or deleted.
- Connectors such as USB, RS-232 and Ethernet are designed so they can only be inserted in the correct orientation.
- You need to be logged in as a super-user or administrator in most operating systems in order to perform system shutdown.

Name

Fire Doors

Context

You are designing interfaces for a system. Your design needs to cover potential failures and malfunctions, to allow for a robust system or system of systems.

Problem

Sometimes, an error in a source system or unexpected output provided by it can trigger a problem in another system that is fed by the source system. This may even deteriorate a system of systems by a ripple or avalanche effect. While solutions such as **Graceful Degradation**[10] and **Partial Failure**[11]can cover for those failures you can anticipate, there is always "the one problem no one ever expected to happen". And since Murphy's Law works overtime on interfaces, failure will strike where least expected, causing maximal harm.

How can you design your system to prevent the avalanche effect triggered by unexpected failure?

Forces

- You want to keep the design of interfaces and protocols simple.
- You cannot foresee all possible combinations of failures or events. Even if you try doing so and sacrifice efficiency along the way, there is a wide range between over-validating input and no validation at all.

Solution

Design each system so that each and every interface can be shut down or disconnected, with minimal loss of functionality on the other interfaces. The shutdown might be a configuration setting, a script or a documented and tested manual procedure. Make sure there is also the appropriate way to bring back the interface.

By shutting down the troublesome interface, you protect your system from being affected by the failures the interface introduces in your system or systems that are fed by your system – the avalanche or ripple effect.

Deciding when to apply a **Fire Door** is key to success in protecting the system and depends upon accurate and timely problem detection.

Problem detection can be done by using the equivalent of **Flame and Smoke Detectors**[†]. **Watchdogs and Sentinels**[13], **Leaky Bucket Counters**[12] on failures and errors, service assurance systems or system performance measurements are examples of such detection probes. The probes can be located either on your system or on interfacing systems.

And in case all else fails, the system users and administrators will be the ones to report a problem…

Resulting Context

Once a problem is isolated to one system, sending unexpected or erroneous input to another system, the offending interface can be shut down. Problem resolution can then be performed without causing unnecessary additional deterioration in other services delivered by the system of systems.

Applying this solution has a price tag - you need to design your system so it will be able to continue to perform with little to no malfunction even when some, most or all of its interfaces are down. Some of these negative impacts may be overcome or reduced by using **Simulatable Interfaces**.

You should not let the existence of this safety feature lull you into a false sense of security – the Titanic was considered "unsinkable" because its hull was constructed of sealable compartments using waterproof doors. Yet, it was not designed to withstand hitting an iceberg.

Making the shutdown of an interface an operation that can be executed easily may result in reducing system functionality if it is given to a "trigger happy" administrator or in case of an over-reactive problem detection mechanism – such as triggering the water sprinklers because of a person lighting a cigarette.

When are **Fire Doors** are activated, be mindful that they may lock inside the system transactions or people. The system must be designed to handle such an emergency scenario with minimal impacts.

Knowing where the **Fire Doors** are, how they can be activated or which conditions cause the system operator to activate them may give a malicious user a way to undermine the system.

Known Uses

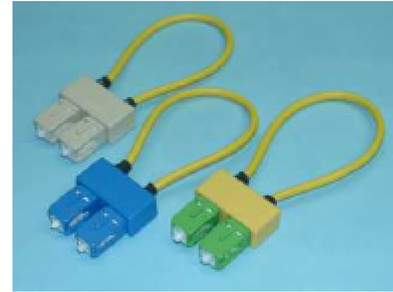
- Fire doors help prevent fire from spreading throughout a building. The picture at the top of this pattern shows a fire door of the rolling screen type.
- Marine vehicles such as ships and submariners are constructed as a set of connected compartments that can be sealed off by watertight doors, in case one compartment is damaged and starts leaking water in.

[†] Pattern thumbnail: **Flame and Smoke Detectors** allow automatic detection of system failure modes. Once a failure mode is identified, **Flame and Smoke Detectors** can trigger corrective or containing actions.

- In the design of a new telecommunication system, a requirement was made for introducing **Fire Doors** on all its interfaces to existing systems. This allows protecting the existing revenue generating cellular infrastructure from potential failures when the new system is introduced.
- Disabling a user's account after three login failures to prevent compromising system security.
- Electric interfaces include a built-in fuse designed to disconnect if the current exceeds a threshold – to avoid risking the interfacing system.
- Firewalls provide an easy mechanism that allows shutting down TCP/IP interfaces on demand or by configuration.
- An age-old workaround in Unix systems, when **Fire Doors** were not planned in advance, is a standard Unix administrator practice when problems are associated with a service: remove the service from the /etc/services file or replace the daemon handling service requests with a dummy daemon. A more recent method is to block the service at one of the firewalls. This allows putting a new **Fire Door** where you already know there is a fire…

Name

Simulatable Interfaces



Context

You are designing and developing interfaces for a system, or troubleshooting an existing system that interfaces with other systems.

Problem

The system you will interact with has not been built yet. Maybe it exists, but putting it at your desk might be too expensive or risky.
Yet without the interfacing system to respond to your requests, you will not be able to complete development and testing of your system.

Forces

- You want to keep the cost of development reasonable and keep to the schedule.
- You may have little of no ability to affect the timely delivery of the interfacing systems.
- Sometimes the cost of the system you will interface with (or its simulator) may be too high to allow using it for tests.
- It is preferable to validate a system under development in small increments during the development process.

Solution

When designing the module that implements your side of the interface, include an option in it to work in a mode that simulates a response for each request it is supposed to send out.

The response might be optimistic – as if the request was successfully process by the target system.

The response might be pessimistic – as if the request was denied.

And you can even make the interface behave according to more complex behavior rules, such as using a sub-string in one of the request fields to determine what the result will be.

Developing a **Simulatable Interface** adds to the cost of developing the interface, increases its complexity and adds more potential failure points.

Simulations allow testing the system only as far as the assumptions and understanding of their developers go – the real interfacing system may behave differently.

Resulting Context

You can continue developing your system without waiting for the development of the target system or depending on its availability.

This simulation mode of the interface can also help when troubleshooting a complex system of systems, making it easier to **Hunt the Lion in the Desert**‡[13], and may help "faking" the existence of a system that failed.

You need to verify any assumptions you make about the interfacing system with its developers, or you run the risk of testing nothing more than your perception of what your system will have to handle.

Developing **Mutual Simulators**§ is complimentary to this pattern. **Mutual Simulators** resolve this risk, since developing a simulation that is too elaborate may drain your development resources or make it too complex to maintain.

Known Uses

- The IP loopback interface – localhost or 127.0.0.1, allows sending requests to your machine. This provides an easy way to test network stack and networking software without any cables, and without generating any external traffic.
- Modems are required by ITU standards to support multiple levels of loopback, to allow for problem isolation, by sending a stream of test patterns, and checking if they are returned garbled.
- The standard RS-232 interface card provides loopbacks for testing by design.
- The SwitchMATE™ cellular network management system developed by Motorola included a tutorial mode that simulated the existence of a cellular network so its users could practice taking cellular network elements out of service. Performing such actions in a live system could cause disruption of service.
- Java In The Small (JITS[14]) uses a loopback mode that is built into its TCP-UDP/IP/SLIP communication stack objects.
- The picture at the top of this pattern shows a loopback cable for fiber optic connectors.
- Mock objects used in Test-Driven Design simulate an interacting object that is not available for testing.

---

‡ A pattern introduced by the author at EuroPLoP2003: A desert is a very big space and it is difficult to find the lion, so how do you hunt the lion in a desert? You draw a line splitting the desert space in two. The lion is either on one side of the line or the other. You cut the half space on the side the lion is again in half. Again the lion will be only on one side of this line. Thus relatively fast you get to a manageable space where it will be easy for you to find and hunt the lion. For more details, see[13].

§Pattern thumbnail: In **Mutual Simulators**, developers working on interfacing systems are responsible to providing the other team with a simulator of the system they are developing. This resolves the problems of having to make assumptions about the behavior of the other system, and provides both teams with early feedback on the system they are developing, helping them improve their system for the next iteration, as well as reducing integration overhead and schedule.

Acknowledgements

I would like to thank the customers and colleagues I have worked with designing products and systems – it is through their perspectives that I have learned.

Credit is also due to Ofra Homsky, for her continued inspiration, and for her feedback that helps these patterns evolve.

I thank my shepherd, Peter Sommerlad for his insights, perspective and feedback – his contributions has helped clarify these patterns.

References

1 Mostly Harmless, Douglas Adams
2 NASA Systems Engineering Handbook, SP-610S, 1995
3 INCOSE Systems engineering handbook, Version 2.07, 2002
4 Integrating Software Engineering and Systems Engineering, Barry Boehm, 1994 INCOSE Symposium
5 Twelve systems engineering roles, Sarah A. Sheard, 1996 INCOSE Symposium
6 USB "square" plugs, Henry Baker, Risks Digest 23.32, April 15, 2004
7 The radiological accident in Sorek 1993, International Atomic Energy Agency, 1993
8 http://www.magic-kinder.com/
9 Regulations leave collectors smuggling "Kinder", Barbara Carton, The Wall Street Journal, June 24, 2002
http://www.sfgate.com/cgi-bin/article.cgi?file=/news/archive/2002/06/24/financial0958EDT0035.DTL
10 TBD – Looking for a reference for this pattern.
11 Small Memory Systems, James Noble and Charles Weir
12 Pattern Mining, Jim Coplien, C++ Report, October 1995 and Fault Tolerant Telecommunication System Patterns, Coplien et. al., PLoP 95
13 Technical Support Patterns, Ofra Homsky and Amir Raveh, EuroPLoP 2003.
14 Java In The Small project Web site - Gilles Grimaud & Fabien Bouquelet aka Meshuggah, Université de Lille, France
http://www.lifl.fr/RD2P/JITS/jits/net/#LOOPBACK