An Interactive Pattern Story about Remote Object Invocation

James Siddle
Independent
www.jamessiddle.net
jim@jamessiddle.net

ABSTRACT
This paper presents an interactive pattern story about remote object invocation, applying lessons learned from previous efforts to write interactive pattern stories, and combining new story content with a revised format that allows the exploration of pattern based designs for learning. The paper presents further lessons learned from the process of writing the new story.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Patterns (e.g., client/server, pipeline, blackboard), and I.2.6 [**Learning**]: Concept learning

General Terms
Documentation, Design, Experimentation, Human Factors, Theory.

Keywords
Patterns, pattern stories, interactive fiction, design, learning.

1. INTRODUCTION
This paper presents an interactive pattern story about remote object invocation, and builds on ideas previously presented in *"Choose Your Own Architecture" - Interactive Pattern Storytelling* [5]. The paper, workshopped at EuroPLoP 2008, proposed that pattern stories can be made more educational and engaging by introducing interactivity such as that found in "Choose Your Own Adventure" [4] books.

Here, an interactive story about remote object invocation is presented. This story applies lessons learned from previous efforts to write interactive pattern stories, combining new story content with a revised format for allowing the exploration of pattern based designs for learning. The goal of this paper is to test this revised format and new content to understand the level of engagement and educational benefits of the form.

After describing the intended audience, a summary of the concept of interactive pattern stories is given, along with a description of the key features of the format that is being used in this paper. The main body of the paper is the interactive story itself, which is preceded by a guide for the reader. Several lessons learned are then presented, based on the experience of writing the story. The paper closes with a map of the interactive story and thumbnail descriptions of the patterns used in the stories, which can be found in the Appendices.

2. INTENDED AUDIENCE
Any software professional may learn something from the story presented in this paper, though early career professionals may find the story particularly useful. Pattern authors, practitioners, and theorists may also find the story of interest.

3. INTERACTIVE PATTERN STORIES
An interactive pattern story is a combination of a pattern story and interactive fiction.

A pattern story is a narrative that describes the application of patterns to solve a complex problem, for example in *"Pattern Oriented Software Architecture: Volume 5, On Patterns and Pattern Languages"* [3] the story of a request handling framework is told through patterns. A pattern story is not meant to be taken literally, rather it is a device used to communicate design through patterns.

Interactive fiction is a form of narrative typically told in second person, where the reader controls the direction of the story. Arguably the most well known examples of interactive fiction are children's "Choose Your Own Adventure" books, which tell adventure stories with many possible outcomes.

The benefit of presenting design narratives as a pattern stories is that they provide a *concrete example* of the application of patterns, in context. This grounds the design in reality, and provides an entry point for readers to understand both the pattern and the context in which it applies.

Interactive pattern stories, therefore, are design narratives where the reader controls the direction of the story. In addition to providing a concrete example of the application of patterns, interactive pattern stories allow the reader to explore concrete consequences of less optimal choices. They are considered to be an engaging and educational means of exploring design.

Alternative presentations of design choices could include interactive pattern sequences – ordered lists of patterns known to solve a given problem, but without the real-world scenario - or

simply pattern languages. The difference is the concrete, real-world scenario that grounds the design narrative and provides an entry point for the reader.

4. FORMAT

The story presented in this paper follows a particular format, based on lessons learned from previous attempts at writing interactive pattern stories. The key features of the format are:

Requirement fulfilment – the story is about fulfilling requirements, both functional and non-functional. These are presented at the start of the story and it is up to the reader to fulfil the requirements.

Tree structure with decision points – the story presented in this paper branches out from a single starting point (step 1). Different branches represent different decisions made by the reader in order to fulfil functional requirements.

Consequences in terms of non-functional requirements – the aim of the story presented here is to give the reader insight into the effect of their design choices on non-functional requirements. So each decision the reader makes has consequences. These consequences are described in terms of the non-functional requirements introduced to the reader in Step 1.

Multiple endings – the story has three endings; one good, one bad, one neutral. This configuration of endings is chosen so to distinguish a small number of endings as significant and to give the reader something to aim for in their reading.

Small selection of patterns – only four patterns are discussed in any detail, to keep the scope of the story manageable. The patterns are summarised in Appendix B. Note also that the patterns are typically used as the embodiment of good design practice, and are denoted using capitals throughout the text.

Supporting text with common narrative – previous attempts at creating interactive pattern stories have resulted in extensive duplication of prose, so common narrative sections are captured in a section at the end of the story, and are summarised in the main flow of the story to keep interactive story text concise.

Main path – the story presented here has a 'main path' where patterns are applied to solve the problems presented to the reader. The other paths are variations of that main path, introduced to explore alternative choices. The main path is presented first, from steps 1 to 6. A main 'negative' path – where the design choices are less than optimal - follows from step 7, followed by a number of other variations. To get the most from the story, the reader may wish to read the main path first, and then explore various alternative paths to explore the different decisions and consequences.

5. READER GUIDANCE

To read the story presented in this paper, simply start reading at step 1 and follow the instructions as they appear. At certain stages in the narrative you will be presented with decisions to make – simply choose a path and turn to the associated step. When you reach an ending, go back to the beginning and try a different route.

Try exploring all possible designs, including positive or negative design paths. Try to read from the beginning to the end, take time to reflect on the story, then go back to the beginning and try a different path to see how things might have turned out differently.

When you come across a pattern name such as TEMPLATE METHOD, remember that you can find a short 'thumbnail' summary and reference in Appendix B – Pattern Thumbnails. You may also be referred to supporting information, which can be found in a sub-section following the main body of the story.

You'll be presented with several requirements at step 1; note that these are kept simple for the sake of brevity.

The only other things you need to know are that the system is being developed in an Object Oriented language such as Java, and (as general back story) will be used for an online supermarket.

Finally, you might want to keep your eyes open for the occasional surprise along the way...

6. THE STORY

6.1 Step 1

You are developing a framework to handle the synchronous invocation of remote objects. You are faced with several requirements to fulfil, and must decide how to fulfil them. These requirements are summarised on cards in figures and described further below. Functional requirements are captured as user stories (Figure 1); non-functional requirements as 'ilities' [1] (Figure 2).

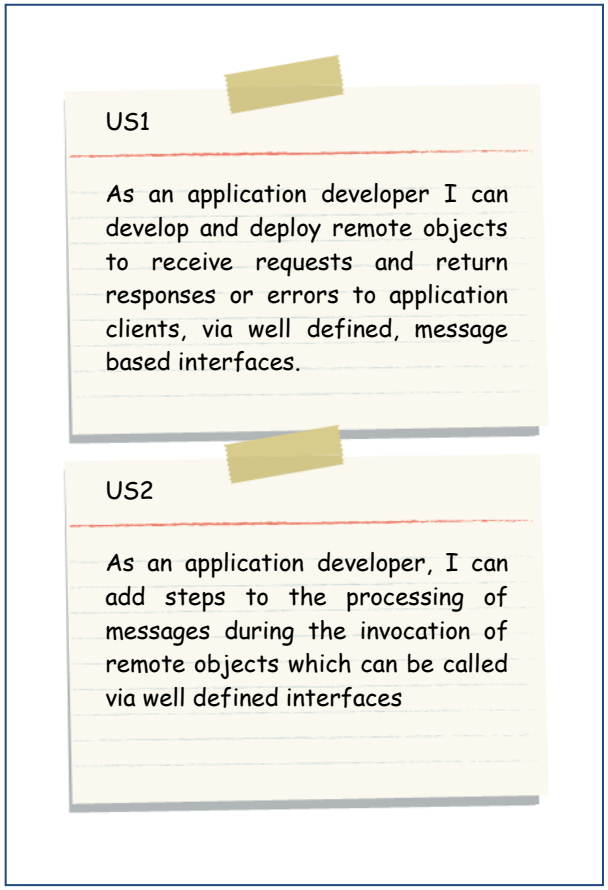

Figure 1 - Functional Requirements

First, your system must be able to receive and process messages for remote objects developed by application developers (US1). Your system must translate the messages into calls onto the remote objects, and then create response messages from any return values or output parameters. Exceptions may be encountered when calling remote objects, so error handling should also be provided.

Several kinds of remote object are required, starting with objects providing stock availability and customer loyalty information respectively.

Each object must be able to receive messages for one or more remote interfaces, which are described in a generic way (see INTERFACE DESCRIPTION pattern [7]). The operations and parameters of these interfaces will be well defined, however their network level representations may vary and new interfaces may be needed over time. Messages that target remote objects will be sent to discrete network endpoints.

Interfaces to support the aforementioned remote objects are expected – specifically for querying stock levels and customer loyalty information. Also several management interfaces are required for enabling and disabling logging, and for querying runtime statistics such as response times or number of queries made.

Your system must also be extensible, so that application developers can add new processing steps to remote object invocations (US2); this extends to introducing extra processing steps to pre-existing remote objects. Message encryption is an expected addition, because of the importance of ensuring customer confidentiality. Beyond functional requirements, flexibility, performance, and maintainability are the key non-functional requirements your system should fulfil.

- Flexibility is needed because remote object implementations may change or new remote interfaces may be added to accommodate new customer requirements, and network level representations of existing interfaces may change for example to support new technologies.
- Performance is a crucial consideration because the remote objects in your system may be called by third parties under service level agreements, so fast response times are desired.
- Maintainability is required because the system is seen as a long term investment so should be easy to maintain during its lifetime.

Figure 3 provides an overview of the baseline architecture of the required system.

Now continue at step 2, below…

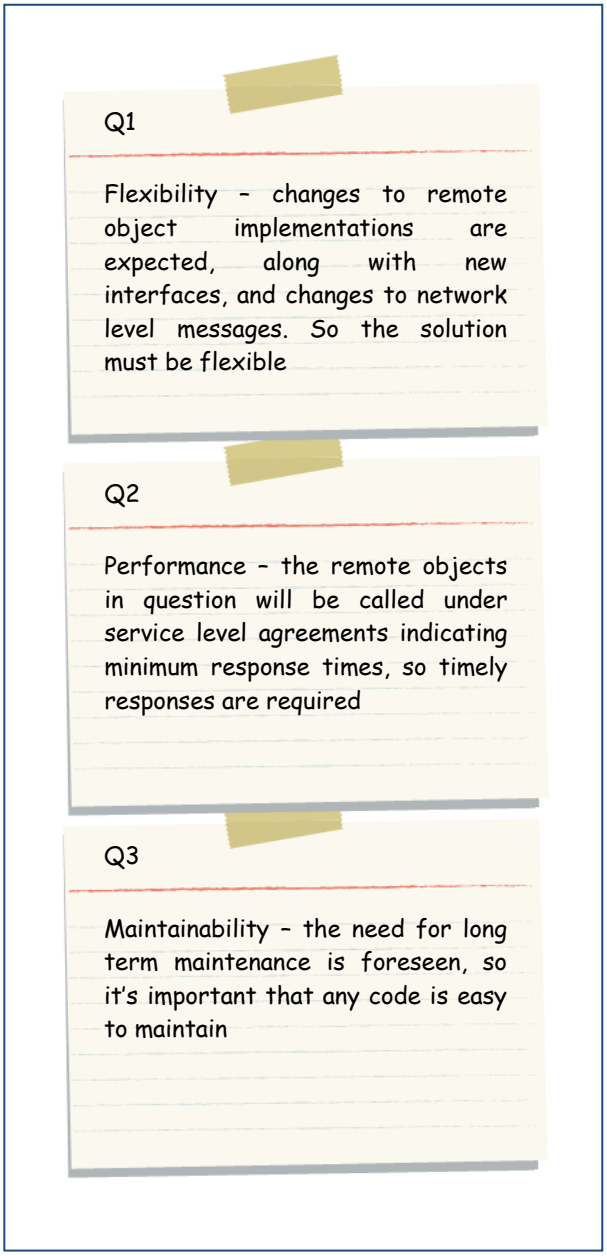

Figure 2 - Non-functional Requirements

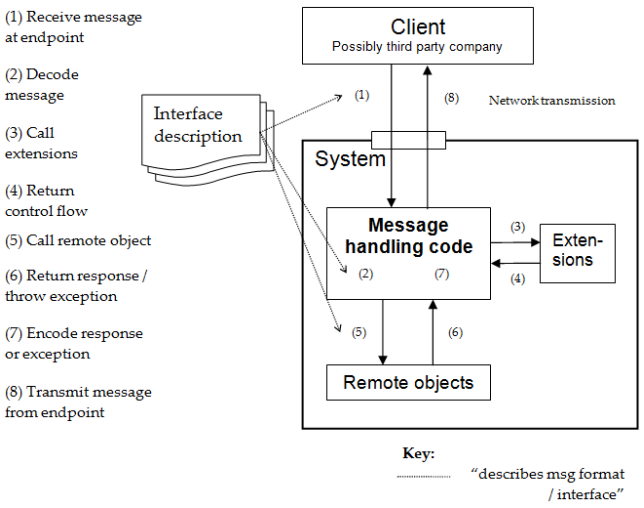

Figure 3 - Baseline Architecture

6.2 Step 2

The first decision you are faced with is how to map from remote interfaces and network messages to instances of objects in your system.

As a starting point, you expect that stock availability and loyalty card objects will provide interfaces providing access to stock availability and loyalty information, respectively. You also expect both objects to offer one or more management interfaces, starting with accessing runtime statistics.

Noting that each remote object has a unique identity, two alternatives come to mind.

First, you can give each remote object its own endpoint. Messages for each remote object will be sent to its unique endpoint, and each endpoint will be backed with code that ensures the associated remote object is called correctly.

Second, you can provide a single endpoint to receive messages for all remote objects in your system, and then introduce an additional layer of logic to dispatch messages to objects.

> *If you provide remote objects with their own endpoint, turn to Step 7.*
>
> *If you provide a single endpoint, turn to Step 3.*

6.3 Step 3

You decide to expose a single endpoint to receive all messages, and then have an additional layer of logic for dispatching to remote objects.

You create INVOKERs to encapsulate remote object dispatch functionality. INVOKERs are responsible for decoding messages, discovering remote objects to call, encoding responses, and handling errors. One INVOKER per remote interface seems about right to you. Upon receiving a message, your system discovers an invoker by partially decoding the message to discover which remote interface is targeted. Your system then looks up an appropriate INVOKER object from a local cache, then passes the message to the INVOKER to process.

Specifically, INVOKERs for stock availability, loyalty information, and runtime statistics are needed to support required interfaces.

On receiving a message, INVOKERs discover which remote object to call by further decoding the message to discover the remote object identity. INVOKERs in your system know how to handle messages for a particular interface, and retain a store of remote objects that implement the interface. Finding the remote object to call means searching the store of remote objects for one that has the object identity given in the message.

To call the remote object, the INVOKER performs further message decoding to discover operation and parameter values. After calling the remote object, the INVOKER encodes a response and returns it; your system then passes the message back to the network to work its way back to the calling client. Exceptions in remote objects are handled similarly - your INVOKERs create response messages that describe the error that occurred.

Figure 4 shows how your system behaves.

Your design has excellent flexibility and maintainability because message dispatching is well encapsulated. Application developers can easily add new message representations and interfaces, for example to give access to personnel databases or to enabling and disabling logging. Developers will find it easier to fix bugs because changes will be isolated to a small number of classes.

Your design also makes good use of network resources because only a single network endpoint is used. This improves performance of applications that include the remote objects by reducing the impact on memory. However you are concerned that it may become a bottleneck which could potentially impact service level agreements offered by applications.

Next, you must decide how to handle message encoding and decoding. The interfaces handled by your INVOKERs share common message elements and parameter types; these are represented in your system as non-primitive types. Both stock availability and loyalty information services, for example, represent product information in a common, multi-field packet format.

However each remote interface is unique, so you must determine how to encode and decode to and from these types for each interface.

> *If you create shared classes for encoding and decoding, turn to Step 4.*
>
> *If you inline encode/decode functionality in your INVOKERs, turn to Step 12.*

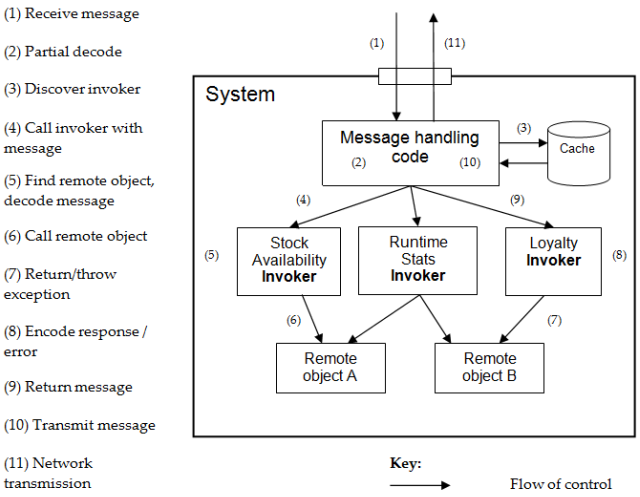

Figure 4 - INVOKER diagram

6.4 Step 4

You decide to encapsulate encoding and decoding of complex types into MARSHALLER classes.

MARSHALLER classes encapsulate the encoding and decoding of complex types such as the product information packet, and are called from your INVOKERs whenever a message is received or a response is ready to be transmitted. Using MARSHALLERs improves flexibility and maintainability, though there is a risk of an adverse effect on performance. See "Encoding and decoding with MARSHALLERs" for a detailed description and assessment of consequences.

You must now decide how to make your system extensible, so that extra processing - such as message encryption - can be introduced. You can introduce an extension interface to call from your INVOKERs, where objects implementing the interface provide additional processing steps; alternatively you can introduce abstract methods in your INVOKER classes so that sub classes can vary the processing that takes place.

> *If you add calls to an extension interface, turn to Step 5.*
>
> *If you add abstract methods for processing additional steps, turn to Step 15.*

6.5 Step 5

You decide to call an extension interface from your INVOKERS and MARSHALLERS, using the INTERCEPTOR pattern.

The INTERCEPTOR pattern introduces calls to an extension interface, via a dispatcher object. The interface is implemented by classes to allow additional message processing steps - starting with a class for encryption. This has an exceptional impact on the flexibility of your system, but an undetermined impact on maintainability and performance. See "Introducing INTERCEPTOR" for a detailed description and assessment of consequences.

You also notice that the introduction of calls to allow additional processing steps is simplified by the encapsulation of dispatching, encoding, and decoding into INVOKERs and MARSHALLERs; which further supports maintainability because the risk of code bloat is reduced.

> *Now turn to 6.*

6.6 Step 6

Congratulations, your remote object invocation system is complete.

You successfully implemented a mechanism for allowing application developers to develop and deploy remote objects accessible via multiple interfaces, and introduced an extensibility mechanism for allowing developers to introduce extra processing steps to remote object invocation.

Overall, you are pleased with your design. The flexibility is exceptional – application developers can introduce new interfaces, new network representations, and additional processing steps easily. Your system also cleanly encapsulates different concerns, so that maintenance of remote objects and additional processing steps will be simple and have little impact. You are a little concerned about the performance of applications that include remote objects deployed in your system, though you have some mitigation strategies should this be an issue.

Great job!

The End

6.7 Step 7

You decide to create distinct endpoints, one for each of remote object in your system.

You back your endpoints with classes for processing remote calls for the different types of object. Each of these message processing classes is associated with one type of remote object, and during initialisation is associated with a specific remote object. This ensures that messages will be always dispatched to the right object in your system. You provide encoding, decoding, and error handling functionality for each object type.

On receiving a message, each message handling class decodes the message to discover the operation and parameters for the remote call, and then passes the invocation on to the associated instance of the remote object. Following the call to the remote object, any return value, output parameters, or error message are encoded then returned to the calling client.

Figure 5 shows the behaviour of your remote object handling classes.

Your design ensures good response times for requests to remote objects because the number of processing layers between the network and the targeted remote object is limited. Unfortunately this is at the cost of excessive use of network resources - every remote object requires a distinct network endpoint which may be a serious limitation when scaling your system. So your design both supports and works against the delivery of service level agreements.

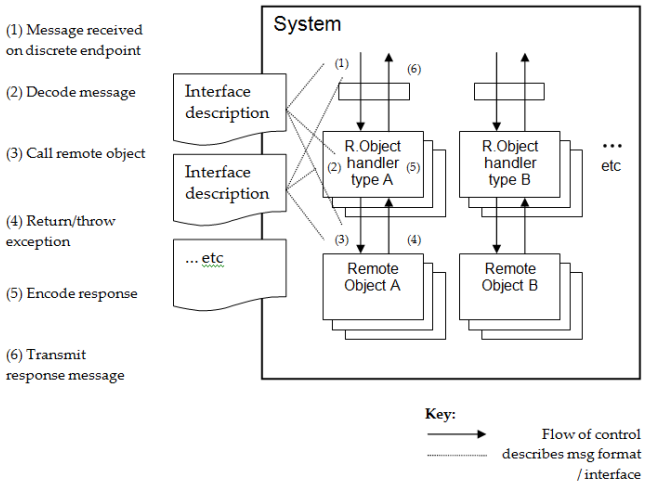

Figure 5 - Handler type and endpoint per remote object

Changes to network endpoints and addresses can be costly and fiddly on both client and server, so application developers may find your system inflexible. Further, the need for a distinct class for each remote object type may result in duplication of encoding, decoding, and error handling, so introducing new network representations or remote interfaces may be costly and thus off-putting to application developers. Such duplication similarly implies application developers will find maintenance more difficult because bugs may be repeated many times.

Now you need to determine how your remote object handlers will deal with encoding and decoding. The interfaces on which remote objects will be called share common message elements and parameter types, which are represented in your system as non-primitive types. However each remote interface is unique and each distinct remote object type may require specific processing. You must determine how to encode and decode to and from these types in your remote object handlers.

> *If you create shared classes for encoding and decoding, turn to 16.*
>
> *If you inline encoding and decoding in your remote object types, turn to 8.*

6.8 Step 8

You decide to create inline encoding and decoding in your remote object handlers. See "Inline encoding and decoding" for a detailed description.

Unfortunately application developers will find your design quite inflexible because the introduction of each new remote object type will require a complete set of new encode and decode functionality to be written. Similarly, variations in network representation and the introduction of new interfaces will require extensive code changes to remote object handlers, further impacting application developers.

However you think your system will perform well, supporting the delivery of service level agreements because messages received for remote object types need only be decoded to obtain the parameters required by the specific remote object type. Further, performance tweaks and enhancements can be applied specifically for each remote object type. Although impacted negatively overall, application developers may find the ability to tweak both network and internal representations of complex types for each remote object type quite useful.

You must now decide how to make your system extensible. You can introduce an extension interface to call, where objects implementing the interface provide additional processing steps; alternatively you can introduce abstract methods in your remote object handlers so that sub classes can vary the processing that takes place.

> *If you add calls to an extension interface, turn to 19*
>
> *If you add calls to abstract methods at key processing points, turn to 9*

6.9 Step 9

You decide to extend your remote object handlers to create TEMPLATE METHODS that call out to abstract methods which can be implemented by subclasses.

Using TEMPLATE METHOD for extensibility improves flexibility, though only statically, and not easily when more than one additional processing step is required. See "Extensibility with TEMPLATE METHOD?" for a detailed description and assessment of consequences.

Reviewing your extensibility mechanism, you discover a further negative consequence of your decisions. Where application developers wish to introduce additional processing steps, every remote object handler must be updated; this is compounded by the fact that there is no common code shared between remote object types – a result of in-lining your encode/decode functionality. Application developers will find further challenges with maintenance because of the many sub-classes introduced by TEMPLATE METHOD, and as with inline encoding and decoding, bug fixes to additional processing steps may need to be duplicated for each remote object type.

> *Now turn to 10*

6.10 Step 10

Congratulations, your remote object invocation system is complete.

Unfortunately, you think you may have made a mistake – application developers will find the flexibility and maintainability of your system poor.

Introducing new remote objects will be time consuming because of fiddly network configuration. The lack of any encode/decode encapsulation means that changes to network representations, new interfaces, or new remote objects all require extensive coding. Your TEMPLATE METHOD may allow some additional processing steps, but only statically.

The use of distinct remote object handlers and dedicated encode/decode functionality may mean bug fixes can be applied to each remote object, but maintenance overall will be hard because of duplication caused by the lack of any encode/decode cohesion, and because of the many layers of inheritance needed for additional processing steps.

At least the performance of the your system should be reasonable because of dedicated processing possible for each remote object; though thinking about it you realise that the poor flexibility and maintainability may soon render this benefit meaningless because application developers will avoid using your framework.

The End

6.11 Step 11

You step through the shimmering orange portal, and find yourself standing on a small, slowly moving platform[1].

You are surrounded by translucent walls, through which you see the occasional indiscernible shape. Below the platform you see only darkness.

The platform appears to be floating of its own accord, perhaps powered by the beam of light that inexplicably appears from around a nearby corner. The platform is travelling along the beam of light towards the corner, and you can only guess what lies there.

The portal you jumped through appears blue from this side, and is moving gradually away from you. Soon it will be beyond reach.

If you jump back through the portal, turn to 24

If you stay on the platform, turn to 21

6.12 Step 12

You decide to create inline encoding and decoding functionality in your INVOKER classes.

This means that each INVOKER is responsible for decoding messages that it receives, and encoding response messages before transmitting them to clients. Unfortunately, this impacts the maintainability of your system negatively - see "Inline encoding and decoding" for a detailed description and consequences. This design also makes it harder for application developers to maintain applications because the introduction of a new network representation will require the creation of an entirely new INVOKER. However you are consoled by the fact that developers can tweak the performance of each INVOKER because of the dedicated encode/decode functionality, supporting successful provision of service level agreements.

You must now decide how to make your system extensible. You can introduce an extension interface to call, where objects implementing the interface provide additional processing steps; alternatively you can introduce abstract methods in your INVOKER classes so that sub classes can vary the processing that takes place.

If you decide to add call outs to an extension interface, turn to 13

If you decide to add calls to abstract methods, turn to 14

6.13 Step 13

You decide to call an extension interface from your INVOKERS, using the INTERCEPTOR pattern.

The INTERCEPTOR pattern introduces calls to an extension interface, via a dispatcher object. The interface is implemented by objects to allow additional message processing steps. This will provide exceptional flexibility to application developers, but you're not sure how they will find maintainability and performance. See "Introducing INTERCEPTOR" for a detailed description and assessment of consequences.

The impact of inlining encode and decode functionality in your INVOKER also means that application developers must add new processing steps related to encoding and decoding to each INVOKER separately. This takes extra effort and further reduces the cohesiveness and thus the maintainability of the INVOKER.

Now turn to 20.

6.14 Step 14

You decide to update your INVOKERs with extension points by adding TEMPLATE METHODs.

Using TEMPLATE METHOD for extensibility improves flexibility for application developers, though only statically, and not easily when more than one additional processing step is required. See "Extensibility with TEMPLATE METHOD?" for a detailed description and assessment of consequences.

You recognise a further impact of inlining encode and decode functionality in your INVOKER – application developers must apply any additional processing steps related to encoding and decoding to each INVOKER separately. In addition to the extra effort, this further reduces the cohesiveness of the INVOKER, giving application developers a maintainability headache. Positively though, you see that the limited number of INVOKERs - one per remote interface - may have avoided the need for lots of new subclasses for remote object types where additional processing steps would have been required, meaning that application developers will find maintenance a little easier.

Now turn to 20

6.15 Step 15

You decide to update your INVOKERs and MARSHALLERs with extension points by adding TEMPLATE METHODs.

Using TEMPLATE METHOD for extensibility improves flexibility for application developers, though only statically, and not easily when more than one additional processing step is required. See "Extensibility with TEMPLATE METHOD" for a detailed description and assessment of consequences.

Reflecting on your design to this point, you see how the encapsulation of interface and encode/decode functionality of your INVOKERs and MARSHALLERs drastically reduced the number of changes needed to introduce extension points during development. You also see that the limited number of INVOKERs - one per remote interface - may have avoided the need for lots of new subclasses for remote object types, easing maintenance for application developers.

Now turn to 20

[1] If you're confused by this, see [6]

6.16 Step 16

You create MARSHALLER classes to perform encoding and decoding.

MARSHALLER classes encapsulate the encoding and decoding of complex types, and are called from your remote object handlers whenever a message is received or a response is ready to be transmitted. Using MARSHALLERs provides flexibility and maintainability to application developers, though there is a risk of an adverse effect on performance. See "Encoding and decoding with MARSHALLERs" for a detailed description and assessment of consequences.

You also realise that even though MARSHALLERs help to avoid duplication, application developers will still require some duplication of calls to the classes in remote object types because no other commonalities, for example at the interface level, have been encapsulated.

You must now decide how to make your system extensible. You can introduce an extension interface to call, where objects implementing the interface provide additional processing steps; alternatively you can introduce abstract methods in your remote object handlers so that sub classes can vary the processing that takes place.

If you add call outs to an extension interface, turn to 17

If you add calls to abstract methods, turn to 18

6.17 Step 17

You decide to call an extension interface from your remote object handlers and MARSHALLERs, using the INTERCEPTOR pattern.

The INTERCEPTOR pattern introduces calls to an extension interface, via a dispatcher object. The interface is implemented by classes to allow additional message processing steps. This will provide excellent flexibility to application developers, though you aren't sure how they will find maintainability and performance. See "Introducing INTERCEPTOR" for a detailed description and assessment of consequences.

In applying the pattern, you come to see the lack of flexibility caused by your decisions up to this point. Application developers must modify the handlers of each remote object type to integrate INTERCEPTOR calls. The duplication of code between remote object types will increase the time and effort required to introduce the extension points, and make the changes more error prone, so while the flexibility of your system does increase, it is costly. However your decision to encapsulate encoding and decoding functionality into MARSHALLERs means that extension points related to encoding and decoding can be easily introduced, so it's not all bad.

Now turn to 20

6.18 Step 18

You decide to update your remote object handlers and marshallers to add extension points via TEMPLATE METHODs.

Using TEMPLATE METHOD for extensibility provides flexibility to application developers, though only statically, and not easily when more than one additional processing step is required. See "Extensibility with TEMPLATE METHOD?" for a detailed description and assessment of consequences.

You can also see that the encapsulation of encode/decode functionality provided via MARSHALLERs is starting to reap benefits because some extension points are now shared between remote object handlers, so the introduction of additional processing steps by application developers requires fewer changes as a result. However application developers must sub-class remote object handlers and marshallers to introduce additional processing steps, and these will be difficult to maintain. But again, it would be even worse if you hadn't encapsulated encoding and decoding in MARSHALLERs.

Now turn to 20

6.19 Step 19

You decide to call an extension interface from your remote object handlers, using the INTERCEPTOR pattern.

The INTERCEPTOR pattern introduces calls to an extension interface, via a dispatcher object. The interface is implemented by classes to allow additional message processing steps. This provides great flexibility to application developers, though you aren't sure about the impact on application maintainability and performance. See "Introducing INTERCEPTOR" for a detailed description and assessment of consequences.

In applying the pattern, you come to see the lack of flexibility caused by your decisions up to this point. Application developers must modify handlers for each remote object type to integrate INTERCEPTOR calls. This lack of maintainability is further compounded by the duplication of encoding and decoding code, because there is no commonality at all in remote object handling. The time and effort involved in making the changes, along with the large amount of duplication involved, means that the changes are also error prone, and will make applications using the framework even less maintainable. So while application developers will benefit from improved flexibility, they will now suffer from poor maintainability.

Now turn to 20

6.20 Step 20

Congratulations, your remote object handling system is complete!

You are unsure whether to be happy or unhappy with your design. It has both good points and bad points. Application developers will find some flexibility, some maintainability, some good performance characteristics. However you think there may have been a more optimal design; a better balancing of the tradeoffs, and you wonder what that might look like.

The End

6.21 Step 21

The platform proceeds slowly towards the corner[2].

As you approach, you start to see a familiar orange flicker on the walls. You get an odd feeling in the pit of your stomach.

[2] If you're confused by this, see [6].

Rounding the corner, you realise with horror that the platform is heading straight towards a flame-filled room. Turning back, you start to move towards the portal, but it's too late - it is beyond reach.

You look back towards the flame-filled room. All you can do is watch as your fate approaches.

The End

6.22 Supporting Text

The following sections contain sections of text referred to in the main body of the story.

6.22.1 Encoding and decoding with MARSHALLERs

"Each MARSHALLER is responsible for translating from a byte stream to a particular complex type, and vice versa. MARSHALLERs for higher level complex types make use of MARSHALLERs for lower level types. Primitive types are encoded directly into byte streams.

The concrete byte stream representation that the MARSHALLER decodes to and from is determined by the remote interfaces that the type is required for; in your system XML is the chosen representation.

You introduce MARSHALLERs classes for the common packet structures that appear in interfaces; one example being a MARSHALLER that encodes to and from the common product information format. You update your classes to use the MARSHALLERs, and where common complex types have the same network representation in multiple interfaces, your classes can use the same MARSHALLER.

Your design allows application developers to flexibly introduce and vary message encoding and decoding, important because of the variations in network representation and interface that are foreseen. You can imagine that new interfaces for handling market data will be introduced in the future, where use of the standard product information format will be necessary. That said, you do wonder if your design will cope with minor variations in network representations for common complex types – the required representation of product information may differ subtly between different services; careful use of inheritance (such as through TEMPLATE METHOD) may be enough to solve this problem.

Long term maintenance of encoding and decoding by application developers should be fairly straightforward because of the clean encapsulation introduced.

You're concerned about the potential impact on application performance however; the additional layer of encoding and decoding means that - similarly to variations in network level representation - performance enhancements for particular interfaces may be difficult to introduce.

Figure 6 provides an example of the 'uses' relationships between marshallers."

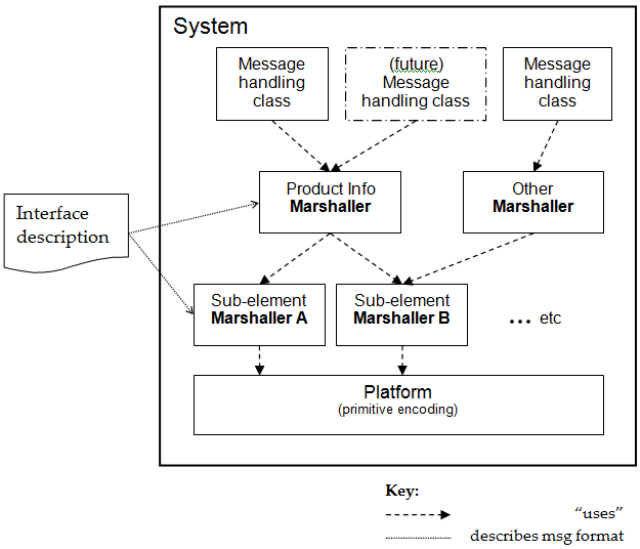

Figure 6 - MARSHALLER diagram

6.22.2 Introducing INTERCEPTOR

"At each point where additional processing steps are needed, you call a dispatcher to invoke any registered INTERCEPTORs that may modify or augment message processing. Each INTERCEPTOR implements an abstract interface that the dispatcher calls for discrete processing steps, and receives a context object - created by your message handling classes - that allows queries and modifications to message processing state.

This means that additional processing steps can be easily added to your system. Message content encryption – important for customer confidentiality, for example, can be added by introducing an INTERCEPTOR that is executed upon message receipt, and before transmitting any response. This INTERCEPTOR decrypts messages before your message handling classes process them further, and encrypts response messages before they are returned to the framework.

The flexibility of this design is exceptional - application developers can introduce additional processing steps without modifying existing code. The INTERCEPTOR pattern even allows the dynamic introduction of processing steps at runtime. Additional processing steps for auditing, message logging and tracing, and possibly even reliability may be introduced as they are needed for different remote objects in the supermarket system. These additions may even be made at runtime, where service level agreements exist.

Application developers will find that maintenance of their additional steps is both helped and hindered - helped because additional processing steps are cleanly encapsulated and loosely coupled with the rest of the system, hindered because INTERCEPTOR's complexity and abstract nature make it difficult to understand. You are also concerned that application performance may not be optimal because the runtime characteristics of the system will depend on which INTERCEPTORs are configured, though the fact that this is entirely configurable alleviates your concerns to a degree.

Figure 7 shows the interactions between message handling classes and interceptors.

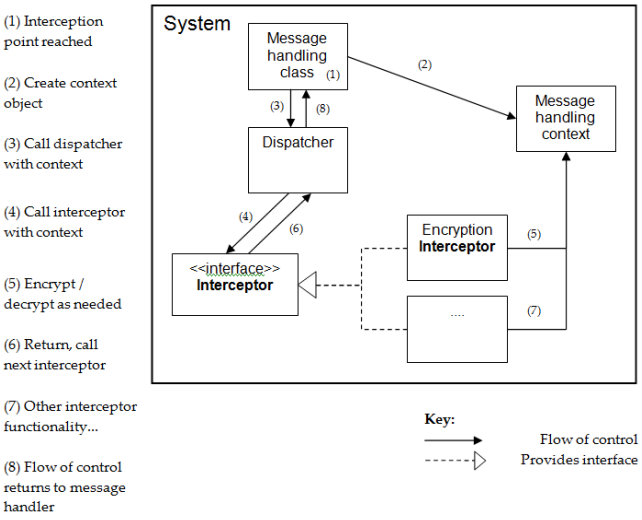

Figure 7 - INTERCEPTOR diagram

6.22.3 Inline encoding and decoding

You ensure that every message handling class explicitly encodes from and decodes to the complex types required by the target remote object. Byte stream representations, as received in incoming messages, are decoded into instances of complex types in a form that is suitable for the remote object being called. Return values and output parameters are transformed directly into byte streams specific to the interface that the object was called on. Similarly, exceptions are caught and encoded into byte streams representing the error, which can be returned to the client.

Figure 8 shows two message handling classes with inline encoding and decoding.

Sadly, this will make maintenance difficult for application developers because the duplication of encode and decode functionality means duplication of bugs, which will make fixes costly. The distribution of encoding and decoding code will also make it harder to identify the root cause of bugs, partly because of poor cohesion, but also because of code bloat.

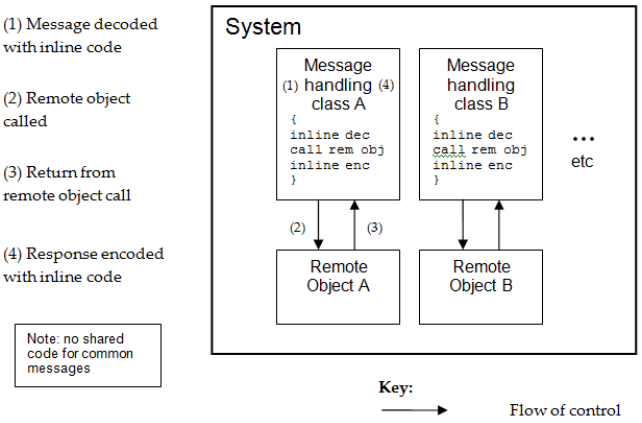

Figure 8 - Inline encoding and decoding

6.22.4 Extensibility with TEMPLATE METHOD?

You identify the points during message processing at which additional processing steps may be required. For now, you decide to add two extension points - one upon message receipt and one prior to response message transmission. You apply the TEMPLATE METHOD pattern, changing your message processing classes into abstract types, and then introducing calls to abstract methods at suitable points in your code to allow extensions to be introduced in sub classes. You introduce a subclass with empty implementations of the abstract methods to be default, non-extended implementation. Extensions can be introduced by sub-classing the abstract class and introducing new processing steps in the abstract methods.

Figure 9 shows an example message handling class that provides template methods for extension points.

The TEMPLATE METHODs provide flexibility to application developers because new message processing steps can be introduced. However there are two major limitations to this. First, adding more than one processing step to a particular extension point will be difficult, and will result in confusing code because of multiple layers of inheritance. Second, the flexibility is static rather than dynamic - the introduction of additional processing steps will require a recompilation and redeployment of your system. Application developers will find some help with maintenance, because bug fixes to additional processing steps will be isolated to sub-classes, though multiple levels of inheritance counters this to some extent.

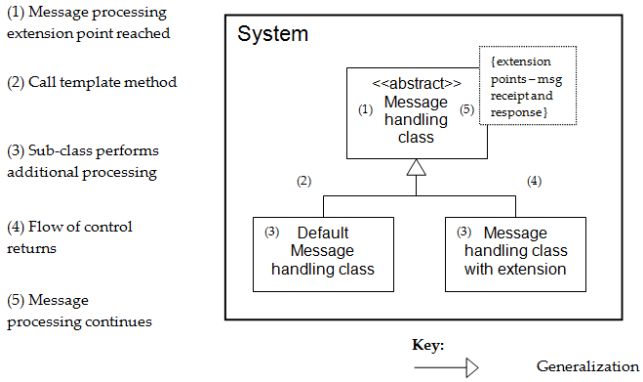

Figure 9 - TEMPLATE METHOD diagram

7. LESSONS LEARNED

The interactive story presented above is an experiment performed to help identify an effective format for interactive pattern stories. Whilst writing this paper, a number of lessons learned (over and above those described in [5]) were discovered:

Interactive pattern stories with branches may have many repeating sections because similar decisions with similar consequences appear across different branches. This may be mitigated to some degree by the use of a section containing common narrative fragments.

Different styles of diagrams suit different narratives – previous stories used class diagrams and code fragments, but the interaction-like diagrams used here are considered to be a good match to the interactive nature of network invocation.

Introducing an order into the presentation of story steps may help readers to understand the different branches that appear[3]; here the 'good' steps were provided, followed by the 'bad' (or less optimal), followed by neutral steps.

Relating design decisions and consequences to concrete user stories or use cases, stated in terms of actual users, ground the narrative in reality and force the writer to make observations or comments in a way that readers are more likely to relate to.

8. SUMMARY

This paper presented a new interactive pattern story about remote object invocation, combining lessons learned from previous writing efforts. A new, refined format was used to present the story, which used the INVOKER, MARSHALLER, INTERCEPTOR, and TEMPLATE METHOD patterns – and several less optimal alternatives - to explore the design of remote object handling systems. Several lessons learned were presented, based on the experience of writing this paper.

9. ACKNOWLEDGMENTS

Thank you to Michael Stal for many useful and insightful comments during the shepherding of this paper for PLoP 2009. Thanks also to the 'Architecture and Design' Workshop group at PLoP 2009 for the many helpful comments.

10. APPENDICES

11. Appendix A – Interactive Story Map

Figure 10 - Interactive Story Map" (which appears at the end of the paper) provides a full map of the routes through the interactive story. The main 'good' path that appears first is highlighted using empty boxes, the neutral paths in grey boxes, and the main 'bad' path in black filled boxes.

12. Appendix B – Pattern Thumbnails

The following pattern thumbnails are based on several publications, including "Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing" [2], and "Remoting Patterns : Foundations of Enterprise, Internet and Realtime Distributed Object Middleware" [7]. Key phrases that summarise the patterns are included below as quotations. Please see the quoted publications for full descriptions.

12.1.1 INTERCEPTOR pattern

"It can be hard to anticipate how the behaviour of a framework may need to be tailored for different environments or applications. Features and attributes of an otherwise stable core set of services may need adaptation or extension. Allow users to tailor a software framework by registering out-of-band service extensions via predefined callback interfaces, known as 'interceptors', then let the framework trigger these extensions automatically when specific events occur." [2]

12.1.2 INVOKER pattern

"When a client sends invocation data across the machine boundary to the server side, the targeted remote object has to be reached somehow. The simplest solution is let every remote object be addressed over the network directly. But this solution does not work for large numbers of remote objects...Provide an INVOKER that accepts client invocations from REQUESTORS. REQUESTORs send objects across the network, containing the ID of the remote object, operation name, operation parameters, as well as additional contextual information. The INVOKER...dispatches the invocation with demarshaled invocation parameters to the targeted remote object." [7]

12.1.3 MARSHALLER pattern

"For remote invocations to work, invocation information has to be transported over the network...Only byte streams are suitable as a data format for transporting this information over the network. Require each non-primitive type used within remote object invocations to be serializable into a transport format that can be transported over a network as a byte stream. Use compatible MARSHALLERS on the client and server side that serialize invocation information." [7]

12.1.4 TEMPLATE METHOD pattern

"Where an object has a common core, but may vary in some behavioural aspects, create a superclass that expresses the common behavioural core then delegate execution of behavioural variants to hook methods that are overridden by subclasses." [2]

13. REFERENCES

[1] Bass, L., Clements, P., Kazman, R., Software Architecture in Practice, 2nd Edition, Addison Wesley, 2003.

[2] Buschmann, F., Henney, K., Schmidt, D.C., Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, John Wiley and Sons, 2007

[3] Buschmann, F., Henney, K., Schmidt, D.C., Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages, John Wiley and Sons, 2007

[4] Packard, E., Choose Your Own Adventure 1: The Cave of Time, Bantam Books, 1979.

[5] Siddle, J., Choose Your Own Architecture - Interactive Pattern Storytelling, EuroPLoP conference proceedings, 2008.

[6] Valve, Portal the video game, http://en.wikipedia.org/wiki/Portal_(video_game), 24th October 2009.

[7] Voelter, M., Kircher, M., Zdun, U., Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware, Wiley Software Patterns Series, 2005

[3] Thank you to my PLoP 2009 shepherd, Michael Stal for this useful suggestion.

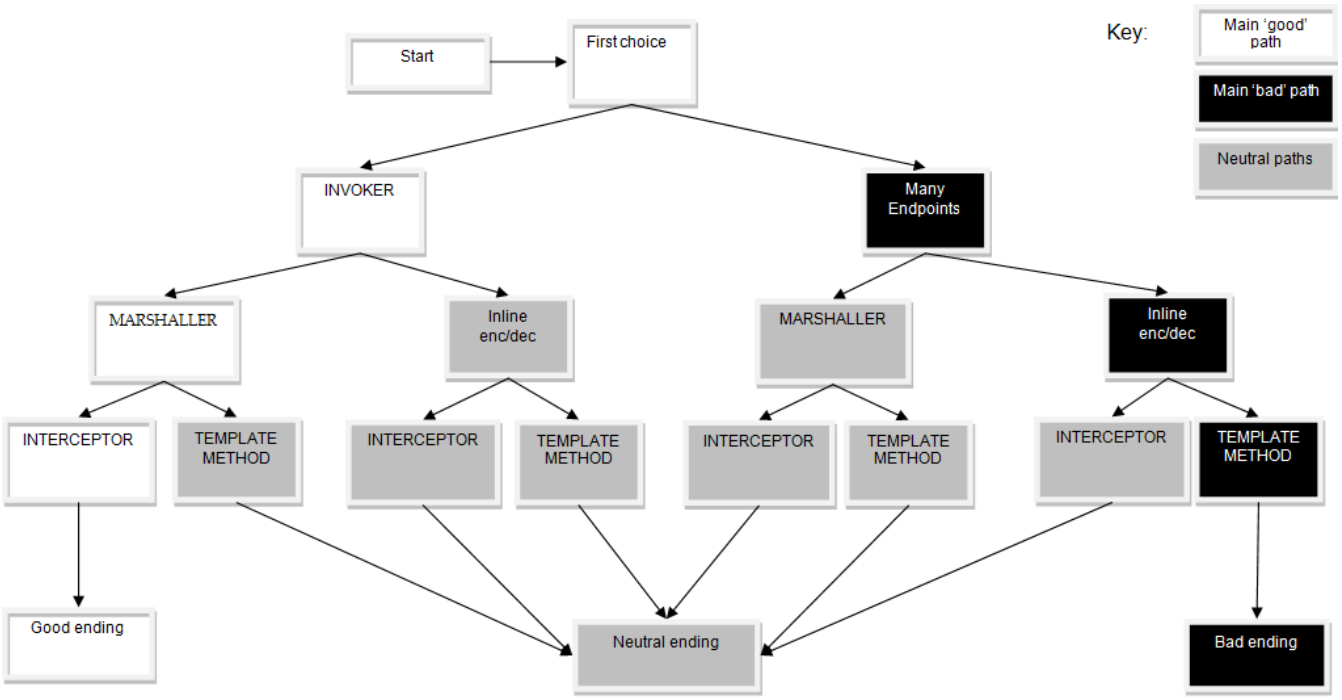

Figure 10 - Interactive Story Map