An Example of the Retrospective Patterns-Based Documentation of a Software System

James Siddle

jim@jamessiddle.net
http://www.jamessiddle.net

Abstract. An example pattern-based documentation that was created retrospectively from pattern applications on an industrial project is presented. In addition to the example documentation, the paper examines the approach taken, divergence of the documentation from the real system, benefits, liabilities, and applicability of the approach. The paper closes by drawing conclusions from the experience of creating the documentation.

Key words: Patterns, software patterns, pattern stories, software system, software design, software architecture, retrospective pattern-based documentation

1 Introduction

This paper examines the retrospective documentation of a concrete software system through patterns, by presenting an example patterns-based documentation based on a particular project and drawing conclusions from the experience of creating the documentation.

The motivation for creating the pattern-based documentation that appears in this paper is to retrospectively capture and communicate the historical intent behind, and contribution of, pattern applications to software development. The documentation in this paper serves as both an example, and as the basis of analysis and conclusions that follow. In practice, such documentation could be expected to support interested parties such as software maintainers in understanding software.

The documentation approach taken captures the contributions that individual patterns made to the concrete software system, in a stepwise fashion. Discrete contributions are captured in individual steps of the documentation, and the entire collection of these contributions make up the complete documentation.

The remainder of the paper is organised as follows: First, the intended audience and key terms are introduced, then a description of the software development project where patterns were applied or recognised retrospectively is provided. The documentation in this paper attempts to capture and communicate the contributions of pattern applications on the project described. This

is followed by a description of the documentation approach, an overview of the documentation that follows, and then the complete patterns-based documentation. An analysis examines differences between the documentation and the real system, along with potential benefits, liabilities, and applicability of the approach. The paper closes by drawing a number of conclusions related to creating patterns-based documentation.

1.1 Intended audience

The ideal reader of this paper is a software practitioner - whether programmer, developer, engineer, or architect with an interest in how software patterns can be employed to capture and communicate the concrete software system that results from their application.

The reader may also be interested in how patterns can be combined, because while applying a pattern in isolation is useful, a complex software system is likely to require the application of more than one pattern. The reader may also be familiar with the idea of architecture patterns and want to know how to fit patterns together into an overall architecture.

1.2 Terminology

To frame the following discussion, it's necessary to introduce the following concepts:

Patterns and Pattern Stories: A pattern describes a recurring solution to a problem that occurs in a particular context by resolving the forces acting in that context. The reader is referred to [8] and [11], arguably the best known patterns works, for an introduction to and examples of patterns.

Another patterns-related concept mentioned in this paper that needs a little introduction is that of a pattern story [9]: A pattern story describes a concrete software system in terms of the patterns used to create it.

Software Architecture: There are many definitions of software architecture in software design literature, Grady Booch's recent definition is particularly suited to understanding the application of software patterns to the creation of software architecture:

> As a noun, design is the named (although sometimes unnameable) structure or behavior of a system whose presence resolves or contributes to the resolution of a force or forces on that system. A design thus represents one point in a potential decision space. A design may be singular (representing a leaf decision) or it may be collective (representing a set of other decisions). [...]
>
> All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change. [3]

Note that cost of change was not necessarily the most significant factor understood by the team developing the software under discussion, who it is thought did not share a common understanding of architecture.

2 Project Context

The software system documented in this paper originated on a project where patterns were applied to create a component middleware software architecture; this project is briefly introduced below. For reasons of confidentiality, the following description has been anonymized.

2.1 Project Introduction

The aim of the project under consideration was to develop the software for an innovative telephony product using C, C++ and Java programming languages, and it was necessary for the software to run on a custom hardware platform that was being developed at the same time. Scrum [17] and XP [4] Agile methodologies were followed on the project.

In addition to functional requirements from the telephony domain, there were also non-functional requirements on the software. In particular, a custom, service-oriented, embedded middleware was required in order to support a product line strategy that was being taken. The key requirements on the middleware were:

- Support for reusable, telephony-domain services
- Dynamic deployment of services
- Platform independence
- Abstraction of service communication
- Location transparent service communication
- Abstracted service execution
- A common approach to management and testing of services
- An extensibility mechanism in the communication path between services

The middleware was also required to support specific services from the telephony domain that had been envisaged as part of the product-line strategy, such as distributed directory lookup services, "presence" propagation services, and journal services to record user actions.

The middleware was developed by a team of eight people over a period of approximately six months, as part of a wider development effort. Early project iterations focussed on elaborating middleware, platform, and application-level architecture. Patterns played an important role in the design and implementation that took place as part of middleware elaboration. The elaboration was driven by requirements (such as those described above) that were drawn from an architecture road map containing a loosely ordered collection of broadly stated architectural concerns.

Figure 1, reproduced from [18], provides an overview of the middleware architecture that was envisaged at the start of the project.

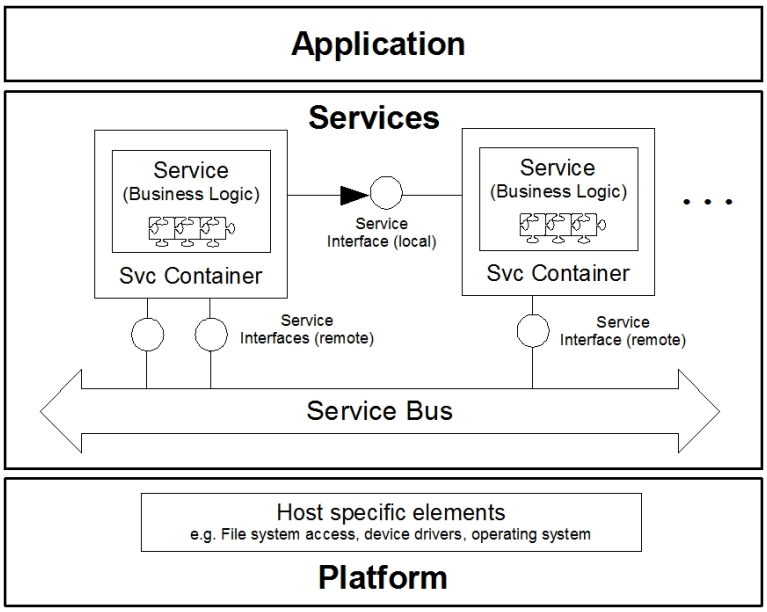

Fig. 1. Envisaged middleware architecture

2.2 Pattern Applications

A broad selection of patterns were applied in creating the middleware. These were drawn from several sources. Several were drawn from the *Pattern Oriented Software Architecture* [11] [12] [13] and *Design Patterns* [8] books, while others were recommended by a knowledgeable source rather than being drawn from a patterns publication. For a fuller picture of the patterns that were applied on the project, the reader is referred to [18], which provides a broad overview of the patterns applied on the project.

3 Documentation Approach

The documentation presented in the following section was created by capturing the structure and behaviour associated with patterns that were either applied or retrospectively recognised in the software system. Four aspects of the approach taken are examined below - purpose, focus, form, and pattern selection.

3.1 Purpose

The patterns-based documentation presented in this paper was created as an example of how patterns can support the communication of software system knowledge between software professionals. Each pattern is included in order to

communicate concepts, structure, and behaviour resulting from pattern applications, informing readers of the important elements of a software system and ultimately supporting the creation and maintenance of quality software.

Section 6.2 below provides further examination of the potential benefits of a patterns-based approach to software documentation.

3.2 Documentation Focus

The focus of the documentation is historical, in order to show the evolution of the middleware up to a specific point in it's elaboration. This is to help the reader to understand the early evolution of the software to support their understanding of why the system is the way it is today. The software development that took place following architecture elaboration followed the course that many projects do, where the clean separation and guidelines associated with the elaborated architecture gradually dissolved and were discarded. Thus, the focus of the documentation is on the original intent of the software, which is what is presented here.

The documentation is also focussed on the structure and behaviour of the software system, specifically the classes and interfaces that resulted from pattern application and the roles and responsibilities taken. This focus was chosen because the aim was to enable readers to quickly orientate themselves with the system, both in it's current form (when subsequent changes have not been made), and with the system as originally intended (where changes have been made).

The documentation does not focus on presenting the software architecture, though the documentation may be architecturally relevant because architectural significance was one criteria used to select patterns for inclusion (see below). The emphasis on structure and behaviour is selected because an extended period of time (approximately 2 years) has passed since the the software was developed, and a focus on concrete software elements is likely to be more accurate than a focus on architecture rationale.

Finally the focus is high-level, (i.e. class and interface), rather than low-level (i.e. method, function). The aim being to present an overall view of the software system rather than to dive into details.

3.3 Documentation Form

A simple form was selected to communicate the software system by way of patterns.

A section is included for each pattern in the documentation, the problem encountered and solution selected are described, and then concrete structure and behaviour associated with the pattern is presented.

Note that section 5.7 actually captures the contribution of two patterns. This slight variation of the form allows for a cleaner presentation of the software system because each pattern's contribution was relatively small, and the two patterns are closely related.

Each section contains:

- Title - pattern name and reference
- Problem and Solution
- System Structure and Behaviour
- Class Diagram

The patterns are presented in an order to allow an understanding of the software system to gradually build up in the readers mind. This is partially possible because the order selected is similar to the order in which patterns were applied and implemented during development, on an iteration by iteration basis. This means that certain software elements (e.g. the 'service' layer) are introduced by one pattern, before being referred to by the documentation associated with following patterns.

Additionally, certain later steps describe refinements to earlier steps. Mostly this is because refinements took place as described, however in some places this approach allows for a simpler and clearer presentation of the software system.

3.4 Patterns Selected, Selection criteria

Patterns were selected for inclusion primarily according to architectural significance, in order to provide an understanding of significant parts of the middleware to the reader. The judgement of the significance of each pattern was made in a subjective manner, according to the understanding of the software system that is held by the author as the architect of the middleware. So while the documentation itself is not architectural in nature, it can be seen as architecturally relevant. This criteria is considered to be valid because the aim of the documentation is to provide an historical introduction to the software rather than to serve as the basis of formal analysis or comprehensive education.

The type of pattern application was another consideration for pattern selection; that is whether a pattern was explicitly applied, implicitly applied, or retrospectively recognised. An examination of selection criteria was performed following the documentation creation, an overview of which can be found the first appendix. This examination suggests that:

- Architectural significance was the main criteria for inclusion.
- Implicitly applied patterns were excluded on the grounds of being obvious to developers (e.g. Caching).
- Some architecturally significant elements were retrospectively recognised as patterns in order to allow their documentation here (e.g. Broker).
- One pattern was excluded because the resulting implementation was not effective at solving the problem (Asynchronous Completion Token). [1]

The pattern story mentioned in [18] provides a detailed description of the patterns that served as the selection pool for the documentation, and the complete selection pool is listed in the appendices. The patterns selected for inclusion in the documentation can be found in table 1 below.

[1] This is thought to be because of a naive understanding of patterns (i.e. the model solution is the pattern), and subsequently a poor selection of context in which to reapply the pattern (i.e. at the service interface rather than middleware level).

4 Documentation Overview

To serve as an introduction to the complete documentation that follows, a documentation overview can be found in Table 1. The table summarises the pattern applications that took place to create the middleware architecture. The three columns in the table represent:

- *Step* - a discrete numbered step in the documentation. The steps correspond to subsections in the documentation.
- *Pattern name* - the name of the pattern (or patterns) that are presented at each step
- *Contribution* - a summary of the contribution of the pattern(s) to the software system

Step	**Pattern**	**Contribution**
1	Layers	Encapsulate major functional areas to enable reuse and independent variation
2	Wrapper Facade	Encapsulate low-level, host-specific functions and data structures
3	Component Configurator	Enable dynamic service deployment and runtime life-cycle management of services
4	Broker	Establish service communication and location transparency
5	Executor	Abstract service execution, support concurrent service execution
6	Explicit Interface	Add explicitly defined service interfaces
7	Encapsulated Context Object	Introduce service discovery context object, make available to services
..	Decoupled Context Interface	Decouple services from context implementation by introducing service discovery interface
8	Proxy	Add client-side object implementing explicit service interface, encapsulates remote communication
9	Invoker	Add server-side object, receives service invocations and invokes explicit service interface; encapsulates service interface invocation
10	Lookup	Provide service discovery mechanism
11	Interceptor	Introduce flexible interception points on communication path between services

Table 1. Documentation overview

5 Pattern-based Software System Documentation

This section presents the documentation of the software system that was developed, in the form described in section 3.

5.1 Layers [11]

Problem and Solution: There is a requirement to create a product line architecture where major building blocks can be reused, but this won't be possible without clean and careful dependency management between the building blocks.

"Application", "Service", and "Platform" LAYERS are introduced. These layers establish high level groupings for software elements that will be introduced later, and introduce some basic concepts such as "Service" and "Platform".

System Structure and Behaviour: Each layer may only depend on those below it in the stack; in this case the Service layer is non-strict in recognition that Application layer code may need to invoke Platform layer functionality. The Platform layer however is strict in order to enforce platform independence.

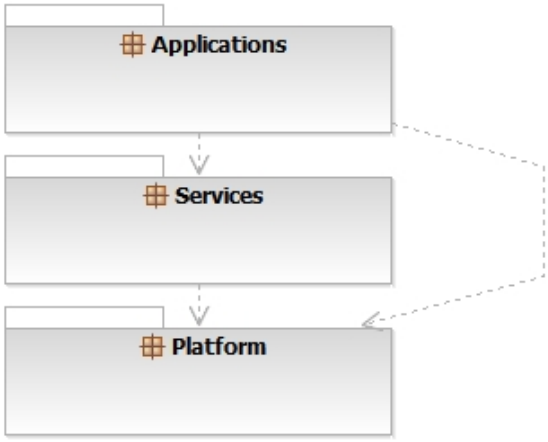

Fig. 2. LAYERS diagram

5.2 Wrapper Facade [12]

Problem and Solution: Service and Application layer code should be platform independent, but how do you achieve this without littering the code with conditional compilation?

Wrapper Facade classes are introduced into the Platform layer, to encapsulate low-level host specific functions and data structures. These classes provide a set of abstractions which in conjunction with the strict Platform layer provide platform independence.

System Structure and Behaviour: `FileAccess` provides access to the file system, `LibraryLoader` provides library management and symbol resolution, `InterProcessCommunication` allows communication between processes and `Threading` supports starting, stopping, and synchronising threads.

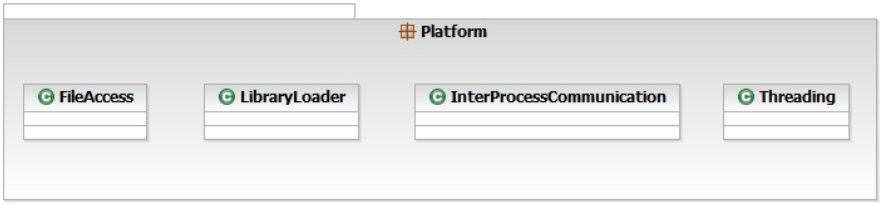

Fig. 3. Wrapper Facade diagram

5.3 Component Configurator [12]

Problem and Solution: Dynamic service deployment and life-cycle management is needed, but how can this be achieved in a consistent, common way, and so that services have to perform as little self-management as possible?

Component Configurator is applied so that service deployment and life-cycle are consistently managed via a common interface, along with "creator functions" from each service's library, and a component descriptor file.

System Structure and Behaviour: `ConcreteComponent1` is created and initialised via a "creator function" in the same library as the component. The `Component-Configurator` class loads the library via the `LibraryLoader` class and calls the creator function within it based on information contained in a component descriptor configuration file. `ConcreteComponent1` implements the `Component` interface, allowing consistent management of components by the `Component-Configurator` and `ComponentRepository` classes. `FileAccess` and `Library-Loader` are Wrapper Facade classes introduced above; the other classes are introduced by Component Configurator. [2]

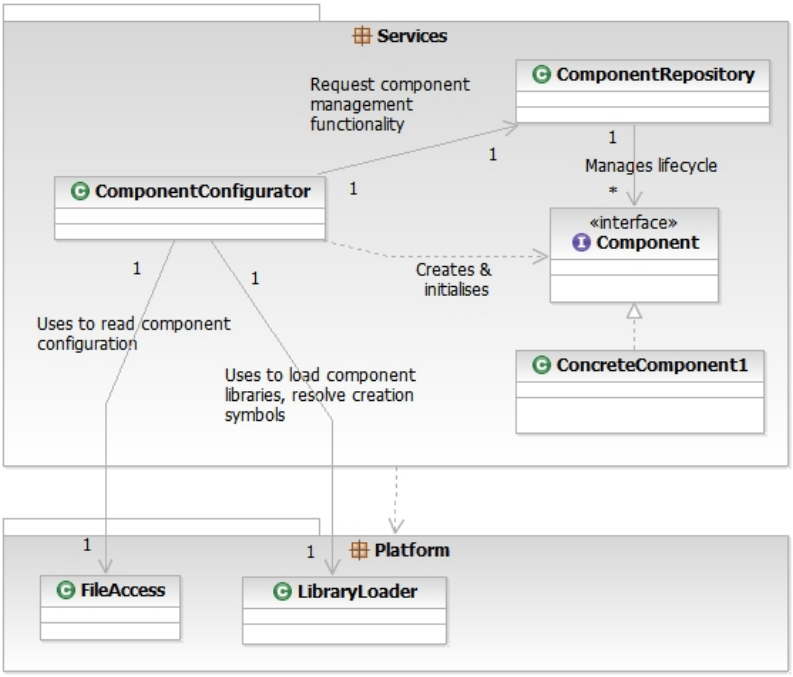

Fig. 4. Component Configurator diagram

[2] From this point on, services are taken to be realised as `Component` objects.

5.4 Broker [11]

Problem and Solution: Flexible service deployment is needed, but if services have direct, hard-wired communication paths to other services the system will not be flexible, so how do you achieve location transparent communication between services?

The BROKER pattern is applied so that services communicate indirectly via an instance of a `Broker` class in a well known location.

System Structure and Behaviour: The `ComponentConfigurator` class creates and initialises an instance of the BROKER `CommunicationChannel` class for each `Component`, then passes it to the `Component` during initialisation so that it can send and receive messages. The `CommunicationChannel` is associated with the `Component` in the `ComponentRepository` to ensure that it is cleaned up correctly. `CommunicationChannel` instances communicate with each other in a location transparent way, by sending and receiving all messages via an instance of the `Broker` class in a well-known location. Low level communication takes place between the `Broker` and `CommunicationChannel` classes via the `InterProcessCommunication` WRAPPER FACADE class.

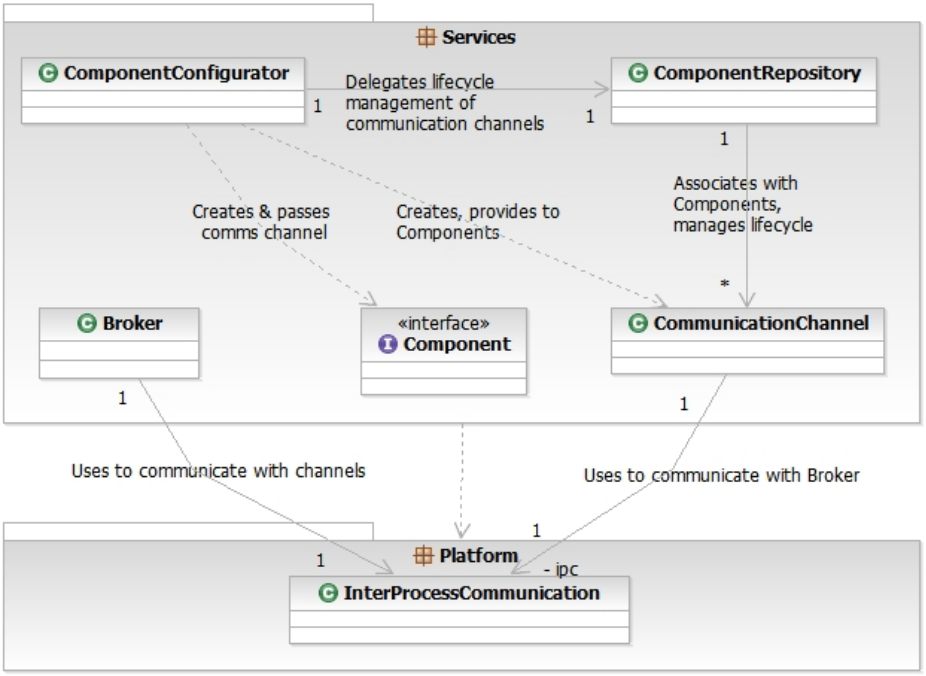

Fig. 5. BROKER diagram

5.5 Executor [6]

Problem and Solution: Services need to be executed when they receive a message, but how can service execution be handled in a consistent way?

EXECUTOR is applied to introduce an `Executor` class which is responsible for handling execution for all services.

System Structure and Behaviour: The `Executor` waits for messages to arrive over a service's `CommunicationChannel` object and for each message that arrives an appropriate thread of execution for message processing is established; the associated `Component` is informed of the message on the resulting thread. The `ComponentConfigurator` class is refined to associate an `Executor` instance with each `Component`, and the `Executor` associated with each `Component` is initialised with the appropriate `CommunicationChannel` object.

The `Executor` uses it's knowledge of the incoming message and `Component` instance to determine a threading policy such as single or multi-threading, or priority. The thread itself is controlled via a WRAPPER FACADE class.

Because `Executor`s are now interacting with `CommunicationChannel` objects on each `Component`s behalf, `CommunicationChannel` objects are no longer passed to `Component`s.

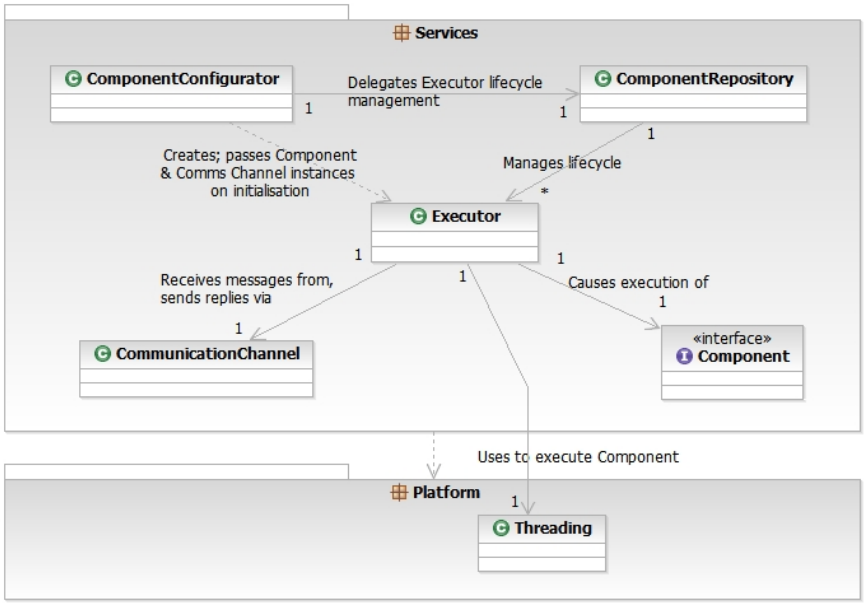

Fig. 6. EXECUTOR diagram

5.6 Explicit Interface [14]

Problem and Solution: Services can send and receive messages to invoke other services, but this tightly couples service invocation with message formatting and transmission; how do you decouple service invocation from component communication or internals?

EXPLICIT INTERFACE is applied to provide a way of calling services via abstract, implementation agnostic interfaces.

System Structure and Behaviour: The `ExplicitInterface` and `ConcreteExplicitInterface` interfaces together provide the implementation of the pattern. `Component`s implement interfaces according to the abstractly defined services that they offer, and call objects that implement the interfaces which define the services they require.

The `ExplicitInterface` class is introduced to allow consistent handling of objects which expose defined services. The explicit interfaces that define services are referred to as 'service interfaces' from this point onwards.

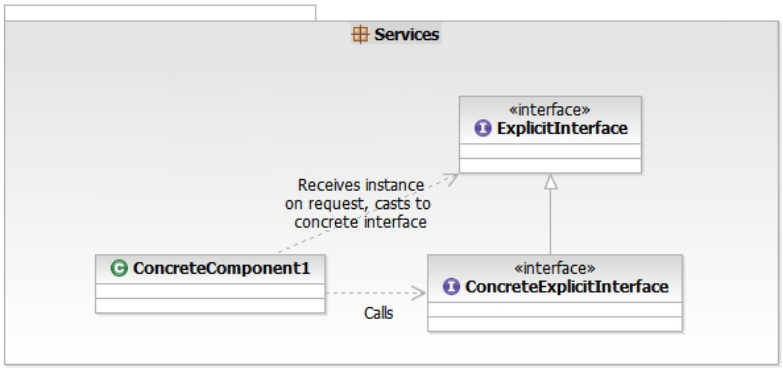

Fig. 7. EXPLICIT INTERFACE diagram

5.7 Encapsulated Context Object, Decoupled Context Interface [9]

Problem and Solution: Services need to discover other services, but how can the services be shielded from potentially complex discovery logic?

An Encapsulated Context Object with a Decoupled Context Interface is introduced to provide a service discovery object that services can call as necessary.

System Structure and Behaviour: The `ComponentConfigurator` class provides the implementation of the Encapsulated Context Object pattern, while the `ServiceContext` interface is the realisation of the Decoupled Context Interface pattern.

`Component`s call the `ServiceContext` interface (which is provided to them on initialisation) to request objects that implement particular services; objects that implement the requested services are returned to the `Component` as instances of `ExplicitInterface`. `Component`s cast returned `ExplicitInterface` instances to specific service interfaces to be able to request required services.

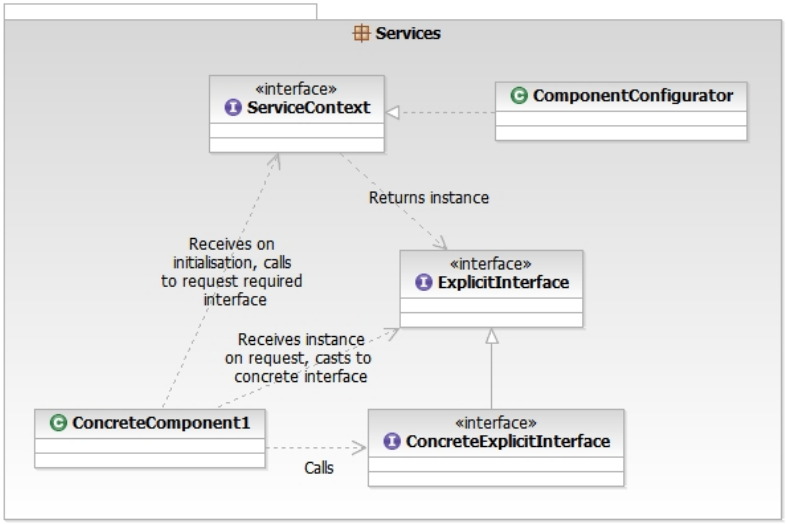

Fig. 8. Encapsulated Context Object and Decoupled Context Interface diagram

5.8 Proxy [8]

Problem and Solution: Services can request other services and obtain Explicit Interfaces to them, but how are service invocations translated into messages then sent via the location transparent Broker?

A Proxy is introduced to encapsulate the remote invocation of service interfaces via the location transparent communication provided by the Broker implementation.

System Structure and Behaviour: `Proxy` objects are provided to `Component`s by the `ComponentConfigurator` on service discovery. The `Proxy` is initialised with the `CommunicationChannel` object of the requesting service, along with addressing information of the remote service that the `Proxy` represents. `Proxy` life-cycle is managed by the `ComponentConfigurator`, in a similar way to `Component` and `CommunicationChannel`s.

`ConcreteProxy` encodes the request (including methods, parameters, and reply information) in a format suitable for transmission, then uses a `CommunicationChannel` to send the request. Replies are received via the same channel, then decoded and returned to the calling `Component` by the `Proxy`.

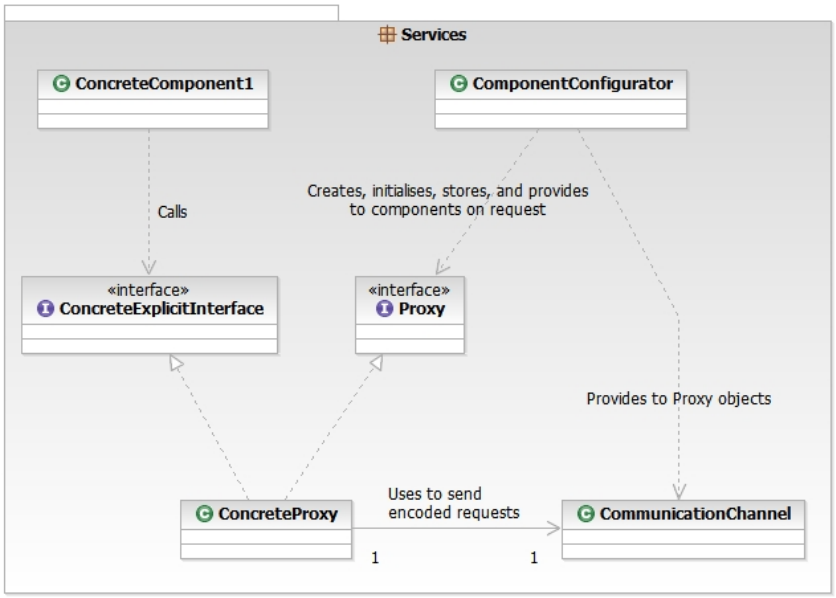

Fig. 9. Proxy diagram

5.9 Invoker [19]

Problem and Solution: Services support explicit service interfaces, but how are messages received from another service via the BROKER translated into invocations on the service itself?

An INVOKER is introduced to encapsulate the receipt and translation of messages into an invocation on a particular service interface.

System Structure and Behaviour: We now see how remote service invocations from `Proxy` objects are handled when they arrive in the locality of the `Component` that provides the remote implementation.

When an `Executor` receives a message for its `Component` it discovers an appropriate `Invoker`. This discovery will be based on the required service interface (named in the message) and will be requested via the `FrameworkContext` instance (another ENCAPSULATED CONTEXT OBJECT with a DECOUPLED CONTEXT INTERFACE). The `Executor` will have received the `FrameworkContext` during initialisation.

The `Executor` delegates invocation of its `Component` to the discovered `Invoker`, which decodes the received message to determine the method to invoke and parameters to pass. As with `Proxy` objects, `Invoker` object lifecycle is managed by the `ComponentConfigurator`, in a similar way to `Component` and `CommunicationChannel`s. After invocation, any return value or parameters are encoded by the `Invoker` into a reply message, which will be sent to the originating communication channel by the `Executor`. To ensure correct routing of replies, the `Proxy` will have associated a unique identifier with the outgoing request message, and requests that it's `CommunicationChannel` passes incoming messages with that identifier to the `Proxy`.

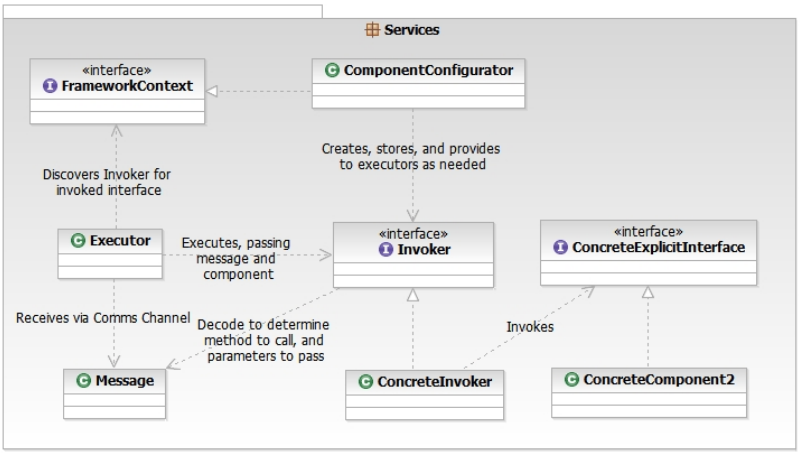

Fig. 10. INVOKER diagram

5.10 Lookup [13]

Problem and Solution: Remote services can now be invoked via explicit service interfaces, but how is service discovery performed?

The introduction of LOOKUP provides resolution of objects that provide required services.

System Structure and Behaviour: A remote `Registry` is introduced that can be consulted to discover the named `CommunicationChannel` location of implementations of particular service interfaces; the `ComponentRepository` is also searched in case required services are provided locally.

A `Component` requests, via the `ServiceContext`, an object that provides a required service. The `ComponentConfigurator` class, in implementing the `ServiceContext` interface, must provide an object back to the requester.

The `ComponentConfigurator` searches the local `ComponentRepository` for any local (i.e. in-process) `Component`s that support the interface. If found, the `Component` is returned directly to the requester. Otherwise, the remote `Registry` is searched, and if a remote object supporting the interface is found, location information is obtained. The location information is used to initialise a `Proxy` which is then returned to the requester.

The remote `Registry` is initialised with remote object interface and location information by `ComponentConfigurator` instances during Component initialisation.

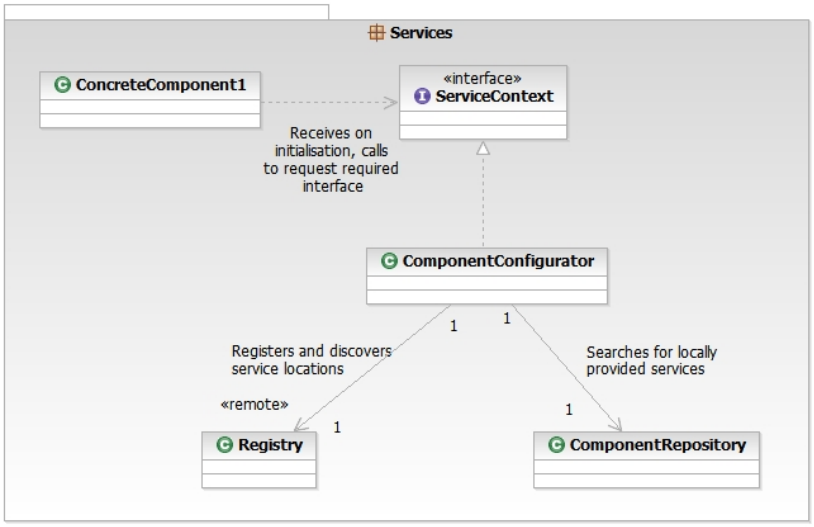

Fig. 11. LOOKUP diagram

5.11 Interceptor [12]

Problem and Solution: An interception point is needed on the communication path between services, but how can this be provided dynamically, without requiring code changes to the communication path?

The INTERCEPTOR pattern provides a flexible, dynamic interception point with the minimum disruption to code. The following text along with the class diagram in figure 12 describe an interception point immediately prior to service execution, when a message is received.

System Structure and Behaviour: When an `Executor` object receives a message for its service, it creates an instance of the `ExecutionInterceptionContext` class and initialises it with the received message. The `Executor` informs an instance of the `ExecutionInterceptionDispatcher` class of the event, passing it the interception context object as a parameter. The dispatcher object is responsible for maintaining a list of interested interceptors, each of which implements the `ExecutionInterceptor` interface. The dispatcher informs the interceptors of the event, passing on the context object it received. The interceptors can examine the message via the context object. The can also interact with the context object to perform any interception activities they wish to, such as redirecting or blocking the message, or performing security checks of statistic gathering.

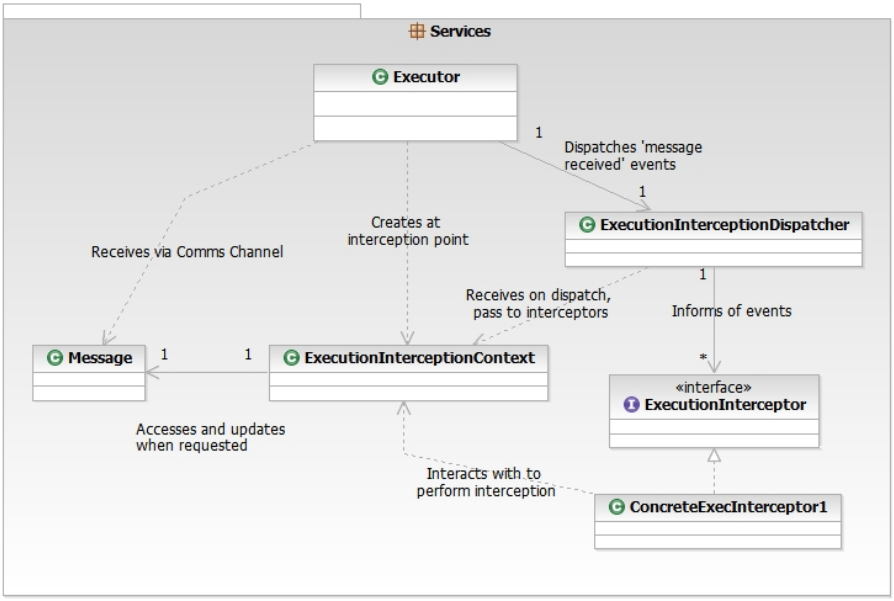

Fig. 12. INTERCEPTOR diagram

6 Analysis

Below, an analysis of the patterns-based documentation is described by examining how the documentation differs from the actual software system, what the benefits and liabilities of the approach are, and where the approach may be applied.

6.1 Differences between Documentation and Software System

The software system as presented in the previous section differs from the actual software system that was developed in two ways.

Firstly, the documentation presents an historical view of the system at a point when the project was transitioning from architecture elaboration to full-scale production, and differs from the final software that was produced, tested, fixed, and ultimately entered maintenance.

Secondly, the system described differs from the actual system in order to more clearly communicate the intent of the original design to the reader. The documentation also presents simplifications and implementation reorderings to educate readers on significant aspects of the design which may be confused by the complexities of the real system.

The differences between the real system and the system as presented are described in the first appendix, and are also discussed in the conclusions section below.

6.2 Benefits and Liabilities

Benefits The main benefit of using patterns as the basis of software documentation is to enable readers to quickly gain a working knowledge of a particular software system, to support the creation and maintenance of quality software. As Alistair Cockburn says in [5], when discussing the application of a "Theory Building View" [10] [16] to software documentation:

> What should you put into the documentation? That which helps the next programmer build an adequate theory of the program. [...] The purpose of the documentation is to jog memories in the reader, set up relevant pathways of thought and experiences and metaphors. [...]
>
> Experienced designers often start their documentation with just: The metaphors; Text describing the purpose of each major component; Drawings of the major interactions between the major components.
>
> These three items alone take the next team a long way to constructing a useful theory of the design.

In this case, the patterns provide the metaphors, and for each pattern there is a description and a diagram describing purposes and interactions of software elements. It is considered that this approach provides just enough documentation to enable the reader to quickly establish the "theory of the design", based

on an understanding of patterns. For readers knowledgeable about patterns, a documentation overview similar to that presented in table 1 may be enough to gain a useful understanding of the software system.

It is also thought that the software system can be better understood by describing many small steps that are similar to the actual evolution of the software system. Each step describes a problem encountered, the pattern used to solve the problem, and the resulting implementation and effect on the evolving software system. This is similar to the tea garden example in [2], where the application of many patterns contribute to an understanding of the architecture as a whole.

The documentation also gradually introduces important entities and concepts from the problem domain, and gives a grounding in the Ubiquitous Language [7] of the project as understood by the writer. Understanding an existing software system via pattern based documentation can help readers to understand the project's underlying problem domain and associated language.

Another important benefit is that the documentation presented is significantly shorter than would normally be found, the consequences of this being that it will be more easily understood and also that it is more likely to be written in the first place.

By including patterns which are considered to be architecturally significant, the documentation can also serve as a kind of architecture introduction, drawing the reader's attention to elements of the software system which are of greater significance than others. Additionally, the patterns not only explain the architecture that was chosen, but also the rationale behind that architecture to some degree.

Additionally, the historical focus of the documentation may provide insight into why a system is the way it is. Due to evolution and maintenance, class names or relationships that no longer make sense may be explained to the reader, however simplifications or intentional differences from the real system may undermine this.

Finally the documentation can provide examples of pattern applications in the context in which they were made, which may be useful for patterns education or research. However the usefulness of the examples will depend on their accuracy - in this paper the documentation does not describe the created system exactly, so the examples should not be taken literally.

Liabilities The main liability of the approach is that the documentation is not an exact representation of the software system. The benefit of being able to quickly build a mental model of the historical evolution of a software system may outweigh this liability. However it is important to present an honest view and to include the differences from the real system in some form, and to explain the purpose for the variations, as shown in the second appendix.

Secondly, this approach may not be applicable for design decisions made that are not motivated by a pattern. This may be the case where the design is so unique that no pattern exists.

The documentation may also be hard to understand for readers who have not been exposed to the patterns, and who may find the documentation confusing. However the provision of references for each pattern may alleviate this, and well-named patterns may help the reader to grasp the purpose of the pattern without further reference.

The fact that low level details of the patterns and the software system are not included may be problematic to readers who wish to understand more about them. By increasing the level of detail, more useful information can be included, but at the cost of a succinct presentation.

Finally, the reader may have a different understanding of the patterns used, resulting in misinterpretation. This may be exacerbated by the fact that multiple, occasionally conflicting versions of patterns can sometimes be found. A description of how the pattern contributed to the software system should help to bridge the gap in the reader's understanding, though this does require the reader to be open to the interpretation of the patterns used.

6.3 Applicability

The documentation approach outlined in this paper may prove applicable when creating software using ad-hoc pattern application, i.e. where patterns are applied as the opportunity arises. This approach was taken on the project under discussion, so naturally should be considered as a candidate for applying the documentation approach.

An Agile approach was taken on the originating project, where patterns were selected and applied in an iterative way. As such the documentation partially reflects the evolution of the software, showing intermediate steps in it's growth. The documentation approach may therefore prove useful where patterns are applied during architecture evolution.

The approach may prove useful when a pattern language based approach to pattern application is taken, where the transition from one pattern to the next is guided by the context resulting from each particular pattern application. The closer match between the initial and resulting context of patterns applied from a pattern language may lead to documentation that is easier to understand if the approach outlined in this paper is taken.

Patterns-based documentation of a software system may also be useful to teaching patterns. In a similar way to pattern stories, the concrete focus and practicality of the documentation may support students in applying patterns by showing them the real-world consequences of pattern applications. Exposing students to several examples of patterns-based documentation with variations of a particular pattern may also help them to understand how patterns provide design guidance, rather than rubber-stamp solutions

7 Conclusions

The following conclusions relate to the process of creating pattern-based documentation, and were drawn by reflecting on the experience of creating the documentation presented.

The main conclusion that can be drawn is that it is important to avoid varying the documentation from the real system if possible. If variations are needed for example to simplify the presentation or to communicate historical intent, then such variations should be called out when they are made along with an indication of why they are being made.

Variations from patterns should also be documented to ensure the reader does not assume the design as described is synonymous with the documented pattern. A potential improvement on the approach would be reorient the documentation primarily around problems faced in design, then to document how each pattern contributed to the design.

Each pattern application should ideally be documented in an ongoing way during design and development. A template document with fields for pattern-based elements such as context, problem, pattern selected, solution implemented, contribution of pattern, variation from actual system etc. may prove useful in such a scenario.

The documentation should also include an indication of whether a pattern was explicitly applied, implicitly applied, or retrospectively recognised to avoid confusion around design intent.

Finally care should be taken during documentation to ensure problem related fields really do describe problems rather than requirements for solutions. The latter are useful, but should be motivated with the problem that resulted in the requirement in the first place.

Acknowledgements

Thanks to Kevlin Henney for suggesting that I write this paper and for feedback and support during it's development, to James Coplien for providing helpful comments and feedback on an early draft of the paper, to James Noble for shepherding the paper for EuroPLoP 2007 and offering many useful comments. Thanks also to workshop group 'Gray' at EuroPLoP 2007, and to the patient reviewers for "Transactions on Pattern Languages of Programming" for providing extensive and in-depth feedback which has improved this paper substantially. Finally, thank you to David Illsley from IBM for helpful review comments, and to IBM generally for supporting the publication of this paper.

References

1. C. Alexander, S. Ishikawa, M. Silverstein, et al: *A Pattern Language*. Oxford University Press, 1997
2. C. Alexander, *The Nature of Order Book 2: The Process of Creating Life*. The Center for Environmental Structure, 2002

3. G. Booch, *Handbook of Software Architecture Blog*, March 2nd, 2006, *On Design* `http://booch.com/architecture/blog.jsp?archive=2006-03.html`
4. K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999
5. A. Cockburn, *Agile Software Development: The Cooperative Game*, 2nd Edition. Pearson Education / Addison Wesley, 1997
6. E. Crahen, Executor. *Decoupling tasks from execution*. VikingPLoP 2002
7. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995
9. K. Henney, *Context Encapsulation. Three Stories, a Language, and Some Sequences*. EuroPLoP 2005. urlhttp://www.two-sdg.demon.co.uk/curbralan/papers.html
10. P. Naur, *Programming as Theory Building*, in *Computing: A Human Activity*. ACM Press, pp. 37-48, 1992
11. F.Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1 - A System of Patterns*. John Wiley and Sons, 1996
12. D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2 - Patterns for Concurrent and Distributed Objects*. John Wiley and Sons, 2000
13. M. Kircher and P. Jain, *Pattern-Oriented Software Architecture Volume 3 - Patterns for Resource Management*. John Wiley and Sons, 2004
14. F.Buschmann, K. Henney, D.C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. John Wiley and Sons, 2007
15. F.Buschmann, K. Henney, D.C. Schmidt, *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. John Wiley and Sons, 2007
16. G. Ryle, *The Concept of Mind*. Harmondsworth, England, Penguin (1963). First published 1949.
17. Schwaber, Ken and Beedle, Mike, *Agile Software Development with SCRUM*. Prentice Hall, Upper Saddle River, NJ, 2001
18. J. Siddle, *Using Patterns to Create a Service-Oriented Component Middleware*, VikingPLoP, 2006. `http://jms-home.mysite.orange.co.uk/docs/patternspaper.pdf`
19. M. Voelter, M. Kircher, U. Zdun , *Remoting Patterns : Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. John Wiley and Sons, 2005.

Appendix: Pattern Selection Pool

Table 2 presents the patterns that were considered for inclusion in the documentation, along with selection criteria analysis performed after documentation creation.

Pattern	**Sig.**	**Exp.**	**Imp.**	**Retr.**
Layers	1	X		
Wrapper Facade	1	X		
Component Configurator	1	X		
Broker	1			X
Executor	1	X		
Explicit Interface	1	X		
Encapsulated Context Object	1	X		
Decoupled Context Interface	1	X		
Proxy	1	X		
Invoker	1	X		
Lookup	1			X
Client-Server	1		X	
Interceptor	2	X		
Lazy Aquizition	2		X	
Pooling	2		X	
Caching	2		X	
Asynchronous Completion Token	2	X		
Observer	3	X		
Template Method	3	X		
Singleton	3	X		

Table 2. Patterns and selection criteria

The patterns in the selection pool were categorized by:

- *Architectural Significance* (Sig.) a mark from one to three, indicating the relative contribution the pattern was considered to have made to the architecture.
- *Explicitly applied* (Exp.) - patterns that were consciously chosen for application during development.
- *Implicitly Applied* (Imp.) pattern that were not consciously applied, but were thought to have been applied by developers because the design captured by the pattern is well known; for example caching.
- *Retrospectively Recognised* (Retr.) indicates that the pattern was not consciously applied, but was recognised in the software system in retrospect.

Appendix: Documentation Differences

The differences between the real system and the system as presented are described below.

Implementation differences

The Wrapper Facade classes shown at step 2 represent the intended rather achieved level of platform independence. Classes emerged over the course of the project, requiring (but not always receiving) rework to ensure architecture conformance. The `Thread` implementation was more complex than shown, and included classes that were closely coupled with service execution infrastructure rather than being general purpose Wrapper Facades.

The collaboration between Broker and Component Configurator classes at step 4 was not taken explicitly on the project; services actually created their own channels for communication until Executor was applied. This is shown as a separate step because it is useful for understanding the software system, i.e. that each service is associated with a communication channel.

The initial implementation of Executor was simpler than that described at step 5. The threading policy was actually introduced later when it was needed.

Proxy objects in the real system were actually associated with their own unique communication channel and shared between services because of concerns over excessive numbers of Proxy objects being created. In hindsight, it was considered better that all messages associated with a particular service should flow over one communication channel, and that the risk of excessive memory consumption from large numbers of proxies was low. As such, the system design was revised in the documentation to associate Proxy objects with service communication channels.

The inheritance hierarchy around `Component`s, `Proxy`, and `ExplicitInterface` classes has been simplified. Certain areas of the real software system had a complex and difficult to understand class hierarchy, with some unnecessary relationships which caused implementation challenges.

The Interceptor shown has been simplified from the actual implementation; on the originating project, the interception point was actually within the misplaced threading wrapper mentioned above.

Some name changes were made - for example `ExplicitInterface` was retrospectively renamed from `Service` in the real system.

Ordering differences

Component Configurator and Broker were applied concurrently and independently, they are presented in order because the extra step introduced by separating the patterns allows for a cleaner presentation.

Additionally, the ordering of steps 6-10 of the documentation has been introduced retrospectively - the patterns shown at these steps were applied in a much more disorderly way compared to the rest of the patterns, which were applied in the order presented (except for the two patterns mentioned in the previous point which were applied simultaneously).