Creating Software Architecture using Pattern Sequences

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

Note to the workshop: Because this is a long paper, please read the the first 16 pages (up to the end of section 4). If you have time to read more, it would be useful to read the next 8 or so pages (section 5). Thanks.

1 *Introduction*

This paper examines the approach of using pattern sequences to create software architecture. Pattern sequences are introduced as a way of combining patterns to solve wider design problems than can be solved by individual patterns. A specific project experience is also described, from which two pattern sequences are drawn. A concrete, step-by-step example of the pattern sequences, based on the project experience, is then presented; this serves to motivate a detailed examination of pattern sequence characteristics which follows. The examination aims to demonstrate why pattern sequences can be employed to create software architecture effectively. The pragmatic issue of how pattern sequences can be applied to real software development finishes off the discussion.

1.1 Intended Audience

The ideal reader of this paper is a software practitioner - whether programmer, developer, engineer, or architect. They are actively involved in creating software, and like the idea of using patterns in their work - perhaps they've come across one or two object oriented design patterns that seem to capture solutions to difficult problems in a simple and easy to understand way.

They also want to understand how patterns can be combined. After all applying a pattern in isolation is all well and good, but complex software just isn't that easy - maybe they've applied design patterns before, only to find their design decisions don't always agree with everyone else's. Or perhaps the reader is a developer or architect familiar with the idea of architecture patterns who wants to know how to fit these things together into an overall architecture.

The reader is also pragmatic. Even if they knew how to go about combining patterns to solve their wider design problems, they recognise that patterns have to be applied in a real project context. They also recognise that many organisations these days are moving towards following an iterative and incremental processes of some kind.

Finally, the reader may be well-informed about patterns and have heard about things called *pattern sequences.* In this case the reader might be looking for a concrete example to help them to understand pattern sequences better.

1.2 Software Architecture and Pattern Sequences

To frame the following discussion, it's necessary to introduce the concepts of software architecture and pattern sequences.

There are many definitions of software architecture in software design literature, but Grady Booch's recent definition is particularly suited to understanding the application of software patterns and pattern sequences to create 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.* [Booch06]

A pattern describes a solution to a problem in particular problem context by resolving the forces acting in that context; a pattern sequence, which is an ordered collection of patterns, resolves many forces and thus results in a collection of design decisions. A sequence of patterns that solve significant design problems therefore, can be seen as a way of creating an architecture.

It's also important to note that while 'cost of change' can reasonably be stated as the most significant factor in design decision making, this is not necessarily a widely held view in software development teams. On the project presented, other more strategic or tactical concerns such as software reuse or flexibility were generally understood as the most significant factors.

The concept of pattern sequences is described in [Porter+04] as follows:

> *A sequence is the ordering of patterns in a given architecture—or perhaps in a family of architectures—that tells the order in which the patterns should be applied. A sequence is a sort of architect's tour of the artifact being built. [...] Good design is about following established sequences.*

By way of example, consider the ENCAPSULATED CONTEXT OBJECT and DECOUPLED CONTEXT INTERFACE patterns, which are described in [Henney06]. A pattern sequence combining these patterns is described as follows:

> *Introduce an ENCAPSULATED CONTEXT OBJECT and define a DECOUPLED CONTEXT INTERFACE*

The above sequence states that shared execution context such as logging or security services should be encapsulated into a discrete object to enable easy propogation throughout a software system. It also states that an interface should subsequently be introduced onto the object. In the resulting architecture, common services and execution related information can be provided to disparate parts of the system in a way that minimizes coupling between context users and context provider, encouraging maintainability and flexibility in the code.

Another patterns-related concept mentioned below that needs a little introduction is that of a *pattern story.* While a pattern sequence describes an ordered collection of patterns that can be applied to create architecture, a pattern story describes an actual architecture in terms of the patterns used to create it [Henney06].

2 *Project Experience - Origin of Sequences and Example*

The pattern sequences and motivating example that are presented in this paper originated on a project where patterns were applied to create a component middleware software architecture; this project is introduced below. For reasons of confidentiality, the following description has been anonymized.

2.1 Project Introduction

The aim of the project in question was to develop the software for an innovative telephony product. C, C++ and Java programming languages were used for development, and it was necessary for the software to run on a custom hardware platform that was being developed at the same time. Scrum [ScBe01] and XP [Beck99] Agile methodologies were followed on the project.

In addition to functional requirements from the telephony domain, there were also non-functional or operational 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 execution; a common approach to management and testing of services; and an extensibility mechanism in the communication path between services.

The middleware was developed by a team of eight people over a period of approximately six months, as part of a wider effort to elaborate the project's software architecture. Early project iterations focussed on elaborating the component middleware, platform, and application-level components. Team members had to coordinate and agree on architecture decisions such as platform abstractions and execution model during this time, and the elaboration was driven by combinations of user-stories and architecture constraints to create a “base-line” software architecture. The middleware elaboration was also pattern-oriented, meaning that the design and implementation of the software, along with the development team's understanding, was based around patterns.

The middleware was also required to support specific services from the telephony domain that had been envisaged as part of the product-line strategy. These include the following, which are reproduced from *"Using Patterns to Create a Service-Oriented Component Middleware"* [Siddle06], a pattern story that describes the patterns applied on the project in detail:

- a service to allow telephone directory lookups both locally within a user's personal directory, and remotely in enterprise directory applications;
- a service to provide "buddy" presence information to local applications, and to propagate user presence information to remote enterprise applications;
- a service to record and manage telephony-related user actions and related events for subsequent user and administrative reference.

The following diagram, also reproduced from the above-mentioned pattern story, provides an overview of the envisaged middleware architecture:

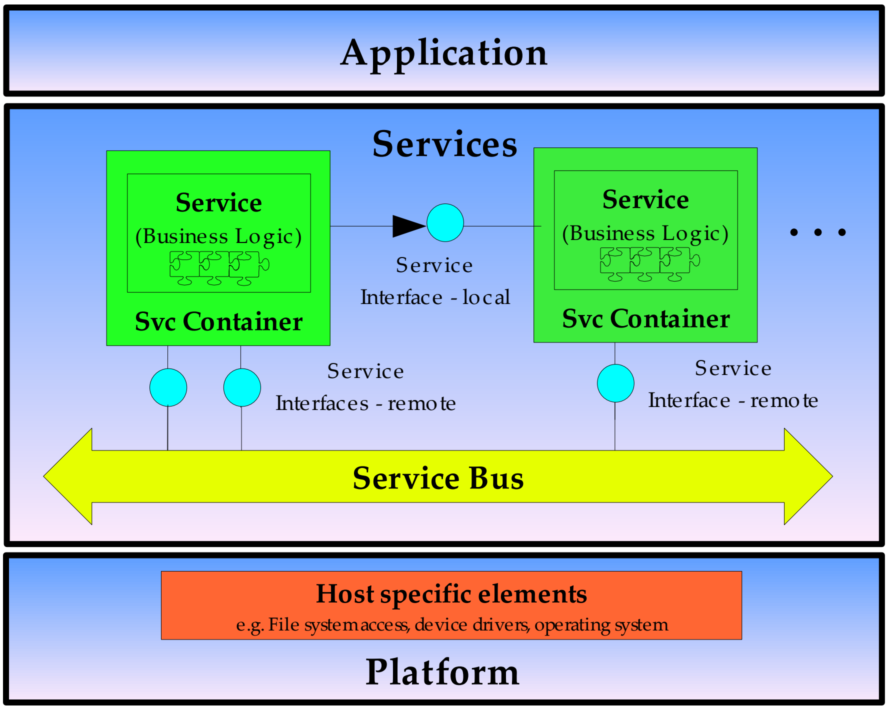


Figure i: Envisaged Middleware Architecture

2.2 Pattern Applications

A broad collection of patterns were applied to create the middleware. These were drawn from several sources, including LAYERS, COMPONENT CONFIGURATOR and INTERCEPTOR from the *"Pattern Oriented Software Architecture"* books [POSA 1-3], SINGLETON and TEMPLATE METHOD from *"Design Patterns"* [GoF], and EXECUTOR [Crahen02] and INVOKER [Voelter+04] that were recommended by a knowledgeable source rather than being drawn from a particular patterns publication. For a fuller picture of the patterns that were applied on the project, the reader is referred to the previously mentioned pattern story.

The following diagram provides a simple, figurative view of the contribution the patterns made to the architecture that was originally envisaged. Not all patterns applied are shown, rather a subset of those applied are included in the diagram by way of introduction. Note that the diagram shows a number of middleware components and the most significant roles from the applied patterns that they embody. In the following diagram, the patterns are italicised:

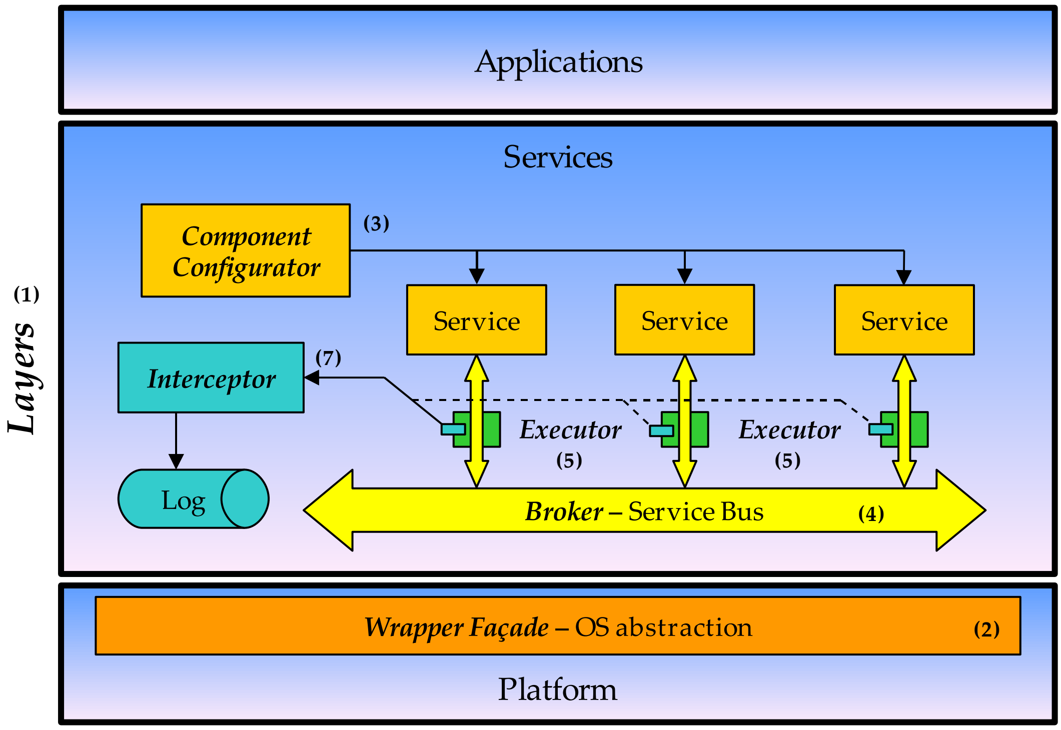
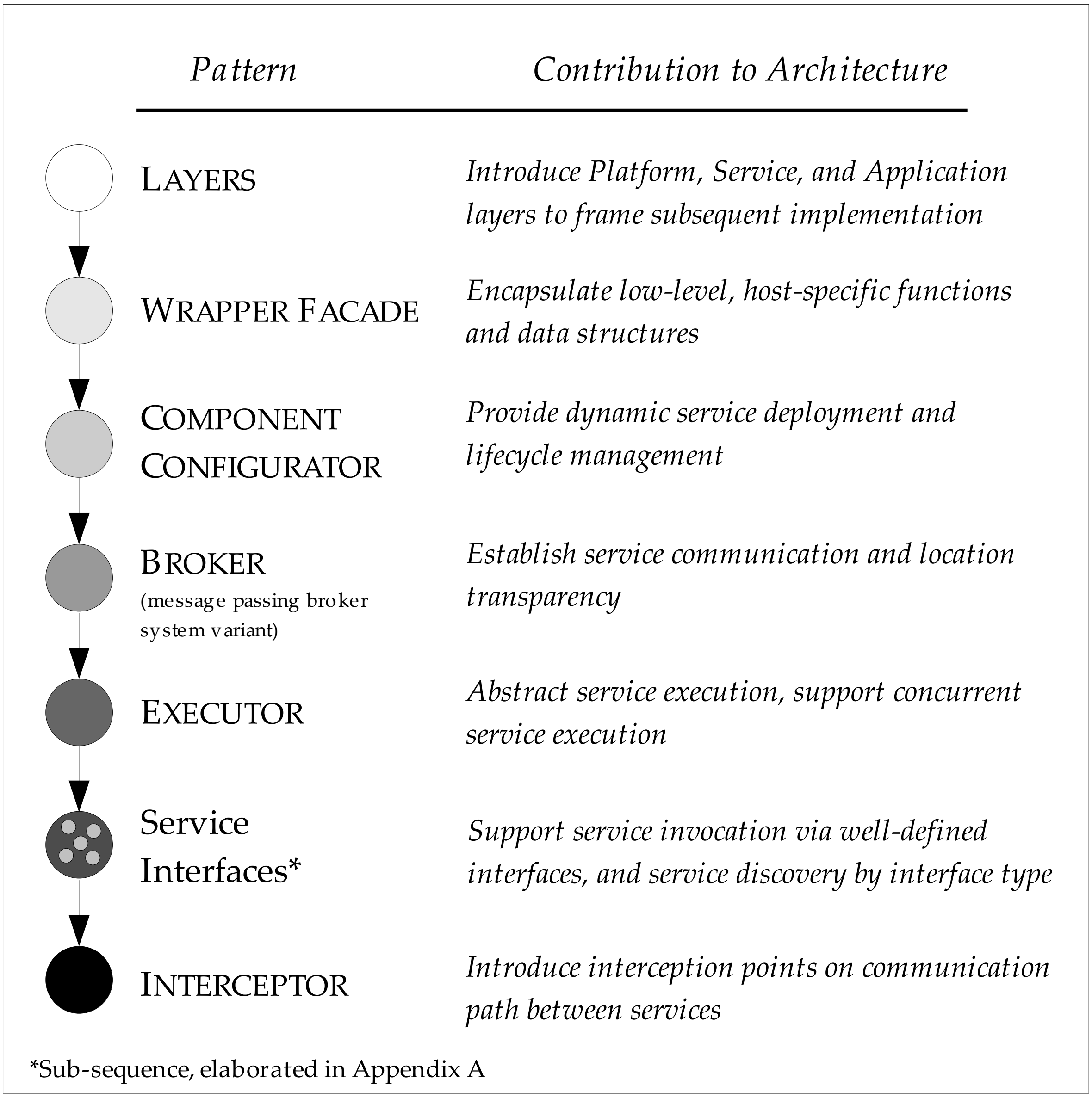


Figure ii: Figurative representation of pattern contribution (in italics) to architecture vision

On reflection, the patterns applications built up the architecture in several discrete, distinct steps - a sequence was observed. Several of these steps have been labelled in the diagram above, though not all steps are included in order to make the diagram easy to understand.

3 *Pattern Sequences*

The following pattern sequence was derived from pattern applications on the project described above; it is proposed as a pattern sequence to solve the same overall problem that was posed to the middleware development team, namely to create a component middleware fulfilling the specific requirements described. The diagram is followed by explanatory text, and a description of how the sequence was derived from the pattern applications that took place:



Pattern	*Contribution to Architecture*
LAYERS	*Introduce Platform, Service, and Application layers to frame subsequent implementation*
WRAPPER FACADE	*Encapsulate low-level, host-specific functions and data structures*
COMPONENT CONFIGURATOR	*Provide dynamic service deployment and lifecycle management*
BROKER (message passing broker system variant)	*Establish service communication and location transparency*
EXECUTOR	*Abstract service execution, support concurrent service execution*
Service Interfaces*	*Support service invocation via well-defined interfaces, and service discovery by interface type*
INTERCEPTOR	*Introduce interception points on communication path between services*

*Sub-sequence, elaborated in Appendix A

Figure iii: Observed pattern sequence

3.1 Explanation

The diagram shows a sequence of patterns that was observed from creating the middleware software architecture introduced above. The step labelled "Service Interfaces" is a *sub*-sequence of the overall middleware architecture sequence, and is derived from a combination of patterns that were applied to introduce support for explicit interfaces between services. The "Service Interfaces" step can be seen as an 'invocation' of the sub-sequence, which is presented in detail in Appendix A - "Proposed Sub-Sequence for Service Interfaces".

The patterns shown above were applied individually to solve design problems that were encountered by the development team in creating the middleware. The patterns in the sub-sequence however were not; they were applied all at once to introduce explicit service interfaces, but not in any discernable order.

As such the overall sequence is treated as an *observed* pattern sequence and is presented here in the main body of the paper, while the sub-sequence is treated as a *proposed* sequence derived from the successful combination of a number of patterns, and is presented in an appendix. The patterns are presented as two sequences to allow a deeper exploration of pattern sequence characteristics below.

3.2 Reasons for Observed Sequencing

So why was sequencing was present in the application of patterns that took place on the project?

Firstly, this is because architectural requirements were captured in a 'road map' that contained a collection of broadly stated architectural concerns, which was loosely ordered according to percieved dependencies. Concerns were taken from the road map and passed to development teams to implement on a per-iteration basis. Examples of concerns in the architecture road map include those project requirements described previously.

Secondly, in certain cases the sequencing was caused by design decisions that were made in relation to the architecture vision. For example "Service" layer elements were expected to only depend on lower-level "Platform" layer elements or other "Service" layer elements. This was dictated by the LAYERS in the architecture vision. As a result, patterns such as WRAPPER FACADE - selected to be applied in the context of the "Platform" layer - would be implemented before or concurrently with "Service" layer elements.

The Agile, iterative context, the presence of a guiding architecture vision and road map, and the use of patterns by the middleware development team ultimately led to a pattern sequence emerging naturally.

3.3 Deriving the Pattern Sequence from the Project

The patterns included in the sequences were selected because of their contribution to the architecture that emerged, and because they show the creation of the architecture from first principles. Patterns were selected for inclusion into the sequences to ensure the key architectural decisions were captured.

The sequencing in the middleware architecture sequence above closely matches the pattern application sequence that occurred during the early stages of creating of the actual architecture. The ordering of pattern applications on the project was the main deciding factor in ordering the pattern sequence. There are however some variations between the sequence presented and the actual implementation sequence that took place, these are described below:

- Some patterns were recognised in the architecture after implementation took place. In terms of the patterns presented here, BROKER and LOOKUP (which is part of the sub-sequence) were identified afterwards.
- COMPONENT CONFIGURATOR and BROKER were actually applied concurrently and independently, they are presented in sequence here because the extra step introduced by separating the patterns in the sequence is a logical one;
- The pattern sequences are not exhausive; not all patterns applied on the project were included;
- As previously mentioned the patterns in the sub-sequence were applied collectively to solve a particular design problem, but not in a discernable order.

For a detailed description of how the patterns were combined to provide service-interfaces on the project, see [Siddle06].

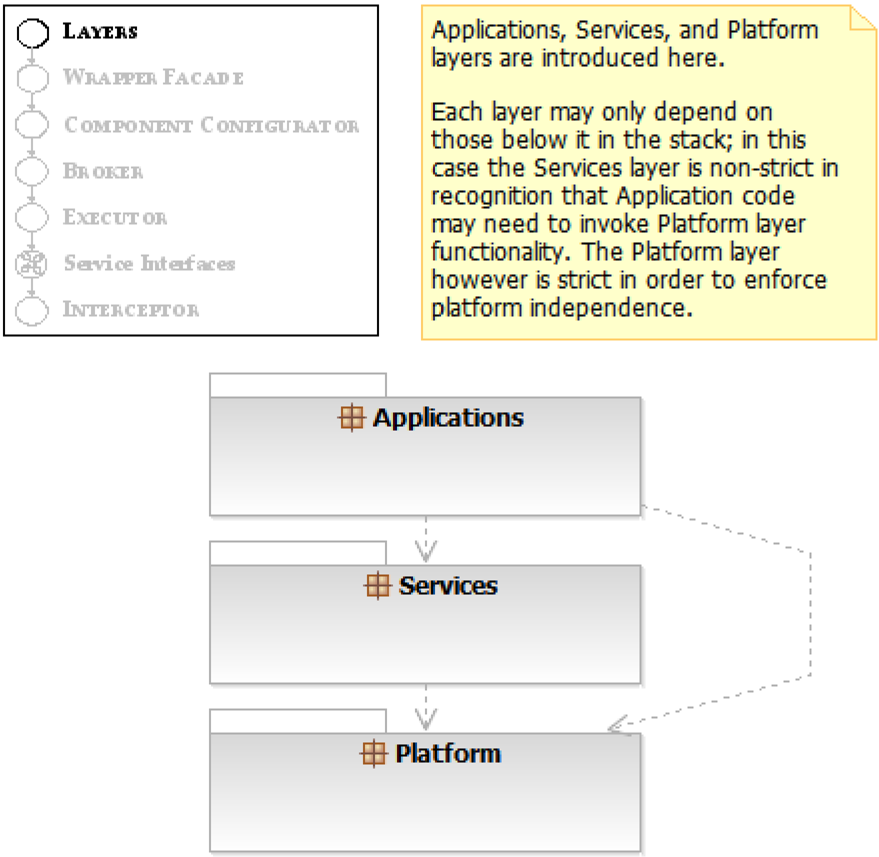
4 *Example Sequence - in Detail*

What would an architecture created by applying the pattern sequences introduced above actually look like? This section of the paper steps through a concrete example of the pattern sequences in the form of UML class diagrams that closely reflect the architectural steps that were taken on the originating project. These diagrams show the core abstractions, roles and responsibilities, and the essential characteristics of relationships between software elements that are introduced by applying the pattern sequence.

That said, some 'massaging' has taken place to simplify the presentation, these simplifications, corrections and caveats are described after the example.

4.1 Step 1 - LAYERS

We start by introducing "Application", "Service", and "Platform" layers. These layers establish high level groupings for software elements that will be introduced into the system later, and introduce some basic concepts such as "Service" and "Platform".



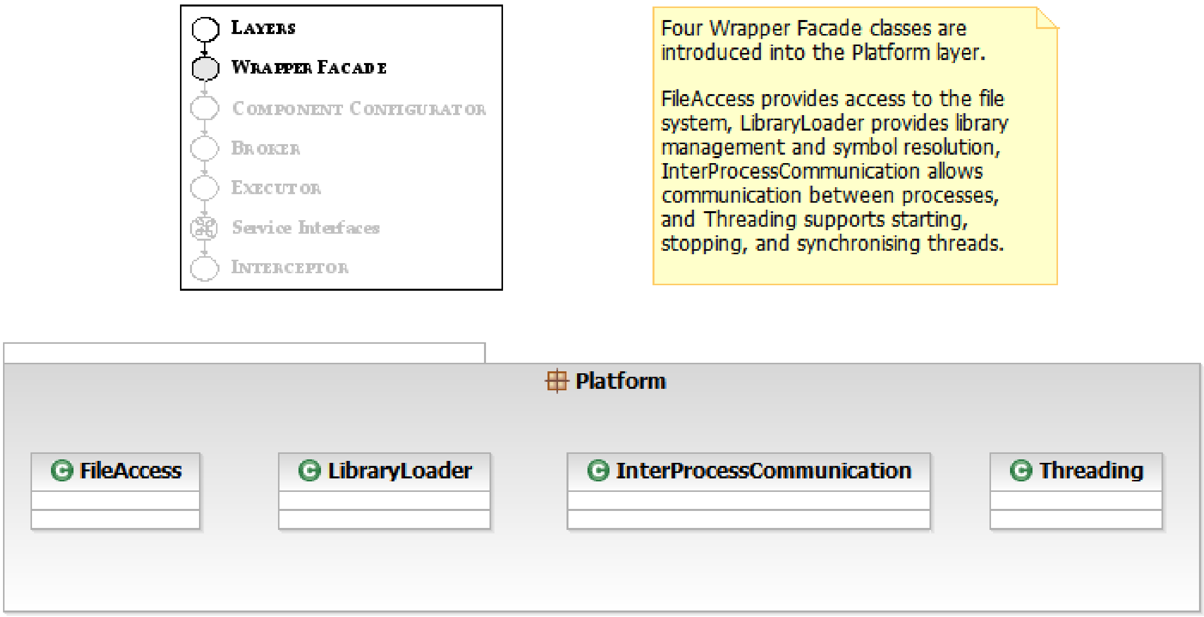
Applications, Services, and Platform layers are introduced here.

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

Figure iv: Step 1 - LAYERS

4.2 Step 2 - WRAPPER FACADE

The next step shows the introduction of a collection of WRAPPER FACADE classes into the architecture's "Platform" layer. These classes provide a set of abstractions which in conjunction with the strict "Platform" layer provide platform independence - one of the goals of the pattern sequence.



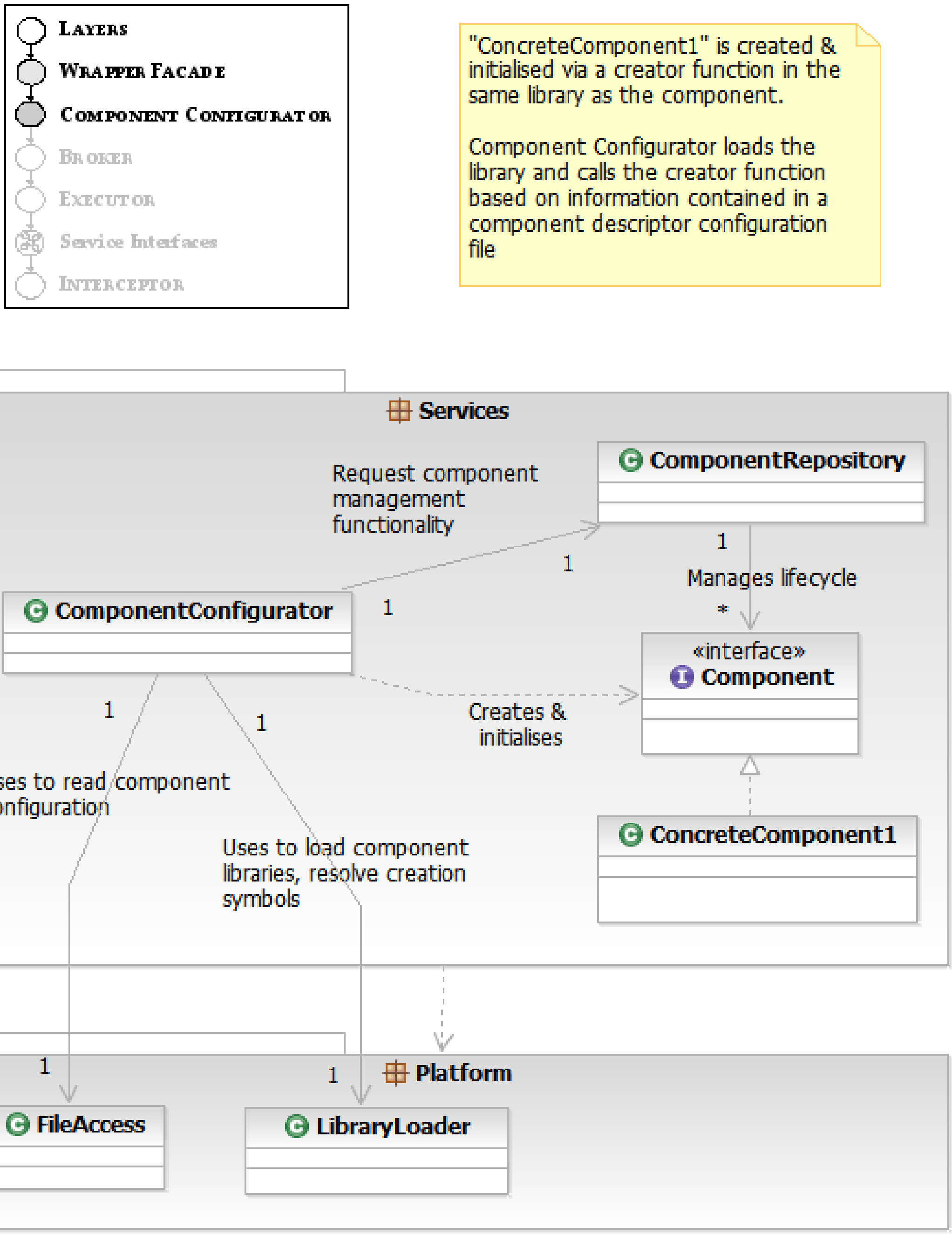
Four Wrapper Facade classes are introduced into the Platform layer.

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.

Figure v: Step 2 - WRAPPER FACADE

4.3 Step 3 - COMPONENT CONFIGURATOR

We now see the architecture after COMPONENT CONFIGURATOR has been applied. `FileAccess` and `LibraryLoader` are WRAPPER FACADE classes introduced above; the other classes are introduced by COMPONENT CONFIGURATOR. The architecture now provides a dynamic mechanism for managing the lifecycle and deployment of platform independent, reusable Services.



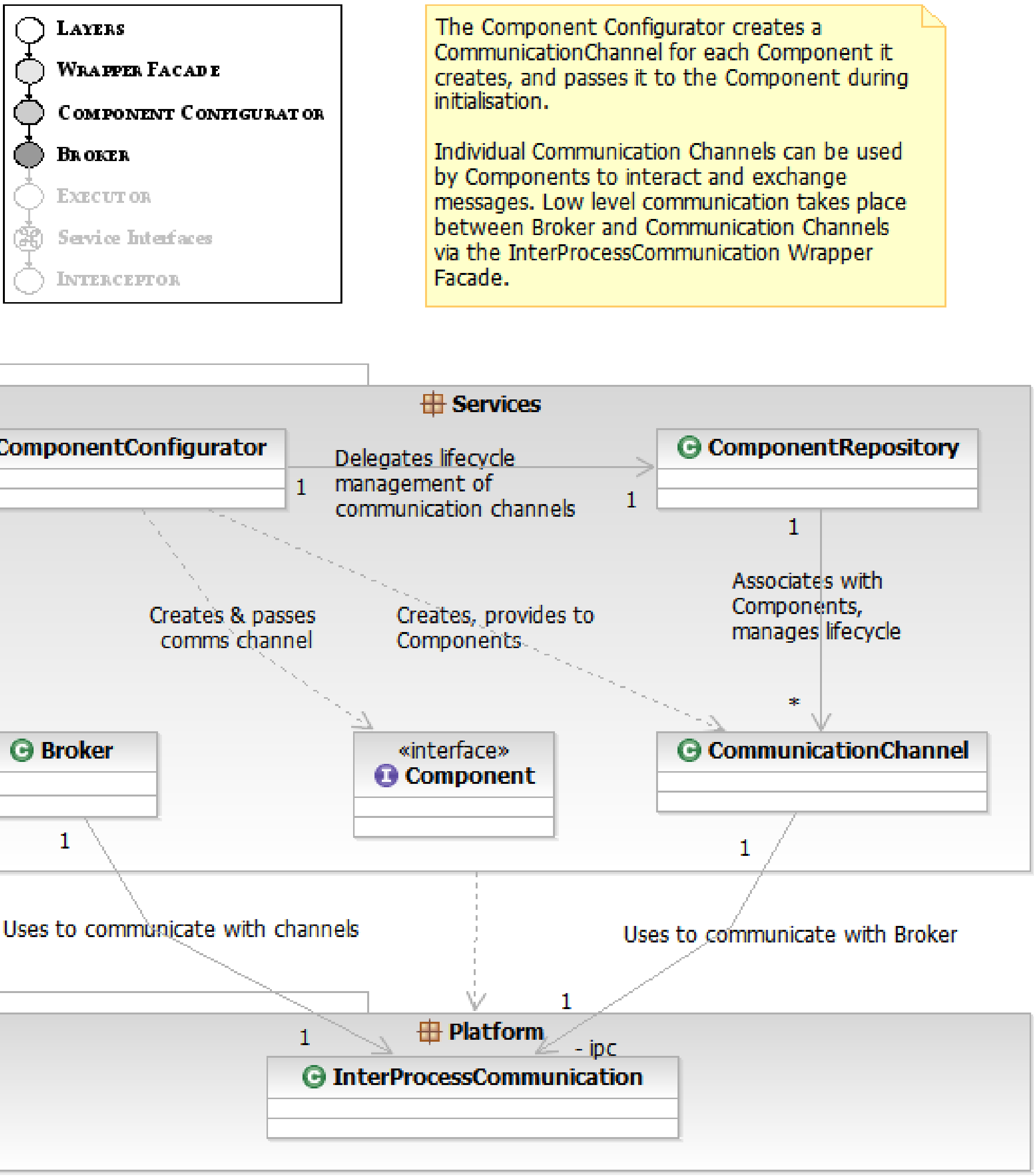
"ConcreteComponent1" is created & initialised via a creator function in the same library as the component.

Component Configurator loads the library and calls the creator function based on information contained in a component descriptor configuration file

Figure vi: Step 3 - COMPONENT CONFIGURATOR

4.4 Step 4 - BROKER

Now the BROKER pattern has been introduced: The `ComponentConfigurator` class creates and initialises an instance of the BROKER `CommunicationChannel` class for each `Component`, then passes it to the `Component` 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. The architecture now provides location transparent communication, in addition to existing capabilities.



LAYERS
WRAPPER FACADE
COMPONENT CONFIGURATOR
BROKER
EXECUTOR
Service Interfaces
INTERCEPTOR
Services
ComponentConfigurator
Delegates lifecycle management of communication channels
1
1
ComponentRepository
1
Associates with Components, manages lifecycle
Creates & passes comms channel
Creates, provides to Components
*
Broker
«interface»
Component
CommunicationChannel
1
1
Uses to communicate with channels
Uses to communicate with Broker
1
1
Platform
- ipc
InterProcessCommunication

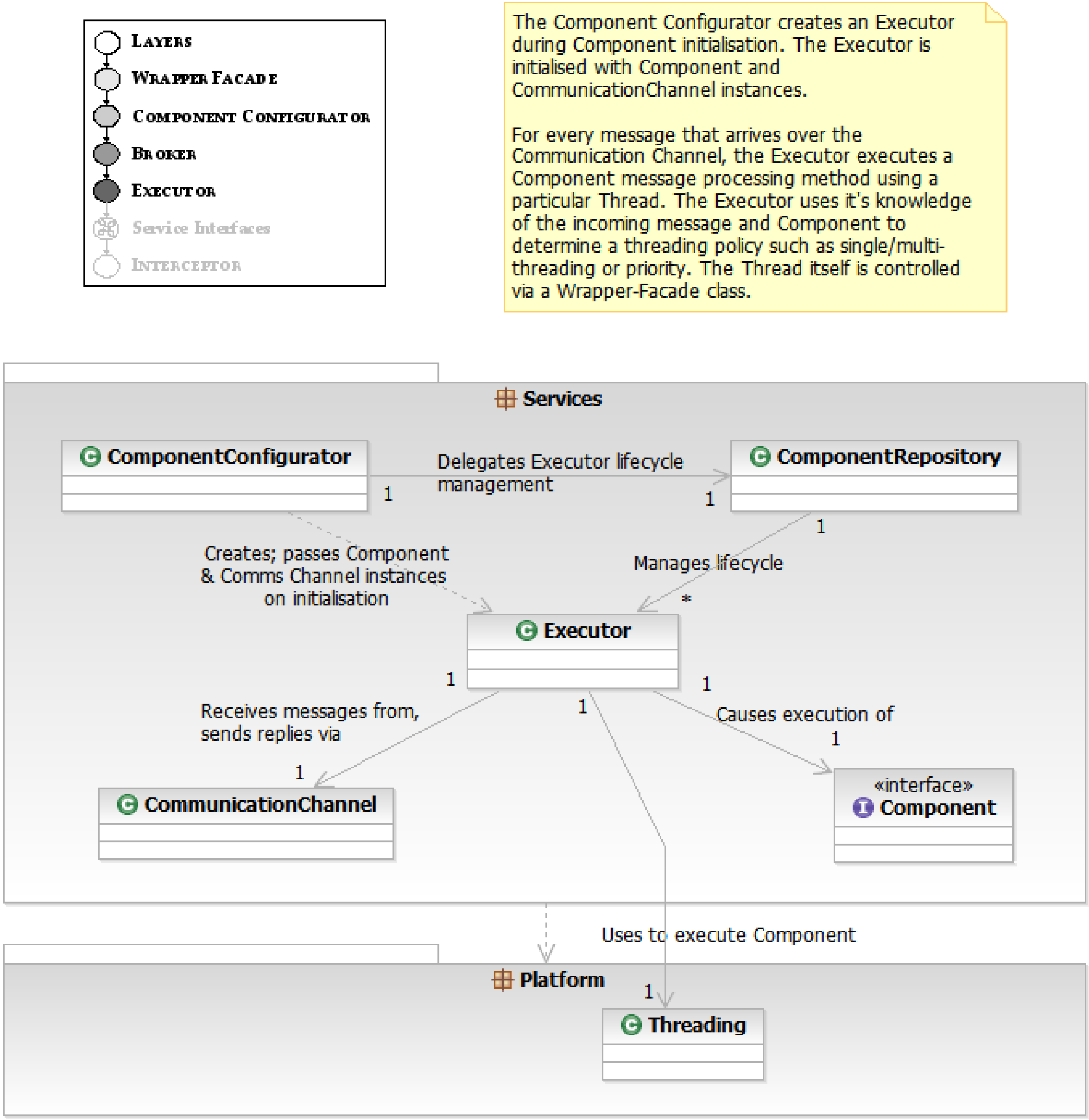
The Component Configurator creates a CommunicationChannel for each Component it creates, and passes it to the Component during initialisation.

Individual Communication Channels can be used by Components to interact and exchange messages. Low level communication takes place between Broker and Communication Channels via the InterProcessCommunication Wrapper Facade.

Figure vii: Step 4 - BROKER

4.5 Step 5 - EXECUTOR

Now EXECUTOR has been applied to introduce an `Executor` class. This class is responsible for handling service execution, and achieves this by waiting for messages to arrive over a service's `CommunicationChannel` object. For each message that arrives, an appropriate thread of execution for message processing is determined (via the `Thread` WRAPPER FACADE); the associated `Component` is informed of the message on the resulting thread. The `ComponentConfigurator` class is again refined to associate an `Executor` with each `Component`, and each service's `Executor` is initialised with the `CommunicationChannel` object associated with the `Component`. The architecture now provides abstracted service execution.



The Component Configurator creates an Executor during Component initialisation. The Executor is initialised with Component and CommunicationChannel instances.

For every message that arrives over the Communication Channel, the Executor executes a Component message processing method using a particular Thread. The Executor uses it's knowledge of the incoming message and Component to determine a threading policy such as single/multi-threading or priority. The Thread itself is controlled via a Wrapper-Facade class.

Figure viii: Step 5 - EXECUTOR

4.6 Step 6 - Service Interfaces

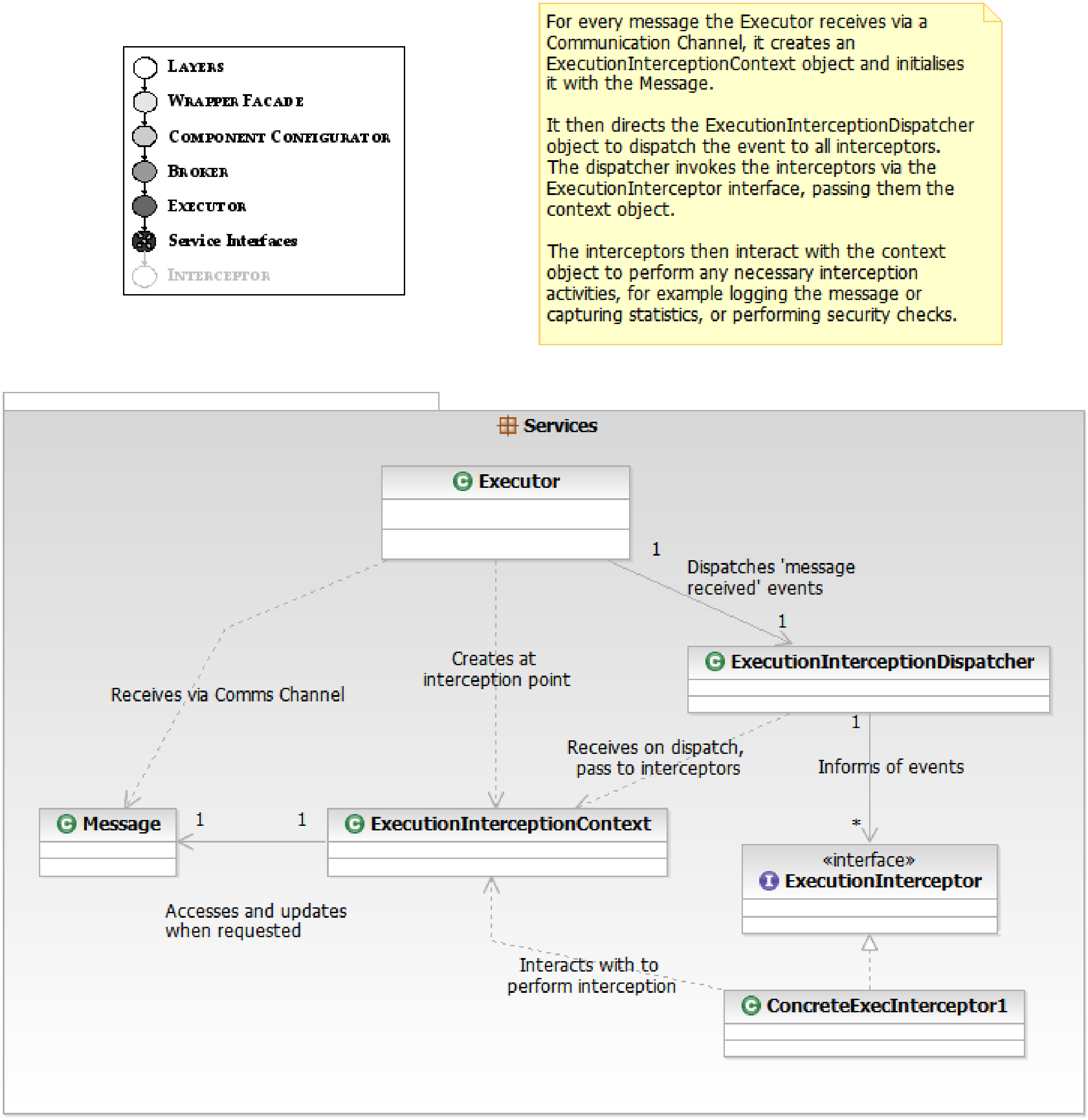
As discussed previously, support for service interfaces was introduced into the architecture by applying several patterns together. A sub-sequence to introduce service interfaces into the example architecture described so far is proposed in Appendix A, accompanied with a continuation of the example to complete the reader's understanding of the emerging architecture.

4.7 Step 7 - INTERCEPTOR

Finally we apply the INTERCEPTOR pattern, which introduces an interception point on the communication path between services; the following diagram shows an interception point immediately prior to service execution, when a message is received.

When an instance of the `Executor` class receives a message for it's 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 `ExecutionInterceptionContext` 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 also have the opportunity to interact with the context object to perform any interception activities they wish to, such as redirecting or blocking the message.

The architecture that results from the application of the INTERCEPTOR pattern supports a flexible and powerful way of interacting with, blocking, or redirecting messages before they result in service invocation. Such a mechanism can be used for logging, statistic gathering, or security checks.


LAYERS
WRAPPER FACADE
COMPONENT CONFIGURATOR
BROKER
EXECUTOR
Service Interfaces
INTERCEPTOR
Services
Executor
1
Dispatches 'message received' events
1
ExecutionInterceptionDispatcher
Creates at interception point
Receives via Comms Channel
1
Receives on dispatch, pass to interceptors
Informs of events
*
Message
1
1
ExecutionInterceptionContext
«interface»
ExecutionInterceptor
Accesses and updates when requested
Interacts with to perform interception
ConcreteExecInterceptor1

For every message the Executor receives via a Communication Channel, it creates an ExecutionInterceptionContext object and initialises it with the Message.

It then directs the ExecutionInterceptionDispatcher object to dispatch the event to all interceptors. The dispatcher invokes the interceptors via the ExecutionInterceptor interface, passing them the context object.

The interceptors then interact with the context object to perform any necessary interception activities, for example logging the message or capturing statistics, or performing security checks.

Figure ix: Step 7 - INTERCEPTOR

4.8 Corrections, Simplifications, and a Caveat

So how does the example above differ from the actual architecture that was created? There are several differences:

- A number of the WRAPPER FACADE classes introduced at step 2 of the example actually emerged over the course of the project, often requiring rework to ensure architecture conformance. Additionally, the actual `Thread` implementation was more complex than that shown, and included classes that were closely coupled with service execution infrastructure rather than being general purpose WRAPPER FACADEs.
- As mentioned previously, 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.
- 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.

The architecture shown in the example reflects the architecture from the originating project at a point in time when the project was transitioning from architecture elaboration to full-scale production. For reasons that won't be elaborated here, the project team's understanding and conformance to the architecture subsequently deteriorated significantly.

As such, the example above should be viewed as an example based on a test-bed architecture, rather than on a full-blown production-proven architecture. The key difference being that the architecture created was proven according to architecture acceptance tests such as acceptable levels of message throughput and platform independence; it was not proven more extensively in the field across many production deployments.

5 *Pattern Sequence Characteristics*

A number of characteristics can be observed in the pattern sequence example above, including:

> *That pattern sequences are defined according to pattern dependencies;*
>
> *the creation and preservation of architecture;*
>
> *the creation of working software at every step;*
>
> *combining pattern applications for design purposes;*
>
> *and pattern sequences as patterns themselves.*

The above characteristics are examined over the next few sections; this discussion is followed by a short examination of how pattern sequences can form the basis of pattern languages [Alexander+77] [POSA5]. It should be noted that characteristics of creating and preserving architecture and of creating working architecture at every step were suggested by the work of Christopher Alexander in [Alexander02]. The concept of gradually creating a 'good' or 'whole' architecture through a series of "structure preserving transformations" is explored extensively thoroughout the referred text.

5.1 Dependencies define the Sequence

A pattern sequence breaks up the overall problem into managable pieces, and solves each sub-problem in turn according to the dependencies between them. Two types of dependencies between patterns can be seen in the observed and proposed sequences: *implementation* dependencies and *conceptual* dependencies.

Implementation dependencies are those where one pattern builds on or refines software elements introduced by another, given the overall problem that is being solved by the sequence. An example of this can be seen in step 3:

> The classes introduced by COMPONENT CONFIGURATOR depend on those introduced by WRAPPER FACADE. The `ComponentConfigurator` class requires the `FileAccess` class to read configuration files. It also requires the `LibraryLoader` class in order to load libraries and resolve code symbols, such as the creator function name, to locations in memory. The implementation dependency exists in this case because of the platform independence goal; it would not be appropriate for the `ComponentConfigurator` class to access low level operating system functions directly.

Step 4 reveals that implementation dependencies are not the same as code dependencies, such as in the example above where one class invokes a method on another class:

> We saw above that the `ComponentConfigurator` class is responsible for managing `Component` lifecycle; this responsibility would naturally include the lifecycle of

associated objects. The `CommunicationChannel` class, introduced in step 4, provides support for location transparent communication. So, in order to support location transparent `Components`, the `ComponentConfigurator` was updated to provide a `CommunicationChannel` for each `Component`.

The resulting implementation shows a code dependency from the `ComponentConfigurator` class (introduced in step 3) to the `CommunicationChannel` class (step 4), but this does not mean there is an equivalent dependency in the pattern sequence. Such a dependency would mean the pattern application at step 3 was taking place without certain required software elements from step 4 being in place.

The dependency is still from step 4 to step 3 because the application of BROKER has refined the implementation of COMPONENT CONFIGURATOR to support location transparency. This refinement is another type of an implementation dependency - according to the overall goals of the sequence, BROKER requires that COMPONENT CONFIGURATOR has been applied first, in order to refine it.

More implementation dependencies can be observed in steps 5 and 7.

In step 5, EXECUTOR builds on the implementation of BROKER by introducing an `Executor` class that reads messages from `CommunicationChannels`. This step shows another refinement dependency, where the `ComponentConfigurator` class now associates an `Executor` with each `Component`. In step 7, INTERCEPTOR refines the implementation of EXECUTOR, such that the `Executor` class invokes an instance of the `ExecutionInterceptionDispatcher` class to support service invocation interception.

Conceptual dependencies are those where domain model concepts introduced by one pattern depend on those introduced by other patterns. The most compelling example of this is how LAYERS introduces concepts such as "Service" and "Platform" that give meaning to subsequent patterns in the sequence:

WRAPPER FACADE introduces "Platform" software elements; other patterns introduce software elements that support the "Services".

COMPONENT CONFIGURATOR, BROKER, and EXECUTOR are all strongly shaped by the concept of a "Service" in the architecture created by the pattern sequence. COMPONENT CONFIGURATOR manages service lifecycle; BROKER provides location transparent communication to services; and EXECUTOR ensures that execution is abstracted from service business logic.

To put the above patterns before LAYERS would beg the question, "Why am I applying this pattern?", "What is it that requires location transparent communication?", "Is it all execution within the system that needs to be abstracted?".

So if the patterns in a sequence are shaped by certain concepts, the patterns that introduce those concepts should ideally come first.

Another example can be found in step 7:

> A service execution INTERCEPTOR is shaped by the concept of service execution, so naturally follows from the LAYERS and EXECUTOR patterns.

5.2 Creation and Preservation of Architecture

The aim of applying pattern sequences is to create architecture, so it should be unsurprising that the sequences show new design problems being tackled and new software elements being introduced at every step. This characteristic can be seen at every step in the pattern sequence and motivating example.

Additionally, the application of each pattern in the sequence introduces new architecture while preserving existing architecture as much as possible. The preservation of existing architecture can be seen in step 2:

> By introducing `FileAccess` and `LibraryLoader` WRAPPER FACADEs in the "Platform" layer, the conceptual dependencies and logical groupings introduced by LAYERS are supported.

It would have been possible to introduce a dependency going upwards in the layers, for example `LibraryLoader` could have invoked a static string manipulation helper method on `ComponentConfigurator`. Such an implementation would have transformed the architecture in a destructive fashion because the previously introduced layers would have been ignored.

Introducing a WRAPPER FACADE implementation in the correct layer with dependencies either within the same layer or going down the layers preserves the architecture.

Similarly in step 4:

> The BROKER's `CommunicationChannel` and `Broker` classes build on the `InterProcessCommunication` WRAPPER FACADE from step 2.

As such the application of BROKER not only introduces location transparent communication between services but also supports platform independence. If `CommunicationChannel` invoked operating system calls directly, the architecture introduced by LAYERS and WRAPPER FACADE would have been invalidated.

In step 5, the established architecture is preserved in a number of ways. Firstly:

> The LAYERS and WRAPPER FACADEs previously introduced are used correctly because the `Executor` requests low-level thread related functionality via the `Thread` WRAPPER FACADE.

Secondly:

> `Executor` instances are created and managed by the `ComponentConfigurator` and are associated with `CommunicationChannel` and `Component` instances in the `ComponentRepository`.

This preserves the existing component configuration architecture. Thirdly:

> The `Executor` class reads messages from the `CommunicationChannel` and dispatches them to the `Component` for processing.

Assuming the `Executor` does not violate location transparency and makes correct use of the `CommunicationChannel`, this preserves the architecture introduced by BROKER.

The characteristic of creating and preserving architecture is thought to be present in the pattern sequence because of the use of Agile to elaborate the original architecture; discrete, working deliverables were expected on a regular basis and extensive architecture rework was not acceptable. As such, each pattern application had to fit with the existing architecture as much as possible.

There are a number of potential benefits from applying pattern sequences that exhibit these characteristics.

In an elaboration exercise where an architecture maintains coherence as it emerges, it may be possible to avoid expensive rework required to ensure architectural conformance, providing a more efficient development effort. It's also thought that a more coherent, understandable software system will emerge, encouraging team understanding and helping to reduce premature software aging.

The coherence is thought to emerge from the sequence because of the ordering of pattern applications according to dependencies between the patterns. Subsequent pattern applications will fundamentally be affected by earlier ones, so by getting the order right, the architecture that emerges should hang together better - roles and responsibilities of functional units will make sense and will embody design decisions taken to balance all architectural forces, rather than just dealing with one force.

The overall coherence of the architecture may also be supported when domain model concepts are introduced in the order of dependencies between them, which can help to ensure that the design decisions are consistent with the underlying domain model. An example that points to the potential effect of incorrect conceptual ordering would be if LAYERS were not initially established, "Service" related design decisions would run the risk of making invalid assumptions or missing important information.

5.3 Working Software Architecture at Each Step

Another interesting potential characteristic of pattern sequences is that of creating a workable software architecture at each step.

In the pattern sequence presented in figure iii, each pattern builds upon the previous patterns to solve a distinct part of the overall problem, and makes no reference to subsequent patterns. As a result of this, each pattern application results in a workable architecture. This is a potential characteristic because it can only be seen in the middleware architecture pattern sequence, not in the sub-sequence that introduces service interfaces. This is explored further below.

The examples shown up to this point describe several distinct software architectures:

> At step 3, the LAYERS, WRAPPER FACADE, and COMPONENT CONFIGURATOR patterns are combined to provide a simple architecture supporting platform independent services.
>
> Step 4 refines the architecture by applying BROKER to allow services to communicate in a location transparent way.
>
> Step 5 refines the architecture further by introducing EXECUTOR to abstract execution away from services.
>
> Finally step 6 introduces support for cleanly defined interfaces between services, and step 7 adds a powerful INTERCEPTOR to provide extensibility and control of the communication path between services.

Each of the architectures described above is useful and coherent in its own right, and each subsequent pattern application adds to the existing architecture to create a new architecture.

But what about that sub-sequence? Applying EXPLICIT INTERFACE doesn't result directly in a new working software architecture, nor does the introduction of a `Proxy` class without an equivalent `Invoker`. This is thought to be because the majority of the patterns in the sub-sequence were not written with the delivery of discrete working software deliverables in mind. It's also because the sub-sequence is speculative - it was derived after the fact from patterns that were applied to solve an overall problem in the architecture. No single pattern was evident that introduced service interfaces while also providing a working solution.

So it may be that this desirable characteristic is apparent in the observed sequence simply by coincidence - that it can be seen in the middleware architecture sequence because of the Agile context, where a complete working deliverable was required at the end of each iteration.

That said, it may be possible derive a pattern sequence that does create working architecture at each step. Where individual patterns don't provide a discrete working step, compound patterns or pattern sequences may do. Such steps would either be solving 'functionally complete' problems - where the solution provides a discrete, useful piece of function to the project stakeholders, or establishing stable but intermediate pieces of working software as part of a larger effort.

This potential characteristic of pattern sequences is a desirable one. It may contribute to the creation of a well-formed, coherent architecture, where each step provides a solid foundation for subsequent steps; it may also help to manage evolving requirements and changing project needs by supporting changes of direction while sustaining a coherent architecture. This idea is explored later in the section related to Agile software development.

5.4 Combine Multiple Pattern Applications for Design

In step 2 of the example that can be found above, the WRAPPER FACADE pattern introduced classes that were used by subsequent patterns in the sequence:

> `FileAccess` and `LibraryLoader` were used by COMPONENT CONFIGURATOR, `InterProcessCommunication` was used by BROKER, and `Thread` was used by EXECUTOR.

Each of these represent a distinct application of the WRAPPER FACADE pattern.

The above statement highlights an important characteristic of pattern sequences: that the patterns do not, or should not reference patterns that appear later in the sequence, but realistically some forward referencing is necessary for implementation purposes.

What does this mean? The WRAPPER FACADE pattern is generic, does not require other patterns to have been applied to provide useful functionality, and could be applied any number of ways to support subsequent development. But, without *some* forward thinking, the middleware development team could have spent six months creating a comprehensive, exhaustively tested, thoroughly reviewed collection of WRAPPER FACADE classes to support every conceivable future need for platform abstraction. Obviously this is not conducive to the efficient use of software development resources or the hoped for success of a project.

So, when applying a pattern from a sequence, some thought should be given to how the resulting implementation will be used; future patterns in a sequence provide at least some of this implementation context. In the motivating example described in this paper, WRAPPER FACADE classes were introduced to support the patterns that appear later in the sequence.

It is also possible to distribute a single pattern, as it appears in a sequence, as multiple pattern applications throughout a pattern sequence application. Why, for example, should the introduction of an `InterProcessCommunication` WRAPPER FACADE precede the application of COMPONENT CONFIGURATOR? The latter would make no use of it, so it might make more sense to distribute the WRAPPER FACADE applications throughout the sequence, introducing them as and when they are needed.

So for *implementation* purposes multiple, similar applications of an individual pattern that appears in a sequence can be distributed throughout the application of the sequence. For design purposes, it's reasonable to describe just one pattern instance and to 'annotate' it with all expected applications.

This difference in the treatment of patterns for design versus implementation can be seen in the motivating example:

> Step 2 shows all instances of WRAPPER FACADE that are required by later patterns, but as

> described previously this does not reflect the actual project experience, where WRAPPER FACADE classes were introduced as needed.

The key point for design is that domain model concepts, characteristic structure, and expected software element roles and responsibilities are introduced at the correct point in the sequence to enable correct reasoning and decision making.

But there is the possibility that a documented pattern sequence will explicitly include a pattern several times. This is likely to occur where multiple instances of the same pattern are solving problems in different contexts, so warrant separate inclusion in the sequence to prompt a separate design activity appropriate to the context. An example of this is provided by INTERCEPTOR:

> Imagine that an interception point is also required in the BROKER pattern's `Broker` class. This interceptor must provide access to messages in a similar way to the service execution interceptor described previously, but only to a limited set of message properties and content in order to prevent performance problems due to the high volume of message throughput handled by the `Broker` class.

In this case, it would be better to include INTERCEPTOR twice in the pattern sequence documentation, because this would hopefully prompt a separate design discussion in each case, appropriate to the quite different contexts.

This characteristic of pattern sequences has been commented on before in [Zdun06], specifically in relation to selecting patterns, where each pattern selection represents an event in the pattern selection process.

5.5 Pattern Sequences as Patterns

A pattern sequence is a solution to a problem in a particular context; it resolves the forces expressed in the architecture vision. Therefore, it is reasonable to state that pattern sequences could themselves be patterns. With some formalisation, the sequences presented in this paper would be *candidate* patterns, though without further examples of them in action it's impossible to known whether they really solve the problems they claim to.

The emphasis of this paper has been understanding patterns and pattern sequences by example rather than presenting new patterns, so the observed and proposed sequences are not presented as candidate patterns here.

The relationship between compound patterns, pattern sequences, and patterns generally is explored further in [Henney06] and extensively in [POSA5].

5.6 From Pattern Sequence to Pattern Language

The patterns applied to create the software architecture were not drawn from a particular pattern language - they were selected in a piecemeal way from a variety of sources to solve each design problem as it arose. The concept of pattern languages was not prominent in the project team's collective knowledge at the time; an alternative approach to pattern application on the project would have been to select a pattern language publication to guide the application of patterns.

However the patterns applied, and to a fuller extent the patterns that were selected for the sequence, were collectively applied to create software architecture in a particular context. As such the proposed sequences can be seen as a reasonable collection of design steps around which a pattern language could be formed - the concrete project background gives the sequence a firm basis in reality.

To form a pattern language, it would be necessary to consolidate the patterns presented to ensure they formed a coherent whole. The sequence as presented combines patterns from a variety of loosely related sources; as such, it is possible that an attempt to follow the sequence as presented would run into problems caused by the inconsistent context of those diverse pattern descriptions. Inconsistency in pattern descriptions is thought to have derailed a number of pattern application attempts on the project described previously; in particular it was difficult for some team members to grasp how the INTERCEPTOR and ASYNCHRONOUS COMPLETION TOKEN [POSA2] patterns should be applied in the emerging architecture.

Such a consolidation of patterns into a 'proto' language is outside the scope of this paper, also pattern languages for *Enterprise, Internet and Realtime Distributed Object Middleware* [Voelter+04] and *Distributed Computing* [POSA4] already exist, so such a process would be an unnecessary duplication of effort.

6 *Applying Pattern Sequences*

Having discussed the characteristics of pattern sequences above, the following sections now examine how can pattern sequences be applied on an actual projects.

6.1 Retelling the Architecture

The pattern sequences presented above were arrived at through a process of reflection, after the software architecture had been created. It may be possible for a sequence to play an effective role in communicating both the problem domain and the architecture that was created, over and above a pattern story describing specific details of the project. A pattern story captures the specifics of pattern application, while a sequence derived from a story is abstracted away from less relevant details.

The proposed sequences gradually introduce the important entities and concepts from the problem domain, and give a grounding in the UBIQUITOUS LANGUAGE [Evans03] of the project - or at the very least the language understood by the sequence creator. Project specific details included in a pattern story may hinder as much as help understanding; understanding an existing architecture via the more generalised pattern sequence can contribute towards understanding the original aims of the project and the underlying problem domain.

The sequences also gradually build up a picture of the architecture in the recipient's mind, in a similar way to the 'tea garden' example in [Alexander02], ensuring that all of the important architectural decisions are understood both individually and in relationship to preceding architectural decisions.

Retelling the architecture in this way ensures that team members joining an existing project are well grounded in the language and domain model of the project, are able to make design decisions consistent with the existing architecture, and have a deeper understanding of the project aims and problem domain than just a pattern story would provide.

6.2 Establishing Initial Architecture Decisions

Pattern sequences can be used to establish initial architecture decisions, prior to code production commencing. If a suitable pattern language can be found for the problem domain where up-front design decisions are desired, a previously followed pattern sequence that is known to create the desired architecture offers a good solution to making up-front design decisions.

It may also be possible to compose a pattern sequence from several pre-selected sub-sequences. For example in [Henney06], several possible sequences from a particular pattern language are described. A project specific pattern sequence could be composed of sub-sequences taken from different sources.

If no pattern sequences from the selected language are known, the least attractive option would be to define a sequence according to the dependencies described in the pattern language.

The first option described above - following a path that is known to lead to success - is the most intuitively appealing. That said, success is not ensured by following a known sequence - after all certain crucial decisions may have been overlooked or simply forgotten when deriving a sequence from actual pattern applications. But an associated pattern story, describing an actual project where the sequence was applied, may offer "backup" to a pattern sequence by providing enough concrete details so that any shortcomings in the sequence can be corrected. Some important decisions may even have been made subconsciously, and missed from both the story and the sequence; the specific context provided by the story may help to identify such decisions if problems are encountered when applying the sequence.

In whichever way the pattern sequence to create a software architecture is determined, whether from existing sequences, sub-sequences, or from a pattern language, the patterns selected should be consistent with one another. One way of achieving this would be to draw the patterns from a single pattern language. For example on a distributed enterprise application project creating an internet portal for a supermarket, it would not make sense to combine patterns or sub-sequences from distributed computing and telecommunications. A coherent, well formed pattern language that is known to solve problems in the problem domain of the project is more likely to help developers to create a coherent, well formed architecture than a confusing set of ill-matched, inconsistent patterns that solve problems in totally different domains.

A pattern sequence chosen to capture initial architectural decisions prior to code production carries with it the first attempt at key design decisions, dependencies and relationships between proposed software elements, and an initial domain model. The ordering of such a pattern sequence also indicates an appropriate order that these aspects of the project can be firmed-up in, to encourage the creation of a well-formed architecture.

6.3 Pattern Sequences as Implementation Road-maps

On the project where the pattern sequences and motivating example originated, patterns were selected and implemented one at a time as required by the architecture road map, vision, and selected user stories, on a per-iteration basis. This was the selected development process.

However using a pattern sequence as the primary way of guiding software development is an alternative approach that is worth considering.

The aim of creating software architecture by following a pattern sequence is to create coherent, well-formed, and understandable software architecture that is fit for purpose. When design problems are tackled one at a time, with each problem solution building upon the solutions to previous problems, it is thought that architecture coherence will be greater than an ad-hoc approach to solving design problems. In practical terms, the aim is for each code element to be written according to the best understanding of the problem domain, keeping every key design decision made so far in mind.

The most compelling benefit of such an approach is the possibility of combining stepwise refinement and growth of a software system, while simultaneously making use of the best known approaches to solving design problems in that system's domain. The benefits of approaching software development in an iterative fashion are well understood:

> *Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred by commercial developers because it allows a potential of reaching the design goals of a customer who does not know how to define what they want.* - from the Wikipedia definition of *Software development process* [Wikipedia]

So, by employing pattern sequences that provide working, functionally complete increments in an ongoing way, it is thought that that the benefits of both can be realised. Such an approach can also offer an easy to understand roadmap of development activities, which though it may ultimately be incorrect, is known to be approximately correct for the particular pattern language 'family member' that the sequence represents. Such an approach may help to improve planning and resourcing aspects of software development.

One of the significant problems with this approach is the availability of known, trusted pattern languages for software development. Though, as previously mentioned, such pattern languages are starting to emerge, until sequences from such languages are employed to implement significant, complex, challenging projects, they will be an unknown quantity. It is big step to trust all design, implementation, planning, and resourcing activities to single pattern language.

6.4 Applying Pattern Sequences on Agile Projects

The relationship between pattern sequences and Agile software development is not simple and is also not the focus of this paper, however the most prominent ways that pattern sequences interact with Agile processes are briefly explored here.

One of the most interesting interactions between pattern sequences and Agile methodologies is in relation to incremental design. Pattern sequences propose a roadmap of design decisions that can be followed to create a coherent, complete and well-formed architecture. In this sense, a pattern sequence can be seen as a "Big Up-Front Design", which is generally unacceptable in Agile methodologies. These methodologies typically evolve software architecture through emerging requirements, along with customer and technical feedback.

That said, there may be a middle-ground where pattern sequences and Agile approaches can be applied together effectively. One known issue with incremental design is that teams must be fully committed to architecture rework as the architecture evolves, or risk building on prototype or partially complete software. If not carefully tracked and reworked early enough in development, such prototype code can become an intrinsic part of the architecture and be very difficult and costly to rework at a later stage. This did in fact occur with early versions of certain middleware architecture components on the project described above.

As such, two possible scenarios present themselves where Agile and pattern sequences can be combined. The first scenario would be when development teams or organisations are transitioning to Agile and are first learning the strong discipline required to perform incremental design effectively. In this scenario, a pattern sequence would limit the set of design decisions that can be made by the team, and while this does limit the evolutionary aspect of Agile, it could make the all the difference in delivering software fit for purpose.

The second scenario would be when there is a reasonable degree of certainty that the pattern sequence will provide a suitable architecture, and the possibility of emerging requirements is not high enough to risk evolving an architecture. In this scenario, the ability to deal with all emerging requirements is traded for a reduction in the risks associated with incremental design.

Note that in both scenarios, the variability provided by the underlying pattern language would allow for some flexibility because subsequent patterns in the sequence can be changed or even ignored as the architecture is being developed. So while applying a pattern sequence on an Agile project will reduce the ability to deal with emerging requirements, it does not eliminate the ability altogether.

Additionally, pattern sequences can support adaptation because the impact of changes in project direction can be more easily assessed. If a new requirement requires changes to previously written software, the sequence can show the other code elements that will be directly affected by the change. It can also show what the far-reaching effects are likely to be; removing a pattern implementation that has influenced subsequent software development may cause inconsistency in the architecture, the sequence can help to determine if this will happen.

Creating working software over comprehensive documentation is also supported, to a degree, by pattern sequences. Pattern sequence literature can be treated as design documentation, thus helping to keep a development project focussed on the creation of useful software rather than documentation. There is a risk though that determining and documenting the 'right' sequence upfront would simply take the place of extended "analysis and design" phases of more traditional software development processes and actually work against Agility.

The Agile focus on team communication and collaboration over processes and tools is also both helped and hindered by pattern sequences; on the one hand a pattern sequence enshrines the proposed solution to the core problem being solved, and carries with it the language and concepts of the problem domain. This helps to create a UBIQUITOUS LANGUAGE which naturally supports agility. However, the adoption of a pattern sequence could result in restrictive and counter-productive processes concieved to ensure 'pattern sequence conformance'.

Communication between team members is also supported by the ongoing coherence of the architecture being developed - individual software elements created during development will be consistent with other software elements, as a result the software system as a whole will be easier to understand and discuss.

7 Summary

This paper demonstrated how pattern sequences can be applied to create software architecture effectively. A particular project experience where patterns were applied to create the architecture was described. On reflection, the pattern applications showed noticable sequencing and contributed in an ongoing way to the incremental growth of the architecture. The reasons behind this sequencing was explored, then pattern sequences to reproduce the architecture were presented.

A motiviating example of the sequences, drawn from the project experience, was then described step-by-step and this served as the basis for a discussion of pattern sequence characteristics that followed. The discussion examined pattern dependencies, the creation and preservation of architecture, and the provision of discrete increments of working software. It was also recognised that similar applications of a single pattern can be combined into one step of a pattern sequence for design purposes, but distributed throughout a pattern sequence application for the purposes of effective implementation; that successful, widely recognised sequences may themselves be patterns; and that sequences may form the basis of pattern languages.

Finally the possible ways of applying sequences were examined - ranging from simply 'retelling' the steps taken to create architecture, through using sequences to establish up-front design decisions, to full adoption of pattern sequences as a development processes. This included a brief examination the interaction between pattern sequences and Agile software development.

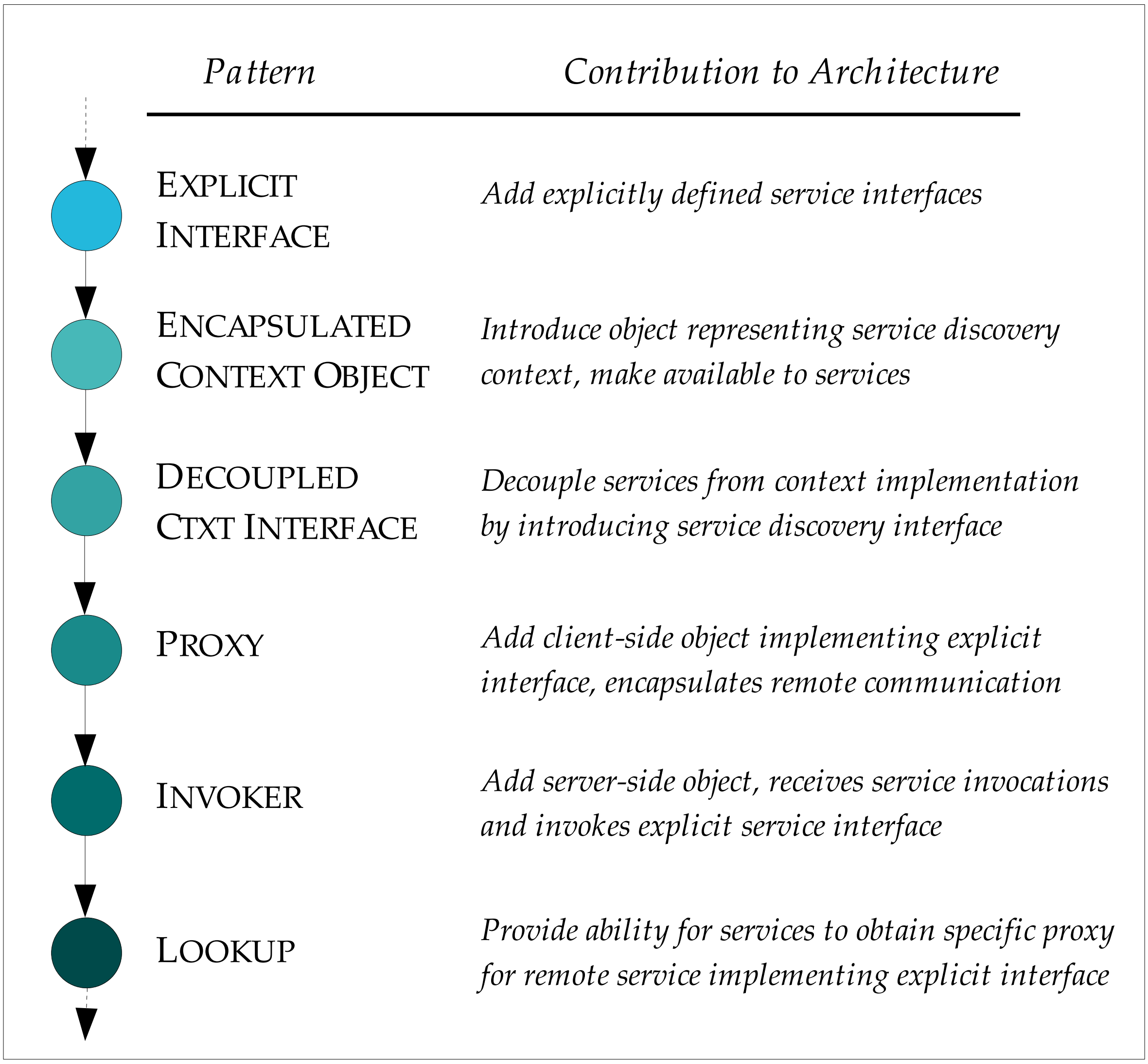
Acknowledgements

Thanks to Kevlin Henney for providing the inspiration for writing this paper and for extensive feedback and support during it's development, to James Coplien for providing helpful comments and feedback on an early draft of the paper, and to James Noble for shepherding the paper for EuroPLoP 2007 and offering many insightful comments.

8 Appendix A - Proposed Sub-Sequence for Service Interfaces

The following pattern sub-sequence is proposed as a way of introducing support for location transparent service interfaces into an architecture emerging from the pattern sequence seen in the main body of this paper. A continuation of the example from the main body of the paper follows. The example shows four views of the architecture from the project that the pattern sequence was drawn from, *after* the patterns had been applied. Discrete steps are not shown because discrete steps were not taken on the project: the architecture emerged somewhat more haphazardly. However these views of the example architecture serve two purposes: they complete the picture of the architecture from the project, and they give an example of what an architecture emerging from the proposed sub-sequence would look like.

Note that the sub-sequence ordering was selected according to conceptual dependencies and according to the creation and preservation of architecture. These are explored in section 5 of this paper.

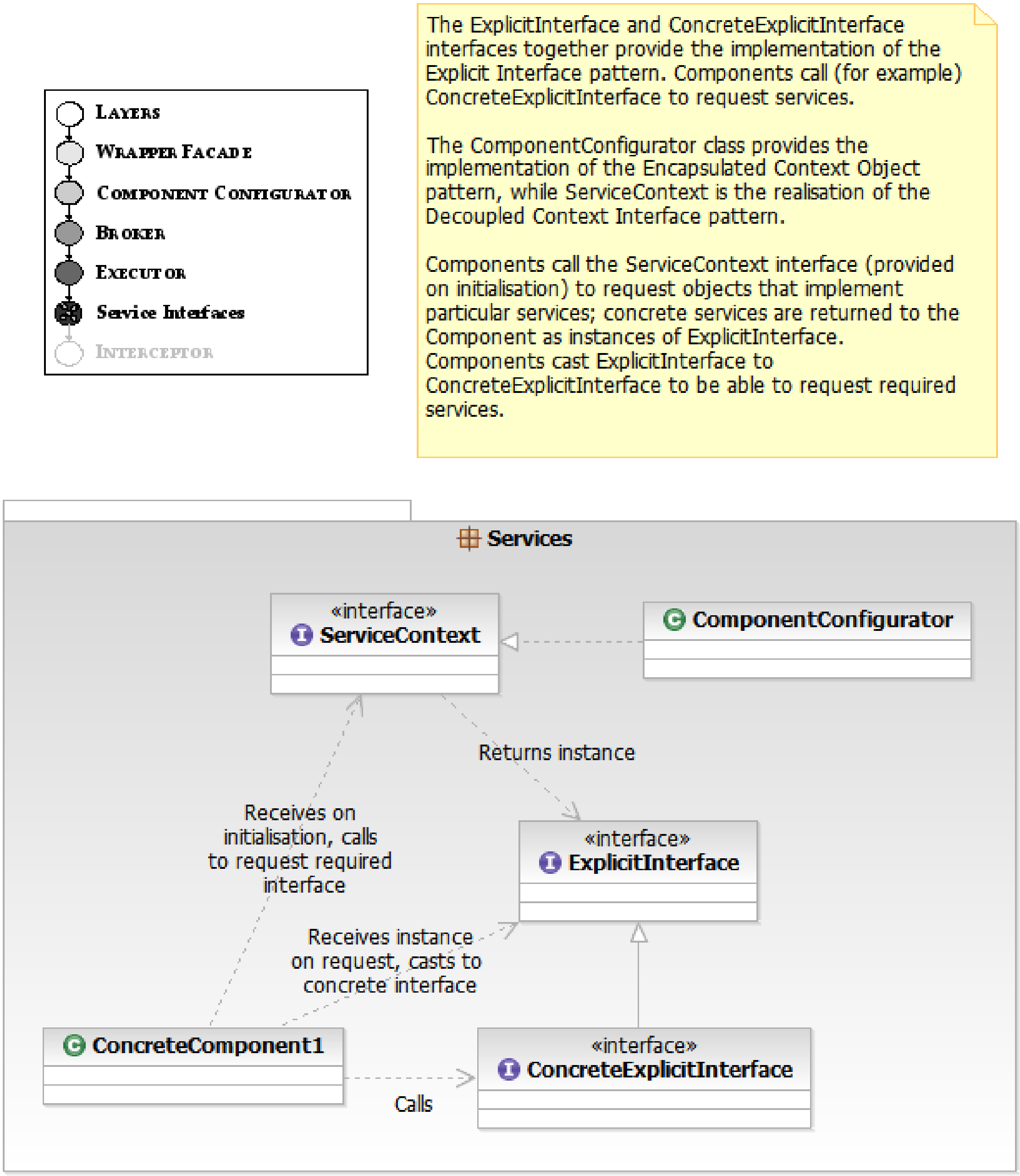


Pattern	*Contribution to Architecture*
Explicit Interface	*Add explicitly defined service interfaces*
Encapsulated Context Object	*Introduce object representing service discovery context, make available to services*
Decoupled Ctxt Interface	*Decouple services from context implementation by introducing service discovery interface*
Proxy	*Add client-side object implementing explicit interface, encapsulates remote communication*
Invoker	*Add server-side object, receives service invocations and invokes explicit service interface*
Lookup	*Provide ability for services to obtain specific proxy for remote service implementing explicit interface*

Figure x: Proposed sub-sequence to add support for service interfaces

8.1 Completed sub-sequence architecture view 1: EXPLICIT INTERFACE, ENCAPSULATED CONTEXT OBJECT, and DECOUPLED CONTEXT INTERFACE

The following diagram shows the software architecture after EXPLICIT INTERFACE, ENCAPSULATED CONTEXT OBJECT, and DECOUPLED CONTEXT INTERFACE have been applied. The architecture shown provides services with a way of discovering and calling service interfaces, while remaining decoupled from the underlying discovery mechanism.



The ExplicitInterface and ConcreteExplicitInterface interfaces together provide the implementation of the Explicit Interface pattern. Components call (for example) ConcreteExplicitInterface to request services.

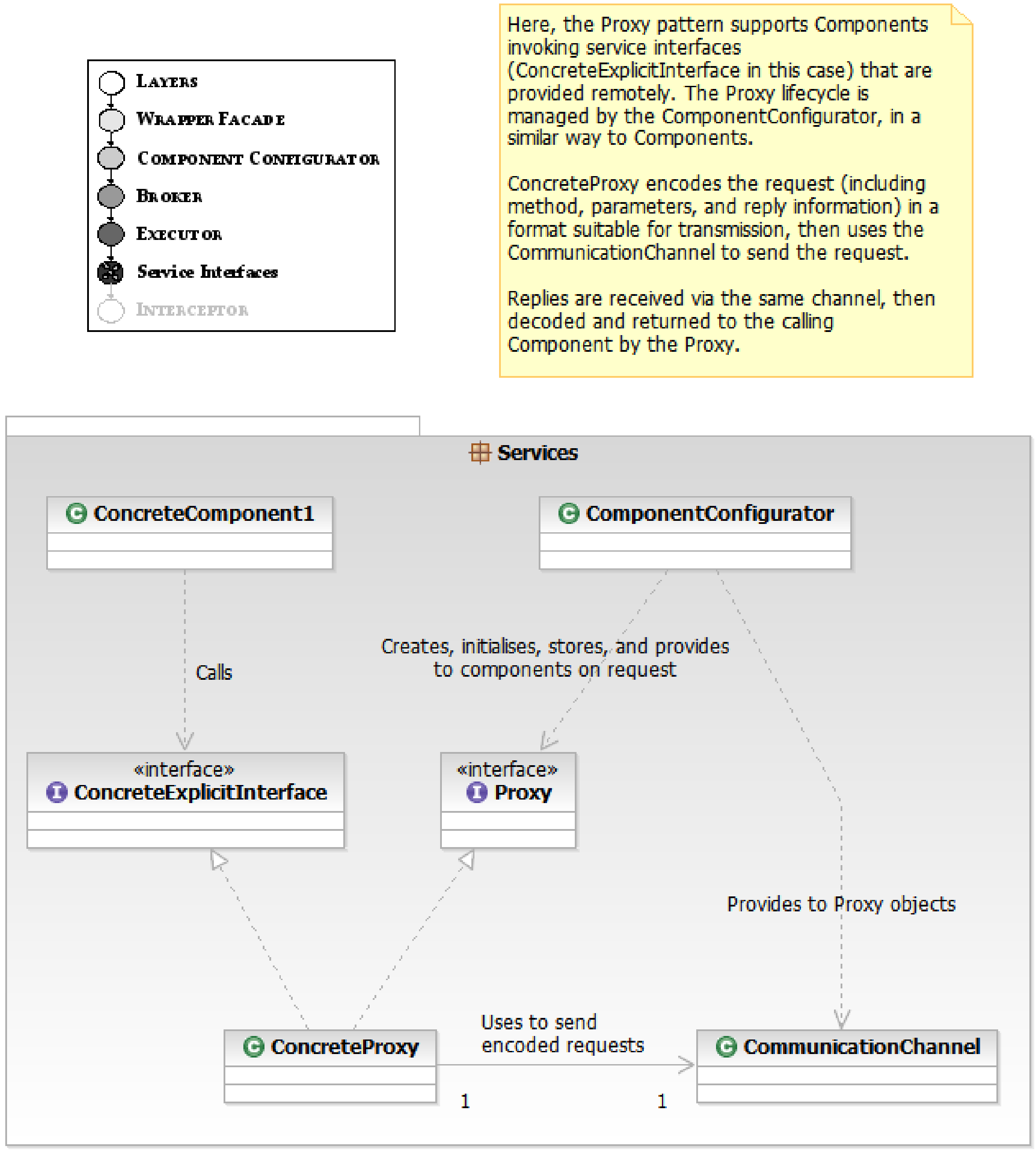
The ComponentConfigurator class provides the implementation of the Encapsulated Context Object pattern, while ServiceContext is the realisation of the Decoupled Context Interface pattern.

Components call the ServiceContext interface (provided on initialisation) to request objects that implement particular services; concrete services are returned to the Component as instances of ExplicitInterface. Components cast ExplicitInterface to ConcreteExplicitInterface to be able to request required services.

Figure xi: Step 6 - EXPLICIT INTERFACE, ENC. CONTEXT OBJECT, DECOUPLED CONTEXT INTERFACE

8.2 Completed sub-sequence architecture view 2: PROXY

The introduction of PROXY allows remote invocation of service interfaces via the location transparent communication provided by the previously applied BROKER implementation.



Here, the Proxy pattern supports Components invoking service interfaces (ConcreteExplicitInterface in this case) that are provided remotely. The Proxy lifecycle is managed by the ComponentConfigurator, in a similar way to Components.

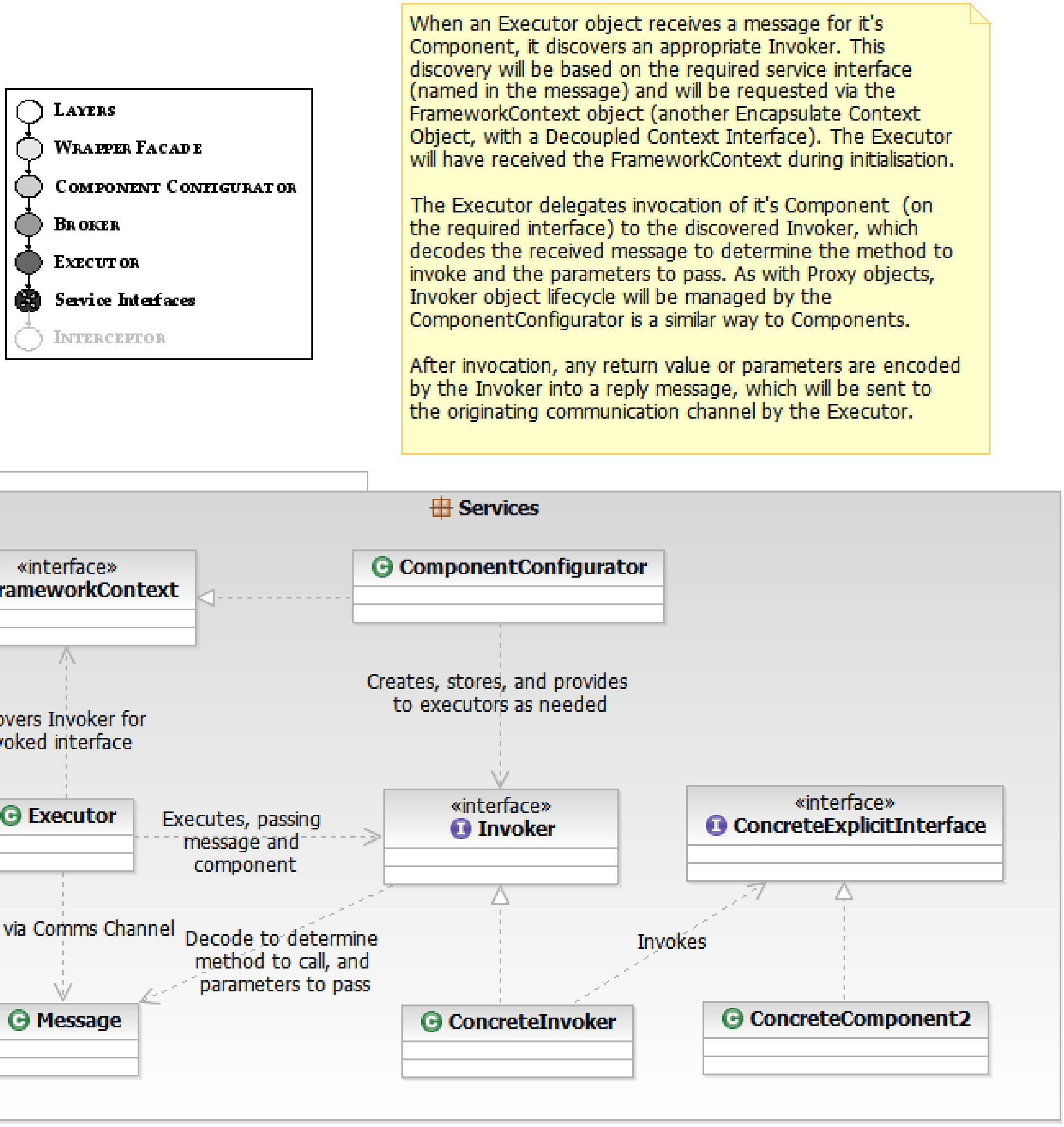
ConcreteProxy encodes the request (including method, parameters, and reply information) in a format suitable for transmission, then uses the CommunicationChannel to send the request.

Replies are received via the same channel, then decoded and returned to the calling Component by the Proxy.

Figure xii: Step 6 - PROXY

8.3 Completed sub-sequence architecture view 3: INVOKER

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. INVOKER implementations, on request from an `Executor`, decode request messages and invoke target `Components`, via desired explicit interfaces. Any return values are encoded and returned to the `Executor`.



When an Executor object receives a message for it's 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 object (another Encapsulate Context Object, with a Decoupled Context Interface). The Executor will have received the FrameworkContext during initialisation.

The Executor delegates invocation of it's Component (on the required interface) to the discovered Invoker, which decodes the received message to determine the method to invoke and the parameters to pass. As with Proxy objects, Invoker object lifecycle will be managed by the ComponentConfigurator is a similar way to Components.

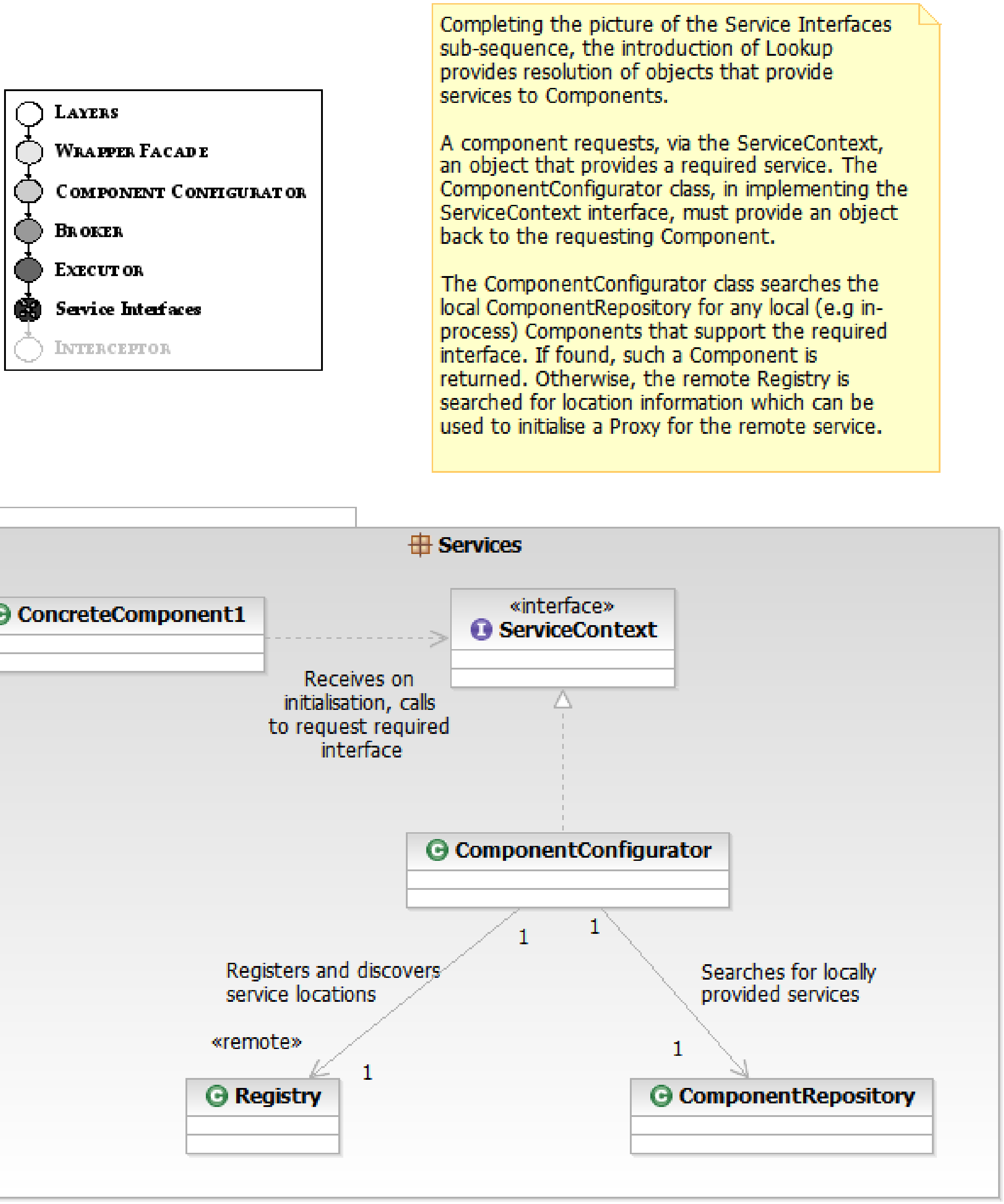
After invocation, any return value or parameters are encoded by the Invoker into a reply message, which will be sent to the originating communication channel by the Executor.

Layers
Wrapper Facade
Component Configurator
Broker
Executor
Service Interfaces
Interceptor
Services
«interface»
FrameworkContext
ComponentConfigurator
Creates, stores, and provides to executors as needed
Discovers Invoker for invoked interface
Executor
Executes, passing message and component
«interface»
Invoker
«interface»
ConcreteExplicitInterface
Receives via Comms Channel
Decode to determine method to call, and parameters to pass
Invokes
Message
ConcreteInvoker
ConcreteComponent2

Figure xiii: Step 6 - INVOKER

8.4 Completed sub-sequence architecture view 4: LOOKUP

The final view of the completed sub-sequence example shows how service-interface discovery is provided in the emerging architecture. In our LOOKUP implementation we introduce a remote `Registry` which 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.



Completing the picture of the Service Interfaces sub-sequence, the introduction of Lookup provides resolution of objects that provide services to Components.

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 requesting Component.

The ComponentConfigurator class searches the local ComponentRepository for any local (e.g in-process) Components that support the required interface. If found, such a Component is returned. Otherwise, the remote Registry is searched for location information which can be used to initialise a Proxy for the remote service.

Figure xiv: Step 6 - LOOKUP

9 References

[Alexander+77] C. Alexander, S. Ishikawa, M. Silverstein, et al *"A Pattern Language"*, Oxford University Press, 1997

[Alexander02] C. Alexander, *"The Nature of Order Book 2: The Process of Creating Life"*, The Center for Environmental Structure (2002)

[Booch06] G. Booch, *"Handbook of Software Architecture - Blog"*, March 2nd, 2006, *"On Design"*

http://booch.com/architecture/blog.jsp?archive=2006-03.html

[Buschmann+04] F. Buschmann and K. Henney, *"Explicit Interface and Object Manager"*, EuroPLoP 2003 Proceedings, Universitätsverlag Konstanz Gmbh (2004)

[Beck99] K. Beck, *"Extreme Programming Explained: Embrace Change "*, Addison-Wesley Professional (1999)

[Crahen02] E. Crahen, *"Executor. Decoupling tasks from execution"*, VikingPLoP 2002

[Evans03] E. Evans, *"Domain-Driven Design: Tackling Complexity in the Heart of Software"*, Addison-Wesley Professional (2003)

[GoF] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *"Design Patterns - Elements of Reusable Object-Oriented Software"*, Addison-Wesley (1995)

[Henney06] K. Henney, *"Context Encapsulation. Three Stories, a Language, and Some Sequences"* (2006),

http://www.two-sdg.demon.co.uk/curbralan/papers.html

[Porter+04] R. Porter, J.O. Coplien, T. Winn, *"Sequences as a Basis for Pattern Language Composition"*, Science of Computer Programming, Elsevier (2004)

[POSA1] 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)

[POSA2] 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)

[POSA3] M. Kircher and P. Jain, *"Pattern-Oriented Software Architecture Volume 3 - Patterns for Resource Management"*, John Wiley and Sons (2004)

[POSA4] F.Buschmann, K. Henney, D.C. Schmidt, *“Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing”,* John Wiley and Sons (2007)

[POSA5] F.Buschmann, K. Henney, D.C. Schmidt, *“Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages”,* John Wiley and Sons (2007)

[Riehle97] D. Riehle, *"Composite Design Patterns"*, OOPSLA (1997) http://www.riehle.org/computer-science/research/1997/oopsla-1997.html

[ScBe01] Schwaber, Ken and Beedle, Mike, *“Agile Software Development with SCRUM”,* Prentice Hall, Upper Saddle River, NJ (2001)

[Siddle06] J. Siddle, *"Using Patterns to Create a Service-Oriented Component Middleware"*, VikingPLoP (2006),

http://jms-home.mysite.orange.co.uk/docs/patternspaper.pdf

[Vlissides98] J. Vlissides, *"Pluggable Factory, Part I"*, C++ Report (1998) http://www.research.ibm.com/designpatterns/pubs/ph-nov-dec98.pdf.

[Voelter+04] M. Voelter, M. Kircher, U. Zdun , *“Remoting Patterns : Foundations of Enterprise, Internet and Realtime Distributed Object Middleware”,* Wiley Software Patterns Series

[Wikipedia] https://www.wikipedia.org

[Zdun06] U. Zdun, *"Systematic pattern selection using pattern language grammars and design space analysis"*, Software - Practice and Experience, John Wiley & Sons (2006)